Accès aux bases de données en Python

Pierre Jaumier

Accès aux bases de données en Python

Plan du diaporama

1. Introduction aux bases de données en Python
2. Utilisation de `sqlite3` (sans ORM)
3. Introduction aux ORM
4. Utilisation de SQLAlchemy
5. Et avec PostgreSQL ?
6. Conclusion & Comparaison

Partie 1 : Accès basique sans ORM (`sqlite3`)

Qu'est-ce que SQLite ?

- Système de gestion de base de données relationnelle **sans serveur**
- Base de données stockée dans un fichier
- Très utile pour les petits projets ou tests
- Module standard de Python : `sqlite3`

Connexion à une base SQLite

```python
import sqlite3

connexion = sqlite3.connect("ma_base.db")  # Créer ou ouvrir une base
curseur = connexion.cursor()
```

On peut maintenant exécuter des requêtes SQL via `curseur.execute(...)`

Exemple : Création d'une table

```python
curseur.execute('''
    CREATE TABLE IF NOT EXISTS utilisateurs (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        nom TEXT,
        email TEXT
    )
''')
connexion.commit()
```

Insertion de données

```python
curseur.execute('''
    INSERT INTO utilisateurs (nom, email)
    VALUES (?, ?)
''', ("Alice", "alice@example.com"))

connexion.commit()
```

Sélectionner des données

```python
curseur.execute("SELECT * FROM utilisateurs")
resultats = curseur.fetchall()

for row in resultats:
    print(row)
```

Avantages / Inconvénients de cette approche

Simplicité
Pas de dépendances externes
Syntaxe SQL manuelle
Moins sécurisée si mal utilisé (injections SQL)
Difficile à maintenir à grande échelle

Partie 2 : Introduction aux ORM

Qu'est-ce qu'un ORM ?

ORM = Object Relational Mapping
Permet de manipuler une base de données comme des objets Python

Exemples : - SQLAlchemy (le plus populaire) - Django ORM - Peewee

Pourquoi utiliser un ORM ?

Abstraction des requêtes SQL
Plus sécurisé (protection contre les injections SQL)
Plus facile à maintenir et à modéliser
Compatible avec plusieurs SGBD (PostgreSQL, MySQL, SQLite…)

Partie 3 : Utilisation de SQLAlchemy

Installation de SQLAlchemy

```
pip install sqlalchemy
conda install conda-forge::sqlalchemy
```

Configuration de la base

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

Base = declarative_base()

engine = create_engine("sqlite:///ma_base.db", echo=True)
Session = sessionmaker(bind=engine)
session = Session()
```

Définir un modèle (classe)

```
class Utilisateur(Base):
    __tablename__ = 'utilisateur'

    id = Column(Integer, primary_key=True)
    nom = Column(String)
    email = Column(String)

# Création des tables
Base.metadata.create_all(engine)
```

Ajouter des données

```
nouveau_user = Utilisateur(nom="Bob", email="bob@example.com")
session.add(nouveau_user)
session.commit()
```

Lire des données

```
utilisateurs = session.query(Utilisateur).all()
for u in utilisateurs:
    print(u.id, u.nom, u.email)
```

Filtrer et modifier

```
user = session.query(Utilisateur).filter_by(nom="Bob").first()
user.email = "robert@example.com"
session.commit()
```

Supprimer

```
session.delete(user)
session.commit()
```

Avantages de SQLAlchemy

Interface orientée objet
Gestion automatique des connexions
Support de multiples bases de données
Migrations possibles (avec Alembic)

Et avec PostgreSQL ?

PostgreSQL est un SGBD relationnel puissant et open-source.
Il est très utilisé dans les applications professionnelles.

Transactions ACID
Support des types JSON, géolocalisation, etc.
Bonne intégration avec Python

Connexion à PostgreSQL avec SQLAlchemy

```
# Installer le driver PostgreSQL pour Python
pip install psycopg2-binary
```

```
from sqlalchemy import create_engine

# Connexion à une base PostgreSQL
engine = create_engine("postgresql+psycopg2://user:password@localhost:5432/nom_base")
```

```python
# Créer une session
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
session = Session()
```

Le reste du code est identique à ce que l'on a vu avec SQLite !

Conclusion

Méthode	Avantages	Inconvénients
`sqlite3`	Simple, intégré à Python	Manuel, peu sûr, pas évolutif
SQLAlchemy	Puissant, flexible, propre	Plus complexe à apprendre

Commencez par `sqlite3` pour comprendre le fonctionnement de SQL
Passez à **SQLAlchemy** quand vos besoins deviennent complexes
Un ORM rend votre code plus propre et maintenable

Documentation officielle : - SQLite Python - SQLAlchemy