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.

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 just one: using distutils/setuptools. To build a Cython module using *distutils* you need first to write a .pyx file. Here I will code a fatorial program, that you can name as fatorial.pyx containing:

```python
#Fatorial of a given number
def fatorial(n):
    if n <= 1:
        return 1
    else:
        return n*fatorial(n - 1)
```

The following could be a corresponding setup.py script:

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

setup(name='Fatorial computation', ext_modules=cythonize("fatorial_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_fatorial_function.py, using the imported function as you see to fit:

```python
from fatorial_program import fatorial

n = int(input('Given your number: '))
print("The fatorial of this number is", fatorial(n))
```

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 and just passing it into Cython we have around ~ 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

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

```
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 a ~ 75% 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:

```
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
```

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).

References

[1] Cython documentation.