Tri d'une table

I Introduction

Lorsque l'on manipule des données en table, il est fréquent de vouloir les trier. Si on reprend l'exemple de la table d'élèves (vu dans le cours précédent), on peut vouloir par exemple afficher la liste de tous les élèves par ordre alphabétique, ou encore trier les élèves selon une note obtenue à un contrôle. Nous nous sommes déjà intéressé aux méthodes de tri, avec les tris par sélection et les tris par insertion. Néanmoins, ces méthodes ont des défauts en fonction des applications. Elles implique, dans un premier temps, la création d'une nouvelle table triée d'après la table précédente, générant une consommation de ressource importante (en fonction de la taille de la table). De plus, en fonction du critère de sélection, il sera nécessaire de modifier le programme (ou la fonction) pour répondre au besoin. Nous allons ici plutôt nous intéresser aux fonctions de tri proposées par le langage Python, qui peuvent être utilisées pour trier des données en table. Réutilisons le tableau **eleve** suivant, en considérant chaque entrée (ligne) comme un dictionnaire.

prénom	jour	mois	année	projet
Brian	1	1	1942	programmer avec style
Grace	9	12	1906	production de code machine
Linus	28	12	1969	un petit système d'exploitation
Donald	10	1	1938	tout sur les algorithmes
Alan	23	6	1912	déchiffrer des codes secrets
Blaise	19	6	1623	machine arithmétique
Margaret	17	8	1936	atterrissage d'un module lunaire
Alan	1	4	1922	ce qu'un programmeur doit savoir
Joseph Marie	7	7	1752	programmer des dessins

II Trier des données en fonction d'une table+

Nous avions évoqué, dans un chapitre précédent, quelques opérations de tri disponibles dans Python, comme la fonction **sorted**. Celle-ci prend en argument un table et en renvoie une version triée, sous la forme d'un *nouveau* tableau.

```
>>> t = [21, 13, 34, 8]
>>> sorted(t)
[8, 13, 21, 34]
```

Ici, cette fonction a comparé les éléments, qui sont des entiers, avec l'ordre usuel. Si on lui passe un tableau contenant des éléments d'un autre type, par exemple des chaînes de caractères, alors il sera également trié, cette fois par ordre alphabétique.

```
>>> sorted(["banane", "pomme", "cassis", "fraise"])
['banane', cassis', 'fraise', 'pomme']
```

Mais si en revanche on essaye de trier le tableau contenant nos élèves avec cette fonction, on obtient une erreur.

```
>>> sorted(eleves)
Traceback (most recent call last):
  File "tri_table.py", line 16, in <module>
    affiche(sorted(t))
TypeError: unorderable types: dict() < dict()
```

Ici, le message d'erreur nous explique que Python ne sait pas comparer deux objets qui sont des dictionnaires. Pour utiliser la fonction **sorted** sur notre tableau de dictionnaires, il faut indiquer comment *se ramener* à des valeurs que Python sait comparer (nombres, chaînes de caractères, n-uplets). Pour cela, on commence par définir une fonction qui prend en argument un élève et renvoie la valeur que l'on souhaite comparer, ici le prénom.

```
def prenom():
  return x["prénom"]
```

Puis, on peut appeler la fonction **sorted** sur le tableau **eleves** en précisant qu'il faut utiliser la fonction **prenom** chaque fois que deux éléments doivent être comparés. On l'indique en passant **key=prenom** à la fonction **sorted**.

```
tri_eleves = sorted(eleves, key=prenom)
```

(Cette syntaxe est analogue à **end=""** pour la fonction **print**.) On dispose maintenant d'un nouveau tableau, **tri_eleves**, dans lequel les éléments (ici les dictionnaires) sont maintenant triés par ordre alphabétique de prénom. Si on veut trier plutôt dans l'ordre inverse, il faut passer une autre option à la fonction **sorted**, soit **reverse=True**.

III Ordre lexicographique et stabilité

Supposons que nous venons de trier notre table selon la note obtenue au dernier contrôle, par exemple pour afficher les résultats. Si beaucoup d'élèves se trouvent avoir la même note, il peut être judicieux de trier aussi par ordre alphabétique. Ainsi, il sera plus facile de retrouver son nom dans la liste. Lorsque l'on trie comme cela selon deux critères, d'abord selon le premier, puis, à valeur égale, selon le second, on appelle cela un *ordre lexicographique*. Il se trouve que la comparaison de Python a la capacité de réaliser un ordre lexicographique.

```
>>> (1, 3) < (1, 7)
True
>>> (1, 7) < (1, 4)
False
>>> (1, 7) < (2, 4)
True
```

On peut donc en tirer parti pour trier très facilement selon les deux critères précédemment mentionnés.

```
def note_puis_nom(x):
  return x["note"], x["nom"]
liste_a_afficher = sorted(eleves, key=note_puis_nom)
```

Ainsi, la fonction **sorted** va comparer les paires renvoyées par notre fonction **note_puis_nom**, ce qui aura l'effet attendu.

III.1 Stabilité

Il y a une autre façon de parvenir au même résultat. Supposons que notre liste soit déjà triée par ordre alphabétique, ce qui est tout à fait réaliste. Lorsqu'on la trie ensuite selon la note, il suffit que chaque entrée *reste dans sa position relative*. Un tri qui a cette propriété est appelé un tri *stable*. Le tri par insertion est stable, ce qui n'est pas le cas du tri par sélection. Il se trouve que la fonction **sorted** fournie par Python réalise un tri stable. On peut donc trier selon les notes notre tableau déjà trié par ordre alphabétique pour obtenir le résultat désiré. De manière plus générale, on peut appliquer successivement plusieurs tris stables à une table, selon des critères différents, et obtenir ainsi un ordre lexicographique. Il faut prendre soin, cependant, de faire les tris *dans l'ordre inverse de priorité*. Dans notre exemple, il faut ainsi trier d'abord par nom, puis par note.

IV Trier en place

Python fournit une autre fonction pour trier des données, abec la notation *tableau*.**sort()**. A la différence de **sorted**, on n'obtient pas un nouveau tableau. Au lieu de cela, le tableau est *modifié*, pour se retrouver trié au final. On dit qu'il s'agit là d'un *tri en place*. On économise ainsi de la mémoire. Les tris par sélection et par

insertion sont deux exemples de tris en place. La fonction **sort** accepte les mêmes options **key** et **reverse**que la fonction **sorted**. On peut donc trier par ordre alphabétique de prénom de la manière suivante :

```
eleves.sort(key=prenom
```

Ici, il n'y a pas de résultat à stocker dans une variable comme nous le faisions avec **sorted**. Le contenu de **eleves** a été modifié. Comme pour **sorted**, la fonction **sort** garanti un tri stable.

V Application ; recherche des plus proches voisins

L'algorithme des plus proches voisins travail à partir d'une table de données et nécessite de déterminer les k lignes "les plus proches" du point cherché. Une manière simple, bien que pas la plus efficace, d'isoler les k plus proches voisins consiste à trier l'intégralité de la table par ordre de proximité avec le point cherché, pour ensuite prendre les k premières lignes. Supposons que l'on considère une table de données **t** dont chaque ligne désigne un point avec des coordonnées (attributs **"x"** et **"y"**) et une classe ou une valeur (attribut **"valeur"**, et que l'on essaie d'estimer la classe ou valeur associée à un point A dont les coordonnées sont données par l'utilisateur. On peut, une fois les coordonnées saisies, définir une fonction **distance_de_A** qui prend en paramètre une ligne de la table et renvoie la distance entre le point décrit par cette ligne et A.

```
x = float(input("abscisse de A: "))
y = float(input("ordonnée de A: "))

def distance_de_A(p):
  return math.sqrt((p['x'] - x) ** 2 + (p['y'] - ') ** 2)
```

Il ne reste alors plus qu'à utiliser la fonction **sorted** sur notre table **t** avec comme clé de tri une fonction **distance_de_A**.

```
tri_t = sorted(t, key=distance\_de\_A)
```

On obtient ainsi une table **tri_t** avec les mêmes lignes que **t**, mais triées par ordre de proximité au point A. On a alors toute liberté d'en sélectionner les k premières lignes avec

```
plus_proche = [tri_t[i] for i in range(k)]
```

Pour ensuite déterminer la moyenne ou la majorité des valeurs de ces lignes.


Info

Mise en œuvre plus efficace. Il y a une certaine inefficacité à trier l'intégralité de la table pour n'en garder que les quelques premiers éléments, notamment si la table est très grande et le nombre de voisins qui nous intéresse petit. Pour une meilleur efficacité, on pourra opter pour une variante des algorithmes dans laquelle on ne conserve jamais plus de k éléments triés.

Savoir-faire

A retenir

Pour trier des données en table, on peut utiliser les **fonctions de tri fournies par Python**, à savoir **sorted** et **sort**. La première renvoie un nouveau tableau et la seconde trie en place. Ces deux fonction assurent la **stabilité**, c'est-à-dire qu'elles conservent l'ordre respectif des éléments égaux.