

Dictionnaires

February 17, 2022

1 Présentation des dictionnaires

Les dictionnaires (type `dict` en Python) permettent de représenter des associations entre des **clés** et des **valeurs**. Par exemple le programme suivant :

```
d = { 'nom' : 'Évariste', 'age' : 23, 42 : True }
```

définit un dictionnaire associé à la variable `d` et qui contient trois clés : `'nom'`, `'age'` et `42`. La valeur associée à la clé `'age'` est l'entier `23`. On remarque donc :

- les clés peuvent être des valeurs de différents types **immuables** : qu'on ne peut pas modifier, donc des **tuple** mais pas des **list**
- les valeurs peuvent être des valeurs de types quelconques.

Pour accéder à la valeur associée à la clé `'nom'` on écrit `d['nom']`. Comme pour un tableau, on peut y accéder en lecture ou en écriture :

```
x = d['nom'] # x contient la valeur 'Évariste'
d['nom'] = 'Ada' # d associe maintenant 'Ada' à la clé 'nom'
```

- Le dictionnaire vide est `{}`
- On peut tester si une clé `c` est dans le dictionnaire `d` avec la valeur `c in d` qui est de type `bool`
- On peut supprimer l'association de la clé `c` dans `d` avec l'instruction `del d[c]` (rarement utile)
- En utilisant une boucle `for` on peut parcourir toutes les clés d'un dictionnaire : `for clé in d: ...`
- **Avancé** : on peut obtenir les valeurs dans `d` à l'aide de `d.values()` ainsi que les couples (clé,valeur) avec `d.items()`.

On suppose que chacune de ces opérations sur un dictionnaire s'effectue en temps constant, c'est-à-dire avec un nombre d'instructions borné et notamment indépendant du nombre de clés.

```
[6]: # Définition d'un dictionnaire
d = { 'nom' : 'Évariste', 'age' : 23, 42 : True }
'nom' in d
```

```
[6]: True
```

```
[7]: # Parcours simple des clés du dictionnaire
# DANS UN ORDRE A PRIORI QUELCONQUE
```

```
for clé in d:
    print('Clé', clé)
```

Clé nom
Clé age
Clé 42

```
[9]: # Parcours des valeurs associées a des clés dans le dictionnaire
for valeur in d.values():
    print('Valeur', valeur)
```

Valeur Évariste
Valeur 23
Valeur True

```
[10]: # Parcours des couples (clé, valeur)
for (clé, valeur) in d.items():
    print('Association', clé, '->', valeur)
```

Association nom -> Évariste
Association age -> 23
Association 42 -> True

Le plus souvent, on se sert d'un dictionnaire en partant d'un dictionnaire vide et en rajoutant des entrées au cours d'un algorithme. On donne un exemple d'une telle utilisation dans la cellule suivante.

```
[19]: # On définit un dictionnaire vide
d = {}

# Ajout d'une entrée
d['toto'] = 12

# Ajout d'une autre entrée
d[5] = 'Test'
d[True] = [1, 2] # la valeur peut être un tableau (list en Python)

# Si on évalue on obtient le dictionnaire voulu
d
```

```
[19]: {'toto': 12, 5: 'Test', True: [1, 2]}
```

2 Représentation en tant que couple de tableaux

Il est possible de simuler un dictionnaire en utilisant deux tableaux :

- un tableau des clés
- un tableau des valeurs

Par exemple, le dictionnaire précédent { 'nom' : 'Évariste', 'age' : 23, 42 : True } pourrait être représenté par les tableaux ['nom', 'age', 42] et ['Évariste', 23, True].

Combien d'opérations élémentaires (accès un élément d'un tableau ici) en fonction du nombre n de clés fait-on pour :

1. Chercher si une clé est présente
2. Accéder à une valeur à partir de sa clé
3. Rajouter une nouvelle clé (on supposera que l'ajout d'un élément à la fin d'un tableau se fait en une opération élémentaire)

Contrairement à un dictionnaire, une même clé peut avoir plusieurs valeurs si elle est présente plusieurs fois, on considérera alors uniquement la valeur de plus grand indice (on dit qu'on a masqué les valeurs précédentes).

Par exemple ([1,1], ['A','B']) associe 'B' à 1.

```
[16]: def couple_vers_dict(cles, valeurs):  
    '''Renvoie le dictionnaire associé aux couple de tableaux (clés,valeurs)  
    Cette fonction sera toujours appelée avec des tableaux de même longueur'''  
    # BEGIN SOLUTION  
    d = {}  
    for i in range(len(cles)):  
        d[cles[i]] = valeurs[i]  
    return d  
    # END SOLUTION
```

```
[20]: from civ import *  
init()  
test(couple_vers_dict, ['nom','age',42],['Évariste',23,True],  
     { 'nom' : 'Évariste', 'age' : 23, 42 : True })  
test(couple_vers_dict, [], [], {})  
test(couple_vers_dict, [1,1,1], [1,2,3], { 1 : 3 })  
valide()
```

OK!: couple_vers_dict(['nom', 'age', 42], ['Évariste', 23, True]) vaut {'nom': 'Évariste', 'age': 23, 42: True}

OK!: couple_vers_dict([], []) vaut {}

OK!: couple_vers_dict([1, 1, 1], [1, 2, 3]) vaut {1: 3}

Tous les tests sont valides

3 Fusion de dictionnaires

Si d1 et d2 sont deux dictionnaires n'ayant pas de clés en commun, on appelle fusion de d1 et de d2 le dictionnaire contenant les clés des deux dictionnaires et leurs associations respectives. Ainsi si d1 = { 1 : 'a', 'b' : 'c' } et d2 = { 3 : True } alors la fusion donnerait { 1 : 'a', 'b' : 'c', 3 : True }.

```
[46]: def fusion(d1, d2):
        '''Renvoie le dictionnaire obtenu en fusionnant d1 et d2'''
        # BEGIN SOLUTION
        d = {}
        for c in d1:
            d[c] = d1[c]
        for c in d2:
            d[c] = d2[c]
        return d
        # END SOLUTION
```

```
[47]: from civ import *
init()
test(fusion, { 1 : 'a', 'b' : 'c' }, { 3 : True }, { 1 : 'a', 'b' : 'c', 3 :
↪True })
valide()
```

OK!: fusion({1: 'a', 'b': 'c'}, {3: True}) vaut {1: 'a', 'b': 'c', 3: True}
Tous les tests sont valides

4 Comptage

Dans la question suivante on va utiliser un dictionnaire pour compter les occurrences dans un tableau, c'est-à-dire le nombre de fois que chaque valeur est présente. Pour cela, on va créer un dictionnaire `occ` tel que si `x` est dans le tableau, alors `occ[x]` contient le nombre de fois qu'il est présent.

Étant donné le tableau `['a', 'b', 'a', 'b', 'c', 'a']` on doit donc renvoyer le dictionnaire `{ 'a' : 3, 'b' : 2, 'c' : 1 }`.

Les valeurs qui ne sont pas dans le tableau ne doivent pas apparaître comme clé dans le dictionnaire.

```
[21]: def comptage(tableau):
        '''renvoie un dictionnaire associant à chaque valeur dans tableau le nombre_
        ↪entier de fois où
        cette valeur est présente'''
        ## BEGIN SOLUTION
        occ = {}
        for valeur in tableau:
            if valeur not in occ:
                occ[valeur] = 0
            occ[valeur] = 1 + occ[valeur]
        return occ
        ## END SOLUTION
```

```
[30]: from civ import *
init()
test(comptage, ['a', 'b', 'a'], { 'a': 2, 'b' : 1 })
```

```
test(comptage, [1,1,1], { 1: 3 })
test(comptage, [], {})
test(comptage, [1,2,1], { 2: 1, 1: 2 })
valide()
```

OK!: comptage(['a', 'b', 'a']) vaut {'a': 2, 'b': 1}
 OK!: comptage([1, 1, 1]) vaut {1: 3}
 OK!: comptage([]) vaut {}
 OK!: comptage([1, 2, 1]) vaut {2: 1, 1: 2}
 Tous les tests sont valides

Avancé : il est possible d'utiliser `d.get(clé, valeur_si_absente)` pour obtenir la valeur associée à une clé et, si cette clé n'est pas associée, renvoyer la valeur `valeur_si_absente`. Ainsi, on peut écrire

```
occ[valeur] = 1 + occ.get(valeur, 0)
```

5 Utilisation d'un dictionnaire comme cache

On va commencer par revenir sur les fonctions récursives mais en calculant des valeurs ici.

On considère la suite de Fibonacci (F_n) définie par :

$$F_0 = 0, F_1 = 1, \forall n \in \mathbb{N}, F_{n+2} = F_{n+1} + F_n$$

```
[26]: def fibo(n):
        '''Calculer le nième terme de la suite de Fibonacci en utilisant
        directement la récurrence'''
        # BEGIN SOLUTION
        if n == 0:
            return 0
        elif n == 1:
            return 1
        else:
            return fibo(n-1) + fibo(n-2)
        # END SOLUTION
```

```
[29]: from civ import *
init()
test(fibo, 6, 8)
test(fibo, 0, 0)
test(fibo, 1, 1)
valide()
```

OK!: fibo(6) vaut 8
 OK!: fibo(0) vaut 0
 OK!: fibo(1) vaut 1
 Tous les tests sont valides

```
[38]: fibo(30) # doit prendre un peu de temps si vous avez ce qu'il faut !
```

```
[38]: 832040
```

Combien votre fonction effectue d'appels récursifs ?

Pour pallier le problème précédent, on va utiliser un dictionnaire comme *cache* comme dans un navigateur où on ne va pas retelecharger une page quand elle n'a pas changé et qu'on en a conservé une copie.

La notion de cache est essentielle en informatique et elle suffit parfois à transformer un algorithme assez naïf en un algorithme efficace.

Pour cela, notre fonction va prendre deux paramètres :

- un entier n comme précédemment
- un dictionnaire **cache** contenant les associations (i, F_i) déjà calculées

Se faisant, on évite l'explosion liée au recalcul des valeurs.

La fonction va donc commencer par identifier si une clé est présente dans le cache et va renvoyer sa valeur, si ce n'est pas le cas, on va calculer récursivement la valeur et on rajoute une entrée dans le cache avant de renvoyer cette valeur. Comme les appels récursifs passent également par le partage de ce cache, on aura évité le problème précédent.

```
[31]: def fibo_cache(n, cache):  
    '''Calcule Fn en servant du dictionnaire cache.  
    Pour calculer Fn on appellera donc initialement fibo_cache(n, {})''  
    # BEGIN SOLUTION  
    if n in cache:  
        return cache[n]  
    if n <= 1:  
        v = n  
    else:  
        v = fibo_cache(n-1,cache) + fibo_cache(n-2,cache)  
    cache[n] = v  
    return v  
    # END SOLUTION
```

```
[42]: from civ import *  
init()  
test(fibo_cache, 6, {}, 8)  
test(fibo_cache, 0, {}, 0)  
test(fibo_cache, 1, {}, 1)  
valide()
```

OK!: fibo_cache(6, {}) vaut 8

OK!: fibo_cache(0, {}) vaut 0

OK!: fibo_cache(1, {}) vaut 1

Tous les tests sont valides

```
[43]: fibo_cache(30, {}) # maintenant ça doit être instantané !
```

```
[43]: 832040
```

```
[44]: fibo_cache(100, {}) # ici aussi alors que c'était inaccessible avec l'algo_
      ↪ précédent
```

```
[44]: 354224848179261915075
```

Il est possible de cacher le `cache` dans la fonction elle-même, mais on ne s'embêtera pas ici, ce n'est pas très grave d'avoir un argument supplémentaire et on peut toujours définir derrière :

```
def fibo(n):
    return fibo_cache(n, {})
```

6 Tri par comptage

Si on connaît à l'avance l'ensemble **ordonné** de valeurs présentes dans un tableau, il est possible d'utiliser le comptage pour en déduire un tri. Par exemple, si on sait que le tableau ne contient que des valeurs entières entre 1 et 4 et qu'on sait qu'il contient 4 fois 1, 3 fois 2 et 1 fois 4, alors trié par l'ordre croissant, c'est forcément le tableau `[1,1,1,1,2,2,2,4]`.

On peut donc en déduire un algorithme de tri appelé **tri par comptage** et qui est en temps linéaire sur le nombre de données et sur le nombre de valeurs possibles. Si le tableau contient n valeur et qu'il y a p valeurs possibles, on dira que sa complexité temporelle en pire cas est en $O(n + p)$.

```
[25]: def tri_comptage(atrier, possibles):
      '''Le tableau atriier contient uniquement des valeurs présentes, dans_
      ↪ l'ordre croissant, dans le tableau
      possibles. On demande de renvoyer le tableau correspondant à atriier avec_
      ↪ les valeurs triées dans l'ordre
      croissant déduit de possibles. On pourra utiliser la fonction comptage_
      ↪ précédemment définie.'''
      ## BEGIN SOLUTION
      occ = comptage(atrier)
      trie = [ None ] * len(atrier)
      i = 0
      for possible in possibles:
          if possible in occ:
              for j in range(occ[possible]):
                  trie[i] = possible
                  i += 1
      return trie
      ## END SOLUTION
```

```
[24]: from civ import *
      init()
```

```

test(tri_comptage, ['a','d','c','c','e','e','c'], ['a','b','c','d','e'], ['a','c',
↪ 'c', 'c', 'c', 'd', 'e', 'e'])
test(tri_comptage, [4,3,2,1], [1,2,3,4], [1,2,3,4])
test(tri_comptage, [1,2,1,3,2], [1,2,3], [1,1,2,2,3])
test(tri_comptage, [], [1], [])
valide()

```

OK!: tri_comptage(['a', 'd', 'c', 'c', 'e', 'e', 'c'], ['a', 'b', 'c', 'd', 'e']) vaut ['a', 'c', 'c', 'c', 'd', 'e', 'e']

OK!: tri_comptage([4, 3, 2, 1], [1, 2, 3, 4]) vaut [1, 2, 3, 4]

OK!: tri_comptage([1, 2, 1, 3, 2], [1, 2, 3]) vaut [1, 1, 2, 2, 3]

OK!: tri_comptage([], [1]) vaut []

Tous les tests sont valides