eBook Gratuit

APPRENEZ cython

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#cython

Table des matières

À propos 1

Chapitre 1: Commencer avec cython 2

Remarques 2

Qu'est ce que Cython? 2

Comment ça marche? 2

Comment l'utiliser pour accélérer mon code? 2

Examples 2

Installer Cython 2

Étape 1: Installation de Cython 2

Système agnostique 3

Ubuntu, Debian 3

les fenêtres 3

Étape 2: Installation d'un compilateur C 3

Ubuntu, Debian 3

MAC 4

les fenêtres 4

Bonjour le monde 4

Code 4

bonjour.pyx 4

test.py 4

setup.py 5

Compiler 5

Chapitre 2: Emballage C ++ 6

Examples 6

Emballage d'une DLL: C ++ en Cython en Python 6

Source DLL C ++: complexFunLib.h et complexFunLib.cpp 6

Source Cython: ccomplexFunLib.pxd et complexFunLib.pyx 7

Source Python: setup.py et run.py 8

Chapitre 3: Emballage C Code 11

Examples 11
Utilisation des fonctions d'une bibliothèque C personnalisée 11
Code 11
test_extern.pxd 11
test_extern.pyx 11
Chapitre 4: Groupement de Cython 12
Examples 12
Regrouper un programme Cython en utilisant pyinstaller 12
Automatisation de la construction (Windows) 12
Ajout de Numpy au bundle 13
Crédits 14

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: cython

It is an unofficial and free cython ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official cython.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Commencer avec cython

Remarques

Qu'est ce que Cython?

Le langage de programmation Cython enrichit Python par un typage statique de type C, la possibilité d'appeler directement les fonctions C et plusieurs autres fonctionnalités. Cela permet d'atteindre des performances de niveau C tout en utilisant une syntaxe de type Python.

Comment ça marche?

Le code Cython est compilé à l'aide du compilateur cython source-to-source pour créer du code C ou C ++, qui à son tour peut être compilé à l'aide d'un compilateur C. Cela permet de créer des extensions pouvant être importées depuis Python ou des exécutables.

Le principal gain de performances que Cython peut atteindre, contrairement à Python pur, provient du contournement de l'API CPython. Par exemple, lors de l'ajout de deux entiers, Python effectue une vérification de type pour chaque variable, trouve une fonction d'ajout qui satisfait les types trouvés et appelle cette fonction. Dans le code C généré par Cython, les types sont déjà connus et un seul appel de fonction est effectué. Par conséquent, Cython brille particulièrement pour les problèmes mathématiques dans lesquels les types sont clairs.

Comment l'utiliser pour accélérer mon code?

Un cas d'utilisation courant, lorsque vous essayez d'accélérer un programme en utilisant Cython, consiste à profiler le code et à déplacer les pièces coûteuses en calculs vers des modules Cython compilés. Cela permet de conserver la syntaxe Python pour la majeure partie du code et d'appliquer l'accélération là où c'est le plus nécessaire.

Examples

Installer Cython

Pour utiliser Cython, deux choses sont nécessaires. Le paquet Cython lui-même, qui contient le `cython` source-à-source `cython` et les interfaces Cython avec plusieurs bibliothèques C et Python (par exemple numpy). Pour compiler le code C généré par le compilateur `cython` , un compilateur C est nécessaire.

Étape 1: Installation de Cython

Système agnostique

Cython peut être installé avec plusieurs systèmes de gestion de paquets indépendants du système. Ceux-ci inclus:

1. PyPI via pip ou easy_install:

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

2. anaconda en utilisant conda:

```
$ conda install cython
```

3. Canopy passionné en utilisant le gestionnaire de paquets enpkg:

```
$ enpkg cython
```

De plus, le code source peut être téléchargé depuis github et installé manuellement en utilisant:

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

Ubuntu, Debian

Pour Ubuntu, les paquets `cython` et `cython3` sont disponibles. Notez que ceux-ci fournissent une version plus ancienne que les options d'installation mentionnées ci-dessus.

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

les fenêtres

Pour Windows, un fichier .whl pouvant être installé à l'aide de pip est fourni par un tiers. Les détails sur l'installation d'un fichier .whl sous Windows peuvent être trouvés ici .

Étape 2: Installation d'un compilateur C

Pour compiler les fichiers C générés par Cython, un compilateur pour C et C ++ est nécessaire. Le compilateur gcc est recommandé et peut être installé comme suit.

Ubuntu, Debian

Le package `build-essential` contient tout ce qui est nécessaire. Il peut être installé à partir des référentiels en utilisant:

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

MAC

Les outils de développement XCode contiennent un compilateur de type gcc.

les fenêtres

MinGW (Minimalist GNU for Windows) contient une version Windows de gcc. Le compilateur de Visual Studio peut également être utilisé.

Bonjour le monde

Un fichier Cython pyx doit être traduit en code C (*numérisé*) et compilé avant de pouvoir être utilisé à partir de Python. Une approche courante consiste à créer un module d'extension qui est ensuite importé dans un programme Python.

Code

Pour cet exemple, nous créons trois fichiers:

- `hello.pyx` contient le code Cython.
- `test.py` est un script Python qui utilise l'extension hello.
- `setup.py` est utilisé pour compiler le code Cython.

bonjour.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))
```

Compiler

Cela peut être fait en utilisant `cython hello.pyx` pour traduire le code en C, puis le compiler en utilisant `gcc` . Un moyen plus simple est de laisser les agents manipuler ceci:

```
$ 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
```

Le fichier objet partagé (.so) peut être importé et utilisé à partir de Python, nous pouvons maintenant exécuter le `test.py` :

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

Lire Commencer avec cython en ligne: https://riptutorial.com/fr/cython/topic/2925/commencer-avec-cython

Chapitre 2: Emballage C ++

Examples

Emballage d'une DLL: C ++ en Cython en Python

Cela illustre un exemple non trivial de l'encapsulation d'un dll C ++ avec Cython. Il couvrira les principales étapes suivantes:

- Créez un exemple de DLL avec C ++ à l'aide de Visual Studio.
- Enveloppez la DLL avec Cython pour qu'il puisse être appelé en Python.

Il est supposé que Cython est installé et peut l'importer avec succès en Python.

Pour l'étape DLL, il est également supposé que vous êtes familiarisé avec la création d'une DLL dans Visual Studio.

L'exemple complet comprend la création des fichiers suivants:

1. `complexFunLib.h` : fichier d'en-tête pour la source DLL C ++
2. `complexFunLib.cpp` : fichier CPP pour la source DLL C ++
3. `ccomplexFunLib.pxd` : fichier "en-tête" Cython
4. `complexFunLib.pyx` : fichier "wrapper" Cython
5. `setup.py` : fichier d'installation de Python pour créer `complexFunLib.pyd` avec Cython
6. `run.py` : Exemple de fichier Python qui importe la DLL compilée Cython

Source DLL C ++: `complexFunLib.h` et `complexFunLib.cpp`

Ignorez ceci si vous avez déjà un fichier source d'en-tête et de DLL. Tout d'abord, nous créons la source C ++ à partir de laquelle la DLL sera compilée à l'aide de Visual Studio. Dans ce cas, nous souhaitons effectuer des calculs de tableau rapides avec la fonction exponentielle complexe. Les deux fonctions suivantes effectuent le calcul `k*exp(ee)` sur les tableaux `k` et `ee` , où les résultats sont stockés dans `res` . Il y a deux fonctions pour adapter à la fois la précision simple et double. Notez que ces exemples de fonctions utilisent OpenMP. Assurez-vous donc que OpenMP est activé dans les options Visual Studio du projet.

Fichier h

```
// 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);
```

Fichier 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]);
    }
}
```

Source Cython: `ccomplexFunLib.pxd` et `complexFunLib.pyx`

Ensuite, nous créons les fichiers source Cython nécessaires pour envelopper la DLL C ++. Dans cette étape, nous faisons les hypothèses suivantes:

- Vous avez installé Cython
- Vous possédez une DLL de travail, par exemple celle décrite ci-dessus

Le but ultime est de créer un fichier `.pyd` pouvant être importé en tant que module Python et exposant les fonctions écrites en C ++.

Fichier PXD

Ce fichier correspond au fichier d'en-tête C ++. Dans la plupart des cas, vous pouvez copier-coller l'en-tête sur ce fichier avec des modifications mineures spécifiques à Cython. Dans ce cas, les types complexes Cython spécifiques ont été utilisés. Notez l'ajout de `c` au début de `ccomplexFunLib.pxd` . Ce n'est pas nécessaire, mais nous avons constaté qu'une telle convention de nommage aide à maintenir l'organisation.

```
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);
```

Fichier PYX

Ce fichier correspond au fichier source `cpp` C ++. Dans cet exemple, nous allons transmettre des pointeurs à des objets Numpy `ndarray` aux fonctions d'importation de DLL. Il est également possible d'utiliser l'objet Cython `memoryview` pour les tableaux, mais ses performances peuvent ne pas être aussi bonnes que `ndarray` objets `ndarray` (cependant, la syntaxe est nettement plus `ndarray`).

```
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)
```

Source Python: `setup.py` et `run.py`

setup.py

Ce fichier est un fichier Python qui exécute la compilation Cython. Son but est de générer le fichier `.pyd` compilé qui peut ensuite être importé par les modules Python. Dans cet exemple, nous avons conservé tous les fichiers requis (ie `complexFunLib.h` , `complexFunLib.dll` , `ccomplexFunLib.pxd` et `complexFunLib.pyx`) dans le même répertoire que `setup.py` .

Une fois ce fichier créé, il doit être exécuté à partir de la ligne de commande avec les paramètres suivants: `build_ext --inplace`

Une fois ce fichier exécuté, il doit générer un fichier `.pyd` sans `.pyd` aucune erreur. Notez que dans certains cas, s'il y a une erreur, le `.pyd` peut être créé mais n'est pas valide. Assurez-vous qu'aucune erreur n'a été générée lors de l'exécution de `setup.py` avant d'utiliser le `.pyd` généré.

```
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

Maintenant, `complexFunLib` peut être importé directement dans un module Python et les fonctions DLL encapsulées sont appelées.

```
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)
```

Lire Emballage C ++ en ligne: https://riptutorial.com/fr/cython/topic/3525/emballage-c-plusplus

Chapitre 3: Emballage C Code

Examples

Utilisation des fonctions d'une bibliothèque C personnalisée

Nous avons une bibliothèque C nommée `my_random` qui produit des nombres aléatoires à partir d'une distribution personnalisée. Il fournit deux fonctions que nous voulons utiliser: `set_seed(long seed)` et `rand()` (et bien d'autres dont nous n'avons pas besoin). Pour les utiliser en Cython, nous devons

1. définir une interface dans le fichier .pxd et
2. appelez la fonction dans le fichier .pyx.

Code

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
```

Lire Emballage C Code en ligne: https://riptutorial.com/fr/cython/topic/3626/emballage-c-code

Chapitre 4: Groupement de Cython

Examples

Regrouper un programme Cython en utilisant pyinstaller

Démarrer à partir d'un programme Cython avec un point d'entrée:

```
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.")
```

Créez un fichier `setup.py` dans le même dossier:

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

Le `python setup.py build_ext --inplace` avec `python setup.py build_ext --inplace` produira une bibliothèque `.pyd` dans un sous-dossier.

Après cela, créez un script Python en utilisant la bibliothèque (par exemple, `main.py`) et placez le fichier `.pyd` à côté:

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

Utilisez PyInstaller pour regrouper `pyinstaller --onefile "main.py"` . Cela créera un sous-dossier contenant l'exécutable d'une taille de 4 Mo + contenant la bibliothèque plus le runtime python.

Automatisation de la construction (Windows)

Pour l'automatisation de la procédure ci-dessus sous Windows, utilisez un `.bat` du contenu similaire:

```
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"
```

Ajout de Numpy au bundle

Pour ajouter Numpy à l'ensemble, modifiez le mot-clé `setup.py` avec `include_dirs` et importez-le dans le script Python de l'encapsuleur pour notifier 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()
```

Lire Groupement de Cython en ligne: https://riptutorial.com/fr/cython/topic/6386/groupement-de-cython

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec cython	Community, J.J. Hakala, Keith L, m00am
2	Emballage C ++	J.J. Hakala, Keith L, Kevin Pasquarella
3	Emballage C Code	m00am
4	Groupement de Cython	Andrii Magalich