Multiprocessing and multithreading in Python

In programming a *process* is a program and its needed resources that has been loaded to the computer's memory and being executed by the central processing unit in conjunction with the arithmetic unit, performing calculations and/or storing data in memory or on the computer's hard drive. The process has its own memory space allocated.

A *thread* is a unit of execution within a particular process and a process can have multiple threads running, using a part of the process' memory and sharing it with other threads. Multithreading aims to increase the utilisation of one of the CPU cores where several threads are here spawned by a single process and each is assigned a particular task.

Multithreading is therefore not very useful when doing massive calculations, however, it is very useful when performing I/O-bound tasks. These are tasks that read and write to the hard drive or passing data in a network such as downloading or uploading files. In these cases, scripts are usually just sitting idle, waiting for the next operation. So instead of halting your script while waiting for a download, for example, the script can keep running doing other tasks until the download has completed. For CPU-bound tasks that relies on heavy calculations, *multiprocessing* is a lot more efficient. Here, tasks are divided on separate cores of the computer's CPU.

Multiprocessing

We are going to implement a simple method that mimics a computationally costly function. We are also going to run this several times and measure the time it takes to complete this function. To start, import these methods/modules into a fresh Python script `example.py`,

```
1 from time import perf_counter as pc
2 from time import sleep as pause
```

Our method will use the `time` module's `sleep()` method to resemble a particular process that takes some time to complete.

```
1 def runner():
2         print("Performing a costly function")
3         pause(1)
4         print("Function complete")
```

We run the method, taking a time mark at the start and end of the process of running the method. We then print how long it took using an f-string, which is a simpler version of `"".format()` where we can insert the variables directly into the string.

```
if __name__ == "__main__":
        start = pc()
        runner()
        end = pc()
        print(f"Process took {round(end-start, 2)} seconds")
```

```
$ python3 example.py
Performing a costly function
Function complete
Process took 1.0 seconds
```

We now run the method ten times to see if it takes roughly ten seconds to run.

```
if __name__ == "__main__":
        start = pc()
        for _ in range(10):
                runner()
        end = pc()
        print(f"Process took {round(end-start, 2)} seconds")
```

```
$ python3 example.py
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Process took 10.02 seconds
```

As the method was called and executed ten times sequentially, it took roughly 10.02 seconds to finish this script, which seems about right.

However, by using Python's `multiprocessing` module, we can execute the method several times in parallel. Today's computers normally consists of processors with multiple cores that can be used simultaneously. This module will help us send tasks to separate cores in the CPU to run them at the same time. Import the module at the top of your script.

```
import multiprocessing as mp
```

To create a process, we instantiate a process object with the method we want to run as the `target` argument (there is also an `args` argument to pass variables in a list to the target method). We can now create multiple process objects by writing the following code.

```
p1 = mp.Process(target=runner)
p2 = mp.Process(target=runner)
```

To initiate the processes, we need to call the `start` method associated with the process object.

```
p1.start()
p2.start()
```

Now let's measure how long the script will run for:

```
if __name__ == "__main__":
        start = pc()
        p1 = mp.Process(target=runner)
        p2 = mp.Process(target=runner)
        p1.start()
        p2.start()
        end = pc()
        print(f"Process took {round(end-start, 2)} seconds")
```

```
$ python3 example.py
Process took 0.02 seconds
Performing a costly function
Performing a costly function
Function complete
Function complete
```

What we can see now is that the process seemed to have taken 0.02 seconds. This was obviously not true as we can see that some print statements were presented after the the last print method in our script. So what happened here? Well, the processes were started simultaneously and the script continued running although these processes were not finished yet. So here we have an issue that we are not sure how long the script actually took to process as the methods are no longer running in succession, but are free to execute individually.

We can handle this by using the `join` method on the process object in order to rejoin these processes once they are all completed.

```
p1.join()
p2.join()
```

Now let's measure how long the script actually takes to run:

```
if __name__ == "__main__":
        start = pc()
        p1 = mp.Process(target=runner)
        p2 = mp.Process(target=runner)
        p1.start()
        p2.start()
        p1.join()
        p2.join()
        end = pc()
        print(f"Process took {round(end-start, 2)} seconds")
```

```
$ python3 example.py
Performing a costly function
Performing a costly function
Function complete
Function complete
Process took 1.1 seconds
```

So here we can see that even though the method ran twice, each time pausing for one second, the script only took 1.1 second to complete, meaning that we cut the processing time in half.

If we want to run a large number of processes, we can initialize them and run them in a loop. However, if we want to rejoin them, this has to be done outside of this loop. Then we need to store our processes in a list so that we can iterate over this list to join them when they are finished. If we join them directly in the loop, they will be running sequentially again.

Here is generic code to run 10 parallel instances of `some_method` with argument `some_var`.

```
processes = []
for _ in range(10):
        p = mp.Process(target=some_method, args=[some_var])
        processes.append(p)
for p in processes:
        p.start()
for p in processes:
        p.join()
```

Futures

Now, if we also want to get some results back from our methods, we have to use the `Manager` class and make use of some worker method that stores the values for us. This can be a bit tedious to handle, so we are now actually going to look at another Python module that can run multiprocesses and handle joins and return variables for us, thus being somewhat more manageable. So the above module was just a way of showing you how these processes work before we make use of the more friendly `concurrent.futures` module.

Remove the `multiprocessing` import and import `concurrent.futures` instead.

```
import concurrent.futures as future
```

Futures has a pool executor that is best run in a context manager (Python `with` statement). We then call the `submit` method, passing in the method we want to run along with any arguments that should be passed to our defined method. To access our method's return arguments, we call the `result` method of the stored process object. This module helps us joining the processes together and will automatically wait on processes to finish before exiting the `with` block.

```
with future.ProcessPoolExecutor() as ex:
        p1 = ex.submit(some_method, some_arg, some_other_arg, ...) # Starts first
        ↪  process
        p2 = ex.submit(some_method, some_arg, some_other_arg, ...) # Starts second
        ↪  process

        r1 = p1.result() # Program waits until p1 is complete before assigning r2
        r2 = p2.result()

print("all done") # Will be printed once all processes are completed
```

Let's start by updating our `runner` method to take some argument and return a string.

```
def runner(n):
        print(f"Performing costly function {n}")
        pause(n)
        return f"Function {n} has completed"
```

Instead of running processes in a loop, we can make use of the executor's `map` method. Python actually has a built-in `map` method which takes a method and a list of variables that is being iterated over and executed once for each item in the list. Using the built-in method looks like this:

```
>>> def sq(x):
        return x**2

>>> for x in map(sq, [2,3,4,5]):
        print(x)
4
9
16
25
```

So, let's define a list of integers that we pass as arguments to our `runner` method. We then create a `results` variable to store our returned arguments that will be returned by the executor's `map` method, and we pass in our method along with the list of integers. Finally, we iterate over the returned `map` object and print the returned results.

```python
if __name__ == "__main__":
        start = pc()

        with future.ProcessPoolExecutor() as ex:
                p = [5, 4, 3, 2, 1]
                results = ex.map(runner, p)

                for r in results:
                        print(r)

        end = pc()
        print(f"Process took {round(end-start, 2)} seconds")
```

```
$ python3 example.py
Performing costly function 5
Performing costly function 4
Performing costly function 3
Performing costly function 2
Performing costly function 1
Function 3 has completed
Function 2 has completed
Function 1 has completed
Function 5 has completed
Function 4 has completed
Process took 5.21 seconds
```

As you can see, the methods were actually executed in a different order than they were called. This is because there is no way of knowing how busy the specific core is that the executor sent the task to. So the methods will be started in different orders each time you run this script. However, the executor handles the return values and make sure that they are returned in order as they have been executed. The `futures` module also has a method called `as_completed` which will return the results in the order they are finished (cannot be used with `map` but with the submit method above using a loop).

Multithreading

Using the `concurrent.futures` module, running multiple threads is as easy as changing the `ProcessPoolExecutor` class to a `ThreadPoolExecutor` class. Otherwise, the procedure is exactly the same.

```python
if __name__ == "__main__":
        start = pc()

        with future.ThreadPoolExecutor() as ex:
                p = [5, 4, 3, 2, 1]
                results = ex.map(runner, p)

                for r in results:
                        print(r)

        end = pc()
        print(f"Process took {round(end-start, 2)} seconds")
```

```
$ python3 example.py
Performing costly function 5
Performing costly function 4
Performing costly function 3
Performing costly function 2
Performing costly function 1
Function 5 has completed
Function 4 has completed
Function 3 has completed
Function 2 has completed
Function 1 has completed
Process took 5.01 seconds
```

As you can see, the output here is somewhat different. The methods are actually executed in the order they are being called. This is because we are no longer using multiple cores to process the task. This code is of course only for illustrative purposes, it is not at all computationally costly or I/O-bound, so the threading procedure works just as well in this case and the process once again finished in around five seconds.

Links for those of you that want to learn more, and see other examples:

https://www.machinelearningplus.com/python/parallel-processing-python/

https://www.geeksforgeeks.org/parallel-processing-in-python/