Memory Management in Python

Ever wondered how Python remembers everything... and then forgets it at the right time?

You are about to understand:

- Reference counting
- PyMalloc & Memory Pools
- Generational GC
- Tools like gc, tracemalloc.

LEVEL-0 : Basic Concepts

Q:1 - What is memory management ?

→ Memory management in Python is a Process of .

- Allocating memory to variables and objects
- Keeping track of references
- Automatically de allo cating unused memory.

Q:2 - What is Garbage Collection?

→ Garbage Collection (GC) is the automatic process of freeing up memory by deleting Objects that are no longer in use.

LEVEL-1: HOW MEMORY IS ORGANIZED IN PYTHON

1. Memory Types

Memory Type	Description
Stack Memory	stores function calls and local variables
Heap Memory	stores objects like lists, dicts, class instance

2. Python's Private Heap

- All Python objects are stored in a Private heap.
- Controlled by the Python memory manager.

3. Memory Blocks and Pools (PyMalloc)

- Python internally uses a custom allocator called PyMalloc.
- Divides memory into arenas → pools → blocks to optimize small object allocation.

LEVEL-2 : PYTHON OBJECT LIFECYCLE

1. Object Creation

```
x = [1, 2, 3]
```

- Python allocates memory for list object
- sets x as a reference to it.

2. Reference Counting

- Every objects Keeps track of how many references point to it.

```
import sys
sys.getrefcount(x)
```

- if ref count = 0 → Object is deleted from memory.

3. Del statement

```
del x    # removes reference
```

LEVEL-3 : GARBAGE COLLECTION STRATEGY

1. Reference Counting (Primary MC Mechanism)

- Automatic
- Fast
- Immediate deallocation when ref count = 0

2. Problem : Reference Cycles

```
class Node :
    def __init__(self):
        self.ref = None

a= Node()
b = Node()
a.ref = b
b.ref = a
```

- a and b reference each other → ref count never goes to 0.
- Memory leaks unless GC handles it.

3. Cyclic Garbage Collector (Secondary GC Mechanism)

- Python's gc module detects reference cycle
- Uses generational garbage collection

LEVEL-4 : GENERATIONAL GARBAGE COLLECTION

Python divides objects into three generations:

Generation	Description
Gen 0	Newest objects
Gen 1	Survived 1 GC cycle
Gen 2	Long-lived objects

- GC runs more frequently on younger generations
- Promotes survivors to older generations

GC Frequency :

- If an objects survives gen0 → moved to gen 1
- gen 2 is collected least frequently.

LEVEL - 5 : THE gc MODULE

1. Basic Usage

```
import gc

gc.collect()              # Run garbage collector manually
gc.get_count()            # Get count of objects in generations
gc.get_threshold()        # GC trigger threshold
```

2. Debugging

```
gc. set_debug (gc. DEDUG_LEAK)
```

3. Tracking Unreachable Objects

```
unreachable = gc. garbage
```

4. Disable / Enable Gc (not recommended unless profiling)

```
gc. disable ()
gc. enable ()
```

LEVEL-6 . MEMORY LEAKS IN PYTHON

Common Cause :

- Reference cycle with _del_() methods
- Closures capturing variable unintentionally
- C extensions or global caches

Avoid Leaks :

- Use weak reference (weakref module)
- Avoid custom destructors unless necessary.
- Break cycles manually if needed.

LEVEL-7 : TOOLS TO MONITOR & OPTIMIZE MEMORY

Tool	Use-Case
gc module	Inspect and control garbage collection
tracemalloc	Track memory allocations and leaks
Obj graph	Visualize object references and leaks
memory profiler	line-by-line memory usage
psutil	Monitor memory usage of entire Process

Example Using tracemalloc :

```
import tracemalloc
tracemalloc.start()
print(tracemalloc.get_trace_memory())
tracemalloc.stop()
```

LEVEL-8 : ADVANCED : Weakref MODULE

What is a weak reference ?

- A reference that does not increase reference count
- Useful to avoid memory leaks in caches or Observe patterns

```
import weakref

Class MyClass:
        pass

obj = MyClass
r = Weak.ref(obj)
print (r())    #returns obj
del obj
print (r())   #returns None
```