`collections`

Introduction

`collections` contient des conteneurs de données spécialisés qui offrent une alternative aux conteneurs généraux de python. Ces conteneurs vont souvent plus loin que les conteneurs de données *built-in* et ont des fonctionnalités plus avancées que nous allons voir ici.

Voici donc les conteneurs dont nous allons parler:

Conteneur	**Utilité**
namedtuple()	une fonction permettant de créer une sous-classe de tuple avec des champs nommés
deque	un conteneur ressemblant a une liste mais avec ajout et suppression rapide a chacun des bouts
ChainMap	permet de linker entre eux plusieurs mappings ensemble pour les gérer comme un tout
Counter	Permet de compter les occurences d'objets hachable
OrderedDict	une sous classe de dictionnaire permettant de savoir l'ordre des entrées
defaultdict	une sous classe de dictionnaire permettant de spécifier une valeur par défaut dans le constructeur

namedtuple()

tout d'abord avant d'utiliser la fonction `namedtuple()` il faut comprendre ce qu'est un tuple. un tuple est une collection immuable de données souvent hétérogène.

```
>>> t = ("cheval", "voiture", "bateau")
>>> t
('cheval', 'voiture', 'bateau')
>>> t[0]
'cheval'
>>> t[-1]
'voiture'
```

ci-dessus on remarque qu'on peut atteindre les champs de notre tuple seulement en spécifiant son index. En utilisant la fonction `namedtuple()` pour créer notre tuple, on peut nommer ses champs.

```
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)     # instanciation par position ou en utilisant le nom du champs
>>> p[0] + p[1]             # indexable comme les tuples de base (11, 22)
33
>>> x, y = p                # on peut le diviser en plusieurs variables (comme un tuple normal)
>>> x, y
(11, 22)
>>> p.x + p.y               # les champs sont accessibles par nom
33
```

```
>>> p                       # lisible dans un style nom=valeur
Point(x=11, y=22)
```

quelques fonctions

mytuple._make(iterable)

cette fonction permet de créer un tuple a partir d'un objet *iterable*.

```
>>> t = [11, 22]

>>> Point(*t)
Point(11, 12)

>>> Point._make(t)
Point(x=11, y=22)
```

mytuple._asdict()

cette fonction retourne un nouveau OrderedDict qui *map* les noms de champs avec leurs valeurs.

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
OrderedDict([('x', 11), ('y', 22)])
```

mytuple._replace(key=args)

cette fonction permet de retourner une nouvelle instance de notre tuple avec une valeures modifiée.

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)
```

mytuple._fields

cette fonction permet de récupérer les noms des champs de notre tuple. elle est utile si on veut créer un nouveau tuple avec les champs d'un tuple existant.

```
>>> p._fields            # retourne les noms de champs
('x', 'y')
>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields) #on créé un nouveau tuple avec le
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

deque

la classe **`collections.deque`** est une généralisation des liste et des piles. les deque sont thread-safe et supporte l'ajout d'une valeur de chaque côté *(pile, liste)*. La preformance lors de l'ajout d'une valeur peut importe le côté est de O(1). Même si les objets de type *list* supportent des opérations similaires elles sont plus

optimisées pour des opérations qui ne change pas leur taille alors qu" un `pop()` ou un `insert()` ont une complexité O(n).

`class collections.deque([iterable[, maxlen]])` cette instruction retourne un deque contenant les valeurs de `iterable` (s'il n'est pas spécifié le deque est vide) et l'argument `maxlen` permet de spécifier une taille maximum (la taille n'a pas de limite s'il nes pas spécifié).

```
>>> d = deque('abc')                # créé un nouveau deque avec 3 valeurs
>>> for elem in d:                  # itères sur les éléments de notre deque
...     print(elem)
a
b
c
```

quelques fonctions

append(x), appendleft(x), extend(iterable) et extendleft(iterable)

`append` ajoute une seule valeure du côté droit du deque et `appendleft` du côté gauche alors que `extend` et `extendleft` permettent d'ajouer plusieurs éléments d'un coup.

```
>>> d.append('z')
>>> d.appendleft('r')
>>> d
deque(['r', 'a', 'b', 'c', 'z'])
>>> d.extend('jkl')
>>> d
deque(['r', 'a', 'b', 'c', 'z','j','k','l'])
```

pop(), popleft(), remove(val) et clear()

`pop` et `popleft` permettent de faire sortire un objet de notre deque alors que `remove` supprime la première occurence de la val passée en paramètre et finalement `clear` vide le deque.

```
>>> d.clear()
>>> d.extends('abc')
>>> d.remove('b')
>>> d
deque(['a', 'c'])
>>> d.pop()
'c'
>>> d.popleft()
'a'
```

ChainMap

`collections.ChainMap` permet de linker plusieurs mappings pour qu'ils soient géré comme un seul. C'est souvent plus rapide que de créer un nouveau dictionnaire et faire plusieurs `update()`.

`class collections.ChainMap(*maps)` cette fonction nous retourne une nouvelle ChainMap. Si il n'y a pas de maps spécifiés en paramètres la ChainMap sera vide.

```
>>> from collections import ChainMap
>>> x = {'a': 1, 'b': 2}
```

```
>>> y = {'b': 10, 'c': 11}
>>> z = ChainMap(y, x)
>>> for k, v in z.items():
      print(k, v)
a 1
c 11
b 10
```

Dans cet exemple on remarque que la clé b a pris la valeur 10 et pas 2 car `y` est passé avant `x` dans le constructeur de ChainMap.

Counter

`collections.Counter` est une sous classe de **`dict`**. qui permet de compter des objets *hachable*. En fait c'est un dictionnaire avec comme clé les éléments et comme valeurs leur nombre.

`class collections.Counter([iterable-or-mapping])` ceci nous retourne un Counter. L'argument permet de spécifier ce que l'on veut mettre dedans et qui doit être compté. Voici un exemple :

```
>>> c = Counter()                           # compteur vide
>>> c = Counter('gallahad')                 #compteur avec un iterable
>>> c = Counter({'red': 4, 'blue': 2})      # un compteur avec un mapping
>>> c = Counter(cats=4, dogs=8)             #un compteur avec key=valeur
```

Contrairement à un dictionnaire si on demande une valeur n'étant pas dans notre liste il retourne 0 et non pas `KeyError`

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # clé inconnue
0
```

quelques fonctions

elements()

retourne une liste de tous les éléments du compteur :

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common([n])

retourne les n éléments les plus présents dans notre compteur :

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('r', 2), ('b', 2)]
```

substract([iterable or mapping])

permet de soustraire des éléments d'un compteur (mais pas de les supprimer) :

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

OrderedDict

les **`collections.OrderedDict`** sont comme les **`dict`**. mais ils se rappelent l'ordre d'entrée des valeurs. Si on itère dessus les données seront retournées dans l'ordre d'ajout dans notre dict.

`class collections.OrderedDict([items])` cette fonction nous retourn un OrderedDict.

Quelques méthodes

popitem(last=True)

Cette fonction fait sortir une paire clé valeur de notre dictionnaire et si l'argument last est a `` `True `` alors les pairs seront retournée en LIFO sinon ce serra en FIFO.

move_to_end(key, last=True)

Cette fonction permet de déplacer une clé à la fin de notre dictionnaire si last est à `True` sinon au début de notre dict.

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d.keys())
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d.keys())
'bacde'
```

defaultdict

La classe **`collections.defaultdict`** est une sous classe de **`dict`**. Elle ajoute une variable et une fonction à la classe **`dict`**. `class collections.defaultdict([default_factory[, ...]])` cette commande nous retourne un objet de type defaultdict.L'argument `default_factory` est par défaut à `None` et les reste des arguments sont traité comme si on utilisait le constructeur de dict.

La fonction ajoutée par defaultdict est `__missing(key)__` elle est appelée par `__getitem()__` de la classe **`dict`**.

l'argument `default_factory` permet de spécifier quelle structure de données va correspondre à une clé dans notre defaultdict. Voici 2 exemples pour mieux comprendre:

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
```

```
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Dans cette exemple on initialise `default_factory` comme une **`list`** ce qui nous permet d'utiliser `append()` pour ajouter des éléments à la liste correspondant à une clé donnée.

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

Dans cet exemple on va utiliser un int au lieu d'une liste et notre defaultdict va s'utiliser comme un compteur.

Conclusion

Chacun des conteneurs vu dans ce tutoriel a une utilité bien définie alors choisissez sagement votre conteneur en fonction de votre problème pour vous simplifier la vie.