Activité 4 — Modifier les données MariaDB depuis Python

Table des matières

Activité 4 — Modifier les données MariaDB depuis Python....1
1. Objectifs....2
2. Situation de départ....2
3. Rappel : les opérations SQL....3
4. Première insertion depuis Python....3
5. Pourquoi commit() ?....4
6. Vérifier l'insertion....5
7. Attention aux exécutions répétées....5
8. Ajouter un livre depuis Python....6
9. La contrainte de clé étrangère....6
10. Première modification avec UPDATE....7
11. Restituer le livre....7
12. Le problème apparaît : les valeurs sont écrites directement dans le SQL....8
13. Pourquoi éviter les f-strings pour SQL ?....8
14. Première requête paramétrée....9
15. Comprendre le ?....9
16. Pourquoi (id_auteur,) et non id_auteur ?....10
17. Requête paramétrée avec UPDATE....10
18. Requête paramétrée avec INSERT....10
19. Plusieurs paramètres....11
20. Récupérer l'identifiant généré par MariaDB....11
21. Pourquoi récupérer l'identifiant ?....12
22. Exemple complet : créer un auteur puis son livre....12
23. Une question importante : que se passe-t-il en cas d'erreur ?....14
24. Découvrir rollback()....14
II. Exercices....15
1. Exercice 1 — Ajouter un auteur....15
2. Exercice 2 — Ajouter un livre....15
3. Exercice 3 — Emprunter un livre....15
4. Exercice 4 — Rendre un livre....16
5. Exercice 5 — Vérifier avant de modifier....16
6. Exercice 6 — Interdire l'emprunt d'un livre déjà emprunté....16
7. Comparaison avec la classe Livre....17
8. Pourquoi les requêtes paramétrées deviennent obligatoires....17
a. À éviter....17
b. À privilégier....18
9. Bilan des opérations Python → MariaDB....18
10. Synthèse....19
11. État du projet....20
12. Ce qui reste à construire....20

Voici l’activité 4. Elle prolonge directement la précédente : après avoir appris à **lire** les données depuis Python, il s’agit maintenant de les **ajouter et les modifier**. La dernière partie introduit les requêtes paramétrées, qui deviendront indispensables pour la suite.

1. Objectifs

À l’issue de cette activité, il sera possible de :

- exécuter une requête `INSERT` depuis Python ;
- ajouter un auteur dans MariaDB depuis un programme Python ;
- ajouter un livre depuis Python ;
- exécuter une requête `UPDATE` depuis Python ;
- modifier l'état de disponibilité d'un livre ;
- comprendre la validation d'une modification ;
- récupérer l'identifiant généré par MariaDB ;
- découvrir les **requêtes paramétrées** ;
- comprendre pourquoi les valeurs ne doivent pas être intégrées directement dans une chaîne SQL ;
- utiliser `commit()` pour valider les modifications.

2. Situation de départ

Dans l'activité précédente, Python pouvait lire les données de MariaDB. Par exemple :

```
curseur.execute("SELECT * FROM auteur")

lignes = curseur.fetchall()

for ligne in lignes:
    print(ligne)
```

Le fonctionnement était :

```
Python
   │
   │ SELECT
   ▼
MariaDB
   │
   │ résultats
   ▼
Python
```

Python pouvait donc **consulter** la base.

L'objectif est maintenant de permettre à Python de **modifier** la base.

Le fonctionnement devient :

```
Python
   │
   │ INSERT / UPDATE
   ▼
MariaDB
   │
   │ modification enregistrée
   ▼
Base de données
```

3. Rappel : les opérations SQL

Les principales opérations étudiées jusqu'à présent sont :

```
SELECT  → lire
INSERT  → ajouter
UPDATE  → modifier
```

Dans cette activité :

```
SELECT
   ↓
lecture

INSERT
   ↓
ajout

UPDATE
   ↓
modification
```

La suppression avec `DELETE` sera étudiée ultérieurement (elle ne présente pas de difficulté).

4. Première insertion depuis Python : ajouter un auteur à la TABLE auteur

Dans MariaDB, l'ajout d'un auteur s'effectue avec :

```
INSERT INTO auteur (prenom, nom)
VALUES ('Victor', 'Hugo');
```

Cette requête peut être envoyée depuis Python exactement comme une requête `SELECT`.

Créez un fichier : `ajouter_auteur.py` Puis écrivez le programme ci-dessous :

```
import mariadb

connexion = mariadb.connect(
    host="localhost",
    port=3306,
    user="bibliotheque_app",
    password="MotDePasseAChanger",
    database="bibliotheque"
)

curseur = connexion.cursor()

curseur.execute(
    """
    INSERT INTO auteur (prenom, nom)
    VALUES ('Victor', 'Hugo')
    """
)

connexion.commit()

print("Auteur ajouté.")

connexion.close()
```

Exécutez : `python ajouter_auteur.py`

Puis vérifiez directement dans MariaDB :

```
SELECT *
FROM auteur;
```

L'auteur ajouté doit apparaître.

```
MariaDB [bibliotheque]> SELECT * FROM auteur;
+----+---------+-------------------+
| id | prenom  | nom               |
+----+---------+-------------------+
|  1 | George  | Orwell            |
|  2 | Antoine | de Saint-Exupéry  |
|  3 | Albert  | Camus             |
|  4 | Jules   | Verne             |
| 10 | Victor  | Hugo              |
+----+---------+-------------------+
5 rows in set (0,001 sec)

MariaDB [bibliotheque]>
```

5. Pourquoi `commit()` ?

Une différence importante apparaît entre `SELECT` et `INSERT`.

Avec : `SELECT` aucune donnée n'est modifiée.

Avec : `INSERT` la base est modifiée.

Pour valider une modification effectuée depuis Python, on utilise :

```
connexion.commit()
```

On peut retenir pour le moment :

```
SELECT                    INSERT / UPDATE / DELETE
  ↓                                ↓
lecture                        modification
                                   ↓
                                commit()
```

Le `commit()` indique que les modifications effectuées dans la transaction doivent être validées.

6. Attention aux exécutions répétées

Le programme précédent contient directement :

```
VALUES ('Victor', 'Hugo')
```

Si le programme est exécuté plusieurs fois, un nouvel auteur sera ajouté à chaque exécution.

Par exemple :

```
1 - George Orwell
2 - Antoine de Saint-Exupéry
3 - Albert Camus
4 - Jules Verne
5 - Victor Hugo
6 - Victor Hugo
7 - Victor Hugo
```

Ce comportement permet de mettre en évidence un problème : le programme ne contrôle pas encore si l'auteur existe déjà. Cette question sera traitée plus tard. Pour l'instant, le but est de comprendre le mécanisme `INSERT`.

7. Ajouter un livre depuis Python

La table `livre` contient une clé étrangère : `auteur_id`

Il faut donc fournir l'identifiant d'un auteur existant. Par exemple :

```
INSERT INTO livre (titre, auteur_id)
VALUES ('Les Misérables', 5);
```

Cette requête peut être exécutée depuis Python :

```
curseur.execute(
    """
    INSERT INTO livre (titre, auteur_id)
    VALUES ('Les Misérables', 5)
    """
)

connexion.commit()
```

Vérifiez ensuite directement dans la base avec : `SELECT *  FROM livre;`

8. La contrainte de clé étrangère

La table `livre` possède :

```
FOREIGN KEY (auteur_id)
REFERENCES auteur(id)
```

Cela signifie que la valeur de `auteur_id` doit correspondre à un auteur existant.

Par exemple :

```
INSERT INTO livre (titre, auteur_id)
VALUES ('Livre impossible', 9999);
```

Cette requête doit provoquer une erreur si aucun auteur possédant l'identifiant `9999` n'existe.

La base de données protège donc la cohérence des données. Ce contrôle n'est pas réalisé uniquement par Python. Il est également assuré par MariaDB grâce à la **clé étrangère**.

9. Première modification avec UPDATE

Dans le programme Python initial, l'emprunt d'un livre modifiait :

```
livre.disponible = False
```

Avec MariaDB, la modification correspondante est :

```
UPDATE livre
SET disponible = FALSE
WHERE id = 1;
```

Depuis Python :

```
curseur.execute(
    """
    UPDATE livre
    SET disponible = FALSE
    WHERE id = 1
    """
)

connexion.commit()
```

Vérifiez ensuite :

```
SELECT *
FROM livre
WHERE id = 1;
```

```
MariaDB [bibliotheque]> SELECT * FROM livre
WHERE id = 1;
+----+-------+-----------+------------+
| id | titre | auteur_id | disponible |
+----+-------+-----------+------------+
|  1 | 1984  |         1 |          0 |
+----+-------+-----------+------------+
1 row in set (0,001 sec)
```

La colonne `disponible` doit maintenant être à `0` ou `FALSE`.

10. Restituer le livre

La restitution correspond à l'opération inverse :

```
UPDATE livre
SET disponible = TRUE
WHERE id = 1;
```

Depuis Python :

```
curseur.execute(
    """
    UPDATE livre
    SET disponible = TRUE
    WHERE id = 1
    """
)
connexion.commit()
```

Vérifiez :

```
SELECT *
FROM livre
WHERE id = 1;
```

Le livre doit de nouveau être disponible.

11. Le problème apparaît : les valeurs sont écrites directement dans le SQL

Les exemples précédents fonctionnent :

```
curseur.execute(
    """
    UPDATE livre
    SET disponible = FALSE
    WHERE id = 1
    """
)
```

Mais imaginons que l'identifiant du livre soit stocké dans une variable : `id_livre = 3`

Une première idée pourrait être de construire la requête ainsi :

```
requete = f"""
UPDATE livre
SET disponible = FALSE
WHERE id = {id_livre}
"""
curseur.execute(requete)
```

Cela peut fonctionner. Mais cette manière de construire les requêtes SQL doit être évitée.

12. Pourquoi éviter les f-strings pour SQL ?

Une requête SQL construite avec une chaîne de caractères peut mélanger :

`code SQL + données fournies par le programme ou l'utilisateur`

Par exemple :

```
requete = f"""
SELECT *
FROM auteur
WHERE nom = '{nom}'
"""
```

Le contenu de `nom` est directement injecté dans le texte SQL. Cela pose notamment un problème de sécurité : une donnée contenant certains caractères peut modifier la requête SQL.
Ce problème porte le nom d'**injection SQL**. La solution consiste à utiliser des **requêtes paramétrées**.

13. Première requête paramétrée

Au lieu de construire :	on écrit :

`requete = f"""`
`SELECT *`
`FROM auteur`
`WHERE id = {id_auteur}`
`"""`
 	
`curseur.execute(requete)`
	`requete = """`
	`SELECT *`
	`FROM auteur`
	`WHERE id = ?`
	`"""`
	`id_auteur = 3`
	`curseur.execute(requete, (id_auteur,))`

La valeur `3` n'est pas concaténée manuellement dans la chaîne SQL. Elle est transmise séparément au connecteur MariaDB.

14. Comprendre le ?

Dans :

```
curseur.execute(
    """
    SELECT *
    FROM auteur
    WHERE id = ?
    """,
    (id_auteur,)
)
```

le caractère `?` représente une valeur qui sera fournie séparément. Le deuxième argument : `(id_auteur,)` contient cette valeur. Si : `id_auteur = 3` le connecteur associe : `? → 3` sans avoir besoin de construire manuellement une nouvelle chaîne SQL.

15. Pourquoi `(id_auteur,)` et non `id_auteur` ?

Cette syntaxe peut sembler étrange : `(id_auteur,)` La virgule est importante. Elle indique qu'il s'agit d'un tuple contenant une seule valeur.

Par exemple : `valeurs = (3,)` est un tuple à un élément.
Alors que : `valeurs = (3)` est simplement l'entier `3`.

Pour une seule valeur, la virgule est donc nécessaire.

16. Requête paramétrée avec `UPDATE`

Le changement d'état d'un livre peut maintenant être réalisé proprement.

```
id_livre = 1

curseur.execute(
    """
    UPDATE livre
    SET disponible = FALSE
    WHERE id = ?
    """,
    (id_livre,)
)

connexion.commit()
```

La valeur de l'identifiant est séparée du texte SQL. Cette forme devra désormais être privilégiée.

17. Requête paramétrée avec `INSERT`

Il est également possible de paramétrer une insertion.

Au lieu d'écrire :

```
curseur.execute(
    """
    INSERT INTO auteur (prenom, nom)
    VALUES ('Victor', 'Hugo')
    """
)

connexion.commit()
```

Il est préférable d'écrire :

```
prenom = "Victor"
nom = "Hugo"

curseur.execute(
    """
    INSERT INTO auteur (prenom, nom)
    VALUES (?, ?)
    """,
    (prenom, nom)
)
connexion.commit()
```

Dans le cas de droite, les deux valeurs sont donc transmises séparément.

18. Requête avec plusieurs paramètres

Une requête peut contenir plusieurs paramètres. Par exemple :

```
titre = "Notre-Dame de Paris"
id_auteur = 5

curseur.execute(
    """
    INSERT INTO livre (titre, auteur_id)
    VALUES (?, ?)
    """,
    (titre, id_auteur)
)

connexion.commit()
```

La correspondance est :
`premier ?` → `titre`
`deuxième ?` → `id_auteur`
L'ordre est donc important.

19. Récupérer l'identifiant généré par MariaDB : `curseur.lastrowid`

Dans la table `auteur`, la colonne `id` est définie ainsi :

```
id INT PRIMARY KEY AUTO_INCREMENT
```

L'identifiant est donc généré automatiquement par MariaDB.

Lorsqu'un nouvel auteur est ajouté, Python n'a pas besoin de choisir son identifiant.

Par exemple :

```
prenom = "Victor"
nom = "Hugo"

curseur.execute(
    """
    INSERT INTO auteur (prenom, nom)
    VALUES (?, ?)
    """,
    (prenom, nom)
)

connexion.commit()
```

MariaDB peut avoir attribué :

```
id = 5
```

Il peut être utile pour Python de connaître cet identifiant. Le curseur permet de récupérer l'identifiant généré : `id_auteur = curseur.lastrowid ( = last row id )`

Puis : `print(id_auteur)` peut afficher : `5`

La valeur exacte dépend évidemment de l'état actuel de la base.

20. Pourquoi récupérer l'identifiant ?

Cette possibilité devient particulièrement intéressante lorsque plusieurs opérations sont liées.
Par exemple :

```
1. créer un auteur
        ↓
2. récupérer son id
        ↓
3. créer un livre utilisant cet id
```

Python peut donc enchaîner :

```
Auteur
  ↓
id généré
  ↓
Livre
```

C'est exactement le mécanisme nécessaire pour respecter la relation : `Auteur 1 ——— N Livre`

21. Exemple complet : créer un auteur puis son livre

Le programme suivant illustre le mécanisme :

```
import mariadb

connexion = mariadb.connect(
    host="localhost",
    port=3306,
    user="bibliotheque_app",
    password="MotDePasseAChanger",
    database="bibliotheque"
)

curseur = connexion.cursor()

prenom = "Mary"
nom = "Shelley"

curseur.execute(
    """
    INSERT INTO auteur (prenom, nom)
    VALUES (?, ?)
    """,
    (prenom, nom)
)

id_auteur = curseur.lastrowid
```

```
titre = "Frankenstein"

curseur.execute(
    """
    INSERT INTO livre (titre, auteur_id)
    VALUES (?, ?)
    """,
    (titre, id_auteur)
)

connexion.commit()

print(f"Auteur créé avec l'identifiant {id_auteur}.")
print("Livre créé.")

connexion.close()
```

Le programme réalise donc :

```
INSERT auteur
      |
      ▼
id généré par MariaDB
      |
      ▼
INSERT livre
      |
      ▼
commit()
```

Remarquez bien que le commit() est seulement effectué lorsque l'ensemble de la transaction est réalisé. Une transaction peut nécessiter plusieurs requêtes.

22. Une question importante : que se passe-t-il en cas d'erreur ?

Le programme précédent réalise deux opérations :

1. `INSERT auteur`
2. `INSERT livre`

Imaginons que la création du livre échoue. On pourrait alors avoir :

`Auteur créé` mais `Livre non créé`

La base contiendrait une opération partiellement réalisée. Dans une application réelle, ce comportement peut être problématique. Il faudra donc apprendre à utiliser les **transactions** et les **rollback**. Cette notion sera approfondie dans une prochaine activité. Pour l'instant, il suffit de retenir :

`connexion.commit()` `valide les modifications.`

`connexion.rollback()` `permet d'annuler les modifications non encore validées.`

23. Découvrir `rollback()`

Considérons :

```
curseur.execute(
    """
    INSERT INTO auteur (prenom, nom)
    VALUES (?, ?)
    """,
    ("Auteur", "Test")
)
```

Tant que : `connexion.commit()` n'a pas été exécuté, la modification n'est pas définitivement validée dans la transaction courante. A ce stade, un `connexion.rollback()` permet d'annuler la modification.

Exemple :

```
curseur.execute(
    """
    INSERT INTO auteur (prenom, nom)
    VALUES (?, ?)
    """,
    ("Auteur", "Test")
)

connexion.rollback()
```

Après le `rollback()`, l'auteur ne doit pas apparaître dans la table.

II. Sauriez-vous faire :

1. Exercice 1 — Ajouter un auteur

Créer un programme qui :

1. ouvre une connexion à MariaDB ;
2. demande un prénom avec `input()` ;
3. demande un nom avec `input()` ;
4. ajoute l'auteur avec une requête paramétrée ;
5. valide la modification avec `commit()` ;
6. récupère l'identifiant généré ;
7. affiche l'identifiant ;
8. ferme la connexion.

Le programme devra utiliser :

```
VALUES (?, ?)
```

et non une f-string.

2. Exercice 2 — Ajouter un livre

Créer un programme permettant d'ajouter un livre.

Le programme demande :

```
Titre :
Identifiant de l'auteur :
```

Puis exécute une requête paramétrée :

```
INSERT INTO livre (titre, auteur_id)
VALUES (?, ?);
```

Le programme devra ensuite valider la modification.

3. Exercice 3 — Emprunter un livre

Créer un programme qui demande :

```
Identifiant du livre :
```

Puis exécute :

```
UPDATE livre
SET disponible = FALSE
WHERE id = ?;
```

La requête devra être paramétrée.

Le programme devra ensuite effectuer :

```
connexion.commit()
```

4. Exercice 4 — Rendre un livre

Créer le programme inverse.

Il devra demander l'identifiant d'un livre puis exécuter :

```
UPDATE livre
SET disponible = TRUE
WHERE id = ?;
```

La requête devra être paramétrée.

5. Exercice 5 — Vérifier avant de modifier

Le programme doit maintenant vérifier qu'un livre existe avant de tenter de le modifier.

Le principe est :

```
demander l'identifiant
        ↓
chercher le livre
        ↓
le livre existe ?
   ↙️          ↘️
 oui           non
  ↓             ↓
UPDATE       message
```

La recherche pourra utiliser :

```
SELECT id, titre, disponible
FROM livre
WHERE id = ?;
```

Si aucune ligne n'est retournée, afficher :

```
Livre non trouvé.
```

Sinon, effectuer la modification.

6. Exercice 6 — Interdire l'emprunt d'un livre déjà emprunté

Le programme doit maintenant reproduire le comportement de la méthode Python :

```
def emprunter(self) -> bool:
    if not self.disponible:
        return False

    self.disponible = False
    return True
```

Le programme Python devra :

1. rechercher le livre ;
2. vérifier qu'il existe ;
3. vérifier qu'il est disponible ;
4. effectuer le `UPDATE` uniquement s'il est disponible ;
5. valider la modification.

Un livre déjà emprunté ne devra donc pas pouvoir être emprunté une deuxième fois.

7. Comparaison avec la classe `Livre`

Dans le programme POO, l'emprunt était réalisé avec :

```
livre.emprunter()
```

La méthode modifiait :

```
self.disponible = False
```

Avec MariaDB, l'opération correspondante devient :

```
UPDATE livre
SET disponible = FALSE
WHERE id = ?;
```

On peut donc établir la correspondance :

Programme POO	MariaDB
`livre.disponible`	`livre.disponible`
`livre.emprunter()`	`UPDATE livre`
`livre.rendre()`	`UPDATE livre`
liste `livres`	table `livre`
`livre.id`	clé primaire `livre.id`
`livre.auteur`	clé étrangère `livre.auteur_id`

Le comportement métier reste similaire, mais le stockage change.

8. Pourquoi les requêtes paramétrées deviennent obligatoires

À partir de maintenant, toute valeur provenant de Python devra être transmise comme paramètre.

a. À éviter

```
id_livre = 3

requete = f"""
SELECT *
FROM livre
WHERE id = {id_livre}
"""
```

```
curseur.execute(requete)
```

b. À privilégier

```
id_livre = 3

curseur.execute(
    """
    SELECT *
    FROM livre
    WHERE id = ?
    """,
    (id_livre,)
)
```

Cette règle devra devenir un réflexe :

> **Les valeurs ne sont pas concaténées dans les requêtes SQL ; elles sont transmises comme paramètres.**

9. Bilan des opérations Python → MariaDB

Le programme Python sait maintenant réaliser les principales opérations :

```
SELECT
   ↓
lire

INSERT
   ↓
ajouter

UPDATE
   ↓
modifier
```

Avec les mécanismes Python correspondants :

```
curseur.execute(…)  pour exécuter la requête,
curseur.fetchone() pour récupérer les résultats,
```

ou :

```
curseur.fetchall() pour récupérer les résultats,
```

et :

```
connexion.commit() pour valider les modifications.
```

10. Synthèse

Le fonctionnement général d'une modification devient :

```
Programme Python
       │
       │ 1. connexion
       ▼
    MariaDB
       │
       │ 2. execute()
       ▼
  requête SQL
       │
       │ 3. modification
       ▼
    MariaDB
       │
       │ 4. commit()
       ▼
 modification validée
```

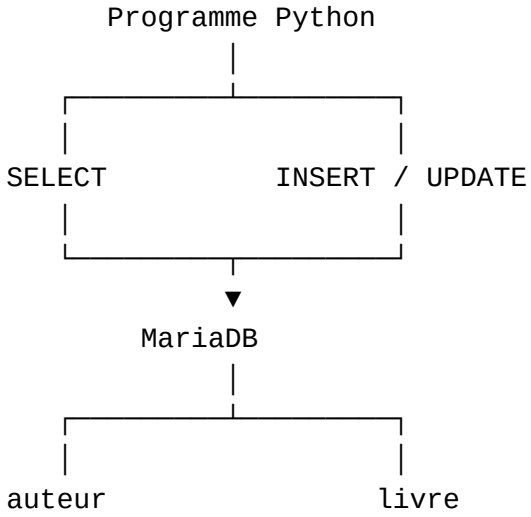
Pour une requête utilisant des données Python :

```
curseur.execute(
    """
    UPDATE livre
    SET disponible = FALSE
    WHERE id = ?
    """,
    (id_livre,)
)
```

La donnée et le code SQL restent séparés.

11. État du projet

Le projet possède maintenant une première couche d'accès aux données :



Python sait désormais :

- se connecter à MariaDB ;
- lire les données ;
- ajouter des données ;
- modifier des données ;
- récupérer un identifiant généré ;
- utiliser des paramètres SQL ;
- valider une modification avec `commit()` ;
- annuler une modification avec `rollback()`.

12. Ce qui reste à construire

Le programme de bibliothèque n'utilise pas encore complètement MariaDB.

Il faudra notamment :

- organiser le code de connexion ;
- éviter de répéter les informations de connexion partout ;
- gérer correctement les erreurs ;
- encapsuler les opérations SQL ;
- retrouver la logique des classes `Auteur`, `Livre` et `Bibliotheque` ;
- gérer les transactions ;
- séparer la logique métier de l'accès aux données.

Une question importante apparaît alors :

Où doit-on placer le code SQL dans une application Python bien organisée ?

Cette question permettra d'introduire progressivement une séparation entre `logique métier` et `accès aux données` . C'est cette séparation qui permettra ensuite de construire une architecture plus propre pour le programme de bibliothèque.