05 - Introduction to Cython

Introduction to Cython

Cython is a programming language specially designed for writing Python extension modules. It's designed to bridge the gap between the nice, high-level, easy-to-use world of Python and the messy, low-level world of C.

A Python function

Consider the following Python function that outputs a list of the first `m` prime numbers. (This is not exactly a Cython example, but it will be useful for comparisons.)

```
def first_primes_python(m):
   primes_list = []
   n = 2
   while len(primes_list) < m:
       n_is_prime = True
       for p in primes_list:
           if n % p == 0:
               n_is_prime = False
               break
       if n_is_prime == True:
           primes_list.append(n)
       n = n + 1
   return primes_list
```

To time a function in Python, use the `time` command.

```
time p = first_primes_python(5000)
```

```
    CPU time: 6.47 s,  Wall time: 6.52 s
```

```

```

First steps with Cython

To *Cythonize* a function, just add `%cython` as the first line in the notebook

cell.

The Sage notebook will take the contents of this cell, convert it to Cython, compile it, and load the resulting function.

```
%cython
def first_primes_cython_v1(m):
   primes_list = []
   n = 2
   while len(primes_list) < m:
       n_is_prime = True
       for p in primes_list:
           if n % p == 0:
               n_is_prime = False
               break
       if n_is_prime == True:
           primes_list.append(n)
       n = n + 1
   return primes_list
```

__home_sal...19_code_sage5_spyx.c __home_sal...code_sage5_spyx.html

Note the speed up we obtained by just adding `%cython`:

```
time p = first_primes_cython_v1(5000)
```

```
CPU time: 0.89 s,  Wall time: 0.90 s
```

```
time p = first_primes_cython_v1(10000)
```

```
CPU time: 3.19 s,  Wall time: 3.23 s
```

```
```

More Cython

Note that two links were returned above. The first one is a link to the C source code file created by Cython from our function. Go take a look. The conversion is a nontrivial process.

The second link above is an html page that identifies Python-to-C and C-to-Python conversions that are taking place. By minimizing such conversions and declaring data types, we can further improve the speed of our function.

Below, some object type declarations are made, we simplify some of the

loops and we use a C array instead of the Python list `primes_list`. But since we want to return the data as a Python list, we convert to a Python list at the end.

```
%cython
def first_primes_v3(int m):
   cdef int c_array[100000]
   cdef int k = 0
   cdef int n = 2
   cdef int i, n_is_prime
   while k < m:
       n_is_prime = 0
       i = 0
       while i < k:
           if n % c_array[i] == 0:
               n_is_prime = 1
               break
           i = i + 1
       if n_is_prime == 0:
           c_array[k] = n
           k = k+1
       n = n + 1
   primes_list = []
   i = 0
   while i < k:
       primes_list.append(c_array[i])
       i = i+1
   return primes_list
```

__home_sal...9_code_sage10_spyx.c __home_sal...ode_sage10_spyx.html

```
time p = first_primes_v3(10000)
```

```
CPU time: 0.24 s,  Wall time: 0.25 s
```

We didn't screw up anything, this function actually does produce primes:

```
first_primes_v3(17)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59]
```

And it agrees with the Sage version of the function:

```
first_primes_v3(10000) == primes_first_n(10000)
```

```
True
```

But the Sage version is much, much better:

```
time p = primes_first_n(10000)
```

```
CPU time: 0.01 s,  Wall time: 0.01 s
```

```
```