**Multi-Processing In Python**

[[Isha Choudhary](https://medium.com/@shahooda637?source=post_page-----32d4b1c97354--------------------------------)](https://medium.com/@shahooda637?source=post_page-----32d4b1c97354--------------------------------)

[Isha Choudhary](https://medium.com/@shahooda637?source=post_page-----32d4b1c97354--------------------------------)

·

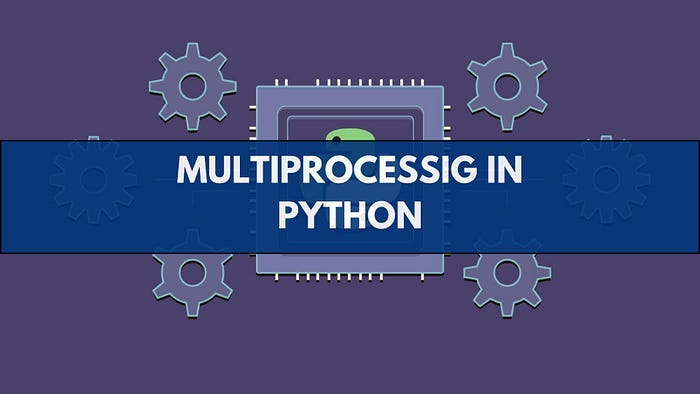
[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fa0449bd47c69&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40shahooda637%2Fmulti-processing-in-python-32d4b1c97354&user=Isha+Choudhary&userId=a0449bd47c69&source=post_page-a0449bd47c69----32d4b1c97354---------------------post_header-----------)

8 min read

·

Oct 6

65



Multiprocessing is a concept in computer programming, or can say a technique which can be used to achieve parallelism and utilizing multiple CPU cores or processors to perform tasks concurrently. This technique is particularly useful for CPU-bound tasks where the program’s performance can benefit from parallel execution. Python provides the ‘multiprocessing’ module to work with multiple processes.

In this article, we will learn all about multiprocessing in python, its theory and practical uses using code.

**What is Multiprocessing?**

Multiprocessing is a computer programming and execution paradigm that involves the simultaneous execution of multiple processes or programs by making it possible to utilize multiple CPU cores or processors within a single computer or across multiple computers based on the problem statement or setup we are working on.

Multiprocessing is a form of parallelism that aims to improve program performance and efficiency by allowing tasks to be executed concurrently, thus using the available computational resources efficiently.

There are several components of multiprocessing as discussed below.

1. Processes: Each task or unit of work or process is encapsulated within a separate process, which is an independent program execution unit. These processes can run concurrently and independently of each other.
2. Parallel Execution: Multiple processes like computations, data processing, or multiple request handling are able to be executed parallelly through multiprocessing, allowing them to perform tasks concurrently.
3. Resource Utilization: Multiprocessing is a technique developed and used to efficiently utilize the processing power of modern computers, especially multi-core processors (such as quad core or octa core), by distributing tasks across multiple CPU cores.
4. Isolation: In multiprocessing, processes are isolated from each other. Which means, that one process’s actions or failures are not affecting other processes directly. This isolated processing enhances system stability.
5. Inter-Process Communication (IPC): Mechanisms for inter-process communication, such as shared memory, pipes, queues, and sockets (we will be discussing these with code later in the article), allow processes to exchange data and coordinate their activities.
6. Scalability: Multiprocessing can be scaled to harness the power of many processors or even distributed computing clusters for solving computationally intensive problems.

In python, we can perform multiprocessing in our code by using the ‘multiprocessing’ module and related libraries and framework, making it possible for us to create parallel and concurrent applications using python programming languages.

**Difference Between Multithreading and Multiprocessing**

We have discussed the concept of multithreading already. Now, image that you are cooking in a kitchen with two of your helpers. Now, multithreading is like having these both helpers in the same kitchen working on a task. This would definitely fasten the process and 2 person are working on one task. Also, these helpers, working in the same kitchen share the same ingredients and can talk to each other, can share tasks easily. But, it can also get messy, and they might bump into each other while working, ruining all the progress.

On the other hand, multiprocessing is like having 2 helpers, working in 2 different kitchen, using their own ingredients and personal space. These helpers can work independently, without disturbing or messing up with each other’s work, but if they need to share ingredients, or talk, then would have to use phone as they are in two different kitchen.

From this example, we can understand that, multi-threading is a great technique for tasks where helpers do not need to move much, as in the processor do not need to use a lot of resources. However, multiprocessing is better for tasks that require a lot of movement, or technically a lot of resources from the cores at the same time.

If we define it in one line, multithreading primarily achieves concurrency, allowing tasks to overlap in time, while multiprocessing achieves true parallelism, as each process runs independently on a seperate CPU core or processor.

This was a general explaination for understanding the difference between the two. You can read the entire article on both the topics to know more.

The very first step to perform multiprocessing in python is to import the module.

import multiprocessing

Now, we want to use multiprocessing to run a program. To see how we do it, let’s create a sample user defined program

def test():  
 print("This is my sample program")  
  
test()  
  
Output:  
This is my sample program

Now, to call this program using multiprocessing, we will use the following code.

def test01():  
 print("This is a sample multiprocessing program")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 m = multiprocessing.Process(target=test01)  
 print("this is my main program")  
  
 #start the program  
 m.start()  
  
 #stop/terminate the program and release the resource  
 m.join()

In the code above, we are using “main” function, as it allows us to create a parent program inside which we can create a child program and also call the program we want to process using multiprocessing. In the above program, the child program prints the text “this is my main program”, while the program we call through multiprocessing prints the text, “This is a sample multiprocessing program”.

“main” program also ensures that the code runs only when the script is executed directly, not when imported as a module.

We can also use multiprocessing in other functions, performing some logical operations, an example is given below.

def square(n):  
 return n\*\*2  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 with multiprocessing.Pool(processes=5) as pool:  
 out = pool.map(square, [12,2,3,5,23,14,26,8,4,6])  
 print(out)  
  
Output:  
[144, 4, 9, 25, 529, 196, 676, 64, 16, 36]

**Parallel Execution of Functions**

**Running Multiple Processes**

We can use multiprocessing for running multiple processes on multiple cores. We create and start multiple processes, each running the function ‘workers’ (as done in the code below), with a set of arguments. We use ‘args’ to specify the list of arguments we want to use for the function.

In the code below, we are using 4 processes to process the program using the concept of multiprocessing. For this, we use the ‘Process’ method of multiprocessing module in python.

def workers(num):  
 print(f"Worker number {num}")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 processes = []  
 for i in range(4):  
 process = multiprocessing.Process(target=workers, args=(i,))  
 processes.append(process)  
 process.start()  
  
 for process in processes:  
 process.join()  
  
Output:  
Worker number 0  
Worker number 1  
Worker number 2  
Worker number 3

**Sharing Data Between Processes**

**Using ‘multiprocessing.Queue’**

‘multiprocessing.Queue’ is a class in Python programming language provided by the ‘multiprocessing’ module, which we use to perform multiprocessing in python. The ‘.Queue’ here is used to facilitate communication and data exchange among different processes in a multiprocessing program.

Some key features and purposes of the ‘multiprocessing.Queue’ can be listed as follows:

1. Inter-Process Communication: It allows multiples processes to exchange data safely by using a common data structure, the queue. Each process can put data into the queue or retrieve data from it.
2. Thread Safe: It is designed in a way which makes it thread-safe and process-safe, which means that it can be safely used in a multi=threaded or multi-process environment without race conditions or data corruption.
3. FIFO: The queue always follows the FIFO i.e., First In First Out principle, which means that the first item placed in the queue will be the first one to come out also.
4. Blocking Operations: When a process tries to retrieve an item from an empty queue, or insert into a filled queue, the operation will block the until condition changes. This process/behavior is useful for synchronization between processes.

‘multiprocessing.Queue’ is a valuable tool when working with parallel or concurrent processing in Python, especially in situations where multiple processes need to share and coordinate data.

An example of how to use the ‘multiprocessing.Queue’ is given in the cell below where the producer process produces or puts the data into the queue, and the consumer process consumes the data from the queue. The ‘multiprocessing.Queue’ ensures safe data exchange between the two processes.

import multiprocessing  
import time  
  
#defining our function we want to apply multiprocessing on  
#01 the producer function to add elements in the queue  
def producer(q):  
 for item in range(5):  
 q.put(item)  
 print(f"Produced: {item}")  
  
  
#02 consumer function to get elements from the queue  
def consumer(q):  
 while True:  
 item = q.get()  
 if item is None:  
 break  
 print(f"Consumed: {item}")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 #creating a multiprocessing queue  
 q = multiprocessing.Queue()  
  
 #creating the producer and consumer processes  
 producer\_process = multiprocessing.Process(target=producer, args=(q,))  
 consumer\_process = multiprocessing.Process(target=consumer, args=(q,))  
  
 #starting the processes  
 producer\_process.start()  
 consumer\_process.start()  
  
 #finish the producer, signal the consumer to exit  
 producer\_process.join()  
 q.put(None) #signaling the consumer about no more data in the queue  
 consumer\_process.join()  
  
  
Output:  
Produced: 0  
Produced: 1  
Consumed: 0Produced: 2  
  
Produced: 3Consumed: 1  
Produced: 4  
Consumed: 2  
  
Consumed: 3  
Consumed: 4

**Pooling Processes**

**Using ‘multiprocessing.Pool’**

‘multiprocessing.Pool’ is another class provided by the module ‘multiprocessing’ in python. It is used for creating and managing a pool of worker processes that can execute tasks in parallel. This class simplifies the process of parallelizing tasks and distributing them among multiple processes.

Some features and uses of ‘multiprocessing.Pool’ are listed below.

* Parallel Processing: ‘multiprocessing.Pool’ is designed to enable parallel processing in python. It allows us to perform multiple tasks concurrently, taking the advantage of multi-core CPUs and potentially improving the performance of CPU-bound operations.
* Task Distribution: We can use ‘multiprocessing.Pool’ to distribute tasks across a pool of worker processes. It abstracts the management of work processes, making it easier to work with parallelism.
* Resource Management: The pool automatically manages the creation and destruction of worker processes based on the available CPU cores. We can specify the number of worker processes to use, and the pool handles the rest.
* Load Balancing: The pool efficiently distributes tasks among worker processes, ensuring that each worker gets an equal share of workload. This load balancing is essential for optimizing parallel processing
* Result Collection: We can collect results from the worker processes. This is particularly useful when tasks produce results that need to be aggregated or processes further.
* Context Manager: It can also work as a context manager, ensuring that the pool is properly closed and worker processes are terminate when it is not longer needed.

The following cells shows an example code to use ‘multiprocessing.Pool’ for multiprocessing the processes through a simple function returning the square of the given number.

def square(x):  
 return x\*\*2   
  
if \_\_name\_\_=='\_\_main\_\_':  
  
 #create a multiprocessing pool with 4 worker processes  
 with multiprocessing.Pool(processes=4) as pool:  
 #distribute the tasks among worker processes and collect results  
 results = pool.map(square, [2,5,3,2,1,7,8,5,6,2,2,3])  
  
 #print the results  
 print(results)  
  
Output:  
[4, 25, 9, 4, 1, 49, 64, 25, 36, 4, 4, 9]

**Advanced Multiprocessing**

Advanced Multiprocessing in python involves utilizing the ‘multiprocessing’ module to handle more complex parallel processing tasks and scenarios. While basic multiprocessing, as discussed involves creating a pool of worker processes to execute functions in parallel, advanced multiprocessing techniques allow us to address more intricate use cases.

Some advanced multiprocessing concepts and techniques are listed below with description for each.

1. Multiprocessing Managers — Python’s ‘multiprocessing’ module includes managers that enable us to share python objects like lists, dictionaries, and custom objects, among processes. This is useful when we need to maintain shared state or data across multiple processes.
2. Process Communication — Advanced multiprocessing often involves more sophisticated communication between processes. For this, we can use mechanisms like ‘multiprocessing.Queue’, ‘multiprocessing.Pipe’, or ‘multiprocessing.Event’ for inter-process communication (IPC).
3. Parallel Map\_reduce — We can implement parallel map-reduce patterns using multiprocessing. Divide a large task into smaller parts, distribute them to worker processes for mapping, and then collect and process the results for reduction. This is particularly valuable for data processing and data analysis tasks.
4. Asynchronous Processing — Combining multiprocessing with asynchronous programming using libraries like ‘asyncio’, which allows us to manage concurrent I/O-bound tasks efficiently. asynchronous programming can be used in combination with multiprocessing for tasks involving network communication or file I/O.
5. Shared Memory — In some cases, we might need to share large datasets or memory-mapped files between processes efficiently. The ‘multiprocessing.shared\_memory’ module can help with this, allowing multiple processes to access shared memory ragions.
6. Dynamic Process Management — In advance scenarios, we might need to dynamicallt create, manage, and terminate processes based on the workload or specific requirements. This can also involve using process pools, dynamic process creation, and process monitoring.
7. Error Handling and Fault Tolerance — In complex multiprocessing setups, we should consider error handling and fault tolerance strategies. This includes handling exceptions, monitoring process health, and implementing strategies for process recovery.

*Thanks for reading!*

*Have you ever tried clicking the clap twice? Do it to experience something new on medium now.*