System Requirements Specification Index

For

Digital Communication Analyzer

Version 1.0

IIHT Pvt. Ltd.
fullstack@iiht.com

TABLE OF CONTENTS

Digital Communication Analyzer
System Requirements Specification

1 PROJECT ABSTRACT

Pixelate Studios, a game development company, is facing performance issues with their asset management system. During gameplay, their Python-based asset loader frequently causes lag spikes due to inefficient memory usage. The development team suspects memory leaks, excessive garbage collection pauses, and inefficient object lifecycle management. You've been tasked with creating a demonstration that explores Python's memory management mechanisms and develops strategies to optimize memory usage in their asset loading system. This assignment will help you understand Python's memory model, reference counting, garbage collection, and memory optimization techniques.

2 BUSINESS REQUIREMENTS:

Screen Name	Console input screen
Problem Statement	1. Demonstrate Python's memory allocation and deallocation mechanisms 2. Analyze memory usage patterns for different data structures 3. Identify and resolve memory leaks caused by circular references 4. Implement memory profiling to track object creation and destruction 5. Optimize memory usage through proper object lifecycle management 6. Document best practices for memory-efficient Python programming

3 CONSTRAINTS

3.1 IMPLEMENTATION REQUIREMENTS

1. Reference Counting Implementation:
   - Create classes with `__del__` method to observe object destruction
   - Demonstrate how multiple references affect object lifecycle
   - Example:
     ```python
     def demonstrate_reference_counting():
         class CountedObject:
             def __init__(self, name):
                 self.name = name
                 print(f"Object '{name}' created")

             def __del__(self):
                 print(f"Object '{self.name}' destroyed")

         # Create object with one reference
         obj = CountedObject("test")
         # Create second reference
         obj2 = obj
         # Remove first reference
         del obj
         # Object still exists due to second reference
         # Remove second reference
         del obj2
         # Now the object is destroyed
     ```
2. Circular Reference Implementation:
   - Create objects that reference each other in a cycle
   - Show how this prevents garbage collection
   - Demonstrate solutions using weak references
   - Example:
     ```python
     def create_circular_reference():
         class Node:
             def __init__(self, name):
                 self.name = name
                 self.neighbors = []
                 print(f"Node '{name}' created")

             def __del__(self):
                 print(f"Node '{self.name}' destroyed")
     ```

```
    # Create nodes
    nodes = [Node(f"Node-{i}") for i in range(3)]

    # Create circular references
    for i in range(3):
        nodes[i].neighbors.append(nodes[(i+1) % 3])

    # Lose external references
    del nodes
    gc.collect()
    # Objects with circular references won't be collected!
```

3. Memory Optimization Implementation:
    - Compare memory usage of different data structures
    - Demonstrate lazy evaluation with generators
    - Implement object pooling for reusing objects
    - Example:

```python
class ObjectPool:
    def __init__(self, factory_func, max_size=10):
        self.factory_func = factory_func
        self.max_size = max_size
        self.pool = []

    def get(self):
        if self.pool:
            return self.pool.pop()
        return self.factory_func()

    def release(self, obj):
        if len(self.pool) < self.max_size:
            self.pool.append(obj)
```

3.2 Technical Constraints

1. The implementation should use only standard library modules
    - `gc` - For interacting with Python's garbage collector
    - `sys` - For measuring object sizes
    - `time` - For basic performance measurements
    - `weakref` - For implementing weak references
    - No third-party libraries or external dependencies are needed.

3.3 EXPECTED OUTCOMES

1. Reference counting demonstration:
    - Students should observe object creation and destruction as references are added and removed
    - Object should only be destroyed when the last reference is removed
2. Circular reference demonstration:
    - Objects with circular references should not be garbage collected
    - The same objects using weak references should be properly garbage collected
3. Memory optimization results:
    - Generators should use significantly less memory than equivalent lists
    - Object pooling should show performance improvements over creating/destroying objects
    - Different data structures should show varying memory efficiency for the same data
4. General understanding:
    - Students should gain insights into Python's memory management mechanisms
    - Students should learn practical techniques to identify and avoid memory leaks
    - Students should understand memory-efficient programming patterns

4. TEMPLATE CODE STRUCTURE:

1. Memory Analysis Functions:

- `track_objects_count()` - Tracks objects before and after garbage collection
- `object_size(obj)` - Measures the memory footprint of an object
- `demonstrate_reference_counting()` - Shows Python's reference counting in action

2. Memory Leak Demonstrations:

- `create_circular_reference()` - Demonstrates memory leaks through circular references
- `fix_circular_reference()` - Shows how to prevent memory leaks using weak references

3. Memory Optimization Techniques:

- `compare_data_structures()` - Compares memory usage of different data structures
- `demonstrate_generator_vs_list()` - Shows memory benefits of generators over lists
- `ObjectPool` class - Implements object pooling for reusing objects

4. Main Program Function:

- Demonstrates different memory management concepts
- Compares memory-efficient vs. inefficient implementations

5. EXECUTION STEPS TO FOLLOW:

1. Run the memory profiling functions with different data structures
2. Observe memory usage patterns with and without circular references
3. Compare memory consumption between naive and optimized implementations
4. Analyze garbage collection behavior with different object types
5. Identify and fix memory leaks in the demonstration code
6. Document memory usage best practices based on observations