Langages de script (Python)

CMTP n° 3 : Itérables qui ne sont pas des séquences : ensembles et dictionnaires

---

Dans CMTP1 et CMTP2 nous avons vu des exemples de types itérables (c.-à-d. sur lesquels on peut itérer par exemple avec l'opérateur `for`) qui sont des séquences (c.-à-d. dont on peut accéder aux élément grâce aux indices), immutables (chaînes de caractères, tuples), comme mutables (listes). Dans ce sujet nous nous s'occuperons d'autres types d'itérables qui ne sont pas des séquences (pas d'indices, pas de "slices") : les *ensembles* et les *dictionnaires* (ou listes d'association).

I) Ensembles

Un ensemble est une collection non ordonnée sans élément dupliqué. Des utilisations basiques concernent par exemple des tests d'appartenance ou des suppressions de doublons. Les ensembles savent également effectuer les opérations mathématiques telles que les unions, intersections, différences et différences symétriques.

```
>>> # comme pour les listes: trois moyens d'introduire les ensembles:
>>> s = {1, 2, 3, 4}                # explicite avec accolades
>>> s2 = {x for x in range(1, 5)}   # par comprehension
>>> s2
{1, 2, 3, 4}
>>> s3 = set(range(1, 5))           # par le constructeur set(iterable)
>>> s3
{1, 2, 3, 4}
>>> {4, 3, 1, 2} == s               # attention elements pas ordonnes
True
>>> u = {0, 0, 2}                   # attention: pas de doublons
>>> u
{0, 2}
>>> s - u                           # difference (elements dans s mais pas dans u)
{1, 3, 4}
>>> s | u                           # union (elements dans s ou dans u ou les deux)
{0, 1, 2, 3, 4}
>>> s & u                           # intersection (elements dans s et dans u)
{2}
>>> s ^ u                           # union exclusive ()
{0, 1, 3, 4}
>>> 2 in s                          # test d'appartenance
True
>>> 2 not in s
False
>>> s[0]                             # pas d'indices (pas sequence)
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    s[0]
TypeError: 'set' object is not subscriptable
>>> s.add(-10)                      # les ensembles sont mutables, comme les listes
>>> s
{1, 2, 3, 4, -10}
>>> s.clear()
>>> s
set()
>>> x = set()                       # ensemble vide
```

```
>>> x
set()
>>> type(x), type({})               # attention, accolades vides est dictionnaire
(<class 'set'>, <class 'dict'>)
```

Pour plus des détails sur les ensembles, voir https://docs.python.org/3/library/stdtypes.html#set.

Exercice 1 : ensembles

1. Écrire une fonction `same_elements(l1, l2)` qui renvoie `True`, si les listes `l1` et `l2` contiennent les mêmes éléments (dans n'importe quel ordre et possiblement dupliqués), `False` sinon.
2. Écrire une fonction `letters()` qui demande à l'utilisateur d'écrire une ligne et renvoie l'ensemble des *lettres en minuscule* (dans n'importe quel ordre) qui apparaissent (en minuscule *comme en majuscule*) dans la ligne rentrée pas l'utilisateur. Par exemple, si l'utilisateur rentre la chaîne de caractères `"ceci␣EST␣une␣phrase!"`, la fonction doit renvoyer un ensemble équivalent à :

   `{'u', 's', 'r', 'p', 'e', 'i', 'a', 't', 'n', 'h', 'c'}`

   Indication : utiliser la fonction `lower` du type `str`, comme aussi la valeur `ascii_lowercase` du module `string` (qu'il faut importer).
3. Écrire une fonction `to_list(s)` qui prend en entrée un ensemble `s` d'entiers et renvoie la liste de ces entiers ordonnée du plus petit au plus grand.
4. Écrire une fonction `even(s)` qui prend en entrée un ensemble `s` d'entiers et renvoie un nouveau ensemble contenant que les nombres paires dans `s`.

II) Dictionnaires

Un autre type de donnée itérable natif dans Python est le dictionnaire, correspondant aux listes d'associations ou tableaux associatifs dans d'autres langages. À la différence des séquences, qui sont indexées par des entiers, les dictionnaires sont indexés par des clés, qui peuvent être de n'importe quel type immuable. Le plus simple est de considérer les dictionnaires comme des ensembles de paires `(clef, valeur)`, les clés devant être uniques (au sein d'un dictionnaire).

```
>>> # trois facons d'introduire les dictionnaires:
>>> d = {"macron": 43, "trump": 74, "xi": 67}      # explicite
>>> d = {n: a for n, a in zip(['macron', 'trump', 'xi'], [43, 74, 67]) }
>>> #   ^- par comprehension
>>> d
{'macron': 43, 'trump': 74, 'xi': 67}
>>> d = dict([("macron", 43), ("trump", 74), ("xi", 67)]) # constructeur dict(iterable)
>>> d
{'macron': 43, 'trump': 74, 'xi': 67}
>>> for x in d :                                   # iteration sur les clefs
...     print (x)
macron
trump
xi
>>> for x in d.items():                            # ... sur les paires (cle, val)
...     print(x)
('macron', 43)
('trump', 74)
('xi', 67)
>>> for (k, v) in d.items():                       # ... idem, avec unpacking
...     print(k)
...     print(v)
macron
```

```
43
trump
74
xi
67
>>> for x in d.values() :                          # ... sur les valeurs
...     print (x)
43
74
67
>>> for x in d.keys() :                            # ... sur les clefs (explicitement)
...     print(x)
macron
trump
xi
>>> d['macron']                                    # acces aux valeurs par clef
43
>>> d['macron'] += 1                               # les dictionaires sont mutables
>>> d['macron']
44
```

Pour plus des détails sur les dictionnaires, voir https://docs.python.org/3/library/stdtypes.html#mapping-types-dict.

Exercice 2 : dictionnaires

1. Écrire une fonction `list_to_dict(l)` qui prend une liste et produit le dictionnaire correspondant, où les clés sont les indices de la liste. Par exemple, `list_to_dict(["petit", "klein", "small"])` retourne `{0: 'petit', 1: 'klein', 2: 'small'}`.
2. Écrire une fonction `chars(w)`, où `w` est une chaîne de caractères, qui renvoie un dictionnaire dont les clés sont les caractères de `w` (sans répétition) et la valeur associée à une clé `k` est le nombre d'occurrences de `k` dans `w`. Écrire ensuite une fonction `print_dict(d)` qui affiche pour un dictionnaire `d` chaque couple clé et valeur, un couple par ligne.

   Par exemple : `print_dict(chars("aacababbcc"))` doit afficher :

   ```
   c 3
   b 3
   a 4
   ```

   Attention à l'efficacité de votre solution.
3. Écrire une fonction `merge(d1, d2)` qui prend en argument deux dictionnaires et retourne un dictionnaire formé des clés communes aux deux dictionnaires et dont les valeurs sont des 2-uplets comprenant les valeurs des deux dictionnaires.

   Par exemple :

   ```
   >>> d1 = {'r': 0.56, 't': 0.78, 'i': 0.23, 'u': 0.35}
   >>> d2 = {'i': 5, 'v': 89, 'p': 65, 't': 21, 'b': 55}
   >>> merge(d1, d2)
   {'i': (0.23, 5), 't':(0.78, 21)}
   ```
4. Écrire une fonction `inverse(d)` qui retourne le dictionnaire où les clés sont les valeurs du dictionnaire `d` passé en paramètre et les valeurs les clés de `d`. Si `d` contient plusieurs occurrences de la même valeur, `inverse(d)` retourne le dictionnaire vide.

III) Implémentation des ensembles

Une façon d'implémenter les ensembles est d'utiliser des tables de hachage. Le but d'une table de hachage est entre autres de disposer d'une structure de données permettant de tester en général très rapidement

(sans parcourir toute une liste par exemple), si un élément est contenu dans un ensemble. Dans cet exercice on propose d'implémenter une simple méthode de hachage permettant de représenter des ensembles d'entiers et de la comparer avec l'implémentation de Python des ensembles.

Exercice 3 :

Ici, une table de hachage de taille n est juste une liste en Python de taille n contenant des entiers (ou `None`). Une fonction de hachage h est une fonction qui associe à une valeur d'un domaine d'entrée un entier entre 0 et $n-1$. Par exemple, si le domaine d'entrée sont les entiers on peut choisir tout simplement $h(k) = k \bmod n$. Une table de hachage peut être utilisée par exemple pour stocker un ensemble (contenant des entiers) de taille n au maximum. Un ensemble vide correspond à une liste `t` où tous les éléments sont `None`. Ajouter un élément k à la table consiste à calculer $p = h(k)$. Ensuite, si la liste `t` à la position p est vide, on stocke k à cette position. Si la liste à la position p contient déjà k, on ne fait rien. Sinon, on a un conflit. On peut résoudre le conflit en remplaçant p avec $(p+1) \bmod n$, $(p+2) \bmod n$, etc. (cette méthode s'appelle linéaire), jusqu'à ce qu'on trouve une place libre dans la liste. Une autre méthode (appelée quadratique) consiste à remplacer p successivement par $(p+1) \bmod n$, $(p+4) \bmod n$, $(p+9) \bmod n$, etc. Pour tester si un élément est dans l'ensemble, on essaie de le retrouver comme si on voulait l'ajouter. Évidemment, si on essaie d'ajouter un élément à une liste pleine, la méthode d'ajout ne s'arrête pas.

1. Écrire une fonction `init_table(n)` qui renvoie une table de hachage (liste) de taille `n` contenant des éléments `None`.
2. Écrire une fonction `add_elt(x,n,t)` qui ajoute l'élément `x` à la table `t` qui est de taille `n` avec la méthode linéaire. Cette fonction renvoie `False` si `t` est pleine et donc elle n'a pas pu ajouter `x`, sinon elle renvoie `True`.
3. Écrire une fonction `in_table(x,n,t)` qui renvoie `True` si l'élément `x` est dans la table de taille `n`, sinon `False`.
4. Écrire une fonction `test_hash(n,l)` qui crée une table de hachage de taille `n` et ensuite effectue les opérations suivantes sur une liste d'entiers `l` :
   – ajoute tous les éléments de `l` à la table
   – teste si tous les entiers de la liste `l` sont bien dans la table.
   À la fin la fonction retourne `True` si le test a marché et `False` sinon.
5. Écrire une fonction `test_set(l)` qui crée un ensemble vide et ensuite effectue les opérations suivantes sur une liste d'entiers `l`
   – ajoute tous les éléments de l à l'ensemble
   – teste si tous les entiers de la liste l sont dans l'ensemble
   À la fin la fonction retourne `True` si le test a marché et `False` sinon.
6. Écrire une fonction `hash_benchmark(k, m, n)` qui crée une liste de `k` entiers choisi au hasard entre 0 et `m` et qui ensuite calcule et affiche le temps d'exécution de la fonction `test_hash(n, l)` et indépendamment de la fonction `test_set(l)` de sorte qu'on puisse comparer les deux. Ensuite faire différents test pour comparer les deux méthodes.
   Indication : Dans le module `random` il y a une fonction `randint` qui produit des entiers au hasard. Dans le module `time` il y a une fonction `process_time` qui donne le temps d'exécution du programme du début jusqu'au point actuel.
7. Ajouter dans les tests la méthode quadratique.
8. (*) Comment pourrait-on enlever un élément de la table ? Écrire une fonction `remove_elt(x, n, t)` qui ôte l'élément `x` à la table `t` de taille `n` avec la méthode linéaire. Cette fonction renvoie `True` si l'élément a été enlevé, elle renvoie `False` si `n` n'était pas dans `t`.
9. (*) Modifier vos fonctions pour représenter des ensembles de chaînes caractères (ou autre types de données).