Day 50

AsyncIO in Python:

What Is asyncio?

asyncio is Python's library for asynchronous programming.

It lets your program:

- Do many things at once
- Without threads
- By waiting efficiently (non-blocking)

Best for I/O-bound tasks (network, disk, timers), not CPU-heavy work.

The Problem asyncio Solves: Synchronous (Blocking) Code. Asynchronous Idea: While one task is *waiting*, another task can *run*.

Key Async Concepts:

Term	Meaning
Event Loop	The scheduler that runs async tasks
Coroutine	A function that can pause & resume
await	Pause execution until result is ready
Task	A scheduled coroutine
Non-blocking	Doesn't stop the whole program

Example:

```
1  import asyncio
2  async def say_hello():
3      print("Hello")
4      await asyncio.sleep(1)
5      print("World")
6  asyncio.run(say_hello())
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORT

PS E:\Python\Day41-50\Day50> python .\main.py
Hello
World
```

What Makes a Function Async?

async def my_function():
...

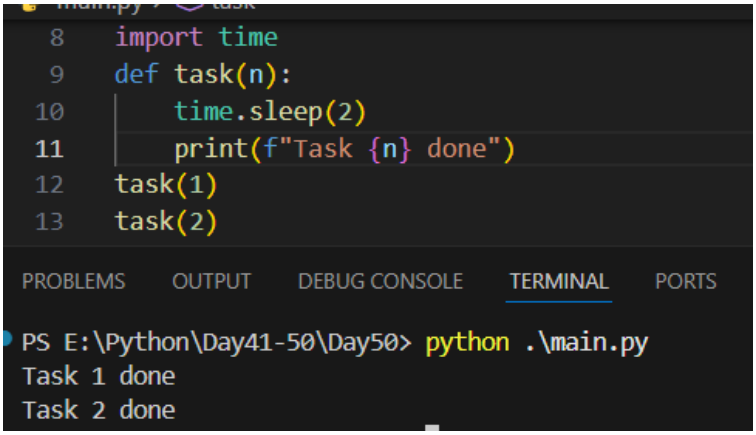
- async def → coroutine function
- Calling it does not run it immediately

await – The Heart of AsyncIO

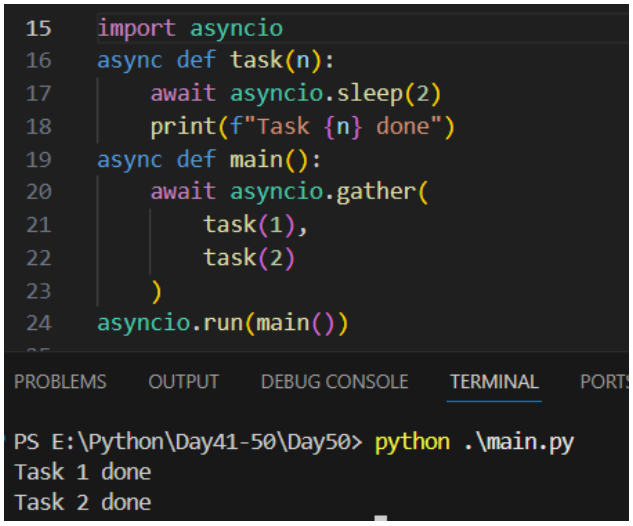
await asyncio.sleep(2)

This means: “Pause this task, let others run, and resume later.”

Example: without asyncio



Example: with asyncio



Async vs Threads vs Processes

Type	Best For
asyncio	I/O-bound tasks
Threads	Blocking I/O
Processes	CPU-bound tasks

Example:

```python
import asyncio
async def fetch_data(n):
    print(f"Fetching {n}")
    await asyncio.sleep(2)
    return f"Result {n}"
async def main():
    results = await asyncio.gather(
        fetch_data(1),
        fetch_data(2),
        fetch_data(3)
    )
    print(results)
asyncio.run(main())
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS E:\Python\Day41-50\Day50> python .\main.py
Fetching 1
Fetching 2
Fetching 3
['Result 1', 'Result 2', 'Result 3']
```

Summary

- asyncio enables efficient concurrency
- Uses coroutines (async / await)
- Perfect for I/O-bound tasks
- Event loop handles scheduling

Multithreading in Python:

What Is Multithreading?

Multithreading allows a program to run multiple threads (tasks) at the same time.

- A thread is the smallest unit of a program that can execute independently.
- Threads share the same memory space.
- Useful for I/O-bound tasks, like downloading files or waiting for network responses.

Why Use Multithreading?

Python threads can help you:

- Perform multiple tasks without waiting for each one to finish
- Improve performance for I/O-bound operations
- Keep a program responsive (e.g., GUI apps)

Note: For CPU-bound tasks, Python threads are limited by the GIL (Global Interpreter Lock). Use multiprocessing instead.

The threading Module

Python provides the threading module to create and manage threads:

Import threading

Example:

```
42  import threading
43  import time
44  def print_numbers():
45      for i in range(1, 6):
46          print(i)
47          time.sleep(1)
48  # Create thread
49  t = threading.Thread(target=print_numbers)
50  # Start thread
51  t.start()
52  # Wait for thread to finish
53  t.join()
54  print("Thread finished")

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS E:\Python\Day41-50\Day50> python .\main.py
1
2
3
4
5
Thread finished
```

Key Methods in threading.Thread

Method	Purpose
start()	Starts the thread
run()	Thread's code (called by start())
join()	Waits for thread to finish
is_alive()	Checks if thread is running

Example: running multiple threads.

```python
import threading
import time
def task(n):
    print(f"Task {n} starting")
    time.sleep(2)
    print(f"Task {n} finished")
threads = []
for i in range(3):
    t = threading.Thread(target=task, args=(i,))
    threads.append(t)
    t.start()
for t in threads:
    t.join()
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS E:\Python\Day41-50\Day50> python .\main.py
Task 0 starting
Task 1 starting
Task 2 starting
Task 0 finished
Task 1 finished
Task 2 finished
```

Thread vs Process

Feature	Thread	Process
Memory	Shared	Separate
GIL	Yes (limits CPU-bound)	No
I/O-bound	Good	Overkill
CPU-bound	Limited	Best

Summary

- threading enables concurrent execution of I/O tasks
- Use Thread, ThreadPoolExecutor, Lock, daemon wisely
- Not ideal for CPU-heavy work (use multiprocessing)
- Essential for responsive or I/O-heavy applications

Multiprocessing in Python:

What Is Multiprocessing?

Multiprocessing allows Python to run multiple processes at the same time, each with its own memory space.

- Each process is independent.
- Bypasses Python's GIL (Global Interpreter Lock).
- Ideal for CPU-bound tasks like heavy calculations, data processing, or simulations.

The multiprocessing Module

Python provides the multiprocessing module to create and manage processes:

Import multiprocessing

Example:

```
70  import multiprocessing
71  def worker(name):
72      print(f"Worker {name} is running")
73  if __name__ == "__main__":
74      p = multiprocessing.Process(target=worker, args=("Alice",))
75      p.start()  # start the process
76      p.join()   # wait for it to finish
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS E:\Python\Day41-50\Day50> python .\main.py
Worker Alice is running
```

Note: Always protect the entry point with if __name__ == "__main__": when using multiprocessing on Windows.

Example: running multiple processes.

```python
import multiprocessing
import time
def task(n):
    print(f"Task {n} started")
    time.sleep(2)
    print(f"Task {n} finished")
if __name__ == "__main__":
    processes = []
    for i in range(3):
        p = multiprocessing.Process(target=task, args=(i,))
        processes.append(p)
        p.start()
    for p in processes:
        p.join()
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS E:\Python\Day41-50\Day50> python .\main.py
Task 0 started
Task 1 started
Task 2 started
Task 0 finished
Task 1 finished
Task 2 finished
```

Process vs Thread

Feature	Thread	Process
Memory	Shared	Separate
GIL	Yes	No (true parallelism)
CPU-bound	Limited	Ideal
I/O-bound	Good	Works but overhead

Sharing Data Between Processes

Processes do not share memory by default. You need:

- multiprocessing.Queue
- multiprocessing.Pipe
- multiprocessing.Manager

Example: using queue.

```python
import multiprocessing
def worker(q):
    q.put("Hello from worker")
if __name__ == "__main__":
    q = multiprocessing.Queue()
    p = multiprocessing.Process(target=worker, args=(q,))
    p.start()
    print(q.get())  # receive data from worker
    p.join()
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS E:\Python\Day41-50\Day50> python .\main.py
Hello from worker
```

Example: using manager director.

```python
from multiprocessing import Process, Manager
def worker(d):
    d["name"] = "Alice"
if __name__ == "__main__":
    with Manager() as manager:
        shared_dict = manager.dict()
        p = Process(target=worker, args=(shared_dict,))
        p.start()
        p.join()
        print(shared_dict)  # {'name': 'Alice'}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS E:\Python\Day41-50\Day50> python .\main.py
{'name': 'Alice'}
```

Summary

- Use multiprocessing for CPU-bound tasks
- Use Process, Queue, Pool, or Manager
- Threads are still better for I/O-bound tasks
- Always protect main entry point with if __name__ == "__main__"

--The End--