How to Use the Multiprocessing Package in Python

Understand multiprocessing in no more than 6 minutes

Multiprocessing is quintessential when a long-running process has to be speeded up or multiple processes have to execute parallely. Executing a process on a single core confines its capability, which could otherwise spread its tentacles across multiple cores. If the time-consuming task has the scope to run in parallel and the underlying system has multiple processors/cores, Python provides an easy-to-use interface to embed multiprocessing.

This article will differentiate Multiprocessing from Threading, guide you through the two techniques used to implement Multiprocessing — Process and Pool, and explore processes' interaction and shared memory concepts.

Multiprocessing vs. Multithreading

Multiprocessing leverages the entirety of CPU cores (multiple processes), whereas Multithreading maps multiple threads to every process. In multiprocessing, each process is associated with its own memory, which doesn't lead to data corruption or deadlocks. Threads utilize shared memory, henceforth enforcing the thread locking mechanism. For CPU-related jobs, multiprocessing is preferable, whereas, for I/O-related jobs (IO-bound vs. CPU-bound tasks), multithreading performs better.

> In Python, the Global Interpreter Lock (GIL) is a lock that allows only a single thread to control the Python interpreter.
>
> In the case of multithreading, which is primarily used for IO-bound jobs, GIL doesn't have much impact as the lock is shared between threads while they are waiting for I/O.
>
> Multiprocessing, on the other hand, allocates a Python Interpreter and GIL to every process.

In the next section, let's look at one of the significant concepts of the multiprocessing package — Process class.

Using Process

The Process class in multiprocessing allocates all the tasks in the memory in one go. Every task created using the Process Class has to have a separate memory allocated.

Imagine a scenario wherein ten parallel processes are to be created where every process has to be a separate system process.

Here's an example (order of the output is non-deterministic):

https://replit.com/@allasamhita/Process-Class?lite=true#main.py

The Process class initiated a process for numbers ranging from 0 to 10. target specifies the function to be called, and args determines the argument(s) to be passed. start method commences the process. All the processes have been looped over to wait until every process execution is complete, which is detected using the join method. join() helps in making sure that the rest of the program runs only after the multiprocessing is complete.

***sleep()** method helps in understanding how concurrent the processes are!*

In the next section, let's look at various processes communication techniques.

How to Implement Pipe

If two processes need to communicate, **Pipe**'s the best choice. A pipe can have two end-points where each has send() and recv() methods. Data in a pipe could get corrupted if two processes (threads) read from or write to the same end-point simultaneously.

https://replit.com/@allasamhita/Pipe-Communication?lite=true#main.py

cube_sender and cube_receiver are two processes that communicate with each other using a pipe.

- The cube of the number 19 is sent from one end of the pipe to the other (x_conn to y_conn). x_conn is the input to the process p1. When send() is called on x_conn, the output is sent to y_conn.
- y_conn is the input to the process p2, which receives the output and prints the resultant cube.

How to Implement Queue

To store the output of multiple processes in a shared communication channel, a queue can be used. For instance, assume that the task is to find the cubes of the first ten natural numbers followed by adding 1 to each number.

Define two functions sum() and cube(). Then define a Queue (q) and call cube() function followed by the add() function.

https://replit.com/@allasamhita/Queue-Communication?lite=true#main.py

The code explains the communication of the object, in our case, q, amongst the two processes. The method empty() is to ascertain if the queue is empty, and get() returns the value stored in the queue.

The order of the result is non-deterministic.

Il precedente materiale è parte del più ampio articolo che potete trovare al link

https://towardsdatascience.com/how-to-use-the-multiprocessing-package-in-python3-a1c808415ec2