How to Improve Python Performance

Author: Chris Sabanty

Claps: 171

Date: Dec 20, 2022

â€œPython canâ€™t scale.â€

While itâ€™s true that Python frequently underperforms in cross-language benchmarks like these JSON serialization benchmarks, Python performance is usually less of an issue than database queries, network requests, or a variety of other bottlenecks. Furthermore, with a few tweaks, itâ€™s possible to make standard CPython perform much closer to the alternatives.

At Klaviyo we use Python at huge scale â€” for example, weâ€™ve created real-time, analytical dashboards that aggregate trillions of events from thousands of different sources. While doing so weâ€™ve used or experimented with a number of different techniques to make Python code faster.

I wrote this to share a few of those tips, ranging from computer science basics to a 99.9999% speed gain with a single Numba decorator, so you can focus on the great aspects of Python.

Confirm that you actually need to optimize Python

Before trying to optimize Python, you should confirm that itâ€™s truly your biggest bottleneck. Frequently Python processing is less of an issue than the following:

- Slow database queries
- Slow network requests or API calls
- Lots of database queries or network requests
- Slow file writing or reading
- Server latency
- Backed-up message queues

While this post will focus on Python speedups, some of the tools and concepts mentioned below will help with other types of issues. Regardless of whatâ€™s slowing your app down, the first step is usually to collect more information.

Add logging or timers to key areas

The simplest way to look for bottlenecks is by adding logging to your code. Hereâ€™s a basic example:

```
INFO:example:my_slow_func took 1.0032 seconds.
```

Logging is easy to implement and might be enough to help figure out whatâ€™s slow. But lots of logs can also make code tougher to follow and use tons of disk space.

Another option is to send timers and metrics to a service like StatsD. StatsD is a network daemon that listens for timers or counters and makes it easier to display them in visual interfaces like Grafana.

Grafana lets you answer questions like, which part of my codebase is slowest on average? Has there been a recent, speed-related regression? Are request times consistent or they do vary? The list goes on and on.

Compared to logging, the biggest con of Grafana and StatsD is the effort needed to set up, scale, and maintain the stack. KoalaTea wrote a post detailing the steps involved with a basic Python, Grafana, and StatsD setup. We also wrote a post four years ago describing how we made Graphite and StatsD scale to millions of metrics.

If youâ€™d rather start with logging, RealPython has a solid article with more elaborate examples.

Profile your code to pinpoint exactly whatâ€™s slow

Pythonâ€™s profilers might provide more insight if logging doesnâ€™t help. They let you see exactly which functions are being called the most and for how long. Hereâ€™s an example using cProfile, which is one of the profilers included in Pythonâ€™s standard library:

```
ncalls tottime percall cumtime percall filename:lineno(function)

1 0.000 0.000 4.278 4.278 profile_example.py:11(build_list_and_sleep)
1 0.000 0.000 3.275 3.275 profile_example.py:8(build_list)
1 0.000 0.000 3.275 3.275 profile_example.py:9(<listcomp>)
25 3.275 0.131 3.275 0.131 profile_example.py:5(calc_big_number)
1 1.003 1.003 1.003 1.003 {built-in method time.sleep}
1 0.000 0.000 0.000 0.000 cProfile.py:117(__exit__)
1 0.000 0.000 0.000 0.000 {method â€ðisableâ€™ of â€˜lsprof.Profilerâ€™ ob
```

The results show that calc_big_number() took an average of 0.131 seconds. However, since it was called 25 times, calc_big_number() took a total of 3.275 seconds and was more of a bottleneck than the time.sleep() call.

One of the biggest cons of cProfile is that it only profiles a single run of code, which is less useful if execution time varies between runs. For example, if a database query is intermittently slow or fast, cProfile might miss it. Additionally, cProfile doesnâ€™t work well with multithreading and doesnâ€™t include memory usage.

Some of these deficiencies are addressed by third-party profilers:

- py-spy lets you directly attach to a running process to get aggregate timings across a variety of runs. It also works better with threads when using the -idle flag.
- Scalene has a bunch of intriguing features including memory usage profiling, GPU profiling, copy volume, and differentiating between optimizable Python time and tougher to optimize C time. Setting up and generating a report is also very easy.

- SnakeViz creates interactive reports for Pythonâ€™s cProfile module.
- Yappiâ€™s biggest selling point is its support for concurrency profiling.

Add a cache or memoization

Once youâ€™ve figured out whatâ€™s slow, you should ask yourself these questions:

- Do any slow functions in my code always return the same result if they receive identical input?
- And if so, how much memory or disk space would storing those slow results require?

In the code snippet above, calc_big_number() always returned the same result and used a negligible amount of memory, so itâ€™s a great candidate for caching or a constant variable. Hereâ€™s the same code using Pythonâ€™s functools.cache() decorator:

```
ncalls tottime percall cumtime percall filename:lineno(function)

1 0.000 0.000 1.131 1.131 profile_example_with_cache.py:13(build_list_and_
1 1.003 1.003 1.003 1.003 {built-in method time.sleep}
1 0.000 0.000 0.128 0.128 profile_example_with_cache.py:10(build_list)
1 0.000 0.000 0.128 0.128 profile_example_with_cache.py:11(<listcomp>)
1 0.128 0.128 0.128 0.128 profile_example_with_cache.py:6(calc_big_number)
1 0.000 0.000 0.000 0.000 cProfile.py:117(__exit__)
1 0.000 0.000 0.000 0.000 {method â€ðisableâ€™ of â€˜lsprof.Profilerâ€™ ob
```

calc_big_number() only needs to be called once now and the program runs in 1.131 seconds rather than 4.278 seconds.

So why not cache everything? Some of the biggest cons of caching are:

1. If your cached result is subject to change, invalidating that cache can lead to more complex and bug-prone code. Itâ€™s frequently said that, â€œThere are only two hard things in computer science: cache invalidation and naming things.â€
2. Memory or disk space can be limited or more expensive.

Having said that, if your code is a candidate for caching, here are a few of the most common ways to implement it.

- Pythonâ€™s functools library has a variety of decorators like cache(), lru_cache(), and cached_property() that let you cache specific functions or properties.
- Redis is a versatile in-memory database that can be used as both a simple key-value or persistent, typed data store. Itâ€™s particularly useful if you want to share information across different processes or servers.
- Memcached is an older, memory-based key-value store. Redis has mostly supplanted it.
- DiskCache: A disk cache is usually slower than a memory cache, but it also usually costs less. You probably donâ€™t even need a third-party library for this.

Choose the right data structure

Another relatively simple optimization is to use the right data structure. Pythonâ€™s lists and dictionaries work well in most cases but sometimes thereâ€™s a better option. For example, if

you want to find an element in a large, unordered collection, a set is usually better than a list. This code compares searching for elements across different collections:

```
list: 5.0387
array: 11.6918
Counter: 0.0083
deque: 5.4151
dict: 0.0066
frozenset: 0.0064
ndarray: 0.6857
OrderedDict: 0.0069
set: 0.0063
tuple: 4.7424
```

All timings in this post are in seconds, and unless otherwise noted, all timings were run using Python 3.11.0 on a 2.8 GHz Quad-Core Intel Core i7 MacBook with 16GB of RAM. Python 3.11.0 speed was nearly identical to Python 3.9.16 for almost all runs.

Some assorted observations:

- As expected, hash table-based data structures like Counter, dict, frozenset, OrderedDict, and set performed the best.
- Using an immutable frozenset offered negligible performance benefits for this example. Tuples were a little faster than lists.
- NumPyâ€™s ndarray was much faster than arrays, lists, tuples, and deques despite not being hash table based, which reflects on the general performance benefits NumPy offers. More on that below.
- Pythonâ€™s array was much slower but it did better in other scenarios. More on that below, too.

The Big-O Algorithm Cheatsheet, also known as Know Thy Complexities, is a solid quick reference to help compare data structures. But you should also be skeptical of thy complexities as languages implement data structures differently.

For example, the closest analog to Pythonâ€™s list is an array, which typically has O(n) appends. But Pythonâ€™s array is actually a dynamic array and has O(1) appends per Python Wikiâ€™s Time Complexity article. This code demonstrates that:

```
deque_append: 0.0309
deque_appendleft: 0.0338
deque_insertleft: 0.0418
list_append: 0.0294
list_insertleft: 12.8794
set_add: 0.0441
```

Based on standard time complexities, youâ€™d think linked list-based deques would append faster than lists. However, Pythonâ€™s lists performed just as well or better. Deques are mostly useful for inserting at the beginning of the collection. The same applies to popping elements.

There are a ton of different data structures. Awareness of the following is sufficient for most problems: arrays, lists, linked lists, queues, stacks, hash tables, trees, and graphs. InterviewCakeâ€™s Data Structure Cheat Sheet covers most of them. Then once youâ€™ve mastered those, Wikipediaâ€™s list of data structures has hundreds of other niche structures.

Choose the right algorithm

If changing data structures doesnâ€™t help, there might be a better algorithm to use.

In the code above, searching through a list took 5.1 seconds because it required a worst-case full traversal of the list. But this wasnâ€™t necessary. Since the lists are sorted, a binary search can be used to avoid the full traversal. Hereâ€™s an example using Pythonâ€™s bisect function and NumPyâ€™s searchsorted:

```
list bisect: 0.0438
list in: 5.2920
list searchsorted: 56.5630
array bisect: 0.0656
array in: 12.2499
array searchsorted: 0.9532
deque bisect: 0.1194
deque in: 5.2470
deque searchsorted: 58.3833
ndarray uint64 bisect: 0.1305
ndarray uint64 in: 0.8032
ndarray uint64 searchsorted: 0.2084
ndarray uint16 bisect: 1.9034
ndarray uint16 in: 0.6267
ndarray uint16 searchsorted: 0.8782
tuple bisect: 0.0428
tuple in: 4.5129
tuple searchsorted: 54.0406
```

For some testing setups, ndarray searchsorted did outperform ndarray bisect and ndarray in.

list bisect (~0.05 seconds) was still slower than searching through a set (~0.008 seconds) but it was much closer. This example also illustrates how NumPy can actually slow down your code in some cases. Numpyâ€™s searchsorted was slower or negligible for all examples and uint16 vs. uint64 performance was surprising.

As with data structures, there are an endless number of algorithms to learn, but knowing a few core ones will help with most issues. Here are a few ways to get started:

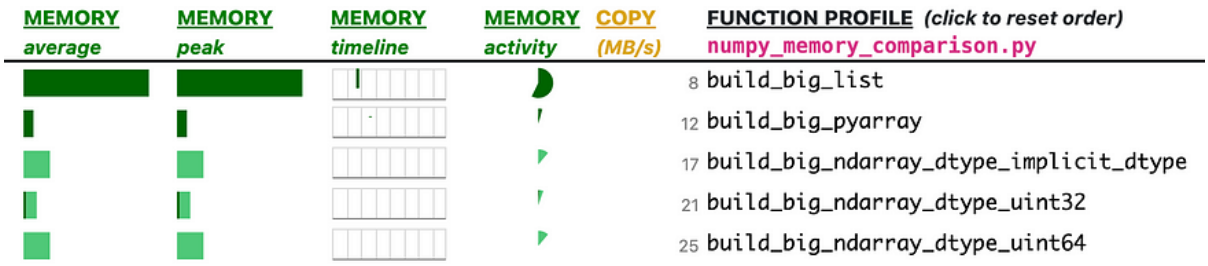
- Grokking Algorithms covers a lot of algorithm concepts in a very accessible way. Topics include sorting algorithms, recursion, search algorithms, dynamic programming, and greedy algorithms.
- TheAlgorithms is a GitHub repo you can clone to experiment with algorithms. Note that since these algorithms are written in pure Python, theyâ€™ll likely be less performant than standard library algorithms or third-party libraries like SciPy, but theyâ€™re great for learning.
- Introduction to Algorithms is far more comprehensive and detailed than Grokking Algorithms but also can be tougher to follow.

Use optimized libraries like NumPy, SciPy, and pandas

One of Pythonâ€™s strengths is there are third-party libraries to solve nearly every imaginable type of problem. A few of the most popular libraries for data science and machine learning are NumPy, SciPy, and pandas. All three have core code written in more performant languages like C, C++, and Fortran, which frequently results in more speed and less memory usage:

```
numpy.sum list: 2.8183
numpy.sum array: 0.0209
numpy.sum ndarray uint64: 0.0138
numpy.sum ndarray uint16: 0.0199
numpy.sum tuple: 2.7809
sum list: 0.2300
sum array: 0.8268
sum ndarray uint64: 2.8986
sum ndarray uint16: 5.6269
sum tuple: 0.2049
```

These results show that NumPyâ€™s sum function was significantly faster when it received arrays but also significantly slower when it received lists. Pythonâ€™s array performed well, too, and its memory usage was even better.



This memory graph was created by the Scalene profiler discussed earlier. It shows that building the list used 37.9MB of memory. On the other hand, the Python array only required 3.0 MB and a uint32 NumPy ndarray used 3.2 MB.

Faster summing is just the tip of the iceberg in terms of what these libraries offer. If youâ€™re thinking about writing a data structure or algorithm, thereâ€™s a good chance it has already been implemented performantly somewhere.

The biggest cons of using third-party libraries are somewhat illustrated above including:

- They can be slower when not used correctly. e.g. summing a NumPy array with Pythonâ€™s built-in sum function.
- They frequently require strict typing. For example, both Pythonâ€™s array and NumPyâ€™s arrays are typed, which can lead to runtime errors or subtle integer overflow issues.
- Thereâ€™s a greater chance of compatibility issues compared to using code from Pythonâ€™s standard library.

But if you keep all of that in mind, third-party libraries are a great option for many performance-critical codebases.

Leverage concurrency or parallelism

If your code is still slow, using concurrency or parallelism is another option. The simplest way to do that is via Pythonâ€™s multithreading, multiprocessing, or asyncio libraries. Which one you choose depends on what type of bottleneck youâ€™re dealing with:

- If youâ€™re waiting on a slow database query or network request, youâ€™re I/O-bound, and youâ€™ll see performance gains with any concurrency approach.
- If youâ€™re waiting on slow calculations or other CPU operations, youâ€™re CPU-bound, and you should likely use multiprocessing or distributed processing frameworks. Pythonâ€™s Global Interpreter Lock makes Python multithreading far less performant than most other languages.

Hereâ€™s an I/O-bound comparison of Pythonâ€™s three core libraries. Without concurrency, this code would take roughly 500 seconds to run.

```
asyncio: 1.0102
multithreading (4 max workers): 125.4517
multiprocessing (4 max workers): 125.5622
multithreading (8 max workers): 63.2284
multiprocessing (8 max workers): 63.4283
multithreading (500 max workers): 1.0470
multiprocessing (500 max workers): 7.5971
multithreading (5000 max workers): 1.0486
multiprocessing (5000 max workers): 8.8079
multithreading (50000 max workers): 1.0530
multiprocessing (50000 max workers): OSError: [Errno 22] Invalid argument
```

In this case, asyncio seems to be the best option due to the overhead of setting up pools as well as the nature of event loop concurrency. This becomes clearer when the example is scaled up from 500 to 50,000 seconds of sleep:

```
asyncio: 1.7381
multithreading (500 workers): 100.4519
multiprocessing (500 workers): 116.7062
multithreading (5000 max workers): RuntimeError: canâ€™t start new thread
multiprocessing (5000 max workers): (computer became unresponsive)
multithreading (50000 max workers): RuntimeError: canâ€™t start new thread
multiprocessing (50000 max workers): (I wouldnâ€™t recommend trying this)
```

Despite this massive difference, asyncio isnâ€™t always the best option. Hereâ€™s one of the reasons why:

```
asyncio: 50.1726
multithreading (50 max workers): 1.0102
```

asyncio struggled because itâ€™s single-threaded and there was slow, synchronous code blocking its event loop.

Think of asyncio as a really fast conveyor belt that can process thousands of items a second. But it can only do so if each item takes milliseconds or microseconds to complete. Each time.sleep() â€œitemâ€ took ~1 second to finish and dramatically slowed things down. The conveyor belt would have worked fine if each item used asyncio.sleep(), which runs in milliseconds or less.

This speaks to another challenge of asyncio, which is that all code must use the async and await keywords. If it doesnâ€™t, youâ€™re forced to rewrite code, find async libraries, or use async converters like sync_to_async(). Python async libraries are currently limited and as seen above, sync_to_async() can cause issues if the synchronous code is slow.

Both asyncio and multithreading share other challenges as well. In addition to issues around thread safety and race conditions, asyncio and multithreading donâ€™t perform well on CPU-bound tasks. Hereâ€™s an example using a quad-core processor:

```
asyncio: 6.0348
multithreading (8 max workers): 6.0127
multiprocessing (8 max workers): 1.6724
multithreading (50 max workers): 6.0043
multiprocessing (50 max workers): 2.8025
```

And hereâ€™s what it looks like when 1_000_000 is changed to 2_000_000:

```
asyncio: 18.1429
multithreading (8 max workers): 18.0095
multiprocessing (8 max workers): 4.9686
multithreading (50 max workers): 18.0477
multiprocessing (50 max workers): 6.1049
```

As expected, multiprocessing performed much better since itâ€™s able to leverage all 4 CPUs without GIL blocking. 50 workers being slower also illustrates that more workers arenâ€™t necessarily better.

One of the biggest cons of multiprocessing is that each process might have a huge memory footprint for large codebases. It also only scales relative to the number of CPUs a machine has. If youâ€™re looking for more concurrency, a distributed processing framework like Apache Spark or Apache Hadoop might be a better choice.

Python also has a few other third-party concurrency libraries with slightly different approaches that might be useful in some scenarios:

- gevent uses green threads. Unlike asyncio, it offers monkey patching for code that wasnâ€™t written using async and await.
- Trio is based on a new way of thinking they call â€œstructured concurrencyâ€.
- Pykka is a Python implementation of the actor model of concurrency.

Use a different Python implementation

CPython, the standard implementation of Python, is great for almost all use cases, and you should stick with it whenever possible since itâ€™s widely supported, maintained, and compatible. However, there are times when a different implementation can produce much faster results.

Hereâ€™s an example of code that does a bunch of addition and subtraction:

```
Run 1: 5.9440
Run 2: 5.9336
Run 3: 5.8404
Run 4: 5.6842
```

CPython consistently ran in ~6 seconds using Python 3.11 and ~7 seconds for Python 3.9.16.

PyPy is one of the most popular and compatible alternatives to CPython. Itâ€™s also often faster thanks to its JIT compiler. Hereâ€™s how the same code does using bin/pypy (Python 3.7) rather than bin/python:

```
Run 1: 0.0818
Run 2: 0.0784
Run 3: 0.0788
Run 4: 0.0842
```

PyPy was around 75x faster but it might not be for other examples. Additionally, while PyPy is one of the most compatible non-standard Python implementations, it still has some differences, which are listed on its website. PyPy can also use more memory and be a few versions behind Pythonâ€™s latest release.

Other popular Python implementations include RustPython for Rust, Jython for Java, and IronPython for C#. There are even a few projects trying to improve standard CPython including Metaâ€™s Cinder and Microsoftâ€™s faster-cpython. Python founder Guido Van Rossum also recently gave a shout-out to nogil, which is an experimental version of Python without a Global Interpreter Lock.

There are way too many implementations to list here but Pythonâ€™s wiki has a comprehensive list and Toptal has a solid comparison of implementations.

Use a compiler like Numba or Cython

CPython is an interpreted language, which offers benefits like dynamic typing, platform independence, and rapid prototyping. Performance is not one of those benefits. Numba and Cython are two different ways to compile Python and potentially improve performance.

Numba is a JIT compiler thatâ€™s similar to what PyPy uses. The difference is it requires changes to your code. Hereâ€™s the same code from earlier using Numbaâ€™s decorator (also using Python 3.9.16):

```
Run 1: 0.000006127
Run 2: 0.000000611
Run 3: 0.000000237
Run 4: 0.000000197
```

Thatâ€™s way faster than both CPython (~6 seconds) and PyPy (~0.08 seconds). Even using the slowest time of 0.000006127 represents a 99.9999% decrease. Itâ€™s so much faster that it feels like thereâ€™s a bug, a layer of caching, or a compiler optimization for the redundant math.

For what itâ€™s worth, adding size as a parameter to add_and_subtract() produced similar times. Removing the subtraction line, which caused integer overflow, was also very fast. Either way, even if this is an authentic performance gain, Numba still does have other risks.

Take note of uint32 in the decorator. This improves speed but can lead to issues if grand_total is larger than a 32-bit unsigned integer. Also, like other performance options, it might not improve speed, might have compatibility issues, and might be a few versions behind Pythonâ€™s latest release.

Cython is an optimizing static compiler rather than a JIT compiler. Its core goals are around interoperability with C and providing C-like performance for Python. Hereâ€™s an example:

```
Run 1: 0.0263
Run 2: 0.0265
Run 3: 0.0268
Run 4: 0.0274
```

Cython outperforms everything except Numba, but there are potentially other approaches that match Numbaâ€™s performance. Cython supports C/C++, parallelism, and multiple different syntax options. This ambiguity speaks to some of the cons of Cython. It has a steeper learning curve, requires more code changes, and needs a build step before running.

Write an extension in C, C++, Rust, or other languages

While Numba and Cython can give you C-like performance, If you truly want the performance of other languages, an extension is probably your best bet. Here are a few solid tutorials:

- Extending Python with C or C++
- Building a Python C Extension Module
- PyO3 User Guide (Rust Bindings for Python)

Keep up to date

Python and the Python ecosystem are constantly evolving, and there are always new libraries or features to learn about. Here are a few good ways to keep up:

- awesome-python is an organized collection of links for nearly every Python library or tool you could possibly need.
- Real Python has a lot of well-written Python tutorials and regularly posts about Python news.
- r/Python is a community of 1,000,000+ Python devs.

Finally, if thereâ€™s anything to take away from this post, donâ€™t try to launch a process pool with 5,000 workers.