I Définition, premiers exemples

Un *dictionnaire* est un objet informatique d'un type nouveau (`dict` en Python) permettant d'associer des *clés* à des *valeurs*. On parle de *table d'association*. Il s'agit mathématiquement du graphe d'une application qu'on définit point par point.

Exemple et syntaxe :
On considère la fonction f définie sur $[\![1,5]\!]$ par : $f(1)=10,\ f(2)=13,\ f(3)=7,\ f(4)=6,\ f(5)=7$.
À cette fonction correspond le dictionnaire `f` défini en *Python* par :

```
f = {1:10, 2:13, 3:7, 4:6, 5:7 }
```

Les éléments de $\mathcal{D}_f = [\![1,5]\!]$ sont les *clés* du dictionnaire.
Les éléments de $f([\![1,5]\!])$ sont les *valeurs* du dictionnaire.
Comme pour une fonction f, un éléments de $\mathcal{D}_f$ ne possède qu'une image :

Les clés d'un dictionnaire sont uniques.

Un dictionnaire permet d'associer entre eux des objets de nombreux types informatiques :

`tel = {'Adrien':'01.41.42.13.56', 'Bernard':'03.14.15.92.65', 'Charlotte':'02.71.82.81.82'}`
est un dictionnaire où les clés comme les valeurs sont des chaînes de caractères (`str`).

`notes = {'Adrien':[12,11,15], 'Bernard':[8,14,11], 'Charlotte':[11,13,14]}`
est un dictionnaire où les valeurs sont des listes (d'entiers).

`NbePremier = {2:'premier', 3:'premier', 4:'non premier'}`

`{}` est le dictionnaire vide.

II Opérations sur les dictionnaires

∗ Appel d'une valeur

Même syntaxe que pour les listes (une liste de taille n est comme un dictionnaire de clés $0,\dots,n-1$) :

```
f[1]
>>> 10
```

Si la clé n'est pas trouvée, un message d'erreur est renvoyé : `key error`.

∗ Taille d'un dictionnaire

Comme pour les listes, `len(f)` renvoie la taille (nombre de clés) du dictionnaire `f`.

∗ Présence d'une clé dans l'ensemble des clés (domaine)

`cle in f` teste si la clé appartient à l'ensemble des clés d'un dictionnaire (renvoie un booléen).

∗ Ajout d'une association

`f[6] = 4` ajoute la clé 6 au dictionnaire `f`, et lui associe la valeur 4.

∗ Modification d'une valeur

`f[6] = 3` modifie la valeur associée à la clé 6.

∗ Parcours des clés

`for c in f :` crée une boucle où chaque clé sera considérée l'une après l'autre.
Remarque : Les listes, matrices et les dictionnaires ne peuvent eux-mêmes être choisis comme clés.

∗ Parcours des valeurs

`for y in f.values() :` crée une boucle où chaque valeur sera considérée l'une après l'autre.
Remarque : Si une valeur apparaît plusieurs fois dans le dictionnaire, elle sera considérée plusieurs fois dans la boucle.

∗ Parcours des associations `clé:valeur`

`for x,y in f.items() :` crée une boucle où chaque couple (clé,valeur) sera considéré.

∗ Copie d'un dictionnaire

Comme pour les listes, l'instruction : `f2 = f` crée un *clône* du dictionnaire `f`. Si `f` est modifié, alors `f2` le sera aussi.
Pour créer une copie indépendante, on doit utiliser : `f2 = f.copy()` , ou : `f2 = dict(f)`.
Rappel pour les listes : `L2 = L.copy()`, ou `L2 = list(L)`, ou `L2 = L[:]`.

III Exercices

1 Éliminer les valeurs multiples d'une liste

Soit la liste : `L = [ rd.randint(1,1000) for _ in range(100) ]`.
Cette liste contient 100 entiers aléatoires, équiprobablement choisis entre 1 et 1000.
On souhaite enlever les valeurs en double (s'il y en a).

1. Écrire une fonction `tableOccurence(liste)`, renvoyant un dictionnaire dans lequel tous les éléments distincts de la liste forment l'ensemble des clés, auxquelles on associe la valeur 1.
2. Écrire une fonction `listeSansDouble(liste)` répondant au problème.

2 Déterminer la valeur la plus fréquente d'une liste

Soit la liste : `L = [ rd.randint(1,100) for _ in range(1000) ]`.
Cette liste contient 1000 entiers aléatoires, équiprobablement choisis entre 1 et 100.
On souhaite connaître la valeur la plus fréquente de cette liste.

1. Écrire une fonction `tableComptage(liste)`, renvoyant un dictionnaire dans lequel tous les éléments distincts de la liste forment l'ensemble des clés, auxquelles on associe leur nombre d'apparitions dans la liste.
2. Écrire une fonction `plusFrequent(liste)` répondant au problème.

3 Tri par comptage, solution avec dictionnaire

Utiliser la fonction `tableComptage` pour définir une fonction `triComptage(liste)` renvoyant la liste triée obtenue à partir d'une liste d'entiers passée en argument.
On pourra créer une liste contenant toutes les clés de `tableComptage(liste)` pour repérer la plus petite et la plus grande clé, puis utiliser une boucle dont l'indice varie entre ces plus petite et plus grande clés.

4 Un dictionnaire correspond-il à une application injective ?

Rappel : Une application est *injective* si et seulement si elle ne prend jamais deux fois la même image :

$$\forall x, x' \in \mathcal{D}_f,\ x \neq x' \Rightarrow f(x) \neq f(x')$$

1. Écrire une fonction `cleValeur(dico)` d'argument un dictionnaire `dico` et renvoyant un dictionnaire `dico2` dans lequel les clés sont les valeurs distinctes de `dico`, auxquelles on associe le nombre de fois où ces valeurs sont présentes dans `dico`.
2. En déduire une fonction `injectif(dico)` renvoyant un booléen répondant à la question.
3. Tester votre fonction avec plusieurs `tableOccurence(liste)` comme défini à la question **1.**

5 Construction d'un index d'un texte

Rappel : La méthode `split` sépare une chaîne de caractères en sous-chaînes, renvoyées dans une liste.

```
'Bonjour tout le monde!'.split()            # Par défaut, la séparation se fait en chaque espace.
>>> ['Bonjour', 'tout', 'le', 'monde!']
'Bonjour tout le monde!'.split('o')         # On choisit ici le caractère séparateur 'o'
>>> ['B', 'nj', 'ur t', 'ut le m', 'nde!']
```

À partir d'un texte, on veut créer un dictionnaire qui en sera l'index : tous les mots distincts du texte seront les clés du dictionnaire, et les valeurs associées seront les listes de positions que ces mots occupent dans le texte.

Exemple : À partir du texte *'L'image d'un intervalle par une fonction continue est un intervalle'*
on souhaite créer l'index :
`index = {"L'image" : [0], "d'un" : [1], 'intervalle' : [2,9], 'par' : [3], ... }`
Pour tester, on pourra utiliser le texte enregistré dans le dossier-classe sous le nom : *Declaration.txt*

Rappel : on extrait le contenu (sous forme d'une chaîne de caractères) d'un fichier-texte grâce aux instructions :

```
T = open('Declaration.txt', 'r')
chaine = T.read()
T.close()
```

Écrire une fonction `index(chaine)` renvoyant un dictionnaire répondant au problème.

6 Reconstruction d'un texte à partir d'un index

Écrire une fonction `texte(dico)` renvoyant une chaîne de caractères dont l'index est `dico`.

7 Analyse d'un texte

Écrire une fonction `occurence(chaine)` qui, à partir d'une chaîne de caractères, renvoie une liste des mots distincts présents dans le texte, du plus fréquent au moins fréquent.

8 Mémoïsation à l'aide d'un dictionnaire

On a vu que la suite récurrente : $\begin{cases} f_0 = 0,\ f_1 = 1 \\ \forall n \geqslant 0,\ f_{n+2} = f_{n+1} + f_n \end{cases}$ peut être définie de façon récursive par :

```
def f(n) :
    if n == 0 : return 0
    if n == 1 : return 1
    return f(n-1) + f(n-2)
```

mais que le nombre d'opérations pour calculer $f(n)$ croît alors exponentiellement.
On utilise un dictionnaire pour réduire le temps de calcul : toute valeur calculée à une étape quelconque est enregistrée dans un dictionnaire.
Compléter le script suivant, qui est une version récursive mémoïsée de la suite (f_n) :

```
def f(n) :
   dico = {}
   def g(k) :
      if k in dico : return ...
      if k == 0 : return ...
      if k == 1 : return ...
      dico[k] = g(...)+g(...)
      return ...
   return g(n)
```

9 Suites de Syracuse, version mémoïsée

On rappelle qu'une suite de Syracuse de premier terme $u_0 = a \in \mathbf{N}^\star$ est définie par

la relation de récurrence : $\forall n \geqslant 0,\ \begin{cases} u_{n+1} = \dfrac{u_n}{2} & \text{si } n \text{ est pair,} \\ u_{n+1} = 3u_n + 1 & \text{sinon.} \end{cases}$

On admet (ce qui n'est pas prouvé) que pour tout $a \in \mathbf{N}^\star$, il existe un rang n_0 tel que la suite de Syracuse de premier terme a vérifie : $u_{n_0} = 1$.
On appelle *vol* d'une suite de Syracuse de premier terme a le plus petit indice n tel que $u_n = 1$.
On le note alors vol(a).
On souhaite connaître le vol de toutes les suites de Syracuse de premier terme $a \in [\![1, N]\!]$, où $N \in \mathbf{N}^\star$.
Pour économiser du temps de calcul, on observe que si une valeur est déjà apparue, il est inutile de refaire tous les calculs.
Par exemple, l'étude du cas $a = 3$ donne : vol(3)= 7, mais aussi vol(10)= 6, vol(5)= 5 etc
On peut donc mémoriser les valeurs de vol dans un dictionnaire, où les clés seront les entiers rencontrés lors du calcul de suites de Syracuse, et les valeurs les vols correspondants.
En s'inspirant de l'exercice **8**, écrire une fonction `syracuse(N)` renvoyant une liste de tous les couples $(a, vol(a))$ pour $a \in [\![1, N]\!]$, en utilisant un dictionnaire enregistrant ces couples.