Basic Tutorial

The Basics of Cython

The fundamental nature of Cython can be summed up as follows: Cython is Python with C data types.

Cython is Python: Almost any piece of Python code is also valid Cython code. (There are a few *Limitations*, but this approximation will serve for now.) The Cython compiler will convert it into C code which makes equivalent calls to the Python/C API.

But Cython is much more than that, because parameters and variables can be declared to have C data types. Code which manipulates Python values and C values can be freely intermixed, with conversions occurring automatically wherever possible. Reference count maintenance and error checking of Python operations is also automatic, and the full power of Python's exception handling facilities, including the try-except and try-finally statements, is available to you – even in the midst of manipulating C data.

Cython Hello World

As Cython can accept almost any valid python source file, one of the hardest things in getting started is just figuring out how to compile your extension.

So lets start with the canonical python hello world:

```
print "Hello World"
```

Save this code in a file named `helloworld.pyx`. Now we need to create the `setup.py`, which is like a python Makefile (for more information see *Source Files and Compilation*). Your `setup.py` should look like:

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

setup(
    ext_modules = cythonize("helloworld.pyx")
)
```

To use this to build your Cython file use the commandline options:

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

Which will leave a file in your local directory called `helloworld.so` in unix or `helloworld.dll` in Windows. Now to use this file: start the python interpreter and simply import it as if it was a regular python module:

```
>>> import helloworld
Hello World
```

Congratulations! You now know how to build a Cython extension. But so far this example doesn't really give a feeling why one would ever want to use Cython, so lets create a more realistic example.

`pyximport`: Cython Compilation the Easy Way

If your module doesn't require any extra C libraries or a special build setup, then you can use the pyximport module by Paul Prescod and Stefan Behnel to load .pyx files directly on import, without having to write a `setup.py` file. It is shipped and installed with Cython and can be used like this:

```
>>> import pyximport; pyximport.install()
>>> import helloworld
Hello World
```

Since Cython 0.11, the **`pyximport`** module also has experimental compilation support for normal Python modules. This allows you to automatically run Cython on every .pyx and .py module that Python imports, including the standard library and installed packages. Cython will still fail to compile a lot of Python modules, in which case the import mechanism will fall back to loading the Python source modules instead. The .py import mechanism is installed like this:

```
>>> pyximport.install(pyimport = True)
```

Fibonacci Fun

From the official Python tutorial a simple fibonacci function is defined as:

```
def fib(n):
    """Print the Fibonacci series up to n."""
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a + b
```

Now following the steps for the Hello World example we first rename the file to have a *.pyx* extension, lets say `fib.pyx`, then we create the `setup.py` file. Using the file created for the Hello World example, all that you need to change is the name of the Cython filename, and the resulting module name, doing this we have:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
```

```
setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("fib", ["fib.pyx"])]
)
```

Build the extension with the same command used for the helloworld.pyx:

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

And use the new extension with:

```
>>> import fib
>>> fib.fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Primes

Here's a small example showing some of what can be done. It's a routine for finding prime numbers. You tell it how many primes you want, and it returns them as a Python list.

`primes.pyx`:

```
 1  def primes(int kmax):
 2      cdef int n, k, i
 3      cdef int p[1000]
 4      result = []
 5      if kmax > 1000:
 6          kmax = 1000
 7      k = 0
 8      n = 2
 9      while k < kmax:
10          i = 0
11          while i < k and n % p[i] != 0:
12              i = i + 1
13          if i == k:
14              p[k] = n
15              k = k + 1
16              result.append(n)
17          n = n + 1
18      return result
```

You'll see that it starts out just like a normal Python function definition, except that the parameter `kmax` is declared to be of type `int` . This means that the object passed will be converted to a C integer (or a `TypeError`. will be raised if it can't be).

Lines 2 and 3 use the `cdef` statement to define some local C variables. Line 4 creates a Python list which will be used to return the result. You'll notice that this is done exactly the same way it would be in Python. Because the variable result hasn't been given a type, it is assumed to hold a Python object.

Lines 7-9 set up for a loop which will test candidate numbers for primeness until the required number of primes has been found. Lines 11-12, which try dividing a candidate by all the primes found so far, are of particular interest. Because no Python objects are referred to, the loop is translated entirely into C code, and thus runs very fast.

When a prime is found, lines 14-15 add it to the p array for fast access by the testing loop, and line 16 adds it to the result list. Again, you'll notice that line 16 looks very much like a Python statement, and in fact it is, with the twist that the C parameter `n` is automatically converted to a Python object before being passed to the append method. Finally, at line 18, a normal Python return statement returns the result list.

Compiling primes.pyx with the Cython compiler produces an extension module which we can try out in the interactive interpreter as follows:

```
>>> import primes
>>> primes.primes(10)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

See, it works! And if you're curious about how much work Cython has saved you, take a look at the C code generated for this module.

Language Details

For more about the Cython language, see *Language Basics*. To dive right in to using Cython in a numerical computation context, see *Cython for NumPy users*.