TP4

Grâce à sqlite, nous pouvons travailler sur une base de données en lecture et en écriture. Cette base constitue la source de données pour le programme. Pour ce TP, nous reprenons la base bibliothèque du TP2. Nous allons consulter et modifier les données de cette base dans un programmme Python comme dans le TP3.

- Créez dans Pycharm un nouveau projet TP4.
- Téléchargez biblio.db et copiez-le dans le dossier de votre projet.
- Créez un programme principal contenant le code suivant et exécutez-le:

```
import sqlite3
import os, sys

def connecte_base(db_name):
    try:
        assert os.path.isfile(db_name)
        db = sqlite3.connect(db_name)
        print("Connexion à ", db_name, "OK.")
        return db
    except:
        print("Erreur de connexion : la base n'existe pas!")
        sys.exit()

db = connecte_base("biblio.db")
c = db.cursor()
```

1. Requêtes

Testez la requête permettant d'afficher tous les livres de la base et n'afficher que l'auteur et le titre de chaque livre. Reprenez au choix une des requêtes 8, 9 ou 10 du TP2 et affichez son résultat.

Attention. Pensez à lancer vos requêtes à l'aide d'un `try..except` pour pouvoir lire les erreurs sans interrompre le programme

```
try:
    liste_tuples = c.execute("SELECT ... ").fetchall()
    for t in liste_tuples:
        print(t)
except sqlite3.OperationalError as err:
    print("Erreur SQL :" + err.args[0])
```

2. Classes Membre

Les objets de type `Membre` vont nous servir à stocker les informations contenues dans la table **Membre**.



- Nous définissons ici une classe `Membre` (correspondant à la table **Membre** de la base) dont les attributs correspondent exactement à ceux de la table : `idMembre`, `nomMembre`, `adrMembre` et `cpMembre`.
- Pour prendre en compte les emprunts effectués, nous lui ajoutons un attribut `emprunts` servant à stocker les identifiants des livres empruntés sous la forme d'une liste de chaînes de caractères (initialement vide).

Ajoutez dans votre projet un fichier `membre.py` contenant les définitions suivantes :

```
class Membre:
    def __init__(self, idMembre, nomMembre, adrMembre, cpMembre):
        self.idMembre = idMembre
        self.nomMembre = nomMembre
        self.adrMembre = adrMembre
        self.cpMembre = cpMembre
        self.emprunts = []
    def emprunte(self, idLivre):
        self.emprunts += [idLivre]
```

Vous devez maintenant tester le bon fonctionnement de cette classe:

- Importez la classe `Membre` dans le programme principal (`from membre import Membre`)

Création d'un nouveau membre

- Initialisez un objet `m` de type Membre avec des valeurs quelconques
- Pour vérifier l'initialisation, affichez son contenu en notation pointée:
  - `m.idMembre`
  - `m.nomMembre`
  - `m.adrMembre`
  - `m.cpMembre`
- Faites-lui emprunter le livre dont le code est `"7089PQIU"`, toujours en notation pointée : `m.emprunte(…)`
- Pour vérifier l'emprunt, affichez la liste des livres empruntés :
  - `m.emprunts`

Lecture de la base

- Ecrivez une requête qui extrait les informations sur le membre d'identifiant 15.
- Initialisez un objet `p` de type Membre avec ces données
- Ecrivez une requête qui extrait les identifiants des livres empruntés par le membre d'identifiant 15.
- utilisez la réponse pour compléter la liste `p.emprunts`
- Affichez le contenu de `p`.

3. Classe MembreDAO



La mise en correspondance entre les objets et les tuples d'un tableau de données s'effectue principalement avec une des quatre opérations suivantes:

- **Création** (***C****reate*) : écriture de nouvelles données dans la base
- **Lecture/recherche** (***R****ead*) : lecture du contenu de la base
- **Mise à jour** (***U****pdate*) : changement du contenu existant
- **Suppression** (***D****elete*) : suppression des données

Les accesseurs d'objets (*Data Access Object* – DAO) sont des interfaces permettent d'implémenter ces opérateurs:

- Les DAO d'une classe A sont des objets servant spécifiquement à interfacer les objets de la classe A avec la base de données
- Ils sont simples d'utilisation et permettent de "cacher" les nombreuses opérations nécessaires pour réaliser chacune des opérations mentionnées

Nous créerons ici une classe `MembreDAO` servant à interfacer les objets de la classe `Membre` avec la base de données.

3.1 Constructeur

Le constructeur prend comme paramètre le nom de la base de données et initialise l'unique attribut `db`.

- Ajoutez un nouveau fichier Python `membreDAO.py` à votre projet
- Recopiez dans ce fichier le code suivant définissant le constructeur de la classe :

```
import sqlite3, os, sys
from membre import Membre

class MembreDAO:
    def __init__(self, db_name):
        try:
            assert os.path.isfile(db_name)
            db = sqlite3.connect(db_name)
            print("Connexion à ", db_name, "OK.")
            self.db = db
        except:
            print("Erreur de connexion : la base n'existe pas!")
            sys.exit()
```

- Testez ce constructeur en créant un objet `a` de type `MembreDAO` dans le programme principal.

3.2 méthode getMembreById

Ajoutez à la classe `MembreDAO` une méthode nommée `getMembreById` qui :

- prend en paramètre un identifiant entier
- effectue une recherche dans la table **Membre**
- initialise un objet de type Membre avec les informations trouvées
- effectue une recherche des livres empruntés par ce membre dans la table **Emprunts**
- ajoute les livres trouvés dans la liste des emprunts de l'objet
- retourne l'objet

```
    def getMembreById(self, idMembre):
        ...
```



Attention, pensez à gérer le cas où le numéro fourni n'est pas présent dans la base

- Dans le programme principal, utilisez l'objet `a` créé précédemment pour définir l'objet `monet` d'identifiant 15, soit:
  - `monet = a.getMembreById(15)`
- Affichez ensuite le contenu de `monet`

3.3 méthode createMembre

La méthode `createMembre` sert à ajouter un nouveau membre dans la base de données. Elle :

- prend en paramètre un objet de type Membre
- effectue une requête d'insertion dans la table **Membre** à partir de l'objet fourni (`INSERT INTO Membre …`)
- ainsi qu'une requête d'insertion dans la table **Emprunt** à partir de la liste des emprunts contenue dans l'objet.

```
    def createMembre(self, membre):
        ...
```



La commande d'insertion utilise les valeurs contenues dans l'objet `membre`. Il est conseillé d'utiliser la syntaxe suivante :

```
        c.execute("INSERT INTO Membre VALUES (?,?,?,?)",
(membre.idMembre,   membre.nomMembre,   membre.adrMembre,
membre.cpMembre))
```



- Il est important de vérifier que l'identifiant de `membre` n'est pas déjà présent dans la base. Dans ce cas, il ne se passe rien (seul un message d'erreur



s'affiche)!
- Pour que les modifications soient prises en compte, il faut ajouter la commande suivante :

```
        self.db.commit()
```

Testez cette fonction dans le programme principal utilisant le membre que vous avez créé à la question 2 (pensez à lui attribuer un identifiant vierge, par exemple 31).

Si tout se passe bien, rien ne s'affiche mais il est possible de verifier que l'objet d'identifiant 30 a bien été inscrit dans la base à l'aide de la fonction `getMembreById` définie précédemment.

3.4 Méthode deleteMembre

Définissez une méthode `deleteMembre` permettant de supprimer un membre dans la base de données à l'aide de son identifiant (`DELETE FROM Membre WHERE …`).

From:
https://wiki.centrale-med.fr/informatique/ - **WiKi informatique**

Permanent link:
https://wiki.centrale-med.fr/informatique/restricted:tc-d:tp4:travaux_pratiques_quatrieme_seance_2017

Last update: **2018/01/22 14:22**