IT Best Kept Secret Is Optimization

How To Make Python Run As Fast As Julia

JeanFrancoisPuget | Dec 1 2015 | Comments (30) | Visits (196028)

Julia vs Python

Should we ditch Python and other languages in favor of Julia for technical computing? That's certainly a thought that comes to mind when one looks at the benchmarks on http://julialang.org/. Python and other high level languages are way behind in term of speed. The first question that came to my mind was different however: did the Julia team wrote Python benchmarks the best way for Python?

My take on this kind of cross language comparison is that the benchmarks should be defined by tasks to perform, then have language experts write the best code they can to perform these tasks. If the code is all written by one language team, then there is a risk that other languages aren't used at best.

One thing the Julia team did right is to publish on github the code they used. In particular, the Python code can be found here.

A first look at this code confirms the bias I was afraid of. The code is written in a C style with heavy use of loops over arrays and lists. This is not the best way to use Python.

I won't blame the Julia team, as I have been guilty of the exact same bias. But I learned the hard lesson: loops on arrays or lists should be avoided at almost any cost as they are really slow in Python, see Python is not C.

Given this bias towards C style, the interesting question (to me at least) is whether we can improve these benchmarks with a better use of Python and its tools?

Before I give the answer below, let me say that I am in no way trying to downplay Julia. It is certainly a language worth monitoring as it is further developed and improved. I just want to have a look at the Python side of things. Actually, I am using this as an excuse to explore various Python tools that can be used to make code run faster.

In what follows I use Python 3.5.1 with Anaconda on a Windows machine. The notebook containing the complete code for all benchmarks below is available on github and on nbviewer.

Comments on various social media make me add this: I am not writing any C code here: if you're not convinced, then try to find any semicolon. All the tools used in this blog run in the standard CPython implementation available in Anaconda or other distributions. All the code below runs in a single notebook. I tried to use the Julia micro performance file from github but it does not run as is with Julia 0.4.2. I had to edit it and replace @timeit by @time to make it run. I also had to add calls to the timed functions before timing them, otherwise the compilation time was included. I ran it with the Julia command line interface, on the same machine as the one used to run Python.

Timing Code

The first benchmark Julia team used is a naive coding of the Fibonacci function.

```
def fib(n):
    if n<2:
        return n
    return fib(n-1)+fib(n-2)
```

This function grows rapidly with n, for instance:

```
fib(100) = 354224848179261915075
```

Note how Python arbitrary precision comes in handy. Coding the same in a language like C would some coding effort to avoid integer overflow. In Julia one would have to use the BigInt type.

All the Julia benchmarks are about running times. Here are the timings with and without BigInt in Julia

```
0.000080 seconds (149 allocations: 10.167 KB)
0.012717 seconds (262.69 k allocations: 4.342 MB)
```

One way to get running times in Python notebooks is to use the magic `%timeit`. For instance, typing

About this blog

Musing about Analytics, Optimization, Data Science, and Machine Learning Leverages Python and Mathematical Optimization. I am now publishing my code (esp notebooks) on git hub at: https://github.com/jfpuget/ My Views are my own.

Related posts

IBM SPSS Modeler 18....
Updated Aug 22 0 0

Checking if an Attri...
Updated Aug 22 0 0

Creating a JSON Stri...
Updated June 21 0 0

Maximo Rest Client E...
Updated June 2 0 0

Fetching Daily Excha...
Updated June 2 0 0

Links

My github repository
@JFPuget on twitter
Optimization Community on Deve...
Free CPLEX Trials
Free Cloud trial
Free Software For Academics An...
Support Forum on developerWork...
CPLEX and OPL products
IBM Decision Optimization Cent...
LinkedIn profile for jfpuget
Michael's Trick Operations Res...
OR in an OB World
Open Courses on Operations Res...
Punk Rock OR

Tags

```
%timeit fib(20)
```

in a new cell and executing it outputs

```
100 loops, best of 3: 3.77 ms per loop
```

It means that the timer did the following:

1. Run `fib(20)` one hundred times, store the total running time
2. Run `fib(20)` one hundred times, store the total running time
3. Run `fib(20)` one hundred times, store the total running time
4. Get the smallest running time from the three runs, divide it by 100, and outputs the result as the best running time for `fib(20)`

The sizes of loops (100 and 3 here) are automatically adjusted by the timer. They may change depending on how fast the timed code runs.

Python timing compares very favorably against Julia timing when BigInt are used: 3 milliseconds vs 12 milliseconds. Python is 4 times faster than Julia when arbitrary precision is used.

However, Python is way slower than Julia's default 64 bits integers. Let's see how we can force the use of 64 bits integers in Python.

Compiling With Cython

One way to do it is to use the Cython compiler. This compiler is written in Python. It can be installed via

```
pip install Cython
```

If you use Anaconda, the installation is different. As it is a bit tricky I wrote a blog entry about it: Installing Cython For Anaconda On Windows

Once installed, we load Cython in the notebook with the `%load_ext` magic:

```
%load_ext Cython
```

We can then compile code in our notebook. All we have to do is to put all the code we want to compile in one cell, including the required import statements, and start that cell with the cell magic `%%cython`:

```
%%cython
def fib_cython(n):
    if n<2:
        return n
    return fib_cython(n-1)+fib_cython(n-2)
```

Executing that cell compiles the code seamlessly. We use a slightly different name for our function to reflect that it is compiled with Cython. Of course, there is no need to do this in general. We could have replaced the previous function by a compiled function of the same name.

Timing it yields

```
1000 loops, best of 3: 1.47 ms per loop
```

Wow, more than 2 times faster than the original Python code! We are now 9 times faster than Julia with BigInt.

We can also try with static typing. We declare the function with the keyword `cpdef` instead of `def`. It allows us to type the parameters of the function with their corresponding C types. Our code becomes.

```
%%cython


cpdef long fib_cython_type(long n):
    if n<2:
        return n
    return fib_cython_type(n-1)+fib_cython_type(n-2)
```

After executing that cell, timing it yields

```
10000 loops, best of 3: 24.4 µs per loop
```

Amazing, we're now at 24 micro seconds, about 150 times faster than the original benchmark ! This compares favorably with the 80 microseconds used by Julia.

One can argue that static typing defeats the purpose of Python. I kind of agree with that in general, and we will see later a way to avoid this without sacrificing performance. But I don't think this is the issue here. The Fibonacci function is meant to be called with integers. What we lose with static typing is the arbitrary precision that Python provides. In the case of Fibonacci, using the C type `long` limits the size of the input parameter

because too large parameters would result in integer overflow.

Note that Julia computation is done with 64 bits integers too, hence comparing our statically typed version with that of Julia is fair.

Caching Computation

We can do better while keeping Python arbitrary precision. The `fib` function repeats the same computation many times. For instance, `fib(20)` will call `fib(19)` and `fib(18)`. In turn, `fib(19)` will call `fib(18)` and `fib(17)`. As a result `fib(18)` will be called twice. A little analysis shows that `fib(17)` will be called 3 times, and `fib(16)` five times, etc.

In Python 3, we can avoid these repeated computations using the functools standard library.

```
from functools import lru_cache as cache

@cache(maxsize=None)
def fib_cache(n):
    if n<2:
        return n
    return fib_cache(n-1)+fib_cache(n-2)
```

Timing this function yields:

```
1000000 loops, best of 3: 127 ns per loop
```

This is an additional 190 times speedup, and about 30,000 times faster than the original ¨Python code! I find it impressive given we merely add an annotation to the recursive function.

Note that Julia also can memoize functions, see this example provided by Ismael V.C.

This automated cache isn't available with Python 2.7. We need to transform the code explicitly in order to avoid duplicate computation in that case.

```
def fib_seq(n):
    if n < 2:
        return n
    a,b = 1,0
    for i in range(n-1):
        a,b = a+b,a
    return a
```

Note that this code makes use of Python ability to simultaneously assign two local variables. Timing it yields

```
1000000 loops, best of 3: 1.81 µs per loop
```

Another 20 times speedup! Let us compile our function, with and without static typing. Note how we use the `cdef` keyword to type local variables.

```
%%cython

def fib_seq_cython(n):
    if n < 2:
        return n
    a,b = 1,0
    for i in range(n-1):
        a,b = a+b,a
    return a

cpdef long fib_seq_cython_type(long n):
    if n < 2:
        return n
    cdef long a,b
    a,b = 1,0
    for i in range(n-1):
        a,b = a+b,a
    return a
```

We can time both versions in one cell:

```
%timeit fib_seq_cython(20)
%timeit fib_seq_cython_type(20)
```

It yields

```
1000000 loops, best of 3: 953 ns per loop
10000000 loops, best of 3: 82 ns per loop
```

We are now at 82 nano seconds with the statically typed code, about *45,000* times faster than the original benchmark!

If we want to compute the Fibonacci number for arbitrary input, then we should stick to the untyped version, which runs with a respectable *30,000* times speedup. Not so bad isn't it?

Compiling With Numba

Let us use another tool called Numba. It is a just in time (jit) compiler for a subset of Python. It does not work yet on all of Python, but when it does work it can do marvels.

Installing it can be cumbersome. I recommend that you use a Python distribution like Anaconda or a Docker image where Numba is already installed. Once installed, we import its jit compiler:

```
from numba import jit
```

Using it is very simple. We only need to add a decoration to the functions we want to compile. Our code becomes:

```
@jit
def fib_seq_numba(n):
    if n < 2:
        return n
    a,b = 1,0
    for i in range(n-1):
        a,b = a+b,a
    return a
```

Timing it yields

```
1000000 loops, best of 3: 216 ns per loop
```

We are faster than the untyped Cython code, and about *17,000* times faster than the original Python code!

Using Numpy

Let's now look at the second benchmark. It is an implementation of the quicksort algorithm. Julia team used that Python code:

```
def qsort_kernel(a, lo, hi):
    i = lo
    j = hi
    while i < hi:
        pivot = a[(lo+hi) // 2]
        while i <= j:
            while a[i] < pivot:
                i += 1
            while a[j] > pivot:
                j -= 1
            if i <= j:
                a[i], a[j] = a[j], a[i]
                i += 1
                j -= 1
        if lo < j:
            qsort_kernel(a, lo, j)
        lo = i
        j = hi
    return a
```

I wrapped their benchmarking code in a function:

```
import random


def benchmark_qsort():
    lst = [ random.random() for i in range(1,5000) ]
    qsort_kernel(lst, 0, len(lst)-1)
```

Timing it yields:

```
100 loops, best of 3: 17.7 ms per loop
```

The above code is really like C code. Cython should do well on it. Besides using Cython and static typing, let us use Numpy arrays instead of lists. Indeed, Numpy arrays are faster than Python lists when their size is large, say thousands of elements or more.

Installing Numpy can take a while, I recommend you use Anaconda or a Docker image where the Python scientific stack is already installed.

When using Cython, we need to import Numpy in the cell to which Cython is applied. Numpy arrays are declared with a special syntax indicating the type of elements of arrays, and the number of dimensions of the array (1D, 2D, etc). The decorators tell Cython to remove bound checking.

```
%%cython
import numpy as np
import cython

@cython.boundscheck(False)
@cython.wraparound(False)
cdef double[:] \
qsort_kernel_cython_numpy_type(double[:] a, \
                               long lo, \
                               long hi):
    cdef:
        long i, j
        double pivot
    i = lo
    j = hi
    while i < hi:
        pivot = a[(lo+hi) // 2]
        while i <= j:
            while a[i] < pivot:
                i += 1
            while a[j] > pivot:
                j -= 1
            if i <= j:
                a[i], a[j] = a[j], a[i]
                i += 1
                j -= 1
        if lo < j:
            qsort_kernel_cython_numpy_type(a, lo, j)
        lo = i
        j = hi
    return a

def benchmark_qsort_numpy_cython():
    lst = np.random.rand(5000)
    qsort_kernel_cython_numpy_type(lst, 0, len(lst)-1)
```

Timing the `benchmark_qsort_numpy_cython()` function yields

```
1000 loops, best of 3: 772 µs per loop
```

We are about 23 times faster than the original benchmark, but this is still not the best way to use Python. The best way is to use the Numpy built-in `sort()` function. Its default behavior is to use the quick sort algorithm. Timing this code

```
def benchmark_sort_numpy():
    lst = np.random.rand(5000)
    np.sort(lst)
```

yields

```
1000 loops, best of 3: 306 µs per loop
```

We are now 58 times faster than the original benchmark! Julia takes 419 micro seconds on that benchmark, hence compiled Python is 40% faster.

I know, some readers will say that I am not comparing apple to apple. I disagree. Remember, the task at hand is to sort an input array using the host language in the best possible way. In this case, the best possible way is to use a built-in function.

Profiling Code

Let use now look at a third example, computing the Mandelbrot set. Julia team used this Python code:

```
def mandel(z):
    maxiter = 80
    c = z
    for n in range(maxiter):
        if abs(z) > 2:
            return n
        z = z*z + c
    return maxiter

def mandelperf():
    r1 = np.linspace(-2.0, 0.5, 26)
    r2 = np.linspace(-1.0, 1.0, 21)
    return [mandel(complex(r, i)) for r in r1 for i in r2]


assert sum(mandelperf()) == 14791
```

The last line is a sanity check. Timing the `mandelperf()` function yields:

```
100 loops, best of 3: 6.57 ms per loop
```

Using Cython yields:

```
100 loops, best of 3: 3.6 ms per loop
```

Not bad, but we can do better using Numba. Unfortunately, Numba does not compile list comprehensions yet. Therefore we cannot apply it to the second function, but we can apply it to the first one. Our code looks like this.

```
@jit
def mandel_numba(z):
    maxiter = 80
    c = z
    for n in range(maxiter):
        if abs(z) > 2:
            return n
        z = z*z + c
    return maxiter

def mandelperf_numba():
    r1 = np.linspace(-2.0, 0.5, 26)
    r2 = np.linspace(-1.0, 1.0, 21)
    return [mandel(complex(r, i)) for r in r1 for i in r2]
```

Timing it yields

```
1000 loops, best of 3: 481 µs per loop
```

Not bad, four times faster than Cython, and 9 times faster than the original Python code!

Can we do more? One way to know is to profile the code. The built-in `%prun` profiler is not precise enough here, and we must use a better profiler known as line_profiler. It can be installed via pip:

```
pip install line_profiler
```

Once installed, we load it:

```
%load_ext line_profiler
```

We can then profile the function using a magic:

```
%lprun -s -f mandelperf_numba mandelperf_numba()
```

It outputs the following in a pop up window.

```
Timer unit: 1e-06 s

Total time: 0.003666 s
File: <ipython-input-102-e6043a6167d6>
Function: mandelperf_numba at line 11

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    11                                           def mandelperf_numba():
    12         1         1994   1994.0     54.4      r1 = np.linspace(-2.0,
0.5, 26)
    13         1          267    267.0      7.3      r2 = np.linspace(-1.0,
1.0, 21)
    14         1         1405   1405.0     38.3      return [mandel_numba(co
mplex(r, i)) for r in r1 for i in r2]
```

We see that the bulk of the time is spent in the first and last lines of our `mandelperf_numba()` function. The last line is a bit complex, let us break it into two pieces, and profile again:

```
def mandelperf_numba():
    r1 = np.linspace(-2.0, 0.5, 26)
    r2 = np.linspace(-1.0, 1.0, 21)
    c3 = [complex(r,i) for r in r1 for i in r2]
    return [mandel_numba(c) for c in c3]
```

Profiler output becomes

```
Timer unit: 1e-06 s

Total time: 0.002002 s
File: <ipython-input-113-ba7b044b2c6c>
Function: mandelperf_numba at line 11

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    11                                           def mandelperf_numba():
    12         1          678    678.0     33.9      r1 = np.linspace(-2.0,
0.5, 26)
    13         1          235    235.0     11.7      r2 = np.linspace(-1.0,
1.0, 21)
    14         1          617    617.0     30.8      c3 = [complex(r, i) for r
in r1 for i in r2]
    15         1          472    472.0     23.6      return [mandel_numba(c)
for c in c3]
```

We see that the calls to the function `mandel_numba()` takes only one fourth of the total time. The rest of the time is spent in the `mandelperf_numba()` function. Optimizing it is worthwhile.

Using Numpy Again

Using Cython isn't helping much here, and Numba does not apply. One way out of this dilemma is to use Numpy again. We will replace the following by Numpy code that produces an equivalent result.

```
return [mandel_numba(complex(r, i)) for r in r1 for i in r2]
```

This code is building what is known as a 2D mesh. It computes the complex number representation of points whose coordinates are given by `r1` and `r2`. Point P_{ij} coordinates are `r1[i]` and `r2[j]`. P_{ij} is represented by the complex number `r1[i] + 1j*r2[j]` where the special constant `1j` represents the unitary imaginary number *i*.

We can code this computation directly:

```
@jit
def mandelperf_numba_mesh():
    width = 26
    height = 21
    r1 = np.linspace(-2.0, 0.5, width)
    r2 = np.linspace(-1.0, 1.0, height)
    mandel_set = np.empty((width,height), dtype=int)
    for i in range(width):
        for j in range(height):
            mandel_set[i,j] = mandel_numba(r1[i] + 1j*r2[j])
    return mandel_set
```

Note that I changed the return value to be a 2D array of integers. That's closer to what we need if we were to display the result.

Timing it yields

```
10000 loops, best of 3: 126 µs per loop
```

We are about 50 times faster than the original Python code! Julia takes 196 micro seconds on that benchmark, hence compiled Python is 60% faster.

[Edited on February 2, 2016]. We can do even better in Python, see How To Quickly Compute Mandelbrot Set In Python.

Vectorizing

Let us look at another example. I am not sure about what is measured to be honest, but here is the code the Julia team used .

```
def parse_int():
    for i in range(1,1000):
        n = random.randint(0,2**32-1)
        s = hex(n)
        m = int(s,16)
        assert m == n
```

Actually Julia's team code has an extra instruction that strips the ending `'L'` in case it is present. That line is required for my Anaconda install, but it is not required for my Python 3 install, therefore I removed it. The original code is:

```
def parse_int():
    for i in range(1,1000):
        n = random.randint(0,2**32-1)
        s = hex(n)
        if s[-1]=='L':
            s = s[0:-1]
        m = int(s,16)
        assert m == n
```

Timing the modified code yields:

```
100 loops, best of 3: 3.29 ms per loop
```

Neither Numba nor Cython seem to help.

As I was puzzled by this benchmark, I profiled the original code. Here is the result:

```
Timer unit: 1e-06 s

Total time: 0.013807 s
File: <ipython-input-3-1d995505b176>
Function: parse_int at line 1
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def parse_int():
     2      1000          699      0.7      5.1      for i in range(1,1000):
     3       999         9149      9.2     66.3          n = random.randint(0
,2**32-1)
     4       999         1024      1.0      7.4          s = hex(n)
     5       999          863      0.9      6.3          if s[-1]=='L':
     6                                                       s = s[0:-1]
     7       999         1334      1.3      9.7          m = int(s,16)
     8       999          738      0.7      5.3          assert m == n
```

We see that most of the time is in generating the random numbers. I am not sure this was the intent of the benchmark...

One way to speed this is to use the Numpy random generator. Given Numpy uses C ints, we must limit the largest value to 2^31 - 1. The code becomes:

```
def parse_int1_numpy():
    for i in range(1,1000):
        n = np.random.randint(0,2**31-1)
        s = hex(n)
        m = int(s,16)
        assert m == n
```

Timing it yields

```
1000 loops, best of 3: 1.05 ms per loop
```

Not bad, more than 3 times faster! Appmying Numba doe snot help, but cython provides some further improvement:

```
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef parse_int_cython_numpy():
    cdef:
        int i,n,m
    for i in range(1,1000):
        n = np.random.randint(0,2**31-1)
        m = int(hex(n),16)
        assert m == n
```

Fiming it yields:

```
1000 loops, best of 3: 817 µs per loop
```

One way to further speed this up is to move the generation of random numbers out of the loop using Numpy. We create an array of random numbers in one step.

```
def parse_int_vec():
    n = np.random.randint(2^31-1,size=1000)
    for i in range(1,1000):
        ni = n[i]
        s = hex(ni)
        m = int(s,16)
        assert m == ni
```

Timing it yields:

```
1000 loops, best of 3: 848 µs per loop
```

Not bad, 4 times faster than the original code, and close to the Cython code speed.

Once we have an array it seems silly to loop over it to apply the `hex()` and the `int()` functions one element at a time. Good news it that Numpy provides a way to apply functions to an array rather than in a loop, namely the `numpy.vectorize()` function. This function takes as input a function that operate one one object at a time It returns a new function that operates on an array.

```
vhex = np.vectorize(hex)
vint = np.vectorize(int)

def parse_int_numpy():
    n = np.random.randint(0,2**31-1,1000)
    s = vhex(n)
    m = vint(s,16)
    np.all(m == n)
    return s
```

This code runs a bit faster, and nearly as fast as the Cython code:

```
1000 loops, best of 3: 733 µs per loop
```

Cython can be used to speed this up.

```
%%cython
import numpy as np
import cython

@cython.boundscheck(False)
@cython.wraparound(False)
cpdef parse_int_vec_cython():
    cdef:
        int i,m
        int[:] n
    n = np.random.randint(0,2**31-1,1000)
    for i in range(1,1000):
        m = int(hex(n[i]),16)
        assert m == n[i]
```

Timing it yields.

```
1000 loops, best of 3: 472 µs per loop
```

Summary

We have described above how to speed up 4 of the examples used by the Julia team. There are 3 more:

- pisum can run 29 times faster with Numba.
- randmatstat can be made 2 times faster by better use of Numpy.
- randmatmul is so simple that no tool can be applied to it.

The notebook containing complete code for all 7 examples is available on github and on nbviewer.

Let's summarize in a table where we are. We display the speedup we get between the original Python code and the optimized one. We also display the tools we used for each of the benchmark example used by the Julia team.

Time in micro seconds	Julia	Python Optimized	Python Original	Julia / Python Optimized	Numpy	Numba	Cython
Fibonacci 64 bits	80	24	NA	3.8			X
Fib BigInt	12,717	1,470	3,770	8.7			
quicksort	419	306	17,700	1.4	X		X
Mandelbrot	196	126	6,570	1.6	X	X	
pisum	34,783	20,400	926,000	1.7		X	
randmatmul	95,975	83,7000	83,700	1.1	X		
parse int	244	472	3,290	0.5	X		X
randmatstat	14,544	83,200	160,000	0.2	X		

This table shows that optimized Python code is faster than Julia for the first 6 examples, and slower for the last 2. Note that for Fibonacci I used the recursive code to be fair.

I do not think that these micro benchmarks provide a definite answer about which language is fastest. For instance, the randmatstat example deals with 5x5 matrices. Using Numpy arrays for that is an overkill. One should benchmark with way larger matrices.

I believe that one should benchmark languages on more complex code. A good example is given in Python vs Julia - an example from machine learning. In that article, Julia seems to outperform Cython. If I have time I'll give it a try with Numba.

Anyway, it is fair to say that on the micro benchmark, Python performance matches Julia performance when the right tools are used. Conversely, we can also say that Julia's performance matches that of compiled Python. This in itself is interesting given Julia does it without any need to annotate or modify the code.

Takeway

Let's pause for a moment. We have seen a number of tools that should be used when Python code performance is critical:

- Profiling with line_profiler.
- Writing better Python code to avoid unnecessary computation.
- Using vectorized operations and broadcasting with Numpy.
- Compiling with Cython or Numba.

Use these tools to get a feel of where they are useful. At the same time, use these tools wisely. Profile your code so that you can focus on where optimization is worth it. Indeed, rewriting code to make it faster can sometime obfuscate it or make it less versatile. Therefore, only do this when the resulting speedup is worth it. Donald Knuth once captured nicely this advice :

" We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

Note however that Knuth's quote does not mean optimization isn't worth it, see for instance Stop Misquoting Donald Knuth! and The 'premature optimization is evil' myth.

Python code can, and should be, optimized when and where it makes sense.

Let me conclude with a list of interesting articles discussing the tools I used and more:

- How to optimize for speed A short optimization guide by scipy team. It also discusses memory profiling.
- A guide to analyzing Python performance. Short intro to various profilers.

- Numba vs. Cython: Take 2, Understanding the FFT Algorithm, and Optimizing Python in the Real World : NumPy, Numba, and the NUFFT . Three interesting posts from Jake Vanderplas. In the latter, he shows how Python plus Numba can yield code only 30% slower than highly optimized Fortran code.
- Enhancing Performance in pandas documentation. A useful guide on how to make pandas code faster.
- Faster code via static typing and Using Cython with NumPy in Cython documentation.
- Numba vs Cython: How to Choose . Title says it all.
- Python Is Not C: Take Two.

Update on August 2, 2016. A reader, Ismael V.C. pointed me to the memoize function of Julia, and I updated the post accordingly.

Update on december 16, 2015. Python 3.4 has a built-in cache that can greatly speed up Fibonacci() function. I updated the post to show its use.

Update on December 17, 2015. Added running times for Julia 0.4.2 on the same machine where Python was run.

Update on Feb 2, 2016. Added the code on github and on nbviewer., and updated timings with Python 3.5.1.

Update on March 6, 2016. Improved parse int code using Numpy.

Tags: *numpy numba python julia cython*

Comments (30) ***Add a Comment*** ***More Actions***

Previous Entry Main Next Entry