## **Diff between synchronous vs asynchronous in python**

In Python, synchronous and asynchronous programming represent two different approaches to handling tasks, especially those that involve I/O operations like file reading/writing, network requests, or database queries.

**Synchronous Programming**

1. **Blocking Nature**: In synchronous programming, tasks are executed sequentially. Each task must complete before the next one begins.
2. **Simple to Understand**: The flow of code is straightforward, resembling the way we write standard linear scripts.
3. **Efficiency**: Suitable for CPU-bound tasks where operations are performed one after the other without needing to wait for external resources.
4. **Example**:

import time

def sync\_task():

print("Task started")

time.sleep(2) # Simulates a blocking I/O operation

print("Task finished")

def main():

for \_ in range(3):

sync\_task()

main()

This will execute sync\_task three times in sequence, with each call blocking the execution for 2 seconds.

**Asynchronous Programming**

1. **Non-Blocking Nature**: Asynchronous programming allows tasks to run concurrently, not blocking the main thread while waiting for I/O operations.
2. **Efficient for I/O-bound Tasks**: Great for tasks like network requests, database queries, or file operations where waiting times are significant.
3. **Complexity**: Requires understanding of concepts like event loops, futures, and coroutines.
4. **Example**:

import asyncio

async def async\_task():

print("Task started")

await asyncio.sleep(2) # Simulates a non-blocking I/O operation

print("Task finished")

async def main():

tasks = [async\_task() for \_ in range(3)]

await asyncio.gather(\*tasks)

asyncio.run(main())

This will initiate all three async\_task instances almost simultaneously, allowing them to run concurrently, with each task taking 2 seconds without blocking others.

**Key Differences**

1. **Execution Flow**:
   * **Synchronous**: Linear and predictable. Each task runs to completion before the next begins.
   * **Asynchronous**: Tasks can start and pause, allowing other tasks to run in the meantime. This is managed by an event loop.
2. **Blocking vs. Non-Blocking**:
   * **Synchronous**: Operations block the execution of further code until they complete.
   * **Asynchronous**: Operations can pause (using await), allowing other operations to run concurrently.
3. **Performance**:
   * **Synchronous**: Can be less efficient for I/O-bound tasks due to idle waiting.
   * **Asynchronous**: Can be more efficient for I/O-bound tasks, as the program can handle other tasks while waiting for I/O operations to complete.
4. **Code Complexity**:
   * **Synchronous**: Easier to write and understand.
   * **Asynchronous**: Requires understanding of async/await patterns, event loops, and can lead to more complex code.

**Use Cases**

* **Synchronous**: Best for simple, linear tasks, or CPU-bound operations where tasks don't have to wait for I/O.
* **Asynchronous**: Ideal for I/O-bound tasks, such as web servers handling multiple requests, reading/writing files, or making network calls concurrently.

Choosing between synchronous and asynchronous approaches depends on the specific requirements of your task and the nature of the operations involved.

## **---------------------------------------------------------------------------------------------------------------------**

## **Difference Between Asynchronous and Multi-Threading Programming**

* Asynchronous programming allows only one part of a program to run at a specific time.
* Consider three functions in a Python program: fn1(), fn2(), and fn3().
* In asynchronous programming, if fn1() is not actively executing (e.g., it’s asleep, waiting, or has completed its task), it won’t block the entire program.
* Instead, the program optimizes CPU time by allowing other functions (e.g., fn2()) to execute while fn1() is inactive.
* Only when fn2() finishes or sleeps, the third function, fn3(), starts executing.
* This concept of asynchronous programming ensures that one task is performed at a time, and other tasks can proceed independently.
* In contrast, in multi-threading or multi-processing, all three functions run concurrently without waiting for each other to finish.
* With asynchronous programming, specific functions are designated as asynchronous using the async keyword, and the asyncio Python library helps manage this asynchronous behavior.

## **Asyncio in python**

asyncio is a Python library used to write concurrent code using the async/await syntax. It provides a way to handle asynchronous I/O, event loops, coroutines, tasks, and more. Here's an overview of key concepts and how to use asyncio:

**Key Concepts**

1. **Event Loop**: The core of every asyncio application. It runs asynchronous tasks and callbacks, performs network IO operations, and runs subprocesses.
2. **Coroutine**: Special functions defined with async def. They can use await to yield control back to the event loop, allowing other tasks to run.
3. **Task**: A wrapper for coroutines. It allows the event loop to execute the coroutine concurrently with other tasks.
4. **Future**: A low-level awaitable object that represents a result that may not be available yet.

**Basic Usage**

Here's a simple example to demonstrate the basics of asyncio:

import asyncio

async def say\_hello():

print("Hello")

await asyncio.sleep(1)

print("World")

async def main():

await say\_hello()

# Run the main coroutine

asyncio.run(main())

**Running Multiple Coroutines Concurrently**

You can run multiple coroutines concurrently using asyncio.gather or by creating tasks:

import asyncio

async def say(text, delay):

await asyncio.sleep(delay)

print(text)

async def main():

task1 = asyncio.create\_task(say("Hello", 2))

task2 = asyncio.create\_task(say("World", 1))

await task1

await task2

asyncio.run(main())

**Common Functions and Methods**

* asyncio.run(coro): Run a coroutine.
* asyncio.create\_task(coro): Schedule the execution of a coroutine as a task.
* await asyncio.gather(\*coros): Run multiple coroutines concurrently and wait for them to finish.
* await asyncio.sleep(delay): Pause the execution of the current coroutine for a given number of seconds.
* await future: Wait for a future to complete and return its result.

**Example: Asynchronous HTTP Requests**

Using aiohttp library with asyncio to make asynchronous HTTP requests:

import aiohttp

import asyncio

async def fetch(session, url):

async with session.get(url) as response:

return await response.text()

async def main():

async with aiohttp.ClientSession() as session:

html = await fetch(session, 'http://example.com')

print(html)

asyncio.run(main())

**Error Handling**

Handling exceptions in coroutines:

import asyncio

async def faulty\_coroutine():

raise ValueError("An error occurred!")

async def main():

try:

await faulty\_coroutine()

except ValueError as e:

print(f"Caught an exception: {e}")

asyncio.run(main())

asyncio is powerful for writing concurrent and asynchronous programs, especially when dealing with I/O-bound and high-level structured network code.