Itérable et itérateur

Complément - niveau basique

Nous résumons dans ce complément les notions d'itérable et d'itérateur.

Itérable

On appelle **itérable** un objet qui peut être **l'objet d'une boucle `for`**

Pour qu'un objet soit itérable il faut

- qu'il dispose d'une méthode `__iter__()` qui renvoie un **itérateur**
- ou qu'il dispose d'une méthode `__getitem__()` qui permette un accès séquentiel par des indices commençant à 0.

Le second point s'applique à quelques types builtin comme `str` (qui donc n'a pas de méthode `__iter__()`), mais en règle générale on est dans le premier cas, et nous y reviendrons en semaine 6 lorsque nous verrons comment rendre nos objets itérables.

Itérateur

Un itérateur est donc **attaché à un itérable**, et sa fonction est de **mémoriser l'état d'une itération** sur cet itérable.

La façon usuelle d'implémenter un itérable consiste à exposer une méthode `next()` qui renvoie l'item suivant à balayer lors de l'itération, et qui lève l'exception `StopIteration` si on se trouve en fin de boucle.

Notez qu'en python3 on a renommé `next` en `__next__`, ce qui est plus en accord avec la convention de nommage des méthodes qui permettent de redéfinir le comportement du langage sur des types utilisateur.

Résumé

- Pour pouvoir faire une boucle `for` sur un objet `spam`, il faut que `spam` soit **itérable**, c'est-à-dire que `spam` a une méthode`__iter__()` qui renvoie un itérateur; l'**itérateur** doit implémenter une méthode `next()` qui renvoie l'élément suivant dans la boucle ou lève l'exception `StopIteration` s'il n'y a plus d'éléments à renvoyer.
- L'**itérateur** est l'objet en charge de conserver l'état de la boucle sur l'itérable, mais rien n'empêche un objet d'être son propre itérateur. Cependant, lorsqu'on fait ce choix, il n'est pas possible - comme on le verra plus tard - d'écrire deux boucles imbriquées sur le même objet, puisqu'on ne dispose que d'un seul itérateur pour mémoriser l'état de

deux boucles. C'est pourquoi les types ***builtin*** itérables ont tous leurs itérateurs implémentés sous la forme d'un objet distinct (sauf, comme on le verra bientôt, dans le cas des fichiers).

Complément - niveau avancé

Pour terminer, signalons que pour rendre tout le modèle clos, on décide par convention qu'un **itérateur** doit toujours pouvoir **être utilisé comme un itérable** - dit plus simplement, on peut faire une boucle sur un itérable ou sur son itérateur.

Pour la cohérence de l'ensemble on décide donc qu'un itérateur, dans le cas où il est un objet distinct de l'itérable, doit également implémenter la méthode `__iter__()` qui retourne l'itérateur lui-même.

Voyons tout ceci sur un exemple simple

```
iterable = range(2)

# on calcule un itérateur pour la liste
iterator = iterable.__iter__()

# ce n'est pas l'objet liste lui même
print iterator is iterable


# vérifions si cet itérateur est itérable:
# il a une méthode __iter__()
# et il se trouve qu'elle renvoie bien l'iterateur lui-même
print iterator.__iter__() is iterator


# l'itérateur est bien itérable;
# ce qui fait qu'on peut aussi bien faire une boucle sur l'iterable lui-même
for i in iterable:
    print 'iterable', i

# que sur l'iterateur
for i in iterator:
    print 'iterator', i
```

Il y a une différence de taille toutefois, c'est qu'à chaque fois qu'on fait une boucle sur l'itérable on rappelle `__iter__()`, ce qui a pour résultat de créer un nouvel itérateur:

```
# on peut imbriquer deux boucles sur la liste - l'itérable
for i in iterable:
    for j in iterable:
        print 'iterable', i, 'x', j


# si on fait la même chose avec un iterateur
```

```
# que l'on crée à la main, on a une mauvaise surprise
iterator = iterable.__iter__()
for i in iterator:
    for j in iterator:
        print 'iterator', i, 'x', j
```

En fait, l'iterateur a été mis à contribution deux fois et a signalé la fin de la boucle.

Épilogue

Tous ces exemples visent uniquement à décortiquer en profondeur le fonctionnement de la boucle `for` et des itérateurs.

En pratique, **on n'appelle pas soi-même** `__iter__()`, c'est la boucle `for`, par exemple, qui crée l'itérateur.

Dans l'exemple

```
for i in iterable:
    for j in iterable:
        print 'iterable', i, 'x', j
```

les deux boucles `for` créent chacune leur itérateur, un pour `i` et un pour `j`, on n'a besoin de s'occuper de rien et tout fonctionne comme souhaité.