Prise en main de SQLite avec Python

ML-Pro

Décembre 2024

Table des matières

1 Utilisation de SQLite3

Dans cette section, nous allons apprendre à utiliser **SQLite3** avec **Python** pour créer et manipuler une base de données. SQLite3 est un moteur de base de données léger qui stocke les données localement dans un fichier *.db*. Il est particulièrement adapté pour des applications embarquées ou pour des projets nécessitant une base de données simple et rapide à mettre en place.

1.1 Initialisation de SQLite3 avec Python

1.1.1 Installation de SQLite3 (si nécessaire)

La bibliothèque `SQLite3` est intégrée par défaut dans Python, vous n'avez donc pas besoin de l'installer séparément. Elle est prête à l'emploi.

1.1.2 Créer une base de données SQLite3 et initialiser une connexion

Pour commencer, nous allons créer un fichier de base de données SQLite3 et établir une connexion à cette base. Voici comment procéder :

```
import sqlite3

# Créer une connexion à la base de données
conn = sqlite3.connect('bibliotheque.db')

# Créer un curseur pour exécuter les requêtes SQL
cursor = conn.cursor()

print("Connexion à la base de données réussie")
```

Ce code créer une connexion à une base de données nommée `bibliotheque.db`. Dans le cas où le fichier `bibliotheque.db` n'existe pas, SQLite3 le créera automatiquement.

On veillera également à activer la vérification des clés étrangères pour éviter les insertions ou suppressions «incohérentes» :

```
# Active la vérification des clés étrangères
cursor.execute("PRAGMA foreign_keys = ON;")
```

1.1.3 Création des tables avec Python

Particularités SQLite3

Avant toute chose, il faut savoir que SQLite3 simplifie grandement l'utilisation de SQL, mais cela peut dans certains cas être trop restrictif. Par exemple, les types des colonnes sont limités à `NULL`, `INTEGER`, `REAL`, `TEXT` et `BLOB`.
Voici la correspondance entre les types python et les types SQLite3 :

Correspondance des types entre Python et SQLite3	
Types python	Types SQLite3
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

Ainsi, à la place de mettre des types `VARCHAR(100)` ou `VARCHAR(15)` on mettra juste `TEXT`. De plus, on remarque que le type `BOOL` n'existe pas (booléen), en fait, la valeur `TRUE` sera équivalente à `1` et `FALSE` à `0`.

Utilisons maintenant la méthode `execute` du curseur SQLite3 pour exécuter nos requêtes SQL et créer les tables. Vous pouvez utiliser le code suivant pour créer toutes les tables que nous avons définies précédemment. À noter que l'on ajoute `IF NOT EXISTS` après le `CREATE TABLE` pour tester si la table existe déjà. Cela sera utile quand on va se reconnecter à la base, pour éviter de relancer les commandes de création, et avoir des erreurs.

```
# Création des tables
cursor.execute('''
CREATE TABLE IF NOT EXISTS Auteurs (
  AuteurID INTEGER PRIMARY KEY,
  Nom TEXT,
  Prénom TEXT,
  Pays TEXT
);
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Genres (
  GenreID INTEGER PRIMARY KEY,
  NomGenre TEXT
);
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Livres (
  LivreID INTEGER PRIMARY KEY,
  Titre TEXT,
  AuteurID INTEGER,
  GenreID INTEGER,
  DatePublication DATE,
  Disponible BOOLEAN,
  FOREIGN KEY (AuteurID) REFERENCES Auteurs(AuteurID),
  FOREIGN KEY (GenreID) REFERENCES Genres(GenreID)
);
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Emprunteurs (
  EmprunteurID INTEGER PRIMARY KEY,
  Nom TEXT,
  Prénom TEXT,
  Email TEXT,
  Téléphone TEXT
);
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Emprunts (
  EmpruntID INTEGER PRIMARY KEY,
  LivreID INTEGER,
  EmprunteurID INTEGER,
  DateEmprunt DATE,
  DateRetourPrévue DATE,
  DateRetourEffective DATE,
  FOREIGN KEY (LivreID) REFERENCES Livres(LivreID),
  FOREIGN KEY (EmprunteurID) REFERENCES Emprunteurs(EmprunteurID)
);
''')

# Valider les modifications
conn.commit()

print("Tables créées avec succès")
```

Attention à bien utiliser la commande `conn.commit()` pour valider les modifications, si vous oubliez de faire cela, aucun changement ne sera pris en compte.

1.1.4 Insertion de données dans les tables

Ensuite, vous pouvez insérer des données dans les tables à l'aide de la commande `INSERT INTO`. Voici comment insérer les données initiales dans vos tables :

```
cursor.execute('''
INSERT INTO Auteurs (AuteurID, Nom, Prénom, Pays) VALUES
(1, 'Hugo', 'Victor', 'France'),
(2, 'Orwell', 'George', 'Royaume-Uni'),
(3, 'Asimov', 'Isaac', 'Russie'),
(4, 'Tolkien', 'J.R.R.', 'Royaume-Uni'),
(5, 'Austen', 'Jane', 'Royaume-Uni'),
(6, 'Dumas', 'Alexandre', 'France'),
(7, 'Bradbury', 'Ray', 'États-Unis'),
(8, 'Camus', 'Albert', 'France'),
(9, 'Verne', 'Jules', 'France'),
(10, 'Hemingway', 'Ernest', 'États-Unis');
''')

cursor.execute('''
INSERT INTO Genres (GenreID, NomGenre) VALUES
(1, 'Roman'),
(2, 'Science-fiction'),
(3, 'Fantasy'),
(4, 'Classique'),
(5, 'Philosophie'),
(6, 'Aventure'),
(7, 'Horreur'),
(8, 'Biographie');
''')

cursor.execute('''
INSERT INTO Livres (LivreID, Titre, AuteurID, GenreID, DatePublication, Disponible) VALUES
(1, 'Les Misérables', 1, 1, '1862-01-01', 1),
(2, '1984', 2, 2, '1949-06-08', 0),
(3, 'Fondation', 3, 2, '1951-01-01', 1),
(4, 'Le Seigneur des Anneaux', 4, 3, '1954-07-29', 1),
(5, 'Orgueil et Préjugés', 5, 4, '1813-01-28', 1),
(6, 'Le Comte de Monte-Cristo', 6, 6, '1844-08-28', 1),
(7, 'Fahrenheit 451', 7, 2, '1953-10-19', 1),
(8, "L'Étranger", 8, 5, '1942-01-01', 0),
(9, 'Vingt mille lieues sous les mers', 9, 6, '1870-06-20', 1),
(10, 'Le Vieil Homme et la Mer', 10, 4, '1952-09-01', 0),
(11, 'Les Trois Mousquetaires', 6, 6, '1844-03-14', 1),
(12, 'Le Château', NULL, 4, '1926-01-01', 1);
''')

cursor.execute('''
INSERT INTO Emprunteurs (EmprunteurID, Nom, Prénom, Email, Téléphone) VALUES
(1, 'Dupont', 'Jean', 'jean.dupont@mail.com', '0601020304'),
(2, 'Martin', 'Lucie', 'lucie.martin@mail.com', '0602030405'),
(3, 'Bernard', 'Paul', 'paul.bernard@mail.com', '0603040506'),
(4, 'Durand', 'Sophie', 'sophie.durand@mail.com', '0604050607'),
(5, 'Lefevre', 'Antoine', NULL, '0605060708'),
(6, 'Roux', 'Marie', 'marie.roux@mail.com', '0606070809'),
(7, 'Moreau', 'Julie', 'julie.moreau@mail.com', '0607080910'),
(8, 'Petit', 'Nicolas', 'nicolas.petit@mail.com', '0608091011'),
(9, 'Girard', 'Laure', 'laure.girard@mail.com', '0609101112'),
(10, 'Andre', 'Thomas', 'thomas.andre@mail.com', NULL),
(11, 'Lam', 'Marc', 'marc.lam@mail.com', '0609101113');
''')

cursor.execute('''
INSERT INTO Emprunts (EmpruntID, LivreID, EmprunteurID, DateEmprunt, DateRetourPrévue, DateRetourEffective) VALUES
(1, 1, 1, '2024-10-10', '2024-10-17', NULL),
(2, 2, 2, '2024-10-11', '2024-10-18', '2024-10-13'),
(3, 3, 3, '2024-10-12', '2024-10-19', NULL),
(4, 4, 4, '2024-10-13', '2024-10-20', '2024-10-17'),
(5, 5, 5, '2024-10-14', '2024-10-21', NULL),
(6, 6, 6, '2024-10-15', '2024-10-22', '2024-10-20'),
(7, 7, 7, '2024-10-16', '2024-10-23', NULL),
(8, 8, 8, '2024-10-17', '2024-10-24', '2024-10-28'),
(9, 9, 9, '2024-10-18', '2024-10-25', NULL),
(10, 5, 10, '2024-10-19', '2024-10-26', NULL),
(11, 11, 1, '2024-10-20', '2024-10-27', '2024-10-25'),
(12, 7, 2, '2024-10-21', '2024-10-28', NULL),
(13, 8, 3, '2024-10-22', '2024-10-29', NULL),
(15, 1, 5, '2024-10-24', '2024-10-31', NULL),
(16, 4, 6, '2024-10-25', '2024-11-01', NULL),
(17, 9, 7, '2024-10-26', '2024-11-02', NULL);
''')

# Valider les modifications
conn.commit()

print("Données insérées avec succès")
```

1.2 Exécuter des requêtes SQL depuis Python

Une fois que les données sont insérées, vous pouvez exécuter des requêtes `SELECT` pour récupérer les informations et les afficher. Par exemple si on veut récupérer toutes les lignes de la table Auteurs :

```
# Récupérer tous les auteurs
cursor.execute('SELECT * FROM Auteurs')
auteurs = cursor.fetchall()

print("Liste des auteurs :")
for auteur in auteurs:
    print(auteur)
```

Attention à l'utilisation du `SELECT *` qui va récupérer toutes les données d'une table. Dans le cas de bases de données très conséquentes, son utilisation n'est pas conseillée, pour éviter de faire planter le SGBD qui tentera de retourner trop de données d'un coup.

1.3 Fermeture de la connexion

N'oubliez pas de fermer la connexion à la base de données à la fin de votre programme pour libérer les ressources :

```
# Fermer la connexion
conn.close()

print("Connexion à la base de données fermée")
```

1.4 Conseils supplémentaires

— **Gestion des exceptions :** Il est recommandé d'utiliser des blocs `try...except` en Python pour gérer les exceptions potentielles lors de l'exécution de vos requêtes SQL. Cela vous aidera à identifier les erreurs et à les corriger plus facilement. Exemple d'utilisation :

```
try:
    cursor.execute('SELECT * FROM Livres')
    livres = cursor.fetchall()
except sqlite3.Error as e:
    print(f"Une erreur s'est produite : {e}")
```

Ce code capture les erreurs liées à l'exécution de la requête SQL et les affiche, facilitant ainsi le débogage.

— **Utilisation des paramètres :** Pour éviter les injections SQL et améliorer la sécurité, utilisez des paramètres dans vos requêtes. Par exemple :

```
# Exécution d'une requête avec des paramètres
cursor.execute('SELECT * FROM Livres WHERE AuteurID = ?', (1,))
```

Au lieu de :

```
# Requête SQL potentiellement vulnérable à l'injection SQL
cursor.execute('SELECT * FROM Livres WHERE AuteurID = 1')
```

— **Formatage des résultats :** Pour une meilleure lisibilité, vous pouvez formater les résultats sous forme de tableau en utilisant des bibliothèques comme `tabulate` ou `pandas`.

Exemple avec `pandas` :

```
import pandas as pd

# Exécuter une requête pour récupérer les livres
cursor.execute('SELECT Titre, AuteurID FROM Livres')
livres = cursor.fetchall()

# Transformer les résultats en DataFrame pandas pour un formatage lisible
df = pd.DataFrame(livres, columns=['Titre', 'AuteurID'])
print(df)
```

Ce code utilise `pandas` pour afficher les résultats de la requête sous forme de tableau, facilitant ainsi l'analyse des données.

1.5 Exécuter les requêtes des exercices précédents

À partir de là, vous pouvez exécuter les requêtes des exercices précédents pour interagir avec la base de données. Par exemple, pour récupérer les livres disponibles :

```
# Exemple de requête pour récupérer les livres disponibles
cursor.execute('SELECT Titre FROM Livres WHERE Disponible = 1')
livres_disponibles = cursor.fetchall()

print("Livres disponibles :")
for livre in livres_disponibles:
    print(livre[0])
```

Vous pouvez adapter les autres requêtes SQL des exercices précédents de la même manière, en utilisant `cursor.execute()` pour exécuter la requête, puis en récupérant les résultats avec `cursor.fetchall()` ou `cursor.fetchone()`.

Récupérer les résultats avec fetchall() et fetchone()

Lorsque vous exécutez une requête SQL qui renvoie des résultats (par exemple, une requête `SELECT`), le curseur (objet `cursor`) contient les résultats de la requête. Ces résultats doivent ensuite être récupérés pour pouvoir être manipulés ou affichés. Pour cela, on utilise les méthodes `fetchall()` et `fetchone()` :

— `fetchall()` : Cette méthode permet de récupérer **tous les résultats** de la requête sous la forme d'une liste de tuples. Chaque tuple représente une ligne du résultat de la requête. Par exemple, si vous avez sélectionné des colonnes, chaque tuple contiendra une valeur pour chaque colonne. Exemple d'utilisation :

```
cursor.execute('SELECT Titre, AuteurID FROM Livres')
tous_les_livres = cursor.fetchall()

for livre in tous_les_livres:
    print(f'Titre : {livre[0]}, Auteur id : {livre[1]}')
```

Dans cet exemple, chaque tuple contient le titre et l'auteur d'un livre, et `fetchall()` renvoie une liste avec toutes les lignes.

— `fetchone()` : Cette méthode permet de récupérer **une seule ligne** du résultat de la requête, sous la forme d'un tuple. Elle est utile lorsque vous savez que la requête renverra au maximum une ligne (par exemple, lors de la recherche par une clé primaire ou un identifiant unique) ou si vous souhaitez traiter les résultats ligne par ligne.

Exemple d'utilisation :

```
cursor.execute('SELECT Titre, AuteurID FROM Livres WHERE LivreID = 1')
livre = cursor.fetchone()

if livre:
    print(f'Titre : {livre[0]}, Auteur ID : {livre[1]}')
else:
    print("Aucun livre trouvé")
```

Ici, `fetchone()` renvoie la première ligne du résultat, ou `None` si aucun résultat n'est disponible.

Ces méthodes sont complémentaires. Utilisez `fetchall()` lorsque vous avez besoin de manipuler plusieurs résultats, et `fetchone()` lorsque vous ne voulez traiter qu'une seule ligne à la fois.

L'idée principale est que fetchall() permet de récupérer plusieurs résultats à la fois, ce qui est pratique lorsque vous interrogez une base de données et que vous attendez de nombreuses réponses. fetchone(), quant à lui, vous permet de travailler avec les résultats un par un, souvent utile lorsque vous savez que la requête ne renverra qu'une seule ligne ou lorsque vous parcourez les résultats progressivement.

2 Projet : Application de Flashcards

Après avoir maîtrisé les concepts de SQL et exploré l'interaction entre Python et SQLite, vous êtes maintenant prêts à mettre en pratique vos compétences en construisant votre propre application de Flashcards. Ce projet vous permettra de créer une base de données complète et d'écrire les fonctions nécessaires pour gérer vos flashcards, thèmes et statistiques. Nous verrons par la suite, après le chapitre sur Streamlit, comment créer l'interface pour gérer cette application, et la rendre dynamique.

2.1 Introduction au projet

Le but de ce projet est de construire une application de flashcards. Ce type d'application permet à l'utilisateur de créer des cartes mentales, avec une information ou question par carte, et d'essayer de répondre à la question ou d'expliquer le concept. La subtilité est que, l'apparition des cartes dépend de si vous arrivez à avoir la bonne réponse. Plus serez à l'aise avec un concept, moins la carte apparaitra, et au contraire, plus vous avez du mal avec un concept, et plus la carte apparaitra fréquemment.

L'objectif de cette partie ici est de développer la base de données et les fonctions de base pour intéragir avec. Puis dans le chapitre de Streamlit, nous développerons la partie interface, pour au final avoir une jolie application de flashcards que vous pourrez utiliser personnellement.

2.2 Schéma de la Base de Données

Votre base de données pour l'application de Flashcards comprendra trois tables principales :

- **cards** : Contient les flashcards.
- **themes** : Contient les thèmes des flashcards.
- **stats** : Contient les statistiques des réponses des utilisateurs.

Détail des Tables

- **cards**
  - `id` : INTEGER, PRIMARY KEY
  - `question` : TEXT
  - `reponse` : TEXT
  - `probabilite` : REAL
  - `id_theme` : INTEGER, FOREIGN KEY REFERENCES themes(id) ON DELETE RESTRICT
- **themes**
  - `id` : INTEGER, PRIMARY KEY
  - `theme` : TEXT
- **stats**
  - `id` : INTEGER, PRIMARY KEY
  - `bonnes_reponses` : INTEGER
  - `mauvaises_reponses` : INTEGER
  - `date` : DATE

Le champ `probabilite` est un réel entre 0.1 et 1.

Gérer la suppression d'un thème

Pour éviter de supprimer une flashcard qui utilise un thème qu'on essaiera de supprimer, on va ajouter `ON DELETE RESTRICT` après la définition de la clé étrangère `id_theme` dans la table `cards`. Par exemple, ici on aurait `FOREIGN KEY(id) REFERENCES themes(id) ON DELETE RESTRICT` Cette contrainte empêche la suppression d'un thème s'il est référencé dans la table cards.

2.3 Fonctions à Implémenter

Votre tâche est de créer les fonctions Python nécessaires pour interagir avec la base de données SQLite. Vous devrez implémenter les fonctions suivantes, en suivant les spécifications et en utilisant les indices fournis.

2.3.1 Initialisation de la Base de Données

Fonction : `init_db()`

Description : Initialise la base de données en créant les tables `cards`, `themes` et `stats` si elles n'existent pas déjà. Elle insère également des thèmes prédéfinis dans la table `themes`.

- Utilisez la fonction `sqlite3.connect()` pour établir une connexion.
- Utilisez `cursor.execute()` pour exécuter les requêtes SQL de création de tables.
- Assurez-vous d'utiliser `IF NOT EXISTS` pour éviter les erreurs si les tables existent déjà.
- Insérez les thèmes en une seule requête `INSERT INTO`.
- Validez les modifications avec `conn.commit()` et fermez la connexion.

```
# Exemple de structure de la fonction init_db()

def init_db():
    conn = sqlite3.connect("flashcards.db")
    c = conn.cursor()
    c.execute("PRAGMA foreign_keys = ON;")

    # Créer les tables et insérer les thèmes

    conn.commit()
    conn.close()
```

2.3.2 Fonctions CRUD pour les Flashcards

Fonctions :

- `create_card(question, reponse, probabilite, id_theme)` pour créer une carte
- `get_card(id)` pour récupérer une carte
- `update_card(id, question, reponse, probabilite, id_theme)` pour mettre à jour une carte avec des nouvelles données
- `delete_card(id)` pour supprimer une carte
- `get_all_cards()` pour récupérer toutes les cartes
- `get_number_of_cards()` pour avoir le nombre total de cartes
- `get_cards_by_theme(id_theme)` pour récupérer les cartes appartenant à un thème particulier

Description : Ces fonctions permettent de créer, lire, mettre à jour et supprimer des flashcards dans la table `cards`.

- Utilisez `INSERT INTO` pour créer une nouvelle carte.
- Utilisez `SELECT` pour lire des cartes spécifiques ou toutes les cartes.
- Utilisez `UPDATE` pour modifier les détails d'une carte.
- Utilisez `DELETE FROM` pour supprimer une carte.
- Utilisez `COUNT(*)` pour obtenir le nombre total de cartes.
- Utilisez des clauses `WHERE` pour filtrer les thèmes par `id`.
- Utilisez des paramètres de requête `?` pour éviter les injections SQL.

```
# Exemple de structure de la fonction create_card()

def create_card(question, reponse, probabilite, id_theme):
    conn = sqlite3.connect("flashcards.db")
    c = conn.cursor()
    c.execute("PRAGMA foreign_keys = ON;")

    c.execute("""
        INSERT INTO cards (question, reponse, probabilite, id_theme)
        VALUES (?, ?, ?, ?)
    """, (question, reponse, probabilite, id_theme))

    conn.commit()
    conn.close()
```

2.3.3 Fonctions CRUD pour les Thèmes

Fonctions :

- `create_theme(theme)` pour créer un thème
- `get_theme(id_theme)` pour récupérer un thème
- `update_theme(id_theme, theme)` pour mettre à jour le thème
- `delete_theme(id_theme)` pour supprimer un thème
- `get_all_themes()` pour récupérer tous les thèmes

Description : Ces fonctions permettent de gérer les thèmes dans la table `themes`.

- Utilisez `INSERT INTO` pour créer un nouveau thème.
- Utilisez `SELECT` pour lire un thème spécifique ou tous les thèmes.
- Utilisez `UPDATE` pour modifier le nom d'un thème.
- Utilisez `DELETE FROM` pour supprimer un thème. (on a déjà gérer le problème de clé étrangère avec les cards)

```python
# Exemple de structure de la fonction create_theme()

import sqlite3

def create_theme(theme):
    conn = sqlite3.connect("flashcards.db")
    c = conn.cursor()
    c.execute("PRAGMA foreign_keys = ON;")

    c.execute("""
        INSERT INTO themes (theme)
        VALUES (?)
    """, (theme,))

    conn.commit()
    conn.close()
```

2.3.4 Fonctions pour les Statistiques

Fonctions :

- `update_stats(is_correct)` pour mettre à jour la base `stats` suivant les indications ci-dessous
- `update_card_probability(card_id, is_correct)` pour mettre à jour la probabilité d'apparition d'une carte
- `get_stats()` pour récupérer les statistiques au travers du temps

Description : Ces fonctions gèrent les statistiques des réponses des utilisateurs, telles que le nombre de bonnes et mauvaises réponses, ainsi que la probabilité d'affichage des cartes.

- Utilisez `SELECT` pour récupérer les statistiques existantes.
- Utilisez `UPDATE` pour modifier les statistiques en fonction des réponses.
- Utilisez `INSERT INTO` pour ajouter de nouvelles entrées de statistiques.
- Calculez les probabilités en fonction des réponses correctes ou incorrectes.
- Utilisez la bibliothèque `datetime` pour gérer les dates.

Indication pour l'implémentation de la fonction update_stats() :

— **Vérification de l'existence d'une entrée pour la date du jour :** Avant de mettre à jour les statistiques, la fonction vérifie si une entrée pour la date du jour existe déjà dans la table `stats`. Utilisez une requête `SELECT` pour récupérer cette entrée :

```
today = datetime.now().strftime('%Y-%m-%d')
c.execute('SELECT * FROM stats WHERE date = ?', (today,))
stats = c.fetchone()
```

Cette requête vérifie si des statistiques existent déjà pour la date courante. Si une entrée existe (donc la variable `stats` est définie), elle sera mise à jour, sinon une nouvelle entrée sera créée.

— **Mise à jour des statistiques existantes :** Si une entrée existe pour la date courante, le nombre de bonnes ou mauvaises réponses est mis à jour en fonction de la variable `is_correct`. Une requête `UPDATE` est alors utilisée :

```
if is_correct:
    bonnes_reponses += 1
else:
    mauvaises_reponses += 1

c.execute(
    """
    UPDATE stats
    SET bonnes_reponses = ?, mauvaises_reponses = ?
    WHERE id = ?
    """,
    (bonnes_reponses, mauvaises_reponses, id),
)
```

Le code ci-dessus met à jour les colonnes `bonnes_reponses` et `mauvaises_reponses` en fonction de la réponse de l'utilisateur.

— **Création d'une nouvelle entrée :** Si aucune entrée n'existe pour la date courante, une nouvelle ligne est insérée dans la table `stats` avec les bonnes ou mauvaises réponses. Cela se fait à l'aide d'une requête `INSERT INTO` :

```
bonnes_reponses = 1 if is_correct else 0
mauvaises_reponses = 0 if is_correct else 1

c.execute(
    """
    INSERT INTO stats (bonnes_reponses, mauvaises_reponses, date)
    VALUES (?, ?, ?)
    """,
    (bonnes_reponses, mauvaises_reponses, today),
)
```

Cette partie du code crée une nouvelle entrée avec la date courante et initialise les statistiques en fonction de la première réponse de la journée.

— **Gestion des dates :** La date du jour est récupérée via la bibliothèque `datetime` et formatée en chaîne de caractères compatible avec SQLite (`YYYY-MM-DD`). Cela garantit que les statistiques sont enregistrées avec précision chaque jour.

```
today = datetime.now().strftime("%Y-%m-%d")
```

— **Gestion des erreurs :** La fonction suppose que la table `stats` existe et qu'elle est correctement structurée. Cependant, il peut être utile d'ajouter des blocs `try...except` autour des opérations SQL pour capturer et gérer les erreurs potentielles, par exemple, si la base de données est corrompue ou si une contrainte d'intégrité est violée.

Indication pour l'implémentation de la fonction `update_card_probability()` :

- **Récupération de la probabilité actuelle :** Utilisez une requête `SELECT` pour obtenir la valeur actuelle de `probabilite` de la carte identifiée par `card_id`.
- **Calcul de la nouvelle probabilité :**
  - Si `is_correct` est `True`, multipliez la probabilité actuelle par 0.9.
  - Si `is_correct` est `False`, multipliez la probabilité actuelle par 1.1.
- **Limitation de la probabilité :** Assurez-vous que la nouvelle probabilité reste entre 0.1 et 1.0 en utilisant les fonctions `max(0.1, min(nouvelle_probabilité, 1.0))`.
- **Mise à jour de la base de données :** Utilisez une requête `UPDATE` pour enregistrer la nouvelle probabilité dans la table `cards` pour la carte correspondante.
- **Utilisation des paramètres :** Utilisez des paramètres `?` dans vos requêtes SQL pour éviter les injections SQL et améliorer la sécurité.
- **Gestion des erreurs :** Ajoutez des vérifications pour vous assurer que la carte existe avant de tenter de mettre à jour sa probabilité.