Concurrency in Python

Last Updated : 02 Oct, 2023

Concurrency is one approach to drastically increase the performance of your Python programs. Concurrency allows several processes to be completed concurrently, maximizing the utilization of your system's resources. Concurrency can be achieved in Python by the use of numerous methods and modules, such as threading, multiprocessing, and asynchronous programming. In this article, we will learn about **What is concurrency in Python**, the processes required to implement it, some good examples, and the output results.

What is Concurrent Programming?

It refers to the ability of the computer to manage multiple tasks at the same time. These tasks might not be compulsory to execute at the exact same moment but they might interleaved or executed in overlapping periods of time. The main goal of concurrency is to handle multiple user inputs, manage several I/O tasks, or process multiple independent tasks.

What is Parallelism?

Parallelism is a subset of concurrency where tasks or processes are executed simultaneously, As we know concurrency is about dealing with multiple tasks, whereas parallelism is about executing them simultaneously to speed computation. The primary goal is to improve computational efficiency and speed up the [performance of the system.

- **Thread-Based Concurrency:** Threading is a technique for producing lightweight threads (sometimes known as "worker threads") in a single step. Because these threads share the same memory region, they are ideal for I/O-bound activities.
- **Process-Based Concurrency:**
  Multiprocessing
  entails the execution of several processes, each with its own memory space. Because it can use several CPU cores, this is appropriate for CPU-bound activities.
- **Coroutine-Based Concurrency**: Asynchronous programming, with
  'asyncio'
  , 'async', and 'await' keywords, is ideal for efficiently managing I/O-bound processes. It enables tasks to pause and hand over control to other tasks during I/O operations without causing the entire program to crash.

Concurrency vs. Parallelism

Concurrency: It refers to the execution of many tasks in overlapping time periods, but not necessarily concurrently. It is appropriate for I/O-bound operations that frequently rely on external resources such as files or network data.

Parallelism: On the other hand, is running many processes at the same time, generally using multiple CPU cores. It is best suited for CPU-intensive jobs requiring lengthy processing.

Steps to Implement Concurrency in Python

1. Select the Appropriate Approach:Determine whether your software is CPU or I/O bound. Threading is best for I/O-bound operations whereas multiprocessing is best for CPU-bound workloads.
2. Import the Appropriate Libraries: Import the threading, multiprocessing, or asyncio libraries, depending on your method.
3. Create Worker Roles: Define the functions or methods that represent the jobs you want to run simultaneously.
4. Create Threads, Processes, and Tasks: To execute your worker functions concurrently, create and launch threads, processes, or asynchronous tasks.
5. Control Synchronization (if necessary): When employing threads or processes, use synchronization methods such as locks or semaphores to prevent data corruption in shared resources.
6. Wait for the job to be finished: Make sure your main application waits for all concurrent jobs to finish before continuing.

Examples of Concurrency in Python

Thread-Based Concurrency

Import the time and threading modules. Print_numbers and print_letters are two functions that simulate time-consuming jobs. To imitate a time-consuming task, these programs print numbers and letters with a 1-second delay between each print. Create two Thread objects, thread1 and thread2, and use the target argument to assign each to one of the functions (print_numbers and print_letters). Use the start() function to start both threads. This starts the parallel execution of the functions. Use the join() method for each thread to ensure that the main thread waits for threads 1 and 2 to finish before continuing. This synchronization step is required to avoid the main program ending prematurely. Finally, print a message indicating that both threads are complete.

Python3

```
import threading
import time
# Function to simulate a time-consuming task
def print_numbers():
    for i in range(1, 6):
        print(f"Printing number {i}")
        time.sleep(1)  # Simulate a delay of 1 second
# Function to simulate another task
def print_letters():
    for letter in 'Geeks':
        print(f"Printing letter {letter}")
        time.sleep(1)  # Simulate a delay of 1 second
# Create two thread objects, one for each function
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

# Start the threads
thread1.start()
thread2.start()

# The main thread waits for both threads to finish
thread1.join()
thread2.join()

print("Both threads have finished.")
```

Output:

```
Printing number 1
Printing letter G
Printing number 2Printing letter e
Printing letter e
Printing number 3
Printing letter kPrinting number 4
Printing letter sPrinting number 5
Both threads have finished.
```

When you run this code, you'll notice that the two threads execute the two functions, print_numbers and print_letters, concurrently. As a result, the output from both methods will be interleaved, and each function will introduce a 1-second delay between prints, imitating a time-consuming operation. The join() functions ensure that the main program awaits the completion of both threads before printing the final message.

Process-Based Concurrency

Bring the multiprocessing module into the program. Define a square(x) function that squares a given number.Make a number list with the integers [1, 2, 3, 4, 5].Using multiprocessing, create a multiprocessing pool. To manage worker processes, use Pool().Apply the square function to each number concurrently with pool.map() to distribute the work among processes.Get a list of the squared result and then print the values that have been squared

Python3

```
import multiprocessing
# Function to square a number
def square(x):
    return x * x
```

```
if __name__ == "__main__":
    # Define a list of numbers
    numbers = [1, 2, 3, 4, 5]

    # Create a multiprocessing pool
    with multiprocessing.Pool() as pool:
        # Use the map method to apply the 'square ' function to each number in parallel
        results = pool.map(square, numbers)
    # Print the results
    print(results)
```

Output:

```
[1, 4, 9, 16, 25]
```

This code parallelizes the square function on each number, utilizing many CPU cores for quicker execution of CPU-bound operations.

Coroutine-Based Concurrency (using asyncio)

The code is broken down as follows: For asynchronous programming, import the asyncio module.Using await asyncio.sleep(1), define an asynchronous function greet(name) that simulates a greeting with a 1-second delay.Define the asynchronous main function main().To schedule the greet function for each name concurrently, use asyncio.gather().Run the main asynchronous function using asyncio in the if __name__ == “__main__”: block.run(main()).

Python3

```
import asyncio
async def greet(name):
    await asyncio.sleep(1)
    print(name)
async def main():
    await asyncio.gather(greet("Geeks"), greet("For"), greet("Geeks"))
# If in a Jupyter notebook or IPython environment:
import nest_asyncio
nest_asyncio.apply()
if __name__ == "__main__":
    asyncio.create_task(main())
```

Output:

```
Geeks
For
Geeks
```

When you run this code, you’ll notice that the greet function is called for each name in turn, and it waits 1 second before printing a greeting. The output will show greetings for **“Geeks,” “For,” and “Geeks”** in an interleaved fashion, demonstrating the non-blocking nature of Python asynchronous programming.

Finally, whether your programs are I/O-bound or CPU-bound, introducing concurrency in Python can dramatically improve their performance. You may improve the efficiency and responsiveness of your Python programs by selecting the proper technique and leveraging packages such as threading, multiprocessing, and asyncio

M sisodi...

Next Article

Enqueue in Queues in Python

Similar Reads

Python | Merge Python key values to list
Sometimes, while working with Python, we might have a problem in which we need to get the values of dictionary from several dictionaries to be encapsulated into one dictionary. This type of problem can be common in domains in

4 min read

Python | Visualizing O(n) using Python
Introduction Algorithm complexity can be a difficult concept to grasp, even presented with compelling mathematical arguments. This article presents a tiny Python program that shows the relative complexity of several typical functions.

3 min read

Python | PRAW - Python Reddit API Wrapper
PRAW (Python Reddit API Wrapper) is a Python module that provides a simple access to Reddit's API. PRAW is easy to use and follows all of Reddit's API rules.The documentation regarding PRAW is located here.Prerequisites: Basic

3 min read

Python program to check if the list contains three consecutive common numbers in Python
Our task is to print the element which occurs 3 consecutive times in a Python list. Example : Input : [4, 5, 5, 5, 3, 8] Output : 5 Input : [1, 1, 1, 64, 23, 64, 22, 22, 22] Output : 1, 22 Approach : Create a list.Create a loop for range size –

3 min read

Python | Convert LaTeX Matrices into SymPy Matrices using Python
A matrix is a rectangular two-dimensional array of data like numbers, characters, symbols, etc. which we can store in row and column format. We can perform operations on elements by using an index (i,j) where 'i' and 'j' stand for an

5 min read

Filter List of Python Dictionaries by Key in Python
Filtering a list of dictionaries in Python based on a specific key is a common task in data manipulation and analysis. Whether you're working with datasets or dictionaries, the ability to extract relevant information efficiently is crucial. In

3 min read

How to Run Another Python script with Arguments in Python
Running a Python script from another script and passing arguments allows you to modularize code and enhance reusability. This process involves using a subprocess or os module to execute the external script, and passing

3 min read

Using Python Environment Variables with Python Dotenv
Python dotenv is a powerful tool that makes it easy to handle environment variables in Python applications from start to finish. It lets you easily load configuration settings from a special file (usually named .env) instead of hardcoding

3 min read

Python program to build flashcard using class in Python
In this article, we will see how to build a flashcard using class in python. A flashcard is a card having information on both sides, which can be used as an aid in memoization. Flashcards usually have a question on one side and an answer

2 min read

Run One Python Script From Another in Python
In Python, we can run one file from another using the import statement for integrating functions or modules, exec() function for dynamic code execution, subprocess module for running a script as a separate process, or os.system()

5 min read