Advanced Python Programming

LESSON 3: Advanced Asynchronous Programming

Learning Objectives:

By the end of this lesson, participants will be able to:

- Understand the event loop and cooperative multitasking fundamentals.
- Run coroutines concurrently using asyncio for improved performance.
- Manage tasks effectively with creation, cancellation, and structured concurrency.
- Apply TaskGroup for safe concurrent execution and exception handling.

Lesson Outline:

I. The Event Loop and Cooperative Multitasking (10 min)

Mental Model: The Event Loop as a Restaurant Manager

Think of the event loop like a restaurant manager coordinating multiple waiters:

- **Single Manager**: One person (thread) coordinates everything
- **Cooperative Staff**: Waiters voluntarily check in ("I'm waiting for the kitchen")
- **No Blocking**: Manager never waits - always finds someone ready to work
- **Event-Driven**: Responds to events like "table 5 is ready" or "kitchen finished order 3"

Let's build this understanding step by step:

Understanding the Event Loop:

- **Single-threaded**: One thread handles all async operations
- **Cooperative**: Tasks must explicitly yield control with `await`
- **Non-blocking**: I/O operations don't block the entire program
- **Event-driven**: Responds to events like I/O completion, timers, etc.

Step 1: Understanding the difference between sync and async

```
import time

# Synchronous approach - blocking
def sync_task(name, duration):
    """Synchronous task that blocks."""
    print(f"Starting {name}")
    time.sleep(duration)  # This BLOCKS everything
    print(f"Finished {name}")
    return f"Result from {name}"

def sync_example():
    """Synchronous execution - one thing at a time."""
    print("=== Synchronous Execution ===")
```

```
    start_time = time.time()

    result1 = sync_task("Task 1", 2)
    result2 = sync_task("Task 2", 1)

    total_time = time.time() - start_time
    print(f"Total time: {total_time:.2f}s")
    print(f"Results: {result1}, {result2}")

# Run synchronous example
sync_example()
```

Key Insight: In synchronous code, everything waits. When one task sleeps, the entire program sleeps.

Step 2: The async difference

```
import asyncio

# Asynchronous approach - non-blocking
async def async_task(name, duration):
    """Async task that yields control."""
    print(f"Starting {name}")
    await asyncio.sleep(duration)  # This YIELDS control
    print(f"Finished {name}")
    return f"Result from {name}"

async def async_example():
    """Asynchronous execution - cooperative multitasking."""
    print("\n=== Asynchronous Execution ===")
    start_time = time.time()

    # Create tasks but don't wait yet
    task1 = asyncio.create_task(async_task("Task 1", 2))
    task2 = asyncio.create_task(async_task("Task 2", 1))

    # Now wait for both to complete
    result1 = await task1
    result2 = await task2

    total_time = time.time() - start_time
    print(f"Total time: {total_time:.2f}s")
    print(f"Results: {result1}, {result2}")

# Run async example
asyncio.run(async_example())
```

Key Insight: In async code, when one task waits (like for I/O), other tasks can run. The event loop manages this switching.

Basic event loop example:

```python
import asyncio
import time

async def say_hello(name, delay):
    """Async function that simulates work with a delay."""
    print(f"Starting task for {name}")
    await asyncio.sleep(delay)  # Non-blocking sleep
    print(f"Hello, {name}! (after {delay}s)")
    return f"Task {name} completed"

async def main():
    """Main async function to coordinate tasks."""
    print("Starting main")

    # Sequential execution (slow)
    start_time = time.time()
    await say_hello("Alice", 2)
    await say_hello("Bob", 1)
    sequential_time = time.time() - start_time

    print(f"Sequential execution took: {sequential_time:.2f}s")

    # Concurrent execution (fast)
    start_time = time.time()
    task1 = asyncio.create_task(say_hello("Charlie", 2))
    task2 = asyncio.create_task(say_hello("Diana", 1))

    result1 = await task1
    result2 = await task2
    concurrent_time = time.time() - start_time

    print(f"Concurrent execution took: {concurrent_time:.2f}s")
    print(f"Results: {result1}, {result2}")

# Run the event loop
if __name__ == "__main__":
    asyncio.run(main())
```

Event loop introspection:

```python
import asyncio

async def examine_event_loop():
    """Examine the current event loop properties."""
    loop = asyncio.get_running_loop()

    print(f"Event loop: {loop}")
    print(f"Is running: {loop.is_running()}")
    print(f"Debug mode: {loop.get_debug()}")
```

```
    # Schedule a callback
    def callback():
        print("Callback executed!")

    loop.call_later(1.0, callback)
    await asyncio.sleep(1.5)  # Wait for callback to execute

# asyncio.run(examine_event_loop())
```

Commentary:

The event loop enables concurrency without the complexity of threading. Understanding when tasks yield control is crucial for writing efficient async code.

II. Running Coroutines Concurrently with asyncio (15 min)

Mental Model: Async as a Juggler

Think of asyncio like a skilled juggler:

- **Multiple Objects**: Can handle many tasks at once
- **Single Person**: One event loop manages everything
- **Timing**: Catches and throws at just the right moments
- **Coordination**: When one ball is in the air, attention goes to others

Python's asyncio module provides several patterns for running coroutines concurrently.

Step 1: Understanding the difference between sequential and concurrent async

```
import asyncio
import time

async def fetch_data(source, delay):
    """Simulate fetching data with a delay."""
    print(f"Starting fetch from {source}")
    await asyncio.sleep(delay)  # Simulate I/O delay
    print(f"Finished fetching from {source}")
    return f"Data from {source}"

async def sequential_execution():
    """Run tasks one after another - SLOW."""
    print("=== Sequential Async Execution ===")
    start_time = time.time()

    # Wait for each one to finish before starting the next
    result1 = await fetch_data("API-1", 2)
    result2 = await fetch_data("API-2", 1)
    result3 = await fetch_data("API-3", 1.5)

    total_time = time.time() - start_time
```

```
    print(f"Sequential total time: {total_time:.2f}s")
    return [result1, result2, result3]

async def concurrent_execution():
    """Run tasks concurrently - FAST."""
    print("\n=== Concurrent Async Execution ===")
    start_time = time.time()

    # Start all tasks at once
    task1 = asyncio.create_task(fetch_data("API-1", 2))
    task2 = asyncio.create_task(fetch_data("API-2", 1))
    task3 = asyncio.create_task(fetch_data("API-3", 1.5))

    # Wait for all to complete
    result1 = await task1
    result2 = await task2
    result3 = await task3

    total_time = time.time() - start_time
    print(f"Concurrent total time: {total_time:.2f}s")
    return [result1, result2, result3]

# Compare both approaches
async def compare_execution():
    await sequential_execution()
    await concurrent_execution()

asyncio.run(compare_execution())
```

Key Insight: Sequential async is still slow (2 + 1 + 1.5 = 4.5s), but concurrent async is fast (max(2, 1, 1.5) = 2s).

Using asyncio.gather() for concurrent execution:

Step 2: Using asyncio.gather() - The Easy Way

`asyncio.gather()` is like saying "start all these tasks and tell me when ALL are done":

```
import asyncio
import aiohttp
import time

async def fetch_url(session, url, delay=1):
    """Simulate fetching data from a URL."""
    print(f"Starting fetch: {url}")
    await asyncio.sleep(delay)  # Simulate network delay
    print(f"Finished fetch: {url}")
    return f"Data from {url}"

async def gather_example():
    """Using gather() to run multiple operations concurrently."""
```

```
    urls = [
        "https://api.example1.com/data",
        "https://api.example2.com/data",
        "https://api.example3.com/data"
    ]

    start_time = time.time()

    # Method 1: Using gather - starts all at once, waits for all
    print("=== Using asyncio.gather() ===")
    tasks = [fetch_url(None, url, delay=2) for url in urls]
    results = await asyncio.gather(*tasks)

    end_time = time.time()
    print(f"Gathered {len(results)} results in {end_time -
start_time:.2f}s")
    print(f"Results: {results}")
    return results

# asyncio.run(gather_example())
```

Mental Model for gather(): Like waiting for all your friends to finish getting ready before leaving for a party - you wait for the slowest person.

Step 3: Understanding exception handling with gather()

async def gather_with_exceptions(): """Handle exceptions in concurrent operations."""

```
async def might_fail(name, should_fail=False):
    await asyncio.sleep(1)
    if should_fail:
        raise ValueError(f"Task {name} failed!")
    print(f"Task {name} succeeded")
    return f"Success: {name}"

print("\n=== Exception Handling with gather() ===")

tasks = [
    might_fail("Task1", False),
    might_fail("Task2", True),   # This will fail
    might_fail("Task3", False)
]

# Default behavior: stops on first exception
try:
    print("Trying gather() without return_exceptions...")
    results = await asyncio.gather(*tasks)
    print(f"All succeeded: {results}")
except ValueError as e:
    print(f"gather() failed fast: {e}")
```

```
# Better behavior: collect all results including exceptions
print("\nTrying gather() with return_exceptions=True...")
results = await asyncio.gather(*tasks, return_exceptions=True)

for i, result in enumerate(results):
    if isinstance(result, Exception):
        print(f"Task {i+1} failed: {result}")
    else:
        print(f"Task {i+1} succeeded: {result}")
```

asyncio.run(gather_with_exceptions())

```
**Key Insight**: `return_exceptions=True` is usually what you want - it
lets you handle failures gracefully instead of crashing everything.
```

Using asyncio.as_completed() for processing results as they arrive:

Mental Model: as_completed() as a Race Finish Line

Think of `as_completed()` like watching runners cross a finish line:

- **Different Speeds**: Some tasks finish faster than others
- **Process Winners**: Handle each result as soon as it's ready
- **No Waiting**: Don't wait for slow runners to process fast ones
- **Order Doesn't Matter**: Process results in completion order, not start order

When to use as_completed(): When you want to process results immediately as they become available, rather than waiting for everything to finish.

```
import asyncio
import random

async def download_file(file_id, size_mb):
    """Simulate downloading a file of given size."""
    # Simulate variable download time based on size + some randomness
    download_time = size_mb * 0.1 + random.uniform(0.5, 2.0)

    print(f"Starting download of file {file_id} ({size_mb}MB)")
    await asyncio.sleep(download_time)
    print(f" Completed download of file {file_id} in
{download_time:.1f}s")

    return {"file_id": file_id, "size_mb": size_mb, "time": download_time}
```

```
async def download_with_progress():
    """Download files and show progress as they complete."""
    files = [
        (1, 10), (2, 5), (3, 15), (4, 8), (5, 12)
    ]

    print("=== Download Progress with as_completed() ===")

    # Create all download tasks
    tasks = [download_file(file_id, size) for file_id, size in files]

    completed_count = 0
    total_size = 0

    # Process results as they complete (not in order!)
    for coro in asyncio.as_completed(tasks):
        result = await coro
        completed_count += 1
        total_size += result["size_mb"]

        print(f" Progress: {completed_count}/{len(files)} files
completed")
        print(f" Total downloaded: {total_size}MB")
        print(f" Last completed: File {result['file_id']}
({result['size_mb']}MB)")
        print("-" * 50)

async def compare_gather_vs_as_completed():
    """Compare gather() vs as_completed() approaches."""

    files = [(1, 3), (2, 1), (3, 2)]  # Different sizes = different
completion times

    print("\n=== Comparison: gather() vs as_completed() ===")

    # Using gather() - wait for all, then process all
    print("Using gather() - batch processing:")
    start_time = asyncio.get_event_loop().time()
    tasks = [download_file(f"G{file_id}", size) for file_id, size in
files]
    results = await asyncio.gather(*tasks)

    print("Processing all results at once:")
    for result in results:
        print(f"  Processed file {result['file_id']}")

    # Using as_completed() - process each as it finishes
    print("\nUsing as_completed() - streaming processing:")
    tasks = [download_file(f"S{file_id}", size) for file_id, size in
files]

    for coro in asyncio.as_completed(tasks):
        result = await coro
        print(f"  Immediately processed file {result['file_id']}")
```

```
# asyncio.run(download_with_progress())
# asyncio.run(compare_gather_vs_as_completed())
```

Key Differences:

- **gather()**: "Wait for everyone, then party" - batch processing
- **as_completed()**: "Celebrate each victory" - streaming processing

When to use each:

- Use **gather()** when you need all results together
- Use **as_completed()** when you want to show progress or start processing early results

```
##### Commentary:
`gather()` is great when you need all results, while `as_completed()` is
better for processing results immediately as they become available.

---

#### III. Task Management and Cancellation (10 min)

**Mental Model: Tasks as Remote Control Devices**

Think of async tasks like devices you can control with a remote:
- **Create**: Turn on the device (start the task)
- **Check Status**: Is it still running? Did it finish? Did it break?
- **Cancel**: Hit the stop button (graceful shutdown)
- **Force Stop**: Pull the power cord (immediate termination)

Understanding task lifecycles is crucial for building robust async
applications.

**Step 1: Understanding task states**

```python
import asyncio

async def simple_task(name, duration):
    """A simple task that we can observe."""
    print(f"Task {name} starting")
    try:
        await asyncio.sleep(duration)
        print(f"Task {name} completed normally")
        return f"Result from {name}"
    except asyncio.CancelledError:
        print(f"Task {name} was cancelled!")
        raise  # Important: re-raise the cancellation

async def task_lifecycle_demo():
    """Demonstrate task creation and lifecycle."""
```
```

```
    print("=== Task Lifecycle Demo ===")

    # Create a task (starts immediately)
    task = asyncio.create_task(simple_task("Demo", 3))

    print(f"Just created task:")
    print(f"  Done: {task.done()}")
    print(f"  Cancelled: {task.cancelled()}")

    # Wait a bit and check again
    await asyncio.sleep(1)
    print(f"After 1 second:")
    print(f"  Done: {task.done()}")
    print(f"  Cancelled: {task.cancelled()}")

    # Wait for completion
    result = await task
    print(f"After completion:")
    print(f"  Done: {task.done()}")
    print(f"  Result: {result}")

# asyncio.run(task_lifecycle_demo())
```

Creating and managing tasks:

Step 2: Task cancellation patterns

```
import asyncio

async def long_running_task(name, duration):
    """A task that can be cancelled gracefully."""
    try:
        print(f"Task {name} starting (will run for {duration}s)")
        for i in range(duration):
            await asyncio.sleep(1)  # Cancellation points
            print(f"Task {name}: step {i+1}/{duration}")
        print(f"Task {name} completed normally")
        return f"Result from {name}"

    except asyncio.CancelledError:
        print(f"Task {name} was cancelled!")
        # Cleanup code can go here
        print(f"Task {name} cleaning up...")
        raise  # Re-raise the cancellation

async def cancellation_demo():
    """Demonstrate task cancellation patterns."""
    print("\n=== Task Cancellation Demo ===")

    # Create two tasks with different durations
    fast_task = asyncio.create_task(long_running_task("Fast", 3))
```

```
    slow_task = asyncio.create_task(long_running_task("Slow", 8))

    # Wait a bit, then cancel the slow task
    await asyncio.sleep(2)

    print("\nChecking task status after 2 seconds:")
    print(f"Fast task - Done: {fast_task.done()}, Cancelled:
{fast_task.cancelled()}")
    print(f"Slow task - Done: {slow_task.done()}, Cancelled:
{slow_task.cancelled()}")

    # Cancel the slow task
    print("\nCancelling slow task...")
    slow_task.cancel()

    # Wait for fast task to complete
    try:
        result = await fast_task
        print(f"Fast task result: {result}")
    except Exception as e:
        print(f"Fast task error: {e}")

    # Handle cancelled task
    try:
        result = await slow_task
        print(f"Slow task result: {result}")
    except asyncio.CancelledError:
        print("Slow task was successfully cancelled")

# asyncio.run(cancellation_demo())
```

Key Insights about Cancellation:

1. **Graceful**: `CancelledError` allows cleanup before stopping
2. **Cooperative**: Tasks must await something to be cancellable
3. **Propagates**: Always re-raise `CancelledError` unless you have a good reason
4. **Immediate**: `task.cancel()` requests cancellation at the next await point

```
##### Timeout handling:

**Mental Model: Timeouts as Alarm Clocks**

Think of timeouts like setting an alarm clock:
- **Set the Alarm**: "Wake me up in 5 seconds, regardless of what I'm
doing"
- **Two Outcomes**: Either you finish your task, or the alarm goes off
- **Forced Wakeup**: If the alarm goes off, you stop what you're doing
immediately

```python
```
```

```
import asyncio

async def may_take_long(duration, task_name="Unknown"):
    """Function that might take too long."""
    print(f"Starting {task_name}, will take {duration}s")
    await asyncio.sleep(duration)
    print(f"Finished {task_name}")
    return f"Completed {task_name} after {duration}s"

async def timeout_examples():
    """Demonstrate different timeout handling patterns."""

    print("=== Timeout Handling Examples ===")

    # Example 1: Task finishes in time
    print("\n1. Task that finishes in time:")
    try:
        result = await asyncio.wait_for(
            may_take_long(2, "Quick Task"),
            timeout=3.0
        )
        print(f" Success: {result}")
    except asyncio.TimeoutError:
        print(" Timed out (this shouldn't happen)")

    # Example 2: Task times out
    print("\n2. Task that times out:")
    try:
        result = await asyncio.wait_for(
            may_take_long(5, "Slow Task"),
            timeout=2.0
        )
        print(f" Success: {result}")
    except asyncio.TimeoutError:
        print(" Task timed out after 2 seconds!")

    # Example 3: Multiple tasks with timeout
    print("\n3. Multiple tasks with timeout:")
    try:
        tasks = [
            may_take_long(1, "Fast"),
            may_take_long(4, "Slow"),  # This will cause timeout
            may_take_long(2, "Medium")
        ]

        results = await asyncio.wait_for(
            asyncio.gather(*tasks),
            timeout=3.0
        )
        print(f" All completed: {results}")
    except asyncio.TimeoutError:
        print(" Some tasks timed out!")

# asyncio.run(timeout_examples())
```

Step 2: Using asyncio.wait() for more control

```
async def timeout_with_wait():
    """Using asyncio.wait for more sophisticated timeout handling."""
    print("\n=== Advanced Timeout with asyncio.wait() ===")

    # Create multiple tasks
    tasks = [
        asyncio.create_task(may_take_long(1, "Task-1")),
        asyncio.create_task(may_take_long(3, "Task-2")),
        asyncio.create_task(may_take_long(5, "Task-3"))
    ]

    # Wait with timeout - see what completed
    done, pending = await asyncio.wait(tasks, timeout=2.0)

    print(f"After 2 seconds:")
    print(f"  Completed tasks: {len(done)}")
    print(f"  Still running: {len(pending)}")

    # Get results from completed tasks
    for task in done:
        try:
            result = await task
            print(f"   Completed: {result}")
        except Exception as e:
            print(f"   Failed: {e}")

    # Cancel remaining tasks
    for task in pending:
        print(f"  Cancelling pending task...")
        task.cancel()

    # Wait for cancellation to complete
    if pending:
        await asyncio.gather(*pending, return_exceptions=True)

    print("All tasks handled!")

# asyncio.run(timeout_with_wait())
```

Key Timeout Patterns:

- `wait_for()`: Simple timeout for single operations
- `wait()`: More control, can handle partial completion
- **Always handle pending tasks**: Cancel them to prevent resource leaks

````
##### Commentary:
Proper task cancellation is crucial for responsive applications. Always
handle `CancelledError` appropriately and include cleanup code when
necessary.

---

#### IV. Structured Concurrency with TaskGroup (Python 3.11+) (10 min)

TaskGroup provides a safer way to manage concurrent tasks with automatic
cleanup and exception handling.

##### Basic TaskGroup usage:

```python
import asyncio

async def worker_task(worker_id, work_time, should_fail=False):
    """A worker task that might fail."""
    try:
        print(f"Worker {worker_id} starting")
        await asyncio.sleep(work_time)

        if should_fail:
            raise ValueError(f"Worker {worker_id} encountered an error!")

        print(f"Worker {worker_id} completed successfully")
        return f"Result from worker {worker_id}"

    except asyncio.CancelledError:
        print(f"Worker {worker_id} was cancelled")
        raise

async def taskgroup_basic_demo():
    """Demonstrate basic TaskGroup usage."""
    try:
        async with asyncio.TaskGroup() as tg:
            # Create multiple tasks
            task1 = tg.create_task(worker_task(1, 2))
            task2 = tg.create_task(worker_task(2, 3))
            task3 = tg.create_task(worker_task(3, 1))

        # All tasks completed successfully
        print(f"Results: {task1.result()}, {task2.result()},
{task3.result()}")

    except* ValueError as eg:
        # Handle exception group
        print(f"Some tasks failed: {eg}")

# Python 3.11+ required
# asyncio.run(taskgroup_basic_demo())
````
````

TaskGroup with exception handling:

```
import asyncio

async def taskgroup_with_errors():
    """Demonstrate TaskGroup exception handling."""
    async def reliable_worker(worker_id):
        await asyncio.sleep(1)
        return f"Worker {worker_id} success"

    async def unreliable_worker(worker_id):
        await asyncio.sleep(2)
        raise RuntimeError(f"Worker {worker_id} failed")

    try:
        async with asyncio.TaskGroup() as tg:
            # Mix of reliable and unreliable workers
            good_task1 = tg.create_task(reliable_worker(1))
            bad_task = tg.create_task(unreliable_worker(2))
            good_task2 = tg.create_task(reliable_worker(3))

        # This won't be reached if any task fails
        print("All tasks completed successfully")

    except* RuntimeError as eg:
        print(f"Caught exception group with {len(eg.exceptions)}
exceptions")
        for exc in eg.exceptions:
            print(f"  Exception: {exc}")

        # Check which tasks completed before the failure
        print(f"Good task 1 done: {good_task1.done()}")
        if good_task1.done() and not good_task1.cancelled():
            print(f"Good task 1 result: {good_task1.result()}")

# Python 3.11+ required
# asyncio.run(taskgroup_with_errors())
```

Alternative pattern for older Python versions:

```
import asyncio

async def structured_concurrency_legacy():
    """Structured concurrency pattern for Python < 3.11."""

    async def managed_worker(worker_id, duration):
        try:
            await asyncio.sleep(duration)
```

```
            return f"Worker {worker_id} completed"
        except asyncio.CancelledError:
            print(f"Worker {worker_id} cancelled")
            raise

    tasks = []
    try:
        # Create tasks
        tasks.append(asyncio.create_task(managed_worker(1, 1)))
        tasks.append(asyncio.create_task(managed_worker(2, 2)))
        tasks.append(asyncio.create_task(managed_worker(3, 3)))

        # Wait for all to complete
        results = await asyncio.gather(*tasks, return_exceptions=True)

        # Process results
        for i, result in enumerate(results):
            if isinstance(result, Exception):
                print(f"Task {i+1} failed: {result}")
            else:
                print(f"Task {i+1} result: {result}")

    except Exception as e:
        print(f"Error in task management: {e}")

    finally:
        # Ensure all tasks are cancelled if something goes wrong
        for task in tasks:
            if not task.done():
                task.cancel()

        # Wait for cancellation to complete
        if tasks:
            await asyncio.gather(*tasks, return_exceptions=True)

# asyncio.run(structured_concurrency_legacy())
```

Commentary:

TaskGroup ensures that if any task fails, all other tasks in the group are cancelled, preventing resource leaks and providing cleaner error handling.

V. Real-World Async Patterns (10 min)

Let's explore practical patterns for building robust asynchronous applications.

Producer-Consumer pattern with queues:

```python
import asyncio
import random

async def producer(queue, producer_id, num_items):
    """Produce items and put them in the queue."""
    for i in range(num_items):
        # Simulate work to create an item
        await asyncio.sleep(random.uniform(0.1, 0.5))

        item = f"item-{producer_id}-{i}"
        await queue.put(item)
        print(f"Producer {producer_id} produced: {item}")

    print(f"Producer {producer_id} finished")

async def consumer(queue, consumer_id):
    """Consume items from the queue."""
    while True:
        try:
            # Wait for an item with timeout
            item = await asyncio.wait_for(queue.get(), timeout=2.0)

            # Simulate processing time
            await asyncio.sleep(random.uniform(0.2, 0.8))

            print(f"Consumer {consumer_id} processed: {item}")
            queue.task_done()

        except asyncio.TimeoutError:
            print(f"Consumer {consumer_id} timed out waiting for items")
            break

async def producer_consumer_demo():
    """Demonstrate producer-consumer pattern."""
    # Create a queue with limited size
    queue = asyncio.Queue(maxsize=5)

    # Create producers and consumers
    producers = [
        asyncio.create_task(producer(queue, i, 3))
        for i in range(2)
    ]

    consumers = [
        asyncio.create_task(consumer(queue, i))
        for i in range(3)
    ]

    # Wait for all producers to finish
    await asyncio.gather(*producers)

    # Wait for queue to be empty
    await queue.join()
```

```
    # Cancel consumers (they run indefinitely)
    for c in consumers:
        c.cancel()

    await asyncio.gather(*consumers, return_exceptions=True)
    print("Producer-consumer demo completed")

# asyncio.run(producer_consumer_demo())
```

Rate limiting and connection pooling:

```
import asyncio
import time

class RateLimiter:
    """Simple rate limiter using semaphore."""

    def __init__(self, max_calls, time_window):
        self.max_calls = max_calls
        self.time_window = time_window
        self.calls = []
        self.lock = asyncio.Lock()

    async def acquire(self):
        async with self.lock:
            now = time.time()

            # Remove old calls outside the time window
            self.calls = [call_time for call_time in self.calls
                          if now - call_time < self.time_window]

            # Check if we can make a new call
            if len(self.calls) < self.max_calls:
                self.calls.append(now)
                return True

            # Calculate wait time
            oldest_call = min(self.calls)
            wait_time = self.time_window - (now - oldest_call)
            await asyncio.sleep(wait_time)

            # Try again
            return await self.acquire()

async def rate_limited_request(limiter, request_id):
    """Make a rate-limited request."""
    await limiter.acquire()

    print(f"Making request {request_id} at {time.time():.2f}")
    await asyncio.sleep(0.1)  # Simulate request processing
```

```
    return f"Response for request {request_id}"

async def rate_limiting_demo():
    """Demonstrate rate limiting."""
    # Allow 3 calls per 2-second window
    limiter = RateLimiter(max_calls=3, time_window=2.0)

    # Create many requests
    tasks = [
        rate_limited_request(limiter, i)
        for i in range(8)
    ]

    start_time = time.time()
    results = await asyncio.gather(*tasks)
    end_time = time.time()

    print(f"Completed {len(results)} requests in {end_time -
start_time:.2f}s")

# asyncio.run(rate_limiting_demo())
```

Commentary:

These patterns are essential for building scalable, robust asynchronous applications that handle real-world constraints like rate limits and resource management.

VI. Performance Monitoring and Debugging (5 min)

Understanding how to monitor and debug async code is crucial for production applications.

Async performance monitoring:

```
import asyncio
import time
import functools

def async_timer(func):
    """Decorator to time async functions."""
    @functools.wraps(func)
    async def wrapper(*args, **kwargs):
        start_time = time.time()
        try:
            result = await func(*args, **kwargs)
            return result
        finally:
            end_time = time.time()
            print(f"{func.__name__} took {end_time - start_time:.3f}s")
```

```
    return wrapper

@async_timer
async def slow_operation(duration):
    """A slow async operation."""
    await asyncio.sleep(duration)
    return f"Completed in {duration}s"

async def monitoring_demo():
    """Demonstrate async monitoring."""
    # Monitor individual operations
    await slow_operation(1.0)
    await slow_operation(0.5)

    # Monitor concurrent operations
    start_time = time.time()
    tasks = [slow_operation(i * 0.5) for i in range(1, 4)]
    results = await asyncio.gather(*tasks)
    total_time = time.time() - start_time

    print(f"Total concurrent execution time: {total_time:.3f}s")

# asyncio.run(monitoring_demo())
```

Commentary:

Monitoring async performance helps identify bottlenecks and ensures your concurrent code is actually running concurrently.

VII. Recap & Best Practices (5 min)

Key Takeaways:

- The event loop enables cooperative multitasking in a single thread.
- Use `asyncio.gather()` when you need all results, `as_completed()` for streaming results.
- Proper task cancellation and timeout handling prevent resource leaks.
- TaskGroup (Python 3.11+) provides structured concurrency with automatic cleanup.
- Producer-consumer patterns and rate limiting are essential for robust applications.

Best Practices:

- Always handle `CancelledError` in long-running tasks.
- Use timeouts to prevent hanging operations.
- Prefer structured concurrency patterns (TaskGroup) when available.
- Monitor performance to ensure true concurrency.
- Use queues for producer-consumer patterns.
- Implement rate limiting for external API calls.

Common Pitfalls:

- Forgetting to await coroutines (they won't run).
- Blocking operations in async functions (use async alternatives).
- Not handling task cancellation properly.
- Creating too many concurrent tasks without limits.

Final Multiple-Choice Question:

What happens when one task in an `asyncio.TaskGroup` raises an exception?

A. Only that task fails, others continue running B. All tasks in the group are immediately cancelled C. The exception is silently ignored D. The entire program crashes

(Answer: B. All tasks in the group are immediately cancelled - TaskGroup provides structured concurrency where failure of one task cancels all others in the group.)