NumPy が物足りない人への Cython 入門（事前公開版）

杜 世橋 FreeBit
@lucidfrontier45

はじめに

サンプルコードの在処

https://github.com/tokyo-scipy/archive
にて第3回目にあるcython_introを参照してください。

NumPy

numpy.ndarray class

数値計算用のN次元配列

* Homogeneous N-dimensional Array
  各要素の型がすべて同じ → CやFortranの配列

* 強力なインデクシング表現
  `A[0:10, 3]` etc

* Universal functionによる直感的な数式の表現
  `y[:] = 3.0 * np.sin(x[:]) + x[:]**2 + e[0,:]` etc

NumPy

* ベクトル化された配列処理
* BLAS/LAPACK を使った行列計算
* 各種乱数の発生
* FFT etc.

NumPy があれば何でもできそう？

NumPy

Implementation	CPU time
Pure Fortran	1.0
Weave with C arrays	1.2
Instant	1.2
F2PY	1.2
Cython	1.6
Weve with Blits++ arrays	1.8
Vectorized NumPy arrays	13.2
Python with lists and Psyco	170
Python with lists	520
Python with NumPy and `u.item(i,j)`	760
Python with NumPy and `u[i,j]`	1520

実はforループを使うと激しく遅い

250×250 偏微分方程式 wilbers et al. 2009 Table 2 より

Cython

* Pythonic なコードを C に変換し、コンパイルして速く実行する。

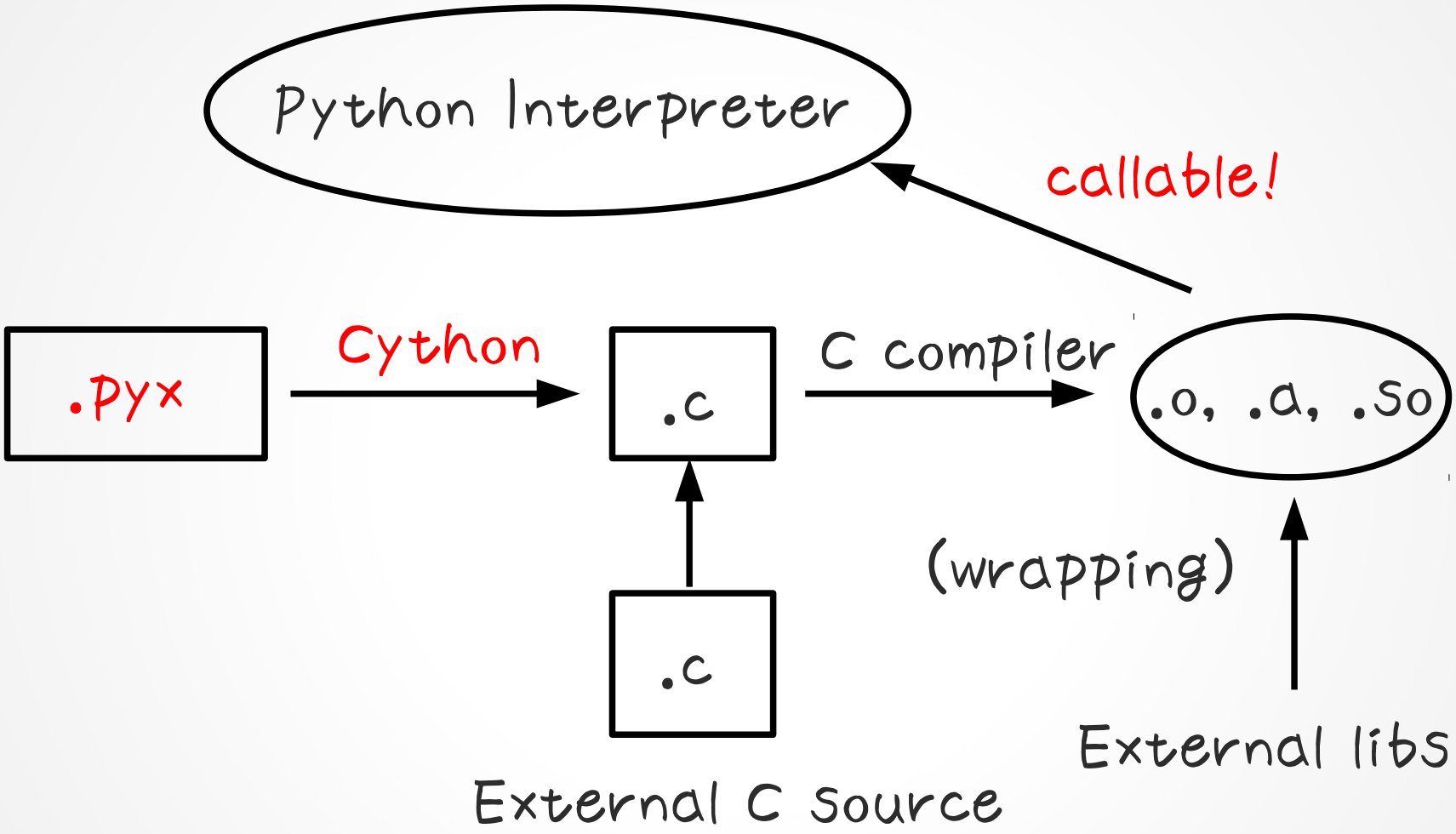
* 既存の C のライブラリへの wrapper としても便利。

* インストールは *nix 系 OS ならばお手軽インストール。

```
$python setup.py build
$(sudo) python setup.py install
```

* Ubuntu や Fedora のレポジトリもある。

Cython の動作イメージ



Cythonのコード

```
def ddotp(a, b):
    len_a = len(a)
    dot_value = 0
    for i in xrange(len_a):
        dot_value += a[i] * b[i]
    return dot_value
```

ちなみにコードをスライドに貼るときにはEclipseからのコピペが便利
ソースはstackoverflow

Cython 使い方


```
$cython ddot.pyx
```

→ ddot.c ができている

```
$gcc -c -O2 -fPIC -I$PYTHON_H_DIR \
                  -I$NUMPY_H_DIR ddot.c
```

→ ddot.o ができている

$PYTHON_H_DIR は Python.h のある場所

$NUMPY_H_DIR は numpy/arrayobject.h とかの (ry

```
$gcc -shared -o ddot.so ddot.o
```

→ ddot.So ができている

後は python で **import ddot** とすれば使える！

setup.py を書く

```
from numpy.distutils.core import setup
from numpy.distutils.misc_util import Configuration

config = Configuration()
config.add_extension("ddot", sources=["ddot.c"])
setup(**config.todict())
```

以後は以下のコマンドで小人さんたちが
.c から .so を作れってくれます

$python setup.py build_ext -i

詳しくはここで公開している setup.py を参照

Cython による高速化

おまじない編

```
1 #cython: boundscheck=False

2 import numpy as np
3 cimport numpy as np
4 cimport cython

  DOUBLE = np.float64
5 ctypedef np.float64_t DOUBLE_t
```

1. グローバルコンパイルディレクティブを指定
2. numpy の __init__.py をインポート
3. Cython に付属している numpy.pxd をインポート
4. Cython の built-in モジュールをインポート
5. np.float64_t 型に別名 DOUBLE_t をつける。C の typedef に相当

Cython による高速化

実践編

```
1 @cython.wraparound(False)
2 def ddot2(np.ndarray[DOUBLE_t, ndim=1] a,
          np.ndarray[DOUBLE_t, ndim=1] b):
3     cdef int i, len_a
      cdef double dot_value = 0.0
      len_a = len(a)
4     for i in xrange(len_a):
          dot_value += a[i] * b[i]
      return dot_value
```

1. ローカルコンパイルディレクティブを指定
2. 関数の引数に型を指定
3. **cdef** で変数の型を指定
4. for ループは自動で C の for ループに変換される

Cythonによる高速化

応用編

```
cdef double ddot_intrn(double *a, double *b,
                       int len_a):
    cdef int i
    cdef double dot_value = 0.0
    for i in xrange(len_a):
        dot_value += a[i] * b[i]
    return dot_value
```

* cdefを使って関数を定義するとPythonからアクセスできなくなるが、よりオーバーヘッドを減らせる。
* 現状ではcdefとnumpyを同時に使えない...
* 引数はdoubleのポインタにしておく。

Cython による高速化

応用編

```
def ddot3(np.ndarray[DOUBLE_t, ndim=1] a,
          np.ndarray[DOUBLE_t, ndim=1] b):
    cdef int i, len_a
    cdef double dot_value = 0.0
    len_a = len(a)
    dot_value = ddot_intrn(<double *>a.data,
                           <double *>b.data, len_a)
    return dot_value
```

* def を使って cdef 関数の wrapper を作っておく
* numpy の data は (char *) 型なので (double *) にキャストして渡す。

Cythonによる高速化

応用編

```
double ddot_c(double *a, double *b, int len){
    int i;
    double dot_value = 0.0;
    for(i=0; i<len; i++){
        dot_value += a[i] * b[i];
    }

    return dot_value;
}
```

Cの関数をCythonを使って呼びだそう！

Cythonによる高速化

連携編

```
double ddot_c(double *a, double *b, int len);
```

まずは通常のCのヘッダーファイルを用意する。

Cythonによる高速化

連携編

```
cdef extern from "ddot_c.h":
    double ddot_c(double *a, double *b, int len_a)

def ddot4(np.ndarray[DOUBLE_t, ndim=1] a,
          np.ndarray[DOUBLE_t, ndim=1] b):
    cdef int i, len_a
    cdef double dot_value = 0.0
    len_a = len(a)
    dot_value = ddot_c(<double *>a.data,
                       <double *>b.data, len_a)
    return dot_value
```

cdef extern~ を用いてCのヘッダーファイルからプロトタイプ宣言

Cythonによる高速化

実験編

```
import numpy as np
import ddot_cython as cyddot

N = 100000
x = np.random.randn(N)
y = np.random.randn(N)
z_npdot = np.dot(x, y)

test_funcs = {"cyddot.ddotp":cyddot.ddotp,
              "cyddot.ddot1":cyddot.ddot1,
              "cyddot.ddot2":cyddot.ddot2,
              "cyddot.ddot3":cyddot.ddot3,
              "cyddot.ddot4":cyddot.ddot4}

for (func_name, dot_func) in sorted(test_funcs.items()):
    z = dot_func(x, y)
    print func_name, np.allclose(z_npdot,  z)
```

np.dotと各実装の値を比較

Cythonによる高速化

実験編


```
import timeit
setup_string = """
import numpy as np
import ddot_cython as cyddot
N = 100000
x = np.random.randn(N)
y = np.random.randn(N)
"""
test_strings = ["np.dot(x, y)", "cyddot.ddotp(x, y)",
                "cyddot.ddot1(x, y)", "cyddot.ddot2(x, y)",
                "cyddot.ddot3(x, y)", "cyddot.ddot4(x, y)"]
n_retry_default = 10000
for test_string in sorted(test_strings):
    n_retry = n_retry_default
    if "ddotp" in test_string:
        n_retry = n_retry_default / 1000
    test_time = timeit.Timer(test_string, setup_string).timeit(n_retry)
    print "%20s used %12.5e s"%(test_string, test_time / n_retry )
```

各実装の実行時間を比較

多次元配列

```
def matmult2(np.ndarray[DOUBLE_t, ndim=2] a,
             np.ndarray[DOUBLE_t, ndim=2] b):
    cdef int i, j, k, n, m, l
    cdef np.ndarray[DOUBLE_t, ndim=2] c
    n = len(a)
    l = len(a[0])
    m = len(b[0])
    c = np.zeros((n, m))
    for i in xrange(n):
        for j in xrange(m):
            c[i, j] = 0.0
            for k in xrange(l):
                c[i, j] += a[i, k] * b[k, j]
    return c
```

* ndim = n と書くと n 次元配列になる
* 関数内の ndarray も cdef をするのを忘れない!

多次元配列

```
cdef void matmult_intrn(int n, int m, int l,
        double *a, double *b, double *c):

    cdef int i, j, k
    for i in xrange(n):
        for j in xrange(m):
            c[i*m + j] = 0.0
            for k in xrange(l):
                c[i*m + j] += a[i*l + k] * b[k*m + j]
```

* cdef を使う場合は void を使い、引数は参照渡し。
* 配列は 1 次元で定義して手動でインデキシングする。
* 配列が Row-Major または Column-Major なのかに注意！

その他の話題

1. C++との連携
2. マルチスレッド
3. クラスのcdef
4. 文字列
5. pxdファイルによる宣言の共有化
6. scipy.sparseとの連携

詳しくは〈http://docs.cython.org/〉を参照！
他にはscikit-learnのsvmがかなり参考になる。

終わり

ご清聴ありがとうございました。