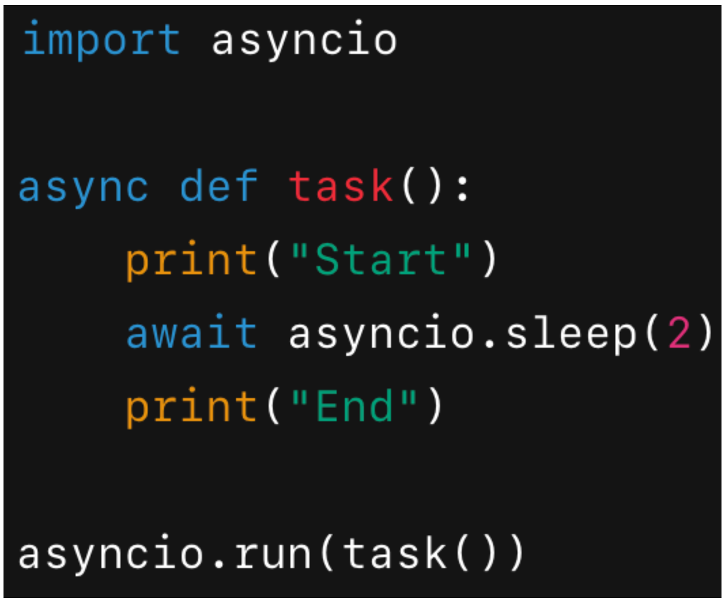
🔷 The Code You Gave

```
import asyncio

async def task():
    print("Start")
    await asyncio.sleep(2)
    print("End")

asyncio.run(task())
```



```
import asyncio

async def task():
    print("Start")
    await asyncio.sleep(2)
    print("End")

asyncio.run(task())
```

We will understand this in **3 levels**:

1. What it *looks like*
2. What it *really does internally*
3. How it is different from normal Python execution

✅ LEVEL 1 — Simple Explanation (What it appears to do)

If you run this, output will be:

```
Start
(wait 2 seconds)
End
```

So at a surface level:

- It prints `"Start"`
- Waits 2 seconds
- Then prints `"End"`

But this is **NOT just a normal sleep — that’s the key difference.**

🔷 Step-by-Step Line-by-Line Explanation

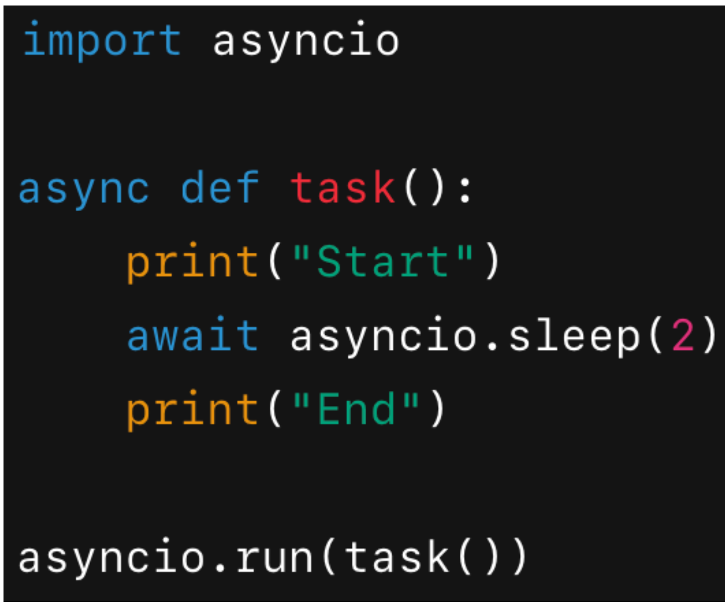
Line 1:

```
import asyncio
```

You are importing Python's **asynchronous programming library**.

This module provides:

- event loop
- async tasks
- coroutines
- await handling



```
import asyncio

async def task():
    print("Start")
    await asyncio.sleep(2)
    print("End")

asyncio.run(task())
```

Think of `asyncio` as:

A traffic controller for multiple tasks in Python.

Line 3:

```
async def task():
```

This is **NOT a normal function**.

This is called a **COROUTINE FUNCTION**.

Calling this function **does NOT run it immediately**.

Let's prove that:

Try this:

```
result = task()
print(result)
```

Output will be something like:

```
<coroutine object task at 0x…>
```

👉 Meaning:
You created a *coroutine object*, not a running function.

Think of it like:

- Normal function → runs when called
- Async function → creates a task that can be scheduled

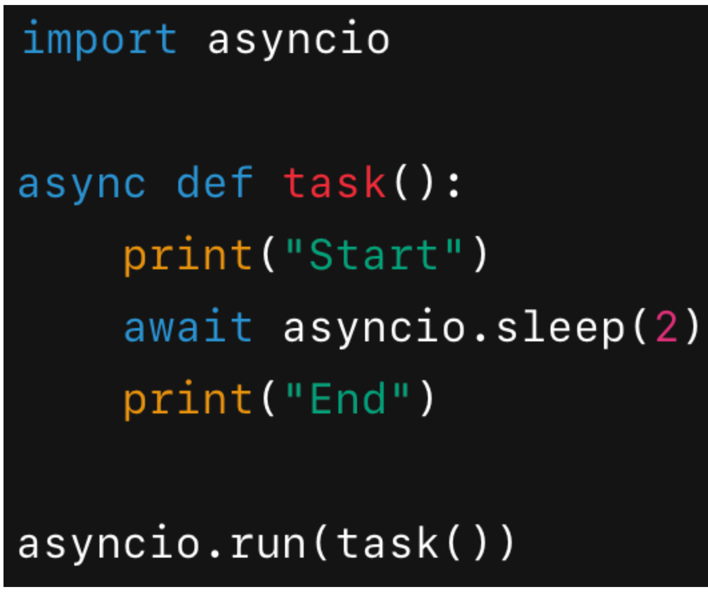
Inside the function

Line 4:

```
print(“Start")
```

This runs **immediately when the coroutine starts executing.**

This is a normal blocking print statement.



```
import asyncio

async def task():
    print("Start")
    await asyncio.sleep(2
    print("End")

asyncio.run(task())
```

Line 5 (MOST IMPORTANT LINE):

```
await asyncio.sleep(2)
```

This is the **core of async programming.**

What does `asyncio.sleep(2)` mean?

It means:

“Pause this task for 2 seconds, but let other tasks run meanwhile.”

This is **NON-BLOCKING sleep**, unlike normal:

```
time.sleep(2)   # blocks entire program
```

With `await asyncio.sleep(2):`

- Python says:
  - “Okay, I’ll pause *this task*.”
  - “I’ll come back to it after 2 seconds.”
  - “Meanwhile, I can run other tasks.”

So `await` means:

Suspend this coroutine and give control back to the event loop.

Line 6:

```
print("End")
```

This runs **after 2 seconds**, when the event loop resumes this task.

Last Line:

```
asyncio.run(task())
```

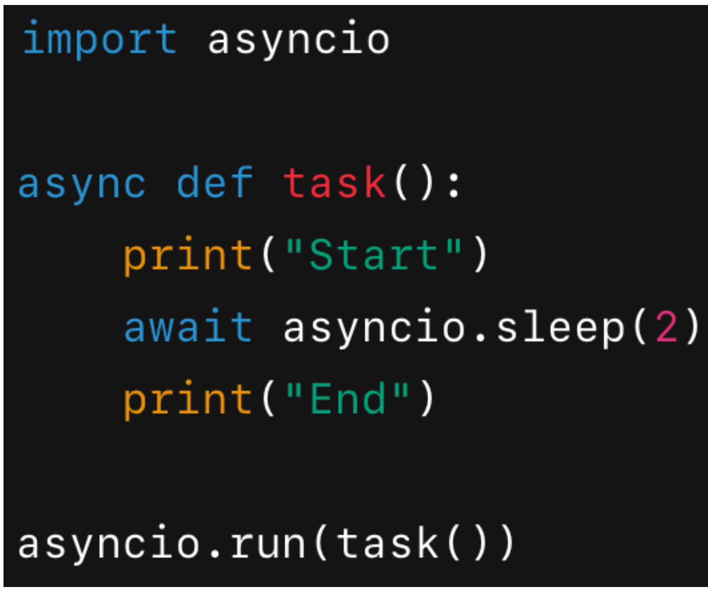
This is very important.

This does 3 things:

1. Creates an **event loop**
2. Puts your `task()` coroutine inside it
3. Runs the loop until the task finishes

You can think of it like:

```
start_event_loop()
run(task)
stop_event_loop()
```



```
import asyncio

async def task():
    print("Start")
    await asyncio.sleep(2)
    print("End")

asyncio.run(task())
```

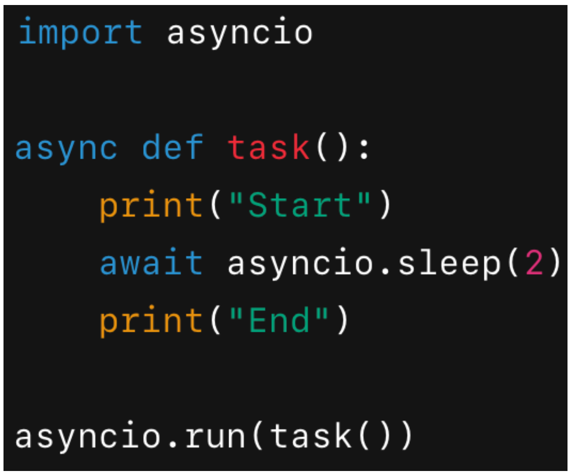
🔷 LEVEL 2 — What is actually happening internally?

Let's visualize execution timeline.

Timeline:

Time	What Python is doing
t = 0	Event loop starts
t = 0	task() begins
t = 0	Prints "Start"
t = 0	Hits `await asyncio.sleep(2)`
t = 0	Task is PAUSED
t = 0 → 2	Event loop is FREE
t = 2	Task is RESUMED
t = 2	Prints "End"
t = 2	Task finishes
t = 2	Event loop stops

So this is **single-threaded but concurrent-style execution.**

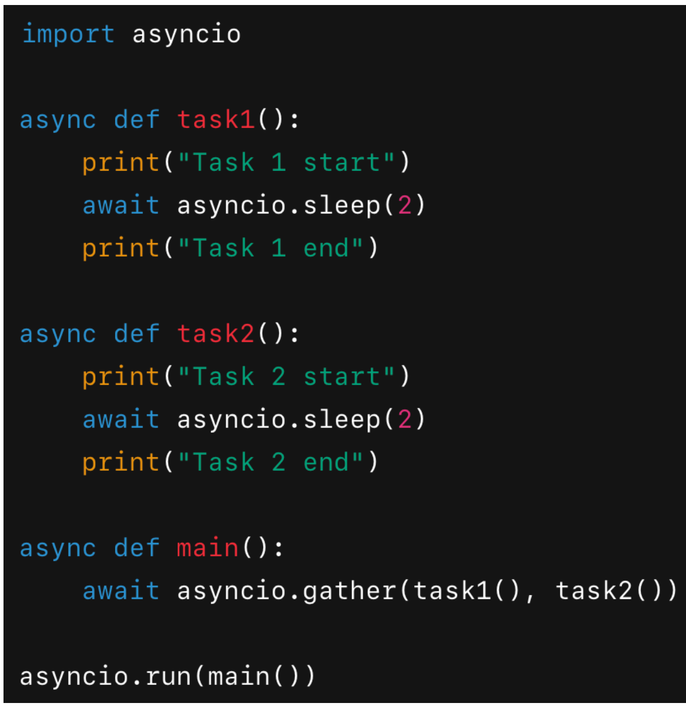


```python
import asyncio

async def task():
    print("Start")
    await asyncio.sleep(2)
    print("End")

asyncio.run(task())
```

🔷 LEVEL 3 — Why is this useful?

Right now you have only **one task**, so it looks useless.

Let’s see power of async by adding another task.



```python
import asyncio

async def task1():
    print("Task 1 start")
    await asyncio.sleep(2)
    print("Task 1 end")

async def task2():
    print("Task 2 start")
    await asyncio.sleep(2)
    print("Task 2 end")

async def main():
    await asyncio.gather(task1(), task2())

asyncio.run(main())
```

Example with TWO tasks

```python
import asyncio

async def task1():
    print("Task 1 start")
    await asyncio.sleep(2)
    print("Task 1 end")

async def task2():
    print("Task 2 start")
    await asyncio.sleep(2)
    print("Task 2 end")

async def main():
    await asyncio.gather(task1(), task2())

asyncio.run(main())
```

Output will be like:

```
Task 1 start
Task 2 start
(wait 2 seconds)
Task 1 end
Task 2 end
```

Instead of taking 4 seconds, it takes **only 2 seconds total**.

That is the real benefit of async.

🔷 Key Core Concepts You Should Remember

✅ 1. `async def` creates a coroutine (not a normal function)

✅ 2. `await` pauses execution without blocking Python

✅ 3. `asyncio.sleep()` is non-blocking

✅ 4. `asyncio.run()` starts and manages the event loop

✅ 5. Async is best for:

- API calls
- Database calls
- Web scraping
- File I/O
- Network requests

Not best for:

- Heavy calculations (use multiprocessing instead)

🔷 One-Line Core Meaning of Your Code

“Run a task that prints ‘Start’, pauses asynchronously for 2 seconds without blocking Python, then prints ‘End’, all managed by an event loop.”