Le module `collections`

Complément - niveau intermédiaire

Maintenant que nous avons terminé le tour des types de base fournis par le langage, vous pourrez trouver dans le module `collections` (https://docs.python.org/2/library/collections.html) quelques types étendus, qui fournissent des extensions souvent commodes des types de base. Nous allons en voir quelques exemples.

`OrderedDict` : un dictionnaire avec mémoire

Nous avons vu que le dictionnaire n'est pas une structure ordonnée

```
cluedo = {}
cluedo['olive'] = 'green'
cluedo['moutarde'] = 'mustard'
cluedo['rose'] = 'scarlett'
cluedo['pervenche'] = 'peacock'
for cle, valeur in cluedo.items():
    print cle, valeur
```

Avec le type `OrderedDict`, on peut conserver l'ordre dans lequel les clés sont entrées, au prix naturellement d'un surcoût en termes d'occupation mémoire :

```
from collections import OrderedDict

cluedo = OrderedDict()
cluedo['olive'] = 'green'
cluedo['moutarde'] = 'mustard'
cluedo['rose'] = 'scarlett'
cluedo['pervenche'] = 'peacock'
for cle, valeur in cluedo.items():
    print cle, valeur
```

`defaultdict` : un dictionnaire avec initialisation automatique

`defaultdict` est une spécialisation du type dictionnaire. Par opposition avec le type `dict` standard, lorsqu'on fait référence à une clé manquante, un mécanisme de `factory` (http://en.wikipedia.org/wiki/Factory_%28object- oriented_programming%29) entre en jeu pour initialiser la valeur associée à la clé.

Par exemple, supposons qu'on ait besoin de gérer un dictionnaire dont les valeurs sont des listes.

```
# on dispose d'enregistrements comme par exemple
enregistrements = [('a', 10), ('b', 20), ('a', 30), ('b', 40)]

# on veut construire un dictionnaire qui
# avec ces entrées ressemblerait à
# {'a' : [10, 30], 'b' : [20, 40]}
```

Avec le type `dict` standard, il faut se livrer à une petite gymnastique du genre de :

```
cumul = {}
for cle, valeur in enregistrements:
    if cle not in cumul:
        cumul[cle] = []
    cumul[cle].append(valeur)
print cumul
```

Ou encore, un peu mieux, toujours avec `dict` mais en utilisant `setdefault` :

```
cumul = {}
for cle, valeur in enregistrements:
    cumul.setdefault(cle, []).append(valeur)
print cumul
```

Avec `defaultdict` on peut préciser comment les valeurs doivent être initialisées, le code est plus lisible :

```
from collections import defaultdict

# la valeur par défaut est une liste vide
cumul = defaultdict(list)
for cle, valeur in enregistrements:
    cumul[cle].append(valeur)
print cumul
```

Complément - niveau avancé

Autres types

Référez-vous à la section sur le module `collections` (https://docs.python.org/2/library/collections.html) dans la documentation standard pour davantage de détails sur les autres types offerts par ce module, comme

- `Counter` (https://docs.python.org/2/library/collections.html#collections.Co unter) une autre spécialisation du type `dict`, dont les valeurs sont des entiers, spécialisée pour compter des occurrences;
- deque (https://docs.python.org/2/library/collections.html#collections.deque) une spécialisation du type `list` optimisée pour ajouter/enlever des éléments aux deux extrémités de la liste;

- namedtuple (https://docs.python.org/2/library/collections.html#collections.n amedtuple) ici il ne s'agit pas d'un type à proprement parler, mais d'une fonction qui permet de créer des types.