What Is the Python Global Interpreter Lock (GIL)?

by Abhinav Ajitsaria

advanced python

Mark as Completed

Share

Table of Contents

Remove ads

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Understanding Python's Global Interpreter Lock (GIL)**

The Python Global Interpreter Lock or GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter.

This means that only one thread can be in a state of execution at any point in time. The impact of the GIL isn't visible to developers who execute single-threaded programs, but it can be a performance bottleneck in CPU-bound and multi-threaded code.

Since the GIL allows only one thread to execute at a time even in a multi-threaded architecture with more than one CPU core, the GIL has gained a reputation as an "infamous" feature of Python.

In this article you'll learn how the GIL affects the performance of your Python programs, and how you can mitigate the impact it might have on your code.

What Problem Did the GIL Solve for Python?

— FREE Email Series —

Python Tricks 💌

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email...

Get Python Tricks »

🔒 No spam. Unsubscribe any time.

Browse Topics
Guided Learning Paths
Basics Intermediate Advanced

api best-practices career community databases data-science data-structures data-viz devops django docker editors flask front-end gamedev gui machine-learning numpy projects python testing tools web-dev web-scraping

Table of Contents

- What Problem Did the GIL Solve for Python?
- Why Was the GIL Chosen as the Solution?
- The Impact on Multi-Threaded Python Programs
- Why Hasn't the GIL Been Removed Yet?
- Why Wasn't It Removed in Python 3?
- How to Deal With Python's GIL

Mark as Completed

Share

Python uses reference counting for memory management. It means that objects created in Python have a reference count variable that keeps track of the number of references that point to the object. When this count reaches zero, the memory occupied by the object is released.

Recommended Video Course
Understanding Python's Global Interpreter Lock (GIL)

Let's take a look at a brief code example to demonstrate how reference counting works:

Python

```
>>> import sys
>>> a = []
>>> b = a
>>> sys.getrefcount(a)
3
```

In the above example, the reference count for the empty list object `[]` was 3. The list object was referenced by `a`, `b` and the argument passed to `sys.getrefcount()`.

Back to the GIL:

The problem was that this reference count variable needed protection from race conditions where two threads increase or decrease its value simultaneously. If this happens, it can cause either leaked memory that is never released or, even worse, incorrectly release the memory while a reference to that object still exists. This can cause crashes or other "weird" bugs in your Python programs.

This reference count variable can be kept safe by adding *locks* to all data structures that are shared across threads so that they are not modified inconsistently.

But adding a lock to each object or groups of objects means multiple locks will exist which can cause another problem—Deadlocks (deadlocks can only happen if there is more than one lock). Another side effect would be decreased performance caused by the repeated acquisition and release of locks.

The GIL is a single lock on the interpreter itself which adds a rule that execution of any Python bytecode requires acquiring the interpreter lock. This prevents deadlocks (as there is only one lock) and doesn't introduce much performance overhead. But it effectively makes any CPU-bound Python program single-threaded.

The GIL, although used by interpreters for other languages like Ruby, is not the only solution to this problem. Some languages avoid the requirement of a GIL for thread-safe memory management by using approaches other than reference counting, such as garbage collection.

On the other hand, this means that those languages often have to compensate for the loss of single threaded performance benefits of a GIL by adding other performance boosting features like JIT compilers.

Remove ads

Why Was the GIL Chosen as the Solution?

So, why was an approach that is seemingly so obstructing used in Python? Was it a bad decision by the developers of Python?

Well, in the words of Larry Hastings, the design decision of the GIL is one of the things that made Python as popular as it is today.

Python has been around since the days when operating systems did not have a concept of threads. Python was designed to be easy-to-use in order to make development quicker and more and more developers started using it.

A lot of extensions were being written for the existing C libraries whose features were needed in Python. To prevent inconsistent changes, these C extensions required a thread-safe memory management which the GIL provided.

The GIL is simple to implement and was easily added to Python. It provides a performance increase to single-threaded programs as only one lock needs to be managed.

C libraries that were not thread-safe became easier to integrate. And these C extensions became one of the reasons why Python was readily adopted by different communities.

As you can see, the GIL was a pragmatic solution to a difficult problem that the CPython developers faced early on in Python's life.

The Impact on Multi-Threaded Python Programs

When you look at a typical Python program—or any computer program for that matter—there's a difference between those that are CPU-bound in their performance and those that are I/O-bound.

CPU-bound programs are those that are pushing the CPU to its limit. This includes programs that do mathematical computations like matrix multiplications, searching, image processing, etc.

I/O-bound programs are the ones that spend time waiting for Input/Output which can come from a user, file, database, network, etc. I/O-bound programs sometimes have to wait for a significant amount of time till they get what they need from the source due to the fact that the source may need to do its own processing before the input/output is ready, for example, a user thinking about what to enter into an input prompt or a database query running in its own process.

Let's have a look at a simple CPU-bound program that performs a countdown:

Python single_threaded.py

```python
import time
from threading import Thread

COUNT = 50000000

def countdown(n):
    while n>0:
        n -= 1

start = time.time()
countdown(COUNT)
end = time.time()

print('Time taken in seconds -', end - start)
```

Running this code on my system with 4 cores gave the following output:

Shell

```
$ python single_threaded.py
Time taken in seconds - 6.20024037361145
```

Now I modified the code a bit to do to the same countdown using two threads in parallel:

Python multi_threaded.py

```
import time
from threading import Thread

COUNT = 50000000

def countdown(n):
    while n>0:
        n -= 1

t1 = Thread(target=countdown, args=(COUNT//2,))
t2 = Thread(target=countdown, args=(COUNT//2,))

start = time.time()
t1.start()
t2.start()
t1.join()
t2.join()
end = time.time()

print('Time taken in seconds -', end - start)
```

And when I ran it again:

Shell

```
$ python multi_threaded.py
Time taken in seconds - 6.924342632293701
```

As you can see, both versions take almost same amount of time to finish. In the multi-threaded version the GIL prevented the CPU-bound threads from executing in parallel.

The GIL does not have much impact on the performance of I/O-bound multi-threaded programs as the lock is shared between threads while they are waiting for I/O.

But a program whose threads are entirely CPU-bound, e.g., a program that processes an image in parts using threads, would not only become single threaded due to the lock but will also see an increase in execution time, as seen in the above example, in comparison to a scenario where it was written to be entirely single-threaded.

This increase is the result of acquire and release overheads added by the lock.

Remove ads

Why Hasn’t the GIL Been Removed Yet?

The developers of Python receive a lot of complaints regarding this but a language as popular as Python cannot bring a change as significant as the removal of GIL without causing backward incompatibility issues.

The GIL can obviously be removed and this has been done multiple times in the past by the developers and researchers but all those attempts broke the existing C extensions which depend heavily on the solution that the GIL provides.

Of course, there are other solutions to the problem that the GIL solves but some of them decrease the performance of single-threaded and multi-threaded I/O-bound programs and some of them are just too difficult. After all, you wouldn’t want your existing Python programs to run slower after a new version comes out, right?

The creator and BDFL of Python, Guido van Rossum, gave an answer to the community in September 2007 in his article “It isn’t Easy to remove the GIL”:

> "I'd welcome a set of patches into Py3k *only if* the performance for a single-threaded program (and for a multi-threaded but I/O-bound program) *does not decrease*"

And this condition hasn't been fulfilled by any of the attempts made since.

Why Wasn't It Removed in Python 3?

Python 3 did have a chance to start a lot of features from scratch and in the process, broke some of the existing C extensions which then required changes to be updated and ported to work with Python 3. This was the reason why the early versions of Python 3 saw slower adoption by the community.

But why wasn't GIL removed alongside?

Removing the GIL would have made Python 3 slower in comparison to Python 2 in single-threaded performance and you can imagine what that would have resulted in. You can't argue with the single-threaded performance benefits of the GIL. So the result is that Python 3 still has the GIL.

But Python 3 did bring a major improvement to the existing GIL—

We discussed the impact of GIL on "only CPU-bound" and "only I/O-bound" multi-threaded programs but what about the programs where some threads are I/O-bound and some are CPU-bound?

In such programs, Python's GIL was known to starve the I/O-bound threads by not giving them a chance to acquire the GIL from CPU-bound threads.

This was because of a mechanism built into Python that forced threads to release the GIL **after a fixed interval** of continuous use and if nobody else acquired the GIL, the same thread could continue its use.

Python

```
>>> import sys
>>> # The interval is set to 100 instructions:
>>> sys.getcheckinterval()
100
```

The problem in this mechanism was that most of the time the CPU-bound thread would reacquire the GIL itself before other threads could acquire it. This was researched by David Beazley and visualizations can be found here.

This problem was fixed in Python 3.2 in 2009 by Antoine Pitrou who added a mechanism of looking at the number of GIL acquisition requests by other threads that got dropped and not allowing the current thread to reacquire GIL before other threads got a chance to run.

How to Deal With Python's GIL

If the GIL is causing you problems, here a few approaches you can try:

Multi-processing vs multi-threading: The most popular way is to use a multi-processing approach where you use multiple processes instead of threads. Each Python process gets its own Python interpreter and memory space so the GIL won't be a problem. Python has a `multiprocessing` module which lets us create processes easily like this:

Python `multiprocess.py`

```
from multiprocessing import Pool
import time

COUNT = 50000000
def countdown(n):
    while n>0:
        n -= 1

if __name__ == '__main__':
    pool = Pool(processes=2)
    start = time.time()
    r1 = pool.apply_async(countdown, [COUNT//2])
    r2 = pool.apply_async(countdown, [COUNT//2])
    pool.close()
    pool.join()
    end = time.time()
    print('Time taken in seconds -', end - start)
```

Running this on my system gave this output:

Shell

```
$ python multiprocess.py
Time taken in seconds - 4.060242414474487
```

A decent performance increase compared to the multi-threaded version, right?

The time didn't drop to half of what we saw above because process management has its own overheads. Multiple processes are heavier than multiple threads, so, keep in mind that this could become a scaling bottleneck.

Alternative Python interpreters: Python has multiple interpreter implementations. CPython, Jython, IronPython and PyPy, written in C, Java, C# and Python respectively, are the most popular ones. GIL exists only in the original Python implementation that is CPython. If your program, with its libraries, is available for one of the other implementations then you can try them out as well.

Just wait it out: While many Python users take advantage of the single-threaded performance benefits of GIL. The multi-threading programmers don't have to fret as some of the brightest minds in the Python community are working to remove the GIL from CPython. One such attempt is known as the Gilectomy.

The Python GIL is often regarded as a mysterious and difficult topic. But keep in mind that as a Pythonista you're usually only affected by it if you are writing C extensions or if you're using CPU-bound multi-threading in your programs.

In that case, this article should give you everything you need to understand what the GIL is and how to deal with it in your own projects. And if you want to understand the low-level inner workings of GIL, I'd recommend you watch the Understanding the Python GIL talk by David Beazley.

Mark as Completed Share

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Understanding Python's Global Interpreter Lock (GIL)**

Python Tricks 💌

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

About **Abhinav Ajitsaria**

Abhinav is a Software Engineer from India. He loves to talk about system design, machine learning, AWS and of course, Python.

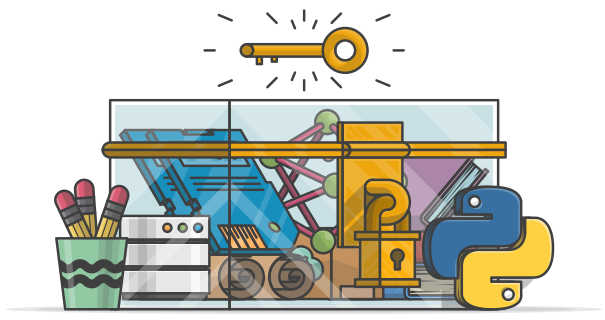
» More about Abhinav

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren

Dan

Master Real-World Python Skills
With Unlimited Access to Real Python



Join us and get access to thousands of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Rate this article:

LinkedIn Twitter Bluesky Facebook Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Topics: **advanced** **python**

Recommended Video Course: Understanding Python's Global Interpreter Lock (GIL)

Related Tutorials:

- An Intro to Threading in Python
- Speed Up Your Python Program With Concurrency
- Async IO in Python: A Complete Walkthrough
- Memory Management in Python
- Primer on Python Decorators

Remove ads

© 2012–2025 Real Python · Newsletter · Podcast · YouTube · Twitter · Facebook · Instagram · Python Tutorials · Search · Privacy Policy · Energy Policy · Advertise · Contact

Happy Pythoning!