Python

Python Concurrency

Part 2: Asyncio and HTTP Operations

April 3, 2025

Source Code

PYTHON | CONCURRENCY (PART 2)

1. Asyncio for Modern Concurrency

1.1. Coroutines and the Event Loop

Asyncio provides a different concurrency model for I/O-bound tasks:

```python
import asyncio
import time

async def say_after(delay, message):
    """Coroutine that waits and then prints a message"""
    await asyncio.sleep(delay)
    print(message)

async def main():
    # Sequential execution
    print("Sequential execution:")
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

```
12      start = time.time()
13
14      await say_after(1, "Hello")
15      await say_after(2, "World")
16
17      print(f"Sequential took {time.time() - start:.2f}
    seconds\n")
18
19      # Concurrent execution
20      print("Concurrent execution:")
21      start = time.time()
22
23      # Create tasks to run concurrently
24      hello_task = asyncio.create_task(say_after(1, "Hello"))
25      world_task = asyncio.create_task(say_after(2, "World"))
26
27      # Wait for both tasks to complete
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

```
    await hello_task
    await world_task

    print(f"Concurrent took {time.time() - start:.2f} seconds")

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

# Output
# Sequential execution:
# Hello
# World
# Sequential took 3.00 seconds

# Concurrent execution:
# Hello
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

```
# World
# Concurrent took 2.00 seconds
```

This example demonstrates the basic concepts of asyncio:

- **Coroutines:** Functions defined with `async def` that can be paused and resumed
- **Tasks:** Scheduled coroutines managed by the event loop
- **Await:** The keyword that yields control back to the event loop until an operation completes
- **Event Loop:** The central execution mechanism that coordinates all asyncio operations

The first example shows the difference between sequential execution (where operations block each other) and concurrent execution (where operations can run concurrently while one is waiting).

1.2. Async with HTTP Requests

A practical example of asyncio for concurrent HTTP requests:

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

PYTHON | CONCURRENCY (PART 2)

```python
import asyncio
import aiohttp
import time

async def fetch_url(session, url):
    """Fetch a URL asynchronously"""
    start = time.time()
    async with session.get(url) as response:
        data = await response.text()
        elapsed = time.time() - start
        return {
            'url': url,
            'status': response.status,
            'size': len(data),
            'time': elapsed
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

```
        }

async def fetch_all(urls):
    """Fetch multiple URLs concurrently"""
    async with aiohttp.ClientSession() as session:
        tasks = [fetch_url(session, url) for url in urls]
        results = await asyncio.gather(*tasks)
        return results

async def main():
    # List of URLs to fetch
    urls = [
        'https://www.python.org',
        'https://docs.python.org',
        'https://www.github.com',
        'https://www.google.com',
        'https://www.wikipedia.org',
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

```
    ]

    # Sequential fetching for comparison
    start = time.time()
    async with aiohttp.ClientSession() as session:
        results = []
        for url in urls:
            results.append(await fetch_url(session, url))
    sequential_time = time.time() - start

    print(f"Sequential fetching: {sequential_time:.2f} seconds")

    # Concurrent fetching
    start = time.time()
    results = await fetch_all(urls)
    concurrent_time = time.time() - start
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

```
49
50      print(f"Concurrent fetching: {concurrent_time:.2f}
    seconds")
51      print(f"Speedup:
    {sequential_time/concurrent_time:.2f}x")
52
53      # Display results
54      for result in results:
55          print(f"{result['url']}: {result['status']}
    ({result['size']} bytes, {result['time']:.2f}s)")
56
57 # Run the main coroutine
58 if __name__ == "__main__":
59      asyncio.run(main())
60
61 # Output
62 # Sequential fetching: 2.26 seconds
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

```
# Concurrent fetching: 0.80 seconds
# Speedup: 2.83x
# https://www.python.org: 200 (51012 bytes, 0.12s)
# https://docs.python.org: 200 (17129 bytes, 0.11s)
# https://www.github.com: 200 (281297 bytes, 0.80s)
# https://www.google.com: 200 (16957 bytes, 0.46s)
# https://www.wikipedia.org: 200 (88783 bytes, 0.36s)
```

This example shows how to use asyncio for a common real-world scenario: fetching multiple web pages concurrently. Key points:

- **aiohttp:** An async-compatible HTTP client/server for Python
- **asyncio.gather():** Executes multiple awaitables concurrently and returns their results
- **Resource Management:** Using `async with` to properly manage session resources
- **Performance Comparison:** Direct comparison of sequential vs. concurrent approaches

This pattern is ideal for I/O-bound applications like web scrapers, API clients, and data fetchers.

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

1.3. Producer-Consumer Pattern with Asyncio

A classic concurrency pattern implemented with asyncio:

```python
import asyncio
import random

async def producer_consumer_example():
    # Create a queue with a maximum size
    queue = asyncio.Queue(maxsize=5)

    async def producer():
        """Produces tasks and adds them to the queue"""
        for i in range(10):
            # Create a task
            task = f"Task-{i}"

```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

```
14              # Put the task in the queue (will wait if
    queue is full)
15              await queue.put(task)
16              print(f"Producer: Created {task}")
17
18              # Simulate variable production time
19              await asyncio.sleep(random.uniform(0.1, 0.5))
20
21          # Signal end of production
22          print("Producer: Done producing")
23
24      async def consumer(name):
25          """Consumes tasks from the queue"""
26          while True:
27              try:
28                  # Get a task from the queue (will wait if
    queue is empty)
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

```
29                 task = await queue.get()
30
31                 # Check for sentinel value signaling end
32                 if task is None:
33                     print(f"{name}: Received shutdown
    signal")
34                     queue.task_done()
35                     break
36
37                 print(f"{name}: Processing {task}")
38
39                 # Simulate processing time
40                 await asyncio.sleep(random.uniform(0.2,
    0.7))
41
42                 print(f"{name}: Completed {task}")
43
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

```
                # Signal task completion
                queue.task_done()

            except asyncio.CancelledError:
                break

    # Start producer and consumer tasks
    producer_task = asyncio.create_task(producer())
    consumer_tasks = [
        asyncio.create_task(consumer(f"Consumer-{i}"))
        for i in range(3)
    ]

    # Wait for the producer to finish
    await producer_task

    # Wait for all items to be processed
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

```
    await queue.join()

    # Send shutdown signal to all consumers
    for _ in range(len(consumer_tasks)):
        await queue.put(None)

    # Wait for consumers to process shutdown signal
    await asyncio.gather(*consumer_tasks)

# Run the example
if __name__ == "__main__":
    asyncio.run(producer_consumer_example())
```

The producer-consumer pattern is a fundamental concurrency pattern that asyncio implements with these key components:

- **asyncio.Queue:** A thread-safe queue designed for use in asyncio applications
- **Backpressure:** Managing production rate with a maximum queue size

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

- **Consumer Pool:** Multiple consumers processing items concurrently
- **Graceful Shutdown:** Proper shutdown signals and waiting for all tasks to complete

This pattern is useful for scenarios like job queues, data processing pipelines, and message processing systems.

2. Conclusion (Part 2)

In this second part of our exploration of concurrency and parallelism in Python, we have focused on asyncio, a modern approach to concurrency:

- We learned how asyncio provides single-threaded concurrency through its event loop architecture and coroutines.
- We explored practical applications of asyncio with HTTP requests using the aiohttp library, demonstrating significant performance improvements for I/O-bound operations.
- We implemented the classic producer-consumer pattern using asyncio.Queue, showing how to build efficient and well-managed data processing pipelines.

Asyncio represents a modern approach to concurrency in Python that is particularly well-suited for I/O-bound operations. Its elegant async/await syntax provides a readable and maintainable way to write concurrent code without the complexities of traditional threading approaches.

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

In Part 3, we will explore hybrid approaches that combine different concurrency models, advanced patterns, and performance optimization techniques to further enhance your Python concurrency skills.

3. References (Part 2)

- Python Documentation. *asyncio - Asynchronous I/O*. Link
- Real Python. *Async IO in Python: A Complete Walkthrough*. Link
- PyPI. *aiohttp - Asynchronous HTTP Client/Server*. Link
- Lynn Root. *Asyncio: We Did It Wrong*. Link
- Luciano Ramalho. *Fluent Python*, Chapter 18: Concurrency with asyncio. O'Reilly Media.
- Translated, Edited and written in collaboration with AI.

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

4. Explore My Other Posts

Enjoyed This Content?

Don't miss the first part of this series:

Python Concurrency Part 1: Understanding the GIL, Threading and Multiprocessing

Learn the fundamentals of concurrent programming in Python, how the Global Interpreter Lock works, and when to use threading versus multiprocessing for optimal performance.

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com - www.piagents.dev

Don’t miss Part 3

In the final part of our series, we’ll explore

advanced patterns and hybrid approaches

to take your Python concurrency skills to the next level!