

# Python - cours intensif pour les scientifiques

## Partie 4 : Optimiser le code : timeit, profilage, Cython, SWIG et PyPy

Par Rick Muller - [Raphaël Seban](#) (traducteur)

Date de publication : 13 août 2014

Pour de nombreux scientifiques, Python est LE langage de programmation par excellence, car il offre de grandes possibilités en analyse et modélisation de données scientifiques avec relativement peu de charge de travail en termes d'apprentissage, d'installation ou de temps de développement. C'est un langage que vous pouvez intégrer en un week-end, puis utiliser une vie durant.

Retrouvez les trois premières parties de cet article :

- **Partie 1 : Présentation de Python**
- **Partie 2 : Modules NumPy et SciPy**
- **Partie 3 : Python avancé**

Les commentaires et les suggestions d'amélioration sont les bienvenus, alors, après votre lecture, n'hésitez pas.

En complément sur Developpez.com

- [Condensé Python pour les scientifiques - Partie 1](#)
- [Condensé Python pour les scientifiques - Partie 2](#)
- [Condensé Python pour les scientifiques - Partie 3](#)
- [Tutoriel Matplotlib](#)

I - Introduction.....	3
II - Timeit.....	3
III - Profilage.....	4
IV - Autres méthodes d'optimisation.....	5
V - Jeu : trouver des nombres premiers.....	5
VI - Remerciements.....	7
VII - Remerciements Developpez.....	7

## I - Introduction

La toute première règle de l'optimisation de code est de ne pas chercher à optimiser du tout. Comme le disait Donald Knuth :

« Nous devrions oublier les petits gains d'efficacité, disons durant 97 % du temps de développement : l'optimisation en amont est la mère de tous les maux. »

La seconde règle de l'optimisation de code est de n'optimiser que lorsqu'on le juge réellement nécessaire. Le langage Python dispose de deux outils pour nous aider en ce sens : un module de chronométrage **# timeit #** et un module de **profilage de code**. Nous aborderons ces outils dans cette rubrique, ainsi que les différentes techniques permettant d'optimiser le code que vous pourriez estimer trop lent.

## IP[y]: Notebook

Cette partie est consultable également au format « notebook » en suivant le lien : **Notes : Partie 4 - Optimiser le code : timeit, profilage, Cython, SWIG et PyPy.**

## II - Timeit

De deux fonctions similaires, le module **timeit** permet de déterminer laquelle est la plus rapide. Souvenez-vous de la fonction factorielle que nous avons écrite précédemment, alors que nous avons remarqué que Python disposait déjà de sa propre fonction factorielle dans le module standard **math**. Y a-t-il une différence de rapidité entre ces deux fonctions ? Le module **timeit** va nous aider à en savoir plus. Par exemple, **timeit** peut nous indiquer la durée d'exécution de chaque fonction.

In :

```
from math import factorial
%timeit factorial(20)
```

1000000 loops, best of 3: 630 ns per loop

Le signe pourcentage (%) placé devant l'appel de **timeit** est un exemple concret de fonction magique IPython, que nous ne traiterons pas ici, une sorte de **Mojo** que IPython ajoute aux fonctions pour qu'elles s'exécutent mieux dans l'environnement IPython. Pour en savoir plus, consultez le **tutoriel IPython**.

En tout cas, la fonction **timeit** évalue l'opération à 1 000 000 de boucles, puis nous informe que le meilleur temps

moyen parmi trois mesures est de 630 nanosecondes par boucle # pour calculer <sup>20!</sup> tout de même ! En comparaison :

In :

```
def fact (n):
    if n <= 0:
        return 1
    return n*fact(n-1)
%timeit fact(20)
```

100000 loops, best of 3: 6.07 us per loop

La fonction factorielle que nous avons écrite est presque 10 fois plus lente. Cela est essentiellement dû au fait que la fonction factorielle standard est écrite en C, puis appelée depuis Python, alors que notre version est entièrement écrite en Python. Si Python est particulièrement pratique à utiliser, sa souplesse et son ergonomie ont toutefois un coût en temps machine. En comparaison, le code C est plus rude à mettre en œuvre, mais sa compilation en code

machine s'avère redoutable. Si vous voulez optimiser votre code sans trop d'effort, écrivez-le avec un langage comme Python, mais délégez les parties les plus lentes à un langage comme C, puis appelez ces fonctionnalités C depuis Python. Nous aborderons quelques techniques en ce sens dans cette rubrique.

### III - Profilage

Le **profilage de code** vient en complément de **timeit** en décortiquant le temps d'exécution de diverses instructions à l'intérieur de la procédure mesurée.

Supposons que nous voulions créer une liste d'entiers pairs. Notre première tentative donne ceci :

```
In :
def evens(n):
    "Return a list of even numbers below n"
    l = []
    for x in range(n):
        if x % 2 == 0:
            l.append(x)
    return l
```

Ce code est-il suffisamment rapide ? Analysons-le avec le profileur Python :

```
In :
import cProfile
cProfile.run('evens(100000)')
```

50004 function calls in 0.036 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.029	0.029	0.036	0.036	<ipython-input-165-9d23d9d62f6b>:1 (evens)
1	0.001	0.001	0.036	0.036	<string>:1(<module>)
50000	0.005	0.000	0.005	0.000	{method 'append' of 'list' objects}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}
1	0.002	0.002	0.002	0.002	{range}

Tout cela semble correct, 0,036 secondes n'est pas un délai si *énorme* ; toutefois, en y regardant de plus près, on s'aperçoit que les appels à `append()` grèvent environ 20 % du temps total. Pouvons-nous faire mieux ? Essayons avec la technique de **liste en compréhension**.

```
In :
def evens2(n):
    "Return a list of even numbers below n"
    return [x for x in range(n) if x % 2 == 0]
```

```
In :
import cProfile
cProfile.run('evens2(100000)')
```

4 function calls in 0.022 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.019	0.019	0.021	0.021	<ipython-input-167-cbb0d0b3fc58>:1 (evens2)
1	0.000	0.000	0.022	0.022	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}
1	0.002	0.002	0.002	0.002	{range}

En remplaçant notre code par une liste en compréhension, nous avons quasiment doublé la vitesse d'exécution !

Toutefois, il semble que la fonction `range()` prenne encore beaucoup de temps. Peut-on s'en passer ? Oui, en utilisant le générateur `xrange()` :

```
In :
def evens3(n):
    "Return a list of even numbers below n"
    return [x for x in xrange(n) if x % 2 == 0]
```

```
In :
import cProfile
cProfile.run('evens3(100000)')
```

3 function calls in 0.019 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.018	0.018	0.018	0.018	<ipython-input-169-3ee1b2b2b034>:1 (evens3)
1	0.001	0.001	0.019	0.019	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

C'est là que le profilage s'avère utile. Notre code a été amélioré d'un facteur 3 avec juste quelques modifications triviales. Nous n'aurions jamais songé à faire ces modifications si nous n'avions pas un outil de profilage aussi efficace. Imaginez un instant ce que vous pourriez faire avec des programmes plus sophistiqués.

## IV - Autres méthodes d'optimisation

Lorsque nous avons comparé nos fonctions factorielles ci-dessus, nous avons remarqué que les fonctionnalités écrites en C sont souvent plus rapides, car mieux optimisées. Une fois que nous avons identifié un goulet d'étranglement dans un programme, nous pouvons le remplacer par une version plus rapide écrite en C. On parle alors d'*extension* de Python. Vous trouverez une [documentation](#) intéressante à ce sujet. Cette façon de faire peut vite devenir fastidieuse, surtout si vous avez de nombreuses fonctionnalités à réécrire en C. Heureusement, il existe d'autres options.

Le logiciel **SWIG** (*Simplified Wrapper and Interface Generator*) permet de générer des interfaces avec non seulement Python, mais aussi Matlab™, Perl, Ruby et plein d'autres langages. SWIG peut analyser les en-têtes C d'un projet donné, puis générer les dépendances Python qui lui serviront d'interface. Utiliser SWIG s'avère substantiellement plus simple que d'écrire les fonctionnalités en C.

Le projet **Cython** comprend # entre autres # un langage d'extension de Python similaire au C. On peut, par exemple commencer par écrire une fonction en Python, puis la compiler sous forme de bibliothèque partagée (DLL, SO) qui peut ensuite être exploitée par des versions plus rapides des fonctionnalités. On peut ensuite ajouter des typages de données statiques plus quelques restrictions pour optimiser davantage le code. Cython est généralement plus facile à mettre en œuvre que SWIG.

Pour finir, **PyPy** est sans doute le moyen le plus facile d'obtenir du code rapide. PyPy compile du Python en un sous-ensemble du langage appelé RPython, qui peut être compilé et optimisé efficacement. D'après certains tests, **PyPy peut être jusqu'à 6 fois plus rapide que le Python standard**.

## V - Jeu : trouver des nombres premiers

Le site [Projet Euler](#) propose des énigmes de programmation qui auraient sans doute intéressé Euler. Le **problème n° 7** pose la question suivante :

Sachant que le 6e nombre premier de la liste 2, 3, 5, 7, 11, 13... est 13, quel est le dix-mille- unième (10001<sup>e</sup>) nombre premier ?

Pour résoudre ce problème, nous avons besoin d'une très longue liste de nombres premiers. Pour commencer, nous allons écrire une fonction basée sur le **crible d'Ératosthène** pour générer tous les nombres premiers inférieurs à n :

```
In :
def primes(n):
    """\
    From python cookbook, returns a list of prime numbers from 2 to < n

    >>> primes(2)
    [2]
    >>> primes(10)
    [2, 3, 5, 7]
    """
    if n==2: return [2]
    elif n<2: return []
    s=range(3,n+1,2)
    mroot = n ** 0.5
    half=(n+1)/2-1
    i=0
    m=3
    while m <= mroot:
        if s[i]:
            j=(m*m-3)/2
            s[j]=0
            while j<half:
                s[j]=0
                j+=m
            i=i+1
            m=2*i+3
    return [2]+[x for x in s if x]
```

```
In :
number_to_try = 1000000
list_of_primes = primes(number_to_try)
print list_of_primes[10001]
```

104759

Vous pourriez penser que Python n'est pas un choix idoine pour un tel calcul, mais même en termes de temps d'exécution, il n'a vraiment pas fallu attendre très longtemps.

```
In :
cProfile.run('primes(1000000)')
```

4 function calls in 0.323 seconds

Ordered by: standard name

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1	0.309	0.309	0.316	0.316	<ipython-input-171-0ec3aaee90fa>:1 (primes)
1	0.007	0.007	0.323	0.323	<string>:1 (<module>)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}
1	0.007	0.007	0.007	0.007	{range}

Environ 1/3 de seconde pour générer la liste de tous les nombres premiers inférieurs à un million (1 000 000). Ce serait sympa de pouvoir remplacer la fonction `range()` par `xrange()` comme précédemment, mais dans le cas ici présent, nous avons réellement besoin d'une liste et non d'un compteur.

## VI - Remerciements

Un grand merci à Alex et Tess pour tout !

Remerciements chaleureux à Barbara Muller et Tom Tarman pour leurs précieuses suggestions.


Ce document (version originale :  **A Crash Course in Python for Scientists**) est publié sous licence **Creative Commons Paternité - Partage à l'identique 3.0 non transposé**. Ce document est publié gratuitement, avec l'espoir qu'il sera utile. Merci d'envisager un don au **Fonds de soutien en mémoire de John Hunter** - article en français à **cet endroit**.



Sandia est un laboratoire pluridisciplinaire géré par la Sandia Corporation®, une filiale de la Lockheed Martin Company®, pour le compte des États-Unis d'Amérique, département de l'Énergie, administration de la Sécurité Nucléaire, sous contrat n° DE-AC04-94AL85000.



## VII - Remerciements Developpez

Nous remercions Rick Muller qui nous a aimablement autorisé à traduire son cours  **A Crash Course in Python for Scientists**.

Nos remerciements à Raphaël SEBAN (**tarball69**) pour la traduction et à Fabien (**f-leb**) pour la mise au gabarit.

Nous remercions également Malick SECK (**milkoseck**) pour sa relecture orthographique.