Python

Python Concurrency

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

March 26, 2025

Source Code

1. Understanding Concurrency and Parallelism

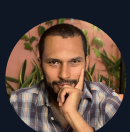
Concurrency and parallelism are fundamental concepts in Python for improving application performance, but they serve different purposes and work in distinct ways:

- **Concurrency:** Dealing with multiple tasks by switching between them (not necessarily simultaneously)
- **Parallelism:** Executing multiple tasks truly simultaneously using multiple processors
- **Performance Impact:** Proper implementation can dramatically speed up I/O-bound or CPU-bound operations

1.1. When to Use Each Approach

Choosing the right approach depends on your task type:

- **For I/O-Bound Tasks** (waiting for external resources):
  - Network requests, file operations, database queries
  - **Best approach:** Threading or Asyncio
- **For CPU-Bound Tasks** (heavy computations):
  - Data processing, image manipulation, numerical calculations
  - **Best approach:** Multiprocessing



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

2. The Global Interpreter Lock (GIL)

2.1. Understanding the GIL's Impact

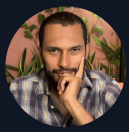
The Global Interpreter Lock (GIL) is a mutex (lock) that protects access to Python objects, preventing multiple threads from executing Python bytecode simultaneously in a single process.

- **Single-Threaded Execution:** Only one thread can execute Python code at any given time
- **I/O Release:** The GIL is released during I/O operations, allowing other threads to run
- **CPU Limitation:** CPU-bound threads still time-share a single CPU core
- **Multiprocessing Bypass:** Multiprocessing avoids the GIL by using multiple processes

Think of the GIL as a talking stick in a meeting—only the person holding it can speak (execute code), and it must be passed around even when multiple people (threads) want to talk simultaneously.

2.2. GIL Workarounds

- **Use multiprocessing** for CPU-intensive tasks



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

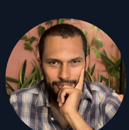
- **Leverage asyncio** for I/O-bound operations
- **Offload to C extensions** that release the GIL
- **Consider alternative implementations** like PyPy for specific use cases

3. Threading in Python

3.1. Thread Basics

Threads share memory space but execute concurrently:

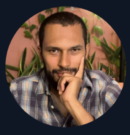
```python
import threading
import time

def worker(name, delay):
    """A simple worker function that demonstrates a thread"""
    print(f"Worker {name}: Starting")
    for i in range(3):
        time.sleep(delay)
        print(f"Worker {name}: Step {i+1}")
```



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

PYTHON | **CONCURRENCY (PART 1)**

```python
    print(f"Worker {name}: Finished")

# Create thread objects
thread1 = threading.Thread(target=worker, args=("A", 0.5))
thread2 = threading.Thread(target=worker, args=("B", 0.8))

# Start the threads
print("Main thread: Starting workers")
thread1.start()
thread2.start()

# Continue with other work in the main thread
print("Main thread: Doing other work")
time.sleep(1)
print("Main thread: Work complete")

# Wait for worker threads to finish
thread1.join()
thread2.join()
print("Main thread: All workers finished")
```

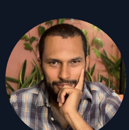


Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

3.2. Thread Safety and Locks

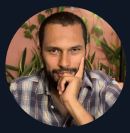
When multiple threads access shared data, synchronization is essential:

```python
import threading
import time

class Counter:
    def __init__(self):
        self.value = 0
        self.lock = threading.Lock()

    def increment(self):
        with self.lock:  # Thread-safe access to shared data
            current = self.value
            time.sleep(0.001)  # Simulate work
            self.value = current + 1
```


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

PYTHON | CONCURRENCY (PART 1)

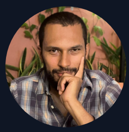
```python

    def get_value(self):
        with self.lock:
            return self.value

def increment_counter(counter, count):
    for _ in range(count):
        counter.increment()

# Create a shared counter
counter = Counter()

# Create threads to increment the counter
threads = []
num_threads = 10
increments_per_thread = 100

for _ in range(num_threads):
    t = threading.Thread(
        target=increment_counter,
```

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

```python
        args=(counter, increments_per_thread)
    )
    threads.append(t)
    t.start()

# Wait for all threads to complete
for t in threads:
    t.join()

# Check the result
expected = num_threads * increments_per_thread
actual = counter.get_value()
print(f"Expected count: {expected}")
print(f"Actual count: {actual}")
print(f"Thread-safe: {expected == actual}")
```

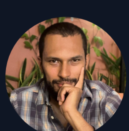
4. Multiprocessing for Parallel Computing


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

4.1. Process Basics

Processes have separate memory spaces and bypass the GIL:

```python
import multiprocessing as mp
import time
import os

def cpu_bound_task(number):
    """A CPU-intensive function that benefits from parallelization"""
    print(f"Process {os.getpid()} processing {number}")
    # Simulate CPU-intensive calculation
    result = 0
    for i in range(10**7):  # 10 million iterations
        result += i * number
    return result

if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 5, 6, 7, 8]
```

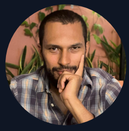


Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

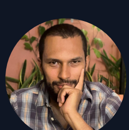
```python

    # Sequential processing
    start_time = time.time()
    sequential_results = [cpu_bound_task(n) for n in numbers]
    sequential_time = time.time() - start_time
    print(f"Sequential processing took {sequential_time:.2f} seconds")

    # Parallel processing
    start_time = time.time()
    with mp.Pool(processes=mp.cpu_count()) as pool:
        parallel_results = pool.map(cpu_bound_task, numbers)
    parallel_time = time.time() - start_time
    print(f"Parallel processing took {parallel_time:.2f} seconds")
    print(f"Speedup: {sequential_time / parallel_time:.2f}x")
```

4.2. Sharing Data Between Processes

Unlike threads, processes require special mechanisms to share data:

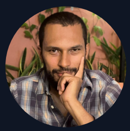

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```python
import multiprocessing as mp

def update_shared_dict(shared_dict, key, value):
    """Update a value in a shared dictionary"""
    shared_dict[key] = value
    print(f"Process {mp.current_process().name} updated shared_dict[{key}] = {value}")

def sum_shared_array(shared_array, result_queue):
    """Calculate sum of a shared array and put result in a queue"""
    total = sum(shared_array)
    result_queue.put(total)
    print(f"Process {mp.current_process().name} calculated sum: {total}")

if __name__ == "__main__":
    # Create a manager to coordinate shared objects
    with mp.Manager() as manager:
        # Create shared objects
        shared_dict = manager.dict()  # Shared dictionary
```



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

```python
        shared_array = manager.list(range(10))  # Shared list
        result_queue = manager.Queue()  # Shared queue

        # Create processes
        processes = []

        # Processes to update the shared dictionary
        for i in range(5):
            p = mp.Process(
                target=update_shared_dict,
                args=(shared_dict, f"key_{i}", i*10)
            )
            processes.append(p)

        # Process to calculate sum of shared array
        p = mp.Process(
            target=sum_shared_array,
            args=(shared_array, result_queue)
        )
        processes.append(p)
```



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

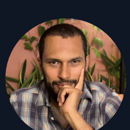
```python

        # Start all processes
        for p in processes:
            p.start()

        # Wait for all processes to finish
        for p in processes:
            p.join()

        # Retrieve and display results
        print("\nShared dictionary:", dict(shared_dict))
        print("Sum from queue:", result_queue.get())
```

5. Synchronization Tools

5.1. Thread Synchronization Primitives

Python provides several mechanisms for coordinating threads:

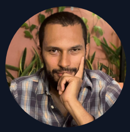
- **Lock:** The most basic synchronization primitive, prevents simultaneous access


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

PYTHON | CONCURRENCY (PART 1)

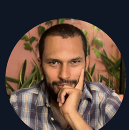
- **RLock:** Reentrant lock that can be acquired multiple times by the same thread
- **Semaphore:** Controls access to a shared resource by multiple threads
- **Event:** Signals between threads when a condition is met
- **Condition:** More sophisticated signaling mechanism for complex coordination
- **Barrier:** Makes threads wait until a specific number of them reach a common point

```python
import threading
import time

# Example: Using a semaphore to limit concurrent resource access
class ConnectionPool:
    def __init__(self, max_connections=3):
        self.semaphore = threading.Semaphore(max_connections)

```



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

```python
    def get_connection(self):
        self.semaphore.acquire()
        return self  # Return a connection object

    def release_connection(self):
        self.semaphore.release()

    def __enter__(self):
        return self.get_connection()

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.release_connection()

# Example usage
def worker(pool, worker_id):
    with pool:
        print(f"Worker {worker_id}: Acquired connection")
        time.sleep(1)  # Simulate work
        print(f"Worker {worker_id}: Released connection")

```



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

```python
# Create a connection pool with max 3 connections
pool = ConnectionPool(3)

# Create and start 10 worker threads
threads = []
for i in range(10):
    t = threading.Thread(target=worker, args=(pool, i))
    threads.append(t)
    t.start()

# Wait for all threads to complete
for t in threads:
    t.join()
```

6. Conclusion (Part 1)

In this first part of our exploration of concurrency and parallelism in Python, we have covered the essential fundamentals:

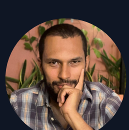
- We understood the difference between concurrency (handling multiple tasks by alternating between them) and parallelism (executing multiple tasks simultaneously).

- We analyzed the Global Interpreter Lock (GIL) and its implications for concurrent programming in Python.
- We explored the `threading` module for concurrent programming oriented to I/O operations.
- We implemented proper synchronization to protect data shared between threads.
- We studied the `multiprocessing` module for true parallel execution.
- We learned about communication mechanisms between processes.

These tools are fundamental for developing efficient Python applications that handle multiple tasks. In the second part, we will dive deeper into `asyncio`, advanced patterns, and strategies for combining different concurrency models.

7. References (Part 1)

- Python Documentation. *threading - Thread-based parallelism*. Link
- Python Documentation. *multiprocessing - Process-based parallelism*. Link
- Real Python. *Python's GIL: A Guide to How It Impacts Performance*. Link



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

- Python Discord. *Concurrency, parallelism, threading and multiprocessing*. Link
- David Beazley. *Understanding the Python GIL*. Link
- Brett Slatkin. *Effective Python*, Item 52-56: Concurrency and Parallelism. Addison-Wesley.
- Translated, Edited and written in collaboration with AI.

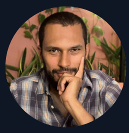
8. Explore My Other Posts

Enjoyed This Content?

Don't miss my previous post about:

Python Context Managers: Elegant Resource Management with the «with» Statement

Discover how Python Context Managers can help you elegantly manage resources, prevent memory leaks, and write cleaner code.



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

www.asanchezyali.com

Ready to Level Up Your Python?

You've mastered powerful concurrency techniques.

Apply these patterns in your next project

and watch your code become faster and more efficient!

Stay tuned for Part 2...