



Using Cython to optimize Python and interface with C

Tiago M. D. Pereira
ANU RSAA

Please download

Examples and source code:

www.mso.anu.edu.au/~tiago/cython_tutorial.tar.gz

Cython:

www.cython.org/Cython-0.11.tar.gz

or, if you have easy_install:

```
$ easy_install cython
```

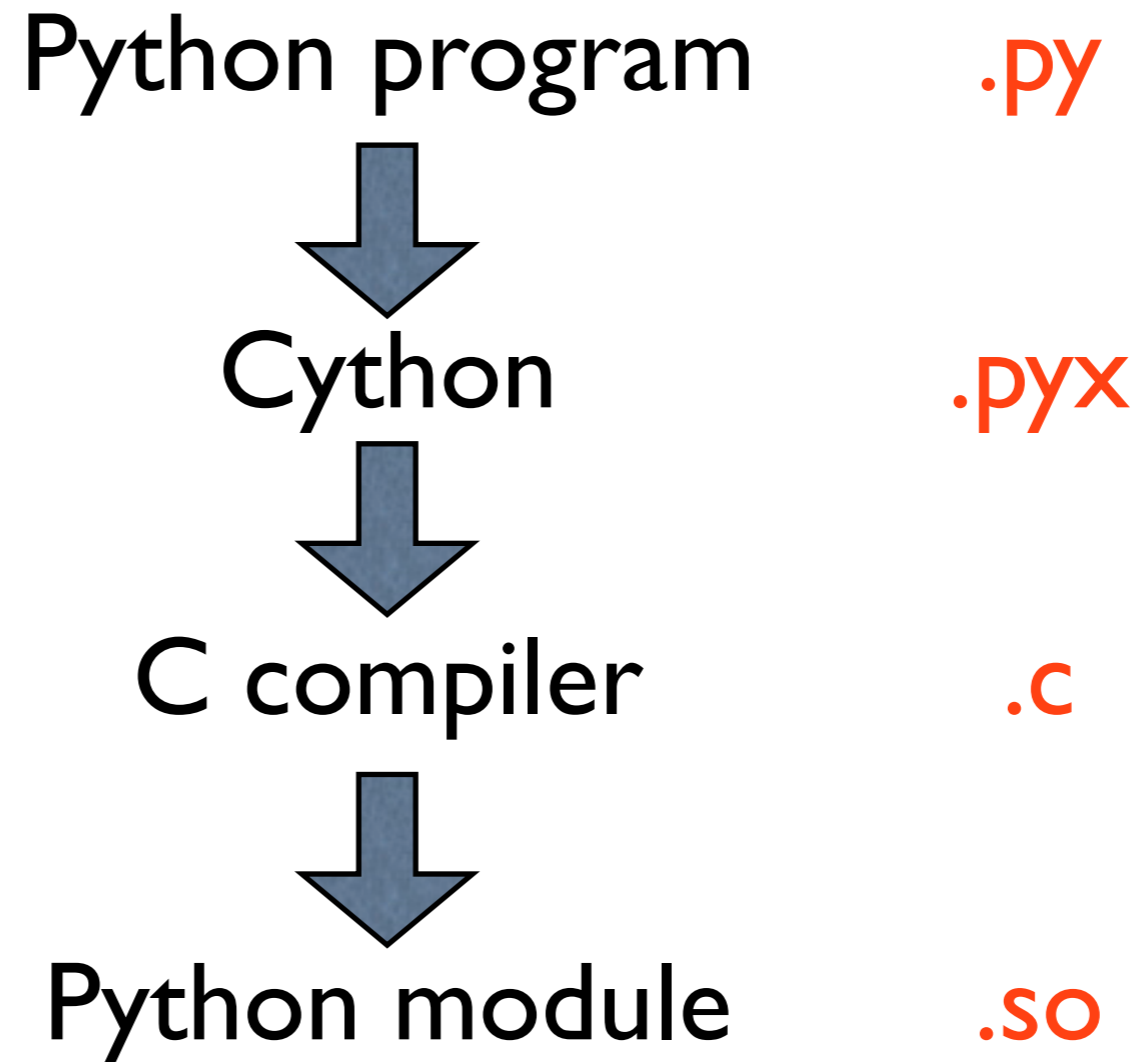
Why bother?

- Why C?
- Several ways to interface Python with other languages (swig, scipy.weave, inline, f2py, ...)
- Cython: a good candidate

Introducing Cython

- Optimize Python code into fast modules
(part I)
- Interface Python with external C code
(part II)

Introducing Cython



First example: hello world

1. Create helloworld.pyx with the following:

```
print "Hello world!"
```

2. Create setup.py:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("helloworld", ["helloworld.pyx"])] )
```

3. Run it

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

4. Import python module

```
>>> import helloworld
```

Optimizing Python

- integrate.py:

files in part I/

```
def f(x):  
    return x*x*x - 3*x
```

```
def integrate(a,b,n):  
    s = 0  
    dx = (b-a)/n  
    for i in range(n):  
        s += f(a+i*dx)  
    return s*dx
```

Optimizing Python

- integrate1.pyx:

files in part I/

```
cdef double f(double x):  
    return x*x*x - 3*x
```

```
def integrate1(double a, double b, int N):  
    cdef double s = 0  
    cdef double dx = (b-a)/N  
    cdef int i  
    for i in range(N):  
        s += f(a+i*dx)  
    return s*dx
```

Use provided setup.py to build module!

```
In [1]: from integrate import *
```

```
In [2]: from integrate1 import *
```

```
In [3]: %timeit -n2 -r3 integrate(0.,5.,1000000)
2 loops, best of 3: 959 ms per loop
```

```
In [4]: %timeit -n2 -r3 integrate1(0.,5.,1000000)
2 loops, best of 3: 7.16 ms per loop
```

Interfacing with C

- sum.c

files in part2/

```
float sum(float *a, int m)
{
    int i;
    float s=0.0;
    for(i=0; i<m; i++)
        s += a[i];
    return s;
}
```

Interfacing with C

- `sum1.pyx`

files in part2/

```
import numpy as np
cimport numpy as np
ctypedef np.float32_t DTYPE_t
```

```
cdef extern float sum(float *a, int m)
```

```
def sum1(np.ndarray[DTYPE_t, ndim=1] a):
    cdef int N = a.shape[0]
    return sum(<float *>a.data, N)
```

Use Makefile to build module! (make `sum1.so`)

Change it for your system..

Interfacing with C

- matmul.c

files in part2/

```
void matmul(float **a, float **b, float **c, int m)
{
    int i, j, k;
    for(i=0; i<m; i++)
        for(j=0; j<m; j++)
            c[i][j] = 0.0;
    for(i=0; i<m; i++)
        for(k=0; k<m; k++)
            for(j=0; j<m; j++)
                c[i][j] += a[i][k]*b[k][j];
}
```

Interfacing with C

- matmul1.pyx

files in part2/

```
import numpy as np
cimport numpy as np
from stdlib cimport free, malloc
cdef extern from "stdlib.h":
    void *memcpy(void *dst, void *src, long n)
DTYPE = np.float32
ctypedef np.float32_t DTYPE_t

cdef extern float matmul(float **a, float **b, float **c, int m)
```

Interfacing with C

- matmul1.pyx

files in part2/

convert from numpy to C array

```
cdef inline float **numpy2c_float(np.ndarray a):
    cdef int m = a.shape[0]
    cdef int n = a.shape[1]
    cdef int i
    cdef float **data
    data = <float **> malloc(m*sizeof(float*))
    for i in range(m):
        data[i] = &<float *>a.data[i*n]
    return data
```

Interfacing with C

- matmul1.pyx

files in part2/

convert from C array to numpy

```
cdef inline np.ndarray c2np_float(float **a, int m, int n):
    cdef np.ndarray[DTYPE_t, ndim=2] result = np.zeros((m, n), dtype=DTYPE)
    cdef float *dest
    cdef int i
    dest = <float *> malloc(m*n*sizeof(float*))
    for i in range(m):
        memcpy(dest + i*n, a[i], m*sizeof(float*))
        free(a[i])
    memcpy(result.data, dest, m*n*sizeof(float*))
    free(dest)
    free(a)
    return result
```

Interfacing with C

- `matmul1.pyx`

files in [part2/](#)

```
def matmul1(np.ndarray[DTYPE_t, ndim=2] a, np.ndarray[DTYPE_t, ndim=2] b):

    cdef int N = a.shape[0]
    cdef int i
    cdef float **a_c
    cdef float **b_c
    cdef float **res

    # check if square arrays:
    if a.shape[1] != N or b.shape[0] != N or b.shape[1] != N:
        raise ValueError, 'matmul1: need square arrays for multiplication!'

    # check if contiguous, if not force C contiguous arrays
    if not (<object>a).flags["C_CONTIGUOUS"]:
        a = a.copy('C')
    if not (<object>b).flags["C_CONTIGUOUS"]:
        b = b.copy('C')
```

Interfacing with C

- `matmul1.pyx`

files in part2/

```
# convert using the function
a_c = npy2c_float(a)
b_c = npy2c_float(b)

# allocate res
res = <float **> malloc(N*sizeof(float*))
for i in range(N):
    res[i] = <float *> malloc(N * sizeof(float))

matmul(a_c,b_c,res,N)

free(a_c)
free(b_c)

# convert to numpy array and free res
result = c2npy_float(res,N,N)

return result
```

Interfacing with C

- Use `integrate.py` and `run_matmul.c` to benchmark!