# Les bases de données par Python

## Module adapté

Il s'agit à travers la syntaxe :
`>>> import sqlite3`
d'utiliser le module adéquat afin d'interagir avec les bases de données de type SQL.

## Connexion à une base

Il est fortement conseillé d'importer le module *os* via
`>>> import os`
afin de naviguer dans l'arborescence du système d'exploitation, et ce dans le but d'interagir avec une base téléchargée ou créer des nouvelles tables dans des répertoires voulus.
`>>> os.chdir('chaîne de caractères de l'emplacement dans l'arbre')`
permet de se placer dans le répertoire de travail désiré.
Ensuite, la syntaxe :
`>>> ma_base = sqlite3.connect('une_base.db')`
permet de créer une base nommée `une_base` si elle n'existe pas encore dans le répertoire courant, la base étant stockée dans la mémoire `ma_base`.
On ouvre ainsi une connexion avec un fichier. À la fin du programme, il faudra penser à procéder à la déconnexion :
`>>> ma_base.close()`

## Requêtes SQL sur une base

Le principe est simple pour intégrer du langage SQL dans les scripts Python. La syntaxe générale est :
`>>> curseur=ma_base.cursor('une_base.db')`
On crée un curseur de lecture à travers lequel on passera et exécutera les commandes SQL.
`>>> curseur.execute(""" requête SQL """)`
`>>> ma_base.commit()`
La méthode `.commit()` lance la commande SQL.

## Exemples de commandes SQL

```
import sqlite3

import os

os.chdir('F:/IPT_MP/BDD_Par_Python')

ma_base=sqlite3.connect('une_base.db')

curseur=ma_base.cursor()

### exécution de commandes SQL  proprement dites

try :
    curseur.execute(""" DROP TABLE formes ;""") # pour supprimer une table lorsque celle-ci existe d
```

```
except :
    pass
curseur.execute(""" CREATE TABLE IF NOT EXISTS formes(
            id INTEGER PRIMARY KEY,
            formes TEXT,
            nbre_cotes INTEGER);""")

curseur.execute(""" INSERT INTO formes VALUES(1,"carré",4);""")
curseur.execute(""" INSERT INTO formes VALUES(2,"losange",4);""")
curseur.execute(""" INSERT INTO formes VALUES(3,"rectangle",4);""")
curseur.execute(""" INSERT INTO formes VALUES(4,"segment",1);""")
curseur.execute(""" INSERT INTO formes VALUES(5,"triangle",3);""")
curseur.execute(""" INSERT INTO formes VALUES(6,"pentagone",5);""")
curseur.execute(""" INSERT INTO formes VALUES(7,"hexagone",6);""")
curseur.execute(""" INSERT INTO formes VALUES(8,"cercle",NULL);""")
curseur.execute(""" INSERT INTO formes VALUES(9,"ellipse",NULL);""")
curseur.execute(""" INSERT INTO formes VALUES(10,"point",0);""")
ma_base.commit()
```

L'avantage de PYTHON est que l'on peut exécuter en boucle un certain nombre d'insertions, selon la syntaxe typique :

```
for k in range(domaine) :
    curseur.execute(
    """INSERT INTO formes (colonne1,colonne2,colonne3) VALUES(?,?,?)""",
    (valeur1,valeur2,valeur2) )
```

On peut alors visualiser sous Sqlite Manager par exemple que l'on a bien confectionné une table comme il faut.
On continue avec les requêtes SQL :

```
curseur.execute(""" SELECT * FROM formes ;""")
premiere_ligne=curseur.fetchone()
print(premiere_ligne)

>>> (1, 'carré', 4)

curseur.execute(""" SELECT * FROM formes ;""")
lignes=curseur.fetchall()
print(lignes)

>>> [(1, 'carré', 4), (2, 'losange', 4), (3, 'rectangle', 4), (4, 'segment', 1), (5, 'triangle', 3),
(6, 'pentagone', 5), (7, 'hexagone', 6), (8, 'cercle', None), (9, 'ellipse', None), (10, 'point', 0)
 (11, 'octogone', 8)]

curseur.execute(""" SELECT * FROM formes WHERE nbre_cotes>5 ;""")
reponse=curseur.fetchall()
for ligne in reponse :
    print("Le polygone nommé", ligne[1]," a ", ligne[2]," côtés.")

>>> Le polygone nommé hexagone  a  6  côtés.
Le polygone nommé octogone  a  8  côtés.
```

```
curseur.execute(""" SELECT COUNT(*),nbre_cotes FROM formes GROUP BY nbre_cotes ;""")
reponse=curseur.fetchall()
for ligne in reponse :
    print(ligne)

>>> (2, None)
(1, 0)
(1, 1)
(1, 3)
(3, 4)
(1, 5)
(1, 6)
(1, 8)
```

À noter que l'on peut encore effectuer des requêtes de ce type en boucle selon le principe :

```
curseur.execute(
""" SELECT * FROM liste_entiers WHERE entiers=? """
,valeur)
```

la variable `valeur` pouvant alors varier dans une boucle...

## Questions

Les questions portent tout d'abord sur la base *geographie.sqlite* téléchargeable. On utilisera avant le module sqlite3 de Python et on pourra vérifier les sorties par Sqlite Manager.

## Question 1)

Donner les noms des communes de plus de 10 000 habitants et de moins de 2 $km^2$.

## Question 2)

Afficher les villes de moins de 5000 habitants, ainsi que le numéro de département, la population et la superficie.

## Question 3)

Quelle est la plus grande commune de France ?

## Question 4)

Quelle est la plus grande commune de France hors de Guyane (973) ?

## Question 5)

Donner la liste des numéros de départements, avec pour chacun le nombre de communes, la population, les départements étant triés par ordre décroissant du nombre de communes.

## Question 6)

Déterminer pour chaque région, le nombre de départements, liste triée par ordre croissant de départements par région.

## Question 7)

Quels sont les cinq départements les moins peuplés en 2010 ?

## Navigation dans un réseau de transports imaginaire

1. Créer via Python une base par la syntaxe `base=sqlite3.connect('reseau.db')`.
2. Construire ensuite les tables suivantes :
   - table premiers comportant deux colonnes : nombres et ligne ; la colonne nombres comporte la liste ordonnée de tous les nombres premiers dans l'ensemble $[\![2, 100]\!]$ et la colonne ligne comporte toujours le champ *'prime'*
   - la table carres comportant deux colonnes : nombres et ligne ; la colonne nombres comporte la liste ordonnée de tous les carrés parfaits dans l'ensemble $[\![2, 100]\!]$ et la colonne ligne comporte toujours le champ *'square'*
   - la table mersenne comportant deux colonnes : nombres et ligne ; la colonne nombres comporte la liste ordonnée de tous les nombres de Mersenne (nombres de la forme $2^k - 1$, où $k \in \mathbb{N}$, dans l'ensemble $[\![2, 100]\!]$ et la colonne ligne comporte toujours le champ *'mersenne'*
   - la table cinq comportant deux colonnes : nombres et ligne ; la colonne nombres comporte la liste ordonnée de tous les entiers ayant 5 comme chiffre des unités et appartenant à l'ensemble $[\![2, 100]\!]$ et la colonne ligne comporte toujours le champ *'five'*
3. Créer une table liste_ligne comportant cinq colonnes : entiers, prime, square, mersenne et five, et indiquant pour chaque entier $k$ dans $[\![2, 100]\!]$ si l'entier $k$ fait partie ou non des lignes correspondantes.

   Par exemple, on dispose des lignes :

   `(4,non,oui,non,non)`

   `(5,oui,non,non,oui)`
4. Ajouter une colonne nommée appartenance, contenant des attribut de type *entier*.
5. Remplir cette nouvelle colonne en attribuant à chaque ligne, le nombre de « oui » dans la ligne.
6. Détruire les lignes où `appartenance=0`.
7. Afficher les lignes où l'attribut `appartenance` est maximale.

## Navigation dans un graphe

On considère un réseau métropolitain comportant quatre lignes de métro : « prime », « square », « mersenne » et « five ».
Chaque ligne comporte des stations dont l'ordre est celui décrit par les lignes dans chaque table associé à chaque ligne.
Par exemple, la ligne « five »comporte dans l'ordre les stations : `5,15,25,` $\cdots$ `85,95`. On peut donc transiter de la station 55 à la station 15 via cette ligne « prime », car on peut emprunter la ligne dans les deux sens de circulation.
En utilisant des tables de la base reseau,

1. rédiger une fonction permettant à partir d'une station `k`, renvoie la liste des stations du réseau accessibles à partir de `k` en un déplacement sur le réseau. Par exemple, la fonction renvoie pour la station `25` renvoie la liste à ordre près `[16,36,15,35]` car on peut emprunter la ligne « square » ou « five ».
2. rédiger une fonction admettant deux stations `dep` et `arr`, puis indique un chemin de longueur minimale permettant via les différentes lignes du réseau d'accéder de la station `dep` à la station `arr`.
3. proposer un chemin minimal partant de `97` vers `4`

4. proposer un chemin minimal partant de 100 vers 63.