Cython: A First Look

Some Jupyter lab notes:

- Jupyter lab let's us make cells and run code in a nicely formatted way
- We also can use things like magic cells - these allow us to do special operations on code
- Rerunning cells is super easy
- For Cython - the notebook abstracts all of the compilation away
- Also for Cython - allows you to profile your code

Typical sieve algorithm:

1. Create a list of integers 2 -> N
2. Start at 2, all factors of it are marked in the list as non-prime (false)
3. Go to next true index
4. Mark all factors of it in the list as false
5. Go to step 3
6. All remaining true indices are prime numbers

Here's a basic sieve implementation. Nothing special.

Might not even be the most efficient!

In [1]:
```
def sieve(sieve_length):
    sieve_table = [True for x in range(sieve_length)]
    sieve_table[0] = False
    sieve_table[1] = False

    for i in range(2,int(sieve_length**0.5)+1):
        if sieve_table[i]:
            for marker in range(i*i, sieve_length, i):
                sieve_table[marker] = False

    return [i for i, t in enumerate(sieve_table) if t]
```

Testing base functionality:

In [2]:
```
primes = sieve(1_000)
print(','.join([str(p) for p in primes]))
```

```
2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,
107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197,199,21
1,223,227,229,233,239,241,251,257,263,269,271,277,281,283,293,307,311,313,317,3
31,337,347,349,353,359,367,373,379,383,389,397,401,409,419,421,431,433,439,443,
449,457,461,463,467,479,487,491,499,503,509,521,523,541,547,557,563,569,571,57
7,587,593,599,601,607,613,617,619,631,641,643,647,653,659,661,673,677,683,691,7
01,709,719,727,733,739,743,751,757,761,769,773,787,797,809,811,821,823,827,829,
839,853,857,859,863,877,881,883,887,907,911,919,929,937,941,947,953,967,971,97
7,983,991,997
```

Everything appears to be working, but how fast is it?

Time for some basic benchmarking!

In [3]:
```
%timeit sieve(1_000_000)
```

```
126 ms ± 6.28 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [4]:
```
%timeit sieve(10_000_000)
```

```
1.47 s ± 29.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Anecdotally - I happen to know this is pretty slow.

First steps into Cython

In [5]:
```
%load_ext Cython
```

In [6]:
```
%%cython

def sieve_magic(sieve_length):
    sieve_table = [True for x in range(sieve_length)]
    sieve_table[0] = False
    sieve_table[1] = False

    for i in range(2,int(sieve_length**0.5)+1):
        if sieve_table[i]:
            for marker in range(i*i, sieve_length, i):
                sieve_table[marker] = False

    return [i for i, t in enumerate(sieve_table) if t]
```

In [7]:
```
primes_magic = sieve_magic(1_000)
print(','.join([str(p) for p in primes_magic]))
```

```
2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,
107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197,199,21
1,223,227,229,233,239,241,251,257,263,269,271,277,281,283,293,307,311,313,317,3
31,337,347,349,353,359,367,373,379,383,389,397,401,409,419,421,431,433,439,443,
449,457,461,463,467,479,487,491,499,503,509,521,523,541,547,557,563,569,571,57
7,587,593,599,601,607,613,617,619,631,641,643,647,653,659,661,673,677,683,691,7
01,709,719,727,733,739,743,751,757,761,769,773,787,797,809,811,821,823,827,829,
839,853,857,859,863,877,881,883,887,907,911,919,929,937,941,947,953,967,971,97
7,983,991,997
```

In [8]:
```
%timeit sieve_magic(1_000_000)
```

```
72.8 ms ± 2.66 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Exploring with Cython

Cython gives us the ability to view how our code has compiled!

Let's try it:

In [9]:
```
%%cython

def sieve_working(int sieve_length):
    sieve_table = [True for x in range(sieve_length)]
    sieve_table[0] = False
    sieve_table[1] = False

    cdef int i, marker
    cdef int upper
    upper = int(sieve_length**0.5) + 1

    for i in range(2,int(sieve_length**0.5)+1):
        if sieve_table[i]:
            for marker in range(i*i, sieve_length, i):
                sieve_table[marker] = False

    return [i for i, t in enumerate(sieve_table) if t]
```

In [10]:
```
%timeit sieve_working(1_000_000)
```

```
46 ms ± 468 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

We've still got quite a bit of yellow - but things are faster for sure!

Splitting things up

It looks like working on these list comprehensions is going to be a struggle... Let's split some things up.

reference

In [11]:
```
%%cython
import cython

def sieve_table_cy(int sieve_length):
    sieve_table = [True for x in range(sieve_length)]
    sieve_table[0] = False
    sieve_table[1] = False

    cdef int i, marker
    cdef int upper
    upper = int(sieve_length**0.5) + 1

    for i in range(2, upper):
        if sieve_table[i]:
            for marker in range(i*i, sieve_length, i):
                sieve_table[marker] = False

    return sieve_table

def sieve_print_cy(table):
    cdef int i
    cdef int t
    cdef list primes
    primes = []
    for i in range(len(table)):
        if table[i]:
            primes.append(i)
    return primes
```

In [12]:
```
%%timeit
table = sieve_table_cy(1_000_000)
# table = sieve_table_cy(1_000)
prime_list = sieve_print_cy(table)
```

```
50.7 ms ± 2.42 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [13]:
```
%%timeit
table = sieve_table_cy(50_000_000)
# table = sieve_table_cy(1_000)
prime_list = sieve_print_cy(table)
```

```
3.9 s ± 85.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In the comparison case: not really faster - but we can see what needs to be done much better

Calling STL Functions

At this point we know that there's more we can do with that inner for loop - but let's have a look at the list access that's being done.

Why don't we replace it with a C++ structure?

In [14]:
```
%reload_ext Cython
```

reference

In [15]:
```
%%cython
# distutils: language=c++

import cython

from libcpp.vector cimport vector

def do_stuff():
    cdef vector[int] totally_a_list
    totally_a_list.push_back(100)
    return totally_a_list[0]
```

In [16]:
```
do_stuff()
```

Out[16]:
```
100
```

that was easy! Let's rewrite our previous code now.

In [17]:
```
%%cython
# distutils: language=c++

import cython

from libcpp.vector cimport vector

def sieve_table_vec(int sieve_length):
    cdef vector[int] sieve_table
    sieve_table.resize(sieve_length, 1)
    sieve_table[0] = 0
    sieve_table[1] = 0

    cdef int i, marker
    cdef int upper
    upper = int(sieve_length**0.5) + 1
    for i in range(2, upper):
        if sieve_table[i]:
            for marker in range(i*i, sieve_length, i):
                sieve_table[marker] = 0

    return sieve_table

def sieve_print_vec(table):
    cdef int i
    cdef vector[int] primes
    for i in range(len(table)):
        if table[i]:
            primes.push_back(i)
    return primes
```

In [18]:
```
%%timeit
table = sieve_table_vec(1_000_000)
# table = sieve_table_vec(1_000)
prime_list = sieve_print_vec(table)
```

```
39.1 ms ± 789 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [19]:
```
%%timeit
table = sieve_table_vec(50_000_000)
# table = sieve_table_vec(1_000)
prime_list = sieve_print_vec(table)
```

```
2.32 s ± 17 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Battling the Inner Loop

There's other smaller optimizations to do for sure - but what about that inner for loop?

In [20]:
```
%%cython
# distutils: language=c++

import cython

from libcpp.vector cimport vector

def sieve_table_fin(int sieve_length):
    cdef vector[int] sieve_table
    sieve_table.resize(sieve_length, 1)
    sieve_table[0] = 0
    sieve_table[1] = 0

    cdef int i, marker
    cdef int upper
    upper = int(sieve_length**0.5) + 1
    for i in range(2, upper):
        if sieve_table[i]:
            marker = i*i
            while marker < sieve_length:
                sieve_table[marker] = 0
                marker += i

    return sieve_table

def sieve_print_fin(table):
    cdef int i
    cdef vector[int] primes
    for i in range(len(table)):
        if table[i]:
            primes.push_back(i)
    return primes
```

In [21]:
```
%%timeit
table = sieve_table_fin(1_000_000)
# table = sieve_table_vec(1_000)
prime_list = sieve_print_fin(table)
```

```
15.1 ms ± 502 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [22]:
```
%%timeit
table = sieve_table_fin(50_000_000)
# table = sieve_table_vec(1_000)
prime_list = sieve_print_fin(table)
```

```
1.31 s ± 16.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```