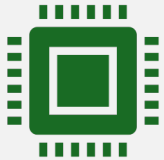
Advanced Python

Surendra Panpaliya

Week1 (Mon, Tue, Thurs)

Day 1: Python Recap + Environment & Tooling



Day 2: Functional Programming & Object-Oriented Design

Day 3: Advanced Python Concepts

Week2 (Mon, Tue, Wed, Thurs)

Day 4: Concurrency and Async Programming

Day 5: Web Services with FastAPI

Day 6: Azure Functions & Cloud Deployment

Day 7: Testing, Linting & Final Project

Day 4: Concurrency and Async Programming

Concurrency: threading vs multiprocessing

Async IO: asyncio, event loop, aiohttp

Profiling Tools: cProfile,

line_profiler, memory_profiler

Day 4: Concurrency and Async Programming

Hands-On Lab:

Build async scraper with aiohttp

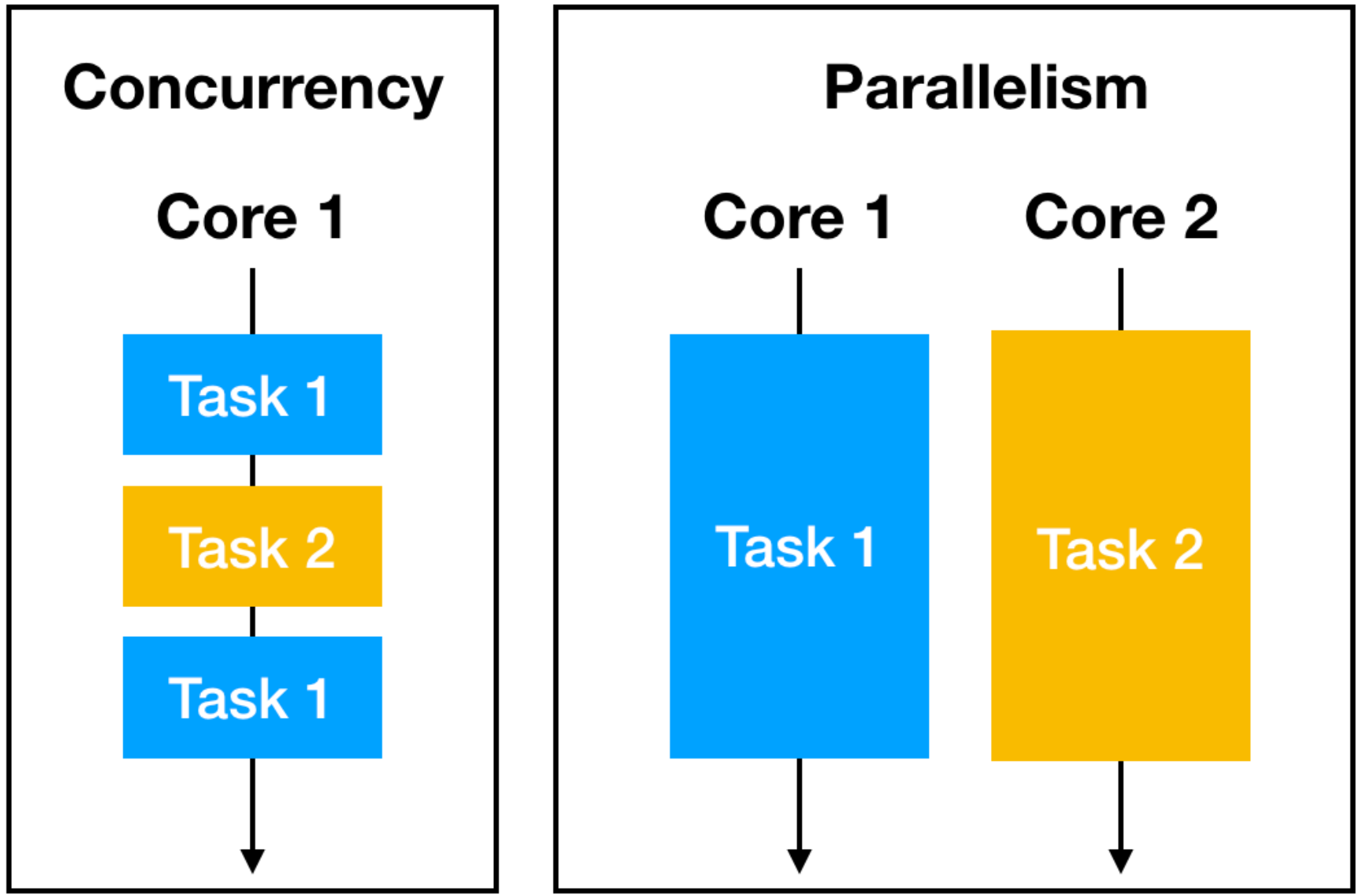
C# async/await vs Python async/await

What is Concurrency?

MULTIPLE TASKS

ARE IN PROGRESS

AT THE SAME TIME.


Concurrency
Core 1
Task 1
Task 2
Task 1
Parallelism
Core 1
Core 2
Task 1
Task 2

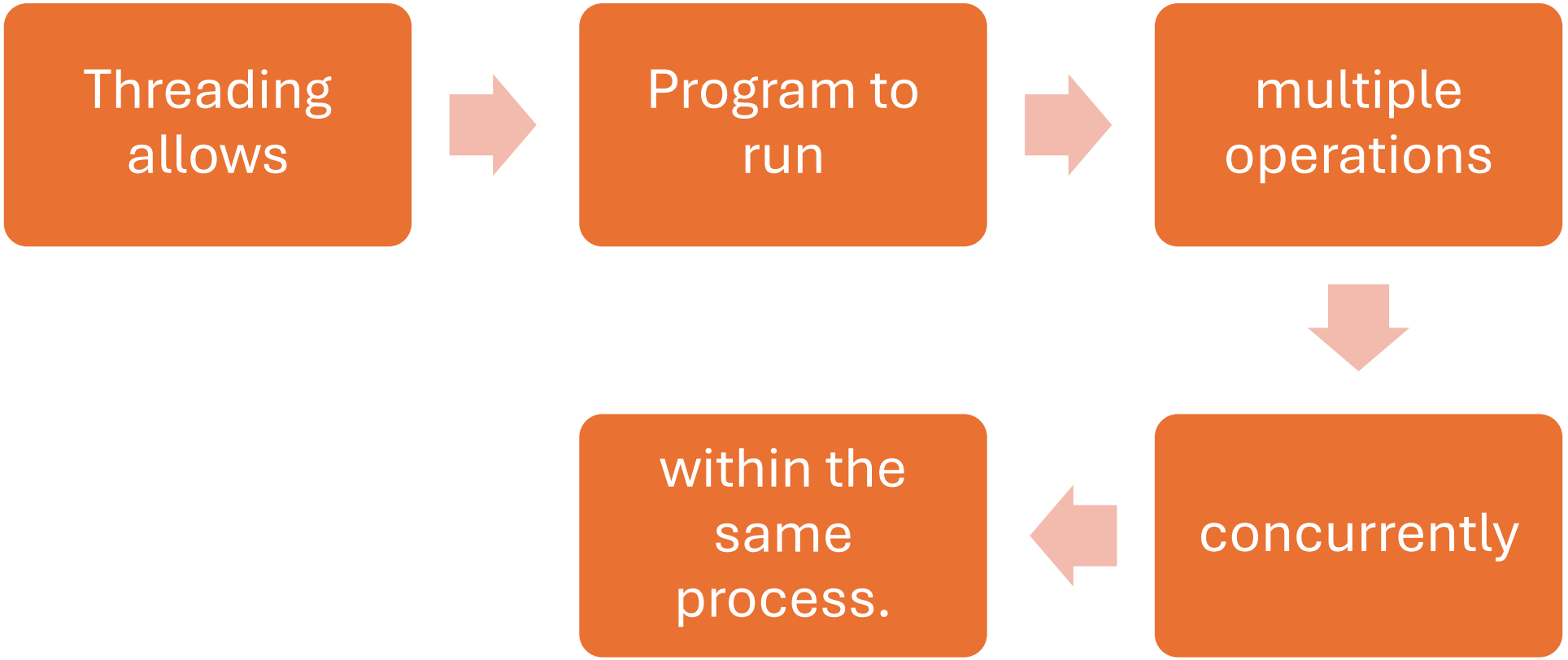
Threading

MULTIPLE THREADS

IN A SINGLE PROCESS

(SHARED MEMORY)

Threading


Threading allows
Program to run
multiple operations
concurrently
within the same process.

Threading

Useful for

I/O-bound tasks like

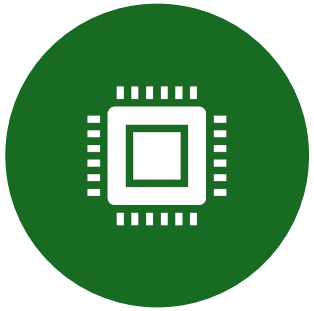
File I/O ,File logging,

API calls,

database writes

Threading

NOT IDEAL FOR



CPU-BOUND TASKS

DUE TO THE GLOBAL INTERPRETER LOCK

GIL IN CPYTHON

Why Threading in Banking?

Use Case	Threading Benefit
Logging	Doesn’t block main transaction
Auditing	Runs asynchronously in the background
Notification sending	SMS/Email notifications run in parallel
Fraud detection	Real-time checks without slowing user ops

Use Locks for Safety

Use threading.Lock()

to prevent **race conditions**

when threads update

shared data (like balance).

Summary

Concept	In Bank App
Threading	Parallel logging/auditing
Locking	Ensures consistent account balance updates
Performance	Improves responsiveness for I/O-bound tasks

What is Multiprocessing?

Technique that runs

multiple processes simultaneously,

each with its own

Python interpreter and

memory space.

Use it when

1. Tasks are CPU-bound

Example

Fraud detection,

Data encryption,

Report generation

Use it when

2. WANT TO UTILIZE

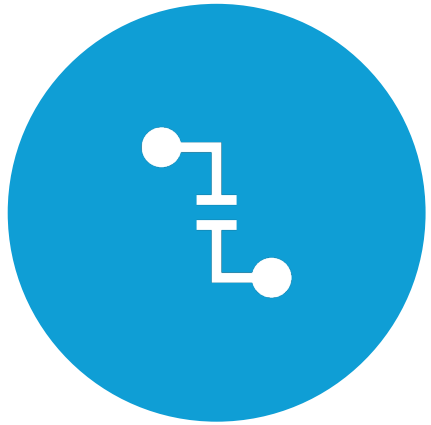
MULTIPLE CORES

FOR PERFORMANCE

What is Multiprocessing?

UNLIKE THREADING,

IT BYPASSES



GLOBAL INTERPRETER LOCK (GIL)

Bank Application Use Case

Imagine a bank needs to:

Process a batch of transactions

(deposits/withdrawals)

Bank Application Use Case

Perform fraud checks or

report generation in parallel

When to Use Multiprocessing in Banking?

Use Case	Why Use Multiprocessing?
Fraud detection	Heavy computation, needs full CPU usage
Data encryption	CPU-intensive encryption algorithms
Batch transaction analysis	Analyze thousands of accounts in parallel
Report generation	Run multi-core PDF/CSV exports

Summary

Feature	Multiprocessing
Parallelism	True parallelism (multi-core)
Use for	CPU-bound tasks
Shared memory	Requires Manager, Value, or Queue
Compared to Threading	Better for performance-heavy tasks

Multithreading vs Multiprocessing

Feature	Multithreading	Multiprocessing
Definition	Runs multiple threads within a single process	Runs multiple processes, each with its own memory
Concurrency Type	Concurrent (via threads)	True parallelism (via processes)
Memory Space	Shared memory (threads share same process memory)	Separate memory (each process has its own memory space)

Multithreading vs Multiprocessing

Feature	Multithreading	Multiprocessing
Global Interpreter Lock (GIL)	Affected by GIL – only one thread executes Python bytecode at a time	Not affected – each process has its own interpreter
Use Case Suitability	Best for **I/O-bound tasks** (e.g., API calls, file I/O)	Best for **CPU-bound tasks** (e.g., calculations, ML jobs)

Multithreading vs Multiprocessing

Feature	Multithreading	Multiprocessing
Overhead	Lightweight – lower memory and context-switching overhead	Heavyweight – higher memory and startup cost
Speed Boost	Improves responsiveness, but not CPU-bound performance	True speed boost on multicore machines
Communication	Shared state; need to manage thread-safety (threading.Lock)	Use multiprocessing.Queue, Pipe, or Manager for IPC

Multithreading vs Multiprocessing

Feature	Multithreading	Multiprocessing
Stability	Risk of **race conditions**, deadlocks	More stable, each process is isolated
Crash Impact	One thread crash can affect the entire process	One process crash usually doesn’t crash others
Debugging	Harder due to shared state and race conditions	Easier (more isolated, reproducible errors)

Multithreading vs Multiprocessing

Feature	Multithreading	Multiprocessing
Startup Time	Fast to start threads	Slower to start processes
Libraries	threading, concurrent.futures.ThreadPoolExecutor	multiprocessing, concurrent.futures.ProcessPoolExecutor
Example Use Case	Bank API scraper, async email/SMS sender, logging	Fraud detection, report generation, data aggregation

Summary

Task Type	Best Choice
I/O-bound	✅ Multithreading
CPU-bound	✅ Multiprocessing
Web scraping	Multithreading or Async IO
Heavy calculations	Multiprocessing
Realtime logging	Multithreading

What is Async I/O?

Lets your program

do more while waiting

What is Async I/O?

Allows your program

to handle many I/O-bound operations

concurrently without threads or processes.

What is Async I/O?

API CALLS

DATABASE QUERIES

FILE OPERATIONS

EXTERNAL SERVICES (SMS, EMAILS, FRAUD DETECTION)

What is Async I/O?

BEST FOR

MAKING MANY API CALLS

READING FILES/NETWORKING WITHOUT BLOCKING

CHATBOTS, CRAWLERS, SERVERS (E.G., FASTAPI)

Event Loop in asyncio

asyncio provides

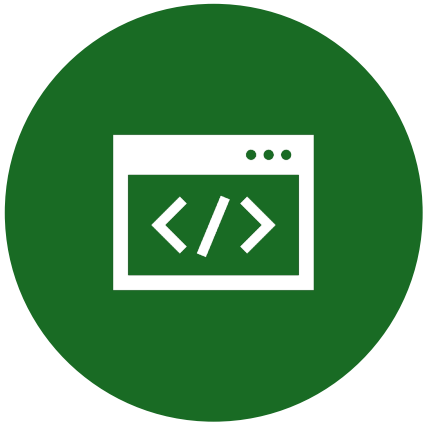
the **event loop**

that runs

async tasks

Event Loop in asyncio

REGISTERS **NON-BLOCKING I/O** OPERATIONS



(LIKE WAITING FOR HTTP RESPONSES) AND

RESUMES THEM WHEN READY

Basic Concepts

Keyword	Purpose
async def	Declares a coroutine (like a lazy function)
await	Pauses and yields control to the event loop
asyncio.run()	Starts the event loop

Key Components

Concept	Description
asyncio	Python’s built-in async framework
async/await	Define and pause/resume asynchronous functions
event loop	Runs and manages asynchronous tasks
aiohttp	Async HTTP client/server for non-blocking API calls

What is Profiling in Python?

PROCESS OF MEASURING

PERFORMANCE OF YOUR CODE

SPECIFICALLY HOW MUCH **TIME** &

MEMORY EACH PART OF

THE CODE CONSUMES.

What is Profiling in Python?

Why is my code slow?

Which functions are taking the most time or memory?

Where should I optimize?

Why Use Profiling?

Purpose	Benefit
Find bottlenecks	Improve performance
Optimize memory usage	Reduce crashes or slowdowns
Measure algorithm efficiency	Choose better logic
Improve scalability	Handle large data or traffic

Types of Profiling

Type	Tool	What it Measures
CPU Profiling	cProfile, line_profiler	Time spent per function/line
Memory Profiling	memory_profiler	Memory used per line
Concurrency Profiling	threading, multiprocessing	Thread/process execution

Common Profiling Tools in Python

Tool	Use Case
cProfile	Function-level time analysis
line_profiler	Line-by-line time analysis
memory_profiler	Line-by-line memory usage

Common Profiling Tools in Python

Tool	Use Case
timeit	Benchmark small code snippets
tracemalloc	Track memory allocations
py-spy / Snakeviz	Visualize profiling results

When Should You Profile?

App is **running slow**

Scaling to large data or users

Need to **optimize startup, memory, or speed**

Comparing **different algorithms**

Summary

Aspect	Details
Profiling	Measuring time and memory usage in code
Goal	Identify bottlenecks and optimize performance
Tools	cProfile, memory_profiler, line_profiler

cProfile – Function-Level CPU Profiler

Analyze how much time

each function takes **in total** and

how many times it is called.

python -m cProfile -s cumtime my_script.py

Understanding Columns in cProfile Output

Column	Meaning
ncalls	Number of times the function was called
tottime	Time spent only in that function, not in sub-functions
percall	tottime / ncalls
cumtime	Cumulative time = time in function + all sub-functions it calls
percall	cumtime / ncalls
filename:lineno(function)	Where the function is defined

line_profiler

Line-by-Line Execution Timing

pip install line_profiler

Decorate the function

@profile

memory_profiler

Line-by-Line Memory Usage

pip install memory_profiler

Decorate with @profile

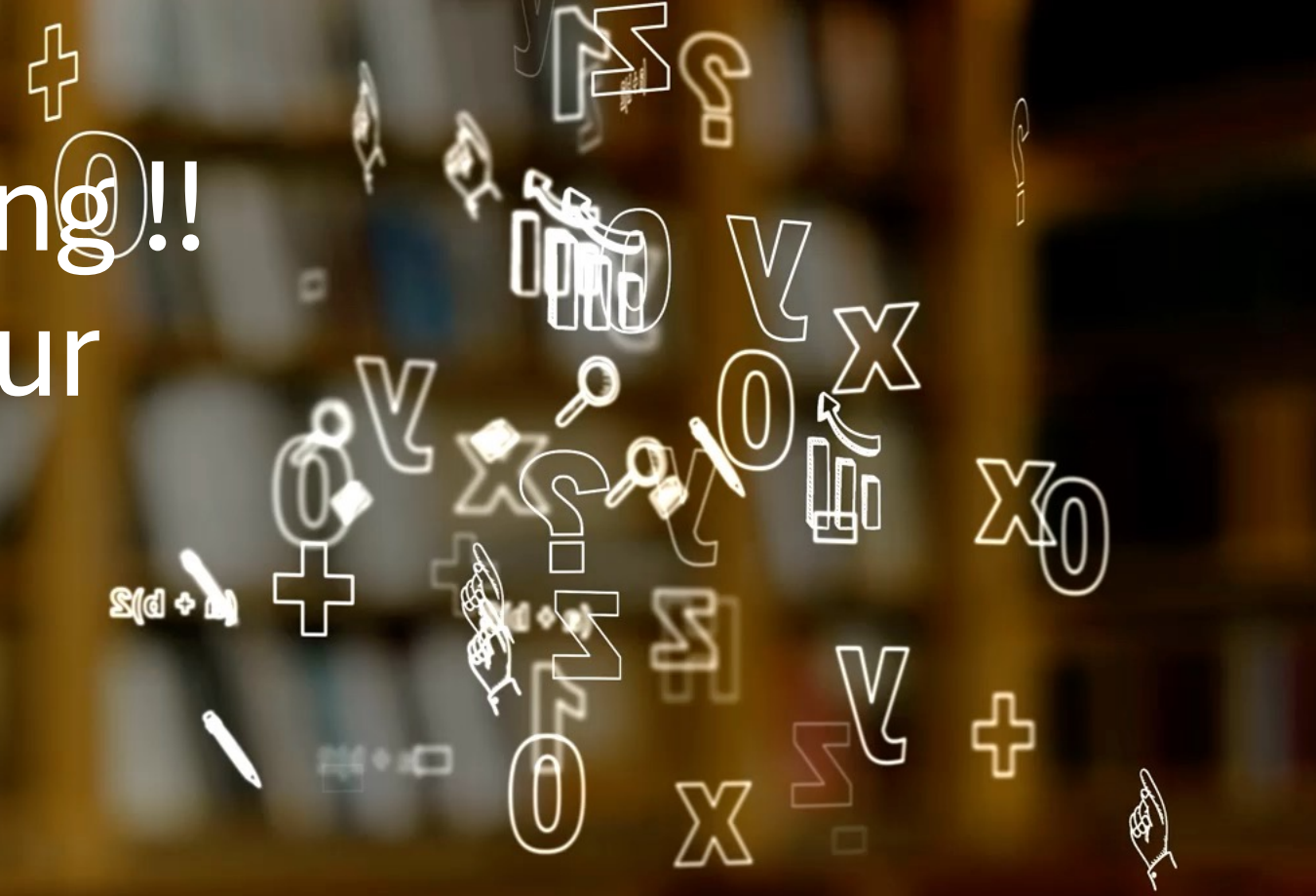
Summary Table

Tool	Tracks	Use Case in Bank App
cProfile	Function-level CPU time	Analyze bottlenecks in transaction engine
line_profiler	Line-level CPU time	Spot slow lines in fraud analysis loop
memory_profiler	Line-level memory usage	Catch memory-heavy operations in batch loads

When to Use Which?

You want to...	Use
Find slow functions in general	cProfile
Know which line is slow in a function	line_profiler
Track memory use for each line	memory_profiler

Happy Learning!! Thanks for Your Patience ☺



Surendra Panpaliya

GKTCS Innovations