.

**Workshop Title: *“Python: Beneath the Surface – Speed, Power, and Precision”***

**Duration**: 3 Days | 8 Hours/Day | Target Audience: Intermediate+ Python Developers

**Day 1 – Python: Behind the Curtain**

**Objective**: Grasp how Python truly works: memory, scope, and the interpreter internals.

**Session 1: Object Internals and Memory Model**

* id(), is, interning, object headers
* Reference counting, gc, weak references
* Small int/cache vs new allocation
* sys.getsizeof, \_\_slots\_\_, memory overhead
* **Best Practice**: Avoiding over-allocation with class optimizations
* **Challenge**: Instrument a class to track memory use and GC cycles

**Session 2: The CPython Execution Engine**

* Bytecode via dis, stack machine model
* How eval\_frame and call stack works
* Frame objects, traceback, inspect
* **Best Practice**: Avoid heavy use of exec, eval, and globals()
* **Challenge**: Build a toy bytecode-to-human language explainer

**Session 3: Scoping, Closures, and Code Objects**

* LEGB rule, cell objects, closure traps
* Accessing \_\_code\_\_, \_\_closure\_\_, co\_consts
* Late binding vs early binding in lambdas
* **Best Practice**: Prefer closures over classes for functional patterns but know limits
* **Challenge**: Create a self-modifying closure (function that rewrites its own logic)

**Session 4: Immutable vs Mutable – Buffer Protocol and Memoryview**

* bytes vs bytearray, slicing effects
* memoryview, array.array, zero-copy techniques
* Copy vs view behavior with NumPy
* **Best Practice**: Use memoryviews for I/O processing instead of full buffer loads
* **Challenge**: Implement a zero-copy windowed view over a binary file

**Homework**:  
Build a class SmartMatrix to support memory-efficient storage of large matrices using memoryview and \_\_slots\_\_.

**Day 2 – Speed, Scale, and the GIL**

**Objective**: Learn to optimize Python code using profiling, native extensions, and concurrency.

**Session 1: Profiling and Optimization**

* cProfile, line\_profiler, timeit, memory\_profiler
* Hotspot identification, algorithmic inefficiencies
* **Best Practice**: Profile before you optimize, avoid micro-optimization
* **Challenge**: Refactor a recursive Fibonacci vs memoized DP with memory and CPU profiling

**Session 2: Vectorization and Loop Elimination**

* NumPy internals, broadcasting rules
* Loop unrolling and JIT (via numba, numexpr)
* **Best Practice**: Never use Python for for numeric ops—vectorize!
* **Challenge**: Rewrite a CPU-bound image processing loop using NumPy + numexpr

**Session 3: Concurrency – Threads, Processes, and Async**

* GIL demystified, true parallelism via multiprocessing
* When to use threading, concurrent.futures, or asyncio
* uvloop, I/O bound optimizations
* **Best Practice**: Use asyncio for I/O; ProcessPoolExecutor for CPU
* **Challenge**: Build a parallel file compressor using multiprocessing and asyncio

**Session 4: Talking to C – ctypes, cffi, and Beyond**

* Using ctypes for calling C functions
* cffi vs ctypes vs cython
* Setup of a minimal setup.py C extension
* **Best Practice**: Use C for core math or data-heavy ops, isolate properly
* **Challenge**: Create a fast C-backed determinant calculator callable from Python

**Homework**:  
Update SmartMatrix to support C-backed methods for:

* matrix multiplication
* determinant
* row slicing with memoryview

**Day 3 – Extending and Embedding Python**

**Objective**: Use Python as a systems language: embedding C/C++, modifying CPython, and best practices.

**Session 1: CPython Source Dive and Custom Build**

* How to build Python from source (GitHub)
* ceval.c, object.c, pycore\_\*, frameobject.c
* Tracing new opcodes, printing trace info
* **Challenge**: Add a custom opcode that logs all function calls

**Session 2: Idiomatic Python for Performance**

* Why """" block comments are bad (memory + parsing)
* Generators vs lists vs iterators
* Lazy evaluation, itertools, generator expressions
* Common anti-patterns (e.g., default mutable args)
* **Best Practice**: Use generators, short functions, avoid premature abstraction
* **Challenge**: Refactor a bloated class using generators and composition

**Session 3: Cython & Pybind11**

* Write Cython versions of critical modules
* Use pyximport, cython.inline, typed memoryview
* pybind11 for C++ integration
* **Challenge**: Port matrix multiplication using both Cython and pybind11; compare speeds

**Session 4: Final Project Hackathon & Review**

* Integrate C backends, profiling, and async ops into SmartMatrix
* Expose a CLI using argparse or click
* Stream large matrix files and compute operations in chunks
* Benchmark vs NumPy, export profiling stats
* **Challenge**:  
  Build a fully optimized **CLI tool**:
  + Can load **multi-GB matrix files** using **lazy memoryview slicing**
  + Computes **determinants and eigenvalues** using a **C extension**
  + Uses **async I/O** for loading and writing files
  + Includes **real-time profiling stats** via decorators and logs