

Base de données

- Base de données SQLite
- Base de données Objet

Pr Amadou Dahirou GUEYE Octobre 2022

Les bases de données SQL

■ Bases de données relationnelles

- Modèle relationnel : tables + algèbre relationnel
- Langage SQL (Structured Query Language)
- Souvent utilisées dans les applications web pour stocker les données
 - Persistance : les données restent même si l'application est arrêtée / relancée

■ Avantages :

- Rapide, notamment pour faire des recherches complexes (requêtes)
- Peut gérer de gros volume de données

■ Inconvénients :

- Le modèle relationnel est difficile à concilier avec le modèle objet
 - Plusieurs approches possibles :
 - Approche non-objet (relationnel pur)
 - Approche mixte relationnel objet, en Python :
 - « Méta-programmation »
 - Module SQLAlchemy

Les bases de données SQL



Plusieurs moteurs de bases de données SQL :

- MySQL / MariaDB



- Base de données SQL optimisé pour des très gros volumes

- PostgreSQL



- Base de données SQL avec des fonctions avancées (procédure stockée, base objet,...)

- Oracle



- système de gestion de base de données relationnelle créé et exécuté par Oracle Corporation.



- Les licences de moteur de base de données Oracle sont entièrement propriétaires, avec des options gratuites et payantes disponibles.

- SQLite3



- Base de données SQL minimaliste



- Une base de données = un fichier



Dans la quasi-totalité des cas, SQLite3 est préférable :

- Plus facile d'emploi



- Intégré à Python (SQLite a été conçu pour être intégré dans le programme même)

- Plus rapide



- SQLite3 tourne dans le même processus que votre programme

- Jusqu'à 3 Go environ

Avantages SQLite

- Pas besoin d'installer ou configurer un serveur pour avoir la base de données
- SQLite stocke en local la base dans un fichier
- Accès plus rapide que les autres bases de données
- Pour l'ouvrir sur le vs code il faut des extensions pour voir la base de données

SQLite3

- Syntaxe SQL classique
- Types de données supportés :
 - integer : entier
 - real : nombre réel (flottant)
 - text : chaîne de caractères (en Unicode / UTF-8)
 - blob : « blob » (chaîne) binaire
 - Utilisé pour stocker des données binaires (ex image JPEG ou PNG) ou des objets sérialisés
- En ligne de commande : installer le programme **sqlite3**
 - Pour interroger la base manuellement :
sqlite3 fichier.sqlite3
 - Pour exécuter un script SQL (= un fichier avec les commandes)
sqlite3 fichier.sqlite3 < script.sql

SQLite3 en Python

- Le module `sqlite3` permet d'utiliser SQLite3 en Python

-*- coding: utf-8 -*-

`import sqlite3`

- Pour se connecter à une base de données :

`db = sqlite3.connect("fichier.sqlite3")`

◆ La base est créée automatiquement si le fichier n'existe pas

- Pour créer une base de données stockées en mémoire vive :

`db = sqlite3.connect(":memory:")`

- Pour fermer la connexion vers la base:

`db.close()`

- Il faut ensuite créer un « curseur » :

`cursor = db.cursor()`

- Le curseur permet d'effectuer des requêtes SQL :

`cursor.execute(" requête SQL ")`

- Si la base est modifiée (requête insert, update, create, etc), il faut enregistrer les changements avec :

`db.commit()`

SQLite3 en Python

■ Créer une table avec SQLite

```
cursor = db.cursor()
```

```
cursor.execute("""
```

```
CREATE TABLE IF NOT EXISTS students(  
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,  
    name TEXT,  
    age INTERGER  
)  
""")  
db.commit()
```

SQLite3 en Python

■ Supprimer une table avec SQLite

```
cursor = db.cursor()
cursor.execute("""
DROP TABLE students
""")
db.commit()
```

■ Insérer des données

- la plus simple étant celle-ci:

```
cursor.execute("""
INSERT INTO students(name, age) VALUES(?, ?)""", ("Sall", 25))
```

- Vous pouvez passer par un dictionnaire

```
data = {"name" : "Da", "age" : 30}
cursor.execute("""
INSERT INTO students(name, age) VALUES(:name, :age)""", data)
```

- Vous pouvez récupérer l'id de la ligne que vous venez d'insérer

```
id = cursor.lastrowid
print('dernier id: %d' % id)
```


SQLite3 en Python

- Il est également possible de faire plusieurs **insert** en une seule fois avec la fonction **executemany** :

```
users = []
```

```
users.append(("Sow", 20))
```

```
users.append(("Seck", 40))
```

```
cursor.executemany("""
```

```
INSERT INTO students(name, age) VALUES(?, ?)""", users)
```

SQLite3 en Python

- Pour les requêtes de type select, les résultats ne sont pas renvoyés directement
 - `cursor.fetchone()` retourne une ligne de résultat
 - `cursor.fetchall()` retourne une liste avec tous les résultats
 - Chaque résultat est un tuple avec les attributs demandés (même s'il n'y en a qu'un seul !)

■ Exemple1 :

- Récupérer la première ligne

```
cursor.execute("""SELECT name, age FROM students""")
user1 = cursor.fetchone()
print(user1)
```

Le résultat est un tuple:

('Da', 30)

- Vous pouvez récupérer plusieurs données de la même recherche en utilisant la fonction **fetchall()**

```
cursor.execute("""SELECT id, name, age FROM students""")
rows = cursor.fetchall()
for row in rows:
    print('{0} : {1} - {2}'.format(row[0], row[1], row[2]))
```

SQLite3 en Python

Exemple2: Pour la recherche spécifique

```
cursor.execute("select nom, prenom, age from table where id = 1")
```

```
nom, prenom, age = cursor.fetchone() # => ("Gueye", "AD", 36)
```

```
cursor.execute("select age from table where id = 1")
```

```
age = cursor.fetchone()[0] # => (36,) : tuple de un seul élément,  
d'où le [0]
```

```
cursor.execute("select nom, prenom, age from table")
```

```
for nom, prenom, age in cursor.fetchall() :
```

```
    print(nom, prenom, age)
```

```
cursor.execute("select nom from table")
```

```
for nom , in cursor.fetchall() : # Attention ne pas oublier la virgule  
    après nom !
```

```
    print(nom)
```

SQLite3 en Python

- Pour utiliser des variables dans les requêtes, on utilise le caractère « ? » et les valeurs sont données ensuite

Exemple :

```
prenom, nom = "AD", "Gueye"  
db_cursor.execute("select * from table where prenom = ? and nom = ?", (prenom, nom))
```

Attention, les valeurs doivent toujours être passées dans un tuple, même s'il n'y en a qu'une

```
nom = "Gueye"  
db_cursor.execute("select * from table where nom = ?", (nom,))
```

- Ne JAMAIS utiliser « %s » pour passer les valeurs des paramètres dans les requêtes

```
db_cursor.execute("select * from table where nom = '%s' " % nom)
```

- ◆ Les paramètres sont saisis par l'utilisateur et peuvent faire l'objet de tentative de piratage (injection SQL)

- ◆ Par exemple si l'utilisateur saisit comme nom ' ; drop table xxx ;

- En revanche on peut être amené à utiliser %s pour le nom de la table / des colonnes

```
colonne = "nom" # Attention à être sûr de l'origine de la valeur de cette variable !
```

```
db_cursor.execute("select * from table where %s = ?" % colonne, (nom,))
```

Gestion des erreurs

- Il est recommandé de toujours encadrer les opérations sur des bases de données et d'anticiper

```
import sqlite3
```

try:

```
    conn = sqlite3.connect('data/users.db')
```

```
    cursor = conn.cursor()
```

```
    cursor.execute(""" CREATE TABLE students(id INTEGER PRIMARY KEY  
AUTOINCREMENT UNIQUE, name TEXT, age INTEGER ) """)
```

```
    conn.commit()
```

except sqlite3.OperationalError:

```
    print('Erreur la table existe déjà')
```

except Exception as e:

```
    print("Erreur")
```

```
    conn.rollback()
```

finally:

```
    conn.close()
```

Gestion des erreurs

■ Les erreurs que vous pouvez intercepter:

Error

DatabaseError

DataError

IntegrityError

InternalError

NotSupportedError

OperationalError

ProgrammingError

InterfaceError

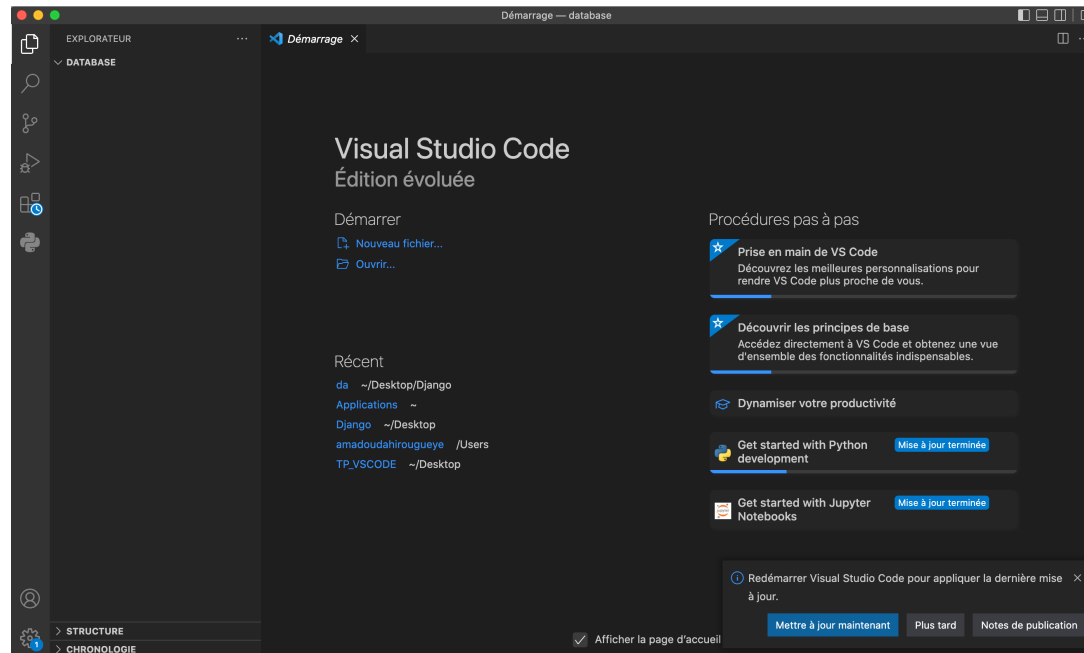
Warning

Partie 2: TP SQLite

TP Base de données SQLite

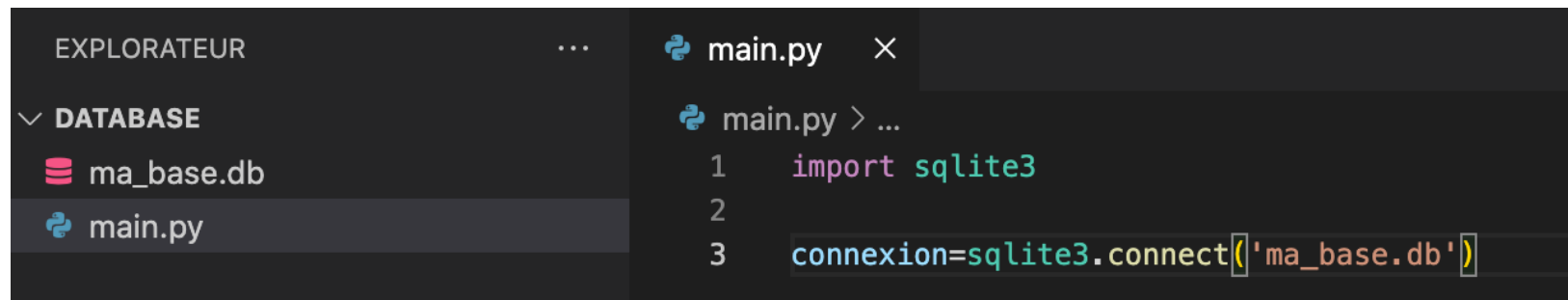
- Créer un dossier dans desktop nommé database, s'y déplacer et ouvrir vs code

```
[amadoudahirougueye@MBP-de-Amadou ~ % cd Desktop  
[amadoudahirougueye@MBP-de-Amadou Desktop % mkdir database  
[amadoudahirougueye@MBP-de-Amadou Desktop % cd database  
amadoudahirougueye@MBP-de-Amadou database % code .
```



Base de données SQLite: création base

- Créons un fichier main.py et un morceau de code qui nous permet d'avoir notre base de données
- Après exécution, on a à gauche notre base créée

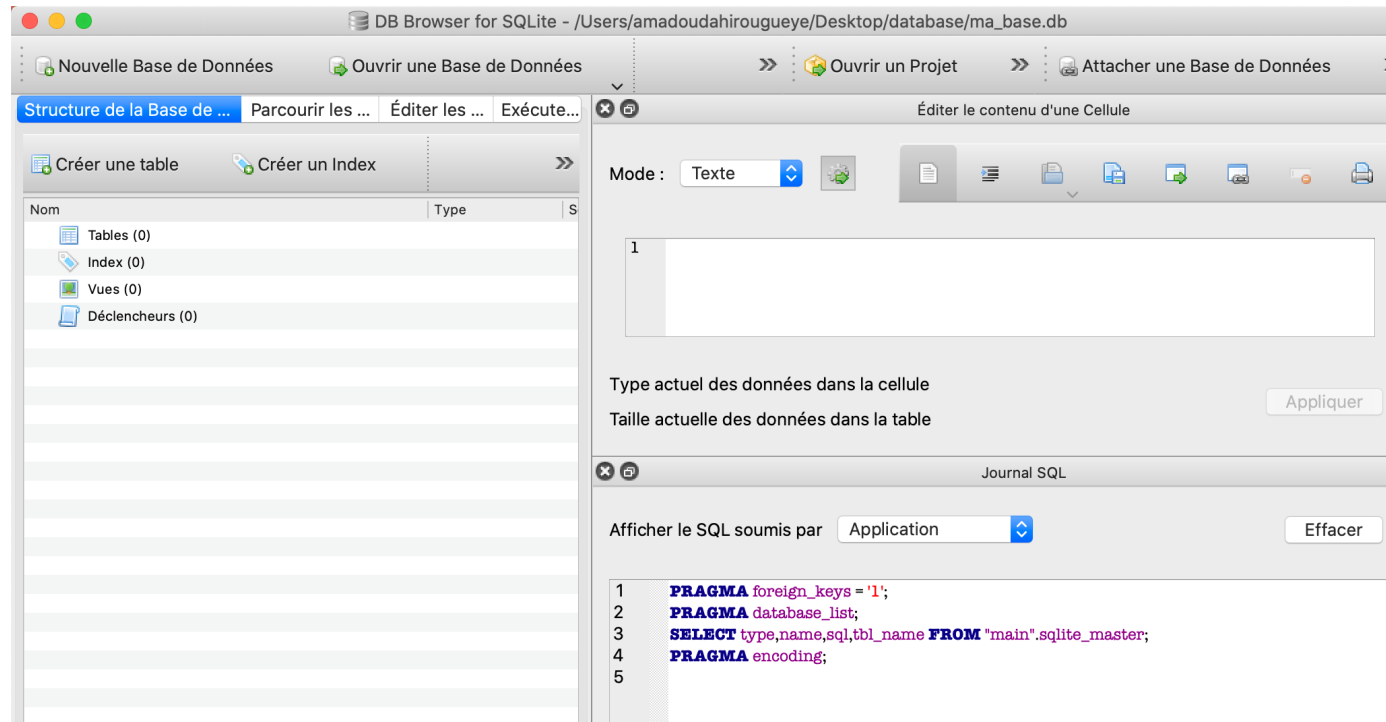


```
EXPLORATEUR    ...    main.py  X
v DATABASE
  ma_base.db
  main.py
main.py > ...
1  import sqlite3
2
3  connexion=sqlite3.connect('ma_base.db')
```

- Il faut fermer la connexion par `connexion.close()` pour libérer l'objet
- L'extension d'une base de données est soit `.bd` ou `.sqlite3`
- **DB browser for SQLite** nous permet de voir, d'ouvrir et d'effectuer des requêtes dans une base de données SQLite. Voir <https://sqlitebrowser.org/>

Base de données SQLite

- Pour importer la base de données depuis DB browser
 - Aller files puis ouvrir base de données et choisir la base de données

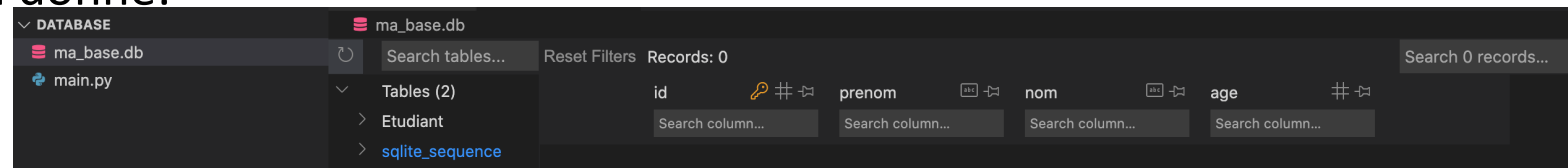


Création de table

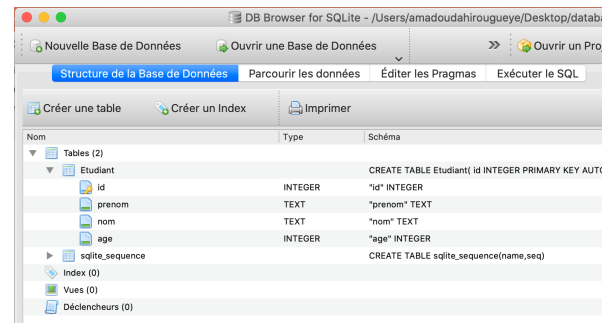
- Création de la table etudiant avec l'ID, le prénom, le nom et l'âge
- Création d'un objet curseur qui nous permet de faire des requêtes

```
main.py > ...
1 import sqlite3
2
3 connexion=sqlite3.connect('ma_base.db')
4 # création de la table étudiant
5 cursor=connexion.cursor()
6 cursor.execute("""
7 CREATE TABLE Etudiant(
8     id INTEGER PRIMARY KEY AUTOINCREMENT,
9     prenom TEXT,
10    nom TEXT,
11    age INTEGER
12 )
13 """)
14 connexion.commit() # pour appliquer la transaction dans la base
15 connexion.close()
```

- A partir de là, on peut exécuter et ouvrir la base avec l'extension SQLite Viewer et ça donne:



- Actualisons DB Browser



Création de table

- Si on exécute de nouveau la requête, il va signaler une erreur parce que la table étudiant a déjà été créée. On peut modifier la création comme suit:

CREATE TABLE IF NOT EXISTS Etudiant

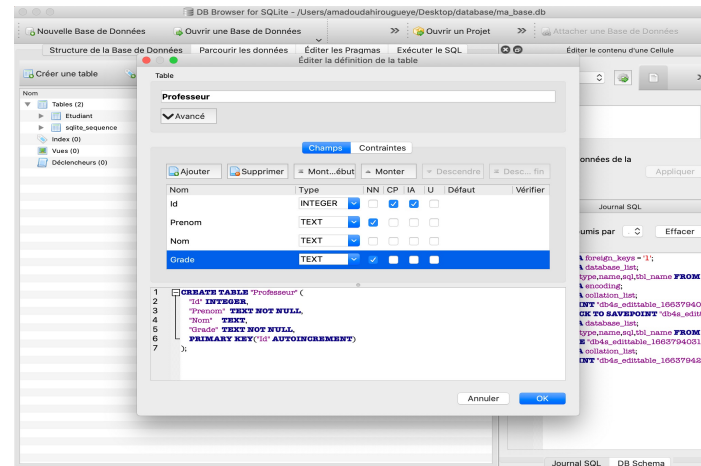
- Pour faire mieux, il faut créer une exception

```
try:
    cursor.execute("""
    CREATE TABLE Etudiant (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        prenom TEXT,
        nom TEXT,
        age INTEGER
    )
    """)
except Exception as error:
    print(error)
```

Insertion au niveau de la base

■ On peut insérer soit avec DB Browser ou avec python

■ Avec DB Browser

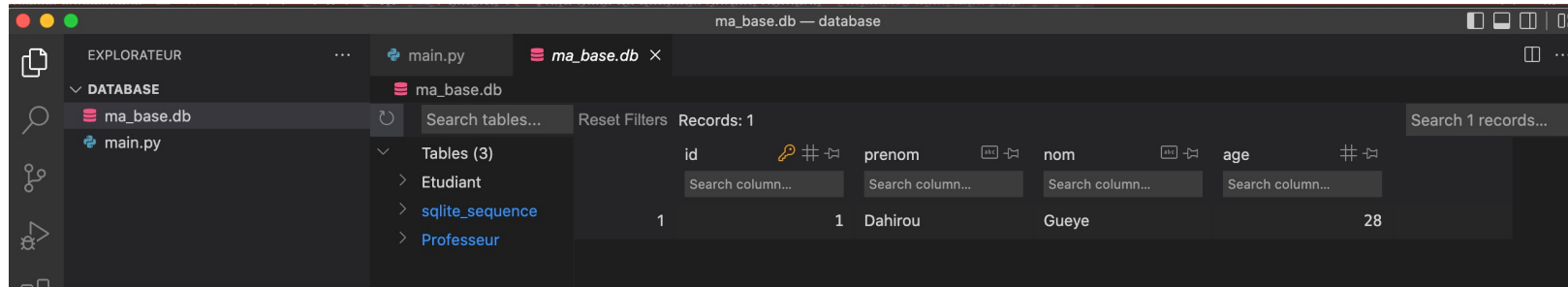


■ Avec python

```
main.py x ma_base.db  
main.py > ...  
1 import sqlite3  
2  
3 connexion=sqlite3.connect('ma_base.db')  
4 # création de la table étudiant  
5  
6 try:  
7  
8 # cursor.execute("""  
9 # CREATE TABLE Etudiant(  
10 #   id INTEGER PRIMARY KEY AUTOINCREMENT,  
11 #   prenom TEXT,  
12 #   nom TEXT,  
13 #   age INTEGER  
14 # )  
15 # """)  
16 cursor=connexion.cursor()  
17 cursor.execute(""" INSERT INTO Etudiant (prenom, nom, age) Values (?, ?, ?) """, ("Dahirou", "Gueye", 28))  
18  
19 except Exception as error:  
20 | print (error)  
21  
22 connexion.commit()  
23 connexion.close()
```

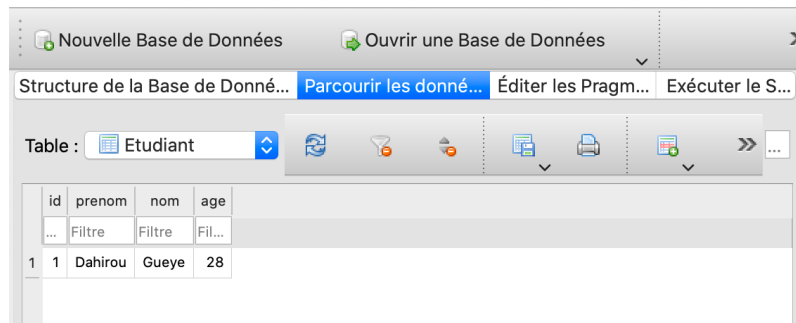
Insertion au niveau de la base

■ Vérification



■ Pour voir la donnée enregistrée dans DB Browser

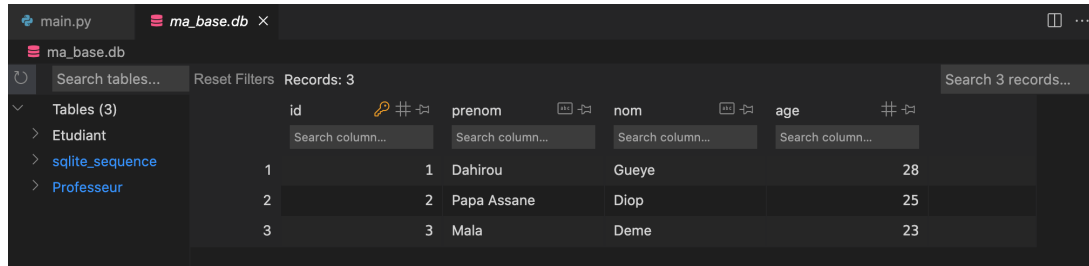
■ Sélectionner la table concernée puis parcourir les données



■ A partir de DB Browser, on peut ajouter un nouvel enregistrement à partir de new record ou insérer un nouvel enregistrement. Il faut enregistrer les modifications. Dans la pratique on va ajouter deux autres étudiants qui ne sont pas montrés ici

Interrogation des données

Visualisation depuis SQLite Viewer



The screenshot shows the SQLite Viewer interface. On the left, a sidebar lists the database 'ma_base.db' and its tables: 'Etudiant', 'sqlite_sequence', and 'Professeur'. The main area displays the 'Etudiant' table with 3 records. The table has columns: id, prenom, nom, and age. The records are: (1, Dahirou, Gueye, 28), (2, Papa Assane, Diop, 25), and (3, Mala, Deme, 23).

id	prenom	nom	age
1	Dahirou	Gueye	28
2	Papa Assane	Diop	25
3	Mala	Deme	23

Pour afficher les enregistrements depuis python

```
16 cursor=connexion.cursor()
17 # cursor.execute(""" INSERT INTO Etudiant (prenom, nom, age) Values (?, ?, ?) """, ("Dahirou", "Gueye", 28))
18 cursor.execute(""" SELECT id, prenom, nom, age FROM Etudiant""")
19 etudiants=cursor.fetchall()
20 print(etudiants)
```

Visualisation en ligne de commande

```
amadoudahirougueye@MBP-de-Amadou database % /usr/local/bin/python3 /Users/amadoudahirougueye/Desktop/database/main.py
[(1, 'Dahirou', 'Gueye', 28), (2, 'Papa Assane', 'Diop', 25), (3, 'Mala', 'Deme', 23)]
amadoudahirougueye@MBP-de-Amadou database %
```

Parcours et affichage des étudiants

for etudiant in etudiants:

print(etudiant[0], etudiant[1], etudiant[2])

```
amadoudahirougueye@MBP-de-Amadou database % /usr/local/bin/python3 /Users/amadoudahirougueye/Desktop/database/main.py
1 Dahirou Gueye
2 Papa Assane Diop
3 Mala Deme
amadoudahirougueye@MBP-de-Amadou database %
```

Interrogation des données

■ Récupération d'un étudiant

```
16 cursor=connexion.cursor()
17 # cursor.execute(""" INSERT INTO Etudiant (prenom, nom, age) Values (?, ?, ?) """, ("Dahirou", "Gueye", 28))
18 cursor.execute(""" SELECT id, prenom, nom, age FROM Etudiant WHERE id=?""", (1,))
19 etudiant=cursor.fetchone()
20 print(etudiant)
```

■ Visualisation

```
amadoudahirougueye@MBP-de-Amadou database % /usr/local/bin/python3 /Users/amadoudahirougueye/Desktop/database/main.py
(1, 'Dahirou', 'Gueye', 28)
amadoudahirougueye@MBP-de-Amadou database %
```


Approche objet

■ Il est souvent compliqué de mélanger modèle relationnel SQL et objet :

- ◆ Les relations entre objets ne peuvent pas être stockées dans la base de données en tant que telles
- ◆ Le modèle relationnel ne supporte pas l'héritage
- ◆ Jusqu'à présent, dans le modèle objet du musée virtuel, TOUS les objets étaient chargés en mémoire
 - L'objectif d'une base de données est justement de ne pas tout charger

■ En pratique, on utilise rarement une base de données SQL « brute » en Python

■ Des modules ORM (*Object Relational Mapper*) permettent de faire la correspondance entre modèle objet et relationnel

- ◆ SQLAlchemy
- ◆ Pony
- ◆ SQLAlchemy
- ◆ ORMythorique
- ◆ ...
- ◆ On peut aussi développer son propre ORM facilement en Python !

SQLObject

- Un ORM qui s'utilise en « surcouche » à une base de données SQL classique
- Supporte plusieurs bases de données SQL :
 - ◆ SQLite3
 - ◆ MySQL, PostgreSQL, FireBird, Sybase, Max DB, MS SQL server
- Prend en charge automatiquement la conversion entre le modèle objet et le modèle relationnel
- Chargement de la base :

```
import sqlobject  
  
connection = sqlobject.connectionForURI("sqlite://" + "fichier.sqlite3")  
sqlobject.sqlhub.processConnection = connection
```
- Un mode « débogage » est disponible : SQLObject affiche toutes les requêtes dans le terminal

```
connection.debug = True
```

SQLObject

■ Création des classes :

- ◆ 1 classe = 1 table
- ◆ 1 attribut d'une classe = 1 colonne dans la table
- ◆ 1 objet = 1 ligne dans la table
- ◆ Les classes doivent hériter de `sqlobject.SQLObject`
- ◆ En général, elles n'ont pas de constructeur
- ◆ Les attributs sont définis directement dans la classe, à l'aide des types d'attributs proposés par `SQLObject`

class Patient(sqlobject.SQLObject) :

```
nom      = sqlobject.StringCol()  
prenom  = sqlobject.StringCol()  
age      = sqlobject.IntCol()
```

Indexe l'attribut nom

```
index_nom = sqlobject.DatabaseIndex("nom")
```

SQLObject

■ Création des classes :

◆ Les types suivants sont définis :

- `sqlobject.StringCol` : chaîne de caractère
- `sqlobject.IntCol` : entier
- `sqlobject.FloatCol` : flottant
- `sqlobject.BoolCol` : booléen
- `sqlobject.ForeignKey` : objet issu d'une autre table
- ...

◆ SQLObject ajoute automatiquement un identifiant `id` numérique

◆ La méthode de classe `createTable()` permet de créer la table automatiquement :

`Patient.createTable(ifNotExists = True)`

◆ Les objets sont créés en appelant le constructeur et en nommant les paramètres :,

`Patient(nom="Gueye", prenom= " AD", age = 36)`

SQLObject

■ Relation 1 - 1 :

- ◆ Une relation 1-1 lie un objet d'une classe à un seul objet d'une autre classe, et réciproquement

```
class Patient(sqlobject.SQLObject) :
```

```
    nom      = sqlobject.StringCol()
```

```
    prenom   = sqlobject.StringCol()
```

```
    lit       = sqlobject.ForeignKey("Lit", cascade = "null")
```

Action en cas de suppression
de l'objet lit ("null" : on
remplace le lit par null / None)

```
class Lit(sqlobject.SQLObject) :
```

```
    numero = sqlobject.IntCol()
```

```
    patient = sqlobject.SingleJoin("Patient")
```

- ◆ Une colonne **lit_id** est automatiquement ajouté à la table **Patient**
- ◆ NB on aurait pu échanger le **ForeignKey** et le **SingleJoin**

SQLObject

■ Relation 1 - * (ou * - 1) :

- ◆ Une relation 1-* lie un objet d'une classe A à plusieurs objets d'une classe B
- ◆ Chaque objet de la classe B est lié à un seul objet de la classe A

class Patient(sqlobject.SQLObject) :

```
nom      = sqlobject.StringCol()
prenom   = sqlobject.StringCol()
ville     = sqlobject.ForeignKey("Ville", cascade = "null")
```

```
index_ville = sqlobject.DatabaseIndex("ville")
```

Indexe pour accélérer
les recherches

class Ville(sqlobject.SQLObject) :

```
nom      = sqlobject.StringCol()
patients = sqlobject.MultipleJoin("Patient")
```

- ◆ Une colonne **ville_id** est automatiquement ajouté à la table **Patient**
- ◆ NB on ne peut mettre la relation dans l'autre sens !

SQLObject

■ Relation * - *

- ◆ Une relation *-* lie un objet d'une classe A à plusieurs objets d'une classe B

- ◆ Chaque objet de la classe B est lié à plusieurs objets de la classe A

class Patient(sqlobject.SQLObject) :

```
    nom    = sqlobject.StringCol()  
    prenom = sqlobject.StringCol()  
    maladie = sqlobject.RelatedJoin("Maladie")
```

class Maladie(sqlobject.SQLObject) :

```
    nom    = sqlobject.StringCol()  
    patient = sqlobject.RelatedJoin("Patient")
```

- ◆ Une table de liaison est automatiquement créée

- ◆ Les méthodes **Patient.addMaladie()**, **Patient.removeMaladie()**, **Maladie.addPatient()**, **Maladie.removePatient()** sont automatiquement ajoutées

SQLObject

- Les objets et les attributs sont ensuite utilisés normalement

```
patient = Patient(nom="Gueye", prenom="AD", age = 36)
```

```
patient.prenom = "Amadou D"
```

```
print(patient.nom)
```

```
maladie = Maladie(nom="Allergie")
```

```
patient.addMaladie(maladie)
```

- ◆ La base de donnée est automatiquement mise à jour !

- Création de ligne

- Modification des champs

- ◆ Les objets créés à partir de la base de données sont créés, mis en cache puis détruit automatiquement lorsqu'ils deviennent inutiles !

- ◆ La méthode **destroySelf()** supprime un objet de la base de données :

```
maladie.destroySelf()
```

- Supprime automatiquement les relations

SQLObject

■ Les requêtes peuvent se faire :

◆ Soit directement en SQL, en récupérant un « curseur » SQL :

```
cursor = sqlobject.sqlhub.processConnection.getConnection().cursor()
cursor.execute("select count(id) from patient")
return cursor.fetchone()[0]
```

◆ Soit dans le langage de requête intégré à SQLObject, par exemple :

```
for patient in Patient.select(Patient.nom == "Gueye") :
    print(patient)

for patient in Patient.select("nom = 'Gueye' ") :
    print(patient)

nb = Patient.select().count()
min_age = Patient.select().min("age")
```

● Pour plus d'information se référer à la documentation de SQLObject :

<http://sqlobject.org/SQLObject.html>

SQLObject

■ Héritage

- ◆ SQLObject supporte l'héritage simple (mais pas multiple)

- ◆ Il faut utiliser la classe **InheritableSQLObject** :

```
import sqlobject, sqlobject.inheritance
```

```
class Personne(sqlobject.inheritance.InheritableSQLObject) :  
    prenom = sqlobject.StringCol()  
    nom    = sqlobject.StringCol()
```

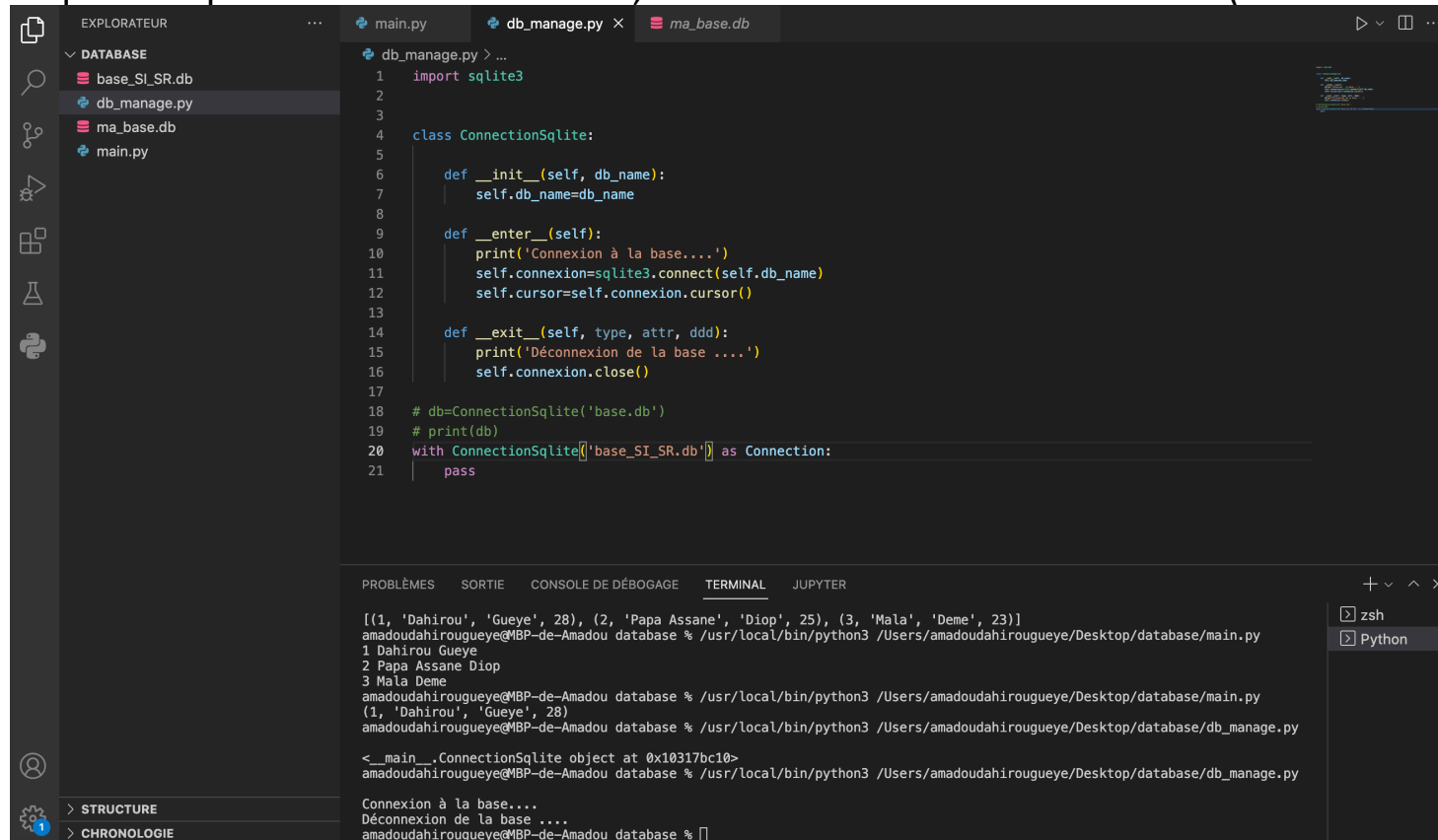
```
class Utilisateur(Personne) :  
    mot_de_passe = sqlobject.StringCol()
```

Création ORM

- On va essayer d'imiter le ORM de Django
- A l'état actuel, pour avoir accès à la base comme la création d'une table, il nous faut exécuter une requête à travers le code SQL
- Du côté, on aura pas besoin de faire des codes SQL
- Pour avoir ça transparent, on va créer notre propre ORM qui va communiquer avec la base de données
- Ce qu'on doit faire, pour accéder à la base on va seulement créer un modèle (une classe) par exemple Etudiant et c'est à nous de renseigner les attributs
- Du côté si on veut créer une table on va aller dans un fichier, on va créer seulement un modèle étudiant et remplir la table
- Créons maintenant un fichier models.py qui va contenir les classes sous forme de tables. Après on ira dans le terminal pour exécuter les commandes qui permettent de créer la base

Création ORM

- On va créer une classe qui va nous permettre de nous connecter et nous déconnecter de la base
- Créons un fichier db_manage.py qui nous permet de gérer la base de données dans laquelle on va définir deux fonctions `__enter__` et `__exit__` qui seront utilisées par les instances.
- A chaque fois qu'on créera une instance, on doit donner le nom de la base (via le constructeur)



The screenshot shows a code editor with three tabs: `main.py`, `db_manage.py`, and `ma_base.db`. The `db_manage.py` file contains the following Python code:

```
1 import sqlite3
2
3
4 class ConnectionSqlite:
5
6     def __init__(self, db_name):
7         self.db_name=db_name
8
9     def __enter__(self):
10         print('Connexion à la base...')
11         self.connexion=sqlite3.connect(self.db_name)
12         self.cursor=self.connexion.cursor()
13
14     def __exit__(self, type, attr, ddd):
15         print('Déconnexion de la base ....')
16         self.connexion.close()
17
18 # db=ConnectionSqlite('base.db')
19 # print(db)
20 with ConnectionSqlite('base_SI_SR.db') as Connection:
21     pass
```

The terminal at the bottom shows the execution of the code:

```
[[('1', 'Dahirou', 'Gueye', 28), ('2', 'Papa Assane', 'Diop', 25), ('3', 'Mala', 'Deme', 23)]]
amadouahirougueye@MBP-de-Amadou database % /usr/local/bin/python3 /Users/amadouahirougueye/Desktop/database/main.py
1 Dahirou Gueye
2 Papa Assane Diop
3 Mala Deme
amadouahirougueye@MBP-de-Amadou database % /usr/local/bin/python3 /Users/amadouahirougueye/Desktop/database/main.py
(1, 'Dahirou', 'Gueye', 28)
amadouahirougueye@MBP-de-Amadou database % /usr/local/bin/python3 /Users/amadouahirougueye/Desktop/database/db_manage.py

<_main_.ConnectionSqlite object at 0x10317bc10>
amadouahirougueye@MBP-de-Amadou database % /usr/local/bin/python3 /Users/amadouahirougueye/Desktop/database/db_manage.py

Connexion à la base...
Déconnexion de la base ....
amadouahirougueye@MBP-de-Amadou database %
```

Création ORM

- Ajoutons à cette classe trois autres fonctions (create_table(), all() et show())

```
9      def __enter__(self):
10          print('Connexion à la base...')
11          self.connexion=sqlite3.connect(self.db_name)
12          self.cursor=self.connexion.cursor()
13
14      def create_table(self):
15          pass
16
17      def all(self):
18          pass
19
20      def show (self):
21          pass
22
```

- Pour rappel, sous Django, le fichier manage.py sert à exécuter le serveur ou le shell ou etc.
- Créons un fichier manage.py
- Compléter le TP pour arriver à la création de votre propre ORM