W3-S8-C4-iterateurs-et-performances

December 15, 2014

1 Itérateur et performances

1.1 Complément - niveau basique

Dans ce complément, nous allons voir pourquoi il est bien souvent preférable d'utiliser un itérateur comme sujet d'une boucle `for`, plutôt que d'itérer sur une énumération explicite comme une liste.

1.1.1 Digression sur le module `time`

Dans ce complément, nous allons faire un peu de mesures de temps d'exécutution et pour cela nous allons utiliser le module `time`.

En fait nous n'aurons besoin que d'une seule des fonctions du module, qui retourne l'heure de l'horloge interne, en secondes :

```
In []: import time
       print time.time()
```

Pour les curieux ce nombre correspond au nombre de secondes écoulées depuis le 1^{er} Janvier 1970.

Le point important ici est que, pour savoir combien de temps prend une opération, on peut faire quelque chose comme

```
In []: from time import time
       # on enregistre l'heure au début
       debut = time()
       # uniquement pour illustrer notre méthode de mesure,
       # nous utilisons time.sleep, qui attend à ne rien faire pendant la durée indiquée
       from time import sleep
       sleep(0.8)
       # on enregistre l'heure à la fin
       fin = time()
       # si tout se passe bien, la durée observée: fin-debut
       # doit nous donner environ 0.8 seconde
       print "Durée observée pour sleep {}s".format(fin-debut)
```

1.1.2 Calculs non-instantanés dans un notebook

Vous remarquerez d'ailleurs que pendant le temps du `sleep`, le nombre en face du label `In[]` est remplacé par une étoile, qui indique que votre interpréteur python est occupé.

Si en manipulant les exemples vous lancez par erreur un calcul trop long, l'interpréteur reste occupé jusqu'à en avoir fini avec ce calcul, et ne pourra pas évaluer d'autres cellules tant qu'il n'aura pas fini. Pensez dans ce cas à faire dans votre notebook *Kernel -> Restart* pour redémarrer votre interpréteur.

1.1.3 `range` *vs* `xrange`

Reprenons notre comparaison entre boucles `for` sur liste et sur itérateur. Pour cela, commençons par un cas très simple. Nous avons vu déjà à de nombreuses reprises la fonction `range`. Il existe également une fonction `xrange` qui renvoie un **itérateur** équivalent, et non pas une liste :

```
In []: for i in range(3):
           print 'range', i

       for i in xrange(3):
           print 'xrange', i
```

1.1.4 Quelle différence alors ?

Imaginez maintenant qu'au lieu de 3 éléments on en ait beaucoup plus; nous allons expérimenter avec plusieurs tailles

```
In []: # de 100.000 à 50 millions
       tailles = [10**5, 10**6, 10**7, 5*10**7]
```

Voyons le temps que prend uniquement la **construction** d'une grosse liste.

```
In []: import time

       for taille in tailles:
           beg = time.time()
           liste = range(taille)
           end = time.time()
           print "Création de la liste de taille {}: {}s".format(taille,end-beg)
```

Si maintenant on construit un itérateur équivalent on mesure un temps beaucoup plus court :

```
In []: import time

       for taille in tailles:
           beg = time.time()
           iterateur = xrange(taille)
           end = time.time()
           print "Création de l'itérateur de taille {}: {}s".format(taille,end-beg)
```

On peut voir que, en tendance, la **création d'un itérateur** de type `xrange` est quasiment **instantanée** quelle que soit la taille, alors que la création d'une liste équivalente prend **un temps beaucoup plus important** et d'autant plus long que la liste est grande.

1.1.5 Ce qu'il faut retenir

Pour résumer ce complément, retenez que :

- la **construction d'une liste**, surtout si elle est très longue, peut avoir un **coût non négligeable** en temps et en mémoire;
- aussi c'est une bonne idée de s'efforcer de **ne créer une liste** que lorsque c'est **réellement nécessaire**;
- et dans tous les autres cas, c'est à dire à chaque fois que la liste n'est qu'un **accessoire de calcul**, et ne représente pas une fin en soi, il faut **préférer** l'utilisation d'**itérateurs**.

1.2 Complément - niveau intermédiaire

1.2.1 Allouer et initialiser de la mémoire prend du temps

Ce phénomène peut vous paraître surprenant si vous n'êtes pas familier avec l'informatique. À première vue, si on juge superficiellement, on peut se demander ce qui se passe.

En fait, pour créer la liste des `taille` premiers entiers, il faut * d'abord allouer suffisamment de mémoire pour tous les ranger * et ensuite remplir les `taille` cases de la liste avec les valeurs

Ces deux opérations semblent banales, mais elles prennent néanmoins un peu de temps, qui à grande échelle devient sensible, comme nous venons de l'expérimenter.

1.2.2 Un itérateur est un objet minuscule

A contrario, un itérateur du type `xrange` ne **contient presque rien**. Cela sera approfondi en semaine 6, mais pour anticiper un peu la fonction d'un iterateur `xrange` consiste uniquement à mémoriser les paramètres de la boucle, et à quelle étape on en est rendu à un moment donné.

Ce qui explique le temps très faible, et constant en fonction de `taille`, que l'on a observé pour la création de nos itérateurs.

1.2.3 Des itérateurs pour tout

Au fur et à mesure de l'évolution de python-2, on a petit à petit ajouté des utilitaires pour calculer des itérateurs plutôt que des listes. C'est le cas par exemple avec les fonctions et méthodes suivantes :

Original	Amélioré
`range`	`xrange`
`dict.keys`	`dict.iterkeys`
`dict.values`	`dict.itervalues`
`dict.items`	`dict.iteritems`

1.2.4 python-3

Parmi les grandes différences entre python-2 et python-3, il y a ceci : pour toutes les méthodes ci-dessus, la sémantique est sytématiquement de retourner un itérateur.

Ainsi en python-3, `range(10)` retourne un objet itérateur. Et il n'y a pas de fonction `xrange`, sachant qu'on peut toujours construire une liste liste en appelant explicitement la fonction `list` comme ceci :

```
~ $ python3
Python 3.4.1 (default, Sep 20 2014, 19:44:17)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.40)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> range(10)
range(0, 10)
>>>
>>> xrange(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'xrange' is not defined
>>>
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>>
```

1.3 Complément - niveau avancé

Pour finir, et pour revenir sur les mesures de performances, voici une astuce qui permet de lancer de petits benchmarkes dans un terminal :

```
$ python -m timeit 'liste=range(10**6)' 'for x in liste: x+1'
10 loops, best of 3: 50.5 msec per loop
```

Ceci met en jeu un certain nombre de choses nouvelles: * python avec l'option -m permet d'importer un module, en l'occurrence ici le module `timeit`; * avec cette forme on peut passer à `timeit` plusieurs instructions; ici nous avons deux instructions, une pour initialiser `liste`, la seconde pour lancer la boucle `for`; * il est possible d'écrire des instructions sur une seule ligne. Ici le dernier argument passé à python est

```
for x in liste: x+1
```

qui est interprété comme une seule ligne. Cette pratique doit absolument rester limitée à de tels usages spécifiques.

Cette forme est pratique notamment parce que `timeit` fait, comme on le voit, plusieurs essais successifs qui donnent un résultat plus représentatif. C'est pourquoi vous la trouverez fréquemment utilisée dans les forums de discussion autour de python.