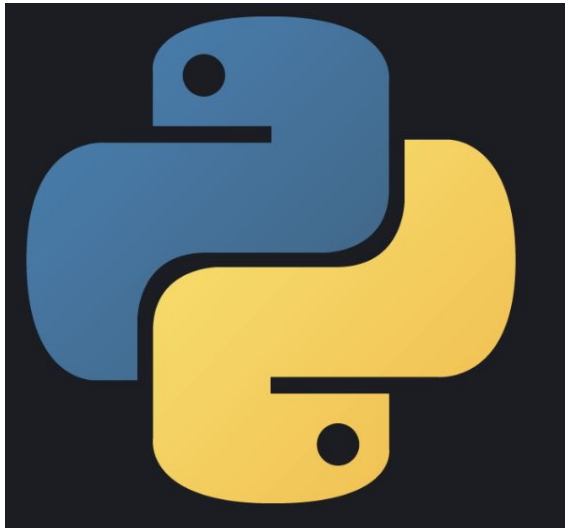


Threads in Python

Intro to Threads

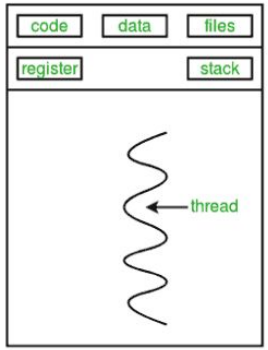
A thread is a separate flow of execution.

This means that your program will have two things happening at once.

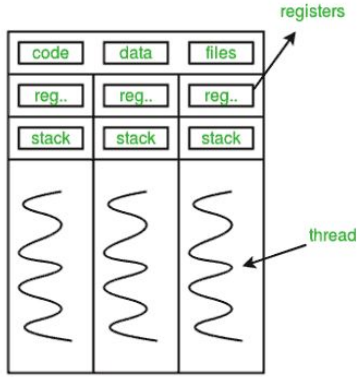
But for most Python 3 implementations the different threads do not actually execute at the same time: **they merely appear to.**

It's tempting to think of threading as having two (or more) different processors running on your program, each one doing an independent task at the same time.

That's almost right. The threads may be running on different processors, but they will only be running one at a time.



single-threaded process



multithreaded process

Intro to Threads - GIL

Getting multiple tasks running simultaneously requires a non-standard implementation of Python.

Or you can use `multiprocessing` which comes with some extra overhead.

The issue with multithread in python is due to interactions with the **Global Interpreter Lock (GIL)** that essentially limit one Python thread to run at a time.

Tasks that spend much of their time waiting for external events **are generally good candidates for threading.**

Problems that require heavy CPU computation and spend little time waiting for external events might not run faster at all.

The Impact on Multi-Threaded Python Programs

When you look at a typical Python program 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.

Example: a user thinking about what to enter into an input prompt or a database query running in its own process.

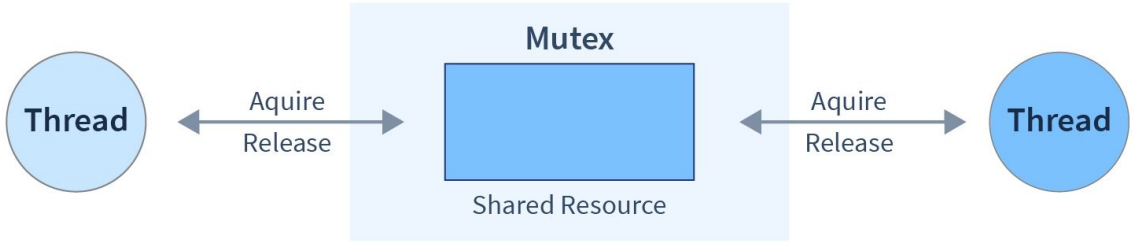
Intro to Threads - What is the GIL?

The Python Global Interpreter Lock or GIL, in simple words, is 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.



Intro to Threads - What are the issues that GIL solves?

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.

```
>>> 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().

Intro to Threads - What are the issues that GIL solves?

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**.

Solution 1: 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.

Intro to Threads - What are the issues that GIL solves?

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**.

Solution 1: 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.

Problem 1: Adding a lock to each object or groups of objects means multiple locks will exist which can cause another problem, namely, **Deadlocks**.

Another side effect would be decreased performance caused by the repeated acquisition and release of locks

Intro to Threads - What are the issues that GIL solves?

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**.

Solution 2: 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.

Intro to Threads - Starting a thread

Now that you've got an idea of what a thread is, let's learn how to make one.

The Python standard library provides `threading`

To start a separate thread, you create a `Thread` instance and then tell it to `.start()`

```python
import threading
import time

def thread_function(name):
    print(f"Thread {name}: starting at {time.time()}")
    time.sleep(2)
    print(f"Thread {name}: finishing at {time.time()}")

if __name__ == "__main__":

    x = threading.Thread(target=thread_function, args=(1,))
    print(f"Main    : before running thread")
    x.start()
    print(f"Main    : wait for the thread to finish")
    # x.join()
    print(f"Main    : all done")
```

Intro to Threads - Starting a thread

When you create a `Thread`, you pass it a function and a tuple containing the arguments to that function.

In this case, you're telling the `Thread` to run `thread_function()` and to pass it `1` as an argument.

```python
import threading
import time

def thread_function(name):
    print(f"Thread {name}: starting at {time.time()}")
    time.sleep(2)
    print(f"Thread {name}: finishing at {time.time()}")

if __name__ == "__main__":

    x = threading.Thread(target=thread_function, args=(1,))
    print(f"Main    : before running thread")
    x.start()
    print(f"Main    : wait for the thread to finish")
    # x.join()
    print(f"Main    : all done")
```

Intro to Threads - Many threads

The example code so far has only been working with two threads: the main thread and one you started with the `threading.Thread` object.

Frequently, you'll want to start a number of threads and have them do interesting work.

```
threads = list()
for index in range(3):
    print(f"Main    : before running thread")
    x = threading.Thread(target=thread_function, args=(index,))
    threads.append(x)
    x.start()

for index, thread in enumerate(threads):
    print(f"Main    : wait for the thread to finish")
    thread.join()
    print(f"Main    : all done")
```

Intro to Threads - Using a ThreadPoolExecutor

There's an easier way to start up a group of threads than the one you saw above.

It's called a `ThreadPoolExecutor`, and it's part of the standard library in `concurrent.futures` (as of Python 3.2).

The easiest way to create it is as a context manager, using the `with` statement to manage the creation and destruction of the pool.

```
with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
    executor.map(thread_function, range(3))
```

Intro to Threads - Using a ThreadPoolExecutor

The code **creates** a `ThreadPoolExecutor` as a **context manager**, telling it how many worker threads it wants in the pool.

It then uses `.map()` to step through an iterable of things, in your case `range(3)`, passing each one to a thread in the pool.

The end of the `with` block causes the `ThreadPoolExecutor` to do a `.join()` on each of the threads in the pool. **It is *strongly* recommended that you use `ThreadPoolExecutor` as a context manager when you can so that you never forget to `.join()` the threads**

```
with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
    executor.map(thread_function, range(3))
```