

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:

```

1 #Factorial of a given number
2 def factorial(n):
3     if n <= 1:
4         return 1
5     else:
6         return n*factorial(n - 1)

```

The following could be a corresponding `setup.py` script:

```

1 from distutils.core import setup
2 from Cython.Build import cythonize
3
4 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:

```

1 from factorial_program import factorial
2 import time
3
4 n = int(input('Given your number: '))
5 t1 = time.time()
6 print("The factorial of this number is", factorial(n))
7 t2 = time.time()
8 print("Total time", t2 - t1)

```

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

```

1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<time.h>
4
5 long double factorial(long double n){
6     if(n==0){
7         return(1);
8     }
9     return(n * factorial(n-1));
10 }
11
12 clock_t start, end;
13 double cpu_time_used;
14
15 int main(){
16     long double n;
17     printf("Given your number\n");
18     scanf("%Lf", &n);
19     start = clock();
20     printf("The factorial of this number is: %Lf\n", factorial(n));
21     end = clock();

```

```

22     cpu_time_used = ((double) (end - start))/CLOCKS_PER_SEC;
23     printf("Total time %f\n", cpu_time_used);
24     return(0);
25 }

```

```

1 import time
2
3 def factorial(n):
4     if n <= 1:
5         return 1
6     else:
7         return n*factorial(n - 1)
8
9 n = int(input('Given your number: '))
10 t1 = time.time()
11 print("The factorial of this number is", factorial(n))
12 t2 = time.time()
13 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: $\Delta t \approx 0.00030s$,
- Cython: $\Delta t \approx 0.00017s$,
- C: $\Delta t \approx 0.00013s$.

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 $\Delta t \approx 0.00278s$, and just passing it into `Cython`, that have runned in $\Delta t \approx 0.00172s$. we have around $\sim 40\%$ speedup.

3.1.1 integration_python.py

```
1 import time
2
3 def f(x):
4     return 1 + x + x**2
5
6 def integrate_f(a, b, N):
7     s = 0
8     dx = (b - a)/N
9     for i in range(N):
10         s += f(a + i*dx)
11     return s*dx
12
13 t1 = time.process_time()
14
15 a = 0
16 b = 1000
17 N = 10000
18
19 print(integrate_f(a, b, N))
20
21 t2 = time.process_time()
22
23 print('Total time:', t2-t1)
```

3.1.2 integration_cython.pyx

```
1 def f(x):
2     return 1 + x + x**2
3
4 def integrate_f(a, b, N):
5     s = 0
6     dx = (b - a)/N
7     for i in range(N):
8         s += f(a + i*dx)
9     return s*dx
```

3.1.3 setup-just_cython.py

```
1 from distutils.core import setup
2 from Cython.Build import cythonize
3
4 setup(name='Integration', ext_modules=cythonize("integration_cython.pyx"))
```

3.1.4 integration-just_cython.py

```
1 import time
2 from integration_cython import integrate_f
3
4 a = 0
5 b = 1000
6 N = 10000
7
```

```

8 | t1 = time.process_time()
9 |
10 | print(integrate_f(a, b, N))
11 |
12 | t2 = time.process_time()
13 |
14 | 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

```

1 | def f(double x):
2 |     return 1 + x + x**2
3 |
4 | def integrate_f(double a, double b, int N):
5 |     cdef int i
6 |     cdef double s, dx
7 |     s = 0
8 |     dx = (b - a)/N
9 |     for i in range(N):
10 |         s += f(a + i*dx)
11 |     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 $\Delta t_{Cython}^{var_typed} \approx 0.00049s$, i.e., a $\sim 80\%$ 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:

```

1 | cdef double f(double x):
2 |     return 1 + x + x**2
3 |
4 | def integrate_f(double a, double b, int N):
5 |     cdef int i
6 |     cdef double s, dx
7 |     s = 0
8 |     dx = (b - a)/N
9 |     for i in range(N):
10 |         s += f(a + i*dx)
11 |     return s*dx

```

The running time of this program is $\Delta t_{\text{Cython}}^{\text{func_typed}} \approx 4.2 \cdot 10^{-5} \text{ s}$. 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 through a NumPy array

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

```
1 import time
2 t1 = time.time()
3
4 import numpy as np
5
6 def computing_total(n):
7     total = 0
8     arr = np.arange(n)
9     for k in arr:
10         total = total + k
11     return total
12
13 n = 10**4
14
15 print(computing_total(n))
16
17 t2 = time.time()
18 t = t2 - t1
19 print('Total time: ', t)
```

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

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

```
1 import numpy as np
2 cimport numpy as np
3
4 def computing_total(int n):
5     cdef unsigned long long int total
6     cdef int k
7     cdef np.ndarray[np.int_t, ndim=1] arr
8     total = 0
9     arr = np.arange(n, dtype=np.int)
10    for k in arr:
11        total += k
12    return total
```

```

1 | from distutils.core import setup
2 | from Cython.Build import cythonize
3 |
4 | setup(name='Computing total', ext_modules=cythonize("total_v3.pyx"))

```

putting those in the Python file:

```

1 | import time
2 | from total_v3 import computing_total
3 |
4 | t1 = time.time()
5 |
6 | n = 10**4
7 | print("The total is", computing_total(n))
8 |
9 | t2 = time.time()
10 |
11 | print('Total time:', t2-t1)

```

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

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

```

1 | import numpy as np
2 | cimport numpy as np
3 |
4 | ctypedef np.int_t DTYPE_t
5 | def computing_total(np.ndarray[DTYPE_t, ndim=1] arr):
6 |     cdef int n
7 |     cdef unsigned long long int total
8 |     cdef int k
9 |     total = 0
10 |     for k in arr:
11 |         total += k
12 |     return total

```

It runs in $\Delta t_{np} \approx 0.00083s$, 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:

```
1 import numpy as np
2 cimport numpy as np
3
4 ctypedef np.int_t DTYPE_t
5
6 def computing_total(np.ndarray[DTYPE_t, ndim=1] arr):
7     cdef int n
8     cdef unsigned long long int total
9     cdef int k
10    cdef int arr_shape = arr.shape[0]
11    total = 0
12    for k in range(arr_shape):
13        total += arr[k]
14    return total
```

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

References

- [1] Cython documentation.
- [2] NumPy Array processing with Cython.