Python Performance Profiling and Optimization 101

By Caleb Bell
Schlumberger Software Expert
Open Source Developer
Software Developer
Chemical Engineering Graduate
Works for Schlumberger

Why does performance matter?

- Development: Faster loading = faster coding
- Testing: Shorter the test run, more often it gets ran- find/fix bugs faster
- Deployment: More requests/server; faster responses; spin up new servers faster

How do I measure speed?

- %timeit macro in Jupyter/Ipython

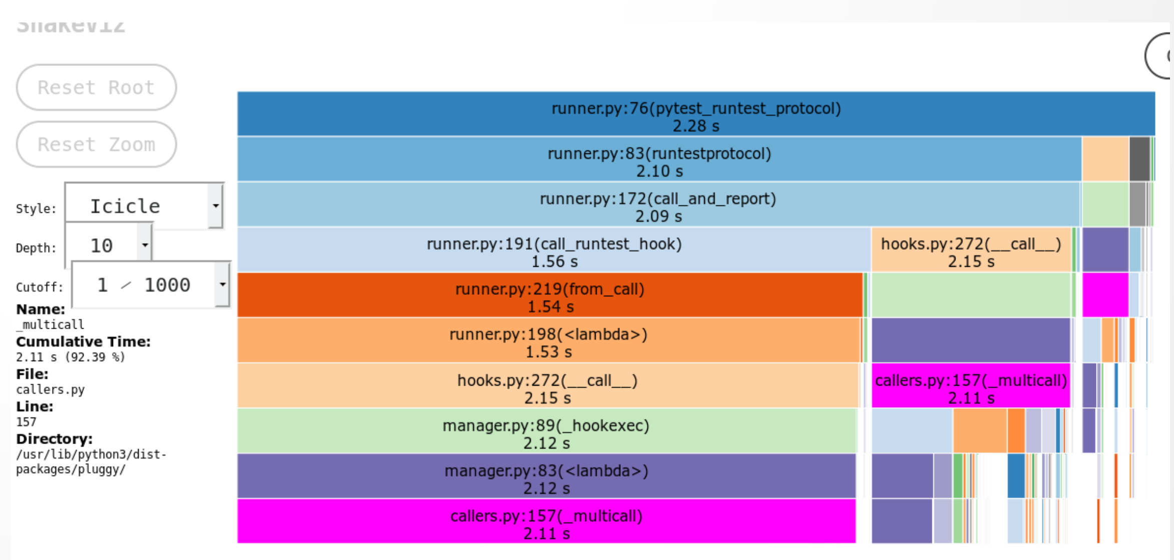
```
In [81]: def double(x): return x*x

In [82]: %timeit double(16.0)
78.3 ns ± 1.25 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

- cProfile and visualizers: snakeviz suggested
  - Times every function call
- line-profiler: Shows time per line

cProfile/snakeviz

- python3 -m snakeviz yourprofile.prof

```
import cProfile
pr = cProfile.Profile()
pr.enable()
my_stuff()
pr.disable()
pr.dump_stats('yourprofile.prof')
```



lineprofiler

```
def to_time():
    a = 1
    b = 2
    c = 3
    d = 4
    return a*b*c*d
import line_profiler
%load_ext line_profiler
%lprun -f to_time to_time()
```

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def to_time():
     2         1            3      3.0     42.9      a = 1
     3         1            1      1.0     14.3      b = 2
     4         1            1      1.0     14.3      c = 3
     5         1            0      0.0      0.0      d = 4
     6         1            2      2.0     28.6      return a*b*c*d
```

What makes Python Slow?

- Lack of a JIT (Just-In-Time Compiler)
- Extreme dynamic behavior
- Belief Python can’t be fast
- Dictionaries for everything
- No Types
- Reliance on strings

What do I do about Python’s slowness?

- Use another language?
  - No! Python is awesome!
- Code the performance critical parts in C/C++/Fortran/Rust?
  - Works great if you are farmiliar with them, and don’t mind crashes & slower development

What do I do about Python’s slowness?

- Use an accelerator?
  - Great Choice!
- Cython
  - Gently introduces you to using C
  - Requires a compiler
  - Have to do all the work yourself
- Numba
  - Great for numerical work only
- PyPy
  - Great for web servers, pure-python stuff
  - Currently slow with numpy

What is Cython?

- Subset of Python with type hints - allows compilation to C
- Looks more like C than Python
- Requires compiler
- No reuse of code

```
cimport cython
from libc.math cimport log

@cython.boundscheck(False)
@cython.wraparound(False)
@cython.cdivision(True)
cpdef nrtl_cy(double [:] X, double [:,:] tau, double [:, :] G):
    cdef int i, j, k, nc
    nc = X.shape[0]
    cdef double A, SumA, SumB, SumC, SumD, SumE, aux, aux2
    cdef double [:] lngama = np.zeros(nc)

    for i in range(nc):
        SumC = SumD = SumE = 0.
        for j in range(nc):
            A = X[j]*G[i,j]
            SumA = SumB = 0.
            for k in range(nc):
                aux = X[k]*G[k,j]
                SumA += aux
                SumB += aux*tau[k,j]
            SumC += A/SumA*(tau[i,j]-SumB/SumA)
            aux2 = X[j]*G[j,i]
            SumD += aux2*tau[j,i]
            SumE += aux2
        lngama[i] = SumD/SumE+SumC
    return lngama.base
```

Sample from phasepy project:
https://github.com/gustavochm/phasepy

What is Numba?

- Translator of Python into assembly (well, actually - LLVM intermediate representation which gets compiled)
- Allows wrapping existing Python with a single decorator to improve performance
- For best performance (`nopython` mode) requires type hints, no dynamic attributes, definitely no metaprogramming!

Numba example with benchmark

- 53 ms Python
- 8.2 ms Numpy
- 4.7 ms Numba optimized
- 2.1 ms Numba Threaded

```
from math import exp
import numpy as np
from numba import njit

def test_exp(n):
    tot = 0.0
    for i in range(n):
        tot += exp(i*1e-10)-1.0
    return tot

def test_exp_np(n):
    return np.sum(np.exp(np.arange(n)*1e-10)-1)

@njit()
def test_exp_numba(n):
    return np.sum(np.exp(np.arange(n)*1e-10)-1.0)

@njit()
def test_exp_faster_numba(n):
    tot = 0.0
    for i in range(n):
        tot += np.exp(i*1e-10)-1.0
    return tot

test_exp_numba_parallel = njit(test_exp_np, parallel=True, fastmath=True)

N = 400000
%timeit test_exp(N)
%timeit -n 100 -r 3 test_exp_np(N)
%timeit -n 100 -r 3 test_exp_numba(N)
%timeit -n 100 -r 3 test_exp_faster_numba(N)
%timeit test_exp_numba_parallel(N)
```

```
53 ms ± 132 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
8.23 ms ± 146 µs per loop (mean ± std. dev. of 3 runs, 100 loops each)
7.92 ms ± 605 µs per loop (mean ± std. dev. of 3 runs, 100 loops each)
4.77 ms ± 316 µs per loop (mean ± std. dev. of 3 runs, 100 loops each)
2.15 ms ± 270 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Cons of Numba

- Cannot distribute compiled code; users have to have Numba
- Eats up 50 MB RAM loading llvmlite
- Doesn't support some language features, i.e. special exception handling
- Focused on numerical computation – not class-heavy code like a webserver

Pros of Numba

- Offers fastest numerical performance
- Can thread expensive computation
- Integrates nicely with numpy
- Can start out without almost any custom code – allows numba as an optional feature

What is PyPy?

- Alternate Python interpreter – shares no code with CPython, which is known as Python
- Just-In-Time compiler with different garbage collection strategy makes code faster
- CPU executes generated assembly, not bytecode on a virtual machine

Cons of PyPy

- C extensions (numpy, scipy, pandas) are slow – sometimes slower than CPython
- JIT has to re-optimize code each time PyPy started
- Uses more memory (1.5-2.0 times)
- Slower startup

Pros of PyPy

- Use the same code for extra performance (if not using numpy, etc.)
- Still python – no extensions needed
- 7x performance increase possible sometimes, even 35x performance increase for simple math!

Optimization tips

- Import functions from a module directly – don't use them like `module.function`
- Avoid importing in functions
- Don't assign your return value – just return it

```
In [79]: %timeit math.sqrt(5.23523523
79.8 ns ± 1.63 ns per loop (mean ± st

In [80]: %timeit sqrt(5.2352352352)
56.9 ns ± 9.43 ns per loop (mean ± st
```

```python
from math import sqrt
```

```python
x = 5.231532532

def import_again(x):
    from math import sqrt
    y = sqrt(x)
    return y

def already_imported(x):
    return sqrt(x)

%timeit import_again(x)
%timeit already_imported(x)
```

```
953 ns ± 12.2 ns per loop (r
151 ns ± 2.6 ns per loop (me
```

Optimization tips

- When checking if a value is in something, use a set (or dict)
- Avoid keyword arguments in short functions

```
a = list(range(1000))
b = set(range(1000))
c = range(1000)
d = np.arange(1000)
%timeit 543 in a
%timeit 543 in b
%timeit 543 in c
%timeit 543 in d
```

```
5.38 µs ± 134 ns per loop (
46.3 ns ± 0.851 ns per loop
92.8 ns ± 1.25 ns per loop
7.84 µs ± 47.8 ns per loop
```

```
def func_all_args(a, b, c, d, e, f):
    return a

def func_default_args(a=1.0, b=2.453, c=32.42, d=2.0,
    return a

a, b, c, d, e, f = 1.0, 2.0, 4.0, 9.0, 64.0, 317.0

%timeit func_all_args(a, b, c, d, e, f)
%timeit func_all_args(a, b, c, d=d, e=e, f=f)
%timeit func_all_args(a=a, b=b, c=c, d=d, e=e, f=f)
%timeit func_default_args(a, b, c, d, e, f)
%timeit func_default_args(a=a, b=b, c=c, d=d, e=e, f=f
```

```
186 ns ± 1.32 ns per loop (mean ± std. dev. of 7 runs,
220 ns ± 5.2 ns per loop (mean ± std. dev. of 7 runs,
286 ns ± 41.5 ns per loop (mean ± std. dev. of 7 runs,
187 ns ± 3.83 ns per loop (mean ± std. dev. of 7 runs,
217 ns ± 5.83 ns per loop (mean ± std. dev. of 7 runs,
```

Optimization tips

- Reuse lists instead of alocating new ones
- Cache repeated references in a list
- Cache calculated values

```python
x = [1.214, 543346.235, 65.87653, 3463.346, 4532.234, 9.0, 4.23, 9.3]
N = len(x)
```

```python
def to_time_bad_jacobian(x):
    jac = []
    for i in range(N):
        row = []
        for j in range(N):
            v = x[i]*1.12412+x[j]*653.3245
            row.append(v)
        jac.append(row)
    return jac
%timeit to_time_bad_jacobian(x)
```

```
12.6 µs ± 660 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```python
jac = [[0.0]*N for _ in range(N)]
def to_time_good_jacobian(x):
    for i in range(N):
        for j in range(N):
            jac[i][j] = x[i]*1.12412+x[j]*653.3245
    return jac
%timeit to_time_good_jacobian(x)
```

```
11.9 µs ± 1.49 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```python
def to_time_best_jacobian(x):
    for i in range(N):
        row = jac[i]
        t0 = x[i]*1.12412
        for j in range(N):
            row[j] = t0 + x[j]*653.3245
    return jac
%timeit to_time_best_jacobian(x)
```

```
7.28 µs ± 65.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Optimization tips

- Avoid .format(); use old-stype % formatting or f-strings if Python 3.6 + only

```
apples = 'golden'
bananas = 'yellow'
numbers = 13253215.1351325

def to_time_old():
    return 'I can use format %s %s %s' %(apples, bananas, numbers)
def to_time_format():
    return 'I can use format {apples} {bananas} {numbers}'.format(
        apples=apples, bananas=bananas, numbers=numbers)
def to_time_f_string():
    return f'I can use format {apples} {bananas} {numbers}'

%timeit to_time_old()
%timeit to_time_format()
%timeit to_time_f_string()
```

```
898 ns ± 8.67 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops
1.51 µs ± 8.36 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops
916 ns ± 15.1 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops
```

Optimization tips

- Use list comprehensions often

```
def time_comprehension():
    return [i for i in range(35) if i > 10]

def time_manual():
    a_list = []
    for i in range(35):
        if i > 10:
            a_list.append(i)
    return a_list

%timeit time_comprehension()
%timeit time_manual()
```

```
1.62 µs ± 16.6 ns per loop (mean ± std. dev.
2.3 µs ± 19.7 ns per loop (mean ± std. dev.
```

Optimization tips

- Cache function results wherever possible, ex: Fibonachi sequence

```
from functools import lru_cache
N = 30
def to_time_naive():
    def fibonacci_number(n):
        if n == 0: return 0
        if n == 1: return 1
        return fibonacci_number(n-1) + fibonacci_number(n-2)
    return [fibonacci_number(i) for i in range(N)]

def to_time_lru():
    @lru_cache(maxsize=1024)
    def fibonacci_number(n):
        if n == 0: return 0
        if n == 1: return 1
        return fibonacci_number(n-1) + fibonacci_number(n-2)
    return [fibonacci_number(i) for i in range(N)]

%timeit -n 1 -r 1 to_time_naive()
%timeit to_time_lru()
```

```
583 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
19.9 µs ± 417 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Optimization tips

- Use math to your advantage
  - log, power rules
  - Integer powers can be multiplications
  - Division can be multiplication of inverse

```python
from math import log, sqrt

def bad_formula(a, b):
    return (((log(a) + log(b))/5)**2)/(2+sqrt(3))

def good_formula(a, b):
    t = 0.2*log(a*b)
    return 0.2679491924311227*t*t # 1/(2+sqrt(3))

%timeit bad_formula(3, 4)
%timeit good_formula(3.0, 4.0)
```

```
498 ns ± 5.7 ns per loop (mean ± std. dev. of 7 ru
220 ns ± 2.16 ns per loop (mean ± std. dev. of 7 r
```

Recommendations

- Profile code regularly but not every day
- Keep well-optimized pure Python code
- Invest in an accelerator carefully
- Prefer accelerators which don’t require much code change (PyPy, numba)
- Writting in C++ and linking with pybind11/boost also an option!

What I do

- Use PyPy as my computations are numeric
  - Numpy alternate implementations sometimes
- Support both CPython and PyPy – balance optimizations between them
- Plan to use numba when available once it supports more of Python
- At work – write some in C++

Questions?

Now, go forth and Optimize!