```python
import asyncio

async def task(name, seconds):
    print(f"{name}: Started")
    await asyncio.sleep(seconds)
    print(f"{name}: Finished after {seconds} sec")

async def main():
    await asyncio.gather(
        task("Task A", 2),
        task("Task B", 3),
        task("Task C", 1)
    )

asyncio.run(main())


# ========================================

import time

def task(name, seconds):
    print(f"{name}: Started")
    time.sleep(seconds)
    print(f"{name}: Finished after {seconds} sec")

def main():
    task("Task A", 2)
    task("Task B", 3)
    task("Task C", 1)

main()
```

Asynchronous Programming in Python

Why Asynchronous Programming?

In traditional (synchronous) Python code, tasks are executed **one after the other**. If one task takes time (like waiting for a file to load or a server to respond), the entire program is **blocked** until it completes.

Asynchronous programming allows multiple tasks to run concurrently by "pausing" and "resuming" based on availability of resources (typically I/O). This leads to efficient usage of time and resources, especially when dealing with I/O-bound operations.

Core Concepts

1. Coroutines

A **coroutine** is a special function that can **pause** and **resume** its execution.

Defined using:

```
async def my_function():
    ...
```

Coroutines don’t run until they are awaited or scheduled in an event loop.

Example:

```
async def greet():
    print("Hello")
    await asyncio.sleep(1)
    print("World")
```

2. `await`

`await` is used **inside an** `async def` **function** to **pause** that function until the awaited task completes. This only works with **awaitable** objects (usually other coroutines or `asyncio` methods).

Example:

```
await asyncio.sleep(2)  # Pauses this coroutine for 2 seconds without blocking others
```

When a coroutine `await`s something:

- It yields control back to the event loop.
- The event loop continues executing other ready tasks.
- After the awaited operation finishes, control returns to the paused coroutine.

Key Insight: Python's Async Model

Python's `asyncio` is **single-threaded** and based on an **event loop**.

- It runs coroutines concurrently using **cooperative multitasking**.
- Each coroutine **voluntarily gives up control** by using `await`.

This is not parallelism (like with threads), but **concurrent scheduling** of tasks.

When Should You Use `await`?

Use `await` when you're dealing with **I/O-bound** tasks:

- Network requests (APIs)
- File I/O (if using async-friendly libraries)
- Database queries
- Timers or delays

Avoid using `await` for **CPU-bound** tasks (e.g., heavy computations) — these should use threads or processes.

Task	Use `await`?	Reason
API Call	Yes	Waiting for server response
Async File Read	Yes	Disk I/O
`asyncio.sleep()`	Yes	Timed pause (non-blocking)
Heavy Computation	No	Blocks event loop

`asyncio.create_task()` vs Sequential `await`

Purpose of `create_task()`

`asyncio.create_task()` schedules a coroutine to **run in the background**, allowing the current function to continue without waiting immediately. It returns a Task object.

Used for **concurrency**:

```python
async def main():
    task = asyncio.create_task(some_coroutine())
    # continue doing other things
    await task
```

Example 1: `create_task()` with Await

```python
import asyncio

async def greet(name):
    print(f"Starting greeting for {name}")
    await asyncio.sleep(2)
    print(f"Hello, {name}!")

async def main():
    task1 = asyncio.create_task(greet("Alice"))
    task2 = asyncio.create_task(greet("Bob"))
    print("Both tasks started")
    await task1
    await task2

asyncio.run(main())
```

Output:

```
Both tasks started
Starting greeting for Alice
Starting greeting for Bob
Hello, Alice!
Hello, Bob!
```

Tasks Without `await` Inside: What's the Point?

If the coroutine you're calling doesn't include any `await`, using `create_task()` or `await` is pointless — because the function just runs straight through.

Example:

```python
async def greet(name):
    print(f"Hello {name}")
```

This is basically synchronous behavior wrapped in `async`. No concurrency benefit.

Running Tasks in a Specific Order — Even If They Take Different Times

If you want tasks to complete **in the order you define**, you have 3 main approaches:

1. Sequential Execution (Strict Order)

```python
async def task(name, delay):
    print(f"Starting {name}")
    await asyncio.sleep(delay)
    print(f"Finished {name}")

async def main():
    await task("Task 1", 3)
    await task("Task 2", 1)
    await task("Task 3", 2)

asyncio.run(main())


# OUTPUT
Starting Task 1
Finished Task 1
Starting Task 2
Finished Task 2
Starting Task 3
Finished Task 3
```

Order is guaranteed, but tasks run one-by-one.

---

2. Concurrent Start, Ordered Wait (`create_task` + ordered `await`)

```python
async def main():
    task1 = asyncio.create_task(task("Task 1", 3))
    task2 = asyncio.create_task(task("Task 2", 1))
    task3 = asyncio.create_task(task("Task 3", 2))

    await task1
    await task2
    await task3
```

All tasks start **immediately**, but we **wait for them in order**. This is useful if you want **efficiency**, but also want to wait in sequence.

---

3. Fully Concurrent + Collecting Results in Order

```
async def get_result(name, delay):
    await asyncio.sleep(delay)
    return f"{name} done"

async def main():
    task1 = asyncio.create_task(get_result("Task 1", 3))
    task2 = asyncio.create_task(get_result("Task 2", 1))
    task3 = asyncio.create_task(get_result("Task 3", 2))

    results = []
    for task in [task1, task2, task3]:
        result = await task
        results.append(result)

    print(results)
```

This ensures **start is parallel**, but **awaiting is controlled**.

Key Insight:

- If you need order, `await` in order.
- If you need concurrency, use `create_task()` or `gather()`.
- If you need both: `create_task()` and await in a controlled sequence.

Real-World Advice

- For **web servers**, always use `create_task()` (or `gather()`) when firing background tasks.
- For **scripts or workflows**, prefer sequential awaits when order matters.
- Use `asyncio.wait()` or `asyncio.gather()` when dealing with multiple APIs or DB calls that can happen concurrently.

Visual Timeline Example

```
async def task_a():
    print("Task A started")
    await asyncio.sleep(3)
    print("Task A finished")

async def task_b():
```

```python
    print("Task B started")
    await asyncio.sleep(2)
    print("Task B finished")

await asyncio.gather(task_a(), task_b())
```

Timeline:

```
0s - Task A started
0s - Task B started
2s - Task B finished
3s - Task A finished
```

Tasks ran **concurrently**, not in parallel, because they were **paused and resumed** smartly by the event loop.

Behind the Scenes: What Happens at `await`

When a coroutine hits an `await`, the following happens:

1. The coroutine **pauses**.
2. Control returns to the **event loop**.
3. The event loop finds another **ready coroutine** and runs it.
4. When the awaited task completes, the original coroutine **resumes from where it left off**.

No threads. No locks. No blocking.

Comparison Table

Feature	Synchronous	Async (`asyncio`)
Blocking	Yes	No (if `await` used correctly)
Concurrency Type	None	Cooperative multitasking
Parallelism	No	No (but efficient concurrency)
Best For	CPU-bound	I/O-bound
Uses `await`	No	Yes

Common Asyncio Tools

- `asyncio.run()` — runs the event loop for you
- `asyncio.gather()` — run multiple coroutines concurrently
- `asyncio.sleep()` — async version of `time.sleep()` (non-blocking)

Real World Example with FastAPI

FastAPI is a modern, async-first web framework in Python. It's built on top of `Starlette` and `Pydantic`, and makes it easy to write asynchronous APIs.

Basic Async FastAPI Route

```python
from fastapi import FastAPI
import asyncio

app = FastAPI()

@app.get("/hello")
async def say_hello():
    await asyncio.sleep(1)  # Simulating a slow I/O operation
    return {"message": "Hello, world!"}
```

- This endpoint will **not block** other requests.
- While sleeping, FastAPI can handle more incoming requests.

Simulating Multiple Concurrent Tasks

```python
@app.get("/multiple")
async def handle_multiple():
    results = await asyncio.gather(
        asyncio.sleep(2),
        asyncio.sleep(3)
    )
    return {"status": "done"}
```

Async DB Query (example with `databases` lib)

```python
from databases import Database

database = Database("sqlite:///./test.db")

@app.on_event("startup")
async def startup():
    await database.connect()

@app.on_event("shutdown")
async def shutdown():
    await database.disconnect()

@app.get("/users")
async def get_users():
    query = "SELECT * FROM users"
    return await database.fetch_all(query)
```

This shows how FastAPI uses `await` to interact with async DB libraries, making I/O operations **non-blocking**.

Final Thoughts

04 Asynchronous Programming in Python

> Asynchronous programming in Python is a powerful tool to write **efficient, non-blocking, scalable** applications.

It shines especially in:

- Web apps (e.g., FastAPI)
- Bots (e.g., Discord bots)
- Networked systems
- File/network heavy utilities
  Use it when you want to **do more while waiting** — without resorting to threads.

```
import requests
import time

def fetch_data_sequential(urls):
    results = []
    start_time = time.time()

    for url in urls:
        response = requests.get(url)
        if response.status_code == 200:
            results.append(response.json())

    elapsed = time.time() - start_time
    print(f"Sequential requests completed in {elapsed:.2f} seconds")
    return results

# Example usage with sequential requests
urls = [
    "https://jsonplaceholder.typicode.com/posts/1",
    "https://jsonplaceholder.typicode.com/posts/2",
    "https://jsonplaceholder.typicode.com/posts/3",
    "https://jsonplaceholder.typicode.com/comments?postId=1",
    "https://jsonplaceholder.typicode.com/albums/1"
]

results = fetch_data_sequential(urls)
print(f"Received {len(results)} responses")

# ============================= Now with asyncio ===================================

import asyncio
import aiohttp

async def fetch_url(session, url):
    async with session.get(url) as response:
        if response.status == 200:
            return await response.json()
        return None

async def fetch_data_async(urls):
    async with aiohttp.ClientSession() as session:
        tasks = [fetch_url(session, url) for url in urls]
        return await asyncio.gather(*tasks)
```

```python
async def main():
    start_time = time.time()

    results = await fetch_data_async(urls)

    elapsed = time.time() - start_time
    print(f"Async requests completed in {elapsed:.2f} seconds")
    print(f"Received {len([r for r in results if r is not None])} responses")

if __name__ == "__main__":
    print("\nRunning sequential version first:")
    fetch_data_sequential(urls)

    print("\nRunning async version now:")
    asyncio.run(main())
```