First steps in Cython

Natalí S. M. de Santi (@natalidesanti)

Python might be one of nowadays most popular programming languages, but it is definitely not the most efficient. Cython is a programming language that makes writing C extensions for the Python language as easy as Python itself. It has support for optional static type declarations as part of the language. The source code gets translated into optimized C/C++ code and compiled as Python extension modules. This allows for both very fast program execution and tight integration with external C libraries, while keeping up the high programmer productivity for which the Python language is well known [1].

Summing up: *Cython is a programming language based on Python with extra syntax to provide static type declarations. This takes advantage of the benefits of Python while allowing one to achieve the speed of C.*

1 Installing Cython

Unlike most Python software, Cython requires a C compiler to be present on the system. For Linux users The GNU C Compiler (gcc) is usually present, or easily available through the package system. On Ubuntu, for instance, the command:

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

will fetch everything you need. The simplest way of installing Cython is by using pip:

```
$ pip install Cython.
```

2 Building Cython code

Cython code must, unlike Python, be compiled. This happens in two stages:

- A .pyx file is compiled by Cython to a .c file, containing the code of a Python extension module.
- The .c file is compiled by a C compiler to a .so file which can be imported directly into a Python session. Distutils or setuptools take care of this part.

There are several ways to build Cython code, but here I will present using distutils/setuptools. To build a Cython module using *distutils* you need first to write a .pyx file. Here I will code a factorial program, that you can name as factorial_program.pyx containing:

```
#Factorial of a given number
def factorial(n):
    if n <= 1:
       return 1
    else:
       return n*factorial(n - 1)
```

The following could be a corresponding setup.py script:

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

setup(name='Factorial computation', ext_modules=cythonize("factorial_program.pyx"))
```

To build, run:

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

A new folder and a .so file will be created in the same directory of the files. Then, simply start a Python session, for example in a using_factorial_function.py, using the imported function as you see to fit:

```
from factorial_program import factorial
import time

n = int(input('Input your number: '))
t1 = time.time()
print("The factorial of this number is", factorial(n))
t2 = time.time()
print("Total time", t2 - t1)
```

We can compare its performance with the following just C and Python factorial programs:

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

long double factorial(long double n){
    if(n==0){
            return(1);
    }
    return(n * factorial(n-1));
}

clock_t start, end;
double cpu_time_used;

int main(){
    long double n;
    printf("Input your number\n");
    scanf("%Lf", &n);
    start = clock();
    printf("The factorial of this number is: %Lf\n", factorial(n));
    end = clock();
    cpu_time_used = ((double) (end - start))/CLOCKS_PER_SEC;
    printf("Total time %f\n", cpu_time_used);
    return(0);
}
```

```python
import time

def factorial(n):
    if n <= 1:
        return 1
    else:
        return n*factorial(n - 1)

n = int(input('Input your number: '))
t1 = time.time()
print("The factorial of this number is", factorial(n))
t2 = time.time()
print("Total time", t2 - t1)
```

In my personal computer, a `Intel Core i7-6500U CPU 2.50GHz`, for the factorial of $N = 100$, the programs runs in the following times:

- `Python`: $\Delta t \simeq 0.00030s$,
- `Cython`: $\Delta t \simeq 0.00017s$,
- `C`: $\Delta t \simeq 0.00013s$.

Of course `C` offers the best performance, while `Python` the worst. Notice that, just compiling the `Python` code using `Cython`, offers $\sim 40\%$ speedup.

3 Faster code via static typing

Cython is a `Python` compiler. This means that it can compile normal `Python` code almost without changes. However, for performance critical code, it is often helpful to add static type declarations, as they will allow `Cython` to step out of the dynamic nature of the `Python` code and generate simpler and faster `C` code - sometimes faster by orders of magnitude.

All `C` types are available for type declarations: integer and floating point types, complex numbers, structs, unions and pointer types. `Cython` can automatically and correctly convert between the types on assignment.

Types are declared via the **cdef keyword**.

3.1 Typing Variables

Simply compiling a simple code in `Cython` gives a reasonable speedup, but when you type the variables it becomes really faster, as you can see by the example of a simple integration. Comparing integration_python.py, that have runned in $\Delta t \simeq 0.00278s$, and just passing it into Cython, that have runned in $\Delta t \simeq 0.00172s$. we have around $\sim 40\%$ speedup.

3.1.1 integration_python.py

```python
import time

def f(x):
    return 1 + x + x**2

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

t1 = time.process_time()

a = 0
b = 1000
N = 10000

print(integrate_f(a, b, N))

t2 = time.process_time()

print('Total time:', t2-t1)
```

3.1.2 integration_cython.pyx

```python
def f(x):
    return 1 + x + x**2

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

3.1.3 setup-just_cython.py

```python
from distutils.core import setup
from Cython.Build import cythonize

setup(name='Integration', ext_modules=cythonize("integration_cython.pyx"))
```

3.1.4 integration-just_cython.py

```python
import time
from integration_cython import integrate_f

a = 0
b = 1000
N = 10000

```

```python
t1 = time.process_time()

print(integrate_f(a, b, N))

t2 = time.process_time()

print('Total time:', t2-t1)
```

With additional type declarations for the iterator i and variables s and dx, this might look like:

3.1.5 integration-static_cython.py

```python
def f(double x):
    return 1 + x + x**2

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

Since the iterator variable i is typed with C semantics, the for-loop will be compiled to pure C code. Typing a, s and dx is important as they are involved in arithmetic within the for-loop; typing b and N makes less of a difference, but in this case it is not much extra work to be consistent and type the entire function. This results in $\Delta t_{Cython}^{var\_typped} \simeq 0.00049s$, i.e., a ~ 80% speedup over the pure Python version.

4 Typing functions

Python function calls can be expensive - in Cython doubly so because one might need to convert to and from Python objects to do the call. Therefore Cython provides a syntax for declaring a C-style function, the **cdef keyword**. **cdef** declares a function in the layer of C language, then they can be called by C and Cython. They cannot be defined inside other functions, can take any type of argument and are quicker to call than def functions because they translate to a simple C function call. You can see this working in the following example:

```python
cdef double f(double x):
    return 1 + x + x**2

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

The running time of this prigram is $\Delta t_{Cython}^{func\_typped} \simeq 4.2 \cdot 10^{-5} s$. Therefore, this provides a speedup of ~ 100 times over pure Python.

Using the cpdef keyword instead of cdef, a Python wrapper is also created, so that the function is available both from Cython (fast, passing typed values directly) and from Python (wrapping values in Python objects). But the performance of this kind of declared functions are not so good as cdef.

5 Numpy array processing with Cython

If we leave the NumPy array in its current form, Cython works exactly as regular Python does by creating an object for each number in the array. To make things run faster we need to define a C data type for the NumPy array as well, just like for any other variable [1, 2].

5.1 Importing Numpy with C

If you are using NumPy you can optimize your code importing special compile-time information about the numpy module using:

```
$ import numpy as np
```

```
$ cimport numpy as np
```

This is done because the Cython "numpy" file has the data types for handling NumPy arrays.

5.2 Defining the NumPy array

The data type for NumPy arrays is **ndarray**, which stands for n-dimensional array. You type writing:

```
$ cdef np.ndarray array_in_question
```

5.3 Defining the datatype of array elements and its number of dimensions

For example, the datatype of the array elements is int. The numpy imported using cimport has a type corresponding to each type in NumPy but with **_t** at the end.

The argument ndim specifies the number of dimensions in the array. It is set to 1 here. Note that its default value is also 1, and thus can be omitted in this example. If more dimensions are being used, we must specify it.

```
$ cdef np.ndarray[np.int_t, ndim = 1] array_in_question
```

Notice that you are only permitted to define the type of the `NumPy` array in this way when it is an argument inside a function, or a local variable in the function - not inside the script body.

When you are using a `np.arange()`, for example, you can specify the type of its container, e.g. integers, using:

```
$ array_in_question = np.arange(max_val, dtype = np.int)
```

5.4 Code: Looping througnh a NumPy array

The simplest `Python` code, for sum about all entries of a given array, is given by:

```python
import time
t1 = time.time()

import numpy as np

def computing_total(n):
    total = 0
    arr = np.arange(n)
    for k in arr:
        total = total + k
    return total

n = 10**4

print(computing_total(n))

t2 = time.time()
t = t2 - t1
print('Total time: ', t)
```

In my personal computer it takes $\Delta t_{Python} \simeq 0.15124s$ to run.

Using all above tricks we can write the following .pyx and `setup` codes:

```
import numpy as np
cimport numpy as np

def computing_total(int n):
    cdef unsigned long long int total
    cdef int k
    cdef np.ndarray[np.int_t, ndim=1] arr
    total = 0
    arr = np.arange(n, dtype=np.int)
    for k in arr:
       total += k
    return total
```

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

setup(name='Computing total', ext_modules=cythonize("total_v3.pyx"))
```

putting those in the `Python` file:

```
import time
from total_v3 import computing_total

t1 = time.time()

n = 10**4
print("The total is", computing_total(n))

t2 = time.time()

print('Total time:', t2-t1)
```

In my personal computer it runs in $\Delta t_{np} \simeq 0.00084s$. In this way we have had almost 100% of sppedup.

In the same way, you can use the `NumPy array` as a function argument as follows:

```
import numpy as np
cimport numpy as np

ctypedef np.int_t DTYPE_t
def computing_total(np.ndarray[DTYPE_t, ndim=1] arr):
    cdef int n
    cdef unsigned long long int total
    cdef int k
    total = 0
    for k in arr:
       total += k
    return total
```

It runs in $\Delta t_{np} \simeq 0.00083s$, i.e., this "changes nothing" in matters of performance.

5.5 Indexing × Iterating over NumPy arrays

`Python` has a way of iterating over arrays:

The loop variable loops through the NumPy array, element by element from the array is fetched and then assigns that element to the loop variable:

```
$ for k in array_in_question:

   $ total += k
```

Looping through the array this way is a style introduced in `Python` but it is not the way that `C` uses for looping through an array.

The normal way for looping through an array for programming languages is to: *create indices starting from* 0 *(sometimes from* 1*) until reaching the last index in the array. Each index is used for indexing the array to return the corresponding element.*

Because `C` does not know how to loop through the array in the `Python` style, then the loop is executed in `Python` style and thus takes much time for being executed.

In order to overcome this issue, we need to create a loop in the normal style that uses indices for accessing the array elements:

1. **Shape:** create a new variable `arr_shape` to store the number of elements within the array;
2. **Use it:** fed the `range()` function to returns the indices for accessing the array elements.

The simple .pyx code is written as follows:

```
import numpy as np
cimport numpy as np

ctypedef np.int_t DTYPE_t

def computing_total(np.ndarray[DTYPE_t, ndim=1] arr):
    cdef int n
    cdef unsigned long long int total
    cdef int k
    cdef int arr_shape = arr.shape[0]
    total = 0
    for k in range(arr_shape):
       total += arr[k]
    return total
```

The running time of this code was $\Delta t_{Cython}^{loop} \simeq 7.5 \cdot 10^{-5} s$. Therefore, it gives the maximuum performance until now.

References

[1] Cython documentation.

[2] NumPy Array processing with Cython.