Ensembles, b-uplets et dictionnaires

I Introduction

On s'intéresse à écrire un programme qui vérifie dans un jeu de donnée que deux personnes sont nées le même jour (nombres réels compris entre 1 et 365). Lorsque l'on réunit 23 personnes, il y a une chance sur deux pour que ce soit le cas. On appel ce phénomène le *paradoxe des anniversaires*. Avec l'approche que nous avons vu dans la séquence précédente, nous pourrions écrire un tableau contenant l'ensemble des dates d'anniversaire, puis parcourir ce tableau à la recherche d'un doublon. Cette méthode, pour 23 personnes, représente $1 + 2 + ... + 22 = 253$ comparaisons. On comprend alors que cette approche n'est valable que pour un faible nombre d'entrée dans notre table. Il existe d'autres approches beaucoup plus pratiques, et qui permettent de garder le "sens" des données renseignées. Passons-les en revue.

II Les ensembles de Python

Notions sur les ensembles

On construit un ensemble en Python en écrivant ses éléments entre accolades et en les séparant par des virgules. On peut par exemple construire l'ensemble **s** contenant les entiers 2, 3, 5 et 7 comme ci-dessous.

```
>>> s = {2, 3, 5, 7}
```

L'ordre des éléments d'un ensemble n'importe pas. En effet, contrairement à un tableau, les éléments d'un ensemble ne sont pas ordonnés. Comme pour les tableaux, on peut obtenir la taille d'un ensemble en appelant la fonction prédéfinie **len**.

```
>>> len(s)
4
>>> 5 in s
True
>>> 4 in s
False
```

On peut tester si un résultat appartient à un ensemble avec la construction **in** de Python, dont le résultat sera un booléen. Même si c'est le mot clef **in** qui est utilisé dans les deux cas, les constructions n'ont rien à voir.
On peut modifier le contenu d'un ensemble en ajoutant ou en supprimant un élément avec les fonction **s.add(...)** et **s.remove(...)**. Il est possible de définir de même un ensemble vide, qui sera rempli par la suite, avec **set()**.

```
>>> ens = set()
>>> ens.add(23)
>>> ens.add(23)
>>> ens.remove(5)
>>> ens
{23}
```

Il est de même possible de construire un ensemble depuis un tableau,

```
>>> l = [3, 1, 4, 7, 2, 9]
>>> s = set(l)
>>> print(s)
{1, 2, 3, 4, 7, 9}
```

ou encore à l'aide d'une construction par compréhension analogue à celle des tableaux.

```
>>> {x*x for x in range(10)}
{0, 1, 4, 64, 36, 9, 16, 49, 81, 25}
```

Finalement, les opérations mathématiques d'**union**, d'**intersection** ou de **différence**, appelées *opérations ensemblistes*, permettent de même la construction d'ensemble.

```
>>> s = {1, 3, 9}
>>> u = {1, 2, 4, 8}
```

```
>>> s | u           # union avec le symbole du "6"
{1, 2, 3, 4, 8, 9}
>>> s & u           # intersection
{1}
>>> s - u           # différence
{9, 3}
>>> s ^ u           # ou exclusif
{2, 3, 4, 8, 9}
```

Ces *opérations ensemblistes* renverront de nouveaux ensembles, sans modifier leurs arguments.

Exemple sur les ensembles

En optant pour une approche par ensemble, on pourrait donc réaliser le programme suivant qui recherche un doublon dans un tableau.

```
def doublon(t):
  """Recherche de doublons dans un tableau t"""
  s = set()
  for x in t:
    if x in s:
      return True
    s.add(x)
  return False
```

Ici, les différents éléments du tableau t sont ajoutés à l'ensemble **s** au fur et à mesure de son parcours, et ce jusqu'à trouver un doublon, dans ce cas on retourne *True*. Autrement, si aucun doublon n'est trouvé, on retourne *False*.

III Les n-uplets

notions sur les n-uplets

Un **n-uplet** est un ensemble de valeurs écrites **entre parenthèses** et séparées par des **virgules**. On peut y préciser tout type de données, qui seront accessibles via un indice. Ainsi, en précisant toutes les données d'un membres, on arrive à l'écriture suivante.

```
>>> x = ("Claudes", 12, 1, 1984)
>>> len(x)
4
>>> print(x[0])
Claudes
```

On pourrait alors facilement regrouper la totalité des membres dans un **tableau de n-uplets** (ici des quadruplets puisque chaque élément est composé de quatre entrées).

```
membres = [("Claudes", 12, 1, 1984), \
           ("Alexandra", 24, 7, 1972), \
           ("Maxence", 7, 2, 1996), \
           ("Basile", 3, 12, 1978),]
```

Pour parcourir l'ensemble des éléments, on utilisera donc une structure de parcours par boucle for, en prenant soin de ne pas comparer les champs n'étant pas des entiers. On pourra alors récupérer les données de chacune des composantes dans une variable spécifique en les assignant simultanément lors du parcours.

```
n, j, m, a = donnee
```

Cette approche permet donc de récupérer dans **n** la composante donnee[0], puis dans **j** la composante donnee[1], etc.

Les n-uplets nommés

En regardant les données précisées dans notre tableau *membres*, il peut être difficile de se rappeler la signification de chacun des champs. Par exemple, un anglo-saxon comprendra, pour la troisième entrée, que Maxence est né le 2 Juillet, alors qu'on souhaitait spécifier le 7 Février. Pour palier à ce problème, on utilise les n-uplets nommés. Les **n-uplets nommés** sont écrits entres **accolades**, et chaque composante est une **paire d'un nom et d'une valeur**, séparés par un **deux-points**. Pour le cas de l'entrée Claudes, on aurait dont l'écriture suivante.

```
>>> x = {"nom":"Claudes", "jour":12, "mois":1, "annee":1984}
```

On pourra bien évidement vérifier la longueur de notre n-uplet nommé à l'aide de la fonction *len(x)* qui ne retournera la valeur **4**. Néanmoins, il n'est plus possible ici de cibler une composante par son indice, mais il faudra préciser le nom de la composante

```
>>> print(x["nom"])
Claudes
```

Cette méthode est très pratique pour retourner une caractéristique spécifique d'un élément. Dès lors que l'on connaît le nom de la composante, on pourra récupérer la valeur associée à l'entrée.

IV Les dictionnaires

Généralités sur les dictionnaires

Les n-uplets sont en faite une construction particulière des dictionnaires, lesquels sont structurés par l'association d'une *valeur* à une *clé*. Un dictionnaire peut être construit en donnant explicitement toutes ses entrées, mais aussi vide, en notant , puis en y ajoutant des valeurs par une affectation tel que **dic[clé] = valeur**.

```
>>> dic = {}
>>> dic["Bob"] = "Père de Claude"
```

L'ordre d'insertion des éléments n'est pas imporant. En particulier, le dictionnaire est affiché en présentant les clés dans un ordre arbitraire, qui n'est ni l'ordre d'insertion, ni l'ordre alphabétique. On accès alors à une valeur avec la construction **dicclé** et on peut tester si le dictionnaire possède une entrée pour une clé spécifique via **clé in dic**. On pourra, comme pour le tableau, modifier une valeur en remplaçant la valeur par une nouvelle affectation, mais également supprimer une entrée

```
>>> dic["Hector"] = "Vendeur"
>>> dic["Hector"] = "Vendeur de légumes"
>>> dic["Hector"]
Vendeur de légumes
>>> del dic["Hector"]
```

Parcours d'un dictionnaire

On peut parcourir toutes les clés d'un dictionnaire avec la boucle **for**. L'ordre de parcours sera encore une fois arbitraire.

```
for clé in dic:
  print("la clé", clé, "est associée à la valeur", dic[clé])
```

On peut également obtenir un tableau contenant toutes les clés de *dic* avec la construction **list(dic.keys)**. Dès lors, chacune des clés n'apparaîtra qu'une fois et une seule. Par la même approche, on peut construire un tableau avec les différentes valeurs à l'aide de la construction **list(dic.values())**. Cette fois si, si plusieurs clés ont la même valeur, le tableau contiendra autant de fois cette valeur. Finalement, on peut construire un tableau contenant les couples clés et valeurs ensemble à l'aide de la construction **list(dic.items())**

```
>>> dic = {"a" = 1, "c" = 3, "d" = 1}
>>> list(dic.keys())
```

```
[c, a, d]
>>> list(dic.values())
[1, 1, 3]
>>> list(dic.items())
[('a', 1), ('d', 1), ('c', 3)]
```

Savoir-faire **A retenir**

Python fournit des notions d'**ensemble**, de **n-uplet**, et de **dictionnaire** pour structurer l'information. Les dictionnaires **généralisent la notation de tableau** en conservant la même syntaxe. Les ensembles et les dictionnaires ofrrent des opérations **très pratiques et efficaces**.

V Exercice de fin de chapitre

On cherche à réaliser un programme comptant le nombre d'occurrence des mots dans un texte. On utilisera le texte *Le tour du monde en 80 jours* de *Jules Vernes* disponible sur le site **http ://www.gutenberg.org** au format **.txt** (référencé comme : *Plain Text UTF-8*). Une fois récupéré, on utilisera la fonction ci-dessous pour extraire chaque mot du texte dans un tableau **tab**

```
def extraction(source):
    f = open(source)
    tab = f.read().split()
    f.close
    return tab
```

Exercice

Exercice 1

Écrire une fonction **occurrence()** qui prend en entrée une table et qui retourne un **dictionnaire** ayant pour **clé** les différents mot du texte, et pour **valeur** le nombre de fois où ce dernier apparaît.

Exercice

Exercice 2

Écrire une fonction **le_plus_long()** qui prend en entrée un dictionnaire et qui retourne la clé la plus utilisée ainsi que son nombre d'apparition (la valeur associée).
Vous devriez obtenir ici la clé *de* avec *2796* occurrences.