🔷 What is THREADING (Big Picture First)

At the deepest level:

A thread is a smaller unit of a process that can run independently but shares the same memory.

Your normal Python program is:

- **1 process**
- **1 thread (main thread)** by default

When you use threading, you create **multiple threads inside the same process.**

🔷 LEVEL 1 — Simple Example of Threading

Look at this basic code first:

```
import threading
import time

def task():
    for i in range(3):
        print("Task running")
        time.sleep(1)

t = threading.Thread(target=task)
t.start()

print("Main thread continues”)
```

What happens (simple view):

Output will look like:

```
Task running
Main thread continues
Task running
Task running
```

- Your program did **two things at the same time**
    - Main program printed message
    - Thread was running in background

🔷 LEVEL 2 — Core Concepts of Threading

🔷 1. Process vs Thread

Think like this:

Concept	Meaning
Process	Entire running program
Thread	A part of that program running in parallel

Example analogy:

- Your computer = city
- Process = a building
- Thread = workers inside that building

All workers (threads) share:

- Same tools
- Same resources
- Same memory

🔷 2. Main Thread

Every Python program starts with **one thread automatically**:

```
print(“Hello")
```

This runs in the **main thread**.

You can check this:

```
import threading

print(threading.current_thread().name)
```

Output:

```
MainThread
```

🔷 3. Creating a New Thread

This line creates a new thread object:

```
t = threading.Thread(target=task)
```
Here:

- `task` is NOT called
- It is just passed as a function for the new thread to run

This is like telling Python:

"Hey, create a new worker that will run this function."

🔷 4. Starting the Thread

```
t.start()
```
This is when:

- New thread is actually created
- Python schedules it to run alongside the main thread

Before `start()`, no new thread exists in execution.

🔷 LEVEL 3 — What happens internally? (Deep view)

When you call `t.start()`:

Python does something like this internally:

1. Creates a new thread in the operating system
2. Assigns your `task()` function to that thread
3. Both threads run in parallel

Now you have:

- **Thread 1 → MainThread**

- **Thread 2 → WorkerThread (your task)**

They run like this (conceptually):

```
Time →
MainThread:  print("Main thread continues")  -----> end
WorkerThread: task() → sleep → task() → sleep → task()
```

They take turns executing.

🔷 LEVEL 4 — Real Execution Model (Very Important)

Even though threads run "in parallel", in Python:

Only ONE thread executes Python code at a time.

This happens because of:

🔷 GIL — Global Interpreter Lock

GIL means:

Python allows only one thread to execute Python bytecode at a time.

So threads in Python are NOT true parallel for CPU work.

They are good for:

- Waiting tasks (I/O tasks)
- Network calls
- File reading
- API requests
- Database queries

They are BAD for:

- Heavy calculations
- Big loops
- Machine learning training
- Image processing

🔷 LEVEL 5 — Example that shows GIL effect

CPU-heavy task:

```
import threading

def count():
    total = 0
    for i in range(10_000_000):
        total += 1

t1 = threading.Thread(target=count)
t2 = threading.Thread(target=count)

t1.start()
t2.start()

t1.join()
t2.join()

print(“Done")
```

Even with two threads:

- It will **not run twice as fast**
- Because GIL allows only one thread to compute at a time

🔷 LEVEL 6 — Where Threading Actually Shines (Core Idea)

Best example: downloading files

```
import threading
import time

def download(file):
    print(f"Downloading {file}")
    time.sleep(2)
    print(f"Finished {file}")
```

```
files = ["a.mp4", "b.mp4", "c.mp4"]

threads = []
for f in files:
    t = threading.Thread(target=download, args=(f,))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print("All downloads complete”)
```

What happens?

Instead of 6 seconds, this finishes in ~2 seconds.

Why?

Because while one thread is waiting (sleep = network wait), another thread runs.

This is the **real power of threading in Python.**

🔷 LEVEL 7 — Important Methods in Threading

1️⃣ start()

Starts the thread.

2️⃣ join()

Waits for thread to finish.

Example:

```
t.start()
t.join()   # main thread waits here
print("Done")
```

Without `join()`:

- Main program might finish before thread completes.

🔷 LEVEL 8 — Mental Model of Threading (Best analogy)

Think of:

- Python = kitchen
- Main thread = head chef
- Worker thread = helper chef

If helper chef is:

- Cutting vegetables (I/O type work) → good
- Cooking on same stove (CPU work) → conflict

That conflict is **GIL.**

🔷 One-Line Core Meaning of Threading

“Threading allows multiple parts of your program to run seemingly in parallel, mainly to avoid waiting time in I/O tasks, but due to GIL it is not truly parallel for CPU work in Python.”