Manipulation de données

SQL (Structured Query Language)

Xavier Gendre

Introduction

Très souvent les données sont regroupées dans des **bases de données**. Ces outils offrent à la fois :

- un système d'organisation des données,
- une manière d'y accéder de façon efficiente.

Le langage SQL (*Structured Query Language*) est majoritairement utilisé pour formuler les requêtes qui permettent de manipuler les données. Il est utilisé par les bases de données les plus populaires (PostgreSQL, MariaDB, SQLite, …).

L'objectif n'est pas d'apprendre SQL (même si certains éléments seront présentés) mais plutôt d'utiliser des outils Python pour accéder aux données présentes sur des serveurs de bases de données.

Deux approches seront présentées pour cela :

- **Pandas** pour récupérer les données et les manipuler depuis Python sous forme de `DataFrame`,
- **Ibis** pour interagir directement avec le serveur de bases de données.

Ces outils offrent une interface de communication avec des bases de données de type SQL à l'aide de **pilotes dédiés**.

Bases de données relationnelles

Les bases de données de type SQL utilisent le paradigme individus/variables :

- les bases contiennent des tables (comme les `DataFrame`),
- les tables contiennent des colonnes (ou champs/*fields*) qui regroupent des informations de même type,
- les enregistrements ou entrées d'une tables correspondent aux lignes de cette table.

Les tables sont reliées entre elles grâce à des identifiants (clés primaires *primary* et clés étrangères *foreign*).

Connexion à un serveur

Exemple de connexion à un serveur PostgreSQL :

```
1 import psycopg2
2
3 con = psycopg2.connect(
4     user="USERNAME",
5     password="PASSWORD",
6     host="HOST",
7     port="5432",
8     database="DATABASE_NAME"
9 )
```

Le module `psycopg2` fournit un pilote (*driver*) pour la base de données voulue.

D'autres modules fournissent des pilotes pour chaque base de données (`mariadb` pour MariaDB, …).

SQLite

Dans la suite, **SQLite** sera utilisé car le module `sqlite3` fait partie de la bibliothèque standard de Python. Au-delà de la connexion, les concepts abordés resteront valables pour toutes les bases de données relationnelles.

SQLite est une base de données relationnelle qui n'est pas basée sur le principe client/serveur.

SQLite permet de travailler sur des bases de données stockées dans des fichiers, voire directement en mémoire vive. Il s'agit d'un outil très simple à mettre en œuvre.

```
1 import sqlite3
2
3 con = sqlite3.connect(":memory:")
```

Une connexion vers une base de données SQLite stockée dans la mémoire vive est établie par l'objet `con`.

Pour interagir avec le serveur de base de données, un curseur (*cursor*) doit être créé. La méthode `execute` permet alors d'exécuter une **requête** (*query*) et la méthode `fetchall` de récupérer les résultats.

```
1 # Liste des tables de la base de données
2 query_tables = "SELECT name FROM sqlite_master WHERE type='table'"
3
4 cursor = con.cursor()
5 cursor.execute(query_tables)
6 cursor.fetchall()
```

```
[]
```

Pour l’instant, la base de données ne contient aucune table.
Des requêtes permettent de créer et de remplir une table.

```
# Création de la table Joueurs
cursor.execute("""
CREATE TABLE Joueurs(
    id INTEGER PRIMARY KEY, name TEXT, subscriber BIT, score INT
)
""")

# Ajout de données dans la table Joueurs
cursor.execute("INSERT INTO Joueurs VALUES(1, 'Bob', 1, 31415)")
cursor.execute("INSERT INTO Joueurs VALUES(2, 'Joy', 1, 42024)")
cursor.execute("INSERT INTO Joueurs VALUES(3, 'Ken', 0, 12345)")

# La table Joueurs existe maintenant
cursor.execute(query_tables)
cursor.fetchall()
```

```
[('Joueurs',)]
```

SQL en bref !

```
1  SELECT var1, var2 FROM table WHERE condition GROUP BY group
```

- `SELECT` Sélection de variables (`*` pour toutes)
- `FROM` Table d'origine
- `WHERE` Filtre sur les lignes
- `GROUP BY` Regroupement
- ...

Le langage SQL exprime des concepts similaires à ce qui a été présenté pour les `DataFrame` de Pandas. Il s'agit de la même organisation des données.

Récupérer les joueurs avec un grand score :

```
query = """
SELECT name, score
FROM Joueurs
WHERE score > 20000
"""
cursor.execute(query)
cursor.fetchall()
```

```
[('Bob', 31415), ('Joy', 42024)]
```

Obtenir la moyenne des scores par groupe :

```
query = """
SELECT subscriber, AVG(score) -- Agrégateurs de SQL
FROM Joueurs
GROUP BY subscriber
"""
cursor.execute(query)
cursor.fetchall()
```

```
[(0, 12345.0), (1, 36719.5)]
```

Sauvegarde

La base de données créée en mémoire vive peut être sauvegardée dans un fichier pour une utilisation ultérieure. Depuis Python 3.7, une connexion établie par le module `sqlite3` dispose d'une méthode `backup` pour cela.

Les changements précédents de la base doivent être validés par la méthode `commit` auparavant.

```
con.commit() # Valide les changements de la base de données
con_backup = sqlite3.connect("backup.db") # Base en fichier
con.backup(con_backup) # Copie de la base en mémoire
con_backup.close() # Fermeture de la base en fichier
```

Fin de connexion

Afin de libérer les resources, la connexion avec la base de données doit être fermée une fois les opérations terminées :

```
con.close()
```

L'objet de connexion peut également être géré par les instructions `with ... as`:

```
with sqlite3.connect("backup.db") as con:
    cursor = con.cursor()
    cursor.execute(query)
    result = cursor.fetchall()

result
```

```
[(0, 12345.0), (1, 36719.5)]
```

SQL avec Pandas

La fonction `read_sql` de Pandas permet d'interagir avec le serveur de bases de données au travers de l'objet de connexion et de manipuler les données sous forme de `DataFrame`.

```
1 import pandas as pd
2
3 con = sqlite3.connect("backup.db")
4 df = pd.read_sql("SELECT * FROM Joueurs", con)
5
6 print(df)
```

```
   id name  subscriber  score
0   1  Bob           1  31415
1   2  Joy           1  42024
2   3  Ken           0  12345
```

La fonction `to_sql` permet de créer de nouvelles tables à partir d'un `DataFrame` et de l'objet de connexion.

```
import random

df_new = pd.DataFrame({
    "id": list(range(5)),
    "value": random.choices(["A", "B"], k=5),
})

# Nouvelle table Hasard sans index
df_new.to_sql(name="Hasard", index=False, con=con)

df = pd.read_sql("SELECT * FROM Hasard", con)
con.close()

print(df)
```

```
   id value
0   0     B
1   1     B
2   2     A
3   3     A
4   4     B
```

Un exemple complet

STAR est un système de vélo en libre-service mis en place par Rennes Métropole. Le fichier `star.db` au format SQLite contient des données sur l'état du système et sur sa topologie à un instant donné.

```
con = sqlite3.connect("data/star.db")
print(
    pd.read_sql(query_tables, con)
)
```

```
        name
0  Topologie
1       Etat
```

Table `Etat`

```
1 df_etat = pd.read_sql("SELECT * FROM Etat", con)
2 df_etat.dtypes
```

```
id                          int64
nom                        object
latitude                  float64
longitude                 float64
etat                       object
nb_emplacements             int64
emplacements_disponibles    int64
velos_disponibles           int64
date                      float64
data                       object
dtype: object
```

Table Topologie

```
df_topologie = pd.read_sql("SELECT * FROM Topologie", con)
df_topologie.dtypes
```

```
id                    int64
nom                  object
adresse_numero       object
adresse_voie         object
commune              object
latitude            float64
longitude           float64
id_correspondance   float64
mise_en_service     float64
nb_emplacements       int64
id_proche_1           int64
id_proche_2           int64
id_proche_3           int64
terminal_cb          object
dtype: object
```

Question

La position GPS de la gare de Rennes est `(48.103712,-1.672342)`.

L'objectif est de trouver les trois stations les plus proches et d'afficher certaines informations utiles :

- nom et adresse des stations,
- uniquement des stations en fonctionnement et disposant d'au moins un vélo disponible.

Les informations nécessaires se trouvent dans les deux tables. Il va donc falloir faire une jointure.

La première étape consiste à écrire la requête de la table à compléter :

```
query_base = """
SELECT
  id,
  POWER((latitude - 48.103712), 2.0)
  + POWER((longitude + 1.672342), 2.0) AS distance
FROM Etat
WHERE
  etat = 'En fonctionnement'
  AND velos_disponibles > 0
"""
```

```
print(
    pd.read_sql(query_base, con)
)
```

```
    id  distance
0    1  0.000072
1    2  0.000104
2    3  0.000142
3   10  0.000038
4   12  0.000018
..  ..       ...
78  62  0.000112
79  66  0.000519
80  69  0.000258
81  85  0.000515
82  86  0.000690

[83 rows x 2 columns]
```

La syntaxe SQL d'une jointure à gauche simple est de la forme suivante :

```
1  SELECT left.var1, left.var2, right.var3
2  FROM left
3  LEFT JOIN right ON left.id = right.id
```

Le résultat de la requête précédente servira de table gauche (*left*) pour les variables `id` et `distance`. Les variables `nom` et `adresse` seront obtenue par jointure avec la table droite (*right*) donnée par `Topologie`.

Requête du problème :

```
query = (
    """
    SELECT
      left.id, left.distance, right.nom,
      (
        COALESCE(right.adresse_numero, '')
        || ' '
        || COALESCE(right.adresse_voie, '')
      ) AS adresse
    FROM (
    """ + query_base
    + """
    ) AS left
    LEFT JOIN Topologie AS right ON left.id = right.id
    ORDER BY distance
    LIMIT 3
    """
)
```

Résultat

```
print(
    pd.read_sql(query, con)
)
```

```
   id  distance                 nom                     adresse
0  15  0.000001  Gares - Solférino       18 Place de la Gare
1  45  0.000003  Gares Sud - Féval     19 B Rue de Châtillon
2  84  0.000003   Gares - Beaumont  22 Boulevard de Beaumont
```

Fin de la connexion :

```
con.close()
```

```
SELECT
  left.id, left.distance, right.nom,
  (
    COALESCE(right.adresse_numero, '')
    || ' '
    || COALESCE(right.adresse_voie, '')
  ) AS adresse
FROM (
  SELECT
    id,
    POWER((latitude - 48.103712), 2.0)
    + POWER((longitude + 1.672342), 2.0) AS distance
  FROM Etat
  WHERE
    etat = 'En fonctionnement'
    AND velos_disponibles > 0
  ) AS left
LEFT JOIN Topologie AS right ON left.id = right.id
ORDER BY distance
LIMIT 3
```

Conclusion (?)

Pour manipuler des données relationnelles, il faut apprendre un peu de SQL 😒

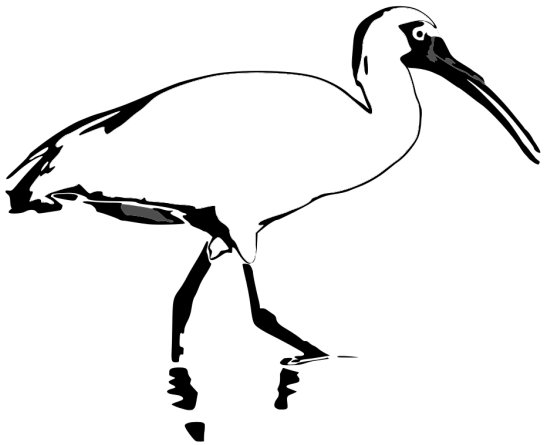
Un tel investissement n'est rentable que si l'on a un usage régulier des bases de données relationnelles.

Les similarités entre le langage SQL et des modules Python tels que Pandas proviennent de la capacité d'abstraction du langage SQL pour décrire la manipulation des données.

Pour une utilisation occasionnelle des bases de données relationnelles, un "traducteur" serait bien utile 😉

Ibis

Ibis offre une interface de programmation pour des moteurs de requête variés dont les bases de données relationnelles.



La présentation de ce module libre distribué sous licence Apache sera limitée à SQLite mais les possibilités sont bien plus nombreuses.

Pour l'anecdote, l'ibis est l'oiseau perché sur l'éléphant qui est le symbole de PostgreSQL 🐘

Connexion avec SQLite

Le module `ibis` propose une option interactive qui facilite l'exploration des données.

```
1 import ibis
2
3 ibis.options.interactive = True
```

La connexion par défaut se fait vers une base de données éphémère en mémoire.

```
1 con = ibis.sqlite.connect()
```

Pour interagir avec une base de données stockée dans un fichier, il suffit de le passer en argument à la fonction `connect`.

```
1 con = ibis.sqlite.connect("backup.db")
```

Gestion des tables avec Ibis

Ibis offre des méthodes pour les opérations les plus courantes. Par exemple, la liste des tables disponibles s'obtient avec `list_tables` (ou l'attribut `tables` pour un simple affichage).

```
1 con.list_tables()
```

```
['Hasard', 'Joueurs']
```

```
1 con.tables
```

```
Tables
------
- Hasard
- Joueurs
```

Il est possible de créer une nouvelle table à partir d'un `DataFrame` avec `create_table`.

```
con.create_table("HasardBis", df)
con.tables
```

```
Tables
------
- Hasard
- HasardBis
- Joueurs
```

La suppression d'une table se fait avec `drop_table`.

```
con.drop_table("HasardBis")
con.tables
```

```
Tables
------
- Hasard
- Joueurs
```

Manipulation des données avec Ibis

Ibis permet de manipuler facilement les jeux de données contenus dans les tables.

```
joueurs = con.table("Joueurs")
joueurs.columns
```

```
['id', 'name', 'subscriber', 'score']
```

```
# Mode interactif
joueurs
```

id	name	subscriber	score
int32	string	decimal	int32
1	Bob	1.0	31415
2	Joy	1.0	42024
3	Ken	0.0	12345

Ibis permet d'exporter le contenu d'une table dans plusieurs formats (CSV, Parquet, …) dont celui du `DataFrame` avec la méthode `to_pandas`.

```
print(
    joueurs.to_pandas()
)
```

```
   id name subscriber  score
0   1  Bob        1.0  31415
1   2  Joy        1.0  42024
2   3  Ken        0.0  12345
```

Les méthodes offertes par Ibis sont moins variées que celles de Pandas mais elles permettent les mêmes opérations classiques (avec des noms similaires à la syntaxe de `dplyr` pour les utilisateurs de R).

- Sélection de colonnes

```
1 joueurs.select("id", "name", "score")
```

id	name	score
int32	string	int32
1	Bob	31415
2	Joy	42024
3	Ken	12345

- Filtre sur les lignes

```
1 joueurs[joueurs.score > 20000]
```

id	name	subscriber	score
int32	string	decimal	int32
1	Bob	1.0	31415
2	Joy	1.0	42024

- Tri des lignes

```
1 joueurs.order_by([joueurs.score])
```

id	name	subscriber	score
int32	string	decimal	int32
3	Ken	0.0	12345
1	Bob	1.0	31415
2	Joy	1.0	42024

- Mutation pour créer ou modifier des colonnes

```
1  joueurs.mutate(new_score = joueurs.score * 2)
```

id	name	subscriber	score	new_score
int32	string	decimal	int32	int64
1	Bob	1.0	31415	62830
2	Joy	1.0	42024	84048
3	Ken	0.0	12345	24690

Les données initiales ne sont pas affectées.

```
1  joueurs.mutate(score = joueurs.score * 2)
```

id	name	subscriber	score
int32	string	decimal	int64
1	Bob	1.0	62830
2	Joy	1.0	84048
3	Ken	0.0	24690

- Agrégation pour résumer l'information

```
1 joueurs.aggregate(
2     n_subscriber = joueurs.subscriber.sum(),
3     min_score = joueurs.score.min(),
4     max_score = joueurs.score.max(),
5     mean_score = joueurs.score.mean(),
6     var_score = joueurs.score.var(),
7 )
```

n_subscriber	min_score	max_score	mean_score	var_score
decimal	int32	int32	float64	float64
2.0	12345	42024	28594.666667	2.261765e+08

- Regroupement

```
(
    joueurs
    .group_by("subscriber")
    .aggregate(
        n = joueurs.count(), # Agrégateur count
        n_names = joueurs.name.nunique(), # Agrégateur nunique
        mean_score = joueurs.score.mean(),
    )
)
```

subscriber	n	n_names	mean_score
decimal	int64	int64	float64
0.0	1	1	12345.0
1.0	2	2	36719.5

- Et bien d'autres choses (jointures, …)

Génération de requêtes SQL

Ibis limite la manipulation des données aux concepts des bases de données relationnelles mais cela permet de mettre en place un mécanisme de traduction et de génération de requêtes SQL avec la fonction `to_sql`.

```
ma_table = joueurs[joueurs.score > 20000]
ibis.to_sql(ma_table)
```

```
SELECT
  t0.id,
  t0.name,
  t0.subscriber,
  t0.score
FROM "Joueurs" AS t0
WHERE
  t0.score > 20000
```

```
ibis.to_sql(
    joueurs[joueurs.score > 20000]
    .select("id", "name", "score")
    .mutate(new_score = joueurs.score * 2)
    .order_by("new_score")
)
```

```
WITH t0 AS (
  SELECT
    t2.id AS id,
    t2.name AS name,
    t2.subscriber AS subscriber,
    t2.score AS score
  FROM "Joueurs" AS t2
  WHERE
    t2.score > 20000
)
SELECT
  t1.id,
  t1.name,
  t1.score,
  t1.new_score
FROM (
  SELECT
    t0.id AS id,
```

```
ibis.to_sql(
    joueurs
    .group_by("subscriber")
    .aggregate(mean_score = joueurs.score.mean())
)
```

```
SELECT
  t0.subscriber,
  AVG(t0.score) AS mean_score
FROM "Joueurs" AS t0
GROUP BY
  1
```

Ces requêtes peuvent ensuite être utilisées directement sur le serveur de bases de données comme présenté dans la partie précédente.

Un exemple complet (avec Ibis)

Connexion vers la base de données

```
1 con = ibis.sqlite.connect("data/star.db")
2 con.tables
```

```
Tables
------
- Etat
- Topologie
```

Gestion des tables avec Ibis

```
1 etat = con.table("Etat")
2 topologie = con.table("Topologie")
```

Table des distances à la gare de Rennes pour les stations fonctionnelles avec au moins un vélo

```
# Potentiellement problématique mais très utile avec Ibis
from ibis import _

table_base = (
    etat[
        (etat.etat == "En fonctionnement")
        & (etat.velos_disponibles > 0)
    ]
    .mutate(
        d_lat = etat.latitude - 48.103712,
        d_lon = etat.longitude + 1.672342,
    )
    .mutate(
        # Usage de _
        distance = _.d_lat * _.d_lat + _.d_lon * _.d_lon
    )
    .select("id", "distance")
)
```

```
1 table_ba
```

id	distance
int32	float64
1	0.000072
2	0.000104
3	0.000142
10	0.000038
12	0.000018
14	0.000052
17	0.000025
20	0.000139
22	0.000207
25	0.000159
…	…

```
1 ibis.to_sql(table_base)
```

```
1  WITH t0 AS (
2    SELECT
3      t2.id AS id,
4      t2.nom AS nom,
5      t2.latitude AS latitude,
6      t2.longitude AS longitude,
7      t2.etat AS etat,
8      t2.nb_emplacements AS nb_emplacements,
9      t2.emplacements_disponibles AS emplacem
10     t2.velos_disponibles AS velos_disponibl
11     t2.date AS date,
12     t2.data AS data,
13     t2.latitude - 48.103712 AS d_lat,
14     t2.longitude + 1.672342 AS d_lon
15   FROM "Etat" AS t2
16   WHERE
17     t2.etat = 'En fonctionnement' AND t2.ve
18 )
19 SELECT
```

Les requêtes générées ne sont pas optimisées mais elles fonctionnent.

Réponse à la question des 3 stations les plus proches :

```
resultat = (
    table_base
    .left_join(topologie, table_base.id == topologie.id) # Jointure
    .mutate(
        adresse = topologie.adresse_numero + " " + topologie.adress
    )
    .select("id", "distance", "nom", "adresse")
    .order_by("distance")
    .limit(3) # Limite du nombre de lignes
)

resultat
```

id	distance	nom	adresse
int32	float64	string	string
15	0.000001	Gares - Solférino	18 Place de la Gare
45	0.000003	Gares Sud - Féval	19 B Rue de Châtillon
84	0.000003	Gares - Beaumont	22 Boulevard de Beaumont

```
ibis.to_sql(resultat)
```

```
WITH t0 AS (
  SELECT
    t5.id AS id,
    t5.nom AS nom,
    t5.latitude AS latitude,
    t5.longitude AS longitude,
    t5.etat AS etat,
    t5.nb_emplacements AS nb_emplacements,
    t5.emplacements_disponibles AS emplacements_disponibles,
    t5.velos_disponibles AS velos_disponibles,
    t5.date AS date,
    t5.data AS data,
    t5.latitude - 48.103712 AS d_lat,
    t5.longitude + 1.672342 AS d_lon
  FROM "Etat" AS t5
  WHERE
    t5.etat = 'En fonctionnement' AND t5.velos_disponibles > 0
), t1 AS (
  SELECT
```

À vous de jouer !