Handling and Sharing Data Between Threads

Learn how to share data between threads

by Aquiles Carattino 6. august 2019 threading threads data sharing

When working with threads in Python, you will find very useful to be able to share data between different tasks. One of the advantages of threads in Python is that they share the same memory space, and thus exchanging information is relatively easy. However, some structures can help you achieve more specific goals.

In the previous article, we have covered how to start and synchronize threads and now it is time to expand the toolbox to handle the exchange of information between them.

Shared Memory

The first and most naive approach is to use the same variables in different threads. We have already used this feature in the previous tutorial, but without discussing it explicitly. Let's see how we can use shared memory through a very simple example:

Latest Articles

Instructions to build the Python for the Lab DAQ
by Aquiles Carattino, 27. marec 2021

Using slots in Python: limit dynamic attribute creation and improve speed
by Aquiles Carattino, 21. marec 2021

Getting started with Basler cameras
by Aquiles Carattino, 27. február 2021

Singletons: Instantiate objects only once
by Aquiles Carattino, 16. január 2021

How Python for the Lab helped the developer of Twingo
by Michal Jablonski, 19. september 2020

```python
from threading import Thread, Event
from time import sleep

event = Event()

def modify_variable(var):
    while True:
        for i in range(len(var)):
            var[i] += 1
        if event.is_set():
            break
        sleep(.5)
    print('Stop printing')


my_var = [1, 2, 3]
t = Thread(target=modify_variable, args=(my_var, ))
t.start()
while True:
    try:
        print(my_var)
        sleep(1)
    except KeyboardInterrupt:
        event.set()
        break
t.join()
print(my_var)
```

The example above is almost trivial, but it has a very important feature. We start a new thread by passing an argument, `my_var`, which is a list of numbers. The thread will increase the values of the numbers by one, with a certain delay. In this example we use events to graciously finish the thread, if you are not familiar with them, check the previous tutorial.

The important piece of code in this example is the `print(my_var)` line. That print statement lives in the main thread, however, it has access to the information being generated within a child thread. This behavior is possible thanks to memory sharing between different threads. Being able to access the same memory space is useful, but it can also pose some risks. In the example above, we have started only one thread, but we are not limited to that. We could, for example, start several threads:

```python
t = Thread(target=modify_variable, args=(my_var, ))
t2 = Thread(target=modify_variable, args=(my_var, ))
t.start()
t2.start()
```

And you would see that `my_var` and its information is shared across all threads. This is good for applications like the one above, in which it doesn't matter which thread adds one to the variable. Or does it? Let's slightly modify the code that runs in the thread. Let's remove the `sleep`:

```
def modify_variable(var):
    while True:
        for i in range(len(var)):
            var[i] += 1
        if event.is_set():
            break
        # sleep(.5)
    print('Stop printing')
```

Now, when we run the code, there will be no sleep in between one iteration and the next. Let's run it for a short time, let's say 5 seconds, we can do the following:

```
from time import time
[...]

my_var = [1, 2, 3]
t = Thread(target=modify_variable, args=(my_var, ))
t.start()
t0 = time()
while time()-t0 < 5:
    print(my_var)
    sleep(1)
event.set()
t.join()
print(my_var)
```

I've suppressed the parts of the code which repeat. If you run this code, you will get as outputs very large numbers. In my case, I got:

```
[6563461, 6563462, 6563463]
```

There is, however, a very important feature to notice. The three numbers are consecutive. This is expected because the starting variable was `[1, 2, 3]` and we are adding one to each variable. Let's start a second thread this time and see what the output is:

```
my_var = [1, 2, 3]
t = Thread(target=modify_variable, args=(my_var, ))
t2 = Thread(target=modify_variable, args=(my_var, ))
t.start()
t2.start()
t0 = time()
while time()-t0 < 5:
    try:
        print(my_var)
        sleep(1)
    except KeyboardInterrupt:
        event.set()
        break
event.set()
t.join()
t2.join()
print(my_var)
```

I've got as an output the following values:

```
[5738447, 5686971, 5684220]
```

You can first note that they are not larger than before, meaning that running two threads instead of one could actually be slower for this operation. The other thing to note is that the values are no consecutive to each other! And this is a very important behavior that can appear when working with multiple threads in Python. If you think really hard, can you explain where this issue is coming from?

In the previous tutorial, we discussed that threads are handled by the operating system, which decides when to spin one on or off. We have no control over what the operating system decides to do. In the example above, since there is no `sleep` in the loop, the operating system will have to decide when to stop one and start another thread. However, that does not explain completely the output we are getting. It doesn't matter if one thread runs first and stops, etc. we are always adding `+1` to each element.

The problem with the code above is in the line `var[i] += 1`, which is actually two operations. First, it copies the value from `var[i]` and ads `1`. Then it stores the value back to `var[i]`. In between these two operations, the operating system may decide to switch from one task to another. In such case, the value both tasks see in the list is the same, and therefore instead of adding `+1` twice, we do it only once. If you want to do it even more noticeable, you can start two threads, one that adds and one that subtracts from a list, and that would give you a quick hint of which thread runs faster. In my case, I got the following output:

```
[-8832, -168606, 2567]
```

But if I run it another time, I get:

```
[97998, 133432, 186591]
```

Note

You may notice that there is a delay between the `start` of both threads, which may give a certain advantage to the first thread started. However, that alone cannot explain the output generated.

How to synchronize data access

To solve the problem we found in the previous examples, we have to be sure that no two threads try to write at the same time to the same variable. For that, we can use a `Lock`:

```
from threading import Lock
[...]
data_lock = Lock()
def modify_variable(var):
    while True:
        for i in range(len(var)):
            with data_lock:
                var[i] += 1
        if event.is_set():
            break
        # sleep(.5)
    print('Stop printing')
```

Note that we added a line `with data_lock:` to the function. If you run the code again, you will see that the values we get are always consecutive. The lock guarantees that only one thread will access the variable at a time.

The examples of increasing or decreasing values from a list are almost trivial, but they point in the direction of understanding the complications of memory management when dealing with concurrent programming. Memory sharing is a nice feature, but it comes with risks also.

Queues

One of the common situations in which threads are used is when you have some slow tasks that you can't optimize. For example, imagine you are downloading data from a website using. Most of the time the processor would be idle. This means you could use that time for something else. If you want to download an entire website (also called scraping), it would be a good solution to download several pages at the same time. Imagine you have a list of pages you want to download, and you start several threads, each one to download one page. If you are not careful on how to implement this, you may end up downloading twice the same, as we saw in the previous section.

Here is where another object can be very useful when working with threads: **Queues**. A queue is an object which accepts data in order, i.e. you put data to it one element at a time. Then, the data can be consumed in the same order, called First-in-first-out (FIFO). A very simple example would be:

```
from queue import Queue

queue = Queue()
for i in range(20):
    queue.put(i)

while not queue.empty():
    data = queue.get()
    print(data)
```

In this example you see that we create a `Queue`, then we put into the queue the numbers from 0 to 19. Later, we create a `while` loop that gets data out of the queue and prints it. This is the basic behavior of queues in Python. You should pay attention to the fact that numbers are printed in the same order in which they were added to the queue.

Coming back to the examples from the beginning of the article, we can use queues to share information between threads. We can modify the function such that instead of a list as an argument, it accepts a queue from which it will read elements. Then, it will output the results to an output queue:

```
from threading import Thread, Event
from queue import Queue
from time import sleep, time

event = Event()

def modify_variable(queue_in, queue_out):
    while True:
        if not queue_in.empty():
            var = queue_in.get()
            for i in range(len(var)):
                var[i] += 1
            queue_out.put(var)
        if event.is_set():
            break
    print('Stop printing')
```

To use the code above, we will need to create two queues. The idea is that we can also create two threads, in which the input and output queue are reversed. In that case, on thread puts its output on the queue of the second thread and the other way around. This would look like the following:

Python Copy

```
my_var = [1, 2, 3]
queue1 = Queue()
queue2 = Queue()
queue1.put(my_var)
t = Thread(target=modify_variable, args=(queue1, queue2))
t2 = Thread(target=modify_variable, args=(queue2, queue1))
t.start()
t2.start()
t0 = time()
while time()-t0 < 5:
    try:
        sleep(1)
    except KeyboardInterrupt:
        event.set()
        break
event.set()
t.join()
t2.join()
if not queue1.empty():
    print(queue1.get())
if not queue2.empty():
    print(queue2.get())
```

In my case, the output I get is:

```
[871, 872, 873]
```

Much smaller than everything else we have seen so far, but at least we managed to shared data between two different threads, without any conflicts. Where does this slow speed come from? Let's try with the scientific approach which is to split the problem and look at each part. One of the most interesting things is that we are checking whether the queue is empty before trying to run the rest of the code. We can monitor how much time it is actually spent running the important part of our program:

```
def modify_variable(queue_in: Queue, queue_out: Queue):
    internal_t = 0
    while True:
        if not queue_in.empty():
            t0 = time()
            var = queue_in.get()
            for i in range(len(var)):
                var[i] += 1
            queue_out.put(var)
            internal_t += time()-t0
        if event.is_set():
            break
    sleep(0.1)
    print(f'Running time: {internal_t} seconds\n')
```

The only changes are the addition of a new variable in the function, called `internal_t`. Then, we monitor the time spent calculating and putting to the new thread. If we run the code again, the output you should get is something like:

```
Running time: 0.0006377696990966797 seconds
Running time: 0.00035738945007324222 seconds
```

This means that out of the 5 seconds in which our program runs, only during about .9 milliseconds we are actually doing something. This is .01% of the time! Let's quickly see what happens if we change the code for using only one queue instead of two, i.e. the input and output queue would be the same:

```
t = Thread(target=modify_variable, args=(queue1, queue1))
t2 = Thread(target=modify_variable, args=(queue1, queue1))
```

With just that change, I've got the following output:

```
Running time: 4.290639877319336 seconds
Running time: 4.355865955352783 seconds
```

That is much better! For the about of 5 seconds in which the program runs, the threads run for a total of 8 seconds. Which is what one would expect of parallelizing. Also, the output of the loops is much larger:

```
[710779, 710780, 710781]
```

Can you try to guess what made our program so slow if we use two queues but reasonably fast if we use the same queue for output and input? You have to remember that when you use threads *blindly* as we have done in the previous example, we leave everything in the hands of the operating system.

We have no control of whether the OS decides to switch from a task to another. In the code above, we check whether the queue is empty. It may very well be that the operating system decides to give priority to a task which is basically not doing anything, but waiting until there is an element in the queue. If this happens out of synchronization, most of the time the program will be just waiting to have an element in the queue (it is always prioritizing the wrong task). While when we use the same task for input and output, it doesn't matter which task it runs, there will always be something to proceed.

If you want to see whether the previous speculation is true or not, we can measure it. We have only one `if` statement to check `queue.empty()`, we can add an `else` to accumulate the time the program is actually not doing anything:

```
def modify_variable(queue_in: Queue, queue_out: Queue):
    internal_t = 0
    sleeping_t = 0
    while True:
        if not queue_in.empty():
            t0 = time()
            var = queue_in.get()
            for i in range(len(var)):
                var[i] += 1
            queue_out.put(var)
            internal_t += time()-t0
        else:
            t0 = time()
            sleep(0.001)
            sleeping_t += time()-t0
        if event.is_set():
            break
    sleep(0.1)
    print(f'Running time: {internal_t} seconds')
    print(f'Sleeping time: {sleeping_t} seconds')
```

In the code above, if the queue is empty, the program will sleep for 1 millisecond. Of course, this is not the best, but we can assume that 1 millisecond will have no real impact on the overall performance of the program. When I run the program above, using two different queues I get the following output:

```
Running time: 0.0 seconds
Sleeping time: 5.001126289367676 seconds
Running time: 0.00018215179443359375 seconds
Sleeping time: 5.001835107803345 seconds
[4126, 4127, 4128]
```

Where it is clear that most of the time the program is just waiting until more data is available on the queue. Since we are sleeping for 1 ms every time there is no data available, we are actually making the program much slower. But I think it is a good example. We can compare it with using the same queue for input and output:

```
Running time: 3.1206254959106445 seconds
Sleeping time: 1.3756272792816162 seconds
Running time: 3.253162145614624 seconds
Sleeping time: 1.136244535446167 seconds
```

Now you see that even if we are wasting some time because of the sleep, most of the time our routine is actually performing a calculation.

The only thing you have to be careful when using the same queue for input and output is that between checking whether the queue is empty and actually reading from it, it could happen that the other thread grabbed the result. This is described in the Queue documentation. Unless we include a `Lock` ourselves, the Queue can be read and written by any threads. The Lock only comes into effect for the `get` or `put` commands.

Extra Options of Queues

Queues have some extra options, such as the maximum number of elements they can hold. You can also define **LIFO** (last-in, first-out) types of queues, which you can read about in the documentation. What I find more useful about `Queues` is that they are written in pure Python. If you visit their source code, you can learn a lot about synchronization in threads, custom exceptions, and documenting.

What is important to note, is that when you work with multiple Threads, sometimes you want to wait (i.e. block the execution), sometimes you don't. In the examples above, we have always been checking whether the Queue was empty before reading from it. But what happens if we don't check it? The method `get` has two options: `block` and `timeout`. The first is used to determine whether we want the program to wait until an element is available. The second is to specify the number of seconds we want it to wait. After that amount of time, an exception is raised. If we set `block` to false, and the queue is empty, the exception is raised immediately.

We can change the function `modify_variable` to take advantage of this:

```
def modify_variable(queue_in: Queue, queue_out: Queue):
    internal_t = 0
    while True:
        t0 = time()
        var = queue_in.get()
        for i in range(len(var)):
            var[i] += 1
        queue_out.put(var)
        internal_t += time()-t0
        if event.is_set():
            break
    sleep(0.1)
    print(f'Running time: {internal_t} seconds\n')
```

With this code, using different queues for input and output, I get the following:

```
Running time: 4.914130210876465 seconds
Running time: 4.937211513519287 seconds

[179992, 179993, 179994]
```

Which is much better than what we were getting before. But, this is not really fair. A lot of time is spent just waiting in the `get` function, but we are still counting that time. If we move the line of `t0 = time()` right below the `get`, the times the code is actually running are very different:

```
Running time: 0.7706246376037598 seconds
Running time: 0.763786792755127 seconds

[177807, 177808, 177809]
```

So now you see, perhaps we should have calculated the time differently also in the previous examples, especially when we were using the same queue for input and output.

If we don't want to program to block while waiting for a get, we can do the following:

```
from queue import Empty
[...]

    try:
        var = queue_in.get(block=False)
    except Empty:
        continue
```

Or, we could specify a timeout, like this:

```
try:
    var = queue_in.get(block=True, timeout=0.001)
except Empty:
    continue
```

In that case, we either don't wait (`block==False`) and we catch the exception, or we wait for up to 1 millisecond (`timeout=0.001`) and we catch the exception. You can play around with these options to see if the performance of your code changes in any way.

Queues to Stop Threads

Up to now, we have always used locks to stop threads, which is, I believe, a very elegant way of doing it. However, there is another possibility, which is to control the flow of threads by appending special information to queues. A very simple example would be to add an element `None` to a queue, and when the function gets it, it stops the execution. The code would look like this:

```
[...]

var = queue_in.get()
if var is None:
    break
```

And then, in the main part of the script, when we want to stop the threads, we do the following:

```
queue1.put(None)
queue2.put(None)
```

If you are wondering why you would choose one or the other option, the answer is actually quite straightforward. The examples we are working with, always have queues with 1 element at most. When we stop the program, we know everything in the queue has been processed. Imagine, however, that the program is processing a collection of elements, with no relation between each other. This would be the case if you would be downloading data from a website, for example, or processing images, etc. You want to be sure you finish processing everything before stopping the thread. In such a case, adding a special value to the queue guarantees that all elements will be processed.

Warning

it is a very wise idea to be sure a queue is empty after you stop using it. If, as before, you interrupt the thread by looking at the status of a lock, the queue may be left with a lot of data in it, and thus the memory will not be freed. A simple while-loop that gets all the elements of a queue solves it.

IO Bound threads

The examples in this article are computationally intensive, and thus they are right on the edge where using multi-threading is not applicable and where all the problems arise (such as concurrency, etc.) We have focused on the limits of multi-threading because if you understand them, you will program with much more confidence. You won't be on your toes hoping for a problem not to arise.

An area where multi-threading excels is in IO (input-output) tasks. For example, if you have a program which writes to the hard drive while it is doing something else, the writing to the hard drive can be safely offloaded to a separate thread, while the rest of the program keeps running. This is also valid if the program waits for user input or network resources to become available, downloads data from the internet, etc.

Example downloading websites

To close this article, let's see an example of downloading websites using threadings, queues, and locks. Even if some performance improvements are possible, the example will show the basic building blocks of almost any threading application of interest.

First, let's discuss what we want to achieve. To keep the example simple, we will download all the websites on a list, and we want to save the downloaded information to the hard drive. The first approach would be to create a for-loop that goes through the list. This code can be found on the Github repository. However, we would like to work with multiple threads.

The architecture we propose therefore is: One Queue that hosts the websites we want to download, one queue that hosts the data to be saved. Some threads going to the websites to download, and each one outputs the data to the other queue. Some threads which read the latter queue and save the data to disk, taking care of not overwriting files. The modules we are going to use for this example are:

```python
import os
from queue import Queue
from threading import Lock, Thread
from urllib import request
```

Note that we are using urllib to downloading data. We then create the queues and the lock we are going to use:

```
website_queue = Queue()
data_queue = Queue()
file_lock = Lock()
```

Now we can proceed to define the functions which will run on separated threads. For downloading data:

```
def download_data():
    while True:
        var = website_queue.get()
        if var is None:
            break
        response = request.urlopen(var)
        data = response.read()
        data_queue.put(data)
```

Here you see that we used the strategy of checking whether the queue has a special element, to be sure that we processed all the websites on the queue before stopping the thread. We download the data from the website and we put it on another queue to be later processed.

The saving requires a bit more care because we have to be sure that no two threads try to write to the same file:

```
def save_data():
    while True:
        var = data_queue.get()
        if var is None:
            break
        with file_lock:
            i = 0
            while os.path.exists(f'website_data_{i}.dat'):
                i += 1
            open(f'website_data_{i}.dat', 'w').close()
        with open(f'website_data_{i}.dat', 'wb') as f:
            f.write(var)
```

The approach is similar to the downloading of data. We wait until a special element is present to stop the thread. Then we acquire a lock to be sure no other thread is looking at the available files to write to. The loop just checks which file number is available. We have to use a lock here because there is a change two threads run the same lines at the same time and find the available file to be the same.

When we write to the file, we don't care about the lock, because we know that only one thread will write to each file. That is why we create the file on one line, while the lock is acquired:

```
open(f'website_data_{i}.dat', 'w').close()
```

But we write the data on a separate line, without the lock:

```
with open(f'website_data_{i}.dat', 'wb') as f:
    f.write(var)
```

This may seem too convoluted for our purposes, and it is true. However, it shows one possible approach in which several threads could be writing to the hard drive at the same time because they are writing to different files. Note that we have used `wb` for the opening of the file. The `w` is because we want to write to the file (not append), and the `b` because the result of reading the `response` is binary and not a string. Then, we need to trigger the threads we want to download and save the data. First, we create a list of websites we want to download. In this case, Wikipedia homepages in different languages:

```
website_list = [
    'https://www.wikipedia.org/',
    'https://nl.wikipedia.org/',
    'https://de.wikipedia.org/',
    'https://fr.wikipedia.org/',
    'https://pt.wikipedia.org/',
    'https://it.wikipedia.org',
    'https://ru.wikipedia.org',
    'https://es.wikipedia.org',
    'https://en.wikipedia.org',
    'https://ja.wikipedia.org',
    'https://zh.wikipedia.org',
]
```

And then we prepare the queues and trigger the threads:

```
for ws in website_list:
    website_queue.put(ws)

threads_download = []
threads_save = []
for i in range(3):
    t = Thread(target=download_data)
    t.start()
    threads_download.append(t)
    t2 = Thread(target=save_data)
    t2.start()
    threads_save.append(t2)
```

With this, we create lists with the threads running for saving and downloading. Of course, the numbers could have been different. Then, we need to be sure we stop the downloading threads:

```
for i in range(3):
    website_queue.put(None)
```

Since we run 3 threads for downloading data, we have to be sure we append 3 `None` to the Queue, or some thread won't stop. After we are sure the downloading finished, we can stop the saving:

```
for t in threads_download:
    t.join()

for i in range(3):
    data_queue.put(None)
```

And then we wait for the saving to finish:

```
for t in threads_save:
    t.join()
print(f'Finished downloading {len(website_list)} websites')
```

Now we know all the threads have finished and the queues are empty. If you run the program, you can see the list of 10 files created, with the HTML of 10 different Wikipedia homepages.

Conclusions

In the previous article, we have seen how you can use threading to run different functions at the same time, and some of the most useful tools you have available to control the flow of different threads. In this article we have discussed how you can share data between threads, exploiting both the fact of the shared memory between threads and by using queues.

Having access to shared memory makes programs very quick to develop, but they can give rise to problems when different threads are reading/writing to the same elements. This was discussed at the beginning of the article, in which we explored what happens when using a simple operator such as `=+` to increase the values of an array by 1. Then we explored how to use Queues to share data between threads, both between the main thread and child threads as between child threads.

To finish, we have shown a very simple example of how to use threads to download data from a website and save it to disk. The example is very basic, but we will expand it in the following article. Other IO (input-output) tasks that can be explored are acquisition from devices such as a camera, waiting for user input, reading from disk, etc.

Article written by Aquiles Carattino

Header Illustration by Tsvetelina Stoynova

Join over 1500 Python deve
and don't miss any upda

Your E-Mail

Subscribe

Or check out our Books
Privacy Policy

Never Stop Learning