Introduction to Python Async Tasks

What is Asynchronous Programming?

Asynchronous programming allows you to write code that can perform other tasks while waiting for external operations such as network requests or file I/O. In contrast to synchronous (blocking) code, async code uses an event loop to schedule and run tasks concurrently.

Key Terms

Coroutine: A special function defined with async def that can pause and resume its execution.

Event Loop: The core of async programming that manages and schedules coroutines and callbacks.

Awaitable: An object that can be used with the await expression, usually coroutines or Futures.

Task: A wrapper for a coroutine that schedules its execution on the event loop.

Basic Async Example

```
import asyncio

async def say_after(delay, message):
    """Pauses for 'delay' seconds then prints 'message'."""
    await asyncio.sleep(delay)
    print(message)

async def main():
    # Schedule two tasks concurrently
    task1 = asyncio.create_task(say_after(1, "Hello"))
    task2 = asyncio.create_task(say_after(2, "World"))

    print("Tasks started")
    # Wait for both tasks to complete
    await task1
    await task2
    print("Tasks completed")

# Run the main coroutine
asyncio.run(main())
```

Explanation of the Example

1. async def defines a coroutine. 2. await suspends execution until the awaitable is done. 3. asyncio.create_task() schedules coroutines to run concurrently. 4. asyncio.run() starts the event loop and runs the given coroutine.

When to Use Async

Use async tasks when dealing with I/O-bound or high-level structured network code. It can improve performance by handling many tasks concurrently without threading overhead.