Python: Asynchronous Workflows with AsyncIO and Aiohttp

Why Asynchronous Programming?

Dan Tofan
PhD

@dan_tofan | https://programmingwithdan.com

Course Introduction

Why asynchronous programming?

Implementing asynchronous HTTP with Aiohttp

Designing efficient workflows with AsyncIO

Integrating AsyncIO and Aiohttp for scalable systems

Globomantics Storyline

Python ecosystem

Performance issues

Many API calls and database queries

Code Execution Models

Synchronous:

- Single thread for sequential operations
- Easy to understand
- Inefficient for I/O-bound operations

Multithreading:

- Run parallel threads
- Risks of synchronization issues
- Efficient for CPU-bound tasks (from 3.13)

Asynchronous:

- Single thread for multiple operations
- Can help CPU-bound operations (from 3.13)
- Efficient for I/O-bound operations

Asynchronous Programming Increases Performance

Multiple concurrent operations

Remove unneeded waiting time

Maximize CPU usage

Best for I/O-bound operations

AsyncIO Building Blocks

Event loop

Coroutines

Tasks

Event Loop

Central coordinator

- Execution flow
- Task management
- Handles I/O events
- Ensures non-blocking execution

Coroutines

Special functions

Syntax: 'async def' instead of 'def'

Execution

- By awaiting
- By wrapping it in a task

Tasks

Wrappers for coroutines

State management

- Tracking state of tasks
- Cancel tasks

Propagate exceptions

```python
import asyncio


async def call_api():
  print("calling")


async def main():
  await call_api()

  task = asyncio.create_task(call_api())
  await task


asyncio.run(main())
```

◀ **Import the asyncio module**

◀ **Coroutine to simulate an API call**

◀ **Main coroutine**

◀ **Execute directly**

◀ **Create new task**

◀ **Wait for task to complete**

◀ **Create new event loop and run the coroutine**

Demo: Getting Started with AsyncIO

Key Demo Points

Awaiting coroutines directly

– No performance gains

Tasks

– Performance gains from concurrency

Blocking operations deny concurrency

– Use async operations