Les dictionnaires en Python

1. Avant de commencer : tableaux simultanés

1.1. Des listes avec le même index

Premier exemple

On considère les deux listes suivantes :

```
animaux = ['Vache', 'Chien', 'Chat', 'Poule', 'Ane', 'Cochon', 'Paon']
cris = ['Meuh', 'Wouf', 'Miaou', 'Cot-cot', 'Hi-Han', 'Gruik', 'Leon']
```

Énoncé

Écrire une fonction `afficheCri` qui prend comme argument le nom d'un animal, et renvoie :

- soit une chaine de caractère telle que dans l'exemple ci-dessous :

```
>>> afficheCri("Vache")
"Le cri de l'animal Vache est Meuh !"
```

- soit la chaine `"Je ne connais pas le cri de Vache"` dans le cas où l'animal n'est pas dans la liste `animaux`.

Réponse

Le principe est de parcourir par indice la liste `animaux`, et de comparer avec l'animal cherché. Comme les indices correspondent entre les listes `animaux` et `cris`, il suffira alors de renvoyer le cri correspondant à l'indice trouvé. Si la boucle se termine, c'est que l'animal ne se retrouve pas dans cette liste, et qu'on renvoie alors la phrase correspondante :

```
def afficheCri(animal) :
    for i in range(len(animaux)) :
        if animaux[i] == animal :
            return cris[i]
    return f"Je ne connais pas le cri de {animal}"
```

Si cette méthode est fonctionnelle, il faut quand même se poser la question de son **efficacité**, et plus particulièrement de son **efficacité en temps**.

Mesurer un temps d'exécution avec `timeit`

Le module `timeit` est un module particulièrement utile pour mesurer des temps d'exécution. Il peut être importé par la commande suivante, que je vous conseille de mettre dans l'entête de votre fichier :

```
import timeit
```

Une fois importé, nous allons utiliser la fonction `timeit` du module pour calculer le temps d'exécution d'une instruction par la commande suivante :

```
timeit.timeit(lambda : afficheCri('Vache'), number = 1000)
```

L'instruction `afficheCri('Vache')` sera alors exécutée 1000 fois, et le temps moyen d'exécution sera affiché (mais sans le résultat d'exécution de la fonction...).

Mesurer le temps

Quel est le temps d'exécution de la fonction `afficheCri` ? Testez sur plusieurs animaux.

1.2. Et avec plus de données ?

Que se passe-t-il si les tableaux sont plus long ?

Deuxième exemple

Enoncé

1. Téléchargez le fichier suivant, et sauvegardez le dans le même dossier que votre code Python actuel.
2. Copiez-collez la fonction `makebigArray` suivante.

```
def makeBigArray() :
    import csv
    with open("Long_Dico.csv","r",encoding = "utf8") as file :
        dicReader = csv.DictReader(file, delimiter=';')
        etablissements =[]
        GPS = []
        for line in dicReader :
            etablissements.append(line['Nom'])
            GPS.append(line['GPS'])
    return etablissements, GPS
```

3. Copiez-collez ensuite la ligne suivante dans votre code, **après la fonction précédente** :

```
etablissements, GPS = makeBigArray()
```

Le code ci-dessous crée deux listes : * la première contient tous les différents établissements présents sur **ParcourSup**, * la deuxième contient les **coordonnées GPS** de chacun de ces établissements.

4. Vérifiez que les listes `etablissements` et `GPS` sont bien de la même dimension.
5. A partir de la fonction `afficheCri`, créez une fonction `afficheGPS` afin qu'elle permette de récupérer les coordonnées GPS d'un établissement dont on a saisi le nom, et qu'elle renvoie `None` si l'établissement n'est pas présent.
6. Testez ensuite la ligne suivante :

```
timeit.timeit(lambda : afficheGPS('Lycée Auguste Pavie (Guingamp - 22)'), number = 1000)
```

Quel est le temps d'exécution ?

7. Testez ensuite la ligne suivante :

```
 timeit.timeit(lambda : afficheGPS('Pavie'), number = 1000)
```

Quel est le temps d'exécution ? Pourquoi est-il plus long que précédemment ?

Limite des listes

Les listes sont des objets très pratiques, mais possédant des limites, en particulier concernant la recherche d'éléments :

- si la liste n'est pas triée, la recherche se fait en $\mathbb{O}(n)$, ce qui signifie que la recherche prend **un temps proportionnel à la longueur de la liste** ;
- si la liste est triée, il est possible d'utiliser des algorithmes de recherches rapides, comme la **recherche dichotomique** (au programme de terminale), qui permettent de rechercher en un temps plus court, en $\mathbb{O}(log(n))$.

Quoi qu'il en soit, **ce temps est très long** comparativement au **temps d'accès** à un élément, quand on connait son indice ! En effet, **peu importe la taille de la liste**, le temps d'accès à un élément reste constant, il est en $\mathbb{O}(1)$.

2. Les dictionnaires en Python

2.1. Premiers aperçus des dictionnaires

Dictionnaires en Python

Un **dictionnaire** est une structure déclarée **entre accolades**, où chaque déclaration est une paire de type `clé : valeur`. La seule limite imposée dans les types des objets `clé` et `valeurs` est que la `clé` doit être d'un *type non-mutable* (un entier, une chaîne de caractères, un tuple, ... mais par contre pas une liste !).

exemple

On considère le code suivant :

```
cris = {"Vache" : "Meuh",
           "Chien" : "Wouf",
           "Chat" : "Miaou",
           "Poule" : "Cot-cot",
           "Ane" : "Hi-Han",
           "Cochon" : "Gruik",
           "Paon" : "Leon"
           }
```

La variable `cris` est associée à un objet de type **dictionnaire**, ce qu'on peut vérifier en testant :

```
type(cris)
```

Terminal de test

Par exemple dans le dictionnaire `cris`, on trouve la paire `"Chat" : "Miaou"` où :

- `"Chat"` est la clé (de type chaine de caractères `str`) ;
- `"Miaou"` est la valeur (aussi de type chaine de caractères `str`).

Accès à un élément

Pour accéder à une valeur `v` à partir de la clé `c` d'un dictionnaire `d`, il suffit d'utiliser la notation suivante :

```
>>> d[c]
v
```

Ce qui signifie que l'appel `d[c]` renvoie la valeur `v`.

Exemple

1. Pour obtenir le cri de la vache, il suffit d'utiliser :

```
>>> cris["Vache"]
'Meuh'
```

2. Pour obtenir le cri du cochon, il suffit d'utiliser :

```
>>> cris["Cochon"]
'Gruik'
```

3. Par contre, si on utilise **une clé qui n'existe pas**, on obtient l'erreur de type `KeyError` suivante :

```
>>> cris['Perroquet']

Traceback (most recent call last):
File "<pyshell>", line 1, in <module>
KeyError: 'Perroquet'
```

ce qui signifie que `"Perroquet"` n'est pas une clé valide du dictionnaire.

Terminal de test

2.2. Manipulation des dictionnaires

Création d'un dictionnaire vide

Il existe deux possibilités pour créer un dictionnaire vide :

```
>>> dico1 = {}
>>> dico2 = dict()
```

Ajout d'un élément

Pour ajouter un couple clé/valeur à un dictionnaire, rien de plus simple :

```
>>> cris['Girafe'] = 'Tic-Tic'
>>> cris
{'Vache': 'Meuh', 'Chien': 'Wouf', 'Chat': 'Miaou', 'Poule': 'Cot-cot', 'Ane': 'Hi-Han', 'Cochon': 'Gruik',
'Paon': 'Leon', 'Girafe': 'Tic-Tic'}
```

Terminal de test

Supprimer un élément

On supprime le couple `clé/valeur` d'un dictionnaire grâce au mot-clé `del` :

```
>>> del cris['Girafe']
>>> cris
{'Vache': 'Meuh', 'Chien': 'Wouf', 'Chat': 'Miaou', 'Poule': 'Cot-cot', 'Ane': 'Hi-Han', 'Cochon': 'Gruik',
'Paon': 'Leon'}
```

Longueur d'un dictionnaire

Un dictionnaire possède une **longueur** : le nombre de clés disponibles. Cette longueur est accessible avec la fonction *built-in* `len()` :

```
>>> len(cris)
7
```

Terminal de test

Test de présence de clés

Il est possible de tester l'existence d'une clé dans le dictionnaire grâce à l'*opérateur* `in` :

```
>>> 'Vache' in cris
True
>>> `vache` in cris
False
```

Par contre, cet opérateur **ne permet pas de tester l'existence d'une valeur** :

```
>>> 'Meuh' in cris
False
```

Terminal de test

Parcourir un dictionnaire

Pour parcourir un dictionnaire, on utilise une boucle `for` comme pour un parcours par élément d'une liste :

```
>>> for animal in cris :
    print(animal)
```

Terminal de test

On peut remarquer que la variable `animal` fait alors référence à une **clé**.

Pour obtenir la valeur associée on peut faire comme ci-dessous :

```
>>> for animal in cris :
    print(animal, " => ", cris[animal])
```

Terminal de test

Ordre d'un dictionnaire

Attention ! Selon les versions de Python, l'ordre obtenu par un parcours du dictionnaire ne respecte pas forcément l'ordre d'introduction ou de création des éléments dans le dictionnaire. Dans les versions supérieures à 3.6, l'implémentation des dictionnaires permet de conserver l'ordre d'introduction des éléments.

Liste des clés, des valeurs et des couples

Le **type dictionnaire** possède plusieurs **méthodes**, permettant d'obtenir les clés, les valeurs et les paires clés/valeurs :

- Pour obtenir la liste des **clés**, on utilise la *méthode* `keys()`, qui renvoie un objet de type `dict_keys`, assimilable à une liste.

```
>>> cris.keys()
 dict_keys(['Vache', 'Chien', 'Chat', 'Poule', 'Ane', 'Cochon', 'Paon'])
```

- Pour obtenir la liste des **valeurs**, on utilise la *méthode* `values()`.

```
>>> cris.values()
dict_values(['Meuh', 'Wouf', 'Miaou', 'Cot-cot', 'Hi-Han', 'Gruik', 'Leon'])
```

- On peut aussi obtenir le couple *clé/valeurs* sous la forme d'un *tuple* par l'intermédiaire de la méthode `items()` :

```
>>> cris.items()
dict_items([('Vache', 'Meuh'), ('Chien', 'Wouf'), ('Chat', 'Miaou'), ('Poule', 'Cot-cot'), ('Ane', 'Hi-
Han'), ('Cochon', 'Gruik'), ('Paon', 'Leon')])
```

Et on peut encore itérer sur cet objet :

```
for item in cris.items() :
    print(f"{item[1]} est le cri de {item[0]}")
```

Terminal de test

Attention à l'utilisation de la méthode `items`

Comme nous le verrons plus loin, l'objectif de l'utilisation des dictionnaires est entre autre d'accélérer certaines opérations par rapport à l'utilisation de listes ou de tuples. La méthode `items` renvoyant une liste de tuples à partir du dictionnaire, il est parfois peu judicieux de l'utiliser (voir la partie sur les tables de hachages ci-dessous).

Tuple unpacking

On peut aussi utiliser la technique du **tuple unpacking** pour extraire chaque élément clé et valeur grâce à la méthode `items`, en utilisant la technique suivante (mais elle est encore une fois moins rapide...):

```
for animal, cri in cris.items() :
    print(f"L'animal {animal} fait {cri} !")
```

Améliorer avec les dictionnaires

Énoncé

Reconstruisez une nouvelle fonction `afficheCri2` sur le même modèle que la première, mais en utilisant la structure de dictionnaire, puis testez la vitesse d'exécution de cette nouvelle fonction.

Réponses

Le code :

```
def afficheCri2(animal) :
    if animal in cris :
        return cris[animal]
    return f"Je ne connais pas le cri de {animal}"
```

Sur une petite structure comme celle-ci, le temps d'exécution n'est pas sensiblement différent (sauf dans le cas où la clé n'est pas dans le dictionnaire).

2.3. Et avec plus de données ?

Recréons à partir du même fichier Parcoursup `Long_Dico.csv` un dictionnaire contenant les établissements et leurs coordonnées GPS :

```
def makeBigDict() :
    import csv
    with open("Long_Dico.csv","r",encoding = "utf8") as file :
```

```
        dicReader = csv.DictReader(file, delimiter=';')
        etablissements=dict()
        for line in dicReader : # pour chaque ligne
            etablissements[line['Nom']] = line['GPS'] # on crée une nouvelle paire clé/valeur dans le
dictionnaire
    return etablissements

dicEtablissements = makeBigDict()
```

Et vérifions que sa taille st bien cohérente :

```
>>> len(dicEtablissements)
6183
```

Un gain de vitesse ?

Enoncé

1. Adaptez de nouveau la fonction `afficheCri2` en une fonction `afficheGPS2` afin qu'elle permette de récupérer les coordonnées GPS d'un établissement dont on a saisi le nom, et qu'elle renvoie `None` si l'établissement n'est pas présent.
2. Comparez le temps d'exécution avec la fonction `afficheGPS`.

Réponses

A venir !

3. Les dictionnaires : des tables de hachage

Cette partie est hors programme. Mais elle n'en demeure pas moins intéressante !

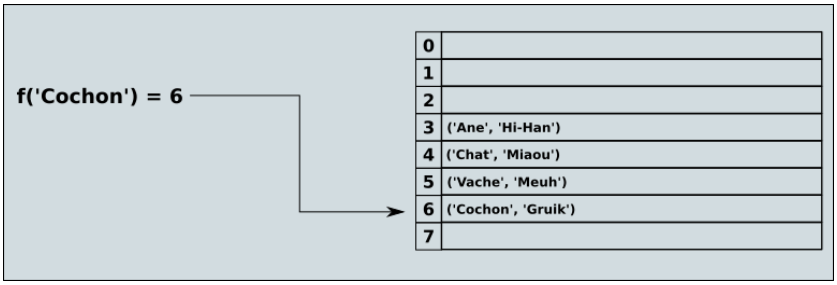
Si la structure de dictionnaire est plus rapide que celle d'un tableau simple, c'est parce qu'elle est construite sur le principe d'une **table de hachage**, à l'aide d'une **fonction de hachage**.

Une **fonction de hachage** est une fonction qui va renvoyer une valeur qui dépendra de l'objet qui lui est passé en argument.

A la création d'un dictionnaire, un tableau vide est créé. Ce tableau est un tableau classique, indexé par des entiers. Lors de l'ajout d'un élément dans un dictionnaire, la fonction de hachage calcule l'image de la clé, donc un nombre, et range dans la case correspondante du tableau nommée **alvéole** (ou **bucket** ou **slot**) la paire clé/valeur.

Prenons comme exemple de fonction de hachage la fonction f qui renvoie le **nombre de lettres de la chaine de caractère de la clé**. Avec cet exemple :

- $f('Vache') = 5$, donc le couple `("Vache", "Meuh")` est rangé dans la case d'indice 5 du tableau.
- $f('Cochon') = 6$, donc le couple `("Cochon", "Gruik")` est rangé dans la case d'indice 6 du tableau.
- $f('Ane') = 3$, donc le couple `("Ane", "Hi-Han")` est rangé dans la case d'indice 3 du tableau.
- $f('Chat') = 4$, donc le couple `("Chat", "Miaou")` est rangé dans la case d'indice 4 du tableau.



Cette méthode de construction apporte un net avantage dans le temps d'accès aux éléments. En effet, comme dans un tableau normal, **accéder à un élément avec son indice est une opération en temps constant (en $O(1)$)**. Et pour tester si une clé est bien

dans le dictionnaire, il suffit de calculer son **hash** et de regarder dans le tableau si la case contient quelque chose.

Cependant, elle implique un certain nombre de règles et de contraintes pouvant être assez gênantes :

1. **La clé d'un objet doit être non mutable**. En effet, si la clé change, la valeur de **hash** renvoyée par la fonction ne sera plus la même et l'objet serait perdu. Ainsi Python impose d'utiliser des objets **non mutables** comme clé, comme des entiers, des chaines de caractères, ou même des tuples :

   ```
   newDic = {(0,0) : 0, (0,1) : 5 , (0,2) : 3 , (0,1,2,3) :4}
   ```

   Remarquez que les clés peuvent être de types différents - tant que la fonction de hachage est capable de calculer, il n'y a pas de problèmes.

   Mais il est impossible d'utiliser des listes comme clé :

   ```
   >>> newDic[[1,0]] = 0
   ```

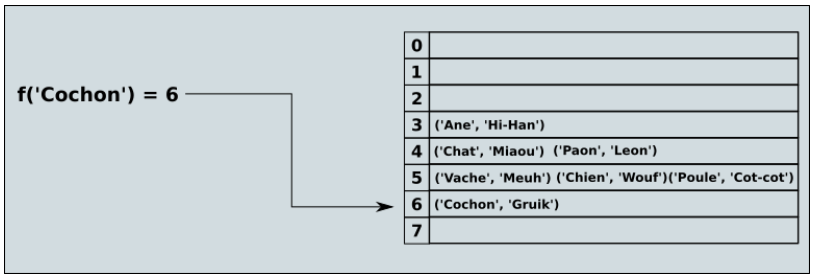
   Terminal de test

2. **Comme pour tous les tableaux, une taille de base est fixée au départ.** Si jamais l'ajout d'un nouveau couple clé/valeur amène à dépasser la taille du tableau initial, un nouveau tableau **2 fois plus grand** est créé et l'ensemble de l'ancien tableau est copié dans ce nouveau qui devient le nouvel objet de référence. Cette copie **peut être couteuse en temps et en mémoire**.
3. Avec certaines fonctions de hachage, **plusieurs clés peuvent avoir la même image**. Par exemple avec la fonction utilisant la longueur des chaines :
   - $f('Vache') = 5$
   - $f('Chien') = 5$

   On obtient alors une **collision**.

   Dans ce cas les deux couples `("Vache","Meuh")` et `("Chien","Wouf")` seront rangés dans la même alvéole, sous la forme d'une liste, ce qui risque de diminuer l'efficacité de la recherche, dans le cas d'alvéoles trop remplies.

   Par exemple, collisions comprises, avec cette fonction de hachage on obtiendrait la table de hachage suivante :



C'est pourquoi il est nécessaire d'avoir **une bonne fonction de hachage** qui limite les possibilités de collision !!!

Heureusement pour nous, **Python fait bien son travail** et utilise une fonction de hachage bien implémentée, dont on peut voir quelques exemples dans les cellules ci-dessous :

```
>>> hash("Vache")
3804037224742576468
>>> hash((0,1))
-1950498447580522560
>>> hash(35)
35
>>> hash(-41.2)
-461168601842745385
```

3.1. Exercices sur les dictionnaires

Exercice 1

Énoncé

1. Écrire un dictionnaire `mois` dont les clés seront les mois de l'année et les valeurs seront le nombre de jours du mois correspondant (année non-bissextiles).
2. Créer une fonction `quelMois` donnant les mois dont le nombre de jours est passé en argument.

Réponses

A venir !

Exercice 2

Énonce

1. Écrire une fonction `occurrence(chaine : str)` qui prend une chaine de caractères en argument, et qui renvoie un dictionnaire contenant le nombre d'occurrences de chaque caractère.

```
    def occurrence(chaine :str ) ->dict :
        """
        Prend en argument une chaine de caractère quelconque mais non vide, et renvoi
        un dictionnaire du nombre d'occurrence de chaque caractère de la chaine,
        y compris les espaces et caractères spéciaux.
        Par exemple :
        >>> occurrence("abc")
        {'a': 1, 'b': 1, 'c': 1}
        >>> occurrence("abaacc")
        {'a': 3, 'b': 1, 'c': 2}
        >>> occurrence("ab ! bc ! ")
        {'a': 1, 'b': 2, ' ': 4, '!': 2, 'c': 1}
        """
        ...
```

2. Écrire une fonction `occurrenceMot(chaine :str)` qui prend en argument une chaine de caractère, et renvoie un dictionnaire contenant le nombre d'occurrence de chaque mot de la chaine .

```
def occurenceMot(chaine :str ) ->dict :
    """
    Prend en argument une chaine de caractère quelconque mais non vide,
     et renvoie un dictionnaire du nombre d'occurrence de chaque mot
     de la chaine, en minuscule. On ne passera en argument que des
     chaines sans caractères de ponctuation.
    Par exemple :
    >>> occurenceMot("Le petit chien")
    {'le': 1, 'petit': 1, 'chien': 1}
    >>> occurence("Le petit chien joue avec le petit chat")
    {'le': 2, 'petit': 2, 'chien': 1, "joue" : 1, "avec" : 1, "chat" : 1 }
        """
    ...
```

Indice : on pourra utiliser la méthode de chaine `split(separateur)` qui renvoie une liste de sous chaines créé à partir du séparateur passé en argument :

```
>>> "Martine va à la plage". split(" ")
["Martine", "va", "à", "la", "plage"]
```

Réponses

A venir !

Exercice 3

(D'après Romain Tavenard, Université de Rennes 2)

Énoncé

On dispose d'un dictionnaire associant à des noms de commerciaux d'une société le nombre de ventes qu'ils ont réalisées. Par exemple :

```
ventes={"Dupont" : 14, "Hervy" : 19, "Geoffroy" : 15, "Layec" : 21}
```

1. Écrivez une fonction qui prend en entrée un tel dictionnaire et renvoie le nombre total de ventes dans la société.
2. Écrivez une fonction qui prend en entrée un tel dictionnaire et renvoie le nom du vendeur ayant réalisé le plus de ventes. Si plusieurs vendeurs sont ex-æquo sur ce critère, la fonction devra retourner le nom de l'un d'entre eux.et

Réponses

A venir !