T3.2 Opérations sur les tables

L'objectif maintenant est d'exploiter les données représentées sous forme d'une table. Puisque celle-ci est une liste de dictionnaires, on va utiliser les techniques de manipulation des listes et dictionnaires déjà rencontrées.

Pour illustrer ces différentes techniques dans les exemples et exercices suivants, on utilisera cette table{:target="_blank"}.

3.2.1 Recherche et agrégation

!!! code "Recherche" On souhaite d'abord vérifier qu'une valeur appartient à une table ou non. Pour cela il suffit de parcourir les enregistrements de la table et vérifier que la valeur recherchée est une valeur de l'enregistrement.

```
```python linenums='1' title="Recherche d'une valeur dans une table (à compléter)"
def recherche(valeur, table: list) -> bool:
    '''
    Renvoie True ou False selon que valeur apparaît ou non dans les valeurs de table.
    '''
    for e in table:
        for v in e. ... () :
            if v == ...
                return ...
    return ...

```
??? check "Correction"
    {{ correction(False,
    "
    ```python linenums='1'
    def recherche(valeur, table: list) -> bool:
        '''
        Renvoie True ou False selon que valeur apparaît ou non dans les valeurs de table.
        '''
        for e in table:
            for v in e.values() :
                if v == valeur:
                    return True
        return False
    ```
    "
) }}
```

{{ initexo(0) }}

!!! example "{{ exercice() }}" === "Énoncé" Dans la pratique, la fonction précédente n'a que peu d'intérêt. On va plutôt rechercher si une valeur est

présente **dans un champ donné**. Ainsi on n'a pas besoin de parcourir toutes les valeurs d'un enregistrement.

```
    Modifier alors la fonction précédente pour qu'elle prenne en paramètre un champ et qu'el

=== "Correction"
    {{ correction(False,
    "
    ```python linenums='1'
    def recherche(valeur, champ: str, table: list) -> bool:
        '''
        Renvoie True ou False selon que valeur apparaît ou non dans les valeurs de table.
        '''
        for e in table:
            if e[champ] == valeur:
                return True
        return False
    ```
    "
) }}
```

!!! example "{{ exercice() }}" === "Énoncé" On souhaite désormais récupérer certaines données si la valeur recherchée est trouvée.

```
    Modifier la fonction précédente (ou en créer une nouvelle) pour qu'elle renvoie la liste

    Par exemple, si `table_sw` est la table contenant les données du fichier 'sw.csv':
    ```python
    >>> recherche('Jedi', 'Statut', 'Nom', table_sw)
    ['Obi-Wan Kenobi', 'Yoda', 'Luke Skywalker', 'Rey']
    ```


=== "Correction"
    {{ correction(False,
    "
    ```python linenums='1'
    def recherche(valeur, champ1: str, champ2:str, table: list) -> bool:
        '''
        Dans la table, renvoie les valeurs de champ2 de si valeur est trouvée dans champ1
        '''
        res = []
        for e in table:
            if e[champ1] == valeur:
                res.append(e[champ2])
        return res
    ```
    "
```

````
    ) }}
````

!!! code “Agrégation” Il s’agit de récupérer une donnée statistique sur les données contenues dans une table, par exemple pour compter le nombre d’enregistrements qui satisfont à une certaine condition.

!!! example “{{ exercice() }}” === “Énoncé” Écrire une fonction `compte` qui prend en paramètre une valeur, un champ et une table et renvoie le nombre d’occurences de la valeur dans le champ de la table.

````
    Par exemple:
    ```python
    >>> compte('Droïde', 'Espèce', table_sw)
    3
    ```


=== "Correction"
    {{ correction(False,
    "
    ```python linenums='1'
    def compte(valeur, champ: str, table: list) -> int:
        '''
        Dans la table, renvoie le nombre d'occurences de valeur dans champ
        '''
        res = 0
        for e in table:
            if e[champ] == valeur:
                res += 1
        return res
    ```

    "
    ) }}
````

3.2.2 Sélection

!!! code “Sélection d’une ligne vérifiant un critère” On cherche ici à créer une nouvelle table en extrayant les enregistrements d’une table existante vérifiant un certain critère. Par exemple:

```
|Nom             |Espèce|Force|Statut|
|----------------|------|-----|------|
|Obi-Wan Kenobi|Humain|oui  |Jedi  |
|Rey           |Humain|oui  |Jedi  |
|Yoda          |Yoda  |oui  |Jedi  |
|Luke Skywalker|Humain|oui  |Jedi  |
```

```
Cela consiste donc à créer une liste en parcourant la table existante et en sélectionnant le

Par exemple, on peut créer la table précédente en sélectionnant les enregistrements dont la
```python
table_jedi = [e for e in table_sw if e['Statut'] == 'Jedi']
```
```

!!! example “{{ exercice() }}” === “Énoncé” 1. Créer la table des personnages d’espèce humaine. 2. Créer la table des personnages d’espèce humaine qui maîtrisent la force.

```
=== "Correction"
    {{ correction(False,
    "
    1.
        ```python
        table_humains = [e for e in table_sw if e['Espèce'] == 'Humain']
        ```
    2.
        ```python
        table_humains_force = [e for e in table_sw if e['Espèce'] == 'Humain' and e['Force']
        ```

    "
) }}
```

!!! code “Sélection de colonnes” On peut également avoir à extraire d’une table certaines colonnes, c’est-à-dire seulement des données d’un ou plusieurs champs. On appelle également cette opération **projection**[^1].

```
[^1]: c'est notamment le vocabulaire des bases de données, au programme de Terminale.

Par exemple, on veut créer la table suivante:

|Nom           |Espèce|
|--------------|------|
|Dark Vador    |Humain|
|Obi-Wan Kenobi|Humain|
|R2-D2         |Droïde|
| ...  |...|
|Jango Fett    |Humain|

Il s'agit donc de créer une nouvelle table (liste) dont les enregistrements sont des dictio
```

!!! example “{{ exercice() }}” === “Énoncé” 1. Compléter la fonction de projection: “‘python linenums=‘1’ title=“Fonction de projection (à compléter)” def projection(table: list, liste_champs: list) -> list: ’’’ Renvoie une table (liste de dictinonaires) des enregistrements de table ne contenant uniquement

les champs appartenant à liste_champs. ''' table_proj = ... for e in ...: table_proj.append({c: v for c, v in ... if ... }) return ...

```

```

```
    2. Utiliser cette fonction pour obtenir la table précédente.
=== "Correction"
    {{ correction(False,
    "
    1.
        ```python linenums='1'
        def projection(table: list, liste_champs: list) -> list:
            '''
            Renvoie une table (liste de dictinonaires) des enregistrements de table ne conte
            les champs appartenant à liste_champs.
            '''
            table_proj = []
            for e in table:
                table_proj.append({c: v for c, v in e.items() if c in liste_champs})
            return table_proj
        ```
    2.
        ```python
        >>> projection(table_sw, ['Nom', 'Espèce'])
        ```

    "
) }}
```

3.2.3 Tri

Trier une table selon un champ permet d'améliorer sa lisibilité selon l'information recherchée. Puisqu'une table est une liste, on pourrait utiliser les algorithmes étudiées au thème 7. Mais ceux-ci sont peu efficaces, comme on a pu le voir.

On va donc utiliser les fonctions natives de Python pour trier une liste: la méthode `sort` et la fonction `sorted`.

!!! warning "Différence fondamentale entre `sort` et `sorted`" - La méthode `sort` trie **en place**, c'est-à-dire qu'elle **modifie** la liste à laquelle on applique la méthode et renvoie `None`: `python >>> l = [4, 0, 1, 3, 2] >>> a = l.sort() >>> l [0, 1, 2, 3, 4] >>> a >>>` - La fonction `sorted` ne modifie pas la liste mais **crée une nouvelle liste**:
`python >>> l = [4, 0, 1, 3, 2] >>> a = sorted(l) >>> a [0, 1, 2, 3, 4] >>> l [4, 0, 1, 3,`

2]

Dans notre cas, on préfèrera ne pas modifier la table étudiée, mais plutôt créer une nouvelle table triée: on utilisera donc la fonction `sorted`.

!!! info “Ordre de tri” Par défaut le tri avec `sorted` se fait dans l’ordre croissant. Si on veut un ordre décroissant, il suffit de passer un argument `reverse` (un booléen) à la fonction:

```
```python
>>> sorted([4, 0, 1, 3, 2], reverse=True)
[4, 3, 2, 1, 0]
```
```

!!! info “Clé de tri” Par défaut:

```
- les objets de type `int` sont triés avec l'ordre *usuel*;
- les objets de type `str` sont triés avec l'ordre alphabétique;
- les objets de type `list` ou `tuple` sont triés avec l'ordre **lexicographique** : on trie
    ```python linenums='1'
    >>> sorted([(2, 5), (1, 3), (1, 2), (2, 7), (1, 6), (0, 10), (3, 4), (2, 0)])
    [(0, 10), (1, 2), (1, 3), (1, 6), (2, 0), (2, 5), (2, 7), (3, 4)]
    ```

Lorsqu'on souhaite trier des données différemment, il faut préciser une **clé de tri**: selo

Cette clé de tri doit impérativement **être une fonction**, que l'on précisera avec l'argume

=== "Exemple 1"
    Pour trier la liste:
    ```python
    l = [(2, 5), (1, 3), (1, 2), (2, 7), (1, 6), (0, 10), (3, 4), (2, 0)]
    ```
    sur le deuxième élément (d'indice 1), la clé de tri doit être la fonction qui, à un tupl

    ```python
    >>> def cle_tri(t: tuple) -> int:
            return t[1]
    >>> sorted([(2, 5), (1, 3), (1, 2), (2, 7), (1, 6), (0, 10), (3, 4), (2, 0)], key=cle_tr
    [(2, 0), (1, 2), (1, 3), (3, 4), (2, 5), (1, 6), (2, 7), (0, 10)]
    ```


=== "Exemple 2"
    Python ne sait pas comparer des dictionnaires (peut-être tout simplement parce que cela

    Voici une table:
    ```python linenums='1'
```
```

```
releve =[{'Nom': 'Alice', 'Anglais': 17, 'Info': 18, 'Maths': 16},
         {'Nom': 'Bob', 'Anglais': 19, 'Info': 13, 'Maths': 14},
         {'Nom': 'Carol', 'Anglais': 15, 'Info': 17, 'Maths': 19}]
```

Pour la trier sur les notes obtenues en Informatique, il faut construire la fonction sui
```python
def cle(dico):
    return dico['Info']
```

Puis on peut trier la table, en précisant par exemple l'ordre décroissant:
```python
classement_info = sorted(releve, key=cle, reverse=True)
```

!!! example "{{ exercice() }}" === "Énoncé" 1. Trier la table `table_sw` selon le champ `'Nom'`.

    2. Trier la table `table_sw` selon le champ `'Espèce'`, puis le champ `'Nom'` (rappel: F
=== "Correction"
    {{ correction(False,
    "
    Tri simple:
    ```python linenums='1'
    def tri(table: list, champ: str) -> list:
        '''
        Renvoie une nouvelle liste correspondant à la table triée selon champ
        '''
        def cle_tri(e):
            return e[champ]

        return sorted(table, key=cle_tri)

    table_triee = tri(table_sw, 'Nom')
    ```

    Tri multiple:
    ```python linenums='1'
    def tri2(table: list, champs: list) -> list:
        '''
        Renvoie une nouvelle liste correspondant à la table triée selon les champs
        de la liste champs
        '''
        def cle_tri(e):
            return tuple(e[champ] for champ in champs)
    ```

````
        return sorted(table, key=cle_tri)

    table_triee = tri2(table_sw, ['Espèce', 'Nom'])
    ```

    "
    ) }}
````

!!! example “{{ exercice() }}” === “Énoncé” On peut résoudre le pydéfi «Le lion de Némée» (cf. T2.1.11 exercice 5) en choisissant la fonction `valeur` comme clé de tri. Faites-le.

````
=== "Correction"
    {{ correction(False,
    "
    ```python linenums='1'
    divinites = 'ARTEMIS ASCLEPIOS ATHENA ATLAS CHARON CHIRON CRONOS DEMETER EOS ERIS EROS C

    def valeur(mot: str) -> int:
        '''
        Renvoie la valeur d'un mot étant la somme des valeurs des lettres le
        constituant.
        '''
        valeurs_lettres = [ord(l) - 64 for l in mot]
        return sum(valeurs_lettres)

    tri_divinites = sorted(divinites, key=valeur)

    ```

    "
    ) }}
````
````