Python Concurrency Tutorial

A comprehensive guide to Python concurrency using Threading and Asyncio with practical examples and performance comparisons.

Threading

1. Basic Threading Concepts

- **Single Thread Creation**: Creating and starting individual threads
- **Daemon Threads**: Understanding daemon vs non-daemon threads
- **Multiple Threads**: Managing multiple threads manually
- **Thread Joining**: Waiting for threads to complete

2. ThreadPoolExecutor (Recommended Approach)

- **Basic Usage**: Using `concurrent.futures.ThreadPoolExecutor`
- **Submit Method**: Individual task submission with `executor.submit()`
- **Map Method**: Batch processing with `executor.map()`
- **Context Management**: Using `with` statements for automatic cleanup

3. Performance Comparisons

- **Sequential vs Concurrent**: Clear timing comparisons
- **Real-world Benchmarks**: Actual performance measurements
- **Threading Benefits**: Understanding when threading helps

4. Practical Real-World Example

- **Image Downloads**: Downloading multiple images concurrently
- **Network I/O**: Demonstrating threading benefits for I/O-bound tasks
- **Error Handling**: Proper exception handling in threaded code

Asyncio

1. Basic Asyncio Concepts

- **Async Functions**: Defining functions with `async def`
- **Await Keyword**: Waiting for asynchronous operations
- **Coroutines**: Working with coroutine objects

2. Asyncio Execution Patterns

- **Sequential Async**: Running async functions one after another
- **Tasks**: Creating and managing tasks with `asyncio.create_task()`
- **Gather**: Using `asyncio.gather()` for concurrent execution
- **Task Groups**: Modern approach with built-in exception handling

3. Synchronization Primitives

- **Locks**: Preventing race conditions with `asyncio.Lock()`
- **Semaphores**: Controlling access to limited resources
- **Events**: Coordination between coroutines
- **Futures**: Low-level result objects

4. Advanced Concepts

- **Race Conditions**: Understanding and preventing concurrent access issues
- **Resource Management**: Proper cleanup and error handling

Key Examples

Basic Threading

```
import threading
import time

def worker_function(name):
    print(f"Thread {name}: starting")
    time.sleep(2)
    print(f"Thread {name}: finishing")

# Create and start thread
thread = threading.Thread(target=worker_function, args=(1,))
thread.start()
thread.join()  # Wait for completion
```

ThreadPoolExecutor (Modern Approach)

```
import concurrent.futures

with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
    # Method 1: Submit individual tasks
    future1 = executor.submit(worker_function, 1)
    future2 = executor.submit(worker_function, 2)

    # Method 2: Map over multiple inputs (cleaner)
    results = executor.map(worker_function, range(3))
```

Threading vs Asyncio Comparison

Aspect	Threading	Asyncio
Best For	I/O-bound tasks with blocking calls	I/O-bound tasks with async libraries
Memory Usage	Higher (each thread ~8MB)	Lower (single thread)
Debugging	More complex (race conditions)	Easier (single-threaded)
Learning Curve	Moderate	Steeper (async/await syntax)

Project Structure

```
Python Concurrency Tutorial
├── Thread.ipynb - Threading concepts and examples
│   ├── Basic Threading Concepts
│   ├── ThreadPoolExecutor Examples
│   ├── Performance Benchmarks
│   └── Real-World Examples (Image Downloads)
└── Asyncio basics/
    ├── 1_await_n_async.py - Basic async/await syntax
    ├── 2.py - Sequential async execution
    ├── 3_tasks.py - Creating and managing tasks
    ├── 4_gather.py - Concurrent execution with gather
    ├── 5_task_groups.py - Modern task groups approach
    ├── 6_futures.py - Low-level future objects
    ├── 7_lock.py - Preventing race conditions with locks
    ├── 8_raceCondition.py - Demonstrating race conditions
    ├── 9_semaphores.py - Resource throttling with semaphores
    └── 10_event.py - Coroutine coordination with events
```

□ Prerequisites

Required Packages

```
pip install requests      # For threading examples
# No additional packages needed for asyncio basics
```

Python Version

- Python 3.7+ (tested with Python 3.10.13)
- asyncio is built into Python 3.7+

Performance Highlights

Threading Performance (10 tasks, 1.5s each)

Approach	Time
Sequential	~15 seconds
Threaded	~1.5 seconds
Speed Up	**~10x faster!**

Image Download Comparison

- **Sequential**: Downloads images one by one (10 images: ~20-30 seconds)
- **Threaded**: Downloads images concurrently (10 images: ~5-8 seconds)
- **Result**: 3-5x speed improvement!

Memory Usage Comparison

```
Sequential:   ~20MB   ████
Threading:    ~100MB  ████████████████
Asyncio:      ~25MB   ████
```

Key Learning Outcomes

After working through these examples, you'll understand:

1. **Threading vs Asyncio**: When to use each approach
2. **Async/await syntax**: Modern Python asynchronous programming
3. **Concurrency patterns**: Tasks, gather, and task groups
4. **Synchronization**: Locks, semaphores, and events
5. **Race conditions**: How to identify and prevent them
6. **Performance benefits**: Real timing comparisons for both approaches

Advanced Concepts Covered

Threading Advanced Topics

- **Context Managers**: Using `with` statements for resource management
- **Future Objects**: Understanding asynchronous result handling
- **Exception Handling**: Proper error handling in concurrent code
- **Performance Measurement**: Using `time.perf_counter()` for accurate timing

Asyncio Advanced Topics

- **Event Loop Management**: Understanding the asyncio event loop
- **Synchronization Primitives**: Locks, semaphores, and events
- **Task Management**: Creating and coordinating multiple tasks
- **Race Condition Prevention**: Proper resource sharing

Best Practices Demonstrated

1. **Choose the right tool**:
    - Use **asyncio** for new projects with I/O-bound tasks
    - Use **threading** when working with existing sync libraries
2. **Resource management**: Always use context managers

3. **Error handling**: Proper exception handling in concurrent code
4. **Performance measurement**: Quantify improvements with timing
5. **Synchronization**: Prevent race conditions with proper locking

When to Use What?

Use Asyncio When:

- Building new applications with I/O-bound tasks
- Need to handle many concurrent operations
- Memory usage is a concern
- Want easier debugging (single-threaded)

Use Threading When:

- Working with existing synchronous libraries
- Need to integrate with legacy code
- Blocking operations that can't be made async

Notes

- **Threading is perfect for I/O-bound tasks** (network requests, file operations)
- **Asyncio is ideal for I/O-bound tasks** when using async libraries
- **Both are ineffective for CPU-bound tasks** due to Python's GIL
- **ThreadPoolExecutor** is the modern, recommended approach over manual threading
- **Proper synchronization is crucial** for both approaches to avoid race conditions
- Always use **context managers** (`with` statements) for resource management