

Forum:[Hack Forums](#)

Thread: [Python Multi-threading Tutorial](#)

Gazo - July 14, 2019, 7:31 pm

Looking through some of the posts here, it seems that people could benefit from a modern Python multi-threading tutorial.

For those who don't know, threading is a way to have your program do multiple things at once. Take for example you are trying to process a massive text file. You need to run a function that processes each line of the text file individually. You could write a program that looks like the following:

```
Code def doStuff(line):  
    # Do stuff to line  
    return line
```

```
with open("big_test_file.txt") as btf:  
    for line in btf:  
        doStuff(line)
```

In a lot of cases this is probably good enough, but as that text file becomes very large your program will become slower and slower because of the need to process every line individually and wait for the line to finish before moving on to the next one.

This is where threading comes in. Threading will allow you to process X number of lines at a time in parallel. This means that you can process a whole lot more data much quicker.

Fortunately for us, Python 3 provides some very useful libraries for handling Threads and making sure that everything starts and stops as intended.

Since threads are spun up and run asynchronously we will need to add a bit of extra code that will allow us to control the number of threads active at a given time, and also monitor all of our active threads and wait for them all to finish before allowing our program to close.

To do this we will use something called a "Thread Pool".

To help you to understand what a thread pool is a why it is useful, I will use the following analogy.

Image a busy bar at peek hours. There is a huge line out the door of people who are trying to get in to get a drink. There are only 5 or 6 bartenders on staff tonight and it was agreed before hand that there can only be a max of 50 people at the bar at a time otherwise the bartenders will become too overwhelmed to server everyone. The bouncer's job outside is to control the flow of people into the bar. He constantly monitors the number of people at the bar and only allows another person in to be served when the number of people at the bar is less than 50.

This helps to illustrate the problem that we face with multi threading. If we spin up an unlimited amount of threads, then we won't have the resources to serve all the function calls. This will result in errors and information not being processed correctly. Just to draw a direct comparison, the bar tenders are analogous to the function being run on each line. The customers are the number of threads and the bouncer is the pool manager. He is managing a pool of 50 people (threads) and

never letting there be any more than 50 people (active threads) be served at any one time.

Now that you know what a thread pool is and why it is important, let's rewrite the above code using a thread pool.

Code from `concurrent.futures` import `ThreadPoolExecutor`

```
def doStuff(line):
    # Do stuff to line
    return line

with open("big_test_file.txt") as btf:
    thread_pool = []
    executor = ThreadPoolExecutor(max_workers=10)
    for line in btf:
        thread = executor.submit(doStuff, line)
        thread_pool.append(thread)

    for active_thread in thread_pool:
        while not active_thread.done():
            pass
        print(active_thread.result())
```

As you can see, we didn't have to change much. All that really changed is how we are now calling `doStuff()`

We import a class called `ThreadPoolExecutor` that will handle the thread pool for us. Then instead of calling `doStuff()` directly we pass the function and the arguments to a method called "`submit()`" in the executor object. This adds a new thread to the pool that will be executed when < 5 threads are active.

The final for loop allows us to check for when a thread is completed. We iterate over the existing threads in the pool (`thread_pool`) and check if it is done using the `.done()` method. While the thread has not completed we wait. When the while condition is no longer satisfied we print the results and print the results of that thread.

And that's it! This is an incredibly useful tool to have in your tool belt. It took me a while to fully understand but once I did I was able to use it everywhere. Some common issues people run into is synchronization of information and excess resource consumption. Managing threads and determine why they failed and when is also a common challenge. That is all for another post though.

I hope this helped! If you have any more questions feel free to ask. I am happy to help.

vorticalbox - July 15, 2019, 3:43 am

problem is the last loop still has to wait for each thread in turn to finish.

would it not be better to have 2 threads one to read lines and the other to consume them?

note: how do I embed paste bins -.-

Gazo - July 15, 2019, 4:41 am

(Jul 15, 2019 03:43 AM)vorticalbox Wrote: [\[CITE\]](#) problem is the last loop still has to wait for each

thread in turn to finish.

would it not be better to have 2 threads one to read lines and the other to consume them?

note: how do I embed paste bins -.-

I am not sure how to embed a pastebin link inside of the thread. I just used the code tags ?

You do make a good point though. There are a few different ways to do this depending on the situation. If you just need to run a bunch of jobs in parallel without caring too much about the results (scrape and insert data into a DB for example), then it doesn't really make much sense to also create consumer threads. However, if you need to also monitor the results of the thread then using a consumer will get you the results as soon as the thread is finished (like in the example I made in the post).

This is definitely a good point that I didn't think to make, thank you.

vorticalbox - July 15, 2019, 5:15 am

(Jul 15, 2019 04:41 AM)Gazo Wrote: [\[CITE\]](#) I am not sure how to embed a pastebin link inside of the thread. I just used the code tags ?

You do make a good point though. There are a few different ways to do this depending on the situation. If you just need to run a bunch of jobs in parallel without caring too much about the results (scrape and insert data into a DB for example), then it doesn't really make much sense to also create consumer threads. However, if you need to also monitor the results of the thread then using a consumer will get you the results as soon as the thread is finished (like in the example I made in the post).

This is definitely a good point that I didn't think to make, thank you.

Thanks only thought about this because I was messing about monitoring messages in a real-time chat, I have one thread listening for messages and then other consuming messages as they came in and in that situation, the producer never ends having things to process,

Gazo - July 15, 2019, 5:19 am

(Jul 15, 2019 05:15 AM)vorticalbox Wrote: [\[CITE\]](#) Thanks only thought about this because I was messing about monitoring messages in a real-time chat, I have one thread listening for messages and then other consuming messages as they came in and in that situation, the producer never ends having things to process,

Yeah, that's an appropriate use case for multiple threads.

The only weird thing I can see with your code is what if the worker thread is slower than the consumer thread?

Code while not q.empty() could evaluate to False causing the consumer thread to close if the worker thread takes too long to load up the queue.

This is why if you know the worker thread is going to take a while you monitor the thread(s) for completion individually like how I have it or in another thread like you suggest.

EDIT:

Also I found out that you can use the [paste] tags around your pastebin link to embed it.

vorticalbox - July 15, 2019, 8:04 am

(Jul 15, 2019 05:19 AM)Gazo Wrote: [\[CITE\]](#) Yeah, that's an appropriate use case for multiple threads.

The only weird thing I can see with your code is what if the worker thread is slower than the consumer thread?

Code while not q.empty() could evaluate to False causing the consumer thread to close if the worker thread takes too long to load up the queue.

This is why if you know the worker thread is going to take a while you monitor the thread(s) for completion individually like how I have it or in another thread like you suggest.

EDIT:

Also I found out that you can use the [paste] tags around your pastebin link to embed it. yeah, I just copied from the project I noted above and tweak quickly to make it run with a file, but yes you would only while the producer is alive && there are items in the queue.

I tried that and it just posted the link :P

nOrr - July 18, 2019, 9:38 am

Please note that the classic implementation of the python interpreter (the one from the website - aka "CPython") does NOT allow parallel execution of threads. Note the distinction between concurrency and parallelism. This is because python (the reference implementation, aka. CPython) uses a GIL - Global Interpreter Lock - meaning even though you run multiple threads they will never run truly at the same time. Each thread will have their few cycles to shine and then the execution is shifted to another thread.

More information: https://en.wikipedia.org/wiki/Global_interpreter_lock

mafiaboy - August 6, 2019, 12:51 pm

Replying so I can come here later.

Influxes - August 10, 2019, 11:17 am

I haven't used this way, but it does seem very helpful. The way I thread is using the threading library. That way, all I need to do it:

```
threading.Thread(myFunction).start()
```

Or if I want to pass arguments

```
threading.Thread(myFunction, args=(argument1, argument2,)).start()
```

I like how the way you do it, you can determine the workers, with the threading library I just use a for loop.