

Itérateurs

Complément - niveau intermédiaire

Dans ce complément nous allons :

- tout d'abord dire quelques mots du module `itertools` qui fournit sous forme d'itérateurs des utilitaires communs qui peuvent être très utiles ;
- puis dans la partie avancée du complément nous allons voir, comme un autre exemple d'itérateurs, comment implémenter notre propre itérateur des permutations d'un ensemble fini.

Le module `itertools`

À ce stade, j'espère que vous savez trouver la documentation du module (<https://docs.python.org/2/library/itertools.html>) que je vous invite à avoir sous la main.

```
❏ import itertools
```

Comme vous le voyez dans la doc, les fonctionnalités de `itertools` tombent dans 3 catégories :

- des itérateurs infinis, comme par exemple `cycle`,
- des itérateurs pour énumérer les combinatoires usuelles en mathématiques, comme les permutations, les combinaisons, le produit cartésien, etc.,
- et enfin des itérateurs correspondants à des traits que nous avons déjà rencontrés, mais implémentés sous forme d'itérateurs.

À nouveau, toutes ces fonctionnalités sont offertes **sous la forme d'itérateurs**.

Pour détailler un tout petit peu cette dernière famille, signalons :

- `chain` qui permet de **concaténer** plusieurs itérables sous la forme d'un **itérateur** :

```
❏ for x in itertools.chain((1, 2), [3, 4]):  
    print x
```

- `izip` qui comme son nom l'indique fait comme `zip` mais sous la forme d'un itérateur :

```
❏ for a,b in itertools.izip((1, 2), [3, 4]):  
    print a, b
```

- `islice` qui est une généralisation de `xrange`. Vous vous souvenez que `xrange` est

similaire à `range` mais renvoie déjà un itérateur. `islice` permet de travailler sur une séquence qui n'est pas les premiers entiers :

```
import string
support = string.ascii_lowercase
print 'rappel string.ascii_lowercase=', support

# range
for x in range(2):
    print x
# islice
for x in itertools.islice(support, 2):
    print x
```

- `imap` et `ifilter` qui sont les équivalents de `filter` et `map`. Ces deux- là sont donc à mettre en rapport avec les fonctions génératrices que l'on a vues dans la séquence précédente, dans le sens que nous allons expliciter.

Vous vous souvenez que `map` et `filter` sont devenus obsolètes avec les compréhensions de liste. Sauf que, les compréhensions retournent, eh bien justement, une liste. Alors qu'en fait, une fois qu'on s'est habitués à penser en termes d'itérateurs, on réalise que c'est souvent dommage que les compréhensions ne retournent pas plutôt un itérateur aussi.

Dans tous les cas, vous voyez que `imap` et `ifilter` sont un peu aux expressions génératrices ce que `map` et `filter` sont aux compréhensions de liste.

Complément - niveau avancé

Implémenter un itérateur de permutations

C'est quoi déjà les permutations ?

En guise de rappel, l'ensemble des permutations d'un ensemble fini correspond à toutes les façons d'ordonner ses éléments ; si l'ensemble est de cardinal n , il possède $n!$ permutations : on a n façons de choisir le premier élément, $n-1$ façons de choisir le second, etc.

Un itérateur sur les permutations est disponible au travers du module standard `itertools`. Cependant il nous a semblé intéressant de vous montrer comment on pourrait écrire nous-mêmes cette fonctionnalité, de manière relativement simple.

Pour illustrer le concept, voici à quoi ressemblent les 6 permutations d'un ensemble à trois éléments :


```
from itertools import permutations

set = {1, 2, 3}
```

```
for p in permutations(set):  
    print p
```

Une implémentation

Voici une implémentation possible pour un itérateur de permutations :

```
 class Permutations:  
    """  
    Un itérateur qui énumère les permutations de n  
    sous la forme d'une liste d'indices commençant à 0  
    """  
    def __init__(self, n):  
        # le constructeur bien sûr ne fait (presque) rien  
        self.n = n  
        # au fur et à mesure des itérations  
        # le compteur va aller de 0 à n-1  
        # puis retour à 0 et comme ça en boucle sans fin  
        self.counter = 0  
        # on se contente d'allouer un itérateur de rang n-1  
        # si bien qu'une fois qu'on a fini de construire  
        # l'objet d'ordre n on a n objets Permutations en tout  
        if n >= 2:  
            self.subiterator = Permutations(n-1)  
  
        # pour satisfaire le protocole de l'itérable  
        def __iter__(self):  
            return self  
  
        # c'est ici bien sûr que se fait tout le travail  
        def next(self):  
  
            # pour n == 1  
            # le travail est très simple  
            if self.n == 1:  
                # on doit renvoyer une fois la liste [0]  
                # car les indices commencent à 0  
                if self.counter == 0:  
                    self.counter += 1  
                    return [0]  
                # et ensuite c'est terminé  
            else:  
                raise StopIteration  
  
            # pour n >= 2  
            # lorsque counter est nul,  
            # on traite la permutation d'ordre n-1 suivante  
            # si next() lève StopIteration on n'a qu'à laisser passer  
            # car en effet c'est qu'on a terminé  
            if self.counter == 0:  
                self.subsequence = self.subiterator.next()  
            #
```

```

# on insère alors n-1 (car les indices commencent à 0)
# successivement dans la sous-séquence
#
# naïvement on écrirait
# result = self.subsequence[0:self.counter] \
#     + [self.n - 1] \
#     + self.subsequence[self.counter:self.n-1]
# mais c'est mettre le nombre le plus élevé en premier
# et donc à itérer les permutations dans le mauvais ordre,
# en commençant par la fin
#
# donc on fait plutôt une symétrie
# pour insérer en commençant par la fin
cutter = self.n-1 - self.counter
result = self.subsequence[0:cutter] + [self.n - 1] \
        + self.subsequence[cutter:self.n-1]
#
# on n'oublie pas de maintenir le compteur et de
# le remettre à zéro tous les n tours
self.counter = (self.counter+1) % self.n
return result

# la longueur de cet itérateur est connue
def __len__(self):
    import math
    return math.factorial(self.n)

```

Ce qu'on a essayé d'expliquer dans les commentaires, c'est qu'on procède en fin de compte par récurrence. Un objet `Permutations` de rang n possède un sous-itérateur de rang $n-1$ qu'on crée dans le constructeur. Ensuite l'objet de rang n va faire successivement (c'est-à-dire à chaque appel de `next()`) :

- appel **0** :
 - demander à son sous-itérateur une permutation de rang $n-1$ (en lui envoyant `next`),
 - la stocker dans l'objet de rang n , ce sera utilisé par les n premier appels,
 - et construire une liste de taille n en insérant $n-1$ à la fin de la séquence de taille $n-1$,
- appel **1** :
 - insérer $n-1$ dans la même séquence de rang $n-1$ mais cette fois 1 cran avant la fin,
- ...
- appel **$n-1$** :
 - insérer $n-1$ au début de la séquence de rang $n-1$,
- appel **n** :
 - refaire `next()` sur le sous-itérateur pour traiter une nouvelle sous-séquence,
 - la stocker dans l'objet de rang n , comme à l'appel **0**, pour ce bloc de n appels,
 - et construire la permutation en insérant **$n-1$** à la fin, comme à l'appel **0**,

■ ...

On voit donc le caractère cyclique d'ordre n qui est matérialisé par `counter`, que l'on incrémente à chaque boucle mais modulo n – notez d'ailleurs que pour ce genre de comportement on dispose aussi de `itertools.cycle` comme on le verra dans une deuxième version, mais pour l'instant j'ai préféré ne pas l'utiliser pour ne pas tout embrouiller ;)

La terminaison se gère très simplement, car une fois que l'on a traité toutes les séquences d'ordre $n-1$ eh bien on a fini, on n'a même pas besoin de lever `StopIteration` explicitement, sauf bien sûr dans le cas $n=1$.

Le seul point un peu délicat, si on veut avoir les permutations dans le "bon" ordre, consiste à commencer à insérer $n-1$ par la droite (la fin de la sous-séquence).

Discussion

Il existe certainement des tas d'autres façons de faire bien entendu. Le point important ici, et qui donne toute sa puissance à la notion d'itérateur, c'est **qu'à aucun moment on ne construit** une liste ou une séquence quelconque de **$n!$ termes**.

C'est une erreur fréquente chez les débutants que de calculer une telle liste dans le constructeur, mais procéder de cette façon c'est aller exactement à l'opposé de ce pourquoi les itérateurs ont été conçus ; au contraire, on veut éviter à tout prix le coût d'une telle construction.

On peut le voir sur un code qui n'utiliserait que les 20 premières valeurs de l'itérateur, vous constatez que ce code est immédiat :

```
def show_first_items(iterable, nb_items):
    """
    montre les <nb_items> premiers items de iterable
    """
    print "Il y a {} items dans l'iterable".format(len(iterable))
    for i,item in enumerate(iterable):
        print item
        if i >= nb_items:
            print '....'
            break

show_first_items(Permutations(12), 20)
```

Ce tableau vous montre par ailleurs sous un autre angle comment fonctionne l'algorithme, si vous observez le 11 qui balaie en diagonale les 12 premières lignes, puis les 12 suivantes, etc..

Ultimes améliorations

Dernières remarques, sur des améliorations possibles – mais tout à fait optionnelles :

- le lecteur attentif aura remarqué qu'au lieu d'un entier `counter` on aurait pu profitablement utiliser une instance de `itertools.cycle`, ce qui aurait eu l'avantage d'être plus clair sur le propos de ce compteur;

- aussi dans le même mouvement, au lieu de se livrer à la gymnastique qui calcule `cutter` à partir de `counter`, on pourrait dès le départ créer dans le cycle les bonnes valeurs en commençant à `n-1`.

C'est ce qu'on a fait dans cette deuxième version; après avoir enlevé la loghorrée de commentaires ça redevient presque lisible ;)

```
import itertools
```

```
class Permutations2:
    """
    Un itérateur qui énumère les permutations de n
    sous la forme d'une liste d'indices commençant à 0
    """
    def __init__(self, n):
        self.n = n
        # on commence à insérer à la fin
        self.cycle = itertools.cycle(range(n)[::-1])
        if n >= 2:
            self.subiterator = Permutations(n-1)
        # pour savoir quand terminer le cas n==1
        if n == 1:
            self.done = False

    def __iter__(self):
        return self

    def next(self):
        cutter = self.cycle.next()

        # quand n==1 on a toujours la même valeur 0
        if self.n == 1:
            if not self.done:
                self.done = True
                return [0]
            else:
                raise StopIteration

        # au début de chaque séquence de n appels
        # il faut appeler une nouvelle sous-séquence
        if cutter == self.n-1:
            self.subsequence = self.subiterator.next()
        # dans laquelle on insère n-1
        return self.subsequence[0:cutter] + [self.n-1] \
            + self.subsequence[cutter:self.n-1]

# la longueur de cet itérateur est connue
def __len__(self):
    import math
    return math.factorial(self.n)
```

```
show_first_items(Permutations2(5), 20)
```