

# TP table de données, requêtes et tri

Traitement de données en tables

Première NSI Lycée du Parc

## Table des matières

|   |    |
|---|----|
| Crédits                                     | 1  |
| 1 Requetes dans une table de données        | 1  |
| 2 Requetes de recherche                     | 2  |
| 3 Requetes avec operation d'agrégation      | 3  |
| 4 Requetes avec sélection de lignes         | 5  |
| 5 Requetes avec projection sur des colonnes | 7  |
| 6 Tri d'une table de données                | 8  |
| 7 Applications à un autre exemple           | 12 |
| 8 Synthèse                                  | 16 |

## Crédits

*Ce TP est largement inspiré :*

- des chapitres 16 et 17 du manuel NSI de la collection Tortue chez Ellipse, auteurs : Ballabonski, Conchon, Filliatre, N'Guyen ;
- du document d'accompagnement des programmes de NSI disponible sur Eduscol

## 1 Requetes dans une table de données



### Définition 1

Dans le cours, on a vu comment charger une **table de données**, enregistrée dans un **fichier CSV**, dans une structure de données permettant un accès efficace aux **enregistrements** et aux valeurs de leurs **attributs**. En Python, une structure de données bien adaptée est un **tableau de dictionnaires**.

Les opérations sur les tableaux et les dictionnaires déjà étudiées vont nous permettre d'interroger une **table de données** à l'aide d'opération appelées **requêtes** :

- pour tester l'appartenance à la table d'un enregistrement vérifiant une certaine condition sur ses attributs ;
- pour calculer une valeur en combinant les valeurs d'attributs de plusieurs enregistrements (**opération d'agrégation**) ;
- pour sélectionner tous les enregistrements vérifiant une certaine condition sur leurs attributs (**opération de sélection par lignes**) ;
- pour projeter les enregistrements sur certains de leurs attributs (**opération de projection sur des colonnes**) ;
- pour trier les enregistrements en fonction d'un ordre prédéfini sur leurs attributs (**opération de tri**).

## 2 Requetes de recherche



### Exercice 1

1. Ouvrir le script 'TP\_Recherche\_Triseleve.py' dans un IDE Python
2. Charger dans une variable `table_clients` la table contenue dans le fichier 'clients.csv' à l'aide de la fonction `lecture_csv` qui est fournie. Il s'agit de la table des clients d'un site marchand que nous avons manipulée dans le cours. Exécuter le test unitaire `test_import_table_clients(table_clients)`.

```
table_clients = lecture_csv('clients.csv', ',')
```

3. Compléter la fonction `recherche_attribut` ci-dessous en respectant sa spécification. Vérifier en exécutant le test unitaire `test_recherche_attribut_table_clients(table_clients)`.

```
def recherche_attribut(table, attribut, valeur):  
    """Paramètres :  
        table un tableau de dictionnaires, table de clients.csv  
        attribut de type str valeur du type d'attribut dans table  
    Valeur renvoyée:  
        Un booléen indiquant si table contient un enregistrement e  
        tel que e[attribut] == valeur"""  
    for enregistrement in table:  
        if .....:  
            return .....  
    return .....
```

4. Compléter la fonction `recherche_attribut_et` ci-dessous en respectant sa spécification. Exécuter le test unitaire `test_recherche_attributs_et(table_clients)`.

```

def recherche_attributs_et(table, attribut1, valeur1, attribut2,
    valeur2):
    """Paramètres :
        table un tableau de dictionnaires, table de clients.csv
        attribut1 de type str, valeur1 du type d'attribut1
        attribut2 de type str, valeur2 du type d'attribut2
    Valeur renvoyée:
        Un booléen indiquant si table contient un enregistrement e
        tel que e[attribut1] == valeur1 et e[attribut2] == valeur2
    """
    for enregistrement in table:
        if .....:
            return .....
    return .....

```

Proposer une modification de `recherche_attributs_et` qui prend en paramètres deux tableaux `attribut` et `valeur` et recherche si la table contient un enregistrement pour lequel tous les attributs listés correspondent aux valeurs listées puis un enregistrement pour lequel au moins un des attributs listés correspond à son homologue dans la liste des valeurs.

### 3 Requêtes avec opération d'agrégation

#### Exercice 2

On travaille toujours avec la table contenue dans le fichier 'clients.csv' avec le script 'TP\_Recherche\_Tris\_Eleve.py' dans un IDE Python.

1. Compléter la fonction `nombre_departement` ci-dessous en respectant sa spécification. On donne une postcondition qui doit être vérifiée.

```

def nombre_departement(table, departement):
    """Paramètres :
        table un tableau de dictionnaires, table de clients.csv
        departement de type str, un numéro de département
    Valeur renvoyée :
        Nombre d'occurrences de departement dans table"""
    compteur = 0
    for enregistrement in table:
        if .....:
            compteur = .....
    return compteur

# postcondition
assert nombre_departement(table_clients, "69") == 481

```

2. Compléter la fonction `nombre_departement` ci-dessous en respectant sa spécification. Exécuter le test unitaire `test_nombre_occurences(table_clients)`.

```
def nombre_occurences(table, attribut, valeur):
    """Paramètres :
        table un tableau de dictionnaires, table de clients.csv
        attribut de type str, valeur du type d'attribut dans table
    Valeur renvoyée :
        Nombre d'occurences d'attribut avec valeur dans table"""
    .....
    .....
    .....
    .....
    .....
    .....
```

3. Compléter la fonction `moyenne_visites` ci-dessous en respectant sa spécification. On donne une postcondition qui doit être vérifiée.

```
def moyenne_visites(table):
    """Paramètres :
        table un tableau de dictionnaires, table de clients.csv
    Valeur renvoyée :
        Moyenne des visites par enregistrement de type float"""
    somme = 0
    taille = 0
    for enregistrement in table:
        .....
        .....
        .....
    return somme / taille

#postcondition
assert moyenne_visites(table_clients) == 76.2807
```

4. Compléter la fonction `minimum_visites` ci-dessous en respectant sa spécification. On donne une postcondition qui doit être vérifiée.

```
def minimum_visites(table):
    """Paramètres :
        table un tableau de dictionnaires, table de clients.csv
    Valeur renvoyée :
        nombre minimum de visites de type int"""
    min_visites = int(table[0]['visites'])
    for enregistrement in table[1:]:
        .....
        .....
```

```

.....
return min_visites

#postcondition
assert minimum_visites(table_clients) == 2

```

5. Compléter la fonction `departement_max_occurence` ci-dessous en respectant sa spécification. On donne une postcondition qui doit être vérifiée.

```

def departement_max_occurence(table):
    """Paramètre : table sous forme de tableau de dictionnaires
    Valeur renvoyée : tuple formé du nombre d'occurrences maximal
    parmi les départements
    et du tableau des départements réalisant ce maximum
    """
    .....
    .....

assert departement_max_occurence(table_clients) == (547, ['59'])

```

## 4 Requêtes avec sélection de lignes



### Exercice 3

On travaille toujours avec la table contenue dans le fichier '`clients.csv`' avec le script '`TP_Recherche_Triseleve.py`' dans un IDE Python.

Une opération de **sélection** consiste à construire une nouvelle table avec les mêmes attributs mais en filtrant les enregistrements (ou lignes) selon une condition logique.

1. Compléter la fonction `selection_departement` ci-dessous en respectant sa spécification. On donne une postcondition qui doit être vérifiée.

```

def selection_departement(table, departement):
    """
    Paramètres :
        table une table sous forme de tableau de dictionnaires
        departement une chaîne de caractères représentant un dé
            partement
    Valeur renvoyée :
        tableau de dictionnaires contenant les enregistrement de
            table dont l'attribut "département" a la valeur passée
            en paramètre
    """
    return [enregistrement
            for enregistrement in table if
                ..... ]

# postcondition
assert selection_departement(table_clients, "69")[0]['email'] == '
    nguyen@noos.fr' \
        and len(selection_departement(table_clients, "69")) == 481

```

2. Compléter la fonction `selection_depart_visites_min` ci-dessous en respectant sa spécification. On donne une postcondition qui doit être vérifiée.

```

def selection_depart_visites_min(table, departement, visites_min):
    """Paramètres :
        table une table sous forme de tableau de dictionnaires
        departement une chaîne de caractères représentant un dé
            partement
        visites_min un entier naturel
    Valeur renvoyée :
        tableau de dictionnaires contenant les enregistrement de
            table dont l'attribut "département" a la valeur passée
            en paramètre et l'attribut visites est >= visites_min
    """
    return [enregistrement for enregistrement in table
            if
                .....]

assert len(selection_depart_visites_min(table_clients,"69",100))
    ==171

```

## 5 Requêtes avec projection sur des colonnes

### Exercice 4

On travaille toujours avec la table contenue dans le fichier 'clients.csv' avec le script 'TP\_Recherche\_Triseleve.py' dans un IDE Python.

Une opération de **projection** consiste à construire une nouvelle table en filtrant les attributs.

1. Compléter la fonction `projection_visites` ci-dessous en respectant sa spécification. On donne une postcondition qui doit être vérifiée.

```
def projection_visites(table):
    """Paramètres :
        table une table sous forme de tableau de dictionnaires
    Valeur renvoyée :
        tableau des valeurs des attributs "visites" pour les
            enregistrements de table
        avec conversion des nombres de visites en entiers
    """
    return [ ..... for enregistrement in table ]

assert projection_visites(table_clients)[:10] == [57, 145, 67, 131,
        76, 52, 65, 3, 101, 18]
```

2. Compléter la fonction `selection_departement_projection_visites` ci-dessous en respectant sa spécification. On donne une postcondition qui doit être vérifiée.

```
def selection_departement_projection_visites(table, departement):
    """Paramètres :
        table une table sous forme de tableau de dictionnaires
    Valeur renvoyée :
        tableau des valeurs des attributs "visites" pour les
            enregistrements de table du département
        passé en paramètre avec conversion des nombres de visites en
            entiers"""
    .....
    .....

assert selection_departement_projection_visites(table_clients, "69")
        [:10] == [43, 52, 127, 53, 41, 117, 31, 86, 107, 145]
```

3. Compléter la fonction `projection_departement_age` ci-dessous en respectant sa spécification. On donne une postcondition qui doit être vérifiée.

```

def projection_departement_age(table, annee):
    """Paramètres :
        table une table sous forme de tableau de dictionnaires
    Valeur renvoyée :
        tableau de dictionnaires avec deux attributs le département
        et l'âge du client
        calculé à partir de sa date de naissance."""
    return [{'département' : enreg['département'],
            'âge' : .....}]
    for enreg in table]

assert projection_departement_age(table_clients, 2021)[:3] == [{'département': '79', 'âge': 61}, {'département': '10', 'âge': 27}, {'département': '73', 'âge': 20}]

```

## 6 Tri d'une table de données



### Méthode

Pour trier une **table de données** implémentée sous forme de **tableau de dictionnaires** en Python, il est préférable d'utiliser les fonctions de tri de la bibliothèque standard plutôt que des tris que nous avons programmés.

D'une part nous avons ainsi la garantie d'une complexité optimale en  $O(n \log(n))$ , d'autre part les fonctions de tri de bibliothèque permettent de paramétrer le critère de tri appelé aussi **clef de tri**.

Il existe deux fonctions de tri dans la bibliothèque standard :

- `sorted` s'applique à un tableau `tab` avec la syntaxe `sorted(tab, key = clef_tri)` et renvoie un nouveau tableau avec les éléments du tableau `tab` triés selon la clef de tri passée en paramètre. Celle-ci est une fonction qui s'applique à un élément du tableau et renvoie une valeur calculée à partir de celle de l'élément. Le tri s'effectue en comparant les valeurs renvoyées par la clef de tri pour chaque élément. Cette valeur peut être un **tuple** puisque dans une table, un élément est un dictionnaire avec éventuellement plusieurs attributs. Dans ce cas, pour comparer deux valeurs, on applique **l'ordre lexicographique** : on compare les premières composantes, puis les secondes etc .... Par défaut la comparaison s'effectue dans l'ordre croissant, mais avec le paramètre optionnel `reverse` on peut demander un ordre décroissant : `sorted(tab, key = clef_tri, reverse = True)`.
- `sort` est une fonction de tri en place, elle ne renvoie pas un nouveau tableau. Elle s'applique à un tableau `tab` avec la syntaxe `tab.sort(key = clef_tri)`, les paramètres sont les mêmes que pour `sorted`.

Attention, `sorted` ne renvoie qu'une copie superficielle du tableau ! Si ses éléments sont des références et c'est le cas de l'implémentation des **tables de données** sous forme de **tableaux de dictionnaires**, il faut effectuer une copie profonde du tableau avec la fonction `deepcopy` du module `copy` pour



obtenir une vraie copie triée du tableau initiale : `sorted(deepcopy(tab), key = clef_tri)`.

On donne ci-dessous quelques exemples. Notons que pour obtenir un tri décroissant selon l'attribut 'note' puis croissant selon l'attribut 'langage', on ne peut pas utiliser une clef de tri qui renvoie le couple d'attributs ('note', 'langage') car les ordres ne sont pas les mêmes selon les composantes. On procède par composition des tris avec `sorted` dans l'ordre inverse des priorités de tri : d'abord selon l'attribut `note` puis selon l'attribut `langage`.

Une propriété importante des fonctions de tri de bibliothèque `sorted` et `sort` est la **stabilité du tri** : si on enchaîne deux tris successifs (par ordre lexicographique ou composition de `sorted`), deux éléments égaux pour le second tri conservent l'ordre du premier tri.

```
>>> table = [{'élève' : 'guido', 'langage' : 'python', 'note' : 19}, {'é
lève' : 'monty', 'langage' : 'python', 'note' : 20}, {'élève' : '
brian', 'langage' : 'c', 'note' : 20}]
>>> def clef_note(enreg):
...     return enreg['note']
>>> def clef_langage(enreg):
...     return enreg['langage']
>>> def clef_langage_note(enreg):
...     return (enreg['langage'], enreg['note'])
>>> def clef_note_langage(enreg):
...     return (enreg['note'], enreg['langage'])
>>> sorted(table, key = clef_note)
[{'élève': 'guido', 'langage': 'python', 'note': 19}, {'élève': 'monty',
 'langage': 'python', 'note': 20}, {'élève': 'brian', 'langage': 'c
', 'note': 20}]
>>> sorted(table, key = clef_note, reverse = True)
[{'élève': 'monty', 'langage': 'python', 'note': 20}, {'élève': 'brian',
 'langage': 'c', 'note': 20}, {'élève': 'guido', 'langage': 'python
', 'note': 19}]
>>> sorted(table, key = clef_langage_note) #ordre lexicographique (
langage croissant, note croissant)
[{'élève': 'brian', 'langage': 'c', 'note': 20}, {'élève': 'guido', '
langage': 'python', 'note': 19}, {'élève': 'monty', 'langage': '
python', 'note': 20}]
>>> sorted(table, key = clef_note_langage) #ordre lexicographique (note
croissant, langage croissant)
[{'élève': 'guido', 'langage': 'python', 'note': 19}, {'élève': 'brian',
 'langage': 'c', 'note': 20}, {'élève': 'monty', 'langage': 'python
', 'note': 20}]
>>> table_tri_lang_cr = sorted(table, key = clef_langage) #on va
composer les tris avec sorted
>>> sorted(table_tri_lang_cr, key = clef_note, reverse = True) # é
quivalent à (note décroissant, langage croissant)
[{'élève': 'brian', 'langage': 'c', 'note': 20}, {'élève': 'monty', '
langage': 'python', 'note': 20}, {'élève': 'guido', 'langage': '

```

```
python', 'note': 19}]
```



## Exercice 5

On travaille toujours avec la table contenue dans le fichier 'clients.csv' avec le script 'TP\_Recherche\_Trис\_Eleve.py' dans un IDE Python.

1. Compléter la clef de tri `clef_departement` pour que `table_tri_departement` soit trié dans l'ordre croissant des numéros de département dans le code ci-dessous. Vérifier la postcondition donnée dans le fichier.

```
def clef_departement(enreg):  
    return .....  
  
table_tri_departement = sorted(table_clients, key = clef_departement  
    )
```

2. Compléter la clef de tri `clef_visites` pour que `table_tri_visites_decroissant` soit trié dans l'ordre décroissant des nombres de visites dans le code ci-dessous. Vérifier la postcondition donnée dans le fichier.

```
def clef_visites(enreg):  
    return .....  
  
table_tri_visites_decroissant = sorted(table_clients, key =  
    clef_visites, reverse = True)
```

3. Compléter la clef de tri `clef_departement_visites` pour que `table_tri_dep_vis_croissant` soit trié dans l'ordre croissant des départements puis des nombres de visites dans le code ci-dessous. Vérifier la postcondition donnée dans le fichier.

```
def clef_departement_visites(enreg):  
    return .....  
  
table_tri_dep_vis_croissant = sorted(table_clients, key =  
    clef_departement_visites)
```

4. Proposer une instruction qui permette de trier `table_clients` d'abord par département croissant puis par nombre de visites décroissant. Vérifier la postcondition donnée dans le fichier.

```
table_tri_dep_crois_vis_decrois = .....  
    .....
```

5. Le fichier `transactions.csv` contient une table de données de nouvelles transactions effectuées sur le site marchand. Chaque transaction est identifiée par deux attributs : 'email' pour l'email du client et 'dépenses' pour le montant de la dépense.

```
email,dépenses
wpereira@orange.fr,104.91
ariviere@tiscali.fr,18.37
```

On souhaite mettre à jour les attributs 'visites' et 'dépenses' de `table_clients` avec les nouvelles transactions. On considère que chaque client est identifié de façon unique par son email et on peut donc insérer efficacement les valeurs de chaque transaction dans `table_clients` avec une recherche dichotomique sur l'attribut 'email'. Bien sûr, il faut d'abord trier `table_clients` selon l'attribut 'email'. Compléter le code ci-dessous. Décommenter les postconditions dans le fichier pour vérifier le code et contrôler le contenu du fichier de sortie 'clients\_maj.csv'.

```
from copy import deepcopy #pour réaliser une copie de table

def recherche_dicho_croissant(element, table, attribut):
    """Paramètres :
        table un tableau de dictionnaires
        attribut de type str
        element une valeur possible pour l'attribut
    Valeur renvoyée : index de la valeur element de attribut dans
        table"""
    debut = 0
    fin = len(table) - 1
    while fin - debut >= 0:
        milieu = (debut + fin) // 2
        if table[milieu][attribut] < element:
            debut = .....
        elif table[milieu][attribut] > element:
            fin = .....
        else:
            return .....
    return None

def clef_email(enreg):
    return enreg['email']

def maj_depenses_table(table, transactions):
    """Paramètres : table et transactions deux tables sous forme de
        tableaux de dictionnaires
    Valeur renvoyée :
        table_tri un tableau de dictionnaires mise à jour des
            attributs 'visites' et 'dépenses'
        de table par les valeurs de transactions"""
    table_tri = sorted(deepcopy(table), key = clef_email)
    table_cible = []
```

```

for enreg in transactions:
    index_email = recherche_dicho_croissant(enreg['email'],
        table_tri, 'email')
    if index_email is not None:
        .....
        .....
        .....
return table_tri

table_clients = lecture_csv('clients.csv', ',')
transactions = lecture_csv('transactions.csv', ',')
table_tri = maj_depenses_table(table_clients, transactions)
ecriture_csv(table_tri, 'clients_maj.csv', ',')

```

## 7 Applications à un autre exemple

### Exercice 6

Nous allons utiliser un fichier nommé `countries.csv` qui contient quelques données sur les différents pays du monde. En voici les premières lignes : les champs sont clairement séparés par des points-virgules.

```

iso;name;area;population;continent;currency_code;currency_name;capital
AD;Andorra;468.0;84000;EU;EUR;Euro;6
AE;United Arab Emirates;82880.0;4975593;AS;AED;Dirham;21
AF;Afghanistan;647500.0;29121286;AS;AFN;Afghani;81

```

Les données sont issues du site <http://www.geonames.org> et ont été légèrement simplifiées. La signification des différents champs est transparente (`currency` signifie devise), à part le dernier champ, nommé `capital` et dont les valeurs sont des numéros d'identifiants de villes que l'on trouvera dans un autre fichier nommé `cities.csv` que nous utiliserons dans le chapitre sur les fusions de tables.

1. On travaille toujours avec le script `'TP_Recherche_Tris_Eleve.py'` dans un IDE Python à la suite des exercices précédents. On commence par charger la table avec la fonction `lecture_csv`, attention le délimiteur de champ n'est pas le symbole `,` mais `;`.

```
table_pays = lecture_csv('countries.csv', ';')
```

2. Compléter la fonction `nombre_europe` ci-dessous en respectant sa spécification. On donne une postcondition qui doit être vérifiée.

```

def nombre_europe(table):
    """Paramètre : table de countries.csv
    Valeur renvoyée : compteur de type int représentant le nombre de
    pays du continent européen
    """

```

```

"""
compteur = 0
for enregistrement in table:
    .....
    .....
return compteur

assert nombre_europe(table_pays) == 52

```

3. Compléter la fonction `selection_europe` ci-dessous en respectant sa spécification. On donne une postcondition qui doit être vérifiée.

```

def selection_europe(table):
    """Paramètre : table de countries.csv
    Valeur renvoyée : table des enregistrements des pays du
        continent européen"""
    return .....

europe = selection_europe(table_pays)
assert len(europe) == 52 and europe[0]['name'] == 'Andorra'

```

4. Compléter la fonction `selection_europe_non_euro` ci-dessous en respectant sa spécification. On donne une postcondition qui doit être vérifiée.

```

def selection_europe_non_euro(table):
    """Paramètre : table de countries.csv
    Valeur renvoyée : table des enregistrements des pays du
        continent européen qui n'ont pas pour monnaie l'euro"""
    return .....
    .....

europe_non_euro = selection_europe_non_euro(table_pays)
assert len(europe_non_euro) == 27 and europe_non_euro[0]['name'] ==
    'Albania'

```

5. Compléter la fonction `projection_aire` ci-dessous en respectant sa spécification. On donne une postcondition qui doit être vérifiée.

```

def projection_aire(table):
    """Paramètre : table de countries.csv
    Valeur renvoyée : tableau des aires (type float) de tous les
        enregistrements"""
    return .....

```

```
assert projection_aire(table_pays)[:5] == [468.0, 82880.0,
647500.0, 443.0, 102.0]
```

6. Compléter la fonction `projection_pays_densite` ci-dessous en respectant sa spécification. On donne une postcondition qui doit être vérifiée.

```
def projection_pays_densite(table):
    """Paramètre : table de countries.csv
    Valeur renvoyée : nouvelle table avec deux attributs 'pays' et '
    densité' de population"""
    return [ {'pays' : enreg['name'],
              'densité' : .....}]
    for enreg in table]

assert projection_pays_densite(table_pays)[:3] == [{'pays': 'Andorra',
    'densité': 179.48717948717947},
{'pays': 'United Arab Emirates', 'densité': 60.033699324324324},
{'pays': 'Afghanistan', 'densité': 44.974959073359074}]
```

7. Écrire une fonction `maximum_densite` respectant la spécification ci-dessous. On donne une postcondition qui doit être vérifiée.

```
def maximum_densite(table):
    """Paramètre : table de countries.csv
    Valeur renvoyée : tuple avec le nom du pays de densité maximale
    de population et cette densité maximale"""

assert maximum_densite(table_pays) == ('Monaco', 16905.128205128207)
```

8. Écrire une fonction `population_par_continent` respectant la spécification ci-dessous. On donne une postcondition qui doit être vérifiée.

```
def population_par_continent(table):
    """Paramètre : table de countries.csv
    Valeur renvoyée : dictionnaire de clefs les identifiants des
    continents
```

```
et de valeurs les populations cumulées des pays leur appartenant
"""
```

```
assert population_par_continent(table_pays) == {'EU': 740017414,
'AS': 4119426856, 'NA': 539886359, 'AF': 1018849428, 'SA':
400143568,
'OC': 36066083}
```

9. Écrire une fonction `densite_max_top5` respectant la spécification ci-dessous. On donne une postcondition qui doit être vérifiée.

```
def densite_max_top5(table):
    """Paramètre : table de countries.csv
    Valeur renvoyée : table avec les noms et les densités des 5 pays
    les plus densément peuplées dans l'ordre décroissant des
    densités de population"""
```

```
assert densite_max_top5(table_pays) == [{'pays': 'Monaco', 'densité'
: 16905.128205128207},
{'pays': 'Singapore', 'densité': 6786.5872672152445},
{'pays': 'Hong Kong', 'densité': 6317.478021978022},
{'pays': 'Gibraltar', 'densité': 4289.846153846154},
{'pays': 'Vatican', 'densité': 2093.181818181818}]
```

## 8 Synthèse

### Synthèse

Lorsqu'une **table de données** contenue dans un **fichier CSV** est chargée dans une structure de données Python comme un **tableau de dictionnaires**, on peut la manipuler avec des requêtes de **recherche**, **d'agrégation**, de **sélection sur les lignes** ou de **projection sur les colonnes**. Il est possible également de trier les enregistrements d'une table avec la fonction de bibliothèque `sorted` en lui passant une fonction clef de tri. Elle garantit la **stabilité du tri** : les éléments égaux conservent leur ordre initial. On peut ainsi extraire des informations d'une table ou construire de nouvelles tables. En classe de terminale, nous étudierons les **bases de données** dans le modèle relationnel, qui peuvent être modélisées par des tables. Les requêtes pour les interroger seront similaires mais exprimées dans un langage spécifique, le [SQL](#).