18, March, 2012

NumPy が物足りない人への Cython 入門

杜 世橋 FreeBit
@lucidfrontier45

自己紹介

* 学生時代の専門は生物物理と機械学習
* 現在はネットワーク関連のエンジニア（IPv6 関連）
* ただの Python 信者
* 仕事はもっぱら C ばかりで Python 欠乏症に悩む

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？

現実は…

NumPy

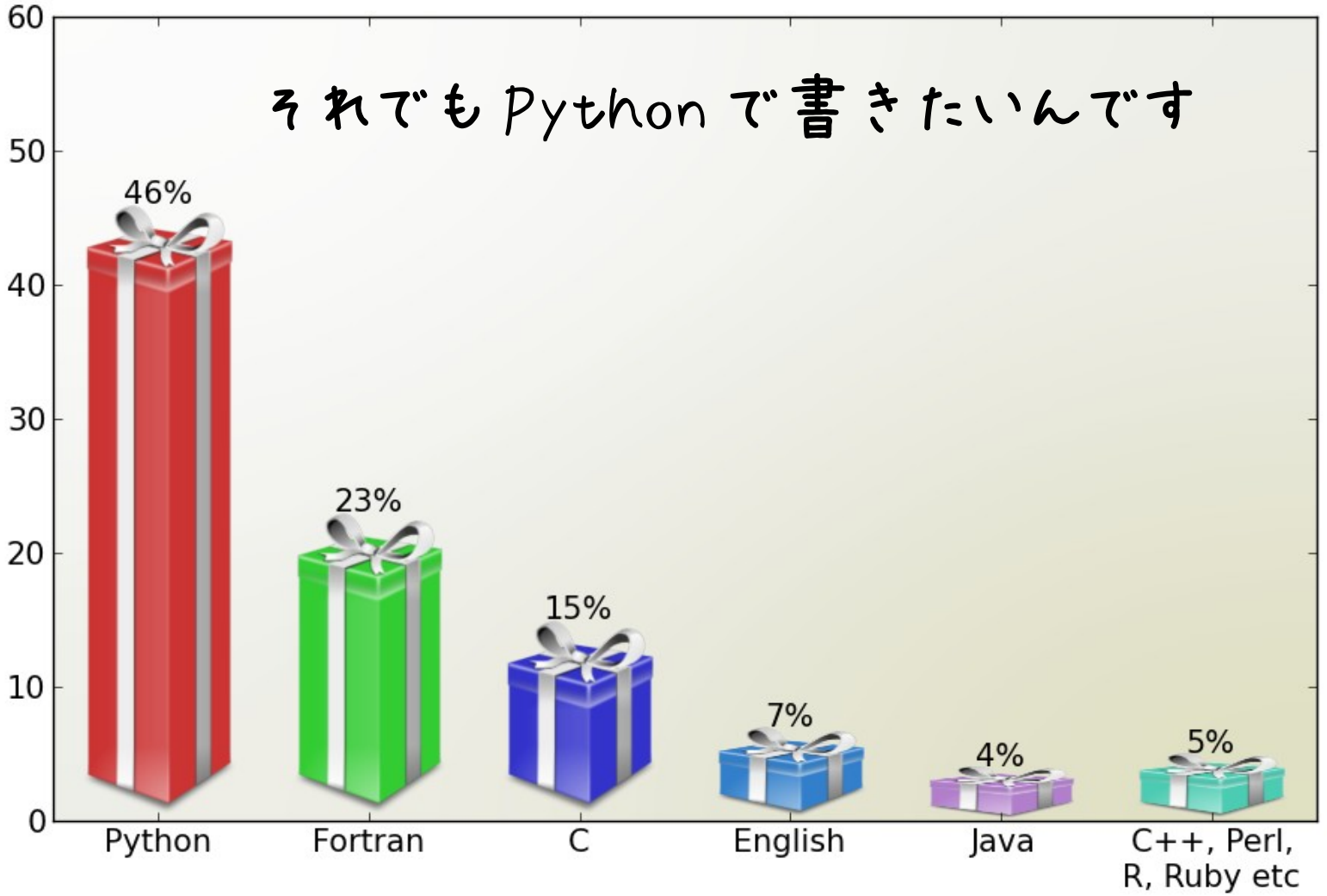
実はforループを使うと激しく遅い

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

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

SciPy の公式サイトにもパフォーマンス比較の記事があったりする。

Popular Languages



(matplotlibを用いて作図!)

Author's wild fancy, 2012

そんな訳で


Cython

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。

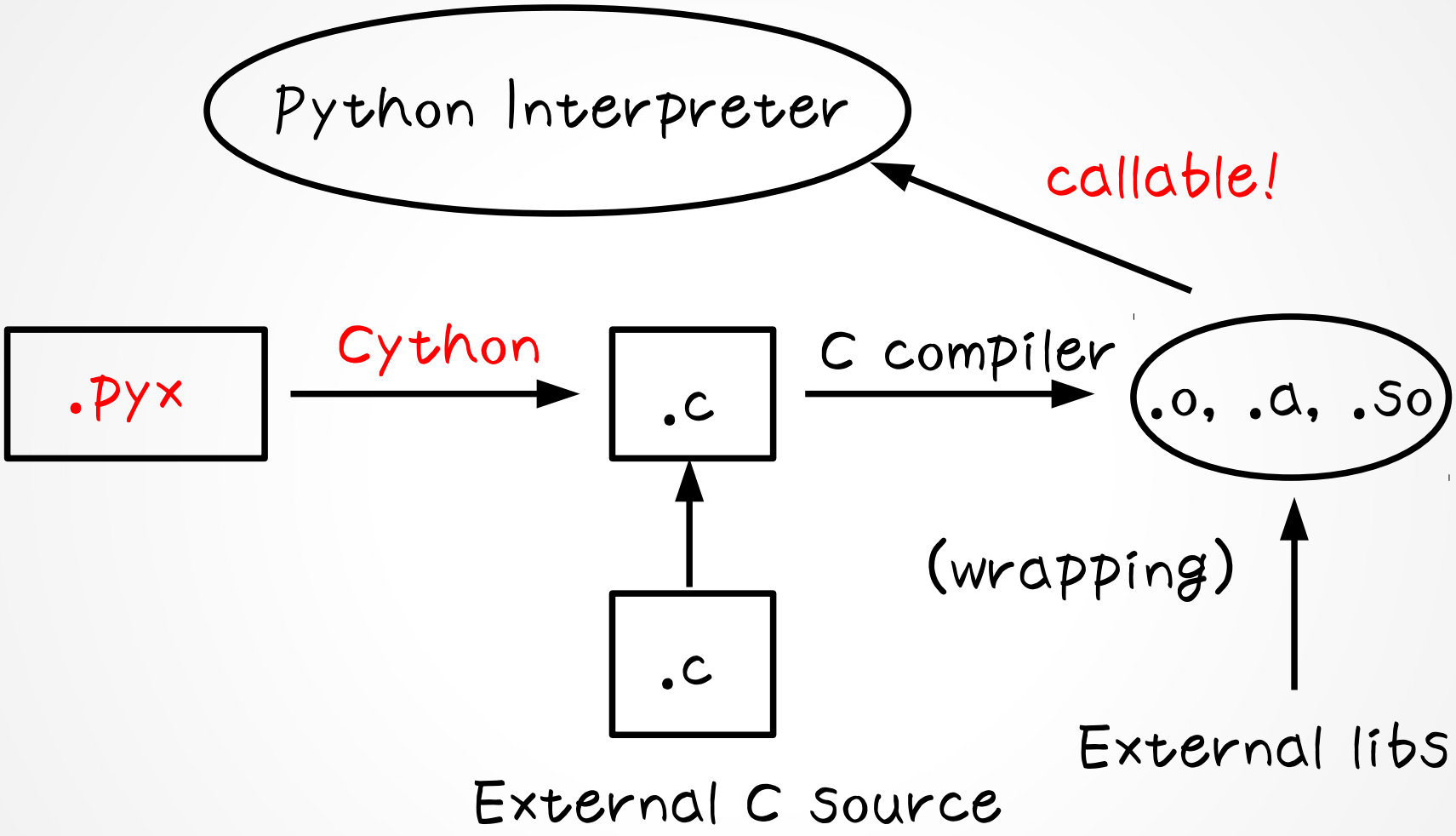
* 既存の C のライブラリへの wrapper としても便利。

* 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

Python のまんまじゃん！

Cython 使い方


```
$cython ddot_cython.pyx
```
→ ddot_cython.c ができている

```
$gcc -c -O2 -fPIC -I$PYTHON_H_DIR \
                  -I$NUMPY_H_DIR ddot_cython.c
```
→ ddot_cython.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_cython.so ddot_cython.o
```
→ ddot_cython.so ができている

後は python で **import ddot_cython** とすれば使える！

めんどい...

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
* ndarray.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 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

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

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！

多次元配列

転置の罠

“At = A.t” と書いた所で At.data は A.data のまま。
Cython の関数に At.data を渡しても転置されていない。

→ 自分で転置をしっかり書くか、あるいはnp.asanyarray
を使ってメモリと引き換えにコピーを作るか。

(*) ちなみに転置が必要になるのは行列積の場合だけであるが、その場合は
BLAS のインターフェイスに転置をするかどうかを指定できるので大丈夫。

構造体を使おう

```
typedef struct {
    int a, b;
    double x, y;
} myStruct;

void initStruct(myStruct *mst);
void showStruct(myStruct mst);
```

struct_test.h。 実装はstruct_test.cを参照のこと。

myStruncやその関数をCythonから使いたい！

構造体を使おう

```
   cdef extern from "struct_test.h":
1      ctypedef struct myStruct:
           int a

2      cdef void initStruct(myStruct *mst)
       cdef void showStruct(myStruct mst)

   def test_struct():
3      cdef myStruct mst
4      initStruct(&mst)
5      showStruct(mst)
6      return mst
```

1. ヘッダーファイルから構造体を定義
2. ヘッダーから関数プロトタイプ宣言
3. cdefを用いて構造体の変数を作成
4. 参照渡し
5. 値渡し
6. returnはdictを返す

So what !?

Cythonを使おう！

基本的にNumPy/SciPyの関数は相当速い。
陽にループが必要なアルゴリズムがターゲット。

例えば動的計画法とか...

Cythonを使おう！

HMMにおけるForwardアルゴリズム

1) Initialization:

$$\alpha_1(i) = \pi_i b_i(O_1), \qquad 1 \le i \le N. \tag{19}$$

2) Induction:

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij}\right] b_j(O_{t+1}), \qquad 1 \le t \le T-1$$

$$1 \le j \le N. \tag{20}$$

3) Termination:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i). \tag{21}$$

詳しくはRawrence 1989を参照。

Cythonを使おう！

```
for i in xrange(N):
    lnalpha[0,i] = lnpi[i] + lnf[0,i]

for t in xrange(1, T):
    for j in xrange(N):
        for i in xrange(N):
            temp[i] = lnalpha[t-1,i] + lnA[i,j]
        lnalpha[t,j] = _logsum(N,temp) + lnf[t,j]
```

Pythonじゃ3重ループなんて恐ろしくて使えやしないが
Cythonならなんて事ない！

@shoyuさんがNumPyで実装していますが、やっぱりtに対するループは回しているみたいです。

宣伝 ①

scikit-learn の hmm に PyVB の
cython モジュールを移植しました。

scikit-learn
https://github.com/scikit-learn/scikit-learn

PyVB
https://github.com/lucidfrontier45

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。
疎行列は scikit-learn の graph_shortest_path がよさげ。

その他の話題

最近はCythonを利用したFwrapなるものもできている。どうやら
Python(Cython(C(Fortran))
のようなイメージで動くらしい。

タダで読めるCython情報

- Cython Tutorial, Dag S. Seljebotn 2010
EuroScipy 2010で発表されたもの。Cython公式ドキュメントをNumPyユーザー向けにまとめた感じ。

- Fast Numerical Computations with Cython,
Dag S. Seljebotn 2009
少し型ばった論文調のもの。Memory layout, SSE and vectorizing C compilers, Parallel computationといった発展的な内容にも言及

URLは載せるのが面倒なので勝手にタイトルでググってください。

タダで読めるCython情報

- Using Cython to Speed up Numerical Python Programs, Wilbers et al. 2009

Cythonと F2PY などの比較。実際に偏微分方程式をそれぞれの方法で実装しており、それを見るだけでも価値はある。

- Cython: The Best of Both worlds, Behnel et al. 2011

Sparse Matrix など、Cythonの usecases が述べられている稀有なドキュメント。なんとFwrapについても言及されている！そして最後の References もいい感じ。一度は読むべき。

第二幕

第二幕

Fortranerのための F2PY 入門!

ふるーい Fortran77

```
      SUBROUTINE DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
      DOUBLE PRECISION ALPHA,BETA
      INTEGER K,LDA,LDB,LDC,M,N
      CHARACTER TRANSA,TRANSB
      DOUBLE PRECISION A(LDA,*),B(LDB,*),C(LDC,*)
      NOTA = LSAME(TRANSA,'N')
      NOTB = LSAME(TRANSB,'N')
      IF (NOTA) THEN
          NROWA = M
          NCOLA = K
      ELSE
          NROWA = K
          NCOLA = M
      END IF
      IF (ALPHA.EQ.ZERO) THEN
          IF (BETA.EQ.ZERO) THEN
              DO 20 J = 1,N
                  DO 10 I = 1,M
                      C(I,J) = ZERO
   10             CONTINUE
   20         CONTINUE
          ELSE
```

新しい Fortran 90/95

```
module mcmin
    use m_sizes
    use m_atoms
    implicit none
    private
    real(8) :: sconst = 12.0d0
    public mcminimize,sconst,setGrdMin,checkClash
    contains
    function mcmin1(xx,g) result(e)
        real(8), intent(in) :: xx(maxvar)
        real(8), intent(out) :: g(maxvar)
        real(8) :: e
        integer :: i,nvar
        real(8) :: derivs(2,maxatm)
        nvar = 0
        do i = 1, n
            nvar = nvar + 1
            x(i) = xx(nvar) * inv_sconst
            nvar = nvar + 1
            y(i) = xx(nvar) * inv_sconst
        end do
        call gradient2(e,derivs)
```

お分かりいただけただろうか？

why Fortran 90/95

* Legacy の活用。

* module のおかげで機能ごとにパッケージを分けやすくなった。

* C よりも（多次元）配列の扱いが楽。
  - allocate(x(N, M)) のように一発で多次元配列を生成！
  - x(1:10, :, 2) のようにインデキシングできる！
  - c[1:4] = a[3:6] + b[2:5] のように書ける！

* Compiler が基本的には C よりも速いコードを生成する。

why Fortran 90/95

← おすすめ！

F2PY

```
subroutine ddot_f(a, b, len_a, dot_value)
    implicit none

    real(8), intent(in) :: a(len_a)
    real(8), intent(in) :: b(len_a)
    integer, intent(in) :: len_a
    real(8), intent(out) :: dot_value

    integer :: i
    dot_value = 0.0d0
    do i = 1, len_a
        dot_value = dot_value + a(i) * b(i)
    enddo
end subroutine
```

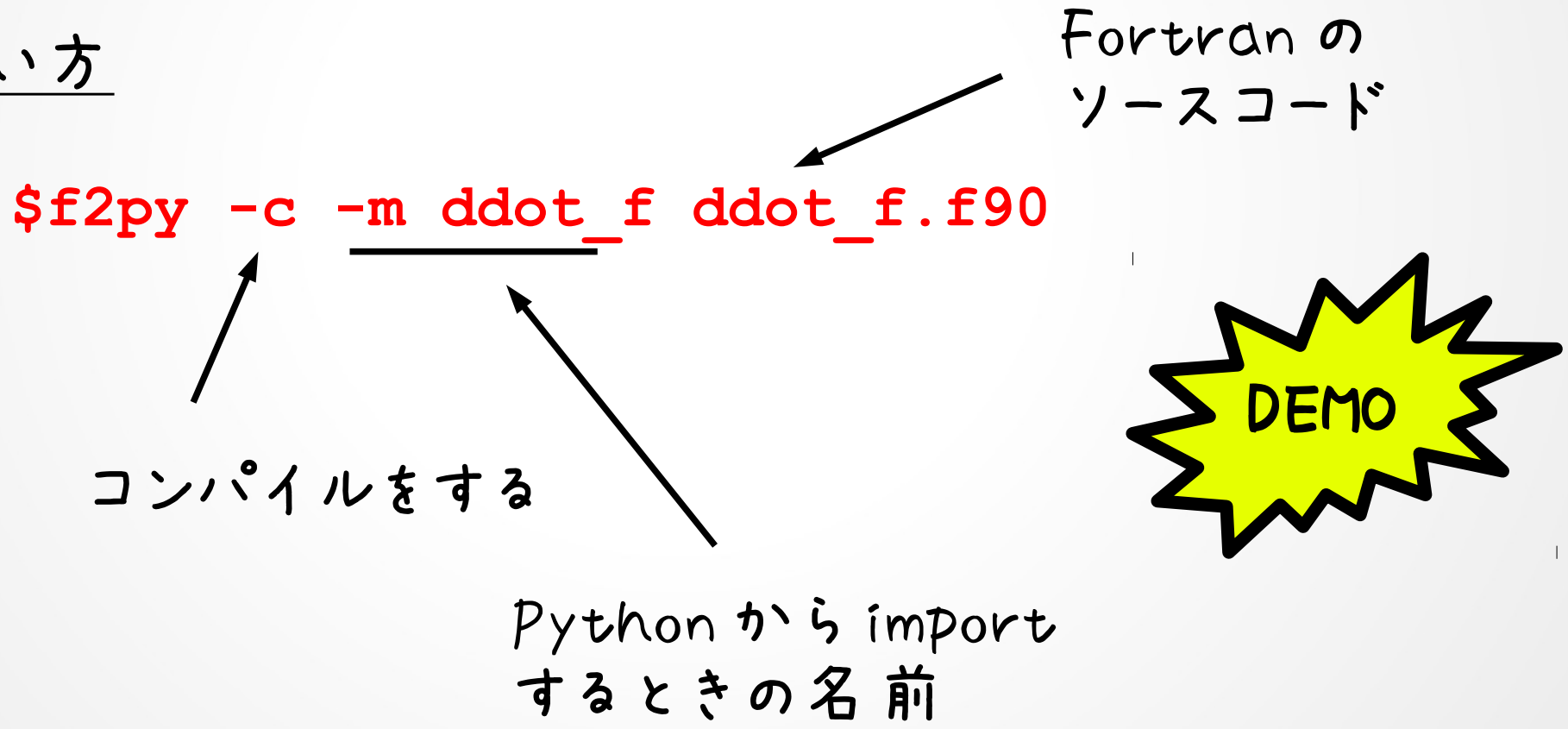
こんな感じの Fortran コードを Python から使おう！

F2PY

INSTALL

NumPy をインストールすれば自動的に入っている！

使い方



F2PY

連携編

.pyf ファイルを書いて外部ライブラリの関数をコール。
Fortran90/95 の interface module ライクな文法
C のヘッダーファイルに相当。

```
python module ddot_f2py
    interface
        subroutine ddot_f(a, b, len_a, dot_value)
            Real(8), intent(in) :: a(len_a)
            Real(8), intent(in) :: b(len_a)
            integer, optional, intent(in) :: len_a=len(a)
            real(8) intent(out) :: dot_value
        end subroutine ddot_f
    end interface
end python module ddot_f2py
```

$f2py -c -m fext fext.pyf -lflib

F2PY

setup.py を書けば build も楽チン

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

config = Configuration()

ddot_sources = ["ddot_f.f90"]
config.add_library("ddot", sources=ddot_sources)
config.add_extension("ddot_f2py",
         sources=["ddot_f2py.pyf"],
         libraries=["ddot"], depends=ddot_sources)

setup(**config.todict())
```

基本的にはCythonの場合の".c"を".f90"に変えるだけ。
F2PYはNumPyの一部なので自動化されている！

F2PY

もっと知りたい人ひとは
scipy.linalg のソースを読むべし。

BLAS/LAPACK の関数を pyf ファイルと
F2PY を使って wrap している。

おまけ

Cython で Fortran を使う！

C と Fortran の連携の知識があれば OK。詳細はググれ。

1. C は Row-Major で Fortran は Column-Major
→`np.asanyarray` で `order` を指定して変換

2. Fortran の関数名は C から見ると最後にアンダースコア”_”がつく。

3. Fortran 関数の引数はすべて参照渡し。

4. Fortran の `subroutine` は C の `void` 型関数に相当

おまけ

```
  cdef extern from "matmult_c.h":
1     void matmult_f_(int *n, int *m, int *l,
                      double *a, double *b, double *c)

  def matmult5(np.ndarray[DOUBLE_t, ndim=2] a,
               np.ndarray[DOUBLE_t, ndim=2] b):
      cdef int n, m, l
      cdef np.ndarray[DOUBLE_t, ndim=2] c
      n = len(a)
      l = len(a[0])
      m = len(b[0])
2     a = np.asfortranarray(a)
      b = np.asanyarray(b, order="F")
      c = np.empty((n, m), order="F")
3     matmult_f_(&n, &m, &l, <double*>a.data,
                 <double*>b.data,  <double*>c.data)
      return c
```

1. 事前にヘッダーに末尾に"_"を付けたプロトタイプ宣言を書いておき、cdef externで(ry
2. Column-Majorの配列に変換or作成。
3. 引数は"&"をつけて参照渡し！

おまけ

よくある質問

＊どうも引数の渡し方や、関数名の”_”が気持ち悪い。
＊美学に反する！

おまけ

対処法

Fortran に対する信仰を養う。

おまけ

Fortran を C で wrap してそれを
Cython から呼べば多少はましか

おまけ

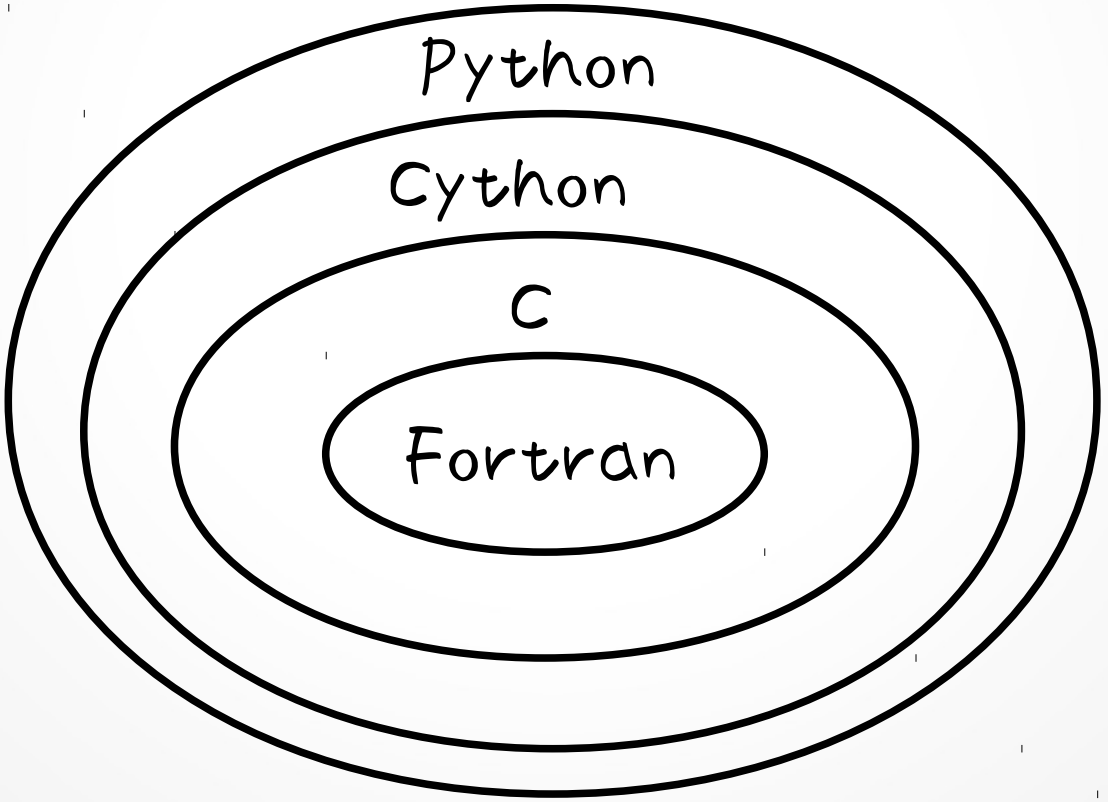
```
void matmult_f(int n, int m, int l,
               double *A, double *B, double *C){
    matmult_f_(&n, &m, &l, A, B, C);
}
```

```
def matmult6(np.ndarray[DOUBLE_t, ndim=2] a,
             np.ndarray[DOUBLE_t, ndim=2] b):

    cdef int n, m, l
    cdef np.ndarray[DOUBLE_t, ndim=2] c
    n = len(a)
    l = len(a[0])
    m = len(b[0])
    a = np.asfortranarray(a)
    b = np.asanyarray(b, order="F")
    c = np.empty((n, m), order="F")
    matmult_f(n, m, l, <double*>a.data,
              <double*>b.data, <double*>c.data)
    return c
```

おまけ

つまり、
Python → Cython → C → Fortran
の順番でcallしていけばいい



おまけ

おまけ

Python(Cython(C(Fortran)))、、、だと？

結局何使えばいいの？

C → Cython 一択。異論は認めない。

Fortran → NumPy との連携は F2PY のほうが簡単。
ただし Cython のほうがいろんなことできる。

Cython は活発に開発がなされており、今後のことを考えるとやっておいて損はない。(はず...

おまけのおまけ

F2PYでCの関数を使う！(← 誰得？

基本的にはpyfファイルにCの関数のインターフェイスを書くだけ

おまけのおまけ

```
python module matmult_f2py
    interface
        subroutine matmult_c(n,m,l,a,b,c)
            intent(c) matmult_c
            intent(c)
            integer, intent(in) :: n, m, l
            real(8), intent(in) :: a(n, l), b(l, m)
            real(8), intent(out) :: c(n, m)
        end subroutine matmult_c
    end interface
end python module matmult_f2py
```

F2PY の拡張仕様である **intent(c)** を書いておけばC の関数、変数だと解釈される。
おそらく内部で配列の order を変換したコピーが生成される。

おまけのおまけ

```
python module matmult_f2py
    interface
        subroutine matmult_c(n,m,l,a,b,c)
            intent(c) matmult_c
            intent(c)
            integer, intent(in) :: n, m, l
            real(8), intent(in) :: a(n, l), b(l, m)
            real(8), intent(out) :: c(n, m)
        end subroutine matmult_c
    end interface
end python module matmult_f2py
```

F2PY の拡張仕様である **intent(c)** を書いておけばC の関数、変数だと解釈される。
おそらく内部で配列の order を変換したコピーが生成される。

ベンチマーク

今回作成した全関数の性能評価と考察

ベンチマーク

自宅の最速マシーン上で実行

スペックと環境

* Intel Core2 Quad Q6700@2.66GHz
* 2GB RAM
* Linux Mint 12 32bit
* gcc 4.6.1 (Ubuntu/Linaro 4.6.1-9ubuntu3)

1000×1000 の行列積を実行！

ベンチマーク

結果

GotoBLAS2

```
          np.dot(x, y) used   9.40690e-02 s
  cymm.matmultp(x, y) used    とにかく遅い
  cymm.matmult1(x, y) used   1.82089e+01 s
  cymm.matmult2(x, y) used   9.69816e+00 s
  cymm.matmult3(x, y) used   6.23167e+00 s
  cymm.matmult4(x, y) used   6.19344e+00 s
  cymm.matmult5(x, y) used   4.77592e+00 s
  cymm.matmult6(x, y) used   4.77813e+00 s
fpmm.matmult_f(x, y) used   4.77768e+00 s
fpmm.matmult_c(x, y) used   6.19341e+00 s
```

np.dot + GotoBLAS2 は神

宣伝 ②

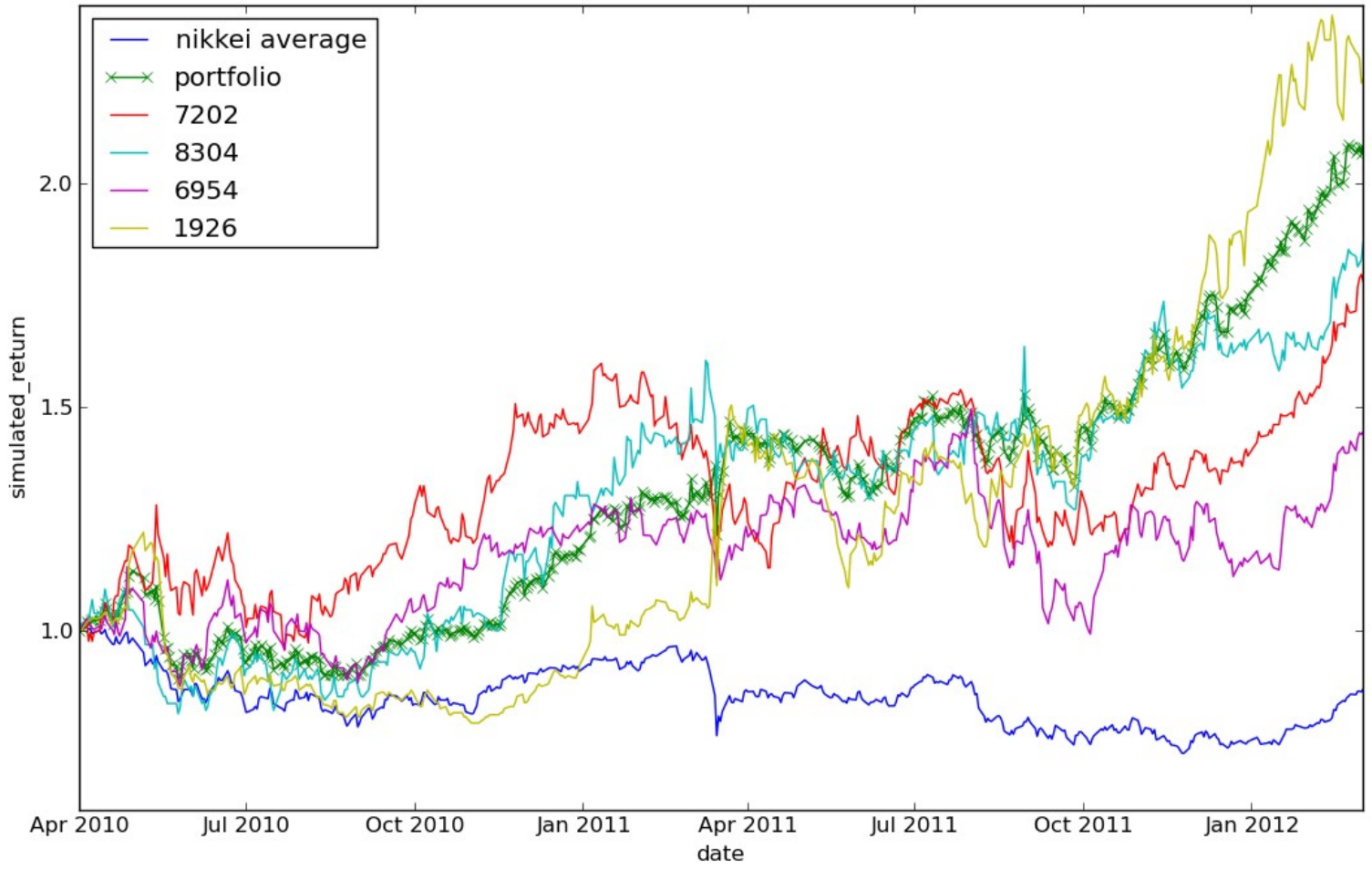
PyFinance 作ってます。共同開発者募集中!

機能

* ndarray を利用した時系列のクラス
* YahooFinance jp から株価の取得
* SQLite への保存 / からの読み込み
* 20 以上のテクニカル指標の計算
* OpenOpt を用いたポートフォリオ最適化
* And More!

github レポジトリにて近日公開予定
https://github.com/lucidfrontier45

宣伝 ②



緑の線が最適化したポートフォリオ

終わり

ご清聴ありがとうございました。

本日使用したソースコードはTokyoScipyのgithubレポジトリから入手可能です。
https://github.com/tokyo-scipy/archive