Utiliser un ORM avec SQLite : Peewee

Manipuler des objets Python au lieu d'écrire du SQL : décrire ses tables sous forme de classes, lire et écrire des enregistrements, et surtout savoir ce qu'un ORM fait dans ton dos — y compris quand il fait mal son travail.

5TTR 6TTR Découverte

Avec le module `sqlite3`, tu écris le SQL toi-même et Python te renvoie les résultats sous forme de tuples. Un **ORM** inverse la perspective : tu manipules des objets Python, et la bibliothèque écrit le SQL. C'est confortable — à condition de savoir ce qui se passe derrière.

🎯 Objectifs

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

1. **Expliquer** ce qu'est un ORM et quel problème il résout.
2. **Décrire** une table existante sous forme de classe Python.
3. **Lire** des données : filtres, tris, jointures, relations inverses.
4. **Compter et agréger** avec `fn.COUNT`, `fn.AVG` et `group_by`.
5. **Écrire** des données dans une transaction.
6. **Repérer** le problème N+1 et le corriger.
7. **Choisir** entre ORM et SQL direct selon la situation.

Le problème que l'ORM résout

Reprends ce code écrit avec `sqlite3` :

```
cur = con.execute("SELECT nom, prenom, classe_id FROM eleve WHERE id = 1")
ligne = cur.fetchone()          # ('Adam', 'Lucas', 1)
```

Tu obtiens un tuple. Pour connaître le nom de la classe, il faut une deuxième requête, puis recoller les morceaux à la main. Or ton programme, lui, raisonne en termes d'**élèves** et de **classes** — pas de tuples et de colonnes.

C'est ce décalage que l'ORM (*Object-Relational Mapping*, correspondance objet-relationnel) supprime :

Base de données	Python
Une table	Une **classe**
Une ligne	Un **objet**
Une colonne	Un **attribut**
Une clé étrangère	Une **référence** vers un autre objet

Avec un ORM, la même chose s'écrit :

```python
e = Eleve.get(Eleve.id == 1)
print(f"{e.prenom} {e.nom} est en {e.classe.nom}")   # Lucas Adam est en 3TTR
```

🧠 **L'ORM NE REMPLACE PAS LE SQL, IL L'ÉCRIT À TA PLACE.** Toutes les requêtes que tu as apprises au cours sur `SELECT` sont toujours exécutées — tu ne les tapes plus. C'est précisément pourquoi il faut connaître le SQL **avant** d'utiliser un ORM : sinon, le jour où une page met huit secondes à s'afficher, tu n'as aucun moyen de comprendre pourquoi.

Installer Peewee

Contrairement à `sqlite3`, Peewee ne fait pas partie de la bibliothèque standard :

```bash
pip install peewee
```

```python
import peewee
print(peewee.__version__)      # 4.5.0
```

⚠️ **ATTENTION EN CHERCHANT DE L'AIDE EN LIGNE.** Peewee est en version 4, mais l'immense majorité des tutoriels et des réponses trouvées sur le web décrivent la version 3. Les bases sont identiques, certains détails non. Réflexe à prendre : **vérifie ta version**, et privilégie la documentation officielle (lien en sidebar) plutôt qu'un billet de blog non daté.

Décrire les tables

On décrit ici des tables **qui existent déjà** dans `ecole.db`. Chaque classe Python correspond à une table, chaque attribut à une colonne.

```python
from peewee import (SqliteDatabase, Model, CharField, IntegerField,
                    ForeignKeyField, CompositeKey, fn, JOIN)

db = SqliteDatabase("ecole.db", pragmas={"foreign_keys": 1})


class BaseModel(Model):
    """Classe mère : évite de répéter la base dans chaque modèle."""
    class Meta:
        database = db


class Classe(BaseModel):
    nom = CharField(unique=True)
    annee = IntegerField()
    local = CharField(null=True)

    class Meta:
        table_name = "classe"


class Eleve(BaseModel):
    nom = CharField()
    prenom = CharField()
    date_naissance = CharField(null=True)
```

```
    classe = ForeignKeyField(Classe, backref="eleves",
                             column_name="classe_id", null=True)

    class Meta:
        table_name = "eleve"
```

Points à retenir :

- `pragmas={"foreign_keys": 1}` — c'est ici que se règle le pragma `foreign_keys`, celui qui est désactivé par défaut. Peewee l'appliquera à chaque connexion, tout seul. Une bonne raison d'aimer les ORM.
- `table_name` — sans lui, Peewee déduit le nom de la table du nom de la classe. On l'écrit explicitement : ça évite toute surprise et documente le lien.
- `column_name="classe_id"` — l'attribut Python s'appelle `classe` (il contient un objet `Classe`), la colonne SQL s'appelle `classe_id` (elle contient un entier). Les deux noms coexistent volontairement.
- `backref="eleves"` — crée le chemin **inverse** : depuis une classe, `ma_classe.eleves` donne ses élèves. Rien n'est ajouté à la base : c'est du confort côté Python.
- `null=True` — reflète le fait que `classe_id` accepte `NULL`, pour l'élève pas encore affecté.

Les autres modèles suivent la même logique. Pour la table d'association, la clé primaire composée se déclare ainsi :

```
class Inscription(BaseModel):
    eleve = ForeignKeyField(Eleve, backref="inscriptions", column_name="eleve_id")
    cours = ForeignKeyField(Cours, backref="inscriptions", column_name="cours_id")

    class Meta:
        table_name = "inscription"
        primary_key = CompositeKey("eleve", "cours")
```

Lire

Un objet précis

```
c = Classe.get(Classe.nom == "6TTR")
print(c.id, c.nom, c.annee, c.local)       # 4 6TTR 6 A23
```

`get()` lève une exception `DoesNotExist` si rien ne correspond. Quand l'absence est un cas normal, utilise :

```
c = Classe.get_or_none(Classe.nom == "7TTR")
print(c)          # None
```

> ⚠ `CLASSE.NOM == "6TTR"` **N'EST PAS UNE COMPARAISON PYTHON.** Ça ne renvoie ni `True` ni `False` : c'est une **expression** que Peewee traduira en `WHERE nom = ?`. D'où l'usage de `&` et `|` (et non `and` / `or`) pour combiner des conditions, avec des parenthèses obligatoires :
>
> ```
> Eleve.select().where((Eleve.classe == c) & (Eleve.nom % "D%"))
> ```

Une liste

```
for e in Eleve.select().where(Eleve.classe == c).order_by(Eleve.nom).limit(3):
    print(e.nom, e.prenom)
```

```
Bruyère Victor
Collard Maya
Dethier Gaspard
```

La correspondance avec le SQL est directe : `.select()` → `SELECT`, `.where()` → `WHERE`, `.order_by()` → `ORDER BY`, `.limit()` → `LIMIT`.

Suivre les relations

```python
e = Eleve.get(Eleve.id == 1)
print(f"{e.prenom} {e.nom} est en {e.classe.nom}")   # Lucas Adam est en 3TTR

print([x.nom for x in c.eleves.order_by(Eleve.nom)])
# ['Bruyère', 'Collard', 'Dethier', 'Evrard', 'Falize', 'Grégoire']
```

`e.classe` suit la clé étrangère, `c.eleves` suit le `backref` en sens inverse. C'est le confort principal de l'ORM — **et son piège principal** : on y revient avec le problème N+1.

Jointure explicite

```python
q = (Eleve.select(Eleve, Classe)
     .join(Classe)
     .where(Classe.nom == "6TTR")
     .order_by(Eleve.nom))

for e in q:
    print(f"{e.prenom} {e.nom} - {e.classe.nom}")
```

Le `select(Eleve, Classe)` demande de ramener **les colonnes des deux tables en une seule requête**. Sans lui, chaque `e.classe.nom` déclencherait une requête supplémentaire.

Compter et agréger

```python
print(Eleve.select().count())                                  # 33
print(Eleve.select().where(Eleve.classe.is_null()).count())    # 1
```

Pour les agrégats, `fn` donne accès aux fonctions SQL :

```python
q = (Classe
     .select(Classe.nom, fn.COUNT(Eleve.id).alias("nb"))
     .join(Eleve, JOIN.LEFT_OUTER)
     .group_by(Classe.id)
     .order_by(fn.COUNT(Eleve.id).desc()))

for row in q:
    print(f"{row.nom:6} {row.nb}")
```

```
3TTR   10
4TTR   9
5TTR   7
6TTR   6
```

Sur trois tables, avec une moyenne arrondie :

```python
q = (Classe
     .select(Classe.nom, fn.ROUND(fn.AVG(Note.valeur), 2).alias("moyenne"))
     .join(Eleve)
     .join(Note, on=(Note.eleve == Eleve.id))
     .group_by(Classe.id)
     .order_by(fn.AVG(Note.valeur).desc()))
```

```
5TTR   13.35
4TTR   13.22
3TTR   13.14
6TTR   12.95
```

Ce sont exactement les résultats du cours sur `SELECT` — mêmes requêtes, autre écriture.

> 💡 `JOIN.LEFT_OUTER` **RESTE INDISPENSABLE.** L'ORM ne devine pas que tu veux garder les classes vides ou les cours sans inscrit : c'est toujours à toi de choisir entre jointure interne et externe. Tout ce que tu as appris sur `JOIN` / `LEFT JOIN` reste vrai, seule la syntaxe change.

Écrire

```python
nouveau = Eleve.create(nom="Martin", prenom="Sacha",
                       date_naissance="2008-05-02", classe=c)
print(nouveau.id)              # 34

nouveau.prenom = "Sasha"
nouveau.save()                 # UPDATE

nouveau.delete_instance()      # DELETE
```

`create()` fait l'`INSERT` **et** le `commit`, puis renvoie l'objet avec son `id`. Remarque `classe=c` : on affecte un **objet**, pas un entier — Peewee en extrait la clé.

Pour un ensemble d'écritures indissociables, `db.atomic()` joue le rôle du `with con:` de `sqlite3` :

```python
try:
    with db.atomic():
        Classe.create(nom="TEST", annee=3)
        Classe.create(nom="TEST", annee=4)      # viole UNIQUE
except peewee.IntegrityError as ex:
    print("IntegrityError :", ex)

print(Classe.select().where(Classe.nom == "TEST").count())
```

```
IntegrityError : UNIQUE constraint failed: classe.nom
0        ← la première création a été annulée elle aussi
```

Les contraintes de la base restent le dernier rempart : c'est SQLite qui refuse, pas Peewee.

Ce que l'ORM fait dans ton dos

Lire le SQL généré

À la moindre hésitation :

```
print(q.sql())
```

```
SELECT "t1"."nom", ROUND(AVG("t2"."valeur"), ?) AS "moyenne"
FROM "classe" AS "t1"
INNER JOIN "eleve" AS "t3" ON ("t3"."classe_id" = "t1"."id")
INNER JOIN "note" AS "t2" ON ("t2"."eleve_id" = "t3"."id") ...
```

Prends l'habitude de le faire quand un résultat te surprend. Tu remarqueras au passage les `?` : **Peewee utilise les place-holders**, donc il est immunisé contre l'injection SQL. C'est un de ses vrais apports en sécurité.

Le problème N+1

C'est **le** défaut classique des ORM, et la raison n°1 des applications lentes.

```
# ⛔ 1 requête pour les élèves + 1 requête PAR élève pour sa classe
for e in Eleve.select().limit(5):
    print(e.nom, e.classe.nom)          # ← chaque e.classe part en base
```

Cinq élèves : six requêtes. Mille élèves : mille une requêtes. Le code paraît identique, mais la base est massacrée.

```
# ✅ une seule requête
for e in Eleve.select(Eleve, Classe).join(Classe).limit(5):
    print(e.nom, e.classe.nom)
```

```
['Adam/3TTR', 'Bastin/3TTR', 'Charlier/3TTR', 'Delvaux/3TTR', 'Englebert/3TTR']
```

> ⚠️ **RIEN NE T'AVERTIT.** Aucune erreur, aucun message : juste une application qui ralentit à mesure que les données grossissent. La règle : **dès que tu accèdes à un objet lié à l'intérieur d'une boucle, demande-toi si tu ne dois pas le charger dans le** `select()` .

ORM ou SQL direct ?

Situation	Choix
CRUD classique : créer, lire, modifier, supprimer des enregistrements	**ORM** — code plus court, plus lisible
Application dont le modèle évolue souvent	**ORM** — un seul endroit à modifier
Requête analytique complexe : sous-requêtes, fenêtrage, agrégats imbriqués	**SQL** — l'ORM devient plus verbeux que le SQL
Traitement en masse (import de 100 000 lignes)	**SQL** — l'ORM crée un objet par ligne
Optimiser une requête lente déjà identifiée	**SQL** — contrôle total
Script court, jetable	`sqlite3` — pas de dépendance à installer

Et rien n'oblige à choisir globalement : Peewee permet d'exécuter du SQL brut quand c'est le bon outil.

```
for row in db.execute_sql("SELECT nom, annee FROM classe ORDER BY annee"):
```

```
    print(row)
```

🧠 **CE QUE CE COURS VOULAIT VRAIMENT TE MONTRER.** Un ORM est un **outil de confort**, pas un remplacement de la compétence SQL. Il écrit à ta place des requêtes que tu dois rester capable de lire, de critiquer et de corriger. Les développeurs qui ont appris l'ORM sans le SQL sont exactement ceux qui, face à une application lente, ne savent pas par où commencer.

🧪 Exercices

Travaille sur une **copie** de `ecole.db`, avec les modèles complets des six tables.

Exercice 1 — Compléter les modèles ★★☆☆☆

Écris les classes `Professeur`, `Cours` et `Note` manquantes, avec leurs `ForeignKeyField`, leurs `backref` et leurs `table_name`.

Vérifie chacune par une requête : `Cours.get(Cours.id == 1).professeur.nom` doit renvoyer `Lambrechts`.

Exercice 2 — Traduire ★★☆☆☆

Réécris en Peewee les cinq requêtes SQL suivantes, et vérifie que tu obtiens les mêmes résultats qu'en SQL direct :

1. Les élèves de 5TTR, par ordre alphabétique.
2. Le nombre de cours par professeur, du plus chargé au moins chargé.
3. Les cours de plus de 2 périodes.
4. La moyenne par cours, arrondie à une décimale.
5. Les cours sans aucun inscrit. *(Indice : `JOIN.LEFT_OUTER` et `fn.COUNT`.)*

Exercice 3 — Chasse au N+1 ★★★☆☆

```
for note in Note.select().limit(20):
    print(note.eleve.prenom, note.eleve.nom, note.cours.intitule, note.valeur)
```

a) Combien de requêtes ce code exécute-t-il ? Compte précisément. b) Réécris-le pour n'en faire qu'une seule. c) Vérifie ta réponse avec `.sql()`, et compare les deux versions sur les 367 notes en mesurant le temps avec `time.perf_counter()`.

Exercice 4 — Inscrire, proprement ★★★☆☆

Écris `inscrire(nom, prenom, naissance, nom_classe, intitules_cours)` qui crée l'élève et l'inscrit aux cours dont les intitulés sont donnés.

Contraintes : tout dans un `db.atomic()` ; si un intitulé de cours n'existe pas, **rien** n'est enregistré et la fonction lève une exception explicite. Teste les deux cas.

Exercice 5 — Comparer les deux approches ★★★★☆

Écris **deux fois** le même programme — une fois avec `sqlite3`, une fois avec Peewee : afficher, pour chaque professeur, ses cours et le nombre total d'élèves inscrits, trié par ce nombre décroissant.

Puis rédige une comparaison de dix lignes : nombre de lignes de code, lisibilité, nombre de requêtes exécutées, facilité à modifier si on ajoutait une colonne. Conclus en disant lequel tu choisirais **pour ce cas précis**, et pourquoi.

✍️ À retenir

- Un ORM fait correspondre **table** ↔ **classe**, **ligne** ↔ **objet**, **colonne** ↔ **attribut**, **clé étrangère** ↔ **référence**.
- Peewee s'installe (`pip install peewee`), contrairement à `sqlite3`. Vérifie ta version : la documentation en ligne parle surtout de la 3.
- `pragmas={"foreign_keys": 1}` règle une bonne fois pour toutes le piège des clés étrangères.
- `Eleve.nom == "x"` construit une **expression SQL**, pas un booléen : on combine avec `&` et `|`, parenthèses obligatoires.
- `backref` donne le chemin inverse d'une relation, sans rien changer à la base.
- Peewee utilise des **place-holders** : pas d'injection SQL possible.
- Le **N+1** est le piège majeur : accéder à un objet lié dans une boucle multiplie les requêtes, sans aucun avertissement. On le corrige avec `select(A, B).join(B)`.
- `.sql()` affiche la requête générée — le premier réflexe de diagnostic.
- L'ORM ne dispense **jamais** de connaître le SQL : il l'écrit, tu dois le relire.

Suite

Tu as fait le tour de SQLite : ce qu'il est, comment construire une base, comment l'interroger, et deux façons de la piloter en Python. La compétence à consolider maintenant, c'est le SQL lui-même — reprends la section SQL, dont tout le contenu s'applique directement à ce que tu viens d'apprendre.