**Concurrency vs Parallelism (short & precise)**

* **Concurrency**: multiple tasks *make progress* at the same time. They may be interleaved on a single CPU core (time-slicing) or run on multiple cores. Useful for structuring programs (e.g., handling many I/O operations).
* **Parallelism**: multiple tasks actually executing **simultaneously** on different CPU cores. This is necessary to speed up CPU-bound work.

## 🧠 The Core Idea

Both concurrency and parallelism deal with doing multiple things at once, but the **meaning of “at once” is different**.

| **Concept** | **Meaning** | **Example** | **CPU cores used** |
| --- | --- | --- | --- |
| **Concurrency** | Managing multiple tasks at the same time — they take turns making progress. | A single waiter handling several tables — takes one order, then another, delivers drinks in between. | Can be **1** core (interleaved execution) or more. |
| **Parallelism** | Executing multiple tasks literally at the same time — true simultaneous execution. | Several waiters, each handling their own table at the same time. | **Multiple** cores working together. |

# 🧵 1. Threads (Multithreading)

### ✅ What is a Thread?

A **thread** is the smallest unit of execution within a process.  
Every Python program starts with a **main thread**, but we can create additional threads to perform tasks concurrently.

### ⚙️ Key characteristics

* **Lightweight**: threads share the same memory space of the parent process, so creating and switching between them is faster than processes.
* **Shared Memory**: since all threads share global variables and data structures, communication between threads is easy — you can access the same variables directly.
* **Danger**: shared access means race conditions can happen if multiple threads modify the same variable simultaneously.

### 🔸 When are threads useful?

Threads are best suited for **I/O-bound tasks** — operations that spend most of their time waiting (e.g., waiting for a file to download, waiting for a database response, etc.).

When one thread is waiting on I/O, the **GIL** is released, allowing another thread to run.  
This means multiple threads can make progress even with the GIL present.

### 💡 Example: Threading in action

import threading, time

def download\_file(name):

    print(f"Starting download: {name}")

    time.sleep(2)  # simulate waiting for I/O

    print(f"Download complete: {name}")

t1 = threading.Thread(target=download\_file, args=("file1",))

t2 = threading.Thread(target=download\_file, args=("file2",))

t1.start(); t2.start()

t1.join(); t2.join()

print("All downloads completed.")

**Why it’s faster:**  
Both downloads “wait” for the network, but while one is waiting, the other runs.  
This overlap is **concurrency**, achieved through **multithreading**.

# 🧩 2. The GIL (Global Interpreter Lock)

### 🔍 What is the GIL?

The **Global Interpreter Lock (GIL)** is a mutex (mutual exclusion lock) used by **CPython**, the default Python implementation.  
It ensures that **only one thread executes Python bytecode at a time**, even if multiple threads exist.

### 🔸 Why does the GIL exist?

It simplifies memory management inside CPython by avoiding race conditions in the interpreter itself.  
Without the GIL, every object access would require fine-grained locks — making the interpreter slower and more complex.

### ⚠️ Consequence

Because of the GIL:

* **Threads can’t run Python code truly in parallel** on multiple cores.
* CPU-bound code (e.g., heavy loops, number crunching) won’t get faster using threads.

Even if you have 8 CPU cores, Python threads will take turns holding the GIL — so your CPU-bound code effectively runs on one core at a time.

### 💡 Example: GIL effect demonstration

import threading, time

def cpu\_task():

    count = 0

    for \_ in range(10\_000\_000):

        count += 1

start = time.time()

t1 = threading.Thread(target=cpu\_task)

t2 = threading.Thread(target=cpu\_task)

t1.start(); t2.start()

t1.join(); t2.join()

end = time.time()

print(f"Total time: {end - start:.2f}s")

Even though there are two threads, this will **not** run twice as fast — because of the GIL.

### 🚀 Ways to work around the GIL

1. **Multiprocessing** – each process has its own interpreter and GIL, allowing true parallel execution.
2. **C extensions / NumPy** – some libraries release the GIL internally while performing operations in C, so multiple cores can be used.
3. **Async I/O** – I/O-bound concurrency using asyncio, which uses a single thread but switches efficiently between tasks.

# ⚙️ 3. Multiprocessing

### ✅ What is Multiprocessing?

The **multiprocessing** module lets you create separate processes, each with its own Python interpreter and GIL.  
That means processes can **run truly in parallel** across multiple CPU cores.

Each process has its own **memory space** and resources, which provides safety (no shared state) but also overhead (memory and inter-process communication).

### ⚙️ When to use Multiprocessing

* **CPU-bound tasks**: heavy computations (e.g., image processing, encryption, data analysis).
* **Independent tasks**: work that can be split into chunks and combined later.

### 💡 Example: true parallelism

from multiprocessing import Process

import time

def cpu\_task(name):

    print(f"{name} started")

    count = 0

    for \_ in range(10\_000\_000):

        count += 1

    print(f"{name} finished")

if \_\_name\_\_ == "\_\_main\_\_":

    start = time.time()

    p1 = Process(target=cpu\_task, args=("Task 1",))

    p2 = Process(target=cpu\_task, args=("Task 2",))

    p1.start(); p2.start()

    p1.join(); p2.join()

    print(f"Total time: {time.time() - start:.2f}s")

Since each process has its own GIL, both truly execute at the same time — **parallelism**.

### 🔸 Trade-offs of Multiprocessing

| **Advantage** | **Disadvantage** |
| --- | --- |
| True parallelism | Higher memory use (each process has its own copy of data) |
| Avoids GIL limitation | Communication between processes is slower (uses IPC) |
| Stable (no shared memory issues by default) | Startup overhead (creating processes is slower than threads) |

### 🧰 Communication between processes

Because processes have separate memory, you need **Inter-Process Communication (IPC)** to share data:

1. **multiprocessing.Queue** – message passing (like a thread-safe mailbox).
2. **multiprocessing.Value / Array** – share basic data with synchronization.
3. **Manager** – allows sharing of more complex data structures (lists, dicts, etc.).

# 🔒 4. Synchronization & Race Conditions

### 🧠 What is a Race Condition?

A **race condition** occurs when two or more threads/processes access and modify shared data at the same time, and the final result depends on the timing of their execution.

Example:

counter = 0

def increment():

    global counter

    for \_ in range(100000):

        counter += 1

If 10 threads run this simultaneously without any locking, they may overwrite each other’s results → counter will be less than expected.

### 🔐 Synchronization — preventing race conditions

We use synchronization primitives to ensure that **only one thread/process accesses shared data at a time**.

#### For Threads:

* threading.Lock()
* lock = threading.Lock()
* with lock:
* counter += 1
* queue.Queue() for thread-safe message passing (no need for locks).

#### For Processes:

* multiprocessing.Lock() or Value.get\_lock() for shared counters.
* multiprocessing.Queue() for passing data safely between processes.

### 🧩 Example: using Lock in threads

import threading

counter = 0

lock = threading.Lock()

def increment():

    global counter

    for \_ in range(100\_000):

        with lock:

            counter += 1  # critical section

threads = [threading.Thread(target=increment) for \_ in range(10)]

[t.start() for t in threads]

[t.join() for t in threads]

print(f"Final counter: {counter}")

Here, with lock: ensures only one thread at a time modifies counter, preventing race conditions.

### ⚙️ In multiprocessing:

from multiprocessing import Process, Value

def increment(counter):

    for \_ in range(100\_000):

        with counter.get\_lock():  # acquire lock

            counter.value += 1

if \_\_name\_\_ == "\_\_main\_\_":

    counter = Value('i', 0)  # 'i' = integer

    processes = [Process(target=increment, args=(counter,)) for \_ in range(4)]

    [p.start() for p in processes]

    [p.join() for p in processes]

    print("Final counter:", counter.value)

# 🧾 Summary Table

| **Concept** | **Memory Sharing** | **True Parallelism** | **Best For** | **Python Module** | **Needs Lock?** |
| --- | --- | --- | --- | --- | --- |
| **Threads** | Shared memory | ❌ (GIL limits) | I/O-bound tasks | threading | Yes (shared data) |
| **Processes** | Separate memory | ✅ (each process has own GIL) | CPU-bound tasks | multiprocessing | Optional (if sharing data) |
| **Async (bonus)** | Shared event loop | ❌ | High concurrency, lightweight I/O | asyncio | No (single-threaded) |

**Practical guidance (when to use)**

* **I/O-bound**: use threads (threading) or async (asyncio) — threads are simple.
* **CPU-bound**: use multiprocessing (Process, Pool) or native libraries that release the GIL.
* **Shared state**: if you need shared memory, consider multiprocessing.Value/Array or use message passing (Queue).

**Fixed & Annotated Example Scripts**

Below are cleaned versions of every file you provided, with brief notes and expected behavior. You can copy/paste and run them.

**1) threading.py — concurrency (I/O-like simulation)**

Demonstrates threads doing independent tasks concurrently (sleep simulates blocking I/O).

# threading.py

import threading

import time

def take\_orders():

    for i in range(1, 4):

        print(f"Taking order for #{i}")

        time.sleep(1)  # simulate I/O / waiting

def brew\_chai():

    for i in range(1, 4):

        print(f"Brewing chai for #{i}")

        time.sleep(2)  # simulate longer I/O

# create threads

order\_thread = threading.Thread(target=take\_orders, name="OrderThread")

brew\_thread = threading.Thread(target=brew\_chai, name="BrewThread")

order\_thread.start()

brew\_thread.start()

# wait for both to finish

order\_thread.join()

brew\_thread.join()

print("All orders taken and chai brewed")

**What it shows:** concurrency — output from both functions interleaves. Good for I/O-bound work.

**2) multiprocessing.py — multiple processes**

Creates processes to run tasks in parallel (each process independent).

# multiprocessing.py

from multiprocessing import Process

import time

def brew\_chai(name):

    print(f"Start of {name} chai brewing")

    time.sleep(3)

    print(f"End of {name} chai brewing")

if \_\_name\_\_ == "\_\_main\_\_":

    chai\_makers = [

        Process(target=brew\_chai, args=(f"Chai Maker #{i+1}",))

        for i in range(3)

    ]

    # start the processes

    for p in chai\_makers:

        p.start()

    # wait for all processes

    for p in chai\_makers:

        p.join()

    print("All chai served")

**What it shows:** parallel execution across processes (if you have multiple CPU cores).

**3) processexample.py — measuring multiprocessing time (CPU-bound demo)**

This uses processes for a CPU-heavy counting loop.

# processexample.py

from multiprocessing import Process

import time

def crunch\_number():

    print("Started the count process ...")

    count = 0

    for \_ in range(10\_000\_000):  # 10 million

        count += 1

    print("Ending the count process ...")

if \_\_name\_\_ == "\_\_main\_\_":

    start = time.time()

    p1 = Process(target=crunch\_number)

    p2 = Process(target=crunch\_number)

    p1.start()

    p2.start()

    p1.join()

    p2.join()

    end = time.time()

    print(f"Total time with multi-processing is {end - start:.2f} seconds")

    # Optional sequential comparison:

    # sequential\_start = time.time()

    # crunch\_number()

    # crunch\_number()

    # sequential\_end = time.time()

    # print(f"Total time sequential is {sequential\_end - sequential\_start:.2f} seconds")

**What it shows:** multiprocessing can speed up CPU-bound work because each process runs on separate CPU core(s).

**4) threads.py — CPU work inside threads (shows GIL effect)**

This shows CPU-bound work inside threads — you will likely **not** get true speedup in CPython.

# threads.py

import threading

import time

def brew\_chai():

    print(f"{threading.current\_thread().name} start brewing...")

    count = 0

    # CPU-bound work

    for \_ in range(10\_000\_000):

        count += 1

    print(f"{threading.current\_thread().name} finish brewing")

thread\_1 = threading.Thread(target=brew\_chai, name="Barista\_1")

thread\_2 = threading.Thread(target=brew\_chai, name="Barista\_2")

start = time.time()

thread\_1.start()

thread\_2.start()

thread\_1.join()

thread\_2.join()

end = time.time()

print(f"total time taken: {end - start:.2f} seconds")

**What to expect:** because of the GIL, two threads performing pure Python counting usually take roughly the same time as running them sequentially (no real parallel speedup).

**5) thread\_download.py — threads for network downloads (I/O-bound example)**

Threads speed this up because network I/O releases the GIL.

# thread\_download.py

import requests

import threading

import time

def download(url):

    print(f"Start downloading from {url}")

    resp = requests.get(url)

    print(f"Finish downloading from {url}, size: {len(resp.content)} bytes")

urls = [

    "https://httpbin.org/image/jpeg",

    "https://httpbin.org/image/png",

    "https://httpbin.org/image/svg",

]

start = time.time()

threads = []

for url in urls:

    t = threading.Thread(target=download, args=(url,))

    t.start()

    threads.append(t)

for t in threads:

    t.join()

end = time.time()

print(f"All downloads done in {end - start:.2f} seconds")

**Important:** network requests are I/O-bound — threads help. Requires requests package.

**6) thread1.py — corrected breakfast example (had a few bugs)**

Fixed time.sleep() and time.time() usages.

# thread1.py

import threading

import time

def boil\_milk():

    print("Boiling Milk ...")

    time.sleep(2)  # simulate boiling

    print("Milk Boiled ...")

def toast\_bun():

    print("Toasting Bun...")

    time.sleep(3)

    print("Done with bun toast")

start = time.time()

t1 = threading.Thread(target=boil\_milk)

t2 = threading.Thread(target=toast\_bun)

t1.start()

t2.start()

t1.join()

t2.join()

end = time.time()

print(f"Breakfast is ready in {end - start:.2f} seconds")

**Note:** time.sleep() must get a numeric argument; also time.time() needs parentheses.

**7) thread2.py — two types of chai with different durations**

Simple demonstration of running two waits concurrently.

# thread2.py

import threading

import time

def prepare\_chai(type\_, wait\_time):

    print(f"{type\_} chai is brewing...")

    time.sleep(wait\_time)

    print(f"{type\_} chai: Ready ...")

t1 = threading.Thread(target=prepare\_chai, args=("Masala", 2))

t2 = threading.Thread(target=prepare\_chai, args=("Ginger", 3))

t1.start()

t2.start()

t1.join()

t2.join()

**8) threadlock.py — using Lock to avoid race conditions**

Illustrates why you need a lock when multiple threads update shared state.

# threadlock.py

import threading

counter = 0

lock = threading.Lock()

def increment():

    global counter

    for \_ in range(100\_000):

        with lock:

            # critical section

            counter += 1

threads = [threading.Thread(target=increment) for \_ in range(10)]

[t.start() for t in threads]

[t.join() for t in threads]

print(f"Final counter: {counter}")

**Without lock** the final counter will often be less than expected due to lost updates (race conditions).

**9) cpu\_threads\_demo.py — threads running CPU heavy task (contrasts with multiprocessing)**

# cpu\_threads\_demo.py

import threading

import time

def cpu\_heavy():

    print("Crunching some numbers ...")

    total = 0

    for i in range(10\*\*7):

        total += i

    print("DONE ✅")

start = time.time()

threads = [threading.Thread(target=cpu\_heavy) for \_ in range(2)]

[t.start() for t in threads]

[t.join() for t in threads]

print(f"Time taken (threads): {time.time() - start:.2f} seconds")

**Note:** With CPython this will usually not be faster than sequential because of the GIL.

**10) process\_Queue.py — sending a message from child process to parent**

Demonstrates multiprocessing.Queue.

# process\_Queue.py

from multiprocessing import Process, Queue, Value

def prepare\_chai(queue):

    queue.put("Masala chai is ready")

if \_\_name\_\_ == "\_\_main\_\_":

    queue = Queue()

    p = Process(target=prepare\_chai, args=(queue,))

    p.start()

    p.join()

    print(queue.get())

**11) process\_two.py — multiprocessing CPU-bound demo**

# process\_two.py

from multiprocessing import Process

import time

def cpu\_heavy():

    print("Crunching some numbers ...")

    total = 0

    for i in range(10\*\*7):

        total += i

    print("DONE ✅")

if \_\_name\_\_ == "\_\_main\_\_":

    start = time.time()

    processes = [Process(target=cpu\_heavy) for \_ in range(2)]

    [p.start() for p in processes]

    [p.join() for p in processes]

    print(f"Time taken (processes): {time.time() - start:.2f} seconds")

**What to expect:** multiprocessing will usually be faster than threads here (assuming multiple CPU cores).

**12) process\_value.py — shared integer with lock**

Use Value to share a numeric value safely across processes.

# process\_value.py

from multiprocessing import Process, Value

def increment(counter):

    for \_ in range(100\_000):

        with counter.get\_lock():

            counter.value += 1

if \_\_name\_\_ == "\_\_main\_\_":

    counter = Value('i', 0)

    processes = [Process(target=increment, args=(counter,)) for \_ in range(4)]

    [p.start() for p in processes]

    [p.join() for p in processes]

    print("Final counter value:", counter.value)

**Note:** .get\_lock() ensures increments are atomic across processes.

**Common bugs & gotchas (you had examples of some of these)**

* Missing if \_\_name\_\_ == "\_\_main\_\_": guard for code that starts processes — on Windows and some environments this is *required* to avoid infinite recursion.
* time.sleep() must be passed an argument.
* time.time vs time.time() — include parentheses to call.
* Accessing thread name: threading.current\_thread().name (not a call).
* Remember requests blocking calls release the GIL, so threads improve throughput.

**Short revision checklist (one-page)**

* Concurrency ≠ Parallelism: concurrency = structure, parallelism = actual simultaneous execution.
* Threads:
  + Share memory, cheap to create.
  + Good for I/O-bound tasks.
  + Must use locks (Lock, RLock) to protect shared state.
  + Use queue.Queue() to pass data safely between threads.
* GIL:
  + CPython has a GIL → only one thread executes Python bytecode at a time.
  + GIL prevents threads from speeding up CPU-bound pure-Python code.
* Multiprocessing:
  + Each process has its own GIL → real parallelism.
  + Higher memory and startup cost.
  + Use multiprocessing.Queue, Value, Array, Manager for IPC/sharing.
  + Always guard with if \_\_name\_\_ == "\_\_main\_\_": on Windows.
* Locks & synchronization:
  + Use with lock: for clarity and safety.
  + Use Event, Condition, Semaphore for more complex coordination.
* When to use what:
  + I/O-bound: threads or async.
  + CPU-bound: multiprocessing or C extensions / native libraries.

**Quick examples to remember**

* I/O-bound → threading or asyncio (easier: threads).
* CPU-bound → multiprocessing.
* Shared mutable state → protect with locks or use message passing (queues).