Profiling and Optimizing Jupyter Notebooks - A Comprehensive Guide

Finding bottlenecks and increasing your speed performance by magnitudes with some tips I came along over the past year.

Muriz Serifovic Follow
Dec 31, 2018 · 11 min read

While we all know that premature micro optimizations are the root of all evil, thanks to Donald Knuth's paper "*Structured Programming With Go To Statements*" [1], eventually at some point in your data exploration process you grasp for more than just the current "working" solution.

The heuristic approach we usually follow considers:

1. Make it **work**.
2. Make it **right**.
3. Make it **fast**.

Before jumping straight to the third point and starting with refactoring our code, it is important to identify where the performance bottlenecks are, to make an informed decision on the course of action we want to follow.

This is a fundamental step if we need to achieve the greatest benefit with the least amount of work. Truth be told, one of the most voluminous mistakes in this setting would be to make an informed conjecture and fine-tune what we believe is the main driver of the issue. By profiling our code, we remove this vulnerability since we will know precisely where the issues are.

Since we're using Jupyter Notebooks here, we may as well want to take advantage of conveniences that come along with it, such as magic commands. **Magic commands** are with no doubt one of the sweet enhancements for extending a notebook's capabilities. In detail, we will take a look at:

- `%time` and `%timeit`
- `%prun` and `%lprun`
- `%mprun` and `%memit`
- `%%heat`
- visualizing the output of a profiling session with `snakeviz`

and study the following:

- pythonic way of coding
- loop optimization method with **vectorization**
- optimization with different algorithms

Our first goal is to identify what's causing us headaches. In general, profiling involves measuring the resource you want to optimize for, whether it is memory usage or CPU time.

In the next examples we will consider how to strive for optimization if our tasks are **"CPU-bound"** (thus time spent in the CPU) and, in contrast to it, how and why to pay particular attention to **memory intensive** tasks.

> ***Note**: In this article I'm **not** diving into parallel or high-performance computing for tackling issues regarding performance bottlenecks. This is out of scope for this post and might be an idea to write about in the future.*

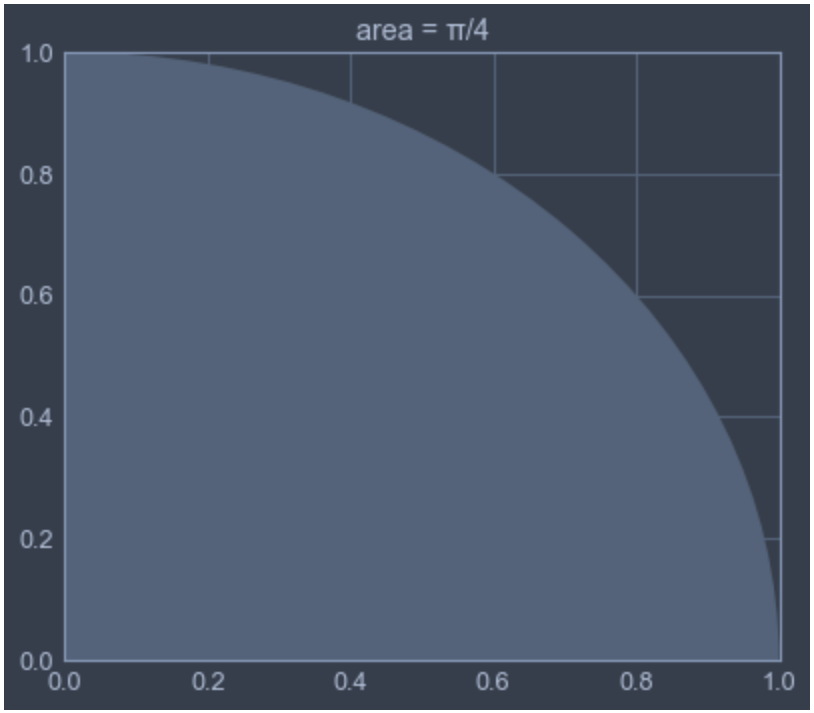
Approximating π with Monte Carlo integration

A Monte Carlo simulation is a method for estimating an answer to a problem by randomly generating samples. They are primarily suited for calculating a "brute force" approximation to the solution of a system which

may be of high dimension, such as DeepMind's AlphaGo Zero where Monte Carlo tree search was being utilized.

We will define a slow method which evaluates pi using random generated data points and then look for ways to optimize. Remember that the area covered by a circle with radius 1 inscribed in a square, equals exactly to a **quarter of pi.**

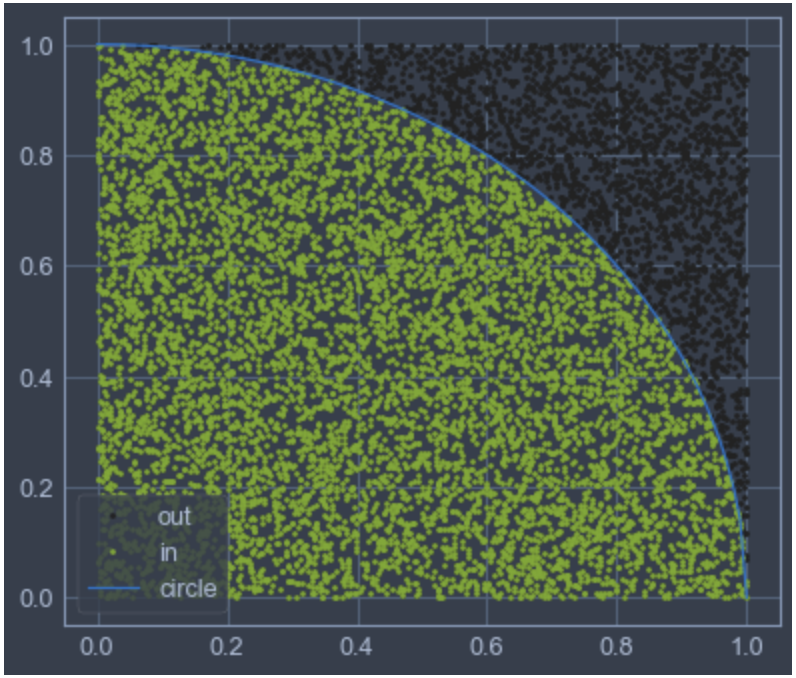
We get the value of pi by taking the ratio of **area of circle** to **area of the square**,



unit quarter-circle

where we approximate the area with our generated random points.

$$\frac{\text{area of circle}}{\text{area of square}} \approx \frac{\text{points in circle}}{\text{total points}}$$



π ≈ 3.1417

Consider the following code where we approximate the integral:

```python
from random import random

def estimate_pi(n=1e7) -> "area":
    """Estimate pi with monte carlo simulation.

    Arguments:
        n: number of simulations
    """
```

```
9       in_circle = 0
10      total = n
11
12      while n != 0:
13          prec_x = random()
14          prec_y = random()
15          if pow(prec_x, 2) + pow(prec_y, 2) <= 1:
16              in_circle += 1 # inside the circle
17          n -= 1
18
19      return 4 * in_circle / total
```

mc_01.py hosted with ♡ by GitHub view raw

In the while loop we generate a random point and check if it falls within the circle or not. If it does, we increment `in_circle` by one. We do this `n` times and return the **ratio** times 4 in order to estimate pi (line 19).

Or in more mathematical terms:

$$I = \int_0^1 \int_0^1 f(x, y) \mathrm{d}x\mathrm{d}y = \frac{\pi}{4}$$

where

$$f(x, y) = \begin{cases} 1 & \text{if } x^2 + y^2 \leq 1 \\ 0 & \text{else} \end{cases}$$

> ***Note**: To make things even worse, the iterative code could be implemented with a recursive solution. The recursive version can be found here.*

Before starting hesitant optimization iterations, it is important to measure the total execution time of the function we want to optimize without any kind of profiler overhead and save it somewhere for later reference.

The magic command `%time` can offer a useful tool for comparing the runtime of different functions (i.e. **benchmarking**).

In [1]: `%time estimate_pi()`

```
CPU times: user 6.2 s, sys: 11.8 ms, total: 6.21 s
Wall time: 6.25 s
```

If we're interested to normalize `%time`, use:

In [2]: `%timeit -r 2 -n 5 estimate_pi()`

with `-r` denoting number of runs and `-n` number of loops. What we get is:

```
6.29 s ± 73.7 ms per loop (mean ± std. dev. of 2 runs, 5 loops each)
```

Our first approach takes **6.29 seconds** to estimate pi which we'll keep in mind and come back onto later on.

cProfile

To actually learn what takes up most of the execution time, python ships with a great profiler, breaking down the execution function by function. It causes our attention to shrink down to critical functions by handing out a high-level view of performance.

In [3]: `%prun estimate_pi()`

```
         40000004 function calls in 10.918 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    6.137    6.137   10.918   10.918 <ipython-input-3-69dbc2813525>:3(estimate_pi)
 20000000    3.279    0.000    3.279    0.000 {built-in method builtins.pow}
 20000000    1.502    0.000    1.502    0.000 {method 'random' of '_random.Random' objects}
        1    0.000    0.000   10.918   10.918 {built-in method builtins.exec}
        1    0.000    0.000   10.918   10.918 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

> ***Note**: You should keep in mind that profiling typically adds an overhead to your code.*

The report shows for each function:

- the number of calls (ncalls)

- the total time (tottime) spent on it excluding calls to subfunctions
- how long each call took (percall, excluding and including)
- the total time (cumtime) including all calls to subfunctions

The output will be written to stdout. If we wish to save the output for later inspection for example with **pstats**, use the **-D** option to save on disk.

In [4]: `%prun -D pi.prof estimate_pi()`

```
*** Profile stats marshalled to file 'pi.prof'.
```

An another useful option is **-s** which enables sorting for a particular column. For example sorting cumulative time in descending order:

In [5]: `%prun -s cumulative estimate_pi()`

```
         40000004 function calls in 10.703 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   10.703   10.703 {built-in method builtins.exec}
        1    0.000    0.000   10.703   10.703 <string>:1(<module>)
        1    5.994    5.994   10.703   10.703 <ipython-input-8-69dbc2813525>:3(estimate_pi)
 20000000    3.226    0.000    3.226    0.000 {built-in method builtins.pow}
 20000000    1.483    0.000    1.483    0.000 {method 'random' of '_random.Random' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Interesting seems to be the number of calls to built-in functions `pow()` and `random()`. Before we address this extensive number of calls, let's have a look at a much more convenient library delivering even more concrete reports.

Line profiler

`%lprun` command yields the time spent on each line of code giving us a line by line report. Since not shipped by default, install the library with pip

```
!pip install line_profiler
```

and load the extension manually in the notebook.

```
%load_ext line_profiler
```

With similar syntax to `%prun`, it's easy as pi(e) to locate hotspots in the code with only difference that functions need to be explicitly defined.

In [6]: **`%lprun`** `-f estimate_pi estimate_pi()`

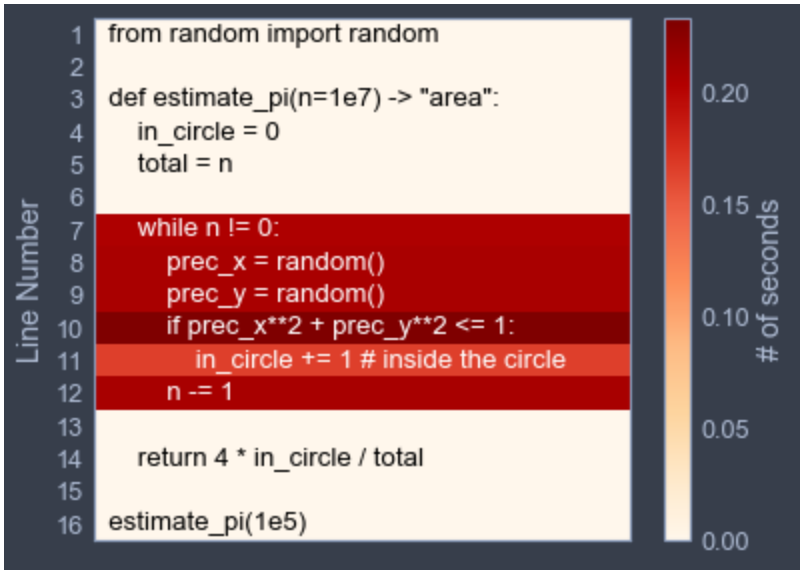
```
Timer unit: 1e-06 s

Total time: 28.0353 s
File: <ipython-input-8-69dbc2813525>
Function: estimate_pi at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           def estimate_pi(n=1e7) -> "area":
     4         1          7.0      7.0      0.0      in_circle = 0
     5         1          1.0      1.0      0.0      total = n
     6
```

```
 7  10000001     4026597.0      0.4      14.4      while n != 0:
 8  10000000     4101831.0      0.4      14.6          prec_x = random()
 9  10000000     4242937.0      0.4      15.1          prec_y = random()
10  10000000     8168531.0      0.8      29.1          if pow(prec_x, 2) + pow(prec_y, 2) <= 1:
11   7851624     3295817.0      0.4      11.8              in_circle += 1 # inside the circle
12  10000000     4199593.0      0.4      15.0          n -= 1
13
14         1           2.0      2.0       0.0      return 4 * in_circle / total
```

Notice the extensive time (29.1%) spent on the *if* statement on line 10. Take in mind that we can even colorize the report above, giving us a more intuitive way to see hot-spots.



All we have to do is insert the `%%heat` command at the top of the cell (to use load the extension `%load_ext heat` after installing with `!pip install py-heat-magic`) and it allows us to see the completely red *while* loop obliquing high cost of CPU-time, clearly showing room for optimization.

I just want to briefly mention that there is a nice library called `snakeviz` that displays profiles as a sunburst in which functions are represented as arcs.

More information can be found here.

Optimize

Before diving in methods which involve dependencies on extern libraries, let's propose the idea to make things more pythonic!

> Pythonic means code that doesn't just get the syntax right but that follows the conventions of the Python community and uses the language in the way it is intended to be used.

Remember that every call to a function is associated with overhead time, thus the vast majority of calls in the loop is something which boggles us

down. The *while* loop is just incrementing a counter by one, if a certain condition is met. To abbreviate the code, we introduce the **`sum()` method, a generator expression** and removal of pow().

```
from random import random

def estimate_pi(n=1e7) -> "area":
    """Estimate pi with monte carlo simulation.

    Arguments:
        n: number of simulations
    """
    return 4 * sum(1 for _ in range(int(n)) if random()**2 + random()**2 <= 1) / n
```

mc_02.py hosted with ♡ by GitHub view raw

Already by doing these three changes we reduce function calls by **30.37%** and gain speed improvement of **41.5 %.**

In [7]: **`%prun`** `estimate_pi()`

```
         27852980 function calls in 6.764 seconds

   Ordered by: internal time

    ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   7852975    4.756    0.000    6.113    0.000 <ipython-input-16-66efe60b7e0c>:4(<genexpr>)
  20000000    1.358    0.000    1.358    0.000 {method 'random' of '_random.Random' objects}
         1    0.650    0.650    6.764    6.764 {built-in method builtins.sum}
         1    0.000    0.000    6.764    6.764 {built-in method builtins.exec}
         1    0.000    0.000    6.764    6.764 <ipython-input-16-66efe60b7e0c>:3(estimate_pi)
         1    0.000    0.000    6.764    6.764 <string>:1(<module>)
         1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

In [8]: `%timeit -r 2 -n 5 estimate_pi()`

```
3.68 s ± 2.39 ms per loop (mean ± std. dev. of 2 runs, 5 loops each)
```

Optimize with Vectorization

Given the fact that we exactly know beforehand how many random numbers should be generated, we can simply make the attempt to place everything **before or outside** the loop.

Remember the *if* statement on line 10 which takes up nearly 30% of computational time. The only information this *if* statement requires are two coordinates, hence can again be placed outside the loop.

If the option is available we should avoid looping code altogether. Especially in data science we're familiar with NumPy and pandas, highly optimized libraries for numerical computation. A big advantage in NumPy are arrays internally based on C arrays which are stored in a contiguous block of memory **(data buffer-based array)**.

```
1  import numpy as np
2
3  def estimate_pi(n=10000000) -> "area":
4      """Estimate pi with monte carlo simulation.
5
6      Arguments:
```

```
6       Arguments:
7           n: number of simulations
8       """
9       xy = np.random.rand(n, 2)
10      inside = np.sum(xy[:, 0]**2 + xy[:, 1]**2 <= 1)
11      return 4 * inside / n
```

mc_03.py hosted with ♡ by GitHub view raw

Here we create all random points as an array with shape `(n, 2)` (line 9) and count how many times the condition is met that the point falls in the circle (line 10).

If we benchmark now the numpy version,

In [9]: `%timeit estimate_pi()`

```
388 ms ± 9.24 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

results in 388ms and therefore **16 times faster** compared to the *while* loop. Numpy's syntax is similar to standard python methods, instead of `sum()` we write `np.sum()` and essentially we're not iterating over a list anymore, instead we use numpy's vectorized routines.

Memory profiling

Juggling with large data sets involves having a clear sight of memory consumption and allocation processes going on in the background. As earlier discussed, there are tools to monitor the memory usage of your notebook.

Use `%memit` in familiar fashion to `%timeit`

In [10]: **`%memit`** `estimate_pi()`

```
peak memory: 623.36 MiB, increment: 152.59 MiB
```

We see that the function uses about **600 mebibytes** for 1e7 simulations. My hypothesis is that allocating the large array contributes to the most part. Proving this hypothesis with `%mprun` which checks the memory usage at every line.

In [11]: **`%mprun`** `-f estimate_pi estimate_pi()`

```
Line #    Mem usage    Increment   Line Contents
================================================
     2     58.6 MiB     58.6 MiB   def estim(n):
     3    211.2 MiB    152.6 MiB       xy = np.random.rand(n, 2)
     4    374.0 MiB    162.8 MiB       inside = np.sum(xy[:, 0]**2 + xy[:, 1]**2 <= 1)
     5    374.1 MiB      0.0 MiB       return 4 * inside / n
```

Very interesting seems **line 4** where incrementation is 162.8 MiB yet on the next line overall memory usage only raises by 0.1. What happens here is that we allocate on the right side of the assignment and then drop memory again since `inside` is not a numpy array anymore.

To put things together below is a one liner doing the same as above with the exception that instead of allocating a large array with shape `(n, 2)` we square x and y points on demand. Although we sacrifice readability, rewriting the expression reduces the number of assignment operations resulting in an even faster solution with **280 ms (22 times faster)**.

```python
import numpy as np

def estimate_pi(n=10000000) -> "area":
    """Estimate pi with monte carlo simulation.

    Arguments:
        n: number of simulations
    """
    return np.sum(np.random.random(n)**2 + np.random.random(n)**2 <= 1) / n * 4
```

mc_04.py hosted with ♡ by GitHub view raw

In [9]: `%timeit estimate_pi()`

```
280 ms ± 3.06 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Out of memory

Creating the array comes with a restriction to available system memory. The allocation process scales linearly with the input parameter `n` . If, for example, we would try to set the number of simulations to `1e10` our kernel would crash while creating the array. Luckily, this method is not requiring one large array, instead we split it into smaller chunks which enables us to scale the number of simulations independent to system memory.

Here we're predefine the size of the array (line 9) which equals to 80 MB. We split the number of simulations to handle 80 MB numpy array and increment `inside` each at the time.

If we simulated beforehand with setting `n` to `1e10` we would try to allocate an array of 80 GB (!) in size, simply not feasible on a standard machine. With the updated method, calculation time scales on equal terms as `n` .

Optimize with a different algorithm

If we test the latter numpy solution, we estimate pi to 3.1426944 with an relative **error of 0.035%** in 280ms. The monte carlo method is a great concept to land at an answer which doesn't promptly have all the earmarks of being deducible through random procedures.

There might me different calculations tackling the issue in a substantially more productive way. The biggest lift to any programming execution will be by changing the general way of tackling the problem. Unsurprisingly, this is the hardest change to achieve as upgrade and revamp of your coding is required.

Chudnovsky algorithm

While there are many ways to calculate pi with high-digit precision [3], a very fast method is Chudnovsky algorithm which was published by the Chudnovsky brothers in 1989 and appears in the following form:

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k (6k)!(13591409 + 545140134k)}{(3k)!(k!)^3 640320^{3k+3/2}}$$

A great write-up can be found on Nick Craig-Wood's website. The implementation of this algorithm is a non-trivial task and left as an exercise for the reader.

Profiling this method results in,

In [12]: `%timeit -r 1 -n 1000 pi_chudnovsky(10**100)`

```
13.6 µs ± 72 ns per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

13.6 microseconds which is not only over **20'000 times faster** compared to our numpy monte carlo version, in addition it successfully calculates pi to the first 100 digits.

Conclusion

Before optimization can take place, we should profile and avoid premature assumptions about possible bottlenecks (whereas the profiler never lies). To monitor CPU-time consumption and memory footprint, Jupyter offers convenient magic commands executable directly in notebook cells. Basic methods of optimization include using generator expressions and list comprehensions over explicit loops. Pure python coding is on rare occasions an instructive advice and should therefore be replaced with optimized equal methods from scientific libraries resulting in huge speed-ups.

> ***Note:*** *Consider taking optimization in Jupyter Notebook with a grain of salt, as there are some aspects to it like non-linear workflow while executing cells, which at first glance seem to be handy, but with yet another abstract layer on top of it, we loose control over how the state is being read and kept in memory. The more abstraction takes place, the harder it gets to reveal underlying issues. (Please correct me if I'm wrong.)*

For part II is planned to explore possibilities with Cython and parallel computing in python.

I welcome feedback and constructive criticism.

Cheers! 🙇

. . .

References

[1] Donald E. Knuth, *Structured Programming With Go To Statements, 1974,* https://pic.plover.com/knuth-GOTO.pdf

[2] Eric C. Anderson, *Monte Carlo Methods and Importance Sampling, 1999,* http://ib.berkeley.edu/labs/slatkin/eriq/classes/guest_lect/mc_lecture_notes.pdf

[3] Wolfram Research, http://functions.wolfram.com/PDF/Pi.pdf

Jake VanderPlas. (2016), *Python Data Science Handbook*

Cyrille Rossant. (2014), *IPython Interactive Computing and Visualization Cookbook*

Gabriele Lanaro. (2013), *Python High Performance Programming*

Python Jupyter Notebook Tech Optimization Programming