Python

Python Concurrency

Part 3: Advanced Patterns and Hybrid Approaches

April 29, 2025

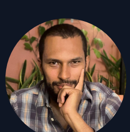
Source Code

PYTHON | CONCURRENCY (PART 3)

1. Combining Approaches for Complex Systems

For real-world applications, you can combine different concurrency models to leverage their respective strengths:

```python
import asyncio
import concurrent.futures
import time

def cpu_bound(number):
    """CPU-bound task (runs in a process)"""
    total = sum(i * i for i in range(number))
    return total

def io_bound(number):
    """I/O-bound task (runs in a thread)"""
    time.sleep(1)  # Simulate I/O
    return number * 2

```

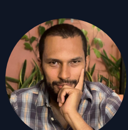
Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

www.asanchezyali.com

PYTHON | **CONCURRENCY (PART 3)**

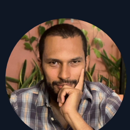
```python
async def main():
    # Create executor pools
    process_pool = concurrent.futures.ProcessPoolExecutor(max_workers=4)
    thread_pool = concurrent.futures.ThreadPoolExecutor(max_workers=10)

    loop = asyncio.get_running_loop()

    # CPU-bound tasks (run in process pool)
    cpu_numbers = [5_000_000, 10_000_000, 15_000_000, 20_000_000]
    cpu_tasks = [
        loop.run_in_executor(process_pool, cpu_bound, number)
        for number in cpu_numbers
    ]

    # I/O-bound tasks (run in thread pool)
    io_numbers = list(range(1, 11))
    io_tasks = [
        loop.run_in_executor(thread_pool, io_bound, number)
```



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

PYTHON | **CONCURRENCY (PART 3)**

```python
        for number in io_numbers
    ]

    # Async I/O-bound tasks (native asyncio)
    async_tasks = [asyncio.sleep(1, result=f"async_{i}") for i in range(5)]

    # Gather all results
    print("Running all tasks concurrently...")
    start = time.time()

    cpu_results = await asyncio.gather(*cpu_tasks)
    io_results = await asyncio.gather(*io_tasks)
    async_results = await asyncio.gather(*async_tasks)

    end = time.time()

    # Show results
    print(f"\nTotal time: {end - start:.2f} seconds")
    print(f"CPU results: {len(cpu_results)} tasks completed")
```



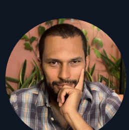
Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```python
    print(f"I/O results: {len(io_results)} tasks completed")
    print(f"Async results: {len(async_results)} tasks completed")

    # Clean up
    process_pool.shutdown()
    thread_pool.shutdown()

if __name__ == "__main__":
    asyncio.run(main())
```

This hybrid approach demonstrates how to:

- **Use ProcessPoolExecutor** for CPU-bound tasks to bypass the GIL
- **Use ThreadPoolExecutor** for blocking I/O operations
- **Use native asyncio** for non-blocking I/O operations
- **Coordinate all approaches** through asyncio's event loop

The example shows how `run_in_executor` allows integration of traditional concurrency approaches with asyncio, creating a unified system that handles different types of workloads optimally.



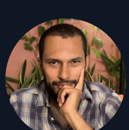
Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

2. Advanced Concurrency Patterns

2.1. Fan-Out/Fan-In Pattern

The fan-out/fan-in pattern is ideal for data parallelism, where you split a large task into smaller subtasks, process them concurrently, and then combine the results:

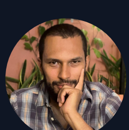
```
import asyncio
import time
import random

async def fan_out_fan_in_example():
    """
    Demonstrates the fan-out/fan-in pattern:
    1. Fan-out: Split a task into multiple subtasks
    2. Process each subtask concurrently
    3. Fan-in: Collect and combine results
    """
    async def process_chunk(chunk_id, data):
```



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

PYTHON | **CONCURRENCY (PART 3)**

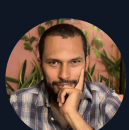
```python
    """Process a chunk of data"""
    print(f"Processing chunk {chunk_id} with {len(data)} items")
    await asyncio.sleep(random.uniform(0.5, 1.5))  # Simulate processing

    # Simulate results (sum of items with processing artifact)
    result = sum(data) * random.uniform(0.9, 1.1)
    print(f"Chunk {chunk_id} processed, result: {result:.2f}")
    return result

# Create a large dataset
dataset = [i * 2 for i in range(1000)]

# 1. Fan-out: Split data into chunks
chunk_size = 100
chunks = [dataset[i:i+chunk_size] for i in range(0, len(dataset), chunk_size)]
print(f"Split dataset into {len(chunks)} chunks of {chunk_size} items each")

```



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

PYTHON | **CONCURRENCY (PART 3)**

```python
    # 2. Process chunks concurrently
    print("\nFanning out processing to multiple tasks...")
    tasks = [
        process_chunk(i, chunk)
        for i, chunk in enumerate(chunks)
    ]

    # 3. Fan-in: Gather all results
    print("\nFanning in results...")
    start_time = time.time()
    results = await asyncio.gather(*tasks)
    end_time = time.time()

    # Combine results (in this case, take the average)
    final_result = sum(results) / len(results)

    print(f"\nAll chunks processed in {end_time - start_time:.2f} seconds")
    print(f"Final result (average): {final_result:.2f}")

```



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

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

This pattern applies to many real-world scenarios like map-reduce operations, batch processing, and parallel data analysis.

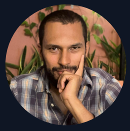
2.2. Task Queue with Priority

A priority-based task queue allows processing important tasks first:

```python
import asyncio
import random

class AsyncTaskQueue:
    """A task queue with priority and worker pool for asyncio"""

    def __init__(self, num_workers=3):
        self.queue = asyncio.PriorityQueue()
```

PYTHON | **CONCURRENCY (PART 3)**

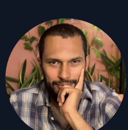
```python
        self.num_workers = num_workers
        self.workers = []
        self.running = False

    async def add_task(self, coro, priority=0):
        """Add a task to the queue with priority (lower is higher)"""
        await self.queue.put((priority, coro))

    async def worker(self, worker_id):
        """Worker that processes tasks from the queue"""
        while self.running:
            try:
                # Get a task from the queue
                priority, coro = await self.queue.get()

                try:
                    print(f"Worker {worker_id}: Processing task with priority {priority}")
                    # Execute the coroutine
                    result = await coro
```



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

PYTHON | **CONCURRENCY (PART 3)**

```python
                    print(f"Worker {worker_id}: Task completed with result: {result}")
                except Exception as e:
                    print(f"Worker {worker_id}: Task failed with error: {e}")
                finally:
                    # Mark task as done
                    self.queue.task_done()
            except asyncio.CancelledError:
                break

    async def start(self):
        """Start the worker pool"""
        self.running = True
        self.workers = [
            asyncio.create_task(self.worker(i))
            for i in range(self.num_workers)
        ]

    async def stop(self):
```



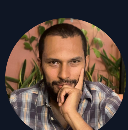
Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

www.asanchezyali.com

PYTHON | **CONCURRENCY (PART 3)**

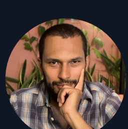
```python
        """Stop the worker pool"""
        self.running = False

        # Wait for all tasks to complete
        await self.queue.join()

        # Cancel all workers
        for worker in self.workers:
            worker.cancel()

        # Wait for all workers to complete cancellation
        await asyncio.gather(*self.workers, return_exceptions=True)

# Example task implementations
async def data_processing_task(task_id, duration):
    """Simulates a data processing task"""
    await asyncio.sleep(duration)  # Simulate work
    return f"Data for task {task_id} processed"

async def demo_task_queue():
```



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

PYTHON | **CONCURRENCY (PART 3)**

```python
    # Create a task queue
    task_queue = AsyncTaskQueue(num_workers=3)

    # Start the worker pool
    await task_queue.start()

    # Add tasks with different priorities
    for i in range(10):
        priority = random.randint(1, 3)  # 1=high, 3=low priority
        duration = random.uniform(0.5, 1.0)

        # Create a task and add it to the queue
        task = data_processing_task(i, duration)
        await task_queue.add_task(task, priority)

        print(f"Added Task {i} with priority {priority}")

    # Wait for all tasks to complete and stop the worker pool
    await task_queue.stop()
    print("All tasks completed, worker pool stopped")
```



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

```python

# Run the demo
if __name__ == "__main__":
    asyncio.run(demo_task_queue())
```

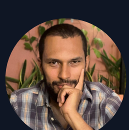
This pattern is useful for building job queues, task schedulers, and work distribution systems.

3. Performance Considerations and Best Practices

3.1. Benchmarking Concurrent Code

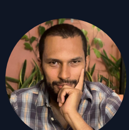
When optimizing Python code with concurrency, it's essential to measure actual performance gains:

```python
import time
import concurrent.futures

def benchmark(func, data, executor_class, max_workers=None):
    """Benchmark a function using different execution methods"""
```


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

PYTHON | **CONCURRENCY (PART 3)**

```python
    # Sequential execution (baseline)
    start = time.time()
    sequential_result = [func(item) for item in data]
    sequential_time = time.time() - start
    print(f"Sequential: {sequential_time:.4f}s")

    # Concurrent execution
    start = time.time()
    with executor_class(max_workers=max_workers) as executor:
        concurrent_result = list(executor.map(func, data))
    concurrent_time = time.time() - start
    print(f"Concurrent: {concurrent_time:.4f}s")
    print(f"Speedup: {sequential_time/concurrent_time:.2f}x")

# Example CPU-bound task
def cpu_task(n):
    """CPU-intensive calculation"""
    return sum(i * i for i in range(n * 100000))

# Example I/O-bound task
```



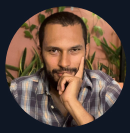
Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```python
def io_task(n):
    """I/O-bound operation (simulated)"""
    time.sleep(0.1)  # Simulate I/O delay
    return n * 2

# Demo for CPU-bound tasks
data = list(range(1, 9))
print("CPU-bound task with ProcessPoolExecutor:")
benchmark(cpu_task, data, concurrent.futures.ProcessPoolExecutor)

# Demo for I/O-bound tasks
print("\nI/O-bound task with ThreadPoolExecutor:")
benchmark(io_task, data, concurrent.futures.ThreadPoolExecutor)
```

3.2. Best Practices for Production Code

For production-grade concurrent Python applications, follow these guidelines:

- **Tool Selection:**
    - I/O-bound → asyncio or threading (asyncio preferred for new code)
    - CPU-bound → multiprocessing

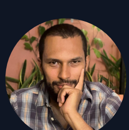


Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

    - Mixed workloads → hybrid approach
- **Resource Management:**
    - Reuse thread/process pools rather than creating new ones
    - Use context managers or ensure proper cleanup in `finally` blocks
    - Monitor memory usage, especially with multiprocessing
- **Error Handling:**
    - Properly catch and handle exceptions in worker functions
    - Use timeouts to prevent hanging operations
    - Implement graceful shutdown mechanisms
- **Avoiding Common Pitfalls:**
    - Thread Safety: Always protect shared resources with locks
    - Deadlocks: Acquire locks in a consistent order
    - Oversubscription: Don't create too many threads or processes

4. Conclusion (Part 3)

In this final part of our exploration of concurrency and parallelism in Python, we've covered advanced techniques that build on the fundamentals:

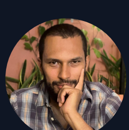


Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

- We’ve learned how to combine different concurrency models (threading, multiprocessing, and asyncio) to create hybrid solutions that leverage the strengths of each approach.
- We’ve explored advanced concurrency patterns like the fan-out/fan-in pattern and priority-based task queues that solve real-world parallelization problems.
- We’ve examined practical benchmarking approaches to quantify performance improvements and make data-driven decisions.
- We’ve identified best practices and common pitfalls to create production-ready concurrent code.

By mastering these techniques, you can develop Python applications that efficiently utilize system resources, respond quickly to events, and process data in parallel. The key is selecting the right concurrency model for each specific task and combining them when needed for complex applications.

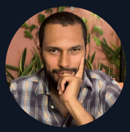
Remember that concurrency is not always the answer—sometimes a simpler sequential solution is more maintainable and even faster for small datasets. Always benchmark your code to ensure that your concurrent solution actually improves performance in your specific use case.


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

5. References (Part 3)

- Python Documentation. *asyncio - Asynchronous I/O*. Link
- Python Documentation. *concurrent.futures - Launching parallel tasks*. Link
- Real Python. *Speed Up Your Python Program With Concurrency*. Link
- Brett Slatkin. *Effective Python*, Item 57-61: Concurrency and Parallelism. Addison-Wesley.
- Caleb Hattingh. *Using Asyncio in Python: Understanding Python's Asynchronous Programming Features*. O'Reilly Media.

- Translated, Edited and written in collaboration with AI.



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

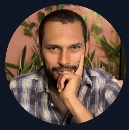
6. Explore My Other Posts

Enjoyed This Content?

Don't miss the previous part of this series:

Python Concurrency Part 2: Asyncio and HTTP Operations

Learn how to use Python's asyncio for efficient I/O operations, working with HTTP requests, and implementing producer-consumer patterns. Master the modern approach to concurrency.



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

Feedback

Found this helpful?

Save, comment and share

April 29, 2025