Synchronous Programming	Multithreading Programming	Multiprocessing Programming	Asynchronous Programming
Define: Synchronous programming is a programming model where operations take place sequentially	**Define:** Multithreading is a programming technique that allows multiple threads of execution within a single process	**Define:** Multiprocessing is a programming technique that involves the execution of multiple processes simultaneously.	**Define:** Asynchronous programming is a programming paradigm that allows tasks to be executed concurrently without blocking the execution of the main program.

Example Code:
 	
import requests
from program_timer import timer
 	
URL = "https://httpbin.org/uuid"
 	
def fetch_uuid(session, url):
  with session.get(url) as response:
 	
print(response.json()['uuid'])
 	
@timer(1,1)
def main():
  with requests.Session() as session:
  for _ in range(100):
  fetch_uuid(session, URL)
	Example Code:
	
	import requests
	from program_timer import timer
	from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
	
	URL = "https://httpbin.org/uuid"
	
	def fetch_uuid(session, url):
	  with session.get(url) as response:
	
	print(response.json()['uuid'])
	
	@timer(1,5)
	def main():
	  with ProcessPoolExecutor(max_workers=100) as executor:
	  with requests.Session() as session:
	
	executor.map(fetch_uuid, [session] * 100, [URL] * 100)
	
	executor.shutdown(wait=True)
		Example Code:
		
		import requests
		from program_timer import timer
		from multiprocessing.pool import Pool
		
		URL = "https://httpbin.org/uuid"
		
		def fetch_uuid(session, url):
		  with session.get(url) as response:
		
		print(response.json()['uuid'])
		
		# @timer(1,1)
		def main():
		  with Pool() as pool:
		  with requests.Session() as session:
		
		pool.starmap(fetch_uuid, [(session, URL) for _ in range(100)])
		
		if __name__ == "__main__":
		  main()
			Example Code:
			
			import asyncio
			import aiohttp
			from program_timer import timer
			from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
			
			URL = "https://httpbin.org/uuid"
			
			async def fetch_uuid(session, url):
			  async with session.get(url) as response:
			  json_data = await response.json()
			  print(json_data['uuid'])
			
			async def func():
			  async with aiohttp.ClientSession() as session:
			  tasks = [fetch_uuid(session, URL) for _ in range(100)]
			  await asyncio.gather(*tasks)
			
			@timer(1,1)
			def main():
			  asyncio.run(func())

Create timer Function:
 	
import timeit
 	
def timer(number, repeat):
def wrapper(func):
runs = timeit.repeat(func, number=number, repeat=repeat)
print(sum(runs) / len(runs))
return wrapper
	Create timer Function:
	
	import timeit
	
	def timer(number, repeat):
	def wrapper(func):
	runs = timeit.repeat(func, number=number, repeat=repeat)
	print(sum(runs) / len(runs))
	return wrapper
		Create timer Function:
		
		import timeit
		
		def timer(number, repeat):
		def wrapper(func):
		runs = timeit.repeat(func, number=number, repeat=repeat)
		print(sum(runs) / len(runs))
		return wrapper
			Create timer Function:
			
			import timeit
			
			def timer(number, repeat):
			def wrapper(func):
			runs = timeit.repeat(func, number=number, repeat=repeat)
			print(sum(runs) / len(runs))
			return wrapper

Install requirements.txt: requests	**Install requirements.txt:** requests	**Install requirements.txt:** requests	**Install requirements.txt:** requests aiohttp
Analysis: The execution time is generally longer because tasks are executed sequentially. It's simple to implement but not efficient for CPU-bound tasks.	**Analysis:** Utilizes multiple threads to perform tasks concurrently, reducing overall execution time. Suitable for I/O-bound tasks but may not be effective for CPU-bound tasks due to Python's Global Interpreter Lock (GIL).	**Analysis:** Executes tasks in parallel processes, beneficial for CPU-bound tasks as it leverages multiple CPU cores. However, inter-process communication overhead may impact performance.	**Analysis:** Handles multiple tasks concurrently without blocking, making it efficient for I/O-bound tasks. However, complex to implement and may require understanding of asynchronous programming concepts.
Execution Time: 29.7765673 seconds	**Execution Time:** 13.4534679823 seconds	**Execution Time:** 12.32436798 seconds	**Execution Time**: 2.534768347 seconds
Conclusion: Synchronous execution is straightforward but lacks efficiency, especially for CPU-bound tasks.	**Conclusion:** Multithreading improves performance for I/O-bound tasks but may face limitations due to the GIL.	**Conclusion:** Multiprocessing is effective for CPU-bound tasks but requires managing inter-process communication.	**Conclusion:** Asynchronous programming offers high concurrency for I/O-bound tasks but is complex to implement and understand.