Python Mastery Practice Plan

Below is a structured list of topics and main problems to practice for mastering Advanced Python concepts, Data Structures and Algorithms, Memory Management, Async Programming, and Testing Frameworks. Each topic includes key practice problems to solidify your understanding.

Module 1: Advanced Python Concepts

Topics

- Object-Oriented Programming (OOP)
- Decorators
- Generators
- Context Managers

Practice Problems

1. **OOP**:
   - Create a `Library` class with `Book` and `Member` subclasses to manage book borrowing/returning (use inheritance, encapsulation).
   - Implement a `Shape` hierarchy (e.g., Circle, Rectangle) with polymorphism for area/perimeter calculations.
2. **Decorators**:
   - Write `log_call` decorator to log function calls with arguments and results.
   - Create a `restrict_access` decorator to limit function execution based on user roles.
3. **Generators**:
   - Build a generator for prime numbers up to n.
   - Implement a generator to yield lines from a large text file chunk by chunk.
4. **Context Managers**:
   - Create a custom context manager to handle database connections (open/close safely).
   - Write a `TempFile` context manager to create and delete temporary files.

Module 2: Data Structures and Algorithms Implementation

Topics

- Arrays/Lists, Linked Lists, Stacks, Queues
- Trees, Graphs
- Sorting, Searching, Recursion, Dynamic Programming

Practice Problems

1. **Arrays/Lists**:
    - Find the first non-repeating element in an array.
    - Merge two sorted lists without extra space.
2. **Linked Lists**:
    - Reverse a singly linked list.
    - Detect and remove a cycle in a linked list.
3. **Stacks/Queues**:
    - Implement a stack using two queues.
    - Design a queue using two stacks.
4. **Trees**:
    - Implement a Binary Search Tree (BST) with insert, search, and delete operations.
    - Find the lowest common ancestor in a binary tree.
5. **Graphs**:
    - Implement Breadth-First Search (BFS) to find shortest path in an unweighted graph.
    - Use Depth-First Search (DFS) to detect cycles in a directed graph.
6. **Sorting/Searching**:
    - Write quicksort and mergesort algorithms from scratch.
    - Implement binary search on a rotated sorted array.
7. **Dynamic Programming**:
    - Solve the knapsack problem (0/1 and fractional).
    - Compute the longest common subsequence of two strings.

Module 3: Memory Management and Performance Optimization

Topics

- Garbage Collection, Reference Counting
- Profiling, Optimization Techniques (e.g., list comprehensions, slots)

Practice Problems

1. **Memory Analysis**:
    - Compare memory usage of a list vs. tuple for 10,000 elements using `sys.getsizeof()`.
    - Identify memory leaks in a script creating recursive objects (e.g., nested lists).
2. **Profiling**:
    - Profile a factorial function (recursive vs iterative) using `cProfile`.
    - Use `timeit` to compare loop vs. list comprehension for summing 1M numbers.
3. **Optimization**:
    - Optimize a slow nested loop for matrix multiplication using NumPy.
    - Refactor a script to use `__slots__` in a class to reduce memory footprint.

Module 4: Async Programming and Concurrency

Topics

- Asyncio, Coroutines, Event Loops
- Threading, Multiprocessing, concurrent.futures

Practice Problems

1. **Async Programming**:
    - Write an async function to fetch and parse 10 URLs concurrently using `aiohttp`.
    - Implement an async producer-consumer pattern with a queue.
2. **Threading**:
    - Create a threaded program to process chunks of a large list in parallel.
    - Simulate a ticket booking system with thread-safe resource access.
3. **Multiprocessing**:
    - Parallelize a CPU-intensive task (e.g., prime number calculation) across multiple processes.
    - Use `Pool` to compute squares of 1M numbers in parallel.
4. **Concurrent Futures**:
    - Use `ThreadPoolExecutor` to download multiple files concurrently.
    - Implement a `ProcessPoolExecutor` for image resizing tasks.

Module 5: Testing Frameworks (pytest, unittest)

Topics

- unittest: Class-based testing
- pytest: Fixtures, Parametrization, Mocking

Practice Problems

1. **unittest**:
    - Write test cases for a calculator class (add, subtract, divide, handle division by zero).
    - Test a function that processes a list for edge cases (empty, single element, duplicates).
2. **pytest**:
    - Create a test suite with fixtures for a database connection simulation.
    - Use parametrization to test a string reversal function with 10 different inputs.
3. **Mocking**:
    - Mock an external API call in a test for a weather app function.
    - Test a file reader function by mocking file I/O operations.

Final Notes

- Practice 2-3 problems per topic daily, increasing difficulty over time.
- Use platforms like LeetCode, HackerRank, or Exercism for additional problems.
- Track progress by maintaining a GitHub repository for your solutions.
- Revisit and refactor solutions after completing each module to reinforce learning.