# Working around the Global Interpreter Lock (GIL)

All Versions

## Examples

### Multiprocessing.Pool

The simple answer, when asking how to use threads in Python is: "Don't. Use processes, instead." The multiprocessing module lets you create processes with similar syntax to creating threads, but I prefer using their convenient Pool object.

Using the code that David Beazley first used to show the dangers of threads against the GIL , we'll rewrite it using multiprocessing.Pool :

#### David Beazley's code that showed GIL threading problems

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

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 end-start
```

Re-written using multiprocessing.Pool:

```
import multiprocessing
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

start = time.time()
with multiprocessing.Pool as pool:
    pool.map(countdown, [COUNT/2, COUNT/2])

    pool.close()
    pool.join()

end = time.time()
print(end-start)
```

Instead of creating threads, this creates new processes. Since each process is its own interpreter, there are no GIL collisions. multiprocessing.Pool will open as many processes as there are cores on the machine, though in the example above, it would only need two. In a real-world scenario, you want to design your list to have at least as much length as there are processors on your machine. The Pool will run the function you tell it to run with each argument, up to the number of processes it creates. When the function finishes, any remaining functions in the list will be run on that process.

I've found that, even using the `with` statement, if you don't close and join the pool, the processes continue to exist. To clean up resources, I always close and join my pools.

### Cython nogil:

Cython is an alternative python interpreter. It uses the GIL, but lets you disable it. See their documentation

As an example, using the code that David Beazley first used to show the dangers of threads against the GIL , we'll rewrite it using nogil:

#### David Beazley's code that showed GIL threading problems

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

t1 = Thread(target=countdown,args=(COUNT/2,))
t2 = 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 end-start
```

### Re-written using nogil (ONLY WORKS IN CYTHON):

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

with nogil:
    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 end-start
```

It's that simple, as long as you're using cython. Note that the documentation says you must make sure not to change any python objects:

> Code in the body of the statement must not manipulate Python objects in any way, and must not call anything that manipulates Python objects without first re-acquiring the GIL. Cython currently does not check this.

## Syntax

## Parameters

## Remarks

### Why is there a GIL?

The GIL has been around in CPython since the inception of Python threads, in 1992. It's designed to ensure thread safety of running python code. Python interpreters written with a GIL prevent multiple native threads from executing Python bytecodes at once. This makes it easy for plugins to ensure that their code is thread-safe: simply lock the GIL, and only your active thread is able to run, so your code is automatically thread-safe.
Short version: the GIL ensures that no matter how many processors and threads you have, *only one thread of a python interpreter will run at one time.*

This has a lot of ease-of-use benefits, but also has a lot of negative benefits as well.

Note that a GIL is not a requirment of the Python language. Consequently, you can't access the GIL directly from standard python code. Not all implementations of Python use a GIL.

**Interpreters that have a GIL:** CPython, PyPy, Cython (but you can disable the GIL with `nogil`)

**Interpreters that do not have a GIL:** Jython, IronPython

### Details on how the GIL operates:

When a thread is running, it locks the GIL. When a thread wants to run, it requests the GIL, and waits until it is available. In CPython, before version 3.2, the running thread would check after a certain number of python instructions to see if other code wanted the lock (that is, it released the lock and then requested it again). This method tended to cause thread starvation, largely because the thread that released the lock would acquire it again before the waiting threads had a chance to wake up. Since 3.2, threads that want the GIL wait for the lock for some time, and after that time, they set a shared variable that forces the running thread to yield. This can still result in drastically longer execution times, though. See the links below from dabeaz.com (in the references section) for more details.

CPython automatically releases the GIL when a thread performs an I/O operation. Image processing libraries and numpy number crunching operations release the GIL before doing their processing.

### Benefits of the GIL

For interpreters that use the GIL, the GIL is systemic. It is used to preserve the state of the application. Benefits include:

- Garbage collection - thread-safe reference counts must be modified while the GIL is locked. *In CPython, all of garbarge collection is tied to the GIL.* This is a big one; see the python.org wiki article about the GIL (listed in References, below) for details about what must still be functional if one wanted to remove the GIL.
- Ease for programmers dealing with the GIL - locking everything is simplistic, but easy to code to
- Eases the import of modules from other languages

## Consequences of the GIL

The GIL only allows one thread to run python code at a time inside the python interpreter. This means that multithreading of processes that run strict python code simply doesn't work. When using threads against the GIL, you will likely have worse performance with the threads than if you ran in a single thread.

## References:

https://wiki.python.org/moin/GlobalInterpreterLock - quick summary of what it does, fine details on all the benefits

http://programmers.stackexchange.com/questions/186889/why-was-python-written-with-the-gil - clearly written summary

http://www.dabeaz.com/python/UnderstandingGIL.pdf - how the GIL works and why it slows down on multiple cores

http://www.dabeaz.com/GIL/gilvis/index.html - visualization of the data showing how the GIL locks up threads

http://jeffknupp.com/blog/2012/03/31/pythons-hardest-problem/ - simple to understand history of the GIL problem

https://jeffknupp.com/blog/2013/06/30/pythons-hardest-problem-revisited/ - details on ways to work around the GIL's limitations