Python Threading

Summary: in this tutorial, you'll learn how to use the Python threading module to develop multi-threaded applications.

Single-threaded applications

Let's start with a simple program:

```
from time import sleep, perf_counter

def task():
    print('Starting a task...')
    sleep(1)
    print('done')


start_time = perf_counter()

task()
task()
```

```
end_time = perf_counter()

print(f'It took {end_time- start_time: 0.2f} second(s) to complete.')
```

How it works.

First, import the `sleep()` and `perf_counter()` functions from the `time` module:

```
from time import sleep, perf_counter
```

Second, define a function (https://www.pythontutorial.net/python-basics/python-functions/) that takes one second to complete:

```
def task():
    print('Starting a task...')
    sleep(1)
    print('done')
```

Third, get the value of the performance counter by calling the `perf_counter()` function:

```
start_time = perf_counter()
```

Fourth, call the `task()` function twice:

```
task()
task()
```

Fifth, get the value of the performance counter by calling the `perf_counter()` function:

```
end_time = perf_counter()
```

Finally, output the time that takes to complete running the `task()` function twice:

```
print(f'It took {end_time- start_time: 0.2f} second(s) to complete.')
```

Here is the output:

```
Starting a task...
done
Starting a task...
done
It took  2.00 second(s) to complete.
```

As you may expect, the program takes about two seconds to complete. If you call the `task()` function 10 times, it would take about 10 seconds to complete.

The following diagram illustrates how the program works:

First, the `task()` function executes and sleeps for one second. Then it executes the second time and also sleeps for another second. Finally, the program completes.

When the `task()` function called the `sleep()` function, the CPU is idle. In other words, the CPU doesn't do anything, which is not efficient in terms of resource utilization.

This program has one process with a single thread, which is called the main thread. Because the program has only one thread, it's called the single-threaded program.

Using Python threading to develop a multi-threaded program example

To create a multi-threaded program, you need to use the Python `threading` module.

First, import the `Thread` class from the `threading` module:

```
from threading import Thread
```

Second, create a new thread by instantiating an instance of the `Thread` class:

```
new_thread = Thread(target=fn,args=args_tuple)
```

The `Thread()` accepts many parameters. The main ones are:

- `target` : specifies a function (`fn`) to run in the new thread.
- `args` : specifies the arguments of the function (`fn`). The `args` argument is a tuple.

Third, start the thread by calling the `start()` method of the `Thread` instance:

```
new_thread.start()
```

If you want to wait for the thread to complete in the main thread, you can call the `join()` method:

```
new_thread.join()
```

By calling the `join()` method, the main thread will wait for the second thread to complete before it is terminated.

The following program illustrates how to use the `threading` module:

```
from time import sleep, perf_counter
from threading import Thread


def task():
    print('Starting a task...')
    sleep(1)
    print('done')
```

```
start_time = perf_counter()

# create two new threads
t1 = Thread(target=task)
t2 = Thread(target=task)

# start the threads
t1.start()
t2.start()

# wait for the threads to complete
t1.join()
t2.join()

end_time = perf_counter()

print(f'It took {end_time- start_time: 0.2f} second(s) to complete.')
```

How it works. (and we'll focus on the threading part only)

First, create two new threads:

```
t1 = Thread(target=task)
t2 = Thread(target=task)
```

Second, start both threads by calling the `start()` method:

```
t1.start()
t2.start()
```

Third, wait for both threads to complete:

```
t1.join()
t2.join()
```

Finally, show the executing time:

```
print(f'It took {end_time- start_time: 0.2f} second(s) to complete.')
```

Output:

```
Starting a task...
Starting a task...
done
done
It took  1.00 second(s) to complete.
```

When the program executes, it'll have three threads: the main thread is created by the Python interpreter, and two threads are created by the program.

As shown clearly from the output, the program took one second instead of two to complete.

The following diagram shows how threads execute:

Passing arguments to threads

The following program shows how to pass arguments to the function assigned to a thread:

```
from time import sleep, perf_counter
from threading import Thread


def task(id):
    print(f'Starting the task {id}...')
    sleep(1)
    print(f'The task {id} completed')


start_time = perf_counter()

# create and start 10 threads
threads = []
for n in range(1, 11):
    t = Thread(target=task, args=(n,))
    threads.append(t)
    t.start()

# wait for the threads to complete
for t in threads:
    t.join()

end_time = perf_counter()

print(f'It took {end_time- start_time: 0.2f} second(s) to complete.')
```

How it works.

First, define a `task()` function that accepts an argument:

```
def task(id):
    print(f'Starting the task {id}...')
```

```
    sleep(1)
    print(f'The task {id} completed')
```

Second, create 10 new threads and pass an id to each. The `threads` list is used to keep track of all newly created threads:

```
threads = []
for n in range(1, 11):
    t = Thread(target=task, args=(n,))
    threads.append(t)
    t.start()
```

Notice that if you call the `join()` method inside the loop, the program will wait for the first thread to complete before starting the next one.

Third, wait for all threads to complete by calling the `join()` method:

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

The following shows the output of the program:

```
Starting the task 1...
Starting the task 2...
Starting the task 3...
Starting the task 4...
Starting the task 5...
Starting the task 6...
Starting the task 7...
Starting the task 8...
Starting the task 9...
Starting the task 10...
The task 10 completed
```

```
The task 8 completed
The task 1 completed
The task 6 completed
The task 7 completed
The task 9 completed
The task 3 completed
The task 4 completed
The task 2 completed
The task 5 completed
It took  1.02 second(s) to complete.
```

It just took 1.05 seconds to complete.

Notice that the program doesn't execute the thread in the order from 1 to 10.

When to use Python threading

As introduced in the process and thread tutorial (https://www.pythontutorial.net/advanced-python/differences-between-processes-and-threads/) , there're two main tasks:

- I/O-bound tasks – the time spent on I/O is significantly more than the time spent on computation
- CPU-bound tasks – the time spent on computation is significantly higher than the time waiting for I/O.

Python threading is optimized for I/O bound tasks. For example, requesting remote resources, connecting a database server, or reading and writing files.

A Practical Python threading example

Suppose that you have a list of text files in a folder e.g., `C:/temp/` . And you want to replace a text with a new one in all the files.

The following single-threaded program shows how to replace a substring with the new one in the text files:

```python
from time import perf_counter


def replace(filename, substr, new_substr):
    print(f'Processing the file {filename}')
    # get the contents of the file
    with open(filename, 'r') as f:
        content = f.read()

    # replace the substr by new_substr
    content = content.replace(substr, new_substr)

    # write data into the file
    with open(filename, 'w') as f:
        f.write(content)


def main():
    filenames = [
        'c:/temp/test1.txt',
        'c:/temp/test2.txt',
        'c:/temp/test3.txt',
        'c:/temp/test4.txt',
        'c:/temp/test5.txt',
        'c:/temp/test6.txt',
        'c:/temp/test7.txt',
        'c:/temp/test8.txt',
        'c:/temp/test9.txt',
        'c:/temp/test10.txt',
    ]

    for filename in filenames:
        replace(filename, 'ids', 'id')
```

```
if __name__ == "__main__":
    start_time = perf_counter()

    main()

    end_time = perf_counter()
    print(f'It took {end_time- start_time :0.2f} second(s) to complete.')
```

Output:

```
It took 0.16 second(s) to complete.
```

The following program has the same functionality. However, it uses multiple threads instead:

```
from threading import Thread
from time import perf_counter


def replace(filename, substr, new_substr):
    print(f'Processing the file {filename}')
    # get the contents of the file
    with open(filename, 'r') as f:
        content = f.read()

    # replace the substr by new_substr
    content = content.replace(substr, new_substr)

    # write data into the file
    with open(filename, 'w') as f:
        f.write(content)
```

```
def main():
    filenames = [
        'c:/temp/test1.txt',
        'c:/temp/test2.txt',
        'c:/temp/test3.txt',
        'c:/temp/test4.txt',
        'c:/temp/test5.txt',
        'c:/temp/test6.txt',
        'c:/temp/test7.txt',
        'c:/temp/test8.txt',
        'c:/temp/test9.txt',
        'c:/temp/test10.txt',
    ]

    # create threads
    threads = [Thread(target=replace, args=(filename, 'id', 'ids'))
            for filename in filenames]

    # start the threads
    for thread in threads:
        thread.start()

    # wait for the threads to complete
    for thread in threads:
        thread.join()


if __name__ == "__main__":
    start_time = perf_counter()

    main()
```

```
    end_time = perf_counter()
    print(f'It took {end_time- start_time :0.2f} second(s) to complete.')
```

Output:

```
Processing the file c:/temp/test1.txt
Processing the file c:/temp/test2.txt
Processing the file c:/temp/test3.txt
Processing the file c:/temp/test4.txt
Processing the file c:/temp/test5.txt
Processing the file c:/temp/test6.txt
Processing the file c:/temp/test7.txt
Processing the file c:/temp/test8.txt
Processing the file c:/temp/test9.txt
Processing the file c:/temp/test10.txt
It took 0.02 second(s) to complete.
```

As you can see clearly from the output, the multi-thread program runs so much faster.

Summary

- Use the Python `threading` module to create a multi-threaded application.
- Use the `Thread(function, args)` to create a new thread.
- Call the `start()` method of the `Thread` class to start the thread.
- Call the `join()` method of the `Thread` class to wait for the thread to complete in the main thread.
- Only use threading for I/O bound processing applications.