T.P. 1 : Dictionnaires

1 Exercices proche du cours

Les deux exercices de ce § 1 sont issus d'un T.P. de S. Decroix en PCSI l'an dernier.

1.1 Relévés de notes

On dispose des relevés de notes suivants pour la classe MP2 :

```
MP2={ "Alain":{"Maths":[12,3,14],"P.C.":[17,5,6],"Anglais":[1,7]},
"Benoit": {"Maths":[11,13,17],"P.C.":[8,1,4],"Anglais":[9,12,18] },
"Celine": {"Maths":[17,15],"P.C.":[12,14,9],"Anglais":[9,3,12] },
"David": {"Maths":[13,15,8],"P.C.":[7,15,10],"Anglais":[19,13,15]}}
```

a) Proposer pour chaque question une commande pour afficher la réponse :
   i) Les notes en maths de Benoît.
   ii) Le nombre d'élèves.
   iii) Si Ludivine est dans la classe

b) Ajouter Elaine qui a 20 et 16 en Maths, 13 et 15 en P.C. et 12,14 et 17 en Anglais.

c) Céline n'a pas eu 9 mais 15 en P.C., corriger sa note.

d) Enlever la matière `Anglais` pour David, qui a choisi de changer de `LV1`.

e) Écrire une fonction `moyenne(table,nom_eleve,matiere)` qui donne la moyenne de l'élève dans la matière donnée.

f) Proposer une fonction `moyenne_generale(table,nom_eleve)` qui donne la moyenne générale de l'élève.On affectera les coefficients suivants : Maths 6, PC 5 , Anglais 3 (on ne compte pas l'ITC ;-)), on veillera à prendre en compte le cas où un étudiant n'aurait bizarrement pas été évalué dans l'une ou l'autre des matières.

   Vérification : on trouvera `8.3` pour la moyenne générale d'Alain et `11.39` pour celle de David (sans l'anglais).

g) Créer une fonction `moyenne_classe(table,matiere)` qui donne la moyenne des élèves dans la matière donnée.

   Vérification : on trouvera `13.86` pour la moyenne de Maths.

1.2 Passagers du titanic, avec ouverture de fichier de csv

Un fichier `csv` (comma separated values) est un fichier de données typiquement exploitable par un tableur. La première ligne du fichier titanic.csv donne ici les noms des colonnes. Le module **csv** (aucune connaissance exigible là-dessus) permet ici d'utiliser la commande `DictReader` qui crée un objet qu'on a choisi d'appeler `reader`, d'un type spécifique, dont on peut parcourir les lignes pour transformer *chaque ligne* du fichier `csv` en un *dictionnaire* où les clefs sont les noms des colonnes du fichier csv initial et les valeurs bien sûr les valeurs de chaque ligne.

```
import csv

reader = csv.DictReader(open('/Users/romain
/Documents/MPSI/Informatique/CSV/tpi_titanic.csv','r')

titanic = []
for row in reader:
    titanic.append(dict(row))
```

a) Déterminer le nombre de survivants.

b) Déterminer le nombre de femmes voyageant en 1ère classe qui n'ont pas survecu.

c) Définir et compléter un dictionnaire `dstat` dont les clés sont les 3 classes possibles de passagers (1ère classe, 2ème classe, 3ème classe) et dont les valeurs sont des listes donc les items sont : `classe:[nb de passagers de cette classe, nb de survivants de cette classe]`

2 Temps de recherche de clefs dans un dictionnaire

Dans ce paragraphe, on va voir la puissance du `in` sur les clefs d'un dictionnaire : on a annoncé dans la partie cours que ce temps est toujours en $O(1)$ quelle que soit la longueur du dictionnaire, on va ici le confirmer expérimentalement.

2.1 Comparaison entre listes, tableaux numpy et dictionnaires

Pour cela, on va comparer les temps de recherche dans une liste aléatoire, d'un tableau numpy aléatoire et d'un dictionnaire aléatoire.

• Pour mesurer les temps d'exécution, on utilisera le module `time` comme suit :

```
import time
t0=time.time()
# exécution de l'algorithme
t1=time.time()
print(t1-t0)
```

• Pour fabriquer des entiers aléatoires, on utilisera le module `random` et la fonction `random.randint(a,b)`.

a) Ecrire une fonction `liste_aleatoire(N: int,a : int,b : int)-> L:list`, qui fabrique une liste de `N` entiers aléatoires pris chacun entre `a` et `b`.

b) Ecrire une fonction `tableau_aleatoire(N: int,a : int,b : int)-> T : np.ndarray` qui fait la même chose mais commençant par créer un tableau numpy de taille N rempli de zéros, puis en le remplissant d'entiers aléatoires. Pour créer un tableau numpy rempli N de zéros entiers, il suffit de faire :

```
import numpy as np
T=np.zeros(N, dtype=int)
```

c) Ecrire enfin une fonction `dico_aleatoire(N: int,a : int,b : int)-> D : dict` qui stocke ces N entiers aléatoires comme `clefs` (non nécessairement distinctes) dans un dictionnaire `D` avec pour valeur associée à la clef par exemple le numéro du tour de boucle où cette clef est rajoutée.

Remarque : à la fin le nombre d'items du dictionnaires est inférieur ou égal à `N` à cause des entiers aléatoires *égaux* que l'on peut tirer.

d) Pour $a, b = 1, 100000$ et les valeurs de N de 100 à 20100 avec un pas de 2000 exécuter les trois fonctions précédentes, pour fabriquer respectivement `L,T,D` et déterminer à chaque tour de boucle le temps d'exécution de la recherche :

```
for k in range(1,10000):
    test=k in L # et de même avec T et D
```

dans chacun de ces trois objets, et afficher le résultat sous la forme de trois courbes avec, en abscisse N et en ordonnée le temps de recherche respectif pour L, T, D.

e) Commenter le résultat obtenu

2.2 Utilisation des dictionnaires pour accélérer une recherche dans une liste

Par rapport au paragraphe précédent, on cherche maintenant à fabriquer une liste d'entiers aléatoires *deux à deux distincts*.

a) Ecrire une fonction `liste_alea_distinct(N,a,b)` qui renvoie une liste de `N` entiers aléatoires deux à deux distincts pris entre a et b. Pour s'assurer qu'ils sont distincts, on pourra à chaque tour de boucle tester avec `in` si le nouvel entier aléatoire proposé est dans la liste courante.

b) Déterminer le temps de calcul de la fonction précédente avec `N,a,b=10**5,10**9,10**10`.

c) On va améliorer sensiblement la fonction du a) à l'aide d'un dictionnaire : pour cela créer une fonction `liste_alea_avec_dico(N,a,b)` qui renvoie la même chose, mais gère, en plus de la liste `L` dont la valeur sera renvoyée, un dictionnaire `D` qui, au départ est vide, et auquel on rajoute chaque entier aléatoire comme *clef* en testant sa présence *comme clef du dictionnaire* plutôt que comme entrée de la liste.

d) Refaire le test du b) avec la fonction du c).

3 Mémoïsation pour les algorithmes récursifs à l'aide de dictionnaires

3.1 Mémoïsation avec une liste sur l'exemple de Fibonacci

a) Rappel, que dire de cet algorithme en terme de nombre d'appels récursifs ?

```
def Fibo(a,b,n):
    if n==0:
        return a
    elif n==1:
        return b
    else :
        return Fibo(a,b,n-1)+Fibo(a,b,n-2)
```

b) Pour éviter ce problème : on peut stocker au fur et à mesurer les résultats obtenus dans une liste Python. Ecrire une fonction récursive `FibolisteR(n,L)` qui prend comme argument un entier `n` et une liste `L` qui va contenir les nombres de Fibonacci calculés (au minimum lors de l'appel principal on met les deux termes initiaux dans `L`). La fonction va compléter la liste `L` pour qu'elle contienne tous les termes de la suite jusqu'au n-ième et renvoie la valeur `L[n]`.
L'idée est de limiter les appels récursifs en ne recalculant pas les nombres qui sont déjà dans `L`. On mettra donc un test conditionnel `if len(L) > n :` (ce qui signifie que le nombre F_n a déjà été calculé) : dans ce cas la fonction retournera `L[n]`. Sinon la fonction fera l'appel récursif pour calculer la valeur à mettre en `L.append()`.

3.2 Mémoisation avec un dictionnaire pour le calcul des coefficients binomiaux

Une façon de calculer les coefficients binomiaux est d'utiliser la formule de récurrence :

$$\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$$

Cette méthode est horrible en si on l'utilise directement récursivement (en impératif ok). Mais on peut sauver la récursion avec la mémoïsation, ici avec un dictionnaire de dictionnaires :

```
def coeff_binomiaux(n,p,D={}):
    if p>n:
        return
```

```
elif p==n or p==0:
    return
else :
# ici 1<=p<=n-1
# on va éviter les appels récursifs inutiles
    if n-1 in D and p-1 in D[n-1]: # ici D[n-1] est aussi un dictionnaire !
        r=
    else:
        r=
    if n-1 in D and p in D[n-1]:
        r=r+
    else:
        r=r+
# avant de renvoyer le résultat on le stocke
if n in D:
    D[n][p]=r
else:
    D[n]={p:r}
return r
```

Travail à faire : compléter le code précédent en comprenant que `D` est un dictionnaire dont les clefs sont les entiers `n` et les valeurs `D[n]` sont elles-même un dictionnaire de clés $p = 1, \ldots, n-1$ avec `D[n][p]` contenant la valeur du binomial voulu.