TP 18
Dictionnaires

Les *dictionnaires* sont un nouveau type de données permettant de contenir d'autres données, comme les listes ou les tuples. Nous en avons en fait déjà croisés au TP précédent et nous allons approfondir.

I Introduction

Lorsque nous avons écrit dans le TP précédent

```
d = {"sexe": "2", "prenom": "EMMA", "annee": "2004", "nombre": "6634"}
```

nous avions déjà affaire à un dictionnaire. Ce qu'on nomme en Python dictionnaire est un ensemble de **valeurs** auxquelles on accède via les **clés**. Ici les clés sont les noms `"sexe"`, `"prenom"`, `"annee"`, `"nombre"`. Les valeurs correspondantes sont obtenues avec la syntaxe `d["sexe"]`, `d["prenom"]` etc.
Cela ressemble donc fort à une liste dans laquelle les indices ne sont pas seulement des nombres entiers, mais peuvent être des chaines de caractères. On les appelle aussi parfois des *tableaux associatifs*, car ils associent une valeur à la clé donnée. Cette structure peut être pratique pour de nombreuses applications.
Quelques remarques sur le fonctionnement des dictionnaires :

1. L'ordre des clés n'a pas vraiment d'importance.
2. Bien sûr, chaque clé ne peut apparaitre qu'une seule fois, sinon cela n'a pas de sens.
3. Les clés peuvent être en fait de beaucoup de types différents : chaines de caractères, mais aussi entiers, flottants, tuples composés de ceux-ci...

Dans un dictionnaire `d`, et pour une valeur notée `k`, l'accès à `d[k]` va déclencher une erreur si `k` n'est pas une clé de `d`. Il est donc souvent utile de pouvoir tester à l'avance cette condition, avec la syntaxe `k in d` dont la négation est `k not in d` :

```
>>> "prenom" in d
True
>>> "age" in d
False
>>> "age" not in d
True
>>> d["age"]
KeyError: 'age'
```

Par contre on peut rajouter des clés au fur et à mesure :

```
>>> d["age"] = 19
>>> d
{'sexe': '2', 'prenom': 'EMMA', 'annee': '2004', 'nombre': '6634', 'age': 19}
```

ou en supprimer

```
>>> del d["nombre"]
>>> d
{'sexe': '2', 'prenom': 'EMMA', 'annee': '2004', 'age': 19}
```

II Création de dictionnaires

Les méthodes suivantes permettent de créer un dictionnaire :

1. En listant tout simplement les clés et les valeurs comme précédemment (on dit *en extension*), entre accolades, de la forme `clé: valeur`.

2. Le dictionnaire vide est noté `{}`. Parfois on souhaite partir d'un dictionnaire vide et ajouter des clés unes par unes.
3. Avec la syntaxe *en compréhension* comme pour les listes, mais où on peut donner à la fois une expression pour les clés et pour les valeurs. Par exemple le dictionnaire suivant

   ```
   d = {x**2: x for x in range(10)}
   ```

   donne le dictionnaire

   ```
   {0: 0, 1: 1, 4: 2, 9: 3, 16: 4, 25: 5, 36: 6, 49: 7, 64: 8, 81: 9}
   ```

   dont les clés sont (certains) nombres et les valeurs correspondantes vont être leur racines carrées. Ce dictionnaire permet donc de calculer directement les racines carrés de ces nombres (et seulement ceux-là), par exemple `d[49]` donne `7` (il n'y a plus de guillemets ici : les clés sont de type `int`). Ainsi les dictionnaires peuvent être utiles pour représenter des applications au sens mathématiques et leur réciproque.
4. Comme pour tous les types, il existe une fonction de conversion `dict()` vers les dictionnaires. On ne peut pas convertir une simple liste en dictionnaire (cela n'a pas de sens !) mais on peut convertir une *liste* de *couples* `(k, v)` que l'on interprétera comme un dictionnaire avec la clé `k` et la valeur correspondante `v`. On peut aussi convertir *deux listes séparées, de même longueur* en dictionnaire, disons une liste `K` qui sera celle des clés et `V` qui sera celle des valeurs. Il faut alors passer par l'intermédiaire de la fonction `zip(K, V)` qui va fournir une liste de couples `(k, v)`, donc le dictionnaire correspondant est `dict(zip(K, V))`.

   ```
   >>> K = ["sexe", "prenom", "annee", "age"]
   >>> V = ["2", "EMMA", "2004", 19]
   >>> list(zip(K, V)) # liste de couples
   [('sexe', '2'), ('prenom', 'EMMA'), ('annee', '2004'), ('age', 19)]
   >>> dict(zip(K, V)) # dictionnaire correspondant
   {'sexe': '2', 'prenom': 'EMMA', 'annee': '2004', 'age': 19}
   ```

   Dans l'autre sens, on peut convertir un dictionnaire en paire de listes ou en liste de couples, voir la section suivante. Mais tout ceci nous sera peu utile en pratique.

Les dictionnaires sont aussi utilisés par le langage Python lui-même pour maintenir des informations sur le programme en cours de fonctionnement… La fonction `globals()` renvoie un dictionnaire des variables actuellement enregistrées dans la session, affichez-le !

III Itération sur un dictionnaire

Dans le TP précédent nos dictionnaire avaient tous quatre clés fixes et bien connues à l'avance. Mais en pratique, une fonction reçoit en argument un dictionnaire et ne sait pas forcément quelles en sont les clés. Il faut donc utiliser une boucle `for` pour parcourir un à un tous les éléments du dictionnaire, tout comme on parcourt les éléments d'une liste de longueur quelconque.
Cependant il y a trois façons de faire…
Reprenons un dictionnaire :

```
d = {"sexe": "2", "prenom": "EMMA", "annee": "2004", "nombre": "6634"}
```

1. Itérer sur les *clés* de `d` : c'est une boucle `for` sur l'objet `d.keys()`, qui fournit unes par unes les clés de `d`.

   ```
   for k in d.keys():
       print(k)
   ```

   ```
   sexe
   prenom
   annee
   nombre
   ```

2. Itérer sur les *valeurs* de `d` : de même, c'est une boucle `for` qui porte sur l'objet `d.values()` qui fournit les valeurs unes par unes.

```
for v in d.values():
    print(v)
```

```
2
EMMA
2004
6634
```

3. Itérer sur les *paires* `(clé, valeur)` de `d` avec l'objet `d.items()`, qui fournit des tuples.

```
for (k, v) in d.items():
    print("clé :", k, "valeur :", v)
```

```
clé : sexe valeur : 2
clé : prenom valeur : EMMA
clé : annee valeur : 2004
clé : nombre valeur : 6634
```

Remarque 1. Bien sûr, en pratique itérer sur les clés fonctionne toujours, puisque si on a `k` alors on a accès à `d[k]`. C'est d'ailleurs le comportement par défaut si on écrit tout simplement `for k in d`. Cependant il faut considérer que chaque opération de rechercher une clé, et de chercher sa valeur correspondante, est une opération lourde. Plus lourde qu'accéder au i-ème élément d'une liste. En effet les éléments d'une liste sont rangés bien régulièrement à la suite dans la mémoire et donc on peut calculer directement l'adresse du i-ème élément et le lire. En revanche, bien que les clés du dictionnaire soient rangées dans la mémoire consécutivement comme dans une liste, à chaque accès il faut parcourir toute cette liste pour trouver où est la clé. Et du coup il faut aussi comparer la chaine de caractères qu'on donne avec chacune des clés présentes, tout ceci est lourd. Ainsi les deux autres méthodes présentées ont bien leur intérêt propre et évitent de rechercher plusieurs fois les mêmes clés dans le dictionnaire.
Le mécanisme qui permet la recherche rapide de clés s'appelle **table de hachage**, disons très brièvement que c'est une méthode pour ranger les clés de façon à pouvoir calculer rapidement où elle se trouve en donnant une certaine formule (fonction de hachage). Cela explique aussi pourquoi les clés ne peuvent pas être de n'importe quel type. Imaginons un dictionnaire contenant pour clés les deux listes `L = [1, 2]` et `M = [1, 3]` (pourquoi pas), avec une valeur pour chacune. Puis faisons `M[1] = 2`. Les listes `L` et `M` deviennent alors égales, du coup il n'y a plus qu'une seule clé ? Comment retrouver alors les valeurs ? En fait, cela est interdit : les clés ne peuvent pas être **mutables**, et doivent être **hachables**.

IV Exercices d'application

Dans cette première série d'exercices, aucune méthode n'est extraordinairement nouvelle. Il s'agit de boucles pour parcourir un dictionnaire et il faut seulement se poser la question du choix de l'une des trois méthodes d'itération précédente. Ensuite, ce sont les mêmes types d'algorithmes que ceux rencontrés de nombreuses fois sur les listes.

Exercice 1. *Itérer sur les valeurs*
On représente une liste de courses par un dictionnaire qui donne, pour chaque produit acheté, le prix en euros.

```
courses = {"pain": 1.20, "camembert": 3.0, "salade": 1.5, "savon": 3.5}
```

1. Écrire une fonction `facture(d)` qui prend en argument un tel dictionnaire et qui calcul le montant total de la facture.
2. Écrire une fonction `est_tropcher(d)` qui retourne `True` si l'un des articles a un prix supérieur à 5 euros, et `False` sinon.

Exercice 2. *Itérer sur les couples*
On représente un porte-monnaie contenant des pièces ou des billets par un dictionnaire `d`, où `d[x]` représente le nombre de billets (ou pièces) de valeurs `x`. Par exemple, le dictionnaire

```
d = {1: 4, 2: 7, 10: 1}
```

représente un porte-monnaie avec 4 pièces de 1 euro, 7 pièces de 2 euros et un billet de 10 euros. Remarquez qu'on ne se préoccupe en fait pas de s'il s'agit de billets ou de pièces, ni si les valeurs de ces pièces existent réellement, tout cela pourrait fonctionner de la même façon dans d'autres systèmes monétaires que l'euro. Dans cet exemple sa somme totale est de 28 euros.
Écrire une fonction `oseille(d)` qui prend en argument un tel dictionnaire représentant un porte-monnaie et qui renvoie la somme d'argent totale que cela représente.

Exercice 3. *Itérer sur les clés*
On représente une recette de cuisine par un dictionnaire contenant dont les clés sont les ingrédients et les valeurs, pour chaque ingrédient, sont la quantité (l'unité est variable selon l'ingrédient : gramme, millilitre, nombre). Par exemple

```
crepes = {"farine": 250, "oeufs": 4, "lait": 300, "beurre": 50, "sucre": 30}
```

1. Écrire une fonction `nombre_ingredients(d)` qui renvoie le nombre d'ingrédients différents de la recette.
2. On est allergique aux noix. Écrire une fonction `sans_noix(d)` qui retourne `True` si la recette ne contient pas de noix, et `False` sinon.
3. On souhaite faire un régime. Écrire une fonction `est_sain(d)` qui retourne `True` si la recette contient moins de 50 grammes de sucre, et `False` sinon.
   Attention car il peut se produire deux choses : ou bien `"sucre"` sera dans les clés avec une valeur qui doit être inférieure à 50, ou bien `"sucre"` ne sera pas du tout dans les clés auquel cas c'est bon aussi.
   Quelques exemples :

   ```
   >>> est_sain(crepes)
   True
   >>> tarte_citron = {"pate": 300, "citron": 500, "sucre": 200, "oeuf": 4}
   >>> est_sain(tarte_citron)
   False
   >>> kebab = {"pain": 200, "viande": 150, "frites": 250, "salade": 30}
   >>> est_sain(kebab)
   True
   ```

V Exercices

Exercice 4. On donne le dictionnaire suivant (recopier tel quel) :

```
chiffres = {"zéro": 0, "un": 1, "deux": 2, "trois": 3, "quatre": 4,
            "cinq": 5, "six": 6, "sept": 7, "huit": 8, "neuf": 9}
```

Le but est d'écrire une fonction `lit_nombre(s)` qui va convertir un texte écrit en toutes lettres qui épelle un nombre, par exemple `s = "deux huit trois"`, en un nombre entier, par exemple ici 283. On s'y prend en améliorant une fonction par plusieurs étapes successives.

1. Dans un premier temps la fonction prend en argument une simple *liste de mots* (par exemple ici : `["deux", "huit", "trois"]`) et affiche uns par uns les chiffres correspondant.
2. Ensuite, étant donnée une chaine de caractère `s`, la méthode `s.split()` « casse » la chaine aux caractères espaces et produit une liste de mots. Ainsi avec notre exemple si `s = "deux huit trois"` alors `s.split()` donne bien la liste `["deux", "huit", "trois"]`.
3. Enfin, il ne faut pas afficher les chiffres individuels mais calculer le nombre qui est représenté. On remarque qu'étant donné un nombre N écrit avec la liste de ses chiffres $N = \underline{a_k \dots a_1 a_0}$ (a_0 est le chiffre des unités, a_1 le chiffre des dizaines, etc) alors le nombre N est égal à

$$N = \Big(\cdots \big(a_k \times 10 + a_{k-1} \big) \times 10 + \cdots \Big) \times 10 + a_0 \tag{1}$$

   par exemple $283 = (2 \times 10 + 8) \times 10 + 3$, ce qui permet de calculer N en lisant ses chiffres, dans une boucle, de gauche à droite ; c'est à dire dans le même ordre qu'on lit les chiffres en français et que lui sont passés les chiffres listés dans `s` dans la boucle.

Exercice 5. On souhaite étudier les anagrammes. Pour cela la première étape est de partir d'un mot, représenté comme une chaine de caractères, et de compter combien de fois apparait chaque lettre. Si on n'avait pas les dictionnaires, il faudrait savoir à l'avance avec combien de lettres on travaille (par exemple 26 — mais alors il n'y a plus de marge pour les accents ou les majuscules) et initialiser une liste de taille 26 comptant combien de fois apparait chaque lettre. C'est ce que nous avons fait dans le DS 4 en étudiant des anagrammes uniquement sur les chiffres de 0 à 9. Les dictionnaire vont permettre de résoudre ce problème plus simplement.

1. Écrire une fonction `compte_lettres(s)` qui prend en argument une chaine de caractères `s` et qui retourne un dictionnaire `d`, dont les clés sont des lettres apparaissant dans `s` et dont la valeur `d[x]` est le nombre de fois que la lettre `x` apparait.
   Pour cela on a besoin d'initialiser un dictionnaire vide au début, puis d'une boucle qui fournit unes par unes les lettres de `s`. Attention car à chaque lettre, il faut tester si elle est déjà dans le dictionnaire (auquel cas incrémenter la valeur), ou sinon l'ajouter dedans simplement.
2. Écrire une fonction `nombre_anagrammes(s)` qui compte le nombre d'anagrammes de `s`.
   On rappelle qu'il s'agit de la factorielle du nombre de lettres de `s`, divisé par le produit des factorielles du nombre de fois que chaque lettre apparait ; on a donc besoin d'appeler la fonction précédente et d'itérer sur le dictionnaire `d` pour calculer ce produit de factorielles. On pourra ré-écrire rapidement la fonction factorielle, ou utiliser celle fournie avec `from math import factorial`.

Exercice 6. *Matrices creuses*
On s'intéresse à des matrices de taille (n, p) contenant une grande majorité de coefficients nuls, et quelques coefficients par-ci par-là non nuls. Plutôt que de stocker en mémoire un tableau entier de $n \times p$ cases dont la plupart vont être nulles, on représente une telle matrice par un dictionnaire `d` dont les clés sont des couples (i, j) ($0 \leqslant i < n$ et $0 \leqslant j < p$ comme d'habitude dans la convention informatique) et la valeur `d[(i, j)]` est le coefficient d'indice (i, j). Si une clé n'est pas présente, on interprète le coefficient correspondant comme étant nul. Par exemple la grosse matrice

$$A = \begin{pmatrix} 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 \\ 0 & -1 & 0 & 0 & 0 \end{pmatrix} \tag{2}$$

sera représentée tout simplement par le dictionnaire

```
d = {(0, 2): 5, (2, 3): 7, (3, 1): -1}
```

Écrire les fonctions `identite(n)` (matrice identité de taille n), `somme(A, B)` (somme de deux matrices), `produit(A, B)` (produit de matrices), `est_triangulaire_superieure(A)` (renvoie `True` si la matrice est triangulaire supérieure, `False` sinon) dans ce contexte.
On a toujours le même problème : il faut tester si une clé est présente ou non avant d'y accéder, et il faut considérer que le coefficient est zéro si la clé n'apparait pas. Se renseigner sur les méthodes `d.setdefault(k, v)` et `d.get(k, v)` ainsi que sur le type `defaultdict` et comment cela permet de traiter ce problème.