Bases de données en Python

Benoît Petitpas : benoit.petitpas@telecom-paristech.fr

Introduction

Dans ce cours nous allons traiter des différentes manières d'agir sur des bases de données avec Python.

Il existe de nombreux logiciels de gestion de bases de données relationnelle sur le marché comme postgresql, mysql, sqlite ... une base de données relationnelle se gère via du SQL, le langage de requête de bases de données.

Nous allons donc dans ce cours étudier cette gestion de bases de données, mais au lieu d'utiliser des SGBD (système de gestion de bases de données), nous allons préférer gérer les bases via Python.

De nombreuses bibliothèques de fonctions ont été développées pour interfacer du code python avec les différents SGBD. Ainsi par exemple psycopg est un paquet Python permettant de gérer une base de données Postgresql en Python.

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 multi-plateforme.

Sqlite est souvent le moyen le plus rapide et le plus simple de gérer une base de données, et comme Python aime la simplicité il existe un paquet standard Python pour faire l'interface entre Python et sqlite.

La base de donnée est une notion des années 60. Mais le modèle relationnel date de 70 et les SGBD de 80. Les bases de données relationnelles sont basées sur la théorie des ensembles avec l'utilisation des opérateurs de l’algèbre relationnel (l'union, la différence, produit cartésien, la projection, la sélection, le quotient et la jointure)

Base de données :

C'est un ensemble d'informations structurées persistant. Autres caractéristiques : Exhaustivité (toutes les infos requises pour le service que l'on attend) et Unicité (pas de doublons)

SGBD :

Qu'est-ce que c'est ? C'est un logiciel qui gère une BDD Autre définition plus précise : Un logiciel permettant de décrire et d'interagir avec un ensemble d'informations structurées pour satisfaire les besoins de plusieurs utilisateurs (en même temps éventuellement).

Table :

C'est une entité (semblable à un tableau) regroupant des caractéristiques exhaustives (sur un sujet donné). Les informations sont rangés dans la table Toutes les données d'une même colonne sont du même type.

Vocabulaire :

TABLE composée de CHAMPS (ou ATTRIBUTS/RUBRIQUES) – C'est simple, il n'y a qu'un type d'objet principal : LA TABLE Une table est la structure enveloppante dans laquelle seront stockées les données au format spécifié. CHAMP : NOM_CHAMP TYPE [LONGUEUR]

Problèmes rencontrés : Homonymie, Redondance (contre le principe d'unicité) , erreurs de saisie

Les services d'un SGBD :

Toujours un seul type d'objet : la table

Les liens sont logiques : ils n'ont pas besoins d'être décrits. Il suffit de les avoir pensés et traduits avec des clés externes.

1. Persistance : Disque Dur
2. Gestion de disque : transparente (pas util pour vous)
3. Partage des données : accès simultané à la même donnée (Contrôle de concurrence)
4. Fiabilité des données : reprise sur panne (pas util pour vous)
5. Sécurité des données : accès aux données protégé et sélectif
6. Interrogation ad hoc : Langage de requête

Les différents modèles :

Hiérarchique : arborescences. Réseau : Plus de connexions entre les éléments (prédéfinies) -> Plus d'interrogations possibles. Relationnel : Le plus utilisé. Orienté objet : CF savoir plus Relationnel étendu : Types complexes, index spatiaux….

Vocabulaire : Un élément d'une relation est appelé ENREGISTREMENT (ou RECORD, TUPLE, N-UPLET)

Langage d'interrogation : QUEL, QBE, SQL (normalisé).

Utilisons le module sqlite de python

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

In [3]:
```
import sqlite3 # nous importons le paquet Python capable d'interagir avec sqlite
```

Se connecter à la base de données

Le paquet sqlite3 contient la méthode `sqlite3.connect` qui offre tous les services de connections une base de données Sqlite

In [5]:
```
db_loc = sqlite3.connect('Notes.db')
# créer ou ouvre un fichier nommé 'Notes.db' qui doit donc se trouver dans le même
# répertoire que l'endroit où Python est lancé sinon :

#db_far = sqlite3.connect('/path/to/the/good/folder/Notes.db')
# ici deux bases sont créer, car aucune des deux n'éxiste

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 cérer une base de données virtuelle).
# Cela permet notamment de vérifier des commandes avant de les lancer sur une base
# de données où les opérations sont irréversibles.
```

Attention : n'oubliez pas que vous ouvrez une connection avec un fichier, il est donc indispensable de fermer cette connection à la fin.

In [6]:
```
db_loc.close()
db_ram.close()
```

Créer et supprimer des tables

Maintenant que la base est créée, il faut en construire l'ossature, c'est-à-dire les tables. Chaque table doit correspondre à une entité réel, par exemple la table 'élèves' correspond à toutes les infirmations nécessaires à la définition d'un élève. Lorqu'une table est défini, on définit en même temps le type et le nom de chaque champs, c'est-à-dire on définit chaque élément décrivant l'entité. Par exemple, lors de la création de la table élève, il faudra définir le champs 'nom' avec son type.

In [7]:
```
db_loc = sqlite3.connect('Notes.db')
cursor = db_loc.cursor()
# cursor est un object auquel on passe les commandes SQL en vue d'être executées.
```

In [8]:
```
cursor.execute('''CREATE TABLE eleve(
            id INTEGER PRIMARY KEY,
            name TEXT,
            first_name TEXT,
            classe TEXT);''')
# execute ne lance pas la commande
# il est important de noter le champ 'id' de type nombre entier qui sert de clef primaire
```

Out[8]:
```
<sqlite3.Cursor at 0x3069490>
```

La clef primaire d'une table dans une base de données relationnelle sert d'identifiant unique à une entité. Par exemple, la clef primaire sert à connaitre un unique élève en particulier. Le nom est une très mauvaise clef primaire car un même nom peut être partagé par plus d'un élève. Par habitude, un champ 'id' est souvent créer pour identifier chaque enregistrement de la table.

In [9]:
```
db_loc.commit()
```

la fonction 'commit' permet de 'lancer' la commande, c'est-à-dire que le commit permet de créer véritablement la table 'eleve' et de la sauvegarder dans 'Notes.db'. Attention à ne jamais oublier de 'commit' vos commandes car sinon rien ne se passera.

In [7]:
```
# Pour supprimer la table il suffit de :
# cursor.execute('''DROP TABLE eleve;''')
# db_loc.commit()
```

Attention : la commande 'commit' est une fonction de 'bd_loc' et non de 'cursor'.

Insérer des données dans la table

In [10]:
```
cursor.execute('''INSERT INTO eleve VALUES (1, 'Petitpas', 'Benoit', '4ieme3');''')
```

Out[10]:
```
<sqlite3.Cursor at 0x3069490>
```

Chaque ajout de données passera par le mot clef SQL "INSERT INTO" suivi du nom de la table dans laquelle vous voulez ajouter des données. Suit alors le mot clef "VALUES" suivi des valeurs à ajouter en vérifiant que le type est le bon.

In [11]:
```
cursor.execute('''INSERT INTO eleve VALUES ("TOTO", "BLIP", "blop", "wizzz");''')
```

```
---------------------------------------------------------------------------
IntegrityError                            Traceback (most recent call last)
<ipython-input-11-a4ff19d72764> in <module>()
----> 1 cursor.execute('''INSERT INTO eleve VALUES ("TOTO", "BLIP", "blop", "wizzz");''')

IntegrityError: datatype mismatch
```

In [12]:
```
db_loc.commit()
```

Un élève a été ajouté à la table. Maintenant comment faire pour ajouter des données de manière plus subtil :

In [13]:
```
name = "Pignon"
first_name = "Francois"
classe = "4ieme3"

cursor.execute('''INSERT INTO eleve VALUES (?,?,?,?);''', (2, name, first_name, classe))
# pourquoi cette formulation avec '?' plutôt que %s, à cause de l'injection
# de SQL qui est une attaque "classique" des hackers see : http://xkcd.com/327/
```

Out[13]:
```
<sqlite3.Cursor at 0x3069490>
```

Remarques : le prénom est francois au lieu de françois, car le c cédille n'est pas standard en sqlite, il aurait fallu précisé à la création de la base le type utf8

Maintenant comment faire des insert plus rapide ?

In [14]:
```
eleves = [(3, "Hallyday", "Johnny", "5iemeB"), (4, "Mitchelle", "Eddy", "4ieme3"), (5, "Rivers", "
Dick", "3ieme2")]
# liste de tuples contenant toutes les informations nécessaires pour ajouter les élèves à la base
de données
cursor.executemany('''INSERT INTO eleve VALUES (?,?,?,?);''', eleves)
```

Out[14]:
```
<sqlite3.Cursor at 0x3069490>
```

Et avec un dictionnaire ?

In [15]:
```
cursor.execute('''INSERT INTO eleve VALUES (:id,:name,:first_name,:classe);''',
               {'id':6,'name':"Cobain", "first_name":"Kurt","classe":"6ieme2"})
```

Out[15]:
```
<sqlite3.Cursor at 0x3069490>
```

Oui mais maintenant je veux ajouter une personne et j'ai perdu le fil du compte des id :

In [16]:
```
last_id = cursor.lastrowid # last_id contient le dernière id ajouté
print('le dernier id est : %s' % last_id)
```

```
le dernier id est : 6
```

Evidemment c'est très util pour ajouter automatiquement un nouvel élève:

In [17]:
```
cursor.execute('''INSERT INTO eleve VALUES (?,?,?,?);''', ((cursor.lastrowid + 1), "Joplin", "Janice", "3ieme5"))
```

Out[17]:
```
<sqlite3.Cursor at 0x3069490>
```

In [18]:
```
db_loc.commit()
```

Accéder aux données

Parce que oui, des données ont été ajoutées mais quelles sont-elles déjà ? Pour selectionner des données, il faut utiliser la commande SQL "SELECT"

In [19]:
```
cursor.execute('''SELECT * FROM eleve;''')
first_eleve = cursor.fetchone() # récupère le premier élève
print(first_eleve)
```

```
(1, u'Petitpas', u'Benoit', u'4ieme3')
```

In [20]:
```
# le nom du premier élève :
print(first_eleve[1])
```

```
Petitpas
```

En effet la réponse est un tuple ! Pour afficher les information concernant le premier élève :

In [21]:
```
print("Le premier eleve s'appelle : %s %s de la classe %s" % (first_eleve[2], first_eleve[1], first_eleve[3]))
# l'indice du tuple commence à 0 mais est réservé à l'id de l'enregistrement
```

```
Le premier eleve s'appelle : Benoit Petitpas de la classe 4ieme3
```

In [22]:
```
# maintenant tous les élèves :
cursor.execute('''SELECT * FROM eleve;''')
eleves = cursor.fetchall()
for eleve in eleves:
    print("L'eleve n %s s'appelle : %s %s de la classe %s" % (eleve[0], eleve[2], eleve[1], eleve[3]))
    # eleve correspond à un enregistrement de la table
    # eleve[indice] correspond au champs correspondant de l'élève en question
```

```
L'eleve n 1 s'appelle : Benoit Petitpas de la classe 4ieme3
L'eleve n 2 s'appelle : Francois Pignon de la classe 4ieme3
L'eleve n 3 s'appelle : Johnny Hallyday de la classe 5iemeB
L'eleve n 4 s'appelle : Eddy Mitchelle de la classe 4ieme3
L'eleve n 5 s'appelle : Dick Rivers de la classe 3ieme2
L'eleve n 6 s'appelle : Kurt Cobain de la classe 6ieme2
L'eleve n 7 s'appelle : Janice Joplin de la classe 3ieme5
```

Maintenant un peu de pratique du SQL :

Pour selectionner des données il faut impérativement le commande SQL "SELECT" suivi du ou des champs souhaités. L'étoile correspondant à l'ensemble des champs. Ensuite il faut obligatoirement le mote clef "FROM" suivi du nom de la table (lui dire d'où doivent provenir les données)

Maintenant il est possible de "filtrer" la selection.

Si l'on veut l'ensemble des nom de famille uniquement, il s'agit d'un filtrage sur les champs :

In [23]:
```
eleves = cursor.execute('''SELECT name FROM eleve;''')
# ici le passage par un argument évite le fetchall() mais les deux commandes sont équivalentes
for eleve in eleves:
    print(eleve)
```

```
(u'Petitpas',)
(u'Pignon',)
(u'Hallyday',)
(u'Mitchelle',)
(u'Rivers',)
(u'Cobain',)
(u'Joplin',)
```

Maintenant je veux seulement les élèves de la 4ieme 3.

Là, la selection se fait sur tous les champs mais "où" la classe est égale à "4ieme3".

Or le SQL ayant tendance à être simpliste le mot clef pour le filtrage des enregistrements est "WHERE":

In [24]:
```
eleves = cursor.execute('''SELECT * FROM eleve WHERE classe = "4ieme3";''')
for eleve in eleves:
    print(eleve)
```

```
(1, u'Petitpas', u'Benoit', u'4ieme3')
(2, u'Pignon', u'Francois', u'4ieme3')
(4, u'Mitchelle', u'Eddy', u'4ieme3')
```

On peut alors mixer les deux :

In [25]:
```
eleves = cursor.execute('''SELECT name, classe FROM eleve WHERE id > 3;''')
for eleve in eleves:
    print("les eleves ayant un id superieur a 3 sont : %s de la classe %s" % (eleve[0], eleve[1]))
    # important, comme je ne sélectionne que le nom et la classe le tuple de
    # sorti n'a que 2 valeurs, ainsi le nom sera à l'indice 0 et la classe à l'indice 1
```

```
les eleves ayant un id superieur a 3 sont : Mitchelle de la classe 4ieme3
les eleves ayant un id superieur a 3 sont : Rivers de la classe 3ieme2
les eleves ayant un id superieur a 3 sont : Cobain de la classe 6ieme2
les eleves ayant un id superieur a 3 sont : Joplin de la classe 3ieme5
```

Et si lors de séléction, je m'aperçois qu'une érreur s'est glissé ?

Mettre à jour et supprimer des enregistrements

Les mots clefs que nous allons utiliser sont "UPDATE" et "DELETE"

In [26]:
```
cursor.execute('''INSERT INTO eleve VALUES (?,?,?,?);''', (8, "andrix", "gimi", "4ieme3"))
```

Out[26]:
```
<sqlite3.Cursor at 0x3069490>
```

In [27]:
```
# Zut c'est Jimmy Hendrix
cursor.execute('''UPDATE eleve SET name = ?, first_name=? WHERE id = ? ''',("Hendrix", "Jimmy", 8)
)
```

Out[27]:
```
<sqlite3.Cursor at 0x3069490>
```

In [28]:
```
# and now :
cursor.execute('''SELECT first_name, name, classe FROM eleve WHERE id == 8;''')
eleve = cursor.fetchone()
print("l'eleve ayant un id egal a 8 est : %s %s de la classe %s"%(eleve[0],eleve[1], eleve[2]))
```

```
l'eleve ayant un id egal a 8 est : Jimmy Hendrix de la classe 4ieme3
```

Ceci est un exemple, cliquez sur le lien de téléchargement pour obtenir le cours complet.