# LEARNING cython

#cython

# Table of Contents

# About

# Chapter 1: Getting started with cython

## Remarks

## What is Cython?

The Cython programming language enriches Python by C-like static typing, the ability to directly call C functions, and several other features. This allows to reach C-level performance while still using a Python-like syntax.

## How does it work?

Cython code is compiled using the cython source-to-source compiler to create C or C++ code, which in turn can be compiled using a C compiler. This allows to create extensions that can be imported from Python or executables.

The main performance gain Cython can reach in contrast to pure Python stems from bypassing the CPython API. For example when adding two integers, Python performs a type check for each variable, finds an add function that satisfies the found types, and calls that function. In the Cython-generated C code, the types are already know and only one function call to is made. Hence, Cython especially shines for mathematic problems in which the types are clear.

## How do I use it to speed up my code?

A common use case, when trying to speed up a program using Cython, is to profile the code and move the computationally expensive parts to compiled Cython modules. This allows to retain Python syntax for the bulk of the code and apply the speedup where it is most needed.

## Examples

### Installing Cython

To use Cython two things are needed.The Cython package itself, which contains the `cython` source-to-source compiler and Cython interfaces to several C and Python libraries (for example numpy). To compile the C code generated by the `cython` compiler, a C compiler is needed.

# Step 1: Installing Cython

## System Agnostic

Cython can be installed with several system agnostic package management systems. These include:

1. PyPI via pip or easy_install:

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

2. anaconda using conda:

   ```
   $ conda install cython
   ```

3. Enthought canopy using the enpkg package manager:

   ```
   $ enpkg cython
   ```

Also the source code can be downloaded from github and installed manually using:

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

### Ubuntu, Debian

For Ubuntu the packages `cython` and `cython3` are available. Note that these provide an older version than the installation options mentioned above.

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

### Windows

For Windows, a .whl file that can be installed using pip is provided by a third party. Details on installing a .whl file on Windows can be found here.

---

## Step 2: Installing a C Compiler

To compile the C files generated by Cython, a compiler for C and C++ is needed. The gcc compiler is recommended and can be installed as follows.

### Ubuntu, Debian

The `build-essential` package contains everything that is needed. It can be installed from the repositories using:

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

### MAC

The XCode developer tools contain a gcc like compiler.

### Windows

MinGW (Minimalist GNU for Windows) contains a Windows version of gcc. The compiler from Visual Studio can also be used.

### Hello World

A Cython pyx file needs to be translated to C code (*cythonized*) and compiled before it can be used from Python. A common approach is to create an extension module which is then imported in a Python program.

## Code

For this example we create three files:

- `hello.pyx` contains the Cython code.
- `test.py` is a Python script that uses the hello extension.
- `setup.py` is used to compile the Cython code.

### 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))
```

## Compiling

This can be done by, using `cython hello.pyx` to translate the code to C and then compile it using `gcc`. An easier way is to let distutils handle this:

```
$ 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
```

The shared object (.so) file can be imported and used from Python, so now we can run the `test.py`
:

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

Read Getting started with cython online: https://riptutorial.com/cython/topic/2925/getting-started-with-cython

# Chapter 2: Cython bundling

## Examples

### Bundling a Cython program using pyinstaller

Start from a Cython program with a entrypoint:

```
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.")
```

Create a `setup.py` file in the same folder:

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

Running it with `python setup.py build_ext --inplace` will produce a `.pyd` library in a subfolder.

After that, create a vanilla Python script using the library (e.g., `main.py`) and put the `.pyd` file beside it:

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

Use PyInstaller to bundle it `pyinstaller --onefile "main.py"`. This will create a subfolder containing the executable of a 4 MB+ size containing the library plus the python runtime.

### Automating build (Windows)

For automation of the above procedure in Windows use a `.bat` of the similar contents:

```
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"
```

## Adding Numpy to the bundle

To add Numpy to the bundle, modify the `setup.py` with `include_dirs` keyword and necessary import the numpy in the wrapper Python script to notify 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()
```

Read Cython bundling online: https://riptutorial.com/cython/topic/6386/cython-bundling

# Chapter 3: Wrapping C Code

## Examples

### Using functions from a custom C library

We have a C library named `my_random` that produces random numbers from a custom distribution. It provides two functions that we want to use: `set_seed(long seed)` and `rand()` (and many more we do not need). In order to use them in Cython we need to

1. define an interface in the .pxd file and
2. call the function in the .pyx file.

## 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
```

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

# Chapter 4: Wrapping C++

## Examples

### Wrapping a DLL: C++ to Cython to Python

This demonstrates a non-trivial example of wrapping a C++ dll with Cython. It will cover the following main steps:

- Create an example DLL with C++ using Visual Studio.
- Wrap the DLL with Cython so that it may be called in Python.

**It is assumed that you have Cython installed and can successfully import it in Python.**

For the DLL step, it is also assumed that you are familiar with creating a DLL in Visual Studio.

The full example includes the creation of the following files:

1. `complexFunLib.h`: Header file for the C++ DLL source
2. `complexFunLib.cpp`: CPP file for the C++ DLL source
3. `ccomplexFunLib.pxd`: Cython "header" file
4. `complexFunLib.pyx`: Cython "wrapper" file
5. `setup.py`: Python setup file for creating `complexFunLib.pyd` with Cython
6. `run.py`: Example Python file that imports the compiled, Cython wrapped DLL

## C++ DLL Source: `complexFunLib.h` and `complexFunLib.cpp`

**Skip this if you already have a DLL and header source file.** First, we create the C++ source from which the DLL will be compiled using Visual Studio. In this case, we want to do fast array calculations with the complex exponential function. The following two functions perform the calculation `k*exp(ee)` on arrays `k` and `ee`, where the results are stored in `res`. There are two functions to accommodate both single and double precision. Note that these example functions use OpenMP, so make sure that OpenMP is enabled in the Visual Studio options for the project.

#### 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);
```

**CPP File**

```
#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]);
    }
}
```

## Cython Source: `ccomplexFunLib.pxd` and `complexFunLib.pyx`

Next, we create the Cython source files necessary to wrap the C++ DLL. In this step, we make the following assumptions:

- You have installed Cython
- You possess a working DLL, e.g. the one described above

The ultimate goal is to create use these Cython source files in conjunction with the original DLL to compile a `.pyd` file which may be imported as a Python module and exposes the functions written in C++.

**PXD File**

This file corresponds the C++ header file. In most cases, you may copy-paste the header over to this file with minor Cython specific changes. In this case, the specific Cython complex types were used. Note the addition of `c` at the beginning of `ccomplexFunLib.pxd`. This is not necessary, but we have found that such a naming convention helps maintain organization.

```
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);
```

### PYX File

This file corresponds to the C++ `cpp` source file. In this example, we will be passing pointers to Numpy `ndarray` objects to the import DLL functions. It is also possible to use the built in Cython `memoryview` object for arrays, but its performance may not be as good as `ndarray` objects (however the syntax is significantly cleaner).

```
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)
```

## Python Source: `setup.py` and `run.py`

### setup.py

This file is a Python file that executes the Cython compilation. Its purpose is to generate the compiled `.pyd` file that may then be imported by Python modules. In this example, we have kept all the required files (i.e. `complexFunLib.h`, `complexFunLib.dll`, `ccomplexFunLib.pxd`, and `complexFunLib.pyx` ) in the same directory as `setup.py`.

Once this file is created, it should be run from the command line with parameters: `build_ext --inplace`

Once this file is executed, it should produce a `.pyd` file without raising any errors. Note that in some cases if there is a mistake the `.pyd` may be created but is invalid. Make sure that no errors were thrown in the execution of `setup.py` before using the generated `.pyd`.

```
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

Now `complexFunLib` may be imported directly into a Python module and the wrapped DLL functions called.

```
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)
```

Read Wrapping C++ online: https://riptutorial.com/cython/topic/3525/wrapping-cplusplus

# Credits

| S. No | Chapters | Contributors |
|---|---|---|
| 1 | Getting started with cython | Community, J.J. Hakala, Keith L, m00am |
| 2 | Cython bundling | Andrii Magalich |
| 3 | Wrapping C Code | m00am |
| 4 | Wrapping C++ | J.J. Hakala, Keith L, Kevin Pasquarella |