Lire une table depuis Python

Une base de données n'existe pas pour être consultée à la main : elle sert à des programmes. Tu ouvres la base depuis Python, tu lui envoies une requête SELECT et tu utilises les lignes obtenues.

4TTR 5TTR 6TTR Découverte

Dans DB Browser, c'est toi qui écris la requête et qui lis le résultat. Dans une vraie application — un site web, un logiciel de gestion, une appli de téléphone — personne n'ouvre DB Browser : c'est **un programme** qui envoie les requêtes et utilise les réponses. Dans cet article, ce programme, c'est toi qui l'écris, en Python.

> 💡 Télécharge `mon-ecole-eleves.db` dans la sidebar, renomme-la `mon-ecole.db` et place-la **dans le même dossier** que tes fichiers Python. La base contient une table `eleve` de douze élèves, avec les colonnes `id`, `nom`, `prenom`, `date_naissance`, `commune`, `email` et `absences`.

🎯 Objectifs

À la fin de cet article, tu seras capable de :

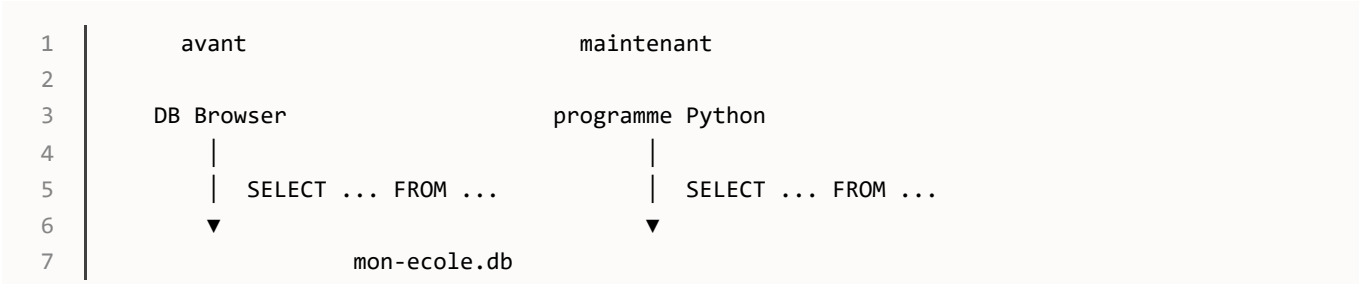
1. Ouvrir et refermer une base SQLite depuis Python.
2. Envoyer une requête `SELECT` et parcourir les lignes obtenues.
3. Récupérer une seule ligne ou toutes les lignes d'un coup.
4. Lire les colonnes d'une ligne par leur position ou par leur nom.
5. Reconnaître une valeur `NULL` dans un programme.
6. Utiliser dans une requête une valeur saisie par l'utilisateur, sans danger.

Python sait déjà parler à SQLite

Python contient un module prévu pour SQLite. Il fait partie de Python lui-même : il n'y a rien à installer.

```
1 | import sqlite3
```

Rappel sur le langage SQL : une requête `SELECT colonnes FROM table` demande à la base d'afficher certaines colonnes d'une table. C'est exactement ce que ton programme va envoyer.



Un premier programme

Crée un fichier `eleves.py` dans le dossier de la base, et écris :

```python
import sqlite3

conn = sqlite3.connect("mon-ecole.db")

curseur = conn.execute("SELECT nom, prenom FROM eleve")

for ligne in curseur:
    print(ligne)

conn.close()
```

Exécute-le. Tu obtiens :

```
('Adam', 'Lucas')
('Bastin', 'Emma')
('Charlier', 'Noah')
('Delvaux', 'Léa')
('Englebert', 'Hugo')
('Fontaine', 'Chloé')
('Gérard', 'Nathan')
('Hubert', 'Manon')
('Istas', 'Louis')
('Jacques', 'Camille')
('Kevers', 'Arthur')
('Lambert', 'Sarah')
```

Regarde le texte entre guillemets : `"SELECT nom, prenom FROM eleve"`. C'est une requête SQL tout à fait ordinaire, rangée dans une chaîne de caractères Python.

> 🧠 **PYTHON NE REMPLACE PAS SQL.** Python transporte la requête jusqu'à SQLite, puis récupère la réponse pour en faire quelque chose.

Ouvrir, travailler, refermer

Tout programme qui utilise une base suit trois étapes.

```
conn = sqlite3.connect("mon-ecole.db")   # 1. ouvrir la base
# ... 2. envoyer des requêtes ...
conn.close()                             # 3. refermer la base
```

📖 NOUVELLE NOTION : LA CONNEXION

Une **connexion** est le lien ouvert entre un programme et une base de données. Toutes les requêtes passent par elle. On l'ouvre au début avec `sqlite3.connect()`, et on la referme à la fin avec `close()`.

En anglais : GB ***connection*** ; *connect* veut dire « connecter » et *close* « fermer ».

⚠️ SQLITE3.CONNECT() CRÉE LE FICHIER S'IL N'EXISTE PAS.

Si tu écris `sqlite3.connect("mon_ecol.db")` par erreur, Python ne dit rien : il crée une base vide. L'erreur n'apparaît qu'à la requête suivante :

```
sqlite3.OperationalError: no such table: eleve
```

Quand une table « disparaît », vérifie d'abord **le nom du fichier** et **le dossier** dans lequel ton programme s'exécute.

Parcourir le résultat

```
curseur = conn.execute("SELECT nom, prenom FROM eleve")
```

`execute` veut dire « exécuter ». Cette ligne envoie la requête à SQLite et range la réponse dans la variable `curseur`.

📖 NOUVELLE NOTION : LE CURSEUR

Un **curseur** est un objet qui donne accès aux lignes du résultat d'une requête, une par une, comme un doigt qui avance dans une liste. On le parcourt avec une boucle `for`.

En anglais : GB ***cursor***.

À chaque tour de la boucle `for`, la variable `ligne` contient la ligne suivante du résultat.

Une ligne est un tuple

Chaque ligne s'affiche entre parenthèses : `('Adam', 'Lucas')`. C'est un **tuple**, une suite de valeurs qu'on ne peut pas modifier. Comme dans une liste, on accède à chaque valeur par sa position, en commençant à `0`.

```
for ligne in curseur:
    print(ligne[1], ligne[0])
```

```
Lucas Adam
Emma Bastin
Noah Charlier
...
```

> ⚠️ Les positions suivent **L'ORDRE DES COLONNES DANS LE `SELECT`**. Avec `SELECT nom, prenom`, `ligne[0]` est le nom. Avec `SELECT prenom, nom`, `ligne[0]` devient le prénom. Change la requête sans changer le programme, et tout s'inverse.

Une ligne ou toutes les lignes

La boucle `for` n'est pas la seule façon de lire un curseur.

`fetchone()` (« aller chercher une ») renvoie **la ligne suivante** du résultat :

```
curseur = conn.execute("SELECT nom, prenom, commune FROM eleve")
print(curseur.fetchone())
print(curseur.fetchone())
```

```
('Adam', 'Lucas', 'Jodoigne')
('Bastin', 'Emma', 'Wavre')
```

Le curseur avance à chaque appel. Quand il n'y a plus de ligne, `fetchone()` renvoie `None`.

`fetchall()` (« aller chercher toutes ») renvoie **une liste** qui contient toutes les lignes :

```
lignes = conn.execute("SELECT prenom FROM eleve").fetchall()
print(lignes)
print(len(lignes))
```

```
[('Lucas',), ('Emma',), ('Noah',), ('Léa',), ('Hugo',), ('Chloé',), ('Nathan',), ('Manon',), (
12
```

Avec une liste, tu peux utiliser tout ce que Python sait faire : `len()` compte les éléments, `lignes[0]` donne la première ligne…

> 💡 **POURQUOI `('LUCAS',)` AVEC UNE VIRGULE ?** La requête ne demande qu'une colonne, mais chaque résultat reste une ligne, donc un tuple. En Python, un tuple d'une seule valeur s'écrit avec une virgule. Pour obtenir la valeur seule, écris `ligne[0]` .

Méthode	Ce qu'elle renvoie	Quand l'utiliser
`for ligne in curseur`	une ligne à chaque tour	traiter les lignes une par une
`fetchone()`	la ligne suivante, ou `None`	quand une seule ligne t'intéresse
`fetchall()`	une liste de toutes les lignes	compter les lignes ou les réutiliser

Des noms plutôt que des numéros

`ligne[1]` ne dit pas ce qu'il contient. Python peut te donner des lignes où chaque valeur se lit **par le nom de sa colonne**. Il suffit d'une ligne juste après la connexion :

```
conn.row_factory = sqlite3.Row
```

Le programme devient :

```
import sqlite3

conn = sqlite3.connect("mon-ecole.db")
conn.row_factory = sqlite3.Row

curseur = conn.execute("SELECT prenom, nom, email FROM eleve")

for ligne in curseur:
    print(ligne["prenom"], ligne["nom"], "-", ligne["email"])

conn.close()
```

`ligne["prenom"]` se lit tout seul, et le programme continue de fonctionner même si tu changes l'ordre des colonnes dans le `SELECT` .

> 💡 À partir de maintenant, ajoute toujours `conn.row_factory = sqlite3.Row` après la connexion.

Quand une valeur est absente

Trois élèves n'ont pas d'adresse e-mail. Dans la base, leur case contient `NULL`, c'est-à-dire aucune valeur. Python le traduit par sa propre valeur « rien » : `None`.

```
for ligne in conn.execute("SELECT prenom, email FROM eleve"):
    print(ligne)
```

```
...
('Noah', None)
...
```

Un programme bien écrit prévoit ce cas :

```
for ligne in curseur:
    if ligne["email"] is None:
        print(ligne["prenom"], ligne["nom"], "- pas d'adresse e-mail")
    else:
        print(ligne["prenom"], ligne["nom"], "-", ligne["email"])
```

```
Lucas Adam - lucas.adam@ecole.be
Emma Bastin - emma.bastin@ecole.be
Noah Charlier - pas d'adresse e-mail
Léa Delvaux - lea.delvaux@ecole.be
Hugo Englebert - hugo.englebert@ecole.be
Chloé Fontaine - pas d'adresse e-mail
Nathan Gérard - nathan.gerard@ecole.be
Manon Hubert - manon.hubert@ecole.be
Louis Istas - louis.istas@ecole.be
Camille Jacques - pas d'adresse e-mail
Arthur Kevers - arthur.kevers@ecole.be
Sarah Lambert - sarah.lambert@ecole.be
```

> 🧠 **`NULL` DANS LA BASE DEVIENT `None` DANS PYTHON.** On le teste avec `is None`.

Une recherche choisie par l'utilisateur

Un programme utile demande souvent quelque chose à l'utilisateur : « quelle commune veux-tu afficher ? ». Il faut alors une requête qui ne garde que certaines lignes.

Pour ça, SQL possède le mot-clé `WHERE` (« où »). `SELECT prenom, nom FROM eleve WHERE commune = 'Wavre'` affiche uniquement les élèves **dont** la commune vaut `Wavre`.

La valeur vient de l'utilisateur. On place un **point d'interrogation** dans la requête, et on donne la valeur à part :

```python
import sqlite3

conn = sqlite3.connect("mon-ecole.db")
conn.row_factory = sqlite3.Row

commune = input("Commune : ")

curseur = conn.execute(
    "SELECT prenom, nom FROM eleve WHERE commune = ?",
    (commune,)
)
lignes = curseur.fetchall()

if len(lignes) == 0:
    print("Aucun élève n'habite à", commune)
else:
    print(len(lignes), "élève(s) trouvé(s) :")
    for ligne in lignes:
        print("-", ligne["prenom"], ligne["nom"])

conn.close()
```

```
Commune : Wavre
2 élève(s) trouvé(s) :
- Emma Bastin
- Nathan Gérard
```

```
Commune : Bruxelles
Aucun élève n'habite à Bruxelles
```

📖 NOUVELLE NOTION : LA REQUÊTE PARAMÉTRÉE

Une **requête paramétrée** contient un `?` à l'endroit où une valeur doit être placée. La valeur est transmise séparément, dans un tuple. SQLite place lui-même la valeur au bon endroit, sans jamais la confondre avec du SQL.

En anglais : GB ***parameterized query***.

⚠️ NE FABRIQUE JAMAIS UNE REQUÊTE EN COLLANT DU TEXTE.

Écrire `"... WHERE commune = '" + commune + "'"` semble plus simple, mais un utilisateur malveillant pourrait taper du SQL à la place d'un nom de commune et faire exécuter ce qu'il veut à ta base. Cette attaque s'appelle une **injection SQL**. Le `?` l'empêche.

💡 LA PETITE VIRGULE DE `(COMMUNE,)`.

En Python, `(commune)` est une simple variable entre parenthèses. Pour créer un tuple d'une seule valeur, il faut la virgule : `(commune,)`.

🧪 Exercices

Tous les programmes ouvrent `mon-ecole.db` et la referment à la fin.

Exercice 1 — La liste d'appel ★☆☆☆☆

Écris un programme qui affiche tous les élèves sous la forme `Lucas ADAM`, avec le nom en majuscules. Utilise `row_factory` et lis les colonnes par leur nom.

> 💡 En Python, `"Adam".upper()` donne `"ADAM"` .

Exercice 2 — Compter et choisir ★☆☆☆☆

Écris un programme qui :

1. affiche le nombre d'élèves de la table, grâce à `fetchall()` et `len()` ;
2. affiche uniquement le premier élève renvoyé par la requête, grâce à `fetchone()`.

Exercice 3 — Comprendre les tuples ★★☆☆☆

Exécute ce programme, **sans** `row_factory` :

```python
import sqlite3

conn = sqlite3.connect("mon-ecole.db")
for ligne in conn.execute("SELECT prenom, absences FROM eleve"):
    print(ligne)
conn.close()
```

1. Qu'affiche-t-il exactement ?
2. Modifie l'affichage pour obtenir `Lucas : 2 demi-jour(s)`, avec `ligne[0]` et `ligne[1]`.
3. Échange les deux colonnes dans le `SELECT` sans toucher au `print`. Que se passe-t-il ?
4. Ajoute `row_factory` et réécris le `print` avec les noms de colonnes. Échange à nouveau les colonnes : que constates-tu ?

Exercice 4 — Les adresses manquantes ★★☆☆☆

Écris un programme qui parcourt tous les élèves et affiche uniquement ceux qui **n'ont pas** d'adresse e-mail, puis leur nombre total.

```
Élèves sans adresse e-mail :
- Noah Charlier
- Chloé Fontaine
- Camille Jacques
Total : 3
```

Fais le tri en Python, avec `is None`.

Exercice 5 — Recherche par commune ★★★☆☆

Reprends le programme de recherche par commune.

1. Tape `Jodoigne`. Combien d'élèves obtiens-tu ?
2. Affiche la table complète dans DB Browser et compte toi-même les élèves de Jodoigne. Trouves-tu le même nombre ?
3. Explique la différence. Qu'est-ce qui, dans les données, piège le programme ?
4. Ce problème vient-il du programme ou des données ? Qui devrait le corriger ?

Exercice 6 — Un petit menu ★★★☆☆

Écris un programme qui affiche en boucle :

```
1. Lister tous les élèves
2. Chercher les élèves d'une commune
3. Afficher les élèves sans adresse e-mail
4. Quitter
```

La connexion est ouverte **une seule fois**, avant la boucle, et refermée quand l'utilisateur quitte. Pour le choix 2, la commune est transmise avec un `?`.

✍️ À retenir

- Le module `sqlite3` fait partie de Python : `import sqlite3`, rien à installer.
- Un programme **ouvre** la connexion avec `sqlite3.connect()`, envoie ses requêtes avec `execute()`, puis la **referme** avec `close()`.
- Le **curseur** donne accès aux lignes : boucle `for`, `fetchone()` pour une ligne, `fetchall()` pour la liste complète.
- Une ligne est un tuple ; avec `conn.row_factory = sqlite3.Row`, on lit ses valeurs par le nom des colonnes.
- `NULL` dans la base devient `None` dans Python.
- Une valeur saisie par l'utilisateur passe par un `?` et un tuple, jamais par une chaîne collée : c'est la protection contre l'**injection SQL**.

Suite

Tu as fait le tour d'une entité : la découvrir, la ranger dans une table, l'identifier et la lire, à la main comme depuis un programme. Révise l'essentiel avec la fiche à étudier, puis passe au niveau Filtrer et analyser ses données

pour poser des questions beaucoup plus précises.

© 2025 Mr. Lambrechts
https://v2.ttrinfo.be
10 | 10