reference

October 19, 2022

1 Cython: A First Look

1.1 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

1.2 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!

[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:

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

```
2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,1
07,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197,199,211,2
23,227,229,233,239,241,251,257,263,269,271,277,281,283,293,307,311,313,317,331,3
37,347,349,353,359,367,373,379,383,389,397,401,409,419,421,431,433,439,443,449,4
57,461,463,467,479,487,491,499,503,509,521,523,541,547,557,563,569,571,577,587,5
93,599,601,607,613,617,619,631,641,643,647,653,659,661,673,677,683,691,701,709,7
19,727,733,739,743,751,757,761,769,773,787,797,809,811,821,823,827,829,839,853,8
57,859,863,877,881,883,887,907,911,919,929,937,941,947,953,967,971,977,983,991,9
97
```

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

Time for some basic benchmarking!

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

```
156 ms ± 25.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

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

```
2 s ± 81 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Anecdotally - I happen to know this is pretty slow.

1.3 First steps into Cython

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

[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]
```

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

```
2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,1
07,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197,199,211,2
23,227,229,233,239,241,251,257,263,269,271,277,281,283,293,307,311,313,317,331,3
37,347,349,353,359,367,373,379,383,389,397,401,409,419,421,431,433,439,443,449,4
57,461,463,467,479,487,491,499,503,509,521,523,541,547,557,563,569,571,577,587,5
```

```
93,599,601,607,613,617,619,631,641,643,647,653,659,661,673,677,683,691,701,709,7
19,727,733,739,743,751,757,761,769,773,787,797,809,811,821,823,827,829,839,853,8
57,859,863,877,881,883,887,907,911,919,929,937,941,947,953,967,971,977,983,991,9
97
```

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

```
72.9 ms ± 2.44 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

2 Exploring with Cython

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

Let's try it:

[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]
```

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

```
40.6 ms ± 307 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!

3 Splitting things up

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

[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
```

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

```
49.7 ms ± 6.3 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

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

```
6.24 s ± 120 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

4 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?

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

[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]
```

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

[16]: 100

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

[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
```

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

```
37.7 ms ± 642 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

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

```
2.38 s ± 97 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

5 Battling the Inner Loop

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

[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
```

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

```
17.5 ms ± 2.6 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

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

```
1.5 s ± 125 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

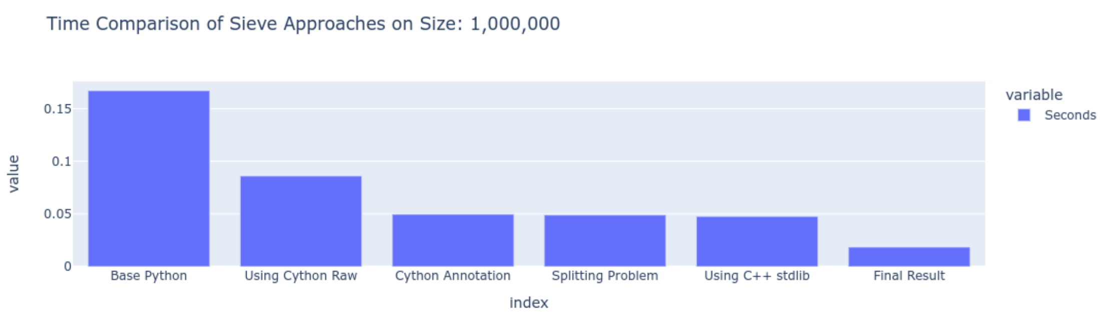
6 Quick Graphical Comparison

[23]:
```
import plotly.express as px
import pandas as pd

def compare_approach(problem_size):
    result_base = %timeit -o sieve(problem_size)
    result_raw = %timeit -o sieve_magic(problem_size) # raw compile
    result_annotate = %timeit -o sieve_working(problem_size) # annotation of␣
 ↪types
    result_split = %timeit -o sieve_print_cy(sieve_table_cy(problem_size)) #␣
 ↪problem split up
    result_stdlib = %timeit -o prime_list =␣
 ↪sieve_print_vec(sieve_table_vec(problem_size)) # use stlib
    result_final = %timeit -o table =␣
 ↪sieve_print_fin(sieve_table_fin(problem_size)) # final win

    result_dict = {
        'Base Python': result_base.best,
        'Using Cython Raw': result_raw.best,
        'Cython Annotation': result_annotate.best,
        'Splitting Problem': result_split.best,
        'Using C++ stdlib': result_stdlib.best,
        'Final Result': result_final.best
```

```
    }

    fig = px.bar(pd.DataFrame.from_dict(result_dict, orient='index',␣
↪columns=['Seconds']),
                 title=f'Time Comparison of Sieve Approaches on Size:␣
↪{problem_size:,}')
    fig.show()
```

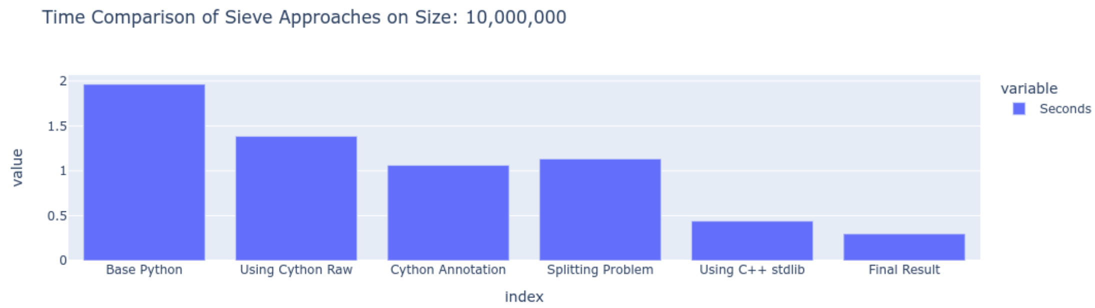
[24]:
```
compare_approach(1_000_000)
```

```
180 ms ± 11.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
90.6 ms ± 4.51 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
53 ms ± 3.14 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
49.4 ms ± 445 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
47.8 ms ± 239 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
18.8 ms ± 157 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```



[25]:
```
compare_approach(10_000_000)
```

```
2.01 s ± 37.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.55 s ± 234 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.13 s ± 49.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.26 s ± 90.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
544 ms ± 69.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
311 ms ± 12.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```



[26]:
```
compare_approach(50_000_000)
```

```
11.4 s ± 383 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
7.67 s ± 129 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
6.06 s ± 89.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
6.39 s ± 60 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
2.49 s ± 78.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.39 s ± 44.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```