

## Module : Informatique

Classes: 2<sup>ème</sup> Années MP – PC – T

# Chapitre 4: Passage à la pratique (Module SQLite3 de Python)

Enseignant : Dr. Khemaies GHALI (ghali\_khemaies@yahoo.fr)

Page Facebook: <https://www.facebook.com/groups/ipein.info/>

Année universitaire 2017/2018

IPEIN 2017/2018

2ème Année - Informatique - Chapitre 4

1/51

## Plan

1. Introduction
2. Création d'une base
3. Création de tables (schéma de la table alimentation)
4. Langage SQL: manipulation des données
  - Projection
  - Sélection
  - Opérateurs SQL
5. Exercice

IPEIN 2017/2018

2ème Année - Informatique - Chapitre 4

2/51

## Introduction (1/2)

- ❑ Dans ce chapitre nous allons traiter des différentes manières d'agir sur **des bases de données avec Python**,
- ❑ Il existe de nombreux logiciels de gestion de BD relationnelle sur le marché comme **postgreSQL, mySQL, SQLite ...**
- ❑ Une **BD relationnelle se gère via le langage SQL**,
- ❑ Dans ce cours nous allons utiliser **SQLite**.
  - SQLite est un **système de gestion de base de données (SGBD)** écrit en C,
  - Qui sauvegarde la base sous forme d'un **fichier multiplateforme**.
  - SQLite est un moyen **plus rapide et très simple** de gérer une BD,
  - SQLite existe comme **paquet standard sous Python: sqlite3**

IPEIN 2017/2018

2ème Année - Informatique - Chapitre 4

3/51

## Introduction (2/2)

- ❑ Une base de données est un **ensemble de tables**, que l'on interroge pour en extraire des informations à l'aide **d'un langage de requêtes, majoritairement SQL**.
- ❑ Avec Python, **pour une application modeste**, le plus simple est d'utiliser le **SQLite**:
  - **SQLite** est une base de données stockée dans un seul fichier (pas besoin de serveur):



IPEIN 2017/2018

2ème Année - Informatique - Chapitre 4

4/51

## La syntaxe SQL et les requêtes de base (1/4)

### 1- Opération de Projection

- En Algèbre relationnelle:

$$\pi_{A1, \dots, An}(R)$$

- Syntaxe en SQL:

SELECT A1, . . . ,An FROM R; Ou

SELECT \* FROM R;

### 2- Opération de Sélection

- En Algèbre relationnelle:

$$\sigma_{P(A)}(R)$$

- Syntaxe en SQL:

SELECT \* FROM R WHERE P(A) ;

## La syntaxe SQL et les requêtes de base (2/4)

### 3- Opération d'Intersection

- En Algèbre relationnelle:

$$R1 \cap R2$$

- Syntaxe en SQL:

(SELECT \* FROM R1) INTERSECT (SELECT \* FROM R2) ;

### 4- Opération d'Union

- En Algèbre relationnelle:

$$R1 \cup R2$$

- Syntaxe en SQL:

(SELECT \* FROM R1) UNION (SELECT \* FROM R2) ;

### 5- Opération de Différence

- En Algèbre relationnelle:

$$R1 \setminus R2 \text{ ou } R1 - R2$$

- Syntaxe en SQL:

(SELECT \* FROM R1) EXCEPT (SELECT \* FROM R2) ;

## La syntaxe SQL et les requêtes de base (3/4)

### 6- Opération de Jointure

- En Algèbre relationnelle:

$$R1 \bowtie_{A=B} R2 = R1 [A=B] R2$$

- Syntaxe en SQL:

SELECT \* FROM R1 JOIN R2 ON A=B;

=> Le produit cartésien s'exprime simplement comme une jointure sans condition ON.

- Dans le cas où l'on souhaite réaliser plusieurs jointures, on pourra utiliser un produit cartésien suivi d'une condition WHERE.

➤ Ainsi, la jointure  $R [A = B] R' [C = D] R''$  s'écrit :

➤ SELECT \* FROM R, R', R'' WHERE A=B AND C=D;

## La syntaxe SQL et les requêtes de base (3/4)

### 7- Opération d'Agrégation

- En Algèbre relationnelle:

$$x \gamma_{f(Y)}(R) = \pi_{A1, \dots, An} \gamma_{f1(B1), \dots, fm(Bm)}(R)$$

$$\gamma_{f(Y)}(R) = \gamma_{f1(B1), \dots, fm(Bm)}(R)$$

- Syntaxe en SQL:

SELECT X, f(Y) FROM R GROUP BY A;

SELECT f(Y) FROM R;

Avec f: COUNT, MIN, MAX, AVG, SUM

- Il est possible de coupler l'agrégation à une projection:

SELECT max(note) AS note FROM eleve WHERE classe=1;

- Il est possible de l'utiliser dans des comparaisons:

SELECT \* FROM eleves WHERE note >= (SELECT avg(note) FROM eleves);

Algèbre relationnelle	SQL
comptage	COUNT
max	MAX
min	MIN
somme	SUM
moyenne	AVG

# La syntaxe SQL et les requêtes de base (4/4)

## 8- Opération de renommage

- ❑ Pour renommer un attribut, on dans une projection le mot-clé **AS**.
  - Par exemple, soit R une relation de schéma  $(A_1, \dots, A_n, C_1, \dots, C_m)$  où l'on souhaite obtenir  $\rho_{A_1, \dots, A_n \leftarrow B_1, \dots, B_n}(R)$
  - On effectue :  
`SELECT A1 AS B1, ..., An AS Bn, C1, ..., Cm FROM R;`
  - Il est donc nécessaire d'indiquer tous les attributs.

## Remarques

- ❑ Les requêtes SQL **ne sont pas sensibles à la casse**,
  - mais il est d'usage de mettre les **mots-clés en majuscules** et les **attributs en minuscules**.
  - Les **valeurs des attributs non numériques** sont écrites entre guillemets simples ou doubles.
- ❑ On note que :
  - Toutes les requêtes de recherche commencent par le mot-clé **SELECT**, qui effectuera une projection en fin de requête.
  - La relation sur laquelle on opère est précisée par **FROM**.
  - Les requêtes SQL se terminent systématiquement par un **point-virgule**.
- ❑ **Attention** : SELECT n'effectue pas une sélection, mais une projection!

## Exercice

- ❑ On considère les relations :

classe			
id	Filière	Numéro	Prof.
1	ST	1	Ghali
2	ST	2	Said
3	SM	1	Bechikh
4	SM	2	Garoui

- ❑ Traduire les requêtes suivantes en langage SQL:

- Obtenir la liste des filières proposées dans cet établissement.
- Obtenir toutes les informations concernant les classes de SM.
- Obtenir les prénoms des élèves des classes 1 et 3.
- Obtenir les noms et les notes des élèves ayant eu une note inférieure à 10.

élève			
Nom	Prénom	Classe	Note
Nabli	Ali	1	12.5
Tounsi	Salah	2	14.6
Abidi	Olfa	4	10
Sassi	Ons	2	8.5
Saidi	Asma	1	15
Béji	Sahbi	3	9.5
Mrad	Walid	1	15.5
Sahli	Amine	3	12

## Exercice – Corrigé (1/2)

- Il suffit d'effectuer une projection sur l'attribut correspondant.

SELECT Filière	
FROM classe;	
ST	
ST	
SM	
SM	

- => On note que SQL ne fusionne pas les doublons dans une table. On peut forcer cette fusion à l'aide du mot-clé **DISTINCT**.

SELECT DISTINCT Filière	
FROM classe;	
ST	
SM	

- Il suffit de sélectionner les classes de SM:

SELECT * FROM classe			
WHERE Filière='SM';			
id	Filière	Numéro	Prof.
3	SM	1	Bechikh
4	SM	2	Garoui

## Exercice – Corrigé (2/2)

3. Les prénoms des élèves des classes 1 et 3 :

```
SELECT Prénom
FROM élève
WHERE Classe=1 OR
Classe=3;
```

Ali
Asma
Sahbi
Walid
Amine

4. les noms et les notes des élèves ayant eu une note inférieure à 10:

```
SELECT Nom, Note
FROM élève
WHERE Note<10;
```

Nom	Note
Sassi	8.5
Béji	9.5

## Rappel sur les chemin « os »

❑ Pour afficher le nom du répertoire courant (`getcwd` = get current working directory).

```
1. >>> import os
2. >>> os.getcwd()
3. 'c:\\Python32'
```

❑ Pour afficher le contenu du répertoire courant:

```
1. >>> os.listdir()
2. 'DLLs', 'Doc', 'ghali', 'ghali_bd.py', 'include',
   'Lib',...
```

❑ Pour créer un nouveau répertoire:

```
1. >>> os.mkdir("Lab")
```

❑ Pour changer le répertoire courant:

```
1. >>> os.chdir('c:/Python32/Lab/')
```

❑ Pour supprimer un fichier/répertoire: `os.remove()`/`os.rmdir()`

## Le module sqlite3 de python (1/2)

❑ Nous devons tout d'abord **importer le module sqlite3** (le 3 indiquant la version de sqlite.)

```
1. >>> import sqlite3
```

❑ Le paquet « sqlite3 » contient la méthode « `sqlite3.connect` » qui offre tous les services de connections à une BD SQLite.

```
1. >>> db_loc = sqlite3.connect('Notes.db')
2. #créer ou ouvre un fichier nommé 'Notes.db' qui
3. # doit donc se trouver dans le même répertoire que
4. # l'endroit où Python est lancé sinon :
5. >>> db_far = sqlite3.connect('/path/Notes.db')
6. #ici deux bases sont créer, car aucune des deux
7. # n'existe
```

## Le module sqlite3 de python (2/2)

```
1. >>> db_ram = sqlite3.connect(':memory:')
```

❑ Cette commande permet de créer une **base de donnée en RAM**

➤ Aucun fichier ne sera créer, il s'agit de créer **une base de données virtuelle**,

➤ Cela **permet de vérifier des commandes** avant de les lancer sur BD où les opérations sont irréversibles!

❑ **Attention** : à la fin, il est indispensable de fermer cette connexion.

```
1. >>> db_loc.close()
```

```
2. >>> db_ram.close()
```

## Création des tables dans BD (1/2)

- ❑ Lorsqu'une table est créée, on définit en même temps le type et le nom de chaque champs.

```
1. >>> cr = db_loc.cursor()
2. #cursor est un objet auquel on passe les
3. # commandes SQL en vue d'être exécutées.
4. >>> cr.execute('''CREATE TABLE eleve(
5.     id INTEGER PRIMARY KEY,
6.     nom TEXT,
7.     prenom TEXT,
8.     classe TEXT);''')
9. <sqlite3.Cursor object at 0x02872920>
10. # execute ne lance pas la commande
11. #il est important de noter le champ 'id' de type
12. # nombre entier qui sert de clef primaire
```

## Création des tables dans BD (2/2)

- ❑ La fonction « commit » permet de lancer la commande,
  - C'est-à-dire qu'elle permet de créer véritablement la table 'eleve' et de la sauvegarder dans la BD 'Notes.db'.
  - Attention à ne jamais oublier de 'commit' aux commandes car sinon rien ne se passera.

```
1. >>> db_loc.commit()
2. #Attention : la commande 'commit' est une
3. # fonction de 'db_loc' et non de 'cursor'.
```

- ❑ Pour supprimer la table il suffit de faire :

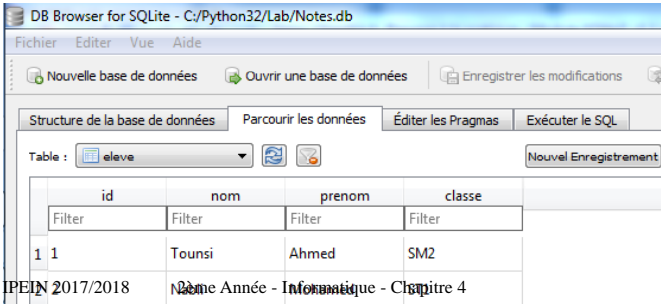
```
1. >>> cr.execute('DROP TABLE eleve;')
2. >>> db_loc.commit()
```

## Insérer des données dans la table (1/4)

```
1. >>> cr.execute('''INSERT INTO eleve VALUES(1,
2.     'Tounsi', 'Ahmed', 'SM2');''')
3. <sqlite3.Cursor object at 0x02872920>
4. >>> db_loc.commit()
❑ Chaque ajout de données passera par le mot clef SQL "INSERT INTO" suivi du nom de la table en question.
❑ Puis le mot clef "VALUES" suivi des valeurs à ajouter en vérifiant leurs types.
1. >>> cr.execute('''INSERT INTO eleve VALUES(toto,
2.     'Nabli', 'Mohamed', 'ST1');''')
3. Traceback (most recent call last):
4.   File "<pyshell#31>", line 1, in <module>
5.     cursor.execute('''INSERT INTO eleve VALUES(toto,
6.     'Nabli', 'Mohamed', 'ST1');''')
7. sqlite3.OperationalError: no such column: toto
```

## Insérer des données dans la table (2/4)

- ❑ Maintenant pour ajouter des données de manière plus subtil :
- ```
1. >>> nom = 'Nabli'
2. >>> prenom = 'Mohamed'
3. >>> classe = 'ST1'
4. >>> cr.execute('''INSERT INTO eleve
5.     VALUES(?,?,?,?);''', (2, nom, prenom, classe))
6. <sqlite3.Cursor object at 0x02872920>
7. >>> db_loc.commit()
```



The screenshot shows the 'DB Browser for SQLite' application. The 'Table' dropdown is set to 'eleve'. The table structure is displayed with columns: id, nom, prenom, and classe. A single record is visible with id=1, nom=Tounsi, prenom=Ahmed, and classe=SM2.

|   | id | nom    | prenom | classe |
|---|----|--------|--------|--------|
| 1 | 1  | Tounsi | Ahmed  | SM2    |

## Insérer des données dans la table (3/4)

❑ Maintenant pour insérer plus rapide des données:

➤ Avec une liste de tuples:

```
1. >>> L_eleves = [(3, 'Essassi', 'Nawres', 'SM5'),
2.               (4, 'Guedidi', 'Ahmed', 'SM5'),
3.               (5, 'Ben Jemaa', 'Azmi', 'ST5'),
4.               (6, 'Cheikh', 'Taieb', 'ST2')]
5. >>> cr.executemany('INSERT INTO eleve VALUES
6.                  (?, ?, ?, ?);', L_eleves)
7. <sqlite3.Cursor object at 0x02872920>
```

➤ Avec un dictionnaire:

```
7. >>> cr.execute('INSERT INTO eleve VALUES
8.               (:id, :nom, :prenom, :classe);',
9.               {'id': 7, 'nom': 'Habbachi', 'prenom': 'Marwen', 'classe': 'ST5'})
10. <sqlite3.Cursor object at 0x02872920>
```

## Insérer des données dans la table (4/4)

❑ Si je veux ajouter une personne et j'ai perdu le fil du compte des id :

```
1. >>> last_id = cr.lastrowid
2. >>> last_id
3. 7
```

❑ Evidemment ceci est très utile pour ajouter automatiquement des nouvelles élèves:

```
1. >>> cr.execute('INSERT INTO eleve VALUES
2.               (?, ?, ?, ?);', ((cr.lastrowid + 1), 'Aloui',
3.               'Roua', 'SM5'))
4. <sqlite3.Cursor object at 0x02872920>
5. >>> db_loc.commit()
```

❑ Pour annulation des dernières modifications (avant commit()) :

```
1. >>> db_loc.rollback()
2. >>> db_loc.commit()
```

## Accéder aux données (1/3)

❑ Pour afficher le résultats d'une requête SQL:

- Fetchone() : résultat récupéré en une ligne unique (tuple)
- Fetchmany(n) : récupère les « n » prochains résultats (liste)
- Fetchall() : récupère toutes les lignes de résultat de la requête (liste)

❑ Pour sélectionner des données, il faut utiliser la commande SQL "SELECT":

```
1. >>> cr.execute('SELECT * FROM eleve;')
2. >>> premier_eleve = cr.fetchone()
3. >>> premier_eleve #est un tuple
4. (1, 'Tounsi', 'Ahmed', 'SM2')
5. print("Le premier eleve s'appelle : %s %s de la
6.       classe %s" % (premier_eleve[2], premier_eleve[1],
7.       premier_eleve[3]))
8. Le premier eleve s'appelle : Ahmed Tounsi de la
9.       classe SM2
```

## Accéder aux données (2/3)

❑ Affichage de tous les élèves :

```
1. >>> cr.execute('SELECT * FROM eleve;')
2. <sqlite3.Cursor object at 0x02977560>
3. >>> eleves = cr.fetchall()
4. #eleves: est une liste des tuples
5. >>> for e in eleves:
6.     print("%s: %s \t %s \t %s" % (e[0], e[1],
7.     e[2], e[3]))
8. 1: Tounsi          Ahmed          SM2
9. 2: Nabli           Mohamed         ST1
10. 3: Essassi         Nawres         SM5
11. 4: Guedidi         Ahmed          SM5
12. 5: Ben Jemaa       Azmi           ST5
13. 6: Cheikh          Taieb          ST2
14. 7: Habbachi        Marwen         ST5
15. 8: Aloui           Roua           SM5
```

## Accéder aux données (3/3)

- ❑ Pour sélectionner des données il faut impérativement la **commande SQL "SELECT"** suivi du ou des champs souhaités.
  - L'étoile correspondant à l'ensemble des champs.
  - Ensuite il faut obligatoirement le **mot clef "FROM"** suivi du nom de la table,
- ❑ Si l'on veut que les « nom » et « classe », il s'agit d'un filtrage (**projection**) sur ces champs :
  1. >>> eleves = cr.execute('SELECT nom, classe FROM eleve;')
  2. >>> for e in eleves:
  3.     print(e)
  4. ('Tounsi', 'SM2')
  5. ('Nabli', 'ST1')
  6. ('Essassi', 'SM5')
  7. ...

## Mise à jour des enregistrements (1/2)

- ❑ Le mot clef que nous allons utiliser est **"UPDATE"**:
  1. >>> cr.execute('INSERT INTO eleve VALUES(?,?,?,?);', (9, "Karma", "Wael", "ST1"))
  2. <sqlite3.Cursor object at 0x02977560>
- ❑ >>> cr.execute ('UPDATE eleve SET nom=?, prenom=? WHERE id=?;', ("lajjem", "Haythem", 8))
- ❑ <sqlite3.Cursor object at 0x02977560>
- ❑ >>> db\_loc.commit()

|   |   |       |      |     |
|---|---|-------|------|-----|
| 8 | 8 | Aloui | Roua | SM5 |
| 9 | 9 | Karma | Wael | ST1 |

## Mise à jour des enregistrements (2/2)

- ❑ Donc, la mise à jour est assez classique :
  - Juste après le mot clef **"UPDATE"** on défini la table à mettre à jour.
  - Ensuite vient le mot clef **"SET"** après lequel on défini les champs à mettre à jour,
  - Puis le mot clef **"WHERE"** permettant de dire quel enregistrement doit être mis à jour.
- ❑ **UPATE relation SET attributs WHERE condition**
- ❑ Recherche des données dans la BD (**sélection + projection**):
  1. >>> cr.execute('SELECT nom, prenom, classe FROM eleve WHERE id==8;')
  2. <sqlite3.Cursor object at 0x02977560>
  3. >>> eleve = cr.fetchone()
  4. >>> print(eleve)
  5. ('lajjem', 'Haythem', 'SM5')

## Suppression des enregistrements (1/2)

- ❑ Le mot clef que nous allons utiliser est **"DELETE"**:
  1. >>> cr.execute('DELETE FROM eleve WHERE id=8;')
  2. <sqlite3.Cursor object at 0x02977560>
- ❑ >>> cr.execute('SELECT prenom, nom, classe FROM eleve WHERE id=8;')
- ❑ <sqlite3.Cursor object at 0x02977560>
- ❑ >>> eleve = cr.fetchone()
- ❑ >>> print(eleve)
- ❑ None

|   |          |        |     |
|---|----------|--------|-----|
| 6 | Cheikh   | Taieb  | ST2 |
| 7 | Habbachi | Marwen | ST5 |
| 9 | Karma    | Wael   | ST1 |



## Suppression des enregistrements (2/2)

❑ Suppression des tous les enregistrement!

```
1. >>> cr.execute('DELETE FROM eleve;')
2. <sqlite3.Cursor object at 0x02977560>
3. >>> eleve = cr.fetchone()
4. >>> print(eleve)
5. None
```

❑ Voilà ! Le contenu de la table a été entièrement supprimé !

❑ Supprimons alors la table avec la commande suivante !

```
1. >>> cr.execute('DROP TABLE eleve;')
2. <sqlite3.Cursor object at 0x02977560>
3. >>> db_loc.commit()
4. >>> db_loc.close()
5. >>> db_ram.close()
```

## Exercice (1/12)

❑ Voici la table « Notes » suivante:

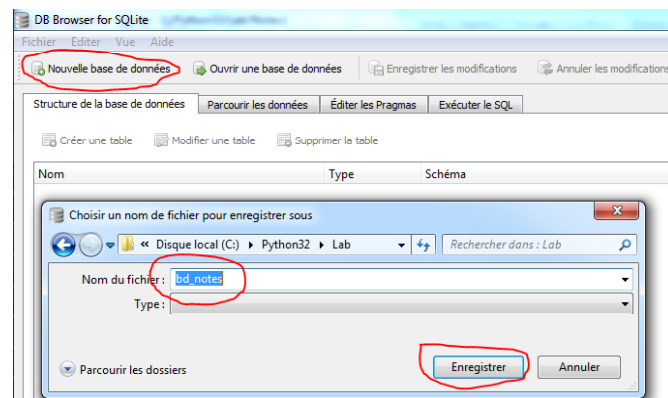
| id | Nom       | Prenom  | Filiere | Classe | Sem1 | Sem2 |
|----|-----------|---------|---------|--------|------|------|
| 1  | Tounsi    | Ahmed   | SM      | 2      | 8    | 7,5  |
| 2  | Nabli     | Mohamed | ST      | 1      | 6,5  | 9    |
| 3  | Essassi   | Nawres  | SM      | 5      | 10   | 15,6 |
| 4  | Guedidi   | Ahmed   | SM      | 5      | 15   | 12   |
| 5  | Ben Jemaa | Azmi    | ST      | 5      | 12,5 | 16   |
| 6  | Cheikh    | Taieb   | ST      | 2      | 11,5 | 12   |
| 7  | Habbachi  | Marwen  | ST      | 5      | 16,5 | 11   |
| 8  | Karma     | Wael    | ST      | 1      | 15   | 10   |
| 9  | Salhi     | Mohamed | SM      | 2      | 9,5  | 8    |

1. Déterminer les notes des classes SM au semestre 2.
2. Trouver les nom et prénom des étudiants de ST qui ont la moyenne au semestre 1.
3. Calculer les moyennes du semestre 2 par filière.
4. Trouver les étudiants (Nom et prénom) qui ont plus que la moyenne de leur filière au semestre 2.

## Exercice (2/12)

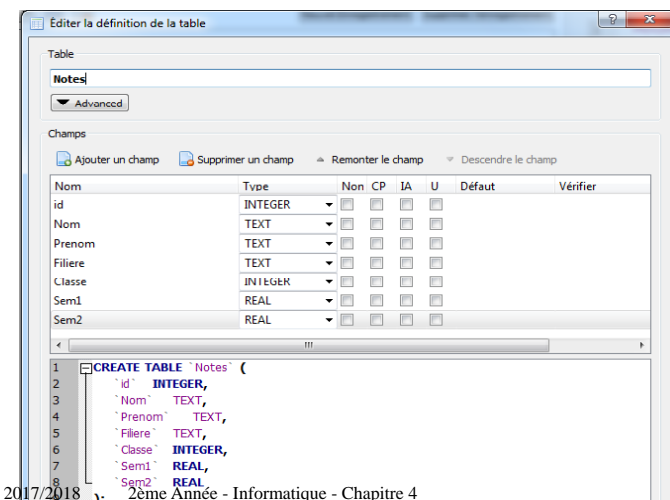
❑ Utilisation de l'outil « DB Browser for SQLite » pour la création de la Base de Données « bd\_notes.db »

➤ Cet outil est disponible à l'adresse: <http://sqlitebrowser.org/>



## Exercice (3/12)

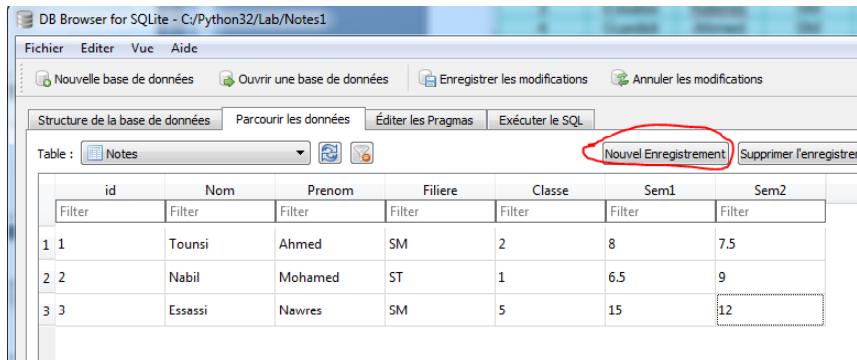
❑ Création de la table « Notes » avec l'outil « DB Browser for SQLite »





## Exercice (4/12)

- ❑ Ajouter le trois premiers enregistrement de la table « Notes » en utilisant l'outil « DB Browser for SQLite »



## Exercice (5/12)

- ❑ Affichage des enregistrements de « Notes » à travers Python:

```
1. >>> db_loc = sqlite3.connect("bd_notes.db")
2. >>> cr = db_loc.cursor()
3. >>> cr.execute('SELECT * FROM Notes;')
4. <sqlite3.Cursor object at 0x029778A0>
5. >>> res = cr.fetchall()
6. >>> print(res)
7. [(1, 'Tounsi', 'Ahmed', 'SM', 2, 8.0, 7.5),
    (2, 'Nabil', 'Mohamed', 'ST', 1, 6.5, 9.0),
    (3, 'Essassi', 'Nawres', 'SM', 5, 15.0, 12.0)]
```

## Exercice (6/12)

- ❑ Ajout de reste des enregistrements à travers Python:

```
1. >>> cr.execute('INSERT INTO Notes
VALUES(?,?,?, ?, ?, ?);', (4, "Guedidi", "Ahmed",
"SM", 5, 15, 12))
2. <sqlite3.Cursor object at 0x029778A0>
3. >>> liste=[(5, "Ben Jemaa", "Azmi", "ST", 5, 12.5, 16),
4. (6, "Cheikh", "Taieb", "ST", 2, 11.5, 12),
5. (7, "Habbachi", "Marwen", "ST", 5, 16.5, 11),
6. (8, "Karma", "Wael", "ST", 1, 15, 10),
7. (9, "Salhi", "Mohamed", "SM", 2, 9.5, 8)]
8. >>> cr.executemany('INSERT INTO Notes
VALUES(?,?, ?, ?, ?, ?);', liste)
9. <sqlite3.Cursor object at 0x029778A0>
10. >>> db_loc.commit()
```

## Exercice (7/12)

1. Les notes des SM au semestre 2 (c'est une sélection + projection):

- En algèbre relationnelle:  $\pi_{\text{sem2}}(\sigma_{\text{Filiere}='SM'}(\text{Notes}))$
- Requête SQL:

```
SELECT Sem2 FROM Notes WHERE Filiere = 'SM';
```

- Code en SQLite3 sous Python:

```
1. >>> cr.execute('SELECT Sem2 FROM Notes WHERE
Filiere = "SM";')
2. <sqlite3.Cursor object at 0x029778A0>
3. >>> res = cr.fetchall()
4. >>> for i in res:
5.     print(i)
6. (7.5)
7. (15.6)
8. (12.0)
9. (8.0)
```

## Exercice (8/12)

2. Les noms et prénoms des étudiants de ST qui ont la moyenne au semestre 1 (sélection + projection):

- En algèbre relationnelle:  $\pi_{Nom, Prenom}(\sigma_{Filiere='ST' \text{ ET } Sem1 \geq 10}(Notes))$
  - Requête SQL: `SELECT Nom, Prenom FROM Notes WHERE Filiere = 'ST' AND Sem1 >= 10;`
  - Code en SQLite3 sous Python:
1. `>>> cr.execute('SELECT Nom, Prenom FROM Notes WHERE Filiere = "ST" AND Sem1 >= 10;')`
  2. `<sqlite3.Cursor object at 0x029778A0>`
  3. `>>> res = cr.fetchall()`
  4. `>>> for i in res: print(i)`
  5. `('Ben Jemaa', 'Azmi')`
  6. `('Cheikh', 'Taieb')`
  7. `('Habbachi', 'Marwen')`
  8. `('Karma', 'Wael')`

## Exercice (9/12)

3. Les moyennes du semestre 2 par filière (c'est une Agrégation):

- En algèbre relationnelle:  $Filiere \bowtie Moyenne(Sem2)(Notes)$
- Requête SQL:

`SELECT Filiere, AVG(Sem2) FROM Notes GROUP BY Filiere;`

- Code en SQLite3 sous Python:

1. `>>> cr.execute('SELECT Filiere, AVG(Sem2) FROM Notes GROUP BY Filiere;')`
2. `<sqlite3.Cursor object at 0x029778A0>`
3. `>>> res = cr.fetchall()`
4. `>>> for i in res:`
5. `print(i)`
6. `('SM', 10.775)`
7. `('ST', 11.6)`

## Exercice (10/12)

4. Les étudiants (Nom et prénom) qui ont plus que la moyenne de leur filière au semestre 2:

- Expression en algèbre relationnelle:

$R1 = Filiere \bowtie Moyenne(Sem2)(Notes)$   
 $\rho_{Moyenne(Sem2) < MoyFiliereSem2}(R1)$   
 $R2 = Notes[Notes.Filiere = R1.Filiere]R1$   
 $R3 = \sigma_{Sem2 > Moyenne(Sem2)}(R2)$   
 $\pi_{Nom, Prenom}(R3)$

Les étudiants (Nom et prénom) qui ont plus que la moyenne de leur filière au semestre 2: c'est une imbrication des requêtes

- Requête SQL:

`SELECT Nom, Prenom FROM`  
`(SELECT * FROM Notes`  
`JOIN (SELECT Filiere, AVG(Sem2) AS MoyFiliereSem2`  
`FROM Notes GROUP BY Filiere) AS R1`  
`ON Notes.Filiere = R1.Filiere)`  
`WHERE Sem2 >= MoyFiliereSem2;`

## Exercice (11/12)

4. Notez les renommages de tables calculées par SELECT et de champs calculés par agrégation, à l'aide de AS :

- `AVG(Sem2) AS MoyFiliereSem2`
- `SELECT Filiere, AVG(Sem2) AS MoyFiliereSem2`  
`FROM Notes GROUP BY Filiere) AS R1`

□ Ainsi que la notation de champs de même nom dans 2 tables différentes: `Notes.Filiere = R1.Filiere`

```
SELECT * FROM (
    SELECT * FROM notes
    JOIN (SELECT Filiere, AVG(Sem2) AS Moy2 FROM notes GROUP BY Filiere) AS R1
    ON R1.Filiere=notes.Filiere) WHERE Sem2>=Moy2;
```

|   | id | Nom       | Prenom | Filiere | classe | Sem1 | Sem2 | Filiere:1 | Moy2   |
|---|----|-----------|--------|---------|--------|------|------|-----------|--------|
| 1 | 3  | Essassi   | Nawres | SM      | 5      | 10.0 | 15.6 | SM        | 10.775 |
| 2 | 4  | Guedidi   | Ahmed  | SM      | 5      | 15.0 | 12.0 | SM        | 10.775 |
| 3 | 5  | Ben Jemaa | Azmi   | ST      | 5      | 12.5 | 16.0 | ST        | 11.6   |
| 4 | 6  | Cheikh    | Taieb  | ST      | 2      | 11.5 | 12.0 | ST        | 11.6   |

## Exercice (12/12)

### ➤ Code en SQLite3 sous Python:

```
1. >>> cr.execute('''
2. SELECT Nom, Prenom FROM (
3.   SELECT * FROM Notes
4.   JOIN (SELECT Filiere,AVG(Sem2) AS MoyFiliereSem2
5.         FROM Notes GROUP BY Filiere) AS R1
6.   ON Notes.Filiere = R1.Filiere)
7. WHERE Sem2 >= MoyFiliereSem2;''')
8. <sqlite3.Cursor object at 0x029778A0>
9. >>> res = cr.fetchall()
10. >>> for i in res:
11.     print(i)
12. ('Essassi', 'Nawres')
13. ('Guedidi', 'Ahmed')
14. ('Ben Jemaa', 'Azmi')
15. ('Cheikh', 'Taieb')
```

## Condition SQL « HAVING » (1/2)

La condition **HAVING** en SQL est presque similaire à **WHERE** à la seule différence que **HAVING** permet de filtrer en utilisant des fonctions d'agrégations telles que **SUM()**, **COUNT()**, **AVG()**, **MIN()** ou **MAX()**.

### ❑ Syntaxe:

```
SELECT colonne1, SUM(colonne2)
```

```
FROM nom_table
```

```
GROUP BY colonne1
```

```
HAVING fonction(colonne2) operateur valeur
```

❑ **Remarque:** **HAVING** est très souvent utilisé en même temps que **GROUP BY** bien que ce ne soit pas obligatoire.

## Condition SQL « HAVING » (2/2)

❑ **Exemple:** Soit une table « achat » qui contient les achats de différents clients avec le coût du panier pour chaque achat.

| id | client | tarif | date_achat |
|----|--------|-------|------------|
| 1  | Pierre | 102   | 2017-10-23 |
| 2  | Simon  | 47    | 2017-10-27 |
| 3  | Marie  | 18    | 2017-11-05 |
| 4  | Marie  | 20    | 2017-11-14 |
| 5  | Pierre | 160   | 2017-12-03 |

➤ Si on souhaite récupérer la liste des clients qui ont commandé plus de 40DT, toute commandes confondu:

```
SELECT client, SUM(tarif)
FROM achat
GROUP BY client
HAVING SUM(tarif) > 40;
```

### Résultat :

| client | SUM(tarif) |
|--------|------------|
| Pierre | 262        |
| Simon  | 47         |

## Résumé : Commandes SQL (1/8)

### 1. Obtention des données:

```
SELECT <liste des noms de colonnes> FROM <liste des noms de tables>;
```

```
SELECT * FROM tab1; /*(toutes les colonnes)*/
```

```
SELECT col1,col3 FROM tab1; /*(une partie des colonnes)*/
```

```
SELECT DISTINCT col1 FROM tab1; /*(élimine les doublons)*/
```

```
SELECT nom AS "nom personne" FROM tab1; /*(renommage des colonnes)*/
```

```
SELECT nom "nom personne" FROM tab1;
```

## Résumé : Commandes SQL (2/8)

### 2. Expression des restrictions:

```
SELECT * FROM tab1 WHERE <condition(s)>;
SELECT * FROM tab1 WHERE ville IN ('Brest','Rennes','Paris');
SELECT * FROM tab1 WHERE age NOT BETWEEN 15 AND 20;
SELECT * FROM tab1 WHERE adresse IS NULL;
SELECT * FROM tab1 WHERE adresse IS NOT NULL;
SELECT * FROM tab1 WHERE ville LIKE 'Gre%'; --commence par Gre
SELECT * FROM tab1 WHERE ville LIKE '%reno%'; --contient reno
```

Avec:

|                 |                                           |
|-----------------|-------------------------------------------|
| > >= < <= = <>  | : comparateur arithmétiques               |
| AND OR NOT      | : comparateur logique                     |
| %               | : n'importe quelle séquence de caractères |
| _ (barre basse) | : n'importe quel caractère                |

## Résumé: Commandes SQL (3/8)

### 3. Tri et présentation des résultats:

```
SELECT * FROM tab1 ORDER BY col1; --tri ascendant par défaut
SELECT * FROM tab1 ORDER BY col5,col7; /*(tri par col5 puis
tri par col7)*/
SELECT * FROM tab1 ORDER BY age ASC, sexe DESC; /*(tri
ascendant ou descendant)*/
SELECT * FROM tab1 ORDER BY age LIMIT 0,10; /* Les 10
premiers résultats*/
SELECT * FROM tab1 ORDER BY age LIMIT 10,5; /* Les 5
suivants*/
```

## Résumé: Commandes SQL (4/8)

### 4. Expression des jointures:

```
SELECT * FROM tab1,tab2; /*(jointure sans qualification =
produit cartésien)*/
SELECT * FROM tab1 JOIN tab2 ON tab1.col1=tab2.col2;
/*(jointure avec égalité = équijointure)*/
SELECT * FROM tab1,tab2 WHERE tab1.col1=tab2.col2; /*(
Même résultat : jointure avec égalité = équijointure)*/
SELECT * FROM tab1 t1,tab2 t2,tab3 t3 WHERE
t1.col1=t2.col2 AND t2.col2=t3.col3; /*(jointures en
cascades)*/
```

## Résumé: Commandes SQL (5/8)

### 5. Regroupements:

```
SELECT * FROM tab1 GROUP BY col1;
```

### 6. Les fonctions statistiques (Agrégations):

|       |                     |
|-------|---------------------|
| AVG   | : moyenne           |
| COUNT | : nombre d'éléments |
| MAX   | : maximum           |
| MIN   | : minimum           |
| SUM   | : somme             |

```
SELECT COUNT(*) FROM tab1;
```

```
SELECT SUM(col1) FROM tab2;
```

## Résumé: Commandes SQL (6/8)

### 7. Sous-requêtes SQL:

```
SELECT * FROM tab1 WHERE prix > (SELECT MIN(prix) FROM tab2);
SELECT * FROM tab1 WHERE nom NOT IN (SELECT nom FROM tab2);
SELECT * FROM tab1 WHERE prix > ALL (SELECT prix FROM tab2);
/* > ALL => sup. à toutes les valeurs */
SELECT * FROM tab1 WHERE prix > ANY (SELECT prix FROM tab2);
/* > ANY sup. à au moins 1*/
```

### 8. Opérateurs ensemblistes:

```
(SELECT * FROM tab1) UNION (SELECT * FROM tab2);
(SELECT * FROM tab1) INTERSECT (SELECT * FROM tab2);
(SELECT * FROM tab1) EXCEPT (SELECT * FROM tab2);
```

## Résumé: Commandes SQL (7/8)

### 9. Insertions:

```
INSERT INTO tab1 VALUES ('abc',5,7); /*toutes les
valeurs doivent être renseignées*/
INSERT INTO tab1(col1,col3) VALUES ('xyz',7); /*on
ne renseigne que les colonnes indiquées, les cols
non précisées sont mises à NULL*/
INSERT INTO tab1 SELECT * FROM tab2; /*copier le
contenu de tab2 dans tab1*/
```

### 10. Mises à jour:

```
UPDATE tab1 SET col7='abc' WHERE col1=1;
```

## Résumé: Commandes SQL (8/8)

### 11. Suppressions:

```
DELETE FROM tab1 WHERE col1=1; --suppression des lignes
DELETE FROM tab1 WHERE col1 IN (SELECT num FROM tab2);
DELETE FROM tab1; --vide complètement la table
```