Dictionnaires et mutabilité

I. Introduction : intérêt des dictionnaires

Nous avons vu qu'avec les listes et les tableaux, il est possible de stocker plusieurs informations dans un même objet informatique. Chacune de ces information est alors identifiée par un indice numérique.

Le problème est que cet indice est très arbitraire et dépend de l'ordre dans lequel on a entré les informations.

Par exemple, si l'on souhaite retenir les informations relatives à une personne, on peut définir la liste :

```
>>> individu = ["Victor", "Hugo", 30, 77.4, 1.72]
>>> print(individu[0])  # Affiche le prénom
>>> print(individu[1])  # Affiche le nom
>>> print(individu[2])  # Affiche l'age
>>> print(individu[3])  # Affiche le poids en kg
>>> print(individu[4])  # Affiche la taille en m
```

Il faut alors se souvenir que pour récupérer l'age, il faut taper `individu[2]` ce qui n'est pas très pratique ! Dans cet exemple, il n'y a que 5 informations mais on pourrait imaginer un exemple beaucoup plus complexe...

Un dictionnaire va permettre de remplacer les indices par des clés qui donneront du sens à la valeur attendue :

```
>>> individu = {
        'prenom': "Victor",
        'nom': "Hugo",
        'age': 30,
        'poids': 77.4,
        'taille': 1.72
    }
>>> print(individu['prenom'])   # Les commentaires sont inutiles
>>> print(individu['nom'])
>>> print(individu['age'])
>>> print(individu['poids'])
>>> print(individu['taille'])
```

Les clés peuvent aussi être des nombres `int` ou `float` mais aussi des booléens `bool` ou des tuples `tuple` !

```
>>> j_aime_pas_les_listes_en_python = {1: "Pomme", 2: "Pêches", 3: "Poires"}
>>> print(j_aime_pas_les_listes_en_python)
{1: 'Pomme', 2: 'Pêches', 3: 'Poires'}
>>> print(j_aime_pas_les_listes_en_python[1])
Pomme
>>> print(j_aime_pas_les_listes_en_python[0])
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError: 0
```

II. Les dictionnaires

Un dictionnaire est donc un nouveau type de donnée : en Python le type dictionnaire s'appelle `dict`.

1) Création d'un dictionnaire

Pour créer un dictionnaire, on peut soit procéder comme dans l'exemple précédent :

```
>>> individu = {'prenom': "Victor", 'nom': "Hugo", 'age': 30, 'poids': 77.4, 'taille'
    : 1.72}
>>> individu = {            # Pour plus de lisibilité
        'prenom': "Victor",
        'nom': "Hugo",
        'age': 30,
        'poids': 77.4,
        'taille': 1.72
    }
```

soit créer un dictionnaire vide puis ajouter des valeurs avec la syntaxe

```
dictionnaire[clé] = valeur
```

Par exemple :

```
>>> individu = {}
>>> individu['prenom'] = "Victor"
>>> individu['nom'] = "Hugo"
>>> individu['age'] = 30
>>> individu['poids'] = 77.4
>>> individu['taille'] = 1.72
```

Il n'y a donc pas besoin de `append` : on ajoute des clés simplement en leur donnant une valeur.

2) Supprimer une clé

Nous avons vu comment ajouter des clés/valeurs dans un dictionnaire. Pour supprimer une clé, on utilise la syntaxe :

```
del dictionnaire[clé]
```

Par exemple :

```
>>> individu = {
        'prenom': "Victor",
        'nom': "Hugo",
        'age': 30,
        'poids': 77.4,
        'taille': 1.72
    }
>>> del individu['prenom']
>>> print(individu)
{'nom': 'Hugo', 'age': 30, 'poids': 77.4, 'taille': 1.72}
```

3) Nombre de clés

Pour calculer le nombre de clés enregistrée, la fonction `len` marche aussi pour les dictionnaires :

```
len(dictionnaire)
```

4) Vérifier l'existence d'une clé

Pour vérifier l'existence d'une clé dans un dictionnaire, on utilise la syntaxe :

```
clé in dictionnaire
```

Par exemple :

```
if 'prenom' in individu:
    print(individu['prenom'])
```

5) Fusionner deux dictionnaires*

Pour fusionner deux dictionnaires, on utilise la syntaxe :

```
dictionnaire1.update(dictionnaire2)
```

Par exemple :

```
>>> d1 = {'prenom': "Victor", 'nom': "Hugo"}
>>> d2 = {'nom': "Smith"}
>>> d1.update(d2)
>>> print(d1)
{'prenom': 'Victor', 'nom': 'Smith'}
>>> print(d2)
{'nom': 'Smith'}
```

alors que

```
>>> d1 = {'prenom': "Victor", 'nom': "Hugo"}
>>> d2 = {'nom': "Smith"}
>>> d2.update(d1)
>>> print(d1)
{'prenom': "Victor", 'nom': "Hugo"}
>>> print(d2)
{'nom': "Hugo", 'prenom': "Victor"}
```

6) Parcourir un dictionnaire

Pour parcourir un dictionnaire, on a trois possibilités :

- ⋆ **Parcourir les clés** :

```
for cle in dictionnaire.keys():
    print(cle)                          # Affiche la clé
    print(dictionnaire[cle])            # Affiche la valeur associée à la clé
```

- ⋆ **Parcourir les clés et les valeurs** :

```
for cle, valeur in dictionnaire.items():
    print(cle)                          # Affiche la clé
    print(valeur)                       # Affiche la valeur associée à la clé
    print(dictionnaire[cle] == valeur)  # Affiche True
```

- ⋆ **Parcourir les valeurs*** :

```
for valeur in dictionnaire.values():
    print(valeur) # Affiche la valeur mais on ne connait pas la clé (ni l'ordre)
```

III. Utilisation des dictionnaires

Soit `L` une liste. On souhaite compter le nombre d'éléments distincts dans cette liste.

⋆ Sans les dictionnaires :

```
def nb_elements_distincts(L):
D = []
for e in L:
    if e not in D:
        D.append(e)
return len(D)
```

⋆ Avec les dictionnaires :

```
def nb_elements_distincts(L):
D = {}
for e in L:
D[e] = 1
return len(D)
```

On peut améliorer la fonction précédente pour récupérer le nombre d'occurence de chaque élément :

```
def occurences(L):
    D = {}
    for e in L:
        if e in D:
            D[e] += 1
        else:
            D[e] = 1
    return D
```

IV. Compléments sur les listes, les dictionnaires et les tuples

1) Objets mutables, non mutables

Les dictionnaires, comme les listes ou les tuples (et dans une certaine mesure les chaînes de caractères), sont des conteneurs. Les dictionnaires et les listes sont mutables, les tuples et les chaînes de caractères ne sont pas mutables. Cela signifie qu'on peut modifier le contenu d'un dictionnaire ou d'une liste une fois défini : on peut ajouter un élément, supprimer un élément, modifier un élément.

```
>>> L = [1, 2, 3]
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> T = (1, 2, 3)
>>> L[0] = 12
>>> print(L)
[12, 2, 3]
>>> D['a'] = 12
>>> print(D)
{'a': 12, 'b': 2, 'c': 3}
>>> T[0] = 12
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

2) Phénomène d'aliasing

Les types mutables ont un comportement particulier lorsqu'on veut copier le contenu d'une variable dans une autre : on parle d'aliasing :

```
>>> L1 = [1, 2, 3]
>>> L2 = L1
>>> L1.append(4)
>>> print(L2)
[1, 2, 3, 4]
>>> D1 = {'a': 1, 'b': 2, 'c': 3}
>>> D2 = D1
>>> D1['d'] = 4
>>> print(D2)
{'a': 12, 'b': 2, 'c': 3, 'd': 4}
```

Les variables `L1` et `L2` désignent la même liste dans la mémoire. De même pour les variables `D1` et `D2`. Ce sont deux noms (deux alias) d'un même objet en mémoire.

3) Effets de bords

Une autre conséquence de la mutabilité et de l'aliasing est les effets de bords de fonctions utilisant ce type de variables :

```
def effet_de_bord_1(L):
    L.append(4)
    L[1] = 12

def effet_de_bord_2(D):
    D['d'] = 4
    del D['a']
```

```
>>> L = [1, 2, 3]
>>> effet_de_bord_1(L)
>>> print(L)
[1, 12, 3, 4]
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> effet_de_bord_2(D)
>>> print(D)
{'b': 2, 'c': 3, 'd': 4}
```

Mais certaines syntaxes n'ont pas cet effet là : si on réaffecte la variable, les modifications seront faites sur une copie. Par exemple :

```
def sans_effet_de_bord(L):
    L = L + [4]
    return L
```

```
>>> L = [1, 2, 3]
>>> sans_effet_de_bord(L)
[1, 2, 3, 4]
>>> print(L)
[1, 2, 3]
```