**Multithreading in Python**

Multithreading in Python allows a program to run multiple threads (smaller units of a process) concurrently. However, due to the **Global Interpreter Lock (GIL)**, Python threads do not execute in parallel for CPU-bound tasks. Instead, they are useful for I/O-bound tasks.

**1. Understanding the GIL and Its Impact**

* The **GIL** ensures that only **one thread** executes Python bytecode at a time.
* This means Python threads are **not useful for CPU-bound tasks** because they can’t truly execute in parallel.
* However, **I/O-bound tasks** (e.g., network requests, file I/O, database queries) benefit from multithreading because they spend most of their time waiting for external resources

**3. When to Use Multithreading**

**✅ Good for I/O-bound tasks**

* Downloading multiple files
* Making API requests
* Reading/writing files
* Handling multiple user requests in web servers

**❌ Not good for CPU-bound tasks**

* Image processing
* Complex mathematical computations
* Machine learning model training
  + Use multiprocessing for true parallelism.

**Multithreading vs Multiprocessing**

| **Feature** | **Multithreading (threading)** | **Multiprocessing (multiprocessing)** |
| --- | --- | --- |
| **Best for** | I/O-bound tasks (e.g., API calls, file I/O) | CPU-bound tasks (e.g., heavy computations) |
| **Uses GIL?** | Yes (one thread runs Python bytecode at a time) | No (separate processes have their own memory) |
| **Runs in parallel?** | No (due to GIL) | Yes (each process runs independently) |
| **Shared memory?** | Yes (threads share memory) | No (each process has separate memory) |

**🔹 Use threading for I/O-bound tasks and multiprocessing for CPU-bound tasks.**

In Python's asyncio module, **tasks** and **coroutines** are closely related but distinct concepts. Here's how they differ:

**1. Coroutine**

* A coroutine is a special type of function in Python defined using async def.
* It represents a computation that can be paused (await) and resumed later.
* Coroutines do not start running immediately when called. Instead, they return a coroutine object that must be awaited or scheduled to execute.

**2. Task**

* A **task** is a wrapper around a coroutine that runs it concurrently.
* It is created using asyncio.create\_task(coroutine).
* Unlike a coroutine, a task starts running **immediately** after creation.
* Tasks allow multiple coroutines to run concurrently.

**Key Differences**

| **Feature** | **Coroutine** | **Task** |
| --- | --- | --- |
| Execution | Doesn't run until awaited | Starts immediately upon creation |
| Object Type | Coroutine object | Task object |
| Scheduling | Runs in order of await | Runs concurrently |
| Example API | async def func(): ... | asyncio.create\_task(func()) |