Advanced Python Programming

LESSON 1: Python Internals & Memory Management

Learning Objectives:

By the end of this lesson, participants will be able to:

- Understand Python's reference counting and garbage collection mechanisms.
- Use memory profiling tools like sys.getsizeof and tracemalloc effectively.
- Optimize memory usage through generators, **slots**, and specialized containers.
- Identify memory leaks and inefficient memory patterns in Python code.

Lesson Outline:

I. Python's Memory Model Overview (10 min)

Python manages memory automatically, but understanding the internals helps you write more efficient code.

Key Concepts:

- **Reference counting**: Python tracks how many references point to each object.
- **Garbage collection**: Handles circular references that reference counting can't clean up.
- **Memory pools**: Python pre-allocates memory for small objects to reduce allocation overhead.
- **Object overhead**: Every Python object has metadata overhead beyond its actual data.

Example of reference counting:

```
import sys

# Create an object and check its reference count
x = [1, 2, 3]
print(f"Reference count: {sys.getrefcount(x)}")  # Usually 2 (x +
temporary in getrefcount)

# Add another reference
y = x
print(f"Reference count: {sys.getrefcount(x)}")  # Usually 3

# Remove a reference
del y
print(f"Reference count: {sys.getrefcount(x)}")  # Back to 2
```

Commentary:

Reference counting provides immediate cleanup for most objects, but circular references require the garbage collector. Understanding this helps explain Python's memory behavior.

II. Memory Profiling with Built-in Tools (15 min)

Python provides several tools to measure and analyze memory usage.

Using sys.getsizeof():

```
import sys

# Compare memory usage of different data structures
numbers_list = [i for i in range(1000)]
numbers_tuple = tuple(numbers_list)
numbers_set = set(numbers_list)

print(f"List size: {sys.getsizeof(numbers_list)} bytes")
print(f"Tuple size: {sys.getsizeof(numbers_tuple)} bytes")
print(f"Set size: {sys.getsizeof(numbers_set)} bytes")

# Check individual object sizes
print(f"Integer: {sys.getsizeof(42)} bytes")
print(f"String: {sys.getsizeof('hello')} bytes")
print(f"Empty list: {sys.getsizeof([])} bytes")
```

Using tracemalloc for detailed tracking:

```
import tracemalloc

# Start tracing memory allocations
tracemalloc.start()

# Code that uses memory
data = []
for i in range(10000):
    data.append(f"Item {i}")

# Get current memory usage
current, peak = tracemalloc.get_traced_memory()
print(f"Current memory usage: {current / 1024 / 1024:.2f} MB")
print(f"Peak memory usage: {peak / 1024 / 1024:.2f} MB")

# Get top memory allocations
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("\nTop 3 memory allocations:")
for stat in top_stats[:3]:
    print(stat)

tracemalloc.stop()
```

Commentary:

These tools help identify memory hotspots in your code. Use them during development to catch memory issues early.

III. Memory Optimization with Generators (10 min)

Generators provide memory-efficient iteration by producing values on-demand rather than storing everything in memory.

List vs Generator comparison:

```
import sys

# Memory-heavy approach: list comprehension
def get_squares_list(n):
    return [x**2 for x in range(n)]

# Memory-efficient approach: generator
def get_squares_generator(n):
    return (x**2 for x in range(n))

# Compare memory usage
n = 100000
squares_list = get_squares_list(n)
squares_gen = get_squares_generator(n)

print(f"List memory: {sys.getsizeof(squares_list)} bytes")
print(f"Generator memory: {sys.getsizeof(squares_gen)} bytes")

# Both produce the same results, but generator uses constant memory
print(f"First 5 from list: {squares_list[:5]}")
print(f"First 5 from generator: {list(next(squares_gen) for _ in
range(5))}")
```

Generator functions for processing large datasets:

```
def process_large_file(filename):
    """Generator that processes file line by line without loading
everything into memory."""
    with open(filename, 'r') as file:
        for line in file:
            # Process each line
            yield line.strip().upper()

# Usage - memory efficient even for huge files
# for processed_line in process_large_file('huge_data.txt'):
#     print(processed_line)
```

Commentary:

Generators are essential for processing large datasets or infinite sequences. They trade some performance for massive memory savings.

IV. Optimizing Classes with slots (15 min)

By default, Python stores instance attributes in a dictionary, which has memory overhead. The `__slots__` attribute can reduce this overhead significantly.

Regular class vs slots comparison:

```
import sys

# Regular class with dynamic attributes
class RegularPoint:
    def __init__(self, x, y):
        self.x = x
        self.y = y

# Optimized class with __slots__
class SlottedPoint:
    __slots__ = ['x', 'y']

    def __init__(self, x, y):
        self.x = x
        self.y = y

# Compare memory usage
regular = RegularPoint(10, 20)
slotted = SlottedPoint(10, 20)

print(f"Regular point: {sys.getsizeof(regular)} bytes")
print(f"Regular point __dict__: {sys.getsizeof(regular.__dict__)} bytes")
print(f"Slotted point: {sys.getsizeof(slotted)} bytes")

# Memory difference becomes significant with many instances
regular_points = [RegularPoint(i, i*2) for i in range(1000)]
slotted_points = [SlottedPoint(i, i*2) for i in range(1000)]

print(f"\n1000 regular points: {sum(sys.getsizeof(p) +
sys.getsizeof(p.__dict__) for p in regular_points)} bytes")
print(f"1000 slotted points: {sum(sys.getsizeof(p) for p in
slotted_points)} bytes")
```

When to use slots:

```
# Good candidate for __slots__: data classes with fixed attributes
class Coordinate:
    __slots__ = ['latitude', 'longitude', 'altitude']

    def __init__(self, lat, lon, alt=0):
        self.latitude = lat
        self.longitude = lon
        self.altitude = alt

    def __repr__(self):
        return f"Coordinate({self.latitude}, {self.longitude},
{self.altitude})"

# Trade-offs of __slots__:
coord = Coordinate(40.7128, -74.0060)
print(coord)

# This would raise AttributeError with __slots__:
# coord.new_attribute = "Can't add this"
```

Commentary:

Use `__slots__` for classes where you create many instances and have fixed attributes. The memory savings can be 40-50% but you lose the flexibility of dynamic attributes.

V. Specialized Containers for Memory Efficiency (10 min)

Python provides specialized containers that can be more memory-efficient than standard types for specific use cases.

Using array for numeric data:

```
import array
import sys

# Standard list vs array for numbers
numbers_list = [i for i in range(1000)]
numbers_array = array.array('i', range(1000))  # 'i' = signed int

print(f"List of 1000 integers: {sys.getsizeof(numbers_list)} bytes")
print(f"Array of 1000 integers: {sys.getsizeof(numbers_array)} bytes")

# Arrays are more memory efficient but less flexible
print(f"List element type: {type(numbers_list[0])}")
print(f"Array element type: {type(numbers_array[0])}")
```

Using collections.deque for efficient queue operations:

```
from collections import deque
import sys

# deque is more memory-efficient for queue operations
regular_list = list(range(1000))
efficient_deque = deque(range(1000))

print(f"List memory: {sys.getsizeof(regular_list)} bytes")
print(f"Deque memory: {sys.getsizeof(efficient_deque)} bytes")

# deque provides O(1) operations on both ends
import timeit

# Benchmark adding to left side
list_time = timeit.timeit(lambda: regular_list.insert(0, 'new'),
number=1000)
deque_time = timeit.timeit(lambda: efficient_deque.appendleft('new'),
number=1000)

print(f"List insert(0): {list_time:.6f} seconds")
print(f"Deque appendleft: {deque_time:.6f} seconds")
```

Commentary:

Choose the right container for your use case. Arrays are great for numeric data, deques for queues, and sets for membership testing.

VI. Practical Memory Optimization Exercise (5 min)

Let's apply what we've learned to optimize a memory-intensive function:

```
# Before: Memory-inefficient version
def process_data_inefficient(data_source):
    # Load everything into memory at once
    all_data = [line.strip() for line in data_source]
    processed = [line.upper() for line in all_data if len(line) > 5]
    results = [f"Processed: {line}" for line in processed]
    return results

# After: Memory-efficient version
def process_data_efficient(data_source):
    # Use generator pipeline
    for line in data_source:
        cleaned = line.strip()
        if len(cleaned) > 5:
            yield f"Processed: {cleaned.upper()}"

# Usage comparison
```

```
sample_data = [f"line {i}" for i in range(10000)]

# Inefficient version loads everything
# result_list = process_data_inefficient(sample_data)

# Efficient version processes on-demand
result_generator = process_data_efficient(sample_data)
# Only process what you need
first_five = [next(result_generator) for _ in range(5)]
print("First 5 results:", first_five)
```

Commentary:

The efficient version uses constant memory regardless of input size, while the inefficient version memory usage grows with input size.

VII. Recap & Best Practices (5 min)

Key Takeaways:

- Use memory profiling tools to identify bottlenecks before optimizing.
- Generators are your friend for large datasets and streaming data.
- `__slots__` can significantly reduce memory for classes with many instances.
- Choose appropriate container types for your specific use case.
- Prefer lazy evaluation and streaming over loading everything into memory.

Best Practices:

- Profile first, optimize second - don't guess where memory issues are.
- Consider the trade-offs: memory vs. flexibility vs. performance.
- Use generators for data pipelines and large dataset processing.
- Apply `__slots__` to data classes with fixed attributes and many instances.

Final Multiple-Choice Question:

Which optimization technique would be MOST effective for reducing memory usage when processing a 10GB CSV file?

A. Using `__slots__` on all classes B. Using generators to process the file line by line C. Converting all lists to tuples D. Using `sys.getsizeof()` to monitor memory

(Answer: B. Using generators to process the file line by line - this enables constant memory usage regardless of file size.)