**EBook Gratuito**

# APPRENDIMENTO cython

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#cython

# Sommario

# Di

# Capitolo 1: Iniziare con Cython

## Osservazioni

## Cos'è Cython?

Il linguaggio di programmazione Cython arricchisce Python con la tipizzazione statica di tipo C, la capacità di chiamare direttamente funzioni C e molte altre funzionalità. Ciò consente di raggiungere prestazioni a livello C pur utilizzando una sintassi simile a Python.

## Come funziona?

Il codice Cython è compilato usando il compilatore cython source-to-source per creare codice C o C ++, che a sua volta può essere compilato usando un compilatore C. Ciò consente di creare estensioni che possono essere importate da Python o eseguibili.

Il guadagno di prestazioni principale che Cython può raggiungere rispetto ai puri steli Python aggira l'API CPython. Ad esempio quando si aggiungono due interi, Python esegue un controllo di tipo per ogni variabile, trova una funzione di aggiunta che soddisfa i tipi trovati e chiama tale funzione. Nel codice C generato da Cython, i tipi sono già noti e viene effettuata una sola chiamata a funzione. Quindi, Cython brilla soprattutto per problemi matematici in cui i tipi sono chiari.

## Come posso usarlo per velocizzare il mio codice?

Un caso d'uso comune, quando si tenta di accelerare un programma usando Cython, è di profilare il codice e spostare le parti computazionalmente costose in moduli Cython compilati. Ciò consente di mantenere la sintassi di Python per la maggior parte del codice e applicare la velocità laddove è più necessaria.

## Examples

### Installare Cython

Per utilizzare Cython sono necessarie due cose. Il pacchetto Cython stesso, che contiene il `cython` source-to-source e le interfacce Cython a diverse librerie C e Python (per esempio numpy). Per compilare il codice C generato dal compilatore `cython` , è necessario un compilatore C.

# Passaggio 1: installazione di Cython

## Sistema agnostico

Cython può essere installato con diversi sistemi di gestione pacchetti agnostici di sistema. Questi includono:

1. PyPI via pip o easy_install:

```
$ pip install cython
$ easy_install cython
```

2. anaconda usando conda:

```
$ conda install cython
```

3. Enthought canopy usando il gestore di pacchetti enpkg:

```
$ enpkg cython
```

Anche il codice sorgente può essere scaricato da github e installato manualmente usando:

```
$ python setup.py install
```

## Ubuntu, Debian

Per Ubuntu sono disponibili i pacchetti `cython` e `cython3` . Si noti che questi forniscono una versione precedente rispetto alle opzioni di installazione menzionate sopra.

```
$ apt-get install cython cython3
```

## finestre

Per Windows, un file .whl che può essere installato tramite pip è fornito da una terza parte. I dettagli sull'installazione di un file .whl su Windows sono disponibili qui .

---

# Passaggio 2: installazione di un compilatore C

Per compilare i file C generati da Cython, è necessario un compilatore per C e C ++. Il compilatore gcc è raccomandato e può essere installato come segue.

## Ubuntu, Debian

Il pacchetto `build-essential` contiene tutto ciò che è necessario. Può essere installato dai repository usando:

```
$ sudo apt-get install build-essential
```

## MAC

Gli strumenti per sviluppatori XCode contengono un compilatore simile a gcc.

### finestre

MinGW (Minimalist GNU per Windows) contiene una versione Windows di gcc. È anche possibile utilizzare il compilatore di Visual Studio.

### Ciao mondo

Un file python di Cython deve essere tradotto in codice C ( *criptato* ) e compilato prima di poter essere utilizzato da Python. Un approccio comune è quello di creare un modulo di estensione che viene poi importato in un programma Python.

# Codice

Per questo esempio creiamo tre file:

- `hello.pyx` contiene il codice Cython.
- `test.py` è uno script Python che usa l'estensione ciao.
- `setup.py` è usato per compilare il codice Cython.

## hello.pyx

```
from libc.math cimport pow

cdef double square_and_add (double x):
    """Compute x^2 + x as double.

    This is a cdef function that can be called from within
    a Cython program, but not from Python.
    """
    return pow(x, 2.0) + x

cpdef print_result (double x):
    """This is a cpdef function that can be called from Python."""
    print("({} ^ 2) + {} = {}".format(x, x, square_and_add(x)))
```

## test.py

```
# Import the extension module hello.
import hello

# Call the print_result method
```

```
hello.print_result(23.0)
```

### setup.py

```
from distutils.core import Extension, setup
from Cython.Build import cythonize

# define an extension that will be cythonized and compiled
ext = Extension(name="hello", sources=["hello.pyx"])
setup(ext_modules=cythonize(ext))
```

## compilazione

Questo può essere fatto usando `cython hello.pyx` per tradurre il codice in C e poi compilarlo usando `gcc` . Un modo più semplice è lasciare che le distutils gestiscano questo:

```
$ ls
hello.pyx  setup.py  test.py
$ python setup.py build_ext --inplace
$ ls
build  hello.c  hello.cpython-34m.so  hello.pyx  setup.py  test.py
```

Il file oggetto condiviso (.so) può essere importato e utilizzato da Python, quindi ora possiamo eseguire `test.py` :

```
$ python test.py
(23.0 ^ 2) + 23.0 = 552.0
```

Leggi Iniziare con Cython online: https://riptutorial.com/it/cython/topic/2925/iniziare-con-cython

# Capitolo 2: Bundling Cython

## Examples

### Raggruppamento di un programma Cython usando pyinstaller

Inizia da un programma Cython con un punto di accesso:

```
def do_stuff():
    cdef int a,b,c
    a = 1
    b = 2
    c = 3
    print("Hello World!")
    print([a,b,c])
    input("Press Enter to continue.")
```

Creare un file `setup.py` nella stessa cartella:

```
from distutils.core import setup
from Cython.Build import cythonize
setup(
    name = "Hello World",
    ext_modules = cythonize('program.pyx'),
)
```

Eseguendolo con `python setup.py build_ext --inplace` produrrà una libreria `.pyd` in una sottocartella.

Successivamente, crea uno script Python vanilla usando la libreria (ad esempio, `main.py` ) e metti il file `.pyd` accanto a esso:

```
import program
program.do_stuff()
```

Usa PyInstaller per raggrupparlo `pyinstaller --onefile "main.py"` . Questo creerà una sottocartella contenente l'eseguibile di una dimensione di 4 MB + contenente la libreria più il runtime di python.

### Configurazione automatica (Windows)

Per l'automazione della procedura di cui sopra in Windows utilizzare un `.bat` dei contenuti simili:

```
del "main.exe"
python setup.py build_ext --inplace
del "*.c"
rmdir /s /q ".\build"
pyinstaller --onefile "main.py"
copy /y ".\dist\main.exe" ".\main.exe"
rmdir /s /q ".\dist"
rmdir /s /q ".\build"
```

```
del "*.spec"
del "*.pyd"
```

## Aggiunta di Numpy al pacchetto

Per aggiungere Numpy al pacchetto, modificare il file `setup.py` con `include_dirs` e importare il numpy nello script Python wrapper per notificare Pyinstaller.

`program.pyx` :

```
import numpy as np
cimport numpy as np


def do_stuff():
    print("Hello World!")
    cdef int n
    n = 2
    r = np.random.randint(1,5)
    print("A random number: "+str(r))
    print("A random number multiplied by 2 (made by cdef):"+str(r*n))
    input("Press Enter to continue.")
```

`setup.py` :

```
from distutils.core import setup, Extension
from Cython.Build import cythonize
import numpy

setup(
    ext_modules=cythonize("hello.pyx"),
    include_dirs=[numpy.get_include()]
)
```

`main.py` :

```
import program
import numpy
program.do_stuff()
```

Leggi Bundling Cython online: https://riptutorial.com/it/cython/topic/6386/bundling-cython

# Capitolo 3: Wrapping C ++

## Examples

### Wrapping di una DLL: da C ++ a Cython a Python

Questo dimostra un esempio non banale di wrapping di una dll C ++ con Cython. Coprirà i seguenti passaggi principali:

- Creare un esempio DLL con C ++ utilizzando Visual Studio.
- Avvolgi la DLL con Cython in modo che possa essere chiamata in Python.

**Si presume che Cython sia installato e possa importarlo con successo in Python.**

Per il passaggio della DLL, si presume anche che si abbia familiarità con la creazione di una DLL in Visual Studio.

L'esempio completo include la creazione dei seguenti file:

1. `complexFunLib.h` : file di intestazione per l'origine DLL C ++
2. `complexFunLib.cpp` : file CPP per l'origine DLL C ++
3. `ccomplexFunLib.pxd` : file "header" di Cython
4. `complexFunLib.pyx` : file "wrapper" di Cython
5. `setup.py` : file di installazione di Python per la creazione di `complexFunLib.pyd` con Cython
6. `run.py` : esempio di file Python che importa la DLL compilata, compilata con Cython

## C ++ DLL Origine: `complexFunLib.h` e `complexFunLib.cpp`

**Salta questo se hai già una DLL e un file sorgente di intestazione.** Per prima cosa, creiamo il sorgente C ++ da cui la DLL verrà compilata usando Visual Studio. In questo caso, vogliamo eseguire calcoli di array veloci con la complessa funzione esponenziale. Le seguenti due funzioni eseguono il calcolo `k*exp(ee)` sugli array `k` ed `ee` , dove i risultati sono memorizzati in `res` . Ci sono due funzioni per ospitare sia la precisione singola che quella doppia. Nota che queste funzioni di esempio usano OpenMP, quindi assicurati che OpenMP sia abilitato nelle opzioni di Visual Studio per il progetto.

**H File**

```
// Avoids C++ name mangling with extern "C"
#define EXTERN_DLL_EXPORT extern "C" __declspec(dllexport)
#include <complex>
#include <stdlib.h>

// Handles 64 bit complex numbers, i.e. two 32 bit (4 byte) floating point numbers
EXTERN_DLL_EXPORT void mp_mlt_exp_c4(std::complex<float>* k,
                                     std::complex<float>* ee,
                                     int sz,
                                     std::complex<float>* res,
```

```
                                    int threads);

// Handles 128 bit complex numbers, i.e. two 64 bit (8 byte) floating point numbers
EXTERN_DLL_EXPORT void mp_mlt_exp_c8(std::complex<double>* k,
std::complex<double>* ee,
                                    int sz,
                                    std::complex<double>* res,
                                    int threads);
```

**File CPP**

```
#include "stdafx.h"
#include <stdio.h>
#include <omp.h>
#include "complexFunLib.h"

void mp_mlt_exp_c4(std::complex<float>* k,
                   std::complex<float>* ee,
                   int sz,
                   std::complex<float>* res,
                   int threads)
{
    // Use Open MP parallel directive for multiprocessing
    #pragma omp parallel num_threads(threads)
    {
        #pragma omp for
        for (int i = 0; i < sz; i++) res[i] = k[i] * exp(ee[i]);
    }
}

void mp_mlt_exp_c8(std::complex<double>* k,
                   std::complex<double>* ee,
                   int sz, std::complex<double>* res,
                   int threads)
{
    // Use Open MP parallel directive for multiprocessing
    #pragma omp parallel num_threads(threads)
    {
        #pragma omp for
        for (int i = 0; i < sz; i++) res[i] = k[i] * exp(ee[i]);
    }
}
```

## Origine Cython: `ccomplexFunLib.pxd` e `complexFunLib.pyx`

Successivamente, creiamo i file sorgente Cython necessari per avvolgere la DLL C ++. In questo passaggio, facciamo le seguenti ipotesi:

- Hai installato Cython
- Hai una DLL funzionante, ad esempio quella descritta sopra

L'obiettivo finale è creare questi file sorgente Cython in combinazione con la DLL originale per compilare un file `.pyd` che può essere importato come un modulo Python ed espone le funzioni scritte in C ++.

**File PXD**

Questo file corrisponde al file di intestazione C ++. Nella maggior parte dei casi, è possibile copiare e incollare l'intestazione su questo file con modifiche specifiche di Cython minori. In questo caso, sono stati utilizzati i tipi di complessi Cython specifici. Notare l'aggiunta di `c` all'inizio di `ccomplexFunLib.pxd` . Questo non è necessario, ma abbiamo scoperto che una tale convenzione di denominazione aiuta a mantenere l'organizzazione.

```
cdef extern from "complexFunLib.h":
    void mp_mlt_exp_c4(float complex* k, float complex* ee, int sz,
                       float complex* res, int threads);
    void mp_mlt_exp_c8(double complex* k, double complex* ee, int sz,
                       double complex* res, int threads);
```

### File PYX

Questo file corrisponde al file di origine c ++ `cpp` . In questo esempio, passeremo i puntatori agli oggetti Numpy `ndarray` alle funzioni DLL di importazione. È anche possibile utilizzare l'oggetto built in Cython `memoryview` per gli array, ma le sue prestazioni potrebbero non essere buone come `ndarray` oggetti `ndarray` (tuttavia la sintassi è significativamente più pulita).

```
cimport ccomplexFunLib  # Import the pxd "header"
# Note for Numpy imports, the C import most come AFTER the Python import
import numpy as np  # Import the Python Numpy
cimport numpy as np  # Import the C Numpy

# Import some functionality from Python and the C stdlib
from cpython.pycapsule cimport *

# Python wrapper functions.
# Note that types can be delcared in the signature

def mp_exp_c4(np.ndarray[np.complex64_t, ndim=1] k,
              np.ndarray[np.complex64_t, ndim=1] ee,
              int sz,
              np.ndarray[np.complex64_t, ndim=1] res,
              int threads):
    '''
    TODO: Python docstring
    '''
    # Call the imported DLL functions on the parameters.
    # Notice that we are passing a pointer to the first element in each array
    ccomplexFunLib.mp_mlt_exp_c4(&k[0], &ee[0], sz, &res[0], threads)

def mp_exp_c8(np.ndarray[np.complex128_t, ndim=1] k,
              np.ndarray[np.complex128_t, ndim=1] ee,
              int sz,
              np.ndarray[np.complex128_t, ndim=1] res,
              int threads):
    '''
    TODO: Python docstring
    '''
    ccomplexFunLib.mp_mlt_exp_c8(&k[0], &ee[0], sz, &res[0], threads)
```

## Origine Python: `setup.py` e `run.py`

### setup.py

Questo file è un file Python che esegue la compilazione di Cython. Il suo scopo è generare il file `.pyd` compilato che possa essere importato dai moduli Python. In questo esempio, abbiamo mantenuto tutti i file richiesti (ad esempio `complexFunLib.h` , `complexFunLib.dll` , `ccomplexFunLib.pxd` e `complexFunLib.pyx` ) nella stessa directory di `setup.py` .

Una volta creato questo file, dovrebbe essere eseguito dalla riga di comando con i parametri: `build_ext --inplace`

Una volta eseguito questo file, dovrebbe produrre un file `.pyd` senza generare errori. Nota che in alcuni casi, se c'è un errore, il file `.pyd` può essere creato ma non è valido. Assicurarsi che non siano stati generati errori nell'esecuzione di `setup.py` prima di utilizzare il file `.pyd` generato.

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

ext_modules = [
    Extension('complexFunLib',
              ['complexFunLib.pyx'],
              # Note here that the C++ language was specified
              # The default language is C
              language="c++",
              libraries=['complexFunLib'],
              library_dirs=['.'])
    ]

setup(
    name = 'complexFunLib',
    cmdclass = {'build_ext': build_ext},
    ext_modules = ext_modules,
    include_dirs=[np.get_include()]  # This gets all the required Numpy core files
)
```

**run.py**

Ora `complexFunLib` può essere importato direttamente in un modulo Python e richiamate le funzioni DLL avvolte.

```
import complexFunLib
import numpy as np

# Create arrays of non-trivial complex numbers to be exponentiated,
# i.e. res = k*exp(ee)
k = np.ones(int(2.5e5), dtype='complex64')*1.1234 + np.complex64(1.1234j)
ee = np.ones(int(2.5e5), dtype='complex64')*1.1234 + np.complex64(1.1234j)
sz = k.size  # Get size integer
res = np.zeros(int(2.5e5), dtype='complex64')  # Create array for results

# Call function
complexFunLib.mp_exp_c4(k, ee, sz, res, 8)

# Print results
print(res)
```

Leggi Wrapping C ++ online: https://riptutorial.com/it/cython/topic/3525/wrapping-c-plusplus

# Capitolo 4: Wrapping C Code

## Examples

### Utilizzo di funzioni da una libreria C personalizzata

Abbiamo una libreria C chiamata `my_random` che produce numeri casuali da una distribuzione personalizzata. Fornisce due funzioni che vogliamo utilizzare: `set_seed(long seed)` e `rand()` (e molte altre non sono necessarie). Per poterli utilizzare in Cython, è necessario

1. definire un'interfaccia nel file .pxd e
2. chiama la funzione nel file .pyx.

# Codice

## test_extern.pxd

```
# extern blocks define interfaces for Cython to C code
cdef extern from "my_random.h":
    double rand()
    void c_set_seed "set_seed" (long seed) # rename C version of set_seed to c_set_seed to
avoid naming conflict
```

## test_extern.pyx

```
def set_seed (long seed):
    """Pass the seed on to the c version of set_seed in my_random."""
    c_set_seed(seed)

cpdef get_successes (int x, double threshold):
    """Create a list with x results of rand <= threshold

    Use the custom rand function from my_random.
    """
    cdef:
        list successes = []
        int i
    for i in range(x):
        if rand() <= threshold:
            successes.append(True)
        else:
            successes.append(False)
    return successes
```

Leggi Wrapping C Code online: https://riptutorial.com/it/cython/topic/3626/wrapping-c-code

# Titoli di coda

| S. No | Capitoli | Contributors |
|---|---|---|
| 1 | Iniziare con Cython | Community, J.J. Hakala, Keith L, m00am |
| 2 | Bundling Cython | Andrii Magalich |
| 3 | Wrapping C ++ | J.J. Hakala, Keith L, Kevin Pasquarella |
| 4 | Wrapping C Code | m00am |