td1a_cython_edit

July 1, 2022

1 1A.soft - Calcul numérique et Cython

Python est très lent. Il est possible d'écrire certains parties en C mais le dialogue entre les deux langages est fastidieux. Cython propose un mélange de C et Python qui accélère la conception.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

1.1 Calcul numérique

On peut mesurer le temps que met en programme comme ceci (qui ne marche qu'avec IPython…timeit) :

[2]:
```
def racine_carree1(x) :
    return x**0.5

%timeit -r 10 [ racine_carree1(x) for x in range(0,1000) ]
```

```
296 µs ± 16.5 µs per loop (mean ± std. dev. of 10 runs, 1,000 loops each)
```

[3]:
```
import math
def racine_carree2(x) :
    return math.sqrt(x)

%timeit -r 10 [ racine_carree2(x) for x in range(0,1000) ]
```

```
260 µs ± 16 µs per loop (mean ± std. dev. of 10 runs, 1,000 loops each)
```

La seconde fonction est plus rapide. Seconde vérification :

[4]:
```
%timeit -r 10 [ x**0.5 for x in range(0,1000) ]
%timeit -r 10 [ math.sqrt(x) for x in range(0,1000) ]
```

```
236 µs ± 36.1 µs per loop (mean ± std. dev. of 10 runs, 1,000 loops each)
169 µs ± 12.6 µs per loop (mean ± std. dev. of 10 runs, 10,000 loops each)
```

On remarque également que l'appel à une fonction pour ensuite effectuer le calcul a coûté environ 100 μs pour 1000 appels. L'instruction `timeit` effectue 10 boucles qui calcule 1000 fois une racine carrée.

1.2 Cython

Le module Cython est une façon d'accélérer les calculs en insérant dans un programme python du code écrit dans une syntaxe proche de celle du C. Il existe différentes approches pour accélérer un programme python :

- Cython : on insère du code [C](http://fr.wikipedia.org/wiki/C_(langage) dans le programme python, on peut gagné un facteur 10 sur des fonctions qui utilisent des boucles de façon intensives.
- autres alternatives :
  - cffi, il faut connaître le C (ne fait pas le C++)
  - pythran
  - numba
  - ...
- PyPy : on compile le programme python de façon statique au lieu de l'interpréter au fur et à mesure de l'exécution, cette solution n'est praticable que si on a déjà programmé avec un langage compilé ou plus exactement un langage où le typage est fort. Le langage python, parce qu'il autorise une variable à changer de type peut créer des problèmes d'inférence de type.
- module implémenté en C : c'est le cas le plus fréquent et une des raisons pour lesquelles Python a été rapidement adopté. Beaucoup de librairies se sont ainsi retrouvées disponibles en Python. Néanmoins, l'API C du Python nécessite un investissement conséquent pour éviter les erreurs. Il est préférable de passer par des outils tels que
  - boost python : facile d'accès, le module sera disponible sous forme compilée,
  - SWIG : un peu plus difficile, le module sera soit compilé par la librairie soit packagé de telle sorte qu'il soit compilé lors de son l'installation.

Parmi les trois solutions, la première est la plus accessible, et en développement constant (Cython changes). L'exemple qui suit ne peut pas fonctionner directement sous notebook car Cython compile un module (fichier *.pyd) avant de l'utiliser. Si la compilation ne fonctionne pas et fait apparaître un message avec `unable for find file vcvarsall.bat`, il vous faut lire l'article Build a Python 64 bit extension on Windows 8 après avoir noté la version de Visual Studio que vous utilisez. Il est préférable d'avoir programmé en C/C++ même si ce n'est pas indispensable.

1.3 Cython dans un notebook

Le module IPython propose une façon simplifiée de se servir de Cython illustrée ici : Some Linear Algebra with Cython. Vous trouverez plus bas la façon de faire sans IPython que nous n'utiliserons pas pour cette séance. On commence par les préliminaires à n'exécuter d'une fois :

[5]:
```
%load_ext cython
```

Puis on décrit la fonction avec la syntaxe Cython :

[6]:
```
%%cython --annotate
cimport cython

def cprimes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
```

```
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

[6]: <IPython.core.display.HTML object>

On termine en estimant son temps d'exécution. Il faut noter aussi que ce code ne peut pas être déplacé dans la section précédente qui doit être entièrement écrite en *cython*.

[7]:
```
%timeit [ cprimes (567) for i in range(10) ]
```

```
5.96 ms ± 176 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

1.4 Exercice : python/C appliqué à une distance d'édition

La distance de Levenshtein aussi appelé distance d'édition calcule une distance entre deux séquences d'éléments. Elle s'applique en particulier à deux mots comme illustré par Distance d'édition et programmation dynamique. L'objectif est de modifier la fonction suivante de façon à utiliser Cython puis de comparer les temps d'exécution.

[8]:
```
def distance_edition(mot1, mot2):
    dist = { (-1,-1): 0 }
    for i,c in enumerate(mot1) :
        dist[i,-1] = dist[i-1,-1] + 1
        dist[-1,i] = dist[-1,i-1] + 1
        for j,d in enumerate(mot2) :
            opt = [ ]
            if (i-1,j) in dist :
                x = dist[i-1,j] + 1
                opt.append(x)
            if (i,j-1) in dist :
                x = dist[i,j-1] + 1
                opt.append(x)
            if (i-1,j-1) in dist :
                x = dist[i-1,j-1] + (1 if c != d else 0)
                opt.append(x)
            dist[i,j] = min(opt)
    return dist[len(mot1)-1,len(mot2)-1]

%timeit distance_edition("idstzance","distances")
```

```
106 µs ± 3.8 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Auparavant, il est probablement nécessaire de suivre ces indications :

- Si vous souhaitez remplacer le dictionnaire par un tableau à deux dimensions, comme le langage C n'autorise pas la création de tableau de longueur variables, il faut allouer un pointeur (c'est du C par du C++). Toutefois, je déconseille cette solution :
  - Cython n'accepte pas les doubles pointeurs : How to declare 2D list in Cython, les pointeurs simples si Python list to Cython.

– Cython n'est pas forcément compilé avec la même version que votre version du compilateur Visual Studio C++. Ce faisant, vous pourriez obtenir l'erreur `warning C4273: 'round' : inconsistent dll linkage`. Après la lecture de cet article, BUILDING PYTHON 3.3.4 WITH VISUAL STUDIO 2013, vous comprendrez que ce n'est pas si simple à résoudre.

Je suggère donc de remplacer `dist` par un tableau `cdef int dist [500][500]`. La signature de la fonction est la suivante : `def cdistance_edition(str mot1, str mot2)`. Enfin, Cython a été optimisé pour une utilisation conjointe avec `numpy`, à chaque fois que vous avez le choix, il vaut mieux utiliser les container numpy plutôt que d'allouer de grands tableaux sur la pile des fonctions ou d'allouer soi-même ses propres pointeurs.

1.5 Cython sans les notebooks

Cette partie n'est utile que si vous avez l'intention d'utiliser Cython sans IPython. Les lignes suivantes implémentent toujours avec Cython la fonction `primes` qui retourne les entiers premiers entiers compris entre 1 et N. On suit maintenant la méthode préconisée dans le tutoriel de Cython. Il faut d'abord créer deux fichiers :

- `example_cython.pyx` qui contient le code de la fonction
- `setup.py` qui compile le module avec le compilateur Visual Studio C++

[9]:
```
code = """
def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
"""

name = "example_cython"
with open(name + ".pyx","w") as f : f.write(code)

setup_code = """
from distutils.core import setup
from Cython.Build import cythonize

setup(
    ext_modules = cythonize("__NAME__.pyx",
                            compiler_directives={'language_level' : "3"})
)
""".replace("__NAME__",name)
```

```
with open("setup.py","w") as f:
    f.write(setup_code)
```

Puis on compile le fichier `.pyx` créé en exécutant le fichier `setup.py` avec des paramètres précis :

[10]:
```
import os
import sys
cmd = "{0} setup.py build_ext --inplace".format(sys.executable)
from pyquickhelper.loghelper import run_cmd
out,err = run_cmd(cmd)
if err != '' and err is not None:
    raise Exception(err)

[ _ for _ in os.listdir(".") if "cython" in _ or "setup.py" in _ ]
```

[10]:
```
['example_cython.c',
 'example_cython.cp38-win_amd64.pyd',
 'example_cython.pyx',
 'setup.py',
 'td1a_cython_edit.ipynb',
 'td1a_cython_edit_correction.ipynb']
```

Puis on importe le module :

[11]:
```
import pyximport
pyximport.install()
import example_cython
```

```
C:\Python395_x64\lib\site-packages\Cython\Compiler\Main.py:369: FutureWarning:
Cython directive 'language_level' not set, using 2 for now (Py2). This will
change in a later release! File: C:\xavierdupre\__home_\GitHub\ensae_teaching_cs
\_doc\notebooks\td1a_soft\example_cython.pyx
  tree = Parsing.p_module(s, pxd, full_module_name)
```

Si votre dernière modification n'apparaît pas, il faut redémarrer le kernel. Lorsque Python importe le module `example_cython` la première fois, il charge le fichier `example_cython.pyd`. Lors d'une modification du module, ce fichier est bloqué en lecture et ne peut être modifié. Or cela est nécessaire car le module doit être recompilé. Pour cette raison, il est plus pratique d'implémenter sa fonction dans un éditeur de texte qui n'utilise pas IPython.
On teste le temps mis par la fonction `primes` :

[12]:
```
%timeit [ example_cython.primes (567) for i in range(10) ]
```

```
5.98 ms ± 696 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Puis on compare avec la version écrites un Python :

[13]:
```
def py_primes(kmax):
    p = [ 0 for _ in range(1000) ]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
```

```
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result

%timeit [ py_primes (567) for i in range(10) ]
```

```
202 ms ± 23 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[14]:

[15]: