TP 18 Python – Dictionnaires

1. Définition d'un dictionnaire

Définition

En informatique, un **dictionnaire** (type `dict` en Python) est un type de structure de données qui associe des **valeurs** à des **clés**.

Plus précisément : chaque objet stocké dans le dictionnaire, appelé **valeur**, est associé une **clé** permettant l'accès à cette **valeur**. Chaque clé **doit être unique** afin de pouvoir retrouver la **valeur** qui lui correspond. La valeur, en revanche, peut être la même pour plusieurs clés différentes.

Remarque. À la différence des listes, des tableaux ou des couples, qui contiennent des valeurs indexées dans des **indices entiers**, et qui sont donc ordonnées séquentiellement, les valeurs d'un dictionnaire sont indexées par des clés qui peuvent être de diverses natures (entiers, flottants, couples, chaînes de caractères, …). Ainsi, la notion de "première valeur", "seconde valeur" n'aura pas de sens a priori dans un dictionnaire.

On peut définir directement un dictionnaire en écrivant les couples (clé : valeur) qu'il contient entre accolades, de la manière suivante :

```
dico = { clé1 : valeur1, clé2 : valeur2, ..., cléN : valeurn }
```

Exemple

1. On souhaite stocker dans un dictionnaire le nombre de pattes de certains animaux. On peut écrire :

```
bestiaire = {"chien" : 4, "oiseau" : 2, "cafard" : 6, "crocodile" : 4,
    "araignée" : 8}
```

   Ici, les clés sont les chaînes de caractères "chien", "oiseau", "cafard", "crocodile" et "araignée", et les valeurs associées sont les entiers 4, 2, 6, 4 et 8 respectivement.

2. Le dictionnaire suivant est aussi valide !

```
mondico = {"premier" : 1, "deuxième" : "Bonjour !", 3 : True}
```

   Ici, les clés sont "premier", "deuxième" et 3, et les valeurs sont 1, "Bonjour !" et le booléen True.

On peut également partir d'un dictionnaire vide ou déjà existant, pour le remplir en ajoutant de nouvelles paires (clé : valeur).

Un dictionnaire vide s'écrit `{}` ou encore `dict()`.

Cette fois, la syntaxe pour ajouter une paire consiste à indiquer directement le nom de la clé à côté du nom du dictionnaire et à lui affecter la valeur voulue :

```
dico[clé] = valeur
```

Exemple

1. Testez les commandes suivantes pour créer un second bestiaire.

```
nouveau_bestiaire = {}
# On aurait aussi pu écrire : nouveau_bestiaire = dict()
nouveau_bestiaire["fourmi"] = 6
nouveau_bestiaire["serpent"] = 0
nouveau_bestiaire["mille-pattes"] = "trop"
print(nouveau_bestiaire)
mondico["4ème"] = 7
print(mondico)
```

Sachant que les clés sont uniques, que fait l'instruction suivante ?

```
nouveau_bestiaire["mille-pattes"] = "au plus 1306"
```

Exécutez les lignes suivantes. Que se passe-t-il ?

```
clé1 = (5, 6)
mondico[clé1] = "c'est OK"
print(mondico)
clé2 = [7, 8]
mondico[clé2] = "et maintenant ?"
```

Explication : **On ne peut donc pas utiliser n'importe quel objet pour définir une clé, il faut qu'il soit immuable**, c'est-à-dire qu'il ne puisse pas être modifié. Ainsi, on pourra utiliser des nombres, tuples ou chaînes de caractères, mais **pas de listes** !

2. Accéder aux valeurs

Comme dans les listes, la syntaxe `dico[clé]` permet de récupérer la valeur associée à une clé donnée dans un dictionnaire Python.
⚠ Si la clé n'est pas une des clés existantes du dictionnaires, alors l'instruction renverra une erreur `KeyError` (l'équivalent de "list index out of range"…).

Exemple

Que font les instructions suivantes ? Prédire avant de vérifier.

a) `bestiaire["chien"]`
b) `bestiaire["fourmi"]`
c) `a = bestiaire["serpent"]`
d) `bestiaire["serpent"] = b`
e) `mondico["deuxième"]`
f) `mondico[1]`

Une bonne manière de traiter le cas où on n'est pas sûr de la présence d'une clé est l'utilisation de l'instruction `clé in dictionnaire`, qui renvoie le booléen `True` ou `False` selon que la clé est présente ou non dans le dictionnaire.
On peut s'en servir dans l'exemple très classique suivant : on veut initialiser à 1 une grandeur que l'on compte lorsqu'on la rencontre la première fois (donc quand la clé n'existe pas encore) et sinon ajouter 1 au décompte de la clé. Par exemple quand on compte les lettres d'un texte, sans savoir quelles lettres apparaissent. On peut alors écrire quelque chose comme :

```
if clé not in dictionnaire: # La clef n'existe pas dans le dictionnaire
    dictionnaire[clé] = 1 # Insertion du couple clé:valeur avec valeur = 1
else:
    dictionnaire[clé] += 1 # La clef existe déjà, on augmente la valeur de 1
```

On peut **parcourir un dictionnaire** à l'aide d'une boucle for, avec la syntaxe :

`for clé in dico:`

Une boucle for sur un dictionnaire permet donc de parcourir l'ensemble des **clés** du dictionnaire, et on a ensuite accès à la valeur en utilisant `dico[clé]` dans le corps de la boucle.

Exemple

```
for clé in bestiaire:
    if bestiaire[clé] != 6:
        print(clé, "est un insecte !")
```

Noter : la fonction `len` est aussi définie sur les dictionnaires, et renvoie le nombre de couples (clé : valeur) du dictionnaire.

3. Exercices – Manipulation de dictionnaires

Exercice 1.

1. Créer un dictionnaire `pokemon` dont les clés sont les mots `carapuce`, `salameche` et `bulbizarre` et dont les valeurs sont le nombre de lettres dans la chaîne de caractères correspondant à la clé.
2. Écrire l'instruction permettant d'ajouter `pikachu` à ce dictionnaire.
3. Créer un second dictionnaire `pokemon_2` dont les clés sont les mots `kaiminus`, `héricendre` et `germignon` et les valeurs le nombre de lettres dans la chaîne de caractères correspondant à la clé. Créer enfin un dictionnaire `starter` constitué des couples (clé:valeur) : `("1ère génération" : pokemon)` et `("2ème génération" : pokemon_2)`. Ainsi chacune des valeurs de `starter` et l'un des dictionnaires précédemment définis. c'est un dictionnaire de dictionnaires !
4. Que renvoient `starter["2ème génération"]["kaiminus"]` et `starter["1ère génération"]["kaiminus"]` ?

Exercice 2.

1. Écrire une fonction `noms_en_n(pokemons, n_lettres)` qui renvoie la liste des noms des pokemons dont le nom est constitué de `n_lettres` lettres. L'argument `pokemons` est un dictionnaire dont la structure est semblable aux variables `pokemon` et `pokemon_2` construites précédemment.
2. Écrire une fonction `starters_en_n(starters, n_lettres)` qui effectue la même tâche que la fonction précédente mais en se basant sur la variable `starter` définie précédemment (cette fois la fonction renvoie donc la liste de tous les starters, des générations 1 et 2 dont le nom est constitué de `n_lettres` lettres).

Exercice 3 (Un problème de comptage)**.** On travaille sur une séquence ADN de longueur N constituée des nucléotides A, T, C, G. Une telle séquence peut être représentée en Python par une chaîne de caractères de la forme `sequence ="AAAGGTCACCGTCGATC...ATGAT"` On appelle n-gramme une sous-chaîne de caractères de taille n extraite de la chaîne de caractère originale, les caractères étant pris d'affilée (pas de pas dans l'extraction). Les 1-gramme de `sequence` sont "A","T","C" et "G". Les 2-grammes sont "AA","AT",...,"TG","CG".

1. Écrire une fonction `stats_lettres(séquence)` qui reçoit en argument une chaîne de caractères `séquence` et renvoie un dictionnaire dont les clés sont les lettres du texte et les valeurs sont le nombre d'occurrences de chaque lettre. On pourra vérifier le bon fonctionnement de votre fonction grâce au test :

```
stats_lettres('ACCTGAAGTC') == {'A':3,'C':3,'T':2,'G':2}
```

2. Écrire une fonction `stats_bigrammes(sequence)` qui renvoie le dictionnaire dont les clés sont les bigrammes (mots de 2 lettres) de la chaîne de caractères `sequence` et les valeurs associées sont le nombre d'occurrences de ces bigrammes. On pourra vérifier la validité de notre fonction avec le test :

```
stats_bigrammes('ACCTAC') == {'AC':2,'CC':1,'CT':1,'TA':1}
```

3. Combien de mots de n lettres différents peut-on constituer à partir des lettres prises dans un alphabet de k symboles ? Que vaut k dans notre exemple ?
   On parle de n-gramme pour désigner un "mot" de n lettres.
4. On souhaite compter le nombre d'apparitions de chaque n-gramme. En pratique, pour des valeurs raisonnablement grandes de n, les n-grammes qui apparaissent dans sequence ne sont issus que d'une fraction réduite de tous les n-grammes possibles.
   Pour n fixé, pour quelles valeurs de N est on certain que tous les n-grammes n'apparaîtront pas dans la séquence de nucléotides ?
5. Écrire une fonction `compte_n_gram(sequence,n)` qui renvoie le dictionnaire dont les clés sont les n-grammes de `sequence` et les valeurs sont les nombres d'apparitions correspondantes. *Indication : c'est juste une généralisation de `stats_bigrammes`.*

Exercice 4. Pour cet exercice, on utilisera le fichier le fichier `germinal.txt` téléchargeable sur le site internet. On pourra aussi utiliser les fonctions de l'exercice précédent.
On peut stocker le fichier sous la forme d'une chaîne de caractères avec les instructions suivantes (qui ne sont pas à retenir) :

```
with open('germinal.txt') as f:
    livre = f.read()
```

On pourra vérifier que le fichier est correctement chargé avec l'instruction `print(livre)`. Tous les symboles typographiques sont considérés comme des caractères possibles : espaces, virgules, points... On ne cherche donc pas que des mots constitués de lettres de l'alphabet (sinon, c'est beaucoup plus difficile !).

1. Combien de bigrammes apparaissent dans le roman Germinal ?
2. Quel est le nombre d'occurrences du 5-grammes 'prépa' dans le roman germinal ?
3. Quel est le trigramme (mot de 3 caractères) dont le nombre d'occurrences est le plus élevé ?

4. Effets de bords

On dit qu'une fonction produit des **effets de bord** lorsqu'elle a d'autres effets sur l'état du programme que la valeur qu'elle renvoie. On ne peut alors plus se représenter cette fonction comme une fonction mathématique : elle ne produit pas simplement un résultat, elle **modifie** son environnement.

Exemple

- Sur une liste, la fonction `len` n'a pas d'effet de bord : elle se contente de renvoyer la longueur de la liste.
- La fonction `append`, en revanche, ne renvoie rien mais agit par effet de bord : elle *modifie* la liste en y ajoutant un élément.
- Les deux fonctions suivantes sont similaires :

```
def carré(x):
    return x * x
```

```
def carré(x):
    print("Carré :", x * x)
    return x * x
```

Les deux renvoient bien x^2, mais celle de droite produit *en plus* un effet de bord : elle affiche cette valeur.

- La fonction suivante additionne deux listes de mêmes tailles et renvoie le résultat :

```
def liste_add(l1, l2):
    for i in range(len(l1)):
        l1[i] += l2[i]
    return l1
```

```
>>> print(liste_add([1, 2, 3], [3, 2, 1]))
[4, 4, 4]  # le résultat attendu

>>> l1 = [1, 2, 3]
>>> print(liste_add(l1, [3, 2, 1]))
[4, 4, 4]  # même résultat

>>> print(l1)
[4, 4, 4]  # …mais l1 a été modifiée !
```

La modification de `l1` est a priori un effet inattendu, et peut facilement être source de bugs.

Remarque. En Python, les effets de bords sont fréquents : produire un effet de bord n'est pas une mauvaise chose. En revanche, les effets de bord *non attendus* sont source de confusion et doivent être évités.

Exercice 5.

1. Écrire une fonction `update(d1, d2)` qui prend en argument deux dictionnaires, et qui *met à jour* le premier pour qu'il représente l'union de $d1$ et $d2$. Dans le cas où une clé est présente dans les deux dictionnaires, on gardera la valeur de $d2$. On ne renverra aucun résultat.
2. Écrire une fonction `union(d1, d2)` qui prend en argument deux dictionnaires et qui renvoie un *nouveau* dictionnaire représentant l'union de $d1$ et $d2$. La priorité sera encore donnée à $d2$ pour les clés communes.
3. Quelle est la différence entre les deux fonctions précédentes ?