Optimizing Python

Joan Creus-Costa

Physics 91SI.

2016

Is it even worth it?

First rule of optimization.

Premature optimization is the root of all evil.

– God

Is it even worth it?

HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

HOW MUCH TIME YOU SHAVE OFF	HOW OFTEN YOU DO THE TASK: 50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
30 MINUTES		6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
1 HOUR		10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
6 HOURS				2 MONTHS	2 WEEKS	1 DAY
1 DAY					8 WEEKS	5 DAYS

Table: Maybe you do not even need to care about optimization!

Source: https://xkcd.com/1205/

Sometimes it is.

- Science can get computationally expensive.
  - Simulations with many steps.
  - Analyzing massive data sets.
  - Machine learning and complex regressions.
  - Realtime calculations.
- But using Python is still desirable.
  - Who wants to use Fortran?
  - Python offers high-level abstraction, much easier to write.

Solutions?

1. Fix the native Python code.
2. Use NumPy and SciPy to their full potential.
3. Non-reference implementations of Python: PyPy.
4. C annotations in Python: Cython.
5. Calling C code directly.
6. Other specialized libraries: Theano & GPU.

Optimizing native Python

- Python is fundamentally slower than some other languages: it's interpreted, not compiled.
- Python interpreter can't make assumptions about what kinds of variables we're using.
- Added abstraction makes everything more expensive than, say, C.

For loops are baaaad


```
In [3]: l = range(1000)
        def f(x):
            return x+2

In [4]: %%timeit
        o = map(f, l)

10000 loops, best of 3: 91 µs per loop

In [5]: %%timeit
        o = []
        for i in l:
            o.append(f(i))
        o

10000 loops, best of 3: 148 µs per loop

In [6]: import numpy as np
        ll = np.arange(1000)

In [7]: %%timeit
        ll + 2

100000 loops, best of 3: 2.8 µs per loop
```

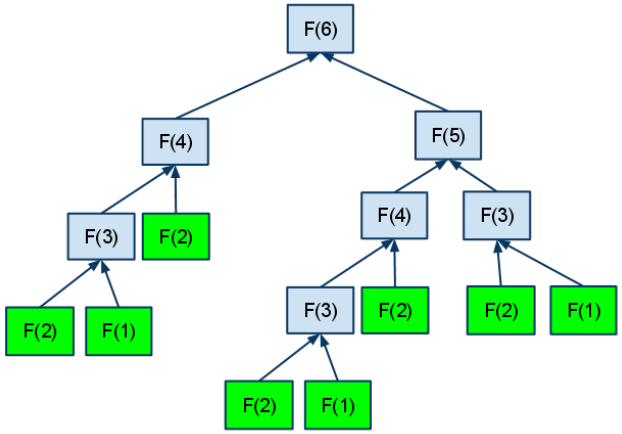
Caching is useful


```
In [12]: def fib(n):
             if n <= 0: return 0
             if n == 1: return 1
             return fib(n-1)+fib(n-2)
         %timeit fib(20)

100 loops, best of 3: 2.55 ms per loop

In [19]: def fib(n):
             cache = {0: 1, 1: 1}
             def innerfib(n):
                 if cache.get(n): return cache[n]
                 res = innerfib(n-1)+innerfib(n-2)
                 cache[n] = res
                 return res
             return innerfib(n)
         %timeit fib(20)

100000 loops, best of 3: 10 µs per loop
```



But function calls are still bad


```
In [5]: def F(n):
            a,b = 0,1
            for _ in range(n-1):
                a, b = b, a + b
            return b
        %timeit F(20)

1000000 loops, best of 3: 1.2 µs per loop

In [7]: from math import sqrt
        %timeit ((1+sqrt(5))**20 - (1-sqrt(5))**20)/((2**20)*sqrt(5))

1000000 loops, best of 3: 749 ns per loop
```

Fun fact:

$$F_n = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}$$

Try it yourself: factorials!!!

All the small things...


```
In [2]: # Python 2.x: range is slow, xrange is better

In [9]: %%timeit
        for i in range(100000):
            i

100 loops, best of 3: 3.17 ms per loop

In [10]: %%timeit
         out = []
         for i in xrange(100000):
             i

1000 loops, best of 3: 1.89 ms per loop

In [7]: # Don't import things _inside_ loops!

In [8]: %%timeit
        for i in xrange(10000):
            import math
            math.sqrt(i)

100 loops, best of 3: 6.4 ms per loop

In [11]: %%timeit
         import math
         for i in xrange(10000):
             math.sqrt(i)

1000 loops, best of 3: 900 µs per loop
```

Less eyeballing: code profiling


```
In [8]: def foo(n):
            phrase = 'repeat me'
            pmul = phrase * n
            pjoi = ''.join([phrase for x in xrange(n)])
            pinc = ''
            for x in xrange(n):
                pinc += phrase
            del pmul, pjoi, pinc

In [12]: %load_ext line_profiler

In [13]: %lprun -f foo foo(10000)
```

```
Timer unit: 1e-06 s

Total time: 0.015519 s
File: <ipython-input-8-285b08168dbb>
Function: foo at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def foo(n):
     2         1            2      2.0      0.0      phrase = 'repeat me'
     3         1           49     49.0      0.3      pmul = phrase * n
     4     10001         5817      0.6     37.5      pjoi = ''.join([phrase for x in xrange(n)])
     5         1            0      0.0      0.0      pinc = ''
     6     10001         3879      0.4     25.0      for x in xrange(n):
     7     10000         5767      0.6     37.2          pinc += phrase
     8         1            5      5.0      0.0      del pmul, pjoi, pinc
```

(Example from http://pynash.org/2013/03/06/timing-and-profiling/)

Other profiling tools

- Several profilers to choose from.
- Suited for different needs: line-by-line, functions, files. . .
- *Do you really need one?*

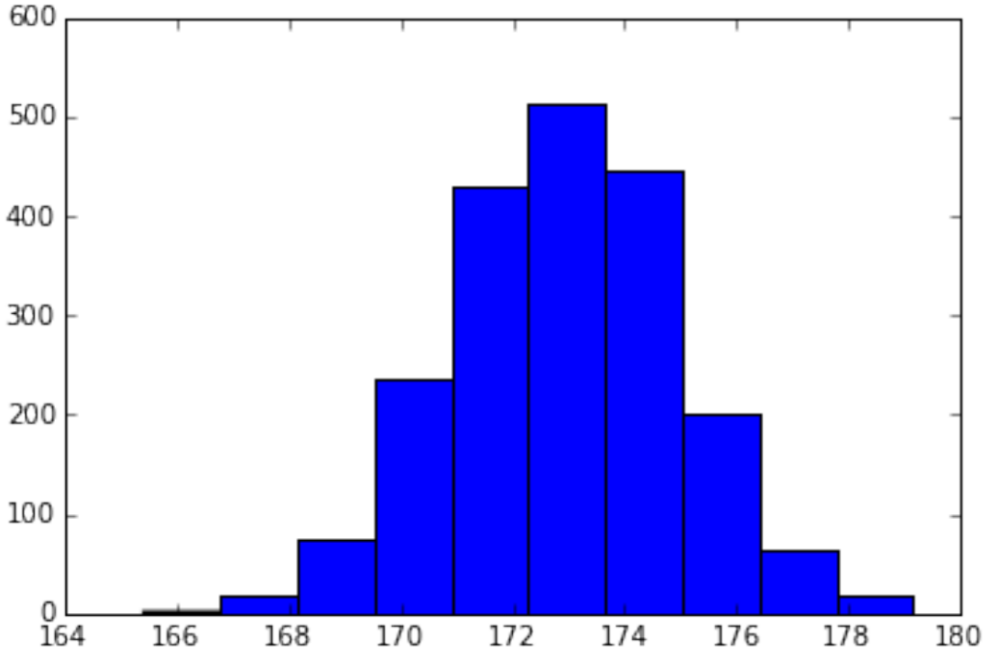
Use NumPy and SciPy to their full potential

- Don't reinvent the wheel: it was already invented in FORTRAN in the 70s.
- NumPy arrays are extremely efficient, and are easy to operate on.
- Anything starting with `np.` is fast, in general.
- Complicated algorithms? Try SciPy first:
  - Fourier transform.
  - (Function) optimization routines.
  - Search algorithms.
  - ...

Arrays are excellent


```
In [40]: # Error propagation. Useful for Physics 67!
         values = np.random.normal(172.9, 2, size=2000)
         plt.hist(values)
```

Out[40]:



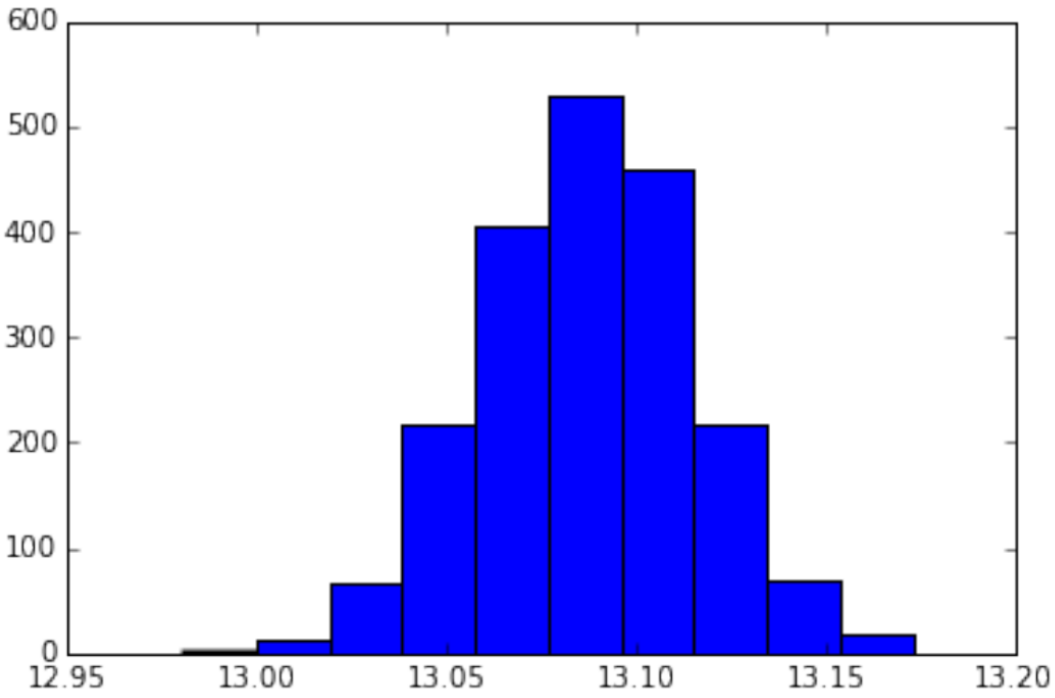

```
In [41]: values.mean(), values.std()
```

Out[41]: (172.89046136607413, 2.0367252814504306)

Arrays are excellent


```
In [44]: complicated = np.log(np.sqrt(values)**5+3*values**2)
         plt.hist(complicated)
```

Out[44]:




```
In [45]: complicated.mean(), complicated.std()
```

Out[45]: (13.086996595828136, 0.028364833750117974)

. . . even in contrived examples


```
In [26]: import numpy as np
         def get_distance(lat, lon, rlat, rlon):
             dlon = (lon-rlon)*np.pi/180
             dlat = (lat-rlat)*np.pi/180
             inverse_angle = (np.sin(dlat / 2) ** 2 + np.cos(rlat*np.pi/180) *
                              np.cos(rlat*np.pi/180) * np.sin(dlon / 2) ** 2)
             haversine_angle = 2 * np.arcsin(np.sqrt(inverse_angle))
             return haversine_angle * 6367

In [38]: n = 40000
         points = np.array([np.random.normal(37, scale=0.1, size=n),
                            np.random.normal(122, scale=0.1, size=n)]).T

In [22]: def average_distance(pointlist):
             numerator = 0
             denominator = 0
             for point in points:
                 dist = get_distance(point[0], point[1], 37, 122)
                 numerator += dist
                 denominator += 1
             return numerator/denominator

In [39]: %timeit average_distance(points)

1 loops, best of 3: 757 ms per loop

In [40]: %timeit get_distance(points[:, 0], points[:, 1], 37, 122).mean()

100 loops, best of 3: 2.47 ms per loop
```

What if I can't always use NumPy?

Use *vectorization*.

```
In [9]: import numpy as np
        np.set_printoptions(threshold=2)
        def f(a,b):
            return a+b

In [18]: numbers = np.random.uniform(-10,10,size=(1000,2))

In [19]: numbers

Out[19]: array([[ 1.07352148,  8.26719023],
                [-9.04961018,  5.12031447],
                [-1.08378406, -4.07415444],
                ...,
                [ 7.80785844,  1.9157844 ],
                [-3.62611227, -2.41889295],
                [ 5.74427794, -6.30248146]])

In [20]: %%timeit
         out = []
         for (a, b) in numbers:
             out.append(f(a, b))

1000 loops, best of 3: 1.19 ms per loop

In [21]: ff = np.vectorize(f)

In [22]: %%timeit
         out = f(numbers[:, 0], numbers[:, 1])

100000 loops, best of 3: 4.9 µs per loop
```

PyPy and alternatives to CPython

- Non-reference implementation of Python.
- Written in Python itself.
- Much faster thanks to a JIT compiler.
- Caveat: *no (good) support for NumPy.*

Advanced techniques: Cython

- Best of both worlds (C and Python) in theory.
- C annotations in Python to work around the interpreter.

A word of caution

Think twice before using it, unless you're slightly masochistic!

Advanced techniques: C extensions

- Call and import C functions directly from Python.
- Esasy if you already know C.
- Very fast once imported – might have some overhead.
- Have to compile C with special `<Python.h>` header.

Advanced techniques: GPU

- CPU is a general-purpose processor. GPU is the processor in the graphics card.
- Highly parallel. Useless for general computation, but glorious for specific, highly-parallel problems (e.g. machine learning, matrix operations).
- Theano: builds upon NumPy, provides `Tensors`, allows to use the GPU with a single line of code.
- Limitations: more verbose, have to rethink computation flow.

A word of caution

Avoid unless a) you're doing machine learning, or b) you hate yourself.