TP 18
Dictionnaires

Les **dictionnaires** sont un nouveau type 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. Travailler avec les dictionnaires sera parfois bien pratique.
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
>>> print(d)
{'sexe': '2', 'prenom': 'EMMA', 'annee': '2004', 'nombre': '6634', 'age': 19}
```

ou en supprimer

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

Le dictionnaire vide est noté tout simplement `{}`. Parfois on souhaite partir d'un dictionnaire vide et ajouter des clés au fur et à mesure.
Il existe aussi, comme pour les listes, une syntaxe **en compréhension**, où on peut donner à la fois une expression pour les clés et pour les valeurs. Étudions par exemple le dictionnaire suivant

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

qui donne

```
{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 un dictionnaire représente une application au sens mathématique, de l'ensemble des clés vers l'ensemble des valeurs, et l'application réciproque correspond simplement à échanger les clés avec les valeurs.
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 !

II Itération sur un dictionnaire

Dans le TP précédent nos dictionnaire avaient tous quatre clés fixes et bien connues à l'avance ; la liste de tous les prénoms était en fait une *liste* de dictionnaires. 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
   ```

Bien sûr, en pratique itérer sur les clés fonctionne toujours, puisque si on a `k` alors on a accès à `d[k]`. Mais il faut considérer que l'accès à une valeur est une opération lourde, bien plus lourde dans les dictionnaire que l'accès aux éléments d'une liste (voir l'annexe). Ainsi itérer directement sur les paires, ou sur les valeurs, est bien plus rapide que d'itérer sur les clés *puis* de chercher les valeurs correspondantes.

III 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_trop_luxueux(d)` qui renvoie `True` si l'un des articles a un prix supérieur à 5 euros, et `False` sinon.

Exercice 2. *Itérer sur les clés*
On représente une recette de cuisine par un dictionnaire 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. Supposons qu'on soit allergique aux noix. Écrire une fonction `est_sans_noix(d)` qui renvoie `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 renvoie `True` si la recette contient moins de 50 grammes de sucre, et `False` sinon.
   Attention car il peut se produire deux situations : 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.

Exercice 3. *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 1 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.

IV Problèmes

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}
```

1. Écrire une fonction `traduit(L)` qui prend en argument une liste de mots parmi ceux-ci, et renvoie la liste des chiffres correspondants.

   ```
   >>> traduit(["deux", "huit", "trois"])
   [2, 8, 3]
   ```

2. Écrire une fonction `nombre(L)` qui prend en argument toujours une liste de mots, et renvoie le nombre que cela forme, de type `int`.

   ```
   >>> nombre(["deux", "huit", "trois"])
   283
   ```

   On remarque qu'étant donné un nombre N écrit avec la liste de ses chiffres $N = \underline{a_k \dots a_1 a_0}$ (où 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 de gauche à droite (donc dans l'ordre naturel pour le problème que nous traitons là).

3. Bonus : étant donnée une chaine de caractères `s`, la méthode `s.split()` « casse » la chaine aux caractères espaces et produit une liste de mots :

```
>>> s = "deux huit trois"
>>> s.split()
['deux', 'huit', 'trois']
```

Ré-écrire la fonction `nombre` pour qu'elle prenne en argument une seule chaine de caractères.

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. Avec les dictionnaires, nous allons pouvoir travailler avec des mots absolument quelconques et sans connaitre les lettres à l'avance.

1. Écrire une fonction `compte_lettres(s)` qui prend en argument une chaine de caractères `s` et qui renvoie 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 (mais réciproquement, il peut y avoir des clés avec une valeur nulle). 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 classiques dans ce contexte :

1. `identité(n)` : matrice identité de taille n,
2. `somme(A, B)` : somme de deux matrices,
3. `produit(A, B)` : produit de matrices,
4. `est_triangulaire_superieure(A)` : renvoie `True` si la matrice est triangulaire supérieure, `False` sinon.

Étonnamment, les fonctions n'ont pas vraiment besoin de connaitre la *taille* de la matrice, et la notion n'a pas de sens ici…

Remarque 1. On tombe toujours sur 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. Cela est assez ennuyeux surtout que tester si une clé est présente dans le dictionnaire est déjà une opération lourde, tout autant que d'accéder à la valeur (voir l'annexe). Se renseigner notamment sur les méthodes `d.setdefault(k, v)` et `d.get(k, v)` ainsi que sur le type `defaultdict` et comment cela permet de traiter élégamment ce problème.

V Annexe : tables de hachage

Nous avons déjà dit plusieurs fois que dans une liste, il faut imaginer que les éléments sont rangés les uns à la suite des autres comme dans des cases de la mémoire. Cela permet d'accéder directement au i-ème élément de la liste.
Dans un dictionnaire, on peut imaginer naïvement ranger à la suite les clés et leurs valeurs, dans des cases aussi. Cependant les problèmes suivants se posent :

1. Pour rechercher une clé du dictionnaire, et sa valeur correspondante, il est nécessaire de parcourir toutes les clés unes par unes, exactement comme quand on cherche un élément dans une liste. Cela peut être long s'il y a beaucoup de clés.
2. De plus, si les clés sont de type chaine de caractères, comparer les clés prend plus de temps que de comparer des nombres ; car pour comparer deux chaines il faut comparer successivement leurs caractères uns par uns.

Ainsi cette méthode peut vite devenir très lourde.
La solution qui a été trouvée s'appelle **table de hachage**. Elle consiste à définir une certaine fonction mathématique (assez abstraite) dite **fonction de hachage**, calculable sur *tous* les objets possibles pouvant servir de clé, dont le résultat est un simple nombre entier qui puisse nous dire où se situe la clé dans la mémoire. Cela résout le problème 2 car comparer des nombres est nettement plus rapide que comparer des chaines de caractères, et cela résout partiellement le problème 1 — au minimum on peut espérer ranger les clés dans l'ordre croissant selon leur valeur de hachage, et rechercher les clés rapidement grâce à la dichotomie.
En fait un problème immédiat se pose : il y a de toute façon beaucoup plus de clés possibles que de nombres et certaines clés auront donc la même valeur par la fonction de hachage (on parle de **collision**), en termes mathématiques la fonction de hachage part d'un ensemble très gros vers un ensemble plus petit et ne peut donc pas être injective (principe des tiroirs...) Cela rend la conception de tables de hachages plus subtile que ce qui est décrit ici. La fonction de hachage doit être créée de telle façon que les collisions ne se produisent pas trop souvent, et si c'est le cas, si deux clés se retrouvent avec la même valeur de hachage, alors tant pis : on les stocke à la suite et on comparera les clés comme dans la méthode naïve.
En Python, la fonction de base `hash(x)` permet de connaitre la valeur de hachage d'un objet `x`, même si ce nombre ne nous dit concrètement pas grand chose...

```
>>> x = 3
>>> hash(x)
3
# OK, pour les nombres entiers c'est eux-mêmes
>>> x = (1, 2)
>>> hash(x)
-3550055125485641917
# que faire de cette information ?
>>> x = "steak"
>>> hash(x)
5425928401636965275
# le steak est haché !
```

Tous les objets ne peuvent pas servir de clé. Imaginons un dictionnaire contenant pour clés les deux listes `L = [1, 2]` et `M = [1, 3]` (pourquoi pas), avec une valeur pour chacune.

```
d = {L: "truc", M: "machin"}
```

Puis faisons `M[1] = 2`. Les listes `L` et `M` deviennent alors égales, et devraient donc avoir la même valeur de hachage, du coup il n'y a plus qu'une seule clé ? Qu'est-ce que `d[[1, 2]]` ? Comment retrouver alors les valeurs ? En fait, cela est interdit et les listes ne sont pas hachables. Les objets hachables ne doivent jamais pouvoir être modifiés, et la fonction de hachage doit toujours renvoyer la même valeur pour un même objet.

```
>>> L = [1, 2]
>>> hash(L)
TypeError: unhashable type: 'list'
```