**Explain the limitations of the python Global Interpreter Lock, and how you could workaround it.**

Accompany code for this question is included in the file GIL\_Example.py

**Limitations and Example**

Python’s Global Interpreter Lock or GIL limits traditional multithreading capabilities of a program by not allowing parallel execution of code, even when functions are allocated to different threads.   
Popular demonstrations of this usually involve timing a function that increments a counter to a large number such as 100,000,000. To explain the limitations, I will focus on this type of example.  
  
Consider the following function in python:

num = 0  
def count(num):  
 while n<100000000:  
 num+= 1

First off, the function will be run normally (single thread) but timed using python built in ‘time’ module.  
We set a start time, then run the count() function, then set the end time.  
Finally we can just print the delta of the start and end times.

start\_time = time.time()

count(num)

end\_time = time.time()

delta = end\_time - start\_time

print("Runtime in standard 1 thread iteration: {} seconds".format(delta))

The output on my machine looks like this:

Runtime in standard 1 thread iteration: 3.051983594894409 seconds

Now to test how the GIL is creating the limitation in multithreading, by spreading our function over 2 threads. We can use Pythons built in threading module for this using the Thread() function of that module.

Since we are using two threads to distribute the workload we need to remember that each thread needs to do just half of the workload. So, we can start the count at half of 100,000,000 which is 50,000,000. (looking back I probably should have made this a decrement function ;/ )

first\_thread = Thread(target=count, args=(50000000)

second\_thread = Thread(target=count, args=(50000000)

start\_time = time.time()

first\_thread.start()

second\_thread.start()

first\_thread.join()

second\_thread.join()

end\_time = time.time()

delta = end\_time - start\_time

print("Runtime in multiple threads: {} seconds".format(delta))

This yields the output:

Runtime in standard 1 thread iteration: 3.0945663452148438 seconds

Runtime in multiple threads: 3.0012619495391846 seconds

As you can see the runtime between the single thread and the multi-threaded functions are virtually the same. This is because of Python’s Global Interpreter Lock. This makes Python very safe to execute, and great for single threaded applications, but terrible if we want to actually utilize multiprocessor CPU’s (hint: we do sometimes!)

**Workaround – Multiprocessing**

Pythons’ multiprocessing library is the key to working around the GIL.  
The python multiprocessing library contains many ways to manipulate threads, queues, and processes in order to get them to run more efficiently.  
I have chosen as a quick example to use the JoinableQueue Class and the Process Class to demonstrate this. I have chosen this method because this is a workaround I have personally used before when needing to run a routine on hundreds of devices in a short amount of time.  
In this example instead of creating a thread for our now halved work I will create a JoinableQueue and then spin up a Process that will utilize the arguments in the Queue in conjunction with our countdown method. It looks like this:

start\_time = time.time()

joinable\_queue = JoinableQueue()

joinable\_queue.put([50000000,50000000])

first\_thread = Process(target=countdown, args=(joinable\_queue,))

end\_time = time.time()

delta = end\_time - start\_time

print("Runtime in multiprocessing: {} seconds".format(delta))

This produced the following output on my machine:

Runtime in standard 1 thread iteration: 3.1285929679870605 seconds

Runtime in multiple threads: 2.9962193965911865 seconds

Runtime in multiprocessing: 0.019400835037231445 seconds

As you can see the time is considerably shorter using multiprocessing.  
This is because when we use multiprocessing each of the objects in the JoinableQueue gets its own python interpreter to use.