Python Advanced Concepts and Performance Optimization

A comprehensive guide for experienced developers seeking to master performance tuning, concurrency, and advanced patterns in Python

Instructor: Chandrashekar Babu

Day 2 Agenda

01

Python Performance Optimization Techniques

Exploring JIT/AOT compilers, PyPy, Numba, Cython and other performance-enhancing tools

02

Python Threads vs Process

Understanding the GIL, Python 3.12 enhancements, and strategies for effective concurrency

03

Decorators and AOP Patterns

Advanced implementation techniques and real-world applications of Aspect-Oriented Programming

Today we'll focus on practical techniques to optimise your Python code and implement advanced programming patterns that can be immediately applied to your projects.

Performance Optimization: Overview

Python's standard implementation (CPython) prioritises correctness, maintainability, and developer productivity over raw performance. This creates opportunities for optimization when speed becomes critical.

Key performance challenges in CPython:

- Interpreted nature (bytecode execution)
- Dynamic typing overhead
- Memory management costs
- Global Interpreter Lock (GIL) limitations

Despite these constraints, Python offers numerous paths to dramatic performance improvements without sacrificing readability or maintainability.

JIT and AOT Compilation in Python

Just-In-Time (JIT) Compilation

Compiles code during execution, typically after identifying "hot" sections that would benefit from optimization

- Adaptive optimization based on runtime data
- Balance between startup time and execution speed
- Example implementations: PyPy, Numba

Ahead-Of-Time (AOT) Compilation

Converts Python code to machine code before execution

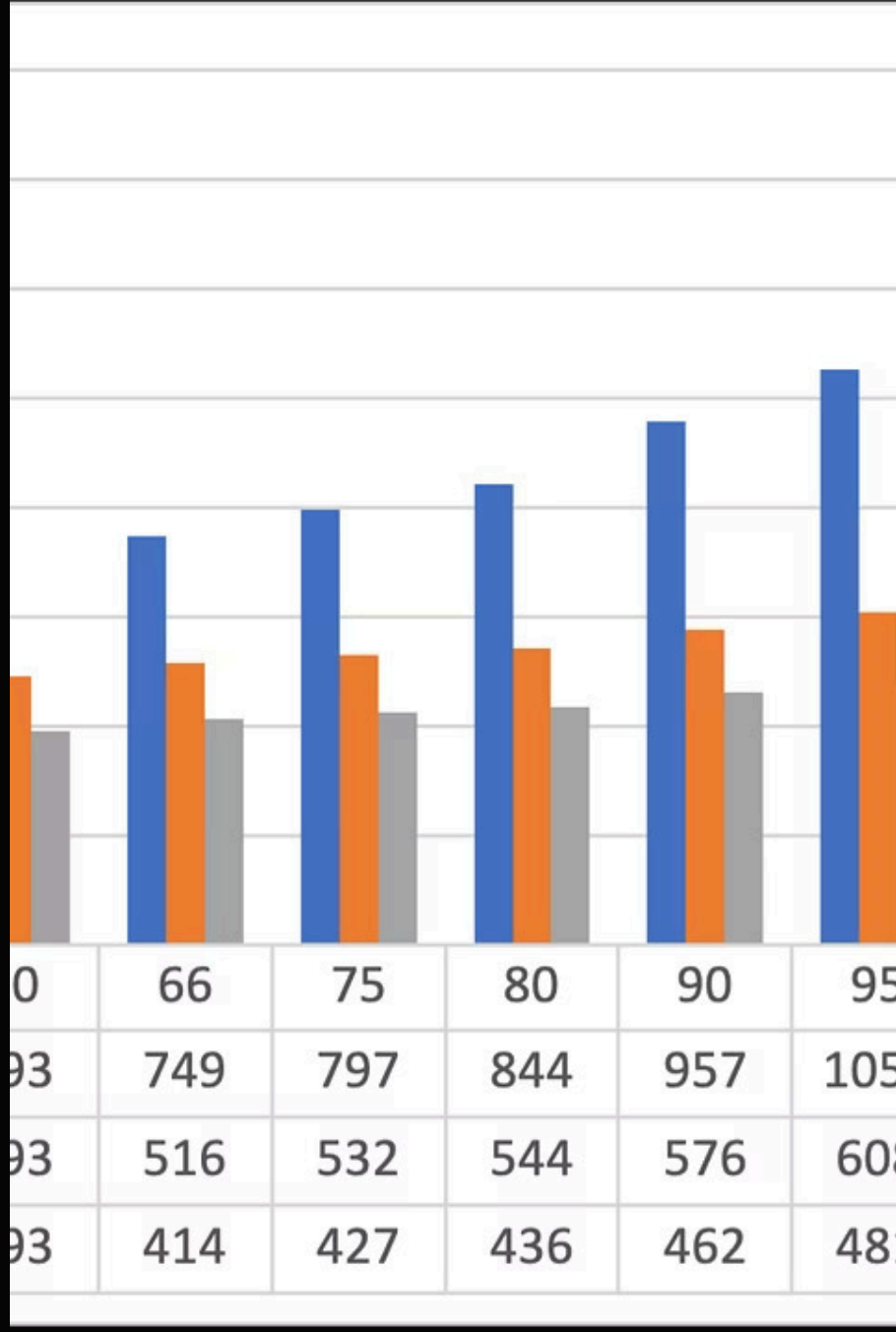
- Eliminates interpreter overhead entirely
- Longer build times, potentially larger binaries
- Example implementations: Cython, Nuitka, Pythran

Both approaches can yield significant performance gains, often 2-100x depending on the workload and optimization strategy. The right choice depends on your specific use case, deployment constraints, and performance requirements.

Python Performance Benchmark

Benchmarks consistently show that alternative Python implementations and compilers can dramatically outperform CPython for computation-heavy workloads. The chart above shows relative performance across different implementation options, with higher numbers indicating better performance.

Note that no single option is universally superior—each has strengths for specific workloads. We'll explore these nuances throughout today's session.



PyPy: A Drop-in JIT Compiler

What is PyPy?

PyPy is an alternative Python interpreter with a built-in Just-In-Time compiler that can dramatically improve execution speed while maintaining compatibility with CPython.

Key benefits:

- Speed improvements of 4-10x for many workloads
- Minimal code changes required (often none)
- Memory-efficient for long-running processes
- Reduced GIL impact for some workloads

PyPy works by identifying frequently executed code paths and compiling them to highly optimized machine code during execution.

PyPy Performance in Action

```
# Example: Computing Mandelbrot set
def mandelbrot(h, w, max_iters):
    y, x = np.ogrid[-1.4:1.4:h*1j, -2:0.8:w*1j]
    c = x + y*1j
    z = c
    divtime = max_iters + np.zeros(z.shape, dtype=int)

    for i in range(max_iters):
        z = z**2 + c
        diverge = z*np.conj(z) > 2**2
        div_now = diverge & (divtime == max_iters)
        divtime[div_now] = i
        z[diverge] = 2

    return divtime

# CPython: ~4.2 seconds
# PyPy: ~0.8 seconds
```

This computation-intensive example demonstrates PyPy's strength: pure Python code with many iterations and mathematical operations can see 5x or greater speedups with no code changes required.

When to Use PyPy

Ideal Use Cases

- Long-running applications
- CPU-bound workloads
- Pure Python codebases
- Scientific computing (with NumPy)
- Web servers and backend services

Challenges & Limitations

- C extension compatibility issues
- Slower startup time
- Increased memory usage initially
- Warm-up period before optimizations
- Version lag behind CPython

PyPy shines in scenarios where the same code paths are executed repeatedly, allowing the JIT compiler to optimize effectively. For short-running scripts or C extension-heavy applications, the benefits may be limited or negative.

Implementing PyPy in Your Project

Installation

```
# Download from pypy.org or use:
apt-get install pypy3  # Debian/Ubuntu
brew install pypy3     # macOS
```

Usage

```
# Instead of:
python3 your_script.py

# Simply use:
pypy3 your_script.py
```

Dependencies

```
# Create a PyPy virtual environment
pypy3 -m venv pypy_env
source pypy_env/bin/activate

# Install compatible packages
pip install numpy  # PyPy-compatible version
```

Most pure Python packages work seamlessly. For C extensions, check the **compatibility list** or use alternatives.

PyPy Integration: Real-world Example

```
# benchmark.py
import time
import sys

def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

start = time.time()
result = fibonacci(35)
end = time.time()

print(f"Result: {result}")
print(f"Time taken: {end - start:.2f} seconds")
print(f"Interpreter: {sys.implementation.name}")
```

Running this with both interpreters:

```
$ python3 benchmark.py
Result: 9227465
Time taken: 5.83 seconds
Interpreter: cpython

$ pypy3 benchmark.py
Result: 9227465
Time taken: 0.32 seconds
Interpreter: pypy
```

This recursive implementation sees approximately 18x speedup with PyPy with zero code changes.

Numba: Targeted JIT Compilation

What is Numba?

Numba is a JIT compiler that translates a subset of Python and NumPy code into highly optimized machine code at runtime using LLVM.

Unlike PyPy, Numba works at the function level rather than the whole program, allowing targeted optimization.

Key Features

- Simple decorator-based API
- Compatible with CPython ecosystem
- GPU acceleration support
- Parallel processing capabilities

Numba excels at numerical algorithms and works particularly well with NumPy-based code, offering near-C speeds for properly annotated functions.

Numba in Action

```
# Standard installation: pip install numba

import numpy as np
from numba import jit
import time

# Function without Numba optimization
def standard_function(x, y):
    result = np.zeros_like(x)
    for i in range(len(x)):
        result[i] = x[i] + y[i]
        for j in range(20):  # Artificial work
            result[i] = result[i] * 0.5 + np.sin(result[i])
    return result

# Same function with Numba JIT decorator
@jit(nopython=True)  # 'nopython' mode for best performance
def numba_function(x, y):
    result = np.zeros_like(x)
    for i in range(len(x)):
        result[i] = x[i] + y[i]
        for j in range(20):  # Artificial work
            result[i] = result[i] * 0.5 + np.sin(result[i])
    return result
```

Simply adding the @jit decorator can yield 10-100x speedups for numerical code, especially with loops and mathematical operations.

Numba Performance Benchmark

```
# Continuing from previous example
# Set up test data
data_size = 10_000_000
x = np.random.random(data_size)
y = np.random.random(data_size)

# Benchmark standard function
start = time.time()
result1 = standard_function(x, y)
standard_time = time.time() - start
print(f"Standard function: {standard_time:.4f} seconds")

# Benchmark Numba function (includes compilation time)
start = time.time()
result2 = numba_function(x, y)
first_numba_time = time.time() - start
print(f"Numba function (first run): {first_numba_time:.4f} seconds")

# Benchmark Numba function (pre-compiled)
start = time.time()
result3 = numba_function(x, y)
second_numba_time = time.time() - start
print(f"Numba function (second run): {second_numba_time:.4f} seconds")
```

Typical output might show the standard function taking 15+ seconds, the first Numba run taking 1-2 seconds (compilation + execution), and subsequent runs taking just 0.1-0.2 seconds (50-100x speedup).

Advanced Numba Techniques

Parallel Processing

```
@jit(nopython=True, parallel=True)
def parallel_sum(a):
    sum = 0.0
    # Automatic parallelization of this loop
    for i in prange(len(a)):
        sum += a[i]
    return sum
```

The 'parallel' option and 'prange' enable multi-core execution, effectively bypassing GIL limitations.

GPU Acceleration

```
from numba import cuda

@cuda.jit
def increment_by_one(an_array):
    # Thread positioning
    i = cuda.grid(1)
    if i < an_array.shape[0]:
        an_array[i] += 1
```

CUDA support allows offloading computation to NVIDIA GPUs for massive parallelism.

These advanced features enable Python code to leverage modern hardware architectures without resorting to C/C++ extensions.

When to Use Numba

Ideal Use Cases

- Numerical algorithms and simulations
- Array-oriented computing
- Signal and image processing
- Machine learning model inference
- Scientific computing and research

Limitations

- Limited Python feature support
- Not all NumPy functions supported
- Cannot JIT arbitrary Python objects
- Debugging can be challenging
- Compilation overhead for small workloads

Numba works best when optimizing computationally intensive, array-based algorithms. It's less effective for I/O-bound operations, string processing, or code heavily dependent on Python's dynamic features.

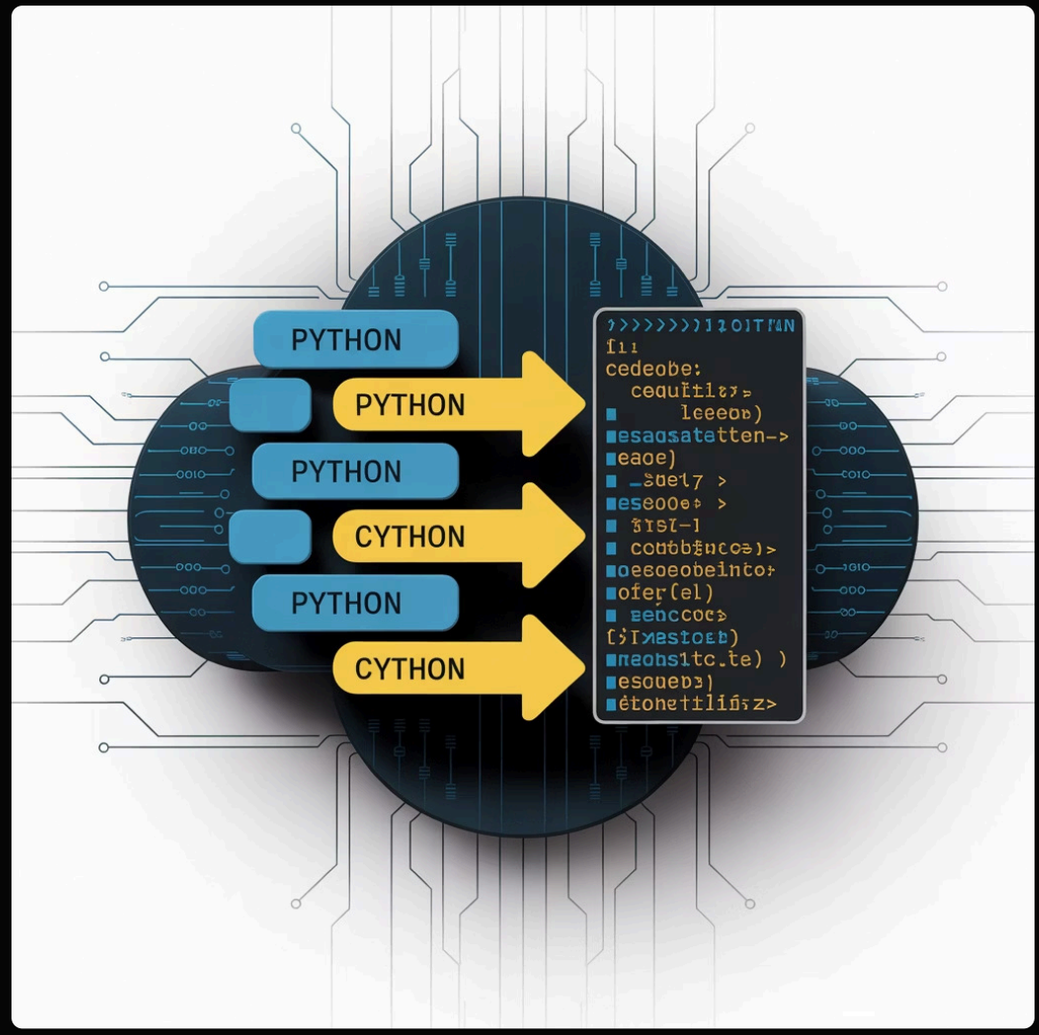
Cython: Bridging Python and C

What is Cython?

Cython is both a language extension of Python and a compiler that translates Python-like code to efficient C or C++ code, which is then compiled to a native extension module.

Key Advantages

- Static typing for massive performance gains
- Direct access to C/C++ libraries
- Ability to release the GIL for true parallelism
- Incremental optimization approach



Cython represents a middle ground between Python's ease of use and C's performance, allowing gradual optimization of critical code paths.

Cython Basics

```
# Pure Python version (slow)
def fibonacci_py(n):
    if n <= 1:
        return n
    return fibonacci_py(n-1) + fibonacci_py(n-2)

# Cython version (fibonacci.pyx)
def fibonacci_cy(int n):  # Type declaration
    if n <= 1:
        return n
    return fibonacci_cy(n-1) + fibonacci_cy(n-2)

# Even faster Cython version
cpdef int fibonacci_cy_opt(int n):  # More C-like
    cdef int a = 0, b = 1, i, temp
    if n <= 0:
        return 0
    for i in range(n-1):
        temp = a + b
        a = b
        b = temp
    return b
```

Adding type declarations with Cython can yield 20-100x speedups for numerical code. Using C-like algorithms pushes performance even further, often 100-1000x faster than pure Python.

Building and Using Cython Extensions

Setup Script (setup.py)

```
from setuptools import setup
from Cython.Build import cythonize

setup(
    name="fibonacci_module",
    ext_modules=cythonize("fibonacci.pyx"),
)
```

Build the extension with:

```
python setup.py build_ext --inplace
```

Using the Extension

```
# Import and use like a normal Python module
import fibonacci

# Compare performance
import time

start = time.time()
fibonacci.fibonacci_cy(35)
cy_time = time.time() - start
print(f"Cython recursive: {cy_time:.4f}s")

start = time.time()
fibonacci.fibonacci_cy_opt(35)
cy_opt_time = time.time() - start
print(f"Cython optimized: {cy_opt_time:.6f}s")
```

Advanced Cython Features

Releasing the GIL

```
cpdef double compute_sum(double[:] arr) nogil:
    cdef int i
    cdef double total = 0.0
    for i in range(arr.shape[0]):
        total += arr[i]
    return total

# Enables true parallel execution with threads
```

Memory Views

```
cpdef process_image(unsigned char[:, :, :] img):
    cdef int height = img.shape[0]
    cdef int width = img.shape[1]
    cdef int channels = img.shape[2]
    cdef int i, j, k

    # Efficient in-place modification
    for i in range(height):
        for j in range(width):
            for k in range(channels):
                img[i, j, k] = 255 - img[i, j, k]
```

These advanced features enable Cython to match or exceed the performance of hand-written C code while maintaining much of Python's readability.

When to Use Cython

Ideal Use Cases

- Performance-critical libraries
- CPU-bound numerical computations
- Wrapping external C/C++ libraries
- Mixed Python/C codebases
- Maximum control over memory usage

Limitations

- Steeper learning curve than Numba
- Requires separate build process
- Platform-specific compilation
- Debugging more complex
- Type declarations can be verbose

Cython is best for projects where maximum performance is critical, especially those needing fine-grained control over memory and C interoperability. For simpler optimizations, Numba may offer a better effort-to-reward ratio.

Alternative Performance Options

Pythran

An ahead-of-time compiler that converts annotated Python modules to C++.

- Focuses on scientific computing
- NumPy-aware optimizations
- SIMD vectorization support
- Simple annotation-based interface

Nuitka

A source-to-source compiler that translates Python code to C, aiming for complete compatibility.

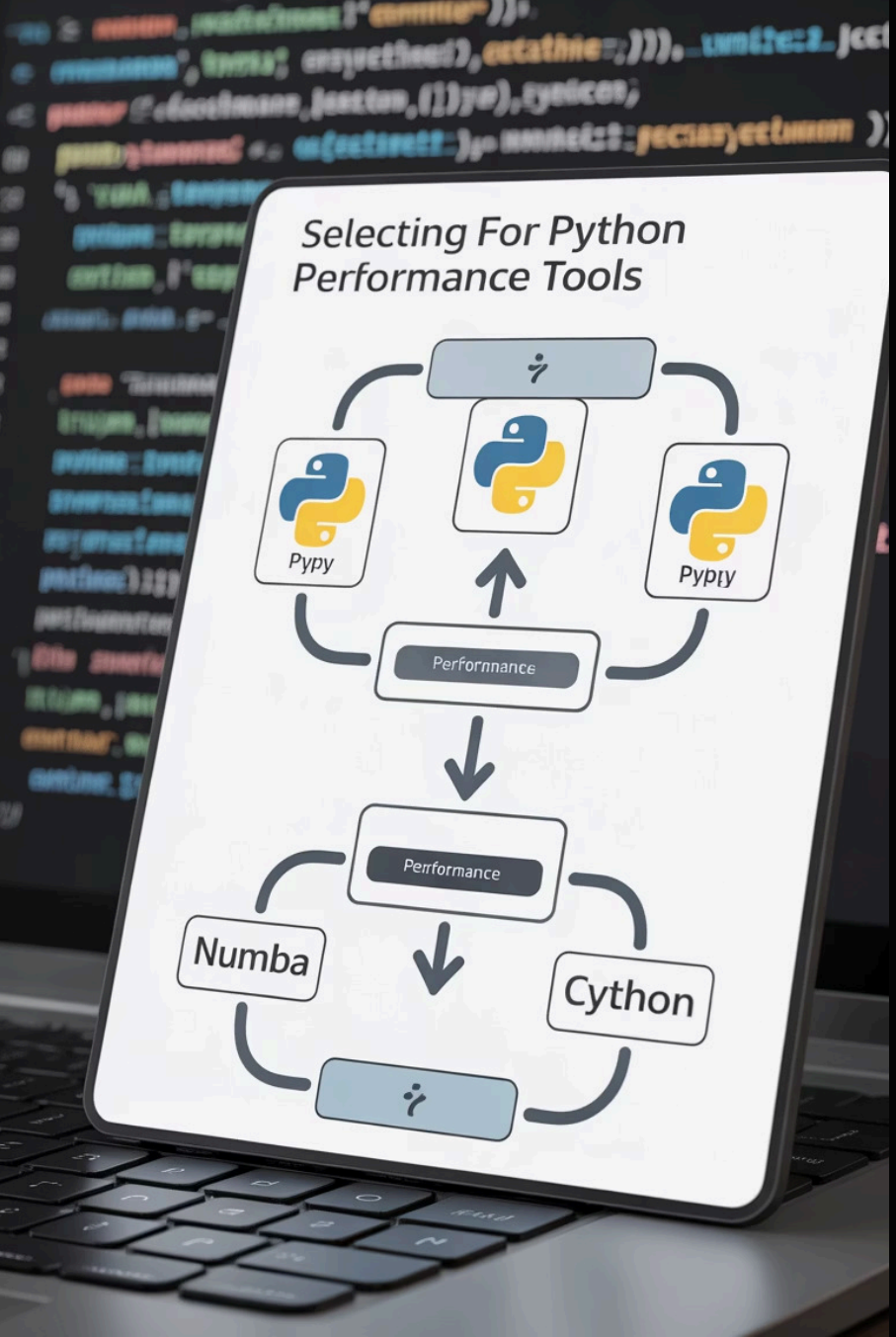
- Compiles entire applications
- Can create standalone executables
- Works with most Python libraries
- Gradual performance improvements

These alternatives offer different trade-offs in terms of compatibility, ease of use, and performance gains. They're worth exploring for specific use cases where PyPy, Numba, or Cython might not be ideal.

Performance Tools Comparison

Tool	Speed Gain	Ease of Use	Compatibility	Best For
PyPy	3-10x	Very Easy	Moderate	Whole applications, long-running processes
Numba	10-100x	Easy	Good	Numerical functions, NumPy-heavy code
Cython	20-1000x	Moderate	Excellent	Critical paths, C integration, maximum control
Pythran	10-100x	Moderate	Limited	Scientific computing, array processing
Nuitka	1.5-5x	Easy	Excellent	Distribution, modest performance needs

The optimal choice depends on your specific use case, performance requirements, and development constraints. Often, a hybrid approach using different tools for different components yields the best results.



Choosing the Right Performance Tool

Follow this decision tree to determine which performance optimization approach best suits your specific needs. Consider factors like development time, required speed improvements, maintenance concerns, and integration requirements with existing systems.

Remember that different parts of your application may benefit from different optimization strategies—there's no one-size-fits-all solution.

Performance Optimization Strategy

1. Profile First

Use tools like cProfile, line_profiler, or py-spy to identify actual bottlenecks before optimizing.

```
python -m cProfile -o profile.stats your_script.py
```

2. Algorithmic Improvements

Optimize algorithms and data structures first—often yields greater gains than compiler optimizations.

3. Targeted Optimization

Apply PyPy, Numba, or Cython only to the performance-critical parts identified by profiling.

4. Measure and Iterate

Continuously benchmark to ensure optimizations are effective and don't introduce regressions.

Following this methodical approach avoids premature optimization and ensures your efforts yield maximum benefit for the time invested.

Python's Concurrency Challenge: The GIL

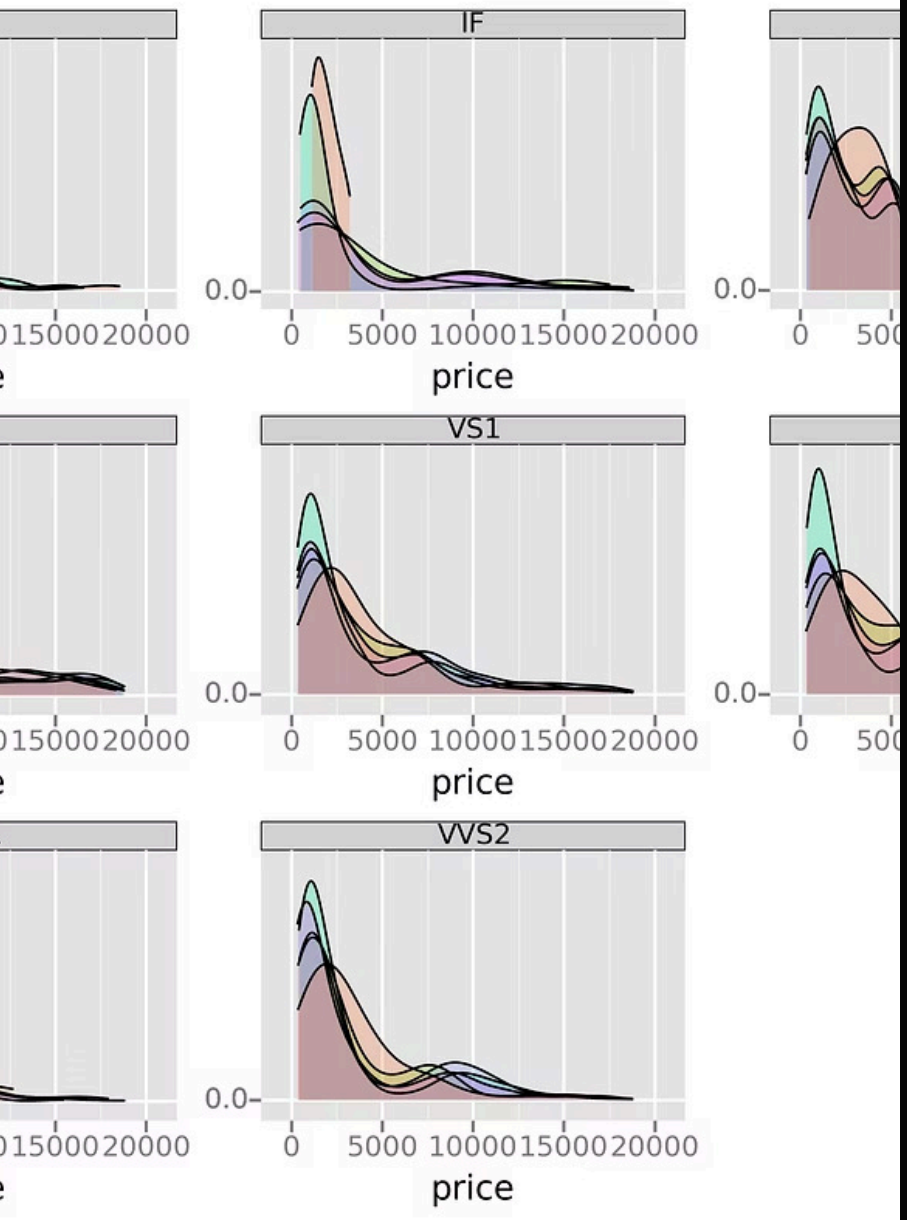
What is the Global Interpreter Lock?

The GIL is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecode simultaneously in a single process.

Why Does It Exist?

- Simplifies memory management
- Ensures thread safety for non-thread-safe C extensions
- Avoids race conditions in the interpreter

The GIL means that Python threads cannot achieve true CPU parallelism on multiple cores—a significant limitation for CPU-bound workloads.



GIL Impact Visualization

The diagram illustrates how the GIL prevents full utilization of multiple CPU cores. Even with many Python threads, CPU-bound operations remain largely serialized due to the GIL, resulting in suboptimal performance scaling.

This limitation is especially pronounced on modern machines with many cores, where only a fraction of the available computing power can be leveraged by a single Python process.

Threads vs Processes in Python

Threads (threading module)

- Share memory space
- Lightweight resource usage
- Quick to create and destroy
- Limited by the GIL for CPU tasks
- Good for I/O-bound workloads

Processes (multiprocessing module)

- Separate memory spaces
- Higher resource overhead
- Slower creation and communication
- Unaffected by the GIL
- Good for CPU-bound workloads

Understanding this fundamental difference is crucial for designing efficient concurrent applications in Python. The right choice depends on your workload characteristics and performance requirements.

When to Use Threads vs Processes

Use Threads When

- Workload is I/O-bound (network, disk, etc.)
- Need to share in-memory data efficiently
- Creating many concurrent tasks (100s+)
- Memory footprint is a concern

Use Processes When

- Workload is CPU-bound (computations)
- Need to leverage multiple CPU cores
- Isolation between tasks is important
- Robustness against crashes is required

Many real-world applications use a hybrid approach: processes for CPU-intensive calculations across cores, with threads handling I/O operations within each process.

Threads and I/O-Bound Tasks

```
import threading
import time
import requests
import concurrent.futures

urls = [
    "https://www.python.org",
    "https://www.github.com",
    "https://www.stackoverflow.com",
    "https://www.wikipedia.org",
    "https://www.bbc.co.uk",
    # ... many more URLs
]

def fetch_url(url):
    """Fetch a URL and return its content length"""
    response = requests.get(url)
    return url, len(response.content)

# Sequential execution
start = time.time()
results = [fetch_url(url) for url in urls]
print(f"Sequential: {time.time() - start:.2f} seconds")

# Threaded execution
start = time.time()
with concurrent.futures.ThreadPoolExecutor(max_workers=10) as executor:
    results = list(executor.map(fetch_url, urls))
print(f"Threaded: {time.time() - start:.2f} seconds")
```

For I/O-bound tasks like network requests, threads can provide significant speedups (often 5-10x) despite the GIL, as threads yield during I/O operations.

Processes and CPU-Bound Tasks

```
import time
import concurrent.futures
import numpy as np

def compute_intensive_task(size):
    """Perform a CPU-intensive calculation"""
    # Create large matrices and multiply them
    matrix_a = np.random.random((size, size))
    matrix_b = np.random.random((size, size))
    return np.matmul(matrix_a, matrix_b).sum()

sizes = [1000] * 8  # 8 large matrix multiplications

# Sequential execution
start = time.time()
results = [compute_intensive_task(size) for size in sizes]
print(f"Sequential: {time.time() - start:.2f} seconds")

# Threaded execution (limited by GIL)
start = time.time()
with concurrent.futures.ThreadPoolExecutor(max_workers=8) as executor:
    results = list(executor.map(compute_intensive_task, sizes))
print(f"Threaded: {time.time() - start:.2f} seconds")

# Process-based execution (bypasses GIL)
start = time.time()
with concurrent.futures.ProcessPoolExecutor(max_workers=8) as executor:
    results = list(executor.map(compute_intensive_task, sizes))
print(f"Multiprocessing: {time.time() - start:.2f} seconds")
```

For CPU-bound tasks, processes can provide near-linear speedup with the number of cores, while threads show minimal improvement due to the GIL.

Python 3.12: Per-Interpreter GIL

What's Changing?

Python 3.12 introduces a significant enhancement to concurrency with support for multiple interpreters per process, each with its own GIL.

Key Benefits

- Multiple GILs in a single process
- True parallelism without process overhead
- Shared memory where needed
- Better integration with embedding applications

This represents a middle ground between the shared-memory efficiency of threads and the parallelism of processes, potentially offering the best of both worlds.

Per-Interpreter GIL: How It Works

```
# Python 3.12+ code (simplified example of the new API)
import interpreters
import _xxsubinterpreters as subinterpreters

# Create a new sub-interpreter with its own GIL
interp_id = subinterpreters.create()

# Function to run in the sub-interpreter
def heavy_computation(x):
    result = 0
    for i in range(10000000):
        result += x * i
    return result

# Run the function in the sub-interpreter
channel_id = interpreters.create_channel()
subinterpreters.run_string(
    interp_id,
    f"""
    import interpreters
    result = {heavy_computation}(42)
    interpreters.channel_send({channel_id}, result)
    """
)

# Get the result back
result = interpreters.channel_recv(channel_id)
print(f"Result: {result}")
```

While the API is still evolving, this demonstrates how multiple interpreters can execute CPU-bound code in parallel within a single process.

Circumventing the GIL

Using PyPy's STM (Software Transactional Memory)

PyPy has an experimental STM implementation that allows multiple threads to execute Python code truly in parallel.

```
# Run with: pypy-stm
# Parallelism happens automatically
```

Still experimental but promising for thread-heavy workloads.

Numba's nogil Mode

Numba can release the GIL for decorated functions, enabling true parallelism.

```
@numba.jit(nogil=True)
def parallel_function(data):
    # This runs without the GIL
    result = 0
    for x in data:
        result += compute(x)
    return result
```

These approaches offer ways to achieve true parallelism with threads in Python, though each comes with its own limitations and trade-offs.

Cython and the GIL

```
# cython: boundscheck=False
# cython: wraparound=False

import numpy as np
cimport numpy as np
from cython.parallel import prange

def parallel_sum(double[:] array):
    cdef double total = 0.0
    cdef int i, n = array.shape[0]

    # Release the GIL for this section
    with nogil:
        # Parallel execution across cores
        for i in prange(n, schedule='static'):
            total += array[i]

    return total
```

Cython's `nogil` context manager and `prange` function allow you to release the GIL and execute code in parallel across multiple cores. This enables true parallelism for computationally intensive sections while maintaining Python compatibility for the rest of your code.

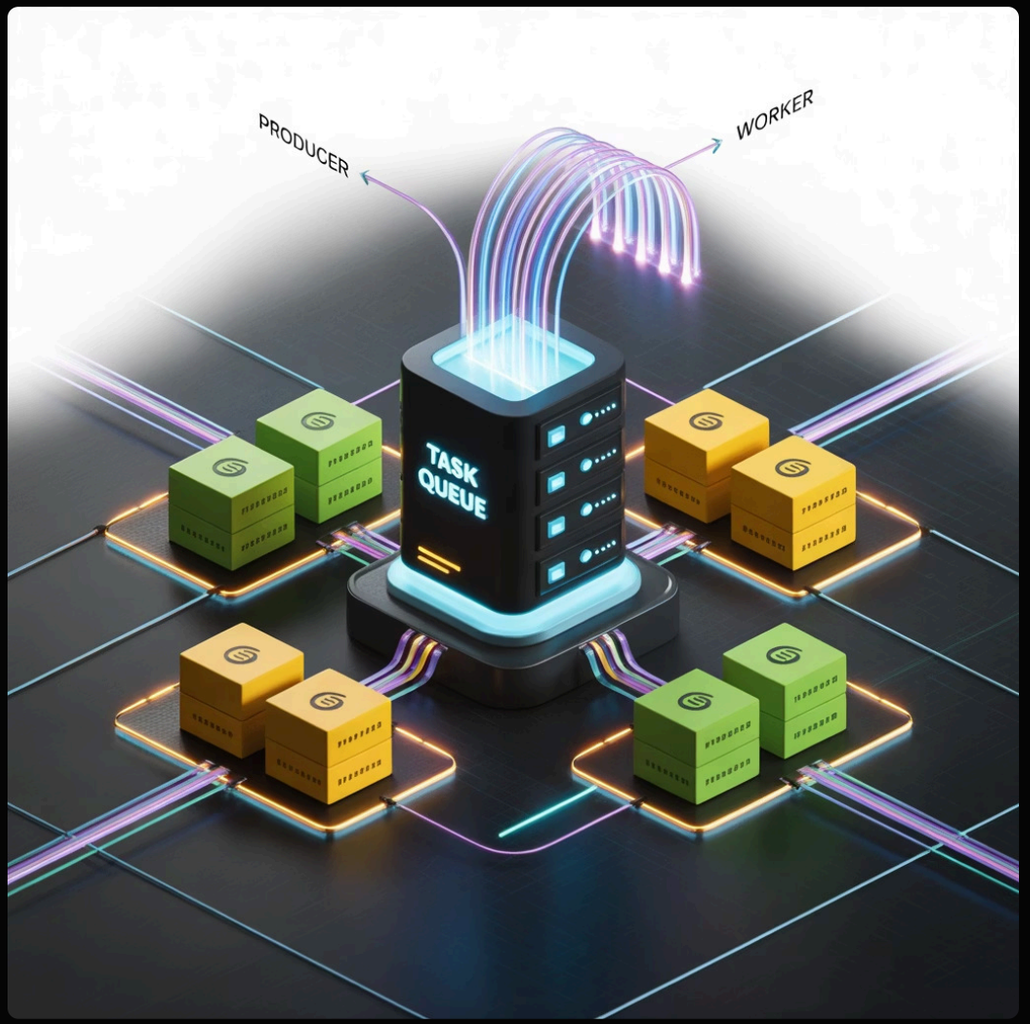
Horizontal Scaling with Task Queues

What are Task Queues?

Task queues provide a way to distribute work across multiple machines, enabling horizontal scaling beyond what's possible with a single Python process.

Popular Python Task Queue Systems

- Celery
- RQ (Redis Queue)
- Dramatiq
- Apache Airflow



Task queues effectively bypass GIL limitations by distributing work across multiple processes or even machines, each with their own Python interpreter.

Celery: Distributed Task Queue

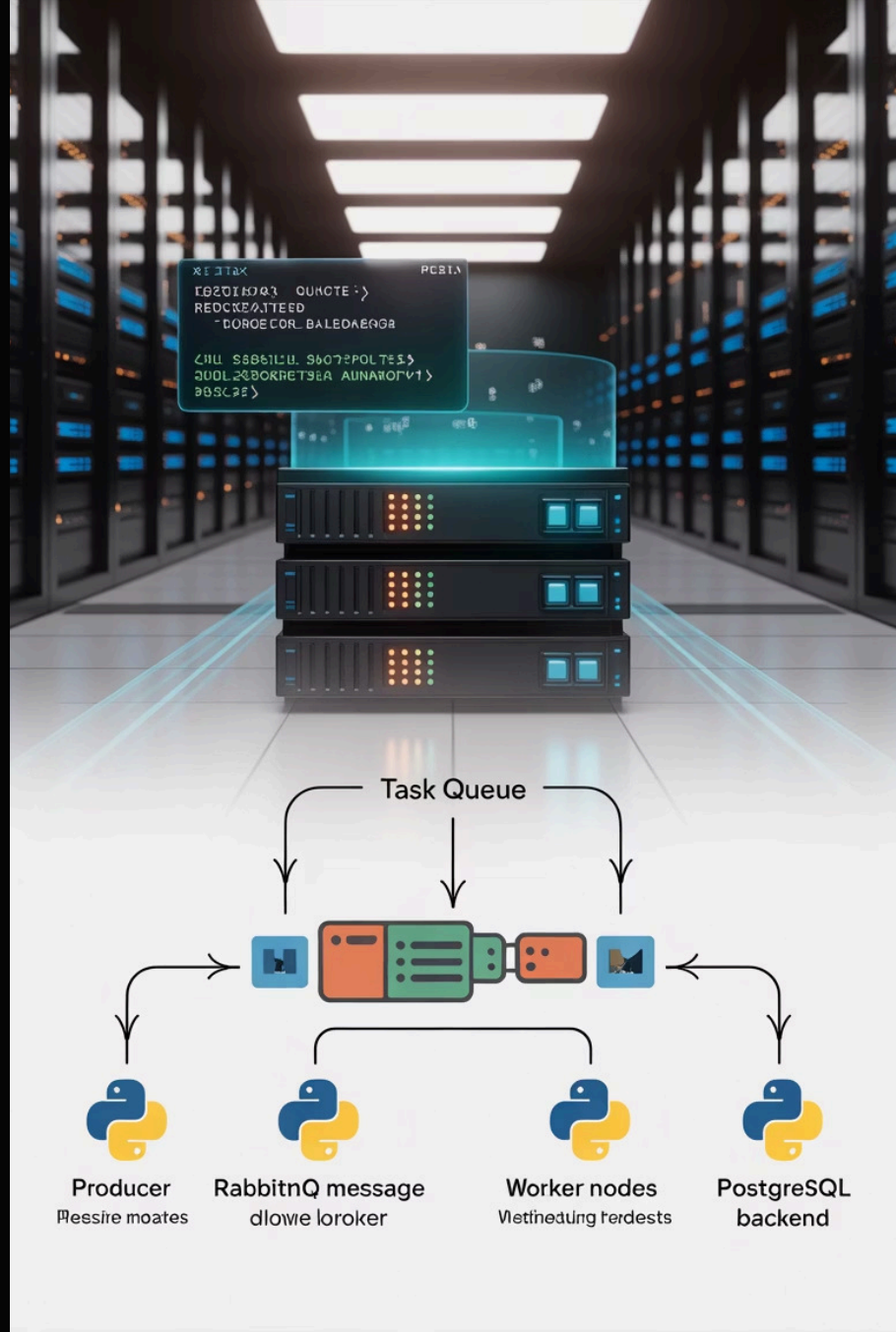
```
# Installation: pip install celery redis

# tasks.py
from celery import Celery

# Create Celery instance with Redis as broker
app = Celery('tasks', broker='redis://localhost:6379/0')

@app.task
def compute_factorial(n):
    """Compute factorial of n"""
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result

# In a separate process/terminal
# celery -A tasks worker --loglevel=info

# In your application
from tasks import compute_factorial

# Asynchronous execution
result = compute_factorial.delay(100000)

# Check if ready and get result
if result.ready():
    print(f"Result: {result.get()}")
```

Celery enables distributing tasks across multiple worker processes or machines, effectively circumventing GIL limitations by leveraging multiple Python interpreters.

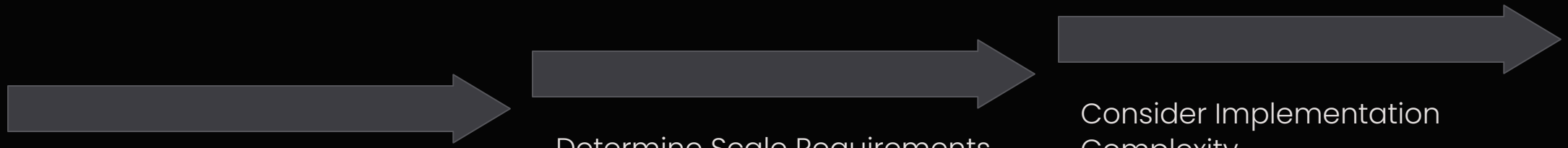
Task Queue Architecture

The distributed architecture of task queues enables horizontal scaling across multiple cores, servers, or even data centers. Tasks are pushed to a message broker, picked up by available workers, processed independently, and results stored in a backend for retrieval.

This approach effectively bypasses GIL limitations by distributing work across multiple Python processes, each with its own interpreter and memory space.



Concurrency Strategy Decision Tree



Identify Workload Type

Is your bottleneck CPU-bound or I/O-bound?

- CPU-bound: calculations, processing
- I/O-bound: network, disk, database

Determine Scale Requirements

Do you need to scale within a machine or across machines?

- Single machine: processes or threads
- Multiple machines: task queues or microservices

Consider Implementation Complexity

Balance performance gains against development and maintenance costs

- Threads: simplest but GIL-limited
- Processes: more complex but bypass GIL
- Task queues: most complex but most scalable

The optimal concurrency strategy depends on your specific workload characteristics, performance requirements, and development constraints.

Decorators: Introduction

What are Decorators?

Decorators are a powerful and elegant way to modify or enhance functions or methods without changing their core implementation.

They follow the principle of Aspect-Oriented Programming (AOP) by separating cross-cutting concerns from business logic.

Key Concepts

- Functions are first-class objects
- Decorators are "wrapper" functions
- Applied using @decorator syntax

Decorators leverage Python's functional programming capabilities to enable clean, reusable code patterns for common operations like logging, timing, or access control.

Basic Decorator Syntax

```
# A simple decorator
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with {args}, {kwargs}")
        result = func(*args, **kwargs)
        print(f"{func.__name__} returned {result}")
        return result
    return wrapper

# Using the decorator
@my_decorator
def add(a, b):
    return a + b

# This is equivalent to:
# add = my_decorator(add)

# When we call the function, the decorator is applied
result = add(3, 5)
# Output:
# Calling add with (3, 5), {}
# add returned 8
```

The @decorator syntax is syntactic sugar for explicitly applying the decorator function to the target function. The wrapper function can perform actions before and/or after the original function executes.

Decorators with Parameters

```
# A decorator that takes parameters
def repeat(times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            results = []
            for _ in range(times):
                results.append(func(*args, **kwargs))
            return results
        return wrapper
    return decorator

# Using the parameterized decorator
@repeat(times=3)
def generate_random():
    import random
    return random.randint(1, 100)

# Calling the decorated function
results = generate_random()
print(results)  # Might output: [42, 17, 89]
```

To create a decorator that accepts its own parameters, we need three nested functions: the outer function captures the decorator parameters, the middle function is the actual decorator, and the innermost function is the wrapper.

Preserving Function Metadata

The Problem

```
def log_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

@log_decorator
def greet(name):
    """Return a greeting message."""
    return f"Hello, {name}!"

print(greet.__name__)  # Outputs: 'wrapper'
print(greet.__doc__)   # Outputs: None
```

The Solution: functools.wraps

```
from functools import wraps

def log_decorator(func):
    @wraps(func)  # Preserves metadata
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

@log_decorator
def greet(name):
    """Return a greeting message."""
    return f"Hello, {name}!"

print(greet.__name__)  # Outputs: 'greet'
print(greet.__doc__)   # Outputs: 'Return a greeting...'
```

Always use @wraps from the functools module to preserve function metadata like name, docstring, and signature, which is essential for debugging and introspection.

Class-based Decorators

```
class CountCalls:
    def __init__(self, func):
        self.func = func
        self.count = 0
        # Preserve metadata
        from functools import update_wrapper
        update_wrapper(self, func)

    def __call__(self, *args, **kwargs):
        self.count += 1
        print(f"{self.func.__name__} has been called {self.count} times")
        return self.func(*args, **kwargs)

@CountCalls
def fibonacci(n):
    """Calculate the Fibonacci number."""
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# Using the decorated function
result = fibonacci(5)
# The count will reflect all calls, including recursive ones
```

Class-based decorators implement the __call__ method to make instances callable like functions. They're useful when you need to maintain state across function calls, as shown with the call counter in this example.

Decorating Classes

```
def add_greeting(cls):
    """Add a greeting method to the decorated class."""
    def greet(self, name):
        return f"{self.__class__.__name__} says hello to {name}!"

    # Add the new method to the class
    cls.greet = greet
    return cls

@add_greeting
class Person:
    def __init__(self, name):
        self.name = name

# Using the decorated class
person = Person("Alice")
print(person.greet("Bob"))  # Outputs: "Person says hello to Bob!"
```

Class decorators can add or modify methods, attributes, or behaviors of a class. This pattern is powerful for implementing cross-cutting concerns across multiple classes, such as logging, validation, or registration.

Stacking Decorators

```
from functools import wraps
import time

def log_execution(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Executing {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

def measure_time(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.2f} seconds")
        return result
    return wrapper

# Applying multiple decorators (order matters!)
@measure_time
@log_execution
def complex_operation(n):
    time.sleep(n)  # Simulate work
    return n * n

# The execution order is inside-out:
# 1. log_execution runs first
# 2. measure_time runs around that
result = complex_operation(1.5)
```

Decorators can be stacked, with each one adding its own behavior. The execution order is from bottom to top (the decorator closest to the function executes first).

Practical Decorator: Retry Pattern

```
import time
from functools import wraps
import random

def retry(max_attempts=3, delay=1):
    """Retry a function call on exception with exponential backoff."""
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            attempts = 0
            while attempts < max_attempts:
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    attempts += 1
                    if attempts == max_attempts:
                        raise  # Re-raise the last exception

                    # Exponential backoff with jitter
                    sleep_time = delay * (2 ** (attempts - 1)) * (0.5 + random.random())
                    print(f"Attempt {attempts} failed: {e}. Retrying in {sleep_time:.2f}s")
                    time.sleep(sleep_time)
        return wrapper
    return decorator

@retry(max_attempts=5, delay=0.5)
def unstable_network_call(url):
    """Simulate an unstable network call that sometimes fails."""
    if random.random() < 0.7:  # 70% chance of failure
        raise ConnectionError("Network unavailable")
    return f"Success: {url} responded with data"
```

This retry decorator implements the exponential backoff pattern, essential for resilient network communication. It's a practical example of how decorators can encapsulate complex error-handling logic.

Practical Decorator: Memoization

```
from functools import wraps

def memoize(func):
    """Cache the results of the function to avoid redundant calculations."""
    cache = {}

    @wraps(func)
    def wrapper(*args, **kwargs):
        # Create a hash key from the arguments
        # For simplicity, we assume args are hashable
        key = str(args) + str(sorted(kwargs.items()))

        if key not in cache:
            cache[key] = func(*args, **kwargs)
            print(f"Cache miss for {func.__name__}{args}")
        else:
            print(f"Cache hit for {func.__name__}{args}")
        return cache[key]

    return wrapper

@memoize
def fibonacci(n):
    """Calculate the nth Fibonacci number recursively."""
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# First call computes the result
result = fibonacci(10)

# Second call uses the cached result
result = fibonacci(10)
```

Memoization caches function results based on input parameters, dramatically improving performance for expensive recursive functions or calculations that are called repeatedly with the same inputs.

Aspect-Oriented Programming (AOP)

What is AOP?

Aspect-Oriented Programming is a programming paradigm that separates cross-cutting concerns from business logic.

Key Concepts

- **Aspect:** A concern that cuts across multiple classes
- **Join Point:** A point in program execution
- **Advice:** Action taken at a join point
- **Pointcut:** A predicate for selecting join points

Python decorators provide a natural way to implement AOP concepts, allowing clean separation of concerns without frameworks.

Implementing AOP with Decorators

```
from functools import wraps
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# An aspect for logging method calls
def log_calls(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        logger.info(f"Calling {func.__name__} with {args}, {kwargs}")
        try:
            result = func(*args, **kwargs)
            logger.info(f"{func.__name__} returned {result}")
            return result
        except Exception as e:
            logger.error(f"{func.__name__} raised {e.__class__.__name__}: {e}")
            raise
    return wrapper

# An aspect for performance monitoring
def monitor_performance(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        import time
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        logger.info(f"{func.__name__} took {end - start:.4f} seconds")
        return result
    return wrapper
```

These decorators implement AOP aspects for logging and performance monitoring, which can be applied across many functions without modifying their implementation.

Publish-Subscribe Pattern with Decorators

```
class EventSystem:
    def __init__(self):
        self.subscribers = {}

    def subscribe(self, event_type, callback):
        if event_type not in self.subscribers:
            self.subscribers[event_type] = []
        self.subscribers[event_type].append(callback)

    def publish(self, event_type, *args, **kwargs):
        if event_type not in self.subscribers:
            return
        for callback in self.subscribers[event_type]:
            callback(*args, **kwargs)

# Create a global event system
event_system = EventSystem()

# Decorator to publish events
def publish_event(event_type):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            result = func(*args, **kwargs)
            event_system.publish(event_type, result, *args, **kwargs)
            return result
        return wrapper
    return decorator

# Decorator to subscribe to events
def subscribe_to(event_type):
    def decorator(func):
        event_system.subscribe(event_type, func)
        return func
    return decorator
```

This implementation demonstrates how decorators can create a publish-subscribe system, decoupling event producers from consumers.

Publish-Subscribe in Action

```
# Using our pub-sub decorators

# Producer
@publish_event("user_created")
def create_user(username, email):
    # Simulate user creation in database
    user_id = hash(username + email) % 10000
    print(f"User created with ID {user_id}")
    return {
        "id": user_id,
        "username": username,
        "email": email
    }

# Consumers
@subscribe_to("user_created")
def send_welcome_email(user_data, *args, **kwargs):
    print(f"Sending welcome email to {user_data['email']}")

@subscribe_to("user_created")
def setup_default_permissions(user_data, *args, **kwargs):
    print(f"Setting up default permissions for user {user_data['username']}")

@subscribe_to("user_created")
def log_user_creation(user_data, *args, **kwargs):
    print(f"AUDIT LOG: User {user_data['id']} created at {time.strftime('%Y-%m-%d %H:%M:%S')}")

# Creating a user will trigger all subscribers
create_user("john_doe", "john@example.com")
```

This example shows how the publish-subscribe pattern enables loose coupling between system components, allowing new subscribers to be added without modifying the producer code.

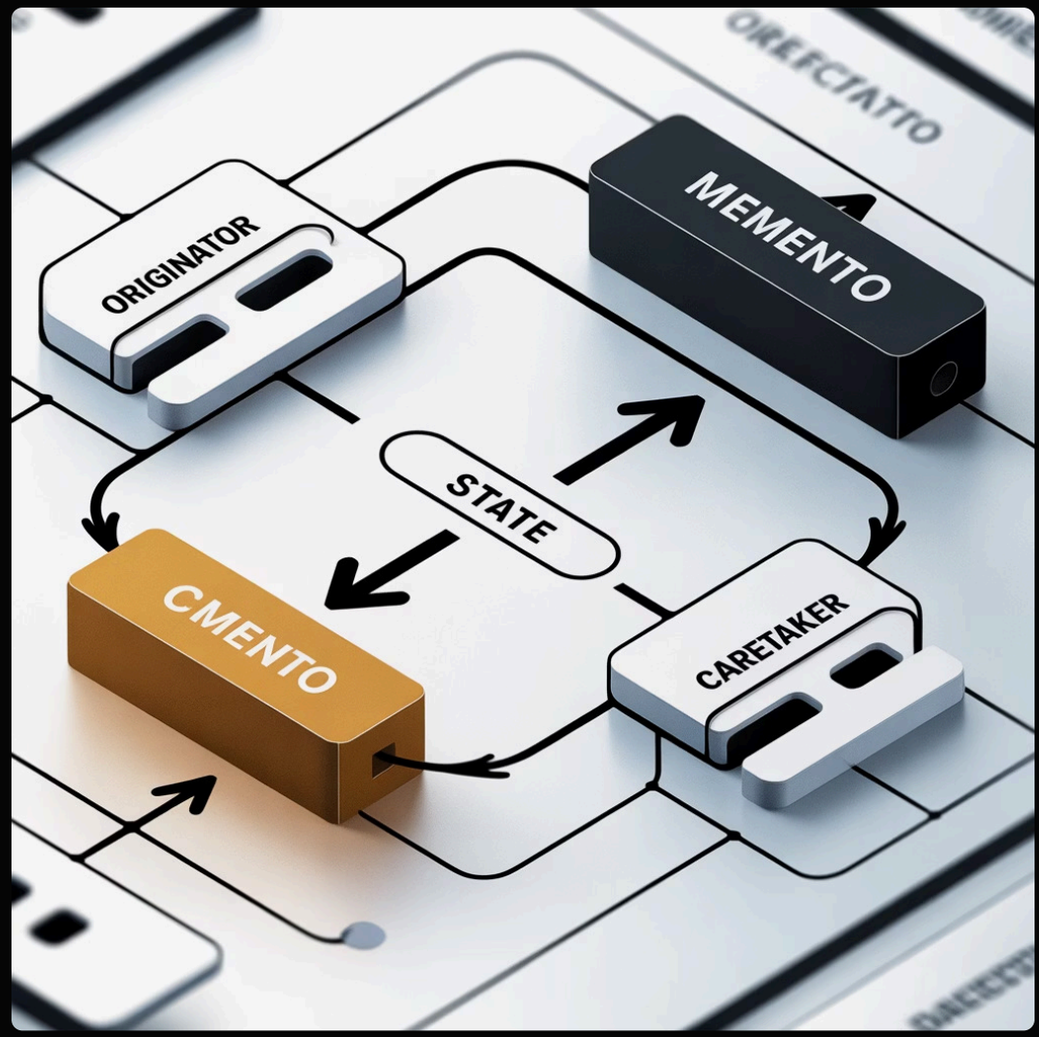
Memento Pattern with Decorators

What is the Memento Pattern?

The Memento pattern captures and externalizes an object's internal state, allowing the object to be restored to this state later without violating encapsulation.

Key Components

- **Originator:** The object whose state is saved
- **Memento:** The stored state snapshot
- **Caretaker:** Manages and protects mementos



Decorators can implement the Memento pattern to provide automatic state management for Python objects, enabling undo/redo functionality.

Implementing Memento with Decorators

```
import copy
from functools import wraps

class Memento:
    """Stores a snapshot of an object's state."""
    def __init__(self, state):
        self._state = copy.deepcopy(state)

    def get_state(self):
        return copy.deepcopy(self._state)

class Caretaker:
    """Manages history of mementos."""
    def __init__(self, max_history=10):
        self._history = []
        self._position = -1
        self._max_history = max_history

    def save_state(self, memento):
        # If we're not at the end of history, truncate the future
        if self._position < len(self._history) - 1:
            self._history = self._history[:self._position + 1]

        self._history.append(memento)
        self._position = len(self._history) - 1

        # Limit history size
        if len(self._history) > self._max_history:
            self._history.pop(0)
            self._position -= 1

    def undo(self):
        if self._position > 0:
            self._position -= 1
            return self._history[self._position]
        return None

    def redo(self):
        if self._position < len(self._history) - 1:
            self._position += 1
            return self._history[self._position]
        return None
```

These classes form the foundation of the Memento pattern, providing a way to capture, store, and restore object states.

Memento Pattern: Decorator Implementation

```
def undoable(state_attributes):
    """Decorator to make a class's methods undoable."""
    def class_decorator(cls):
        # Add caretaker to the class
        original_init = cls.__init__

        def new_init(self, *args, **kwargs):
            original_init(self, *args, **kwargs)
            self._caretaker = Caretaker()
            # Save initial state
            self._save_state()

        cls.__init__ = new_init

        # Add methods for state management
        def _save_state(self):
            state = {}
            for attr in state_attributes:
                state[attr] = getattr(self, attr)
            memento = Memento(state)
            self._caretaker.save_state(memento)

        def _restore_state(self, memento):
            if memento:
                state = memento.get_state()
                for attr, value in state.items():
                    setattr(self, attr, value)

        def undo(self):
            memento = self._caretaker.undo()
            self._restore_state(memento)
            return memento is not None

        def redo(self):
            memento = self._caretaker.redo()
            self._restore_state(memento)
            return memento is not None

        # Add methods to the class
        cls._save_state = _save_state
        cls._restore_state = _restore_state
        cls.undo = undo
        cls.redo = redo

        # Decorate all public methods to save state after execution
        for name, method in list(cls.__dict__.items()):
            if callable(method) and not name.startswith('_') and name not in ['undo', 'redo']:
                @wraps(method)
                def wrapped_method(self, *args, method=method, **kwargs):
                    result = method(self, *args, **kwargs)
                    self._save_state()
                    return result

                setattr(cls, name, wrapped_method)

        return cls

    return class_decorator
```

This class decorator automatically adds undo/redo capability to any class by tracking changes to specified attributes.

Memento Pattern: Usage Example

```
@undoable(['text', 'cursor_position'])
class TextEditor:
    def __init__(self):
        self.text = ""
        self.cursor_position = 0

    def insert(self, text):
        """Insert text at the current cursor position."""
        before = self.text[:self.cursor_position]
        after = self.text[self.cursor_position:]
        self.text = before + text + after
        self.cursor_position += len(text)

    def delete(self, count=1):
        """Delete 'count' characters before the cursor."""
        if self.cursor_position > 0:
            before = self.text[:max(0, self.cursor_position - count)]
            after = self.text[self.cursor_position:]
            deleted_count = self.cursor_position - len(before)
            self.text = before + after
            self.cursor_position -= deleted_count

    def move_cursor(self, position):
        """Move the cursor to the specified position."""
        self.cursor_position = max(0, min(position, len(self.text)))

    def __str__(self):
        return f"Text: '{self.text}', Cursor: {self.cursor_position}"
```

The @undoable decorator automatically adds undo/redo functionality to this TextEditor class, tracking changes to the text and cursor position without cluttering the business logic.

Memento Pattern: Demo

```
# Create a text editor
editor = TextEditor()

# Make some changes
editor.insert("Hello")
print(editor)  # Text: 'Hello', Cursor: 5

editor.insert(" world")
print(editor)  # Text: 'Hello world', Cursor: 11

editor.move_cursor(5)
editor.insert("beautiful ")
print(editor)  # Text: 'Hello beautiful world', Cursor: 15

# Undo the last change
editor.undo()
print(editor)  # Text: 'Hello world', Cursor: 11

# Undo again
editor.undo()
print(editor)  # Text: 'Hello', Cursor: 5

# Redo
editor.redo()
print(editor)  # Text: 'Hello world', Cursor: 11

# Make a new change (this will clear the redo history)
editor.delete(6)
print(editor)  # Text: 'Hello', Cursor: 5

# Try to redo (should fail because we made a new change)
success = editor.redo()
print(f"Redo successful: {success}")  # Redo successful: False
print(editor)  # Text: 'Hello', Cursor: 5
```

This demonstration shows how the Memento pattern enables undo/redo functionality in a TextEditor, with all the state management handled by our decorator.

Built-in Decorators: @property

```
class Temperature:
    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        """Get the temperature in Celsius."""
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Temperature below absolute zero is not possible")
        self._celsius = value

    @property
    def fahrenheit(self):
        """Get the temperature in Fahrenheit."""
        return self._celsius * 9/5 + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        self.celsius = (value - 32) * 5/9

    @property
    def kelvin(self):
        """Get the temperature in Kelvin."""
        return self._celsius + 273.15

    @kelvin.setter
    def kelvin(self, value):
        self.celsius = value - 273.15
```

The @property decorator transforms methods into attribute-like accessors, enabling encapsulation, validation, and computed properties. This example demonstrates property getters, setters, and derived properties.

Built-in Decorators: @classmethod and @staticmethod

@classmethod

```
class Date:
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year

    @classmethod
    def from_string(cls, date_string):
        day, month, year = map(int, date_string.split('-'))
        return cls(day, month, year)

    @classmethod
    def today(cls):
        import datetime
        now = datetime.datetime.now()
        return cls(now.day, now.month, now.year)
```

Class methods receive the class as their first argument (cls) and can create and return instances of the class.

@staticmethod

```
class MathOperations:
    @staticmethod
    def is_prime(n):
        """Check if a number is prime."""
        if n <= 1:
            return False
        if n <= 3:
            return True
        if n % 2 == 0 or n % 3 == 0:
            return False
        i = 5
        while i * i <= n:
            if n % i == 0 or n % (i + 2) == 0:
                return False
            i += 6
        return True

    @staticmethod
    def gcd(a, b):
        """Calculate greatest common divisor."""
        while b:
            a, b = b, a % b
        return a
```

Static methods don't receive a special first argument and behave like regular functions that happen to be in a class's namespace.

Decorator Best Practices

Do

- Use @functools.wraps to preserve metadata
- Keep decorators focused on one responsibility
- Document how decorators modify behavior
- Consider performance implications for frequently called functions
- Use class-based decorators for maintaining state

Don't

- Create decorators that radically change function behavior
- Overuse decorators for simple operations
- Nest too many decorators (>3) on a single function
- Modify function arguments without clear documentation
- Write decorators with side effects that could surprise users

Following these best practices ensures that decorators enhance your code's clarity and maintainability rather than obfuscating it.

Key Takeaways

Performance Optimization

Use PyPy for drop-in performance gains, Numba for numerical computing, and Cython for maximum speed and C integration. Always profile first to target optimizations effectively.

Concurrency Management

Understand the GIL's limitations and choose the right tool: threads for I/O-bound work, processes for CPU-bound work, and task queues for horizontal scaling.

Decorators & AOP

Use decorators to implement cross-cutting concerns and design patterns. They enable clean separation of business logic from infrastructure concerns like logging and error handling.

Contact: Chandrashekar Babu <training@chandrashekar.info> for further inquiries or custom training sessions. Thank you for participating!