AIを利用した
ナンプレの問題自動生成

Cython化
詳細説明

2022年1月24日版

株式会社タイムインターメディア

知識工学センター

UECアライアンスセンター

目次

はじめに

Cython版では、ナンプレを作るためにアルゴリズムなどの説明はしません。それらは、Python版に書かれているので、そちらを見てください。

ここでは、Pythonのプログラムを、どのようにしてCython版に書き換えるかについて説明します。そのため、ナンプレのソルバーの書き換えだけでなく、Cythonの導入的な話を最初に入れています。

CあるいはC++言語で既にプログラミングをしている人にとっては、かなり簡単です。
それ以外のコンパイル型のプログラミング言語を使っている、特に型指定などを必要とするプログラミング言語の場合も、すんなり習得できると思います。

Cythonは、基本的には、Pythonの上位互換の高速な(高速にできる)プログラミング言語です。基本的な違いは、Pythonでは変数、関数（メソッド）に型がありませんが、Cythonでは型を指定することで、Cのように高速になります。

Cなどのコンパイラ言語を知らなくても、もちろんCythonは習得できます。Cythonではコンパイルをし、CまたはC++のソースコードを作ってから、C/C++のソースをコンパイルします。このあたりは、まとめて行う方法があるので、問題ないでしょう。

では、順番に説明していきます。

第1部　準備

まず、インストールなど、Cythonを動かせるようにする必要があります。

次に、Pythonのどの部分をCythonに書き換えるかを調べます。遅い部分だけをCythonに書き換えるだけで大丈夫で、Pythonのプログラム全体を書き換えるのは無駄です。
そのために、遅い部分を調べるツールについて説明します。

次に、非常に簡単な例を用いて、実際にPythonのプログラムをCythonにして実行速度などを調べてみましょう。

動作環境

ここの説明で使っているパソコンの動作環境を説明します。

LinuxのUbuntu20を使用しています。
PythonおよびCythonは、さまざまなOS上で動作するはずですが、上記OSでの動作について説明します。その他の環境で動作させる場合は、適当に調べて対応してください。

```
$ head /etc/os-release
NAME="Ubuntu"
VERSION="20.04.3 LTS (Focal Fossa)"
以下省略

$ lscpu
・・・略・・・
モデル名:          Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
・・・略・・・
```

i5で、Ubuntuの20を動作させています。
今どき、ノートパソコンでももっと高性能かと思いますが、これで十分Cythonの開発などができます。

Cythonのインストール

インストールは極めて簡単です。
Pythonが動くようになっていれば、以下の1行だけでCythonをインストールできます。

```
$ pip install cython
```

インストールが終わったら、バージョンを確認してみましょう。

```
$ cython -V
Cython version 0.29.24
```

プロファイリング

まず、遅いプログラムのどの部分が遅いか調べるツールを使って、問題部分を特定しましょう。実行中の各関数の呼び出し回数や実行時間などを詳細に計測し報告するプロファイラと呼ばれるツールが色々用意されているものです。ここでは、Pythonで標準で用意されているcProfileを使って調べてみます。

せっかく遅いプログラムであるPython版のナンプレ自動生成プログラムがあるので、これをプロファイリングしてみましょう。

コマンドラインからプロファイラを起動するのは、以下の形式になります。

```
python3 -m cProfile [-o output_file] [-s sort_order] test.py
      -o はプロファイルで集めたデータを出力するファイルの指定
      -s はプロファイル結果の並べ順
```

とりあえず、ヒント数24のパターン6個で6問自動生成しながら、プロファイリングしてみます。-s には tottime (トータル時間順)を指定してみました。

```
Python$ python3 -m cProfile -s tottime NP.py -s data/24P.txt
No.1   H 24
- X - - - X - - -
X - - X - - X - -
- - X - - - - X X
- - - - X - - X X
- - - X - X - - -
X X - - X - - - -
X X - - - - X - -
- - X - - X - - X
- - - X - - - X -
*SUCCESS  TRY 0

中略

No.6   H 24
```

```
- - X X - - - - -
- - X - - X - - -
- - - - - X - X X
- X X X - X - - X
- - - - - - - - -
X - - X - X X X -
X X - X - - - - -
- - - X - - X - -
- - - - - X X - -
*SUCCESS  TRY 0
- - 1 2 - - - - -
- - 4 - - 1 - - -
- - - - - 7 - 5 9
- 7 9 6 - 4 - - 8
- - - - - - - - -
8 - - 3 - 2 4 7 -
6 9 - 4 - - - - -
- - - 9 - - 2 - -
- - - - - 5 3 - -

total 6  failure 0
Time  13.455162 sec
         432865 function calls (430692 primitive calls) in 13.585 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    69894    6.311    0.000    6.469    0.000 solver.py:146(checkBlock)
    31817    2.753    0.000    2.784    0.000 solver.py:172(checkHline)
    31770    2.752    0.000    2.793    0.000 solver.py:194(checkVline)
    43074    0.732    0.000    0.732    0.000 solver.py:86(setValue)
   157869    0.588    0.000    0.604    0.000 solver.py:216(checkCell)
     4370    0.161    0.000    0.161    0.000 solver.py:236(blankCount)
     1235    0.060    0.000    0.593    0.000 solver.py:41(setProblem)
     1235    0.058    0.000   12.780    0.010 solver.py:53(checkLoop)
      112    0.027    0.000    0.027    0.000 {built-in method io.open_code}
       19    0.016    0.001    0.018    0.001 {built-in method _imp.create_dyna
      316    0.014    0.000    0.014    0.000 {built-in method builtins.compile
      112    0.011    0.000    0.011    0.000 {built-in method marshal.loads}
     2494    0.004    0.000    0.004    0.000 {method 'astype' of 'numpy.ndarra
     1237    0.004    0.000    0.004    0.000 {built-in method numpy.core._mult
  226/225    0.004    0.000    0.005    0.000 {built-in method builtins.__build
```

```
      427    0.004    0.000   13.410    0.031 generator.py:124(changeXCells)
     1235    0.003    0.000    0.017    0.000 solver.py:34(initialize)
```

コマンドラインからpyton3を直接起動してNP.pyを動かし、プロファイリングを行い、結果は、プログラムの終了直後に出力されます。

非常に長いのですが、上記はプロファイリングの最初の部分だけを示しています。
1行が、１つの関数呼び出しに対応しています。
カラムtottime（トータル時間）でソートしています。この時間は、その関数の中で消費された時間で、cumtime(累積時間)のカラムは、その関数及びその関数から呼び出された関数の合計時間です。

これをみると、最初の方は、全部solver.pyの中の関数ということが分かります。明らかに、solver.pyの中の関数が遅くて、全体が遅くなっているのが分かります。

solver.py以外は、ほとんど時間を使っていないようなので、Pythonのままで十分なようですので、solver.pyだけをPython化します。

こんなにきれいに分かれることは少なく、１つのファイルの特定のわずかの関数だけが遅いことが普通です。遅い部分を別のファイル（別モジュール）にできれば、別モジュールになった部分だけをCython化します。まあ、長いPythonのファイルの中の一部に遅い部分があり、その部分だけをCython化することも、一応出来ます。

コンパイル

solver.pyをCython化することが決まりました。
とりあえず、Pythonフォルダの中身をごっそりコピーして、Cythonフォルダというのを作っておきましょう。（こんなことしなくてもOKなのですが）

Cythonのファイルの拡張子は　.pyx　とすることになっているので、まずsolver.pyxを作ります。

```
Cython$ cp solver.py solver.pyx
```

solver.pyxの先頭に、以下の2行を加えます。

```
#distutils: language=c++
```

```
#cython: language_level=3
```

languageでC++を指定します。CよりC++の方が、何かと便利かも知れないのでやっているだけで、これがない場合にはコンパイル結果 solver.c ができ、あれば solver.cpp ができます。

language_levelは3を指定します。CythonはPython２という古いPythonに準拠しているので、Python３で書かれたPythonプログラムは、コンパイル時にエラーが出ることがあります。まあ、いまさらPython２を使うことは少ないでしょうから、３を指定してください。

次に、コンパイルのために、setup.py を用意します。

```
Python$ cat setup.py
from distutils.core import setup
from Cython.Build import cythonize

setup(ext_modules=cythonize(['solver.pyx']))
```

Cythonizeの引数に、.pyxファイルのリストを与えれば、まとめてコンパイルします。
さらに、setup行を何行も書くことで、複数のプログラムをコンパイルすることもできます。
これらは、makefileと同様の動きをします。ソース、C/C++のソース、.soファイル（シェアード・ライブラリ/シェアード・モジュール）のタイムスタンプを見て、更新されている場合にだけコンパイルを行います。

コンパイルは、次のように行います。

```
Cython$ python3 setup.py build_ext --inplace
```

これを実行すると、まずC/C++が作られ、続いて.soが作られます。

```
Cython$ ls solver.*
solver.cpp  solver.cpython-38-x86_64-linux-gnu.so  solver.py  solver.pyx
```

これで、実行するための準備はできたので、Python版とCython版の速度比較をしてみましょう。

Cython化はsolver.pyに対してだけ行ったので、500問の問題を解く時間の比較をしてみます。

まず、Pythonで500問を解いてみます。

```
Python$ python3 NP.py -s data/Problem500.txt

      中略

Total 500   Success 500
Time    10.896430 sec
```

次に、Cythonでの実行です。実行コマンドは全く同じです。

```
Cython$ python3 NP.py -s data/Problem500.txt

      中略

Total 500   Success 500
Time    9.875354 sec
```

10.9秒が9.9秒になったので1割のスピードアップですが、誤差に毛が生えた程度です。

Cythonのプログラムは、基本的にPythonのアッパーコンパチですので、そのまま、あるいはわずかの変更を加えるだけでコンパイルし、実行することができまる。つまり、Python特有の性質もそのまま動くようにコンパイルされているので、ほとんど高速になりません。

もしsolver.pyxのCython化のための修正作業が終わったら、どのくらいの速度になるか見てみます。

```
Cython$ python3 NP.py -s data/Problem500.txt

      中略

Total 500   Success 500
Time    0.088417 sec
```

10.9秒が0.0884秒になっていて　10.9 / 0884 = 123 倍速になっています。

この高速化は、元のsolver.py に、どのような加筆修正がされて達成されたのか、これからじっくりと見ていきましょう。

第2部　Pythonはなぜ遅い

オブジェクトサイズ

オブジェクトのサイズと型を調べてみよう。

```
Python$ python3
Python 3.8.10 (default, Nov 26 2021, 20:14:08)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> a = 1
>>> a, sys.getsizeof(a), type(a)
(1, 28, <class 'int'>)
>>> a = 123.456
>>> a, sys.getsizeof(a), type(a)
(123.456, 24, <class 'float'>)
>>> a = 'abc'
>>> a, sys.getsizeof(a), type(a)
('abc', 52, <class 'str'>)
>>> a = [1,2,3,'abc']
>>> a, sys.getsizeof(a), type(a)
([1, 2, 3, 'abc'], 88, <class 'list'>)
>>> a = { 'a':1, 'b':2 }
>>> a, sys.getsizeof(a), type(a)
({'a': 1, 'b': 2}, 232, <class 'dict'>)
>>> a = { 'a','x','t','a' }
>>> a, sys.getsizeof(a), type(a)
({'x', 't', 'a'}, 216, <class 'set'>)
```

整数変数が28バイト、でも浮動小数点になると24バイトと減っている。
全体的に、非常にメモリサイズが大きくなっている。
つまり、Pythonは実行時にとてもメモリを食って重くなるということだ。

そもそも、Pythonを使っている時に、型を意識することはほどんど無いはず。NumPyを使うときには型を指定することがあるが、通常オブジェクトの型を指定することはない。
それでもちゃんと処理できているということは、各オブジェクトは、自分の型を知っている、つまり自分の型をオブジェクトの中に隠し持っている。

以下のように、数字が入っている変数に+1は可能だが、文字列が入っている変数に+1しようとすると、エラーとなる。

```
>>> a = 0
>>> a+1
1
>>> a = 'a'
>>> a+1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

その他にも、ガーベッジコレクションのための情報も保持しているはずである。
色々なものをオブジェクトが抱え込んでいるため、Cたど4バイト、8バイト程度で住むものが数十バイトになってしまっている。これではメモリを大量に食ってしまい、重いソフトになってしまう。

動的型付

Pythonでは、変数などの型は指定しません。変数同士の演算処理が行われる時、演算処理が可能な型に調整されて処理が行われます。

この調整を、コンパイル時に行うのが静的型付けで、２つの型によっては、非常に公立の良いオブジェクトに変換できます。しかし、Pythonのような動的型付けの言語で、なおかつインタープリタ言語の場合、コンパイル時に最適化することは困難で、実行の度に型を調べて、可能な型に調整してから演算するという面倒なことを行っています。つまり、Pythonでは、コンパイル時に型を利用した最適化はほぼできません。
動的型付は遅いので、Cythonに書き直す時、C言語のように静的型付を行うことで高速化します。

インタープリタ

Pythonはインタープリタ言語で、要するにソースコードを、実行が通過する毎にソースを毎回、たとえ何回目の実行であっても、ソースコードを解釈し直し、実行します。
インタープリタ言語は、コンパイル言語に比較して、通常数十倍から数百倍遅くなってしまいます。

第1部の最後で、solver.pyからsolver.pyxを作ってコンパイルして実行したけれど、1割程度しか速くならなかったけれど、これは動的型付が残っているし、オブジェクトのサイズもPythonのときのままでした。
これらをまとめて解消することで、100倍程度の高速化ができます。

第3部　書き換え練習

それでは、非常に短いプログラムで、徐々にCython化に慣れていくことにしよう。

円周率

目標：型指定の基本の習得

速度比較を行うため、収束の遅い級数で円周率を求めてみる。

ライプニッツの公式：

$$1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\cdots=\frac{\pi}{4}$$

この公式を4倍することで円周率を求める。

Python

```
def pi_1(n):
      sum = 0.0
      sign = 1

      for k in range(1,n,2):
            sum += sign*4/k
            sign = -sign
      return sum
```

Cython

```
def pi_1(long long n):
      cdef:
            double sum = 0.0
            int sign = 1
            long long k

      for k in range(1,n,2):
            sum += sign*4.0/k
            sign = -sign
      return sum
```

これを、PythonとCythonのコードはこんな感じになる。
赤字は、Cythonにしたときに追加・修正された箇所である。

関数内の変数の宣言はまとめて cdef: 以下に書いているが、以下でもOK.

```
      cdef  double sum = 0.0
      cdef  int sign = 1
      cdef long long k
```

ファイルは、pi_formula.py および cpi_formula.pyx である。
Cythonのファイルをpi_formula.pyxとすると、モジュール名がどちらもpi_formulaになり、コンパイルされて pi_formula.*.soが存在すれば、Cythonのモジュールがインポートされ、それが無ければ、Pythonのモジュールがインポートされる。
ここでは、速度比較をするため、別名にしてある。

```
def pi_2(n):
      sum = 0.0

      for k in range(1,n,4):
            sum += 4/k - 4/(k+2)
      return sum


def pi_3(n):
      n = (n // 4) * 4
      sum = 0.0

      for k in range(n,0,-4):
            sum += 4/(k-3) - 4/(k-1)
      return sum
```

```
def pi_2(long long n):
      cdef:
            double sum = 0.0
            long long k

      for k in range(1,n,4):
            sum += 4.0/k - 4.0/(k+2)
      return sum

def pi_3(long long n):
      n = (n / 4) * 4
      cdef:
      double sum = 0.0
      long long k

      for k in range(n,0,-4):
            sum += 4.0/(k-3) -
                        4.0/(k-1)
      return sum
```

def pi_3(n)のｎは４の倍数に調整するために、Pythonでは//による整数除算を行っているが、Cythonでは整数同士の除算は　/　で整数除算が行えるので変えた。
なお、// のままにしておいても、整数除算になるので、変更しなくても構わない。

では、実行してみよう。コンパイルは済んでいるものとする。
青がCythonに対するものである。
test関数の第１引数はpi_1,pi_2,pi_3を選択肢、第２引数はnである。

```
$ python3
>>> import pi_formula
>>> import cpi_formula
>>> pi_formula.test(1,100000000)
pi=3.1415926335902506    delta=-2.000e-08    time=3.484742 sec
>>> cpi_formula.test(1,100000000)
pi=3.1415926335902506    delta=-2.000e-08    time=0.091834 sec
>>> pi_formula.test(1,1000000000)
pi=3.141592651589258    delta=-2.001e-09    time=34.350203 sec
>>> cpi_formula.test(1,1000000000)
pi=3.141592651589258    delta=-2.001e-09    time=0.576653 sec


>>> pi_formula.test(2,200000000)
pi=3.1415926445762157    delta=-9.014e-09    time=4.548132 sec
>>> pi_formula.test(2,300000000)
pi=3.1415926445762157    delta=-9.014e-09    time=6.766601 sec


>>> pi_formula.test(3,200000000)
pi=3.1415926435897936    delta=-1.000e-08    time=5.426949 sec
>>> pi_formula.test(3,300000000)
pi=3.141592646923127    delta=-6.667e-09    time=8.081230 sec
>>> pi_formula.test(3,1000000000)
pi=3.141592651589794    delta=-2.000e-09    time=26.852395 sec
>>> cpi_formula.test(3,1000000000)
pi=3.141592651589794    delta=-2.000e-09    time=0.568189 sec
>>> cpi_formula.test(3,10000000000)
pi=3.1415926533897935    delta=-2.000e-10    time=5.613567 sec
>>> cpi_formula.test(3,100000000000)
pi=3.1415926535697936    delta=-2.000e-11    time=55.833034 sec
```

pi_3(n)関数で、n=1000000000 (10億)の場合の実行時間を比べる。

Python　26.852395秒

Cython　0.568189秒

26.852395/0.568189 = 47.26倍

浮動小数点計算が中心の演算では、だいたい50倍くらい高速化されるようである。

素数

目標：リストの代わりにC++のベクトルを利用

Python

```
## primes_list.py
import numpy as np
import sys, time

def prime(nb_primes):
	pbuf = []

	len_p = 0  # The current number of elements in p.
	n = 2
	while len(pbuf) < nb_primes:
		# Is n prime?
		for i in pbuf:
			if n % i == 0:
			break

		# If no break occurred in the loop, we have a prime.
		else:
			pbuf.append(n)
		n += 1

	return pbuf

if __name__ == '__main__':
	tm_start = time.time()
	n = int(sys.argv[1])
```

```
    ps = prime( n )
    tm_end = time.time()
    print( "primes={}\ntime={} sec".format(ps[-10:],tm_end-tm_start) )
```

関数prime(nb_primes)の引数には、求める素数の個数を与えます。
素数であることが分かった数を、pbufというリストに溜めていきます。
変数ｎが素数候補で、素数リストpbufのどの素数でも割り切れなかったら、素数リストpbufに加えます。
pbufの長さは、見つけた素数の個数です。

自然なの作りなので、これ以上の説明は省略します。

Cython

```
#  primes_vector.pyx
#cython: language_level=3
#distutils: language=c++

from libcpp.vector cimport vector

def primes(unsigned int nb_primes):
    cdef  int         n, i
    cdef  vector[int]  pbuf       # 素数のvector
    cdef unsigned int  sz = pbuf.size()
    pbuf.reserve(nb_primes) # allocate memory for 'nb_primes' elements.

    n = 2
    while sz < nb_primes:  # size() for vectors is similar to len()
        for i in pbuf:
            if n % i == 0:
            break
        else:
            pbuf.push_back(n)  # push_back is similar to append()
            sz = pbuf.size()
        n += 1
    return list(pbuf)
```

コンパイル

```
$ python3 setup.py build_ext --inplace
```

テスト

```
## primes_vector_test.py: test module for preimes_vector.pyx

import sys
import primes_vector
import time

if __name__ == '__main__':
      tm_start = time.time()
      ps = primes_vector.primes( int(sys.argv[1]) )
      tm_end = time.time()
      print( "primes={}\ntime={} sec".format(ps[-10:],tm_end-tm_start) )
```

実行

```
$ python3 primes_list.py 10
primes=[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
time=8.58306884765625e-06 sec
$ python3 primes_vector_ctest.py 10
primes=[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
time=1.5974044799804688e-05 sec
$ python3 primes_list.py 10000
primes=[104677, 104681, 104683, 104693, 104701, 104707, 104711, 104717, 104723,
104729]
time=2.1075708866119385 sec
$ python3 primes_vector_ctest.py 10000
primes=[104677, 104681, 104683, 104693, 104701, 104707, 104711, 104717, 104723,
104729]
time=0.16365647315979004 sec
```

10000個の素数の場合の実行時間を比べる。

Python　2.1075708866119385秒

Cython　0.16365647315979004秒

2.10757/0.163656 = 12.88倍

リストの代わりにC++のvectorを使ったので、100倍にはならず、12.88倍で終わった。固定サイズのリストなら配列にするのもあり、その場合はもっと高速になると思われる。

PythonとCythonの関係

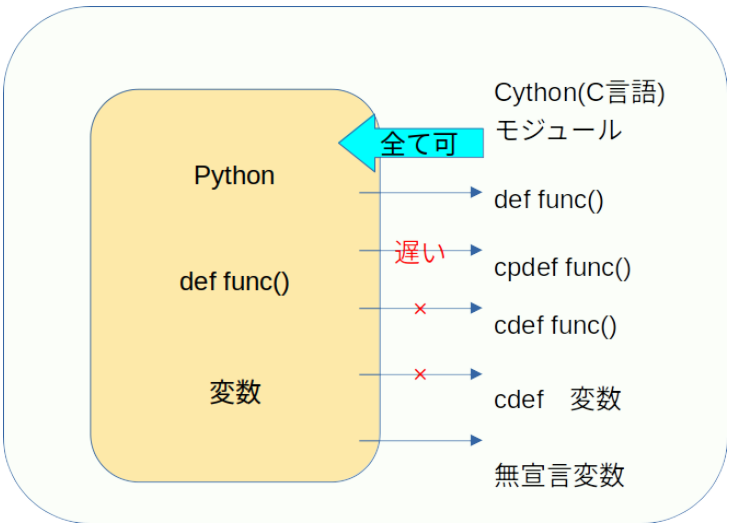
CythoはPythonの上位互換です。
つまり、Cythonモジュールの中では、Pythonでできることは全てできます。ただし、コンパイル不能のようなものは除きます。
CythonからPythonモジュールの中も自由に使えます。

逆に、Cythonモジュールの中でPythonの仕様以外のこと、例えばcdefによる関数の宣言、cdefによる変数宣言などはPythonモジュール側から参照できません。

cpdefという宣言が可能で、この宣言はPython版のdefとCython版のcdefの両方の機能を持っています。そのため、cpdefで宣言された関数は、Pythonモジュールから呼び出すことができます。
cpdefの関数は、Python、Cython両モジュールから呼び出せますが、実行速度に大きな差があります。Cythonから呼び出せば、cdefの場合とほぼ同じ速度で動きますが、Pythonモジュールから呼び出すと非常に遅くなります。
そのため、

Pythonモジュール → Cythonのdef関数 → Cythonのcpdef関数

というように、いちどCythonのdef関数でラッピングした方が、直接呼ぶよりも速くなります。



ｎ以下の正整数の和を求める関数　foo_何とか　をCythonモジュールで、色々な宣言で行った。

cy_foo.pyx

```
## foo関数  def/cdef/cpdef の比較

def foo_cdefcall(n):
      return foo_cdef(n)

def foo_cpdefcall(n):
      return foo_cpdef(n)

def foo_def(n):
      sum = 0
      for i in range(n):
            sum += i
      return sum

def foo_defcdef(n):
      cdef int nn = n, i
      cdef long long sum = 0
      for i in range(nn):
            sum += i
      return sum

cdef long long foo_cdef(int n):
      cdef int i
      cdef long long sum = 0
      for i in range(n):
            sum += i
      return sum

cpdef long long foo_cpdef(int n):
      cdef long i
      cdef long long sum = 0
      for i in range(n):
            sum += i
      return sum
```

cy_foo_test.py

```
## cy_foo_test.py

import sys, time
import cy_foo

def printinfo(fname,n,sum,tm):
      print("{}({})={}\ttime={} sec"\
            .format(fname,n,sum,tm))

n = 100000000                  #  1億

# Python call
tm_start = time.time()
sum = cy_foo.foo_def(n)
tm = time.time() - tm_start
printinfo("foo_def",n,sum,tm)

# foo_cdef()はpythonからは呼び出せない

tm_start = time.time()
sum = cy_foo.foo_defcdef(n)
tm = time.time() - tm_start
printinfo("foo_defcdef",n,sum,tm)

tm_start = time.time()
sum = cy_foo.foo_cdefcall(n)
tm = time.time() - tm_start
printinfo("foo_cdefcall",n,sum,tm)

tm_start = time.time()
sum = cy_foo.foo_cpdefcall(n)
tm = time.time() - tm_start
printinfo("foo_cpdefcall",n,sum,tm)

tm_start = time.time()
sum = cy_foo.foo_cpdef(n)
tm = time.time() - tm_start
printinfo("foo_cpdef",n,sum,tm)
```

5つの宣言の方法が違う関数をCythonモジュールのcy_foo.pyxの中に用意して、それらをテストするPythonのモジュールをcy_foo_test.pyに書いた。
実行すると、以下のようになった。

```
$ python3 cy_foo_test.py
foo_def(100000000)=4999999950000000    time=3.3251638412475586 sec
foo_defcdef(100000000)=4999999950000000    time=0.03223705291748047 sec
foo_cdefcall(100000000)=4999999950000000    time=0.028878211975097656 sec
foo_cpdefcall(100000000)=4999999950000000    time=0.028731346130371094 sec
foo_cpdef(100000000)=4999999950000000    time=0.05724501609802246 sec
```

1億回ループしているので、０から１億-１の和は32ビット整数では無理なので、64ビット整数(long long)を利用している。
Python側は、どんなり大きな整数でもオーバーフローしないので、Pythonではオーバーフローは気にしないで良い。

Pythonタイプの関数では、3.32516秒かかっていたが、cdef宣言された関数を呼び出している関数は0.0288782秒で終わっている。
3.32516/0.028878 = 115 倍の高速化になっている。

pdefの関数を、defの関数で間接的（ラッピング）に呼んだのがfoo_cpdefcallで、0.028731秒で、cdef宣言した関数と実行時間は変わらない。
しかし、Python側から直接呼んだ場合foo_cpdefは、0.0572450秒掛かっていて、cdef宣言した関数の約２倍の時間が掛かっている。
これから、cpdefの関数は、Pythonからでも、Cythonからでも自由に呼び出して使えるが、Python側から呼び出した場合、実行速度が半減してしまうようだ。

第3部　書き換え説明

お待たせしました。

ここからは、書き換えしたsolver.pyxの順に、書き換え部分について説明していきます。

```
 9 #distutils: language=c++
```

Cythonのコンパイル結果の出力をC++(.cpp)に指定。

```
 11 #cython: language_level=3
```

Python3を指定。

```
 12 #cython: profile=False
```

プロファイラを無効に。

Cython内の関数も対象にする場合には、Trueにしてコンパイルすること。

```
 13 #cython: boundscheck=False
 14 #cython: wraparound=False
```

リスト・配列の境界チェックと負数対応をオフ。

これらをTrueにすると、境界チェックをしなくなるので、配列などを使っていたら、2から3割くらい高速になる。

```
 21 DEF EXCEPTION_CODE = -99
```

以下は、例外処理のための例外コード(-99)をマクロ EXCEPTION_CODE として定義したものである。

C言語で言えば、#define EXCEPTION_CODE -99 に相当する。
Cythonでは、= が必要なので、注意。

```
23 cdef int SIZE = parameter.SIZE
24 cdef int SUBSIZE = parameter.SUBSIZE
```

Pythonのparameter.pyの中でSIZE と SUBSIZE を定義しているため、これらはPythonのオブジェクトであり、このままCythonモジュール(solver.py)の中で使うと、Python型とCython(C)型との間で型変換が行われてしまい、大変遅くなる。
それで、Cythonのint(C++のint)型に入れることで、以降の処理ではint型にアクセスすることになり、高速になる。

```
26 cdef int[:,:] board
27 cdef char[:,:,:] candidate
```

盤面boardをint型の２次元配列として定義します。まだサイズは指定していません。
候補candidateは３次元ブール値配列として定義します。まだサイズは未指定。
ブール値を扱うのですが、char型と定義し、True/Falseを入れまる。
この配列は、numpyの配列(numpy.ndarray)とは異なり、memoryviewという方法でメモリを確保して配列として使うことをこの2行は定義しています。詳しいことは高度になりますので、Cythonの Typed Memoryviews を参照のこと。

```
44 cdef initialize():
```

初期化関数は、定義にcdefを用いて、Cythonの関数とした。
関数の中身は変更なし。
この中で、numpyの配列をboard，candidate に代入することで、メモリが確保され、2次元、3次元配列として使えるようになる。

```
51 cdef int setProblem( bd ) except? EXCEPTION_CODE:
52     cdef int r, c
```

問題の設定関数であるが、整数intを返すCythonの関数として定義し、内部で例外が発生した場合には、EXCEPTION_CODEをエラーの戻り値とする。

関数内で使用する変数r,cを型宣言することで、速くなる。

```
57                     if setValue( r, c, bd[r][c] ) == EXCEPTION_CODE:
58                             return EXCEPTION_CODE
```

値をセットするときに、例外が発生することがあり、Pythonだと例外が伝わってくる

が、Cythonだと戻り値に指定した例外値が帰ってくるので、それを捕らえて、例外値を呼び出し側に戻す。
このあたり、PythonとCythonで違いが出てくる。
Cythonモジュールの中でも、関数宣言がdefで行われていると、Pythonとして扱われ、try-except で例外処理を行うことができる。

```
62 cdef bint checkLoop() except? EXCEPTION_CODE:
63     cdef bint changed, ret
64     cdef int r, c
```

延々とチェックする関数を定義している。
この関数の戻り値は、True/False/EXCEPTION_CODE の3種類があり、bint型にしてある。
関数内の変数の型宣言を行い、速くしている。

```
73                  if ret == EXCEPTION_CODE:
74                          return EXCEPTION_CODE
```

直前の関数checkBlock(c,r)の戻り値retが、DEFで定義した例外値の場合、例外値を返す。つまり、例外を呼び出し側に伝搬させる。
関数checkLoopの中で、同様の処理が4回ある。
この関数以外でも、同様の処理があるが、以下では説明を省略する。

```
113 cdef bint setValue( int r, int c, int v ) except? False:
114     cdef int i, n, r0, c0
```

引数の型、関数の型、関数内の変数の型を定義する。
また、例外処理の場合の戻り値を定義する。

```
128             r0 = (r//SUBSIZE)*SUBSIZE
129             c0 = (c//SUBSIZE)*SUBSIZE
```

ここの整数除算 // はそのままにしている。
Cythonの場合、整数/整数 は整数除算になるので、/ でも良いのだが、あえてそのままにしている。

```
140 def getAnswer():
141     ans = np.zeros((SIZE,SIZE)).astype(int)
142     NP.copyBoard( board, ans )
143     return ans
```

他のPythonのモジュールから呼び出されるし、速度を気にするような関数ではないのでdefのままにしてある。

ansは、numpyの全要素が0の配列を作るzerosで作られたnumpyの配列(numpy.ndarray)である。
Boardは、solverモジュールの最初で型宣言されている配列である。
NP.copyBoard(board, ans) にて、boardの内容をansにコピーしている訳だが、型のことは何も考えずに与えているが、これできちんとコピーされる。
NPモジュール内のNP.copyBoard関数は、以下のように定義されている。

```
92  def copyBoard( fr, to ):
93      for r in range(SIZE):
94              for c in range(SIZE):
95                      to[r][c] = fr[r][c]
```

この関数は、numpy.ndarray型の配列がやってくるものと想定して書かれているが、Cythonでの配列とndarryの間をうまく処理してくれるので、何も修正していない。

```
145     def getValue( int r, int c ):
```

引数の型だけ指定している。
他のPythonモジュールから呼ばれるので、defのままにしてある。
戻り値は、Pythonオブジェクトとしての整数になる。

```
148     def getCandidate( int r, int c ):
149             return candidate[r][c]
```

引数の型だけ指定している。
他のPythonモジュール(generator.py)から呼ばれるので、defのままにしてある。
Candidateは3次元の配列なので、candidate[r][c]は1次元の配列になるが、関数の型を指定していない。
この関数を呼び出す側は次のようになっている。

```
generator.py  137             cans = solver.getCandidate(r,c)
```

Python版では、numpyのnumpy.ndarray配列として扱われていたのだが、Cythonの getCandidate(r,c)からの戻り値の型はndarrayではあるものの、MemoryViewとして確保されたなかのndarrayである。

```
<MemoryView of 'ndarray' object>
```

ということで、PythonでもCythonでも、ndarrayとして処理されるので問題ないようだ。

```
153     def printCandidate():
```

候補配列をの表示であり、速度は気にしなくてもよいので、変更なしでもよいのだが、つい変数だけ型宣言してしまった。

```
240  cdef bint checkBlock( int c0, int r0 ) except? EXCEPTION_CODE:
241      cdef:
242           int    n, cnt, col, row, r, c, can
243           bint   changed, exist
```

ブロックの検査だが、関数の型、使用変数の宣言、例外について変更があるが、既に説明したことと同じなので、説明は省略する。

また、これ以降に出てくる関数も、Cythonとして既に説明したこと以上のことは何もない単純な変更だけなので、説明は省略する。

以上

著者紹介
コンピュータ歴：TK-80など8ビットコンピュータ向けコンパイラ、画像処理、３Ｄソリッドモデラー、ポケットコンピュータ、日本語情報処理、X Window、Unix、Linux、科学技術計算処理、パズルプログラミングなどの開発を経て、最近は人工知能・進化計算を中心にプログラミング活動。
著作：『Cプログラミング診断室』、『Cプログラミング専門課程』、『(コ)の業界のオキテ』、『実践遺伝的アルゴリズム』など。
直近の寄稿：数学セミナー2022年2月号　特集◎パズルと数学「数独（ナンバープレイス）の自動生成とヒント数」
合体ナンプレの問題作成で280合体ナンプレのギネス世界記録(TM)を保持。

進化計算　天啓｜TENKEI

株式会社タイムインターメディア
知識工学センター
東京都調布市小島町1-1-1
電気通信大学 UECアライアンスセンター217

puzzle@timedia.co.jp
藤原博文

2021年3月24日