NANYANG TECHNOLOGICAL UNIVERSITY

SC1007: DATA STRUCTURES AND ALGORITHMS

Introduction & memory management in Python

College of Computing and Data Science

Introduction to Memory Management

- Memory management refers to the critical process of allocating, deallocating, and coordinating computer memory effectively.
- The primary objective of memory management is to ensure that all processes execute smoothly and efficiently utilize system resources.
- In Python, this is achieved through a combination of dynamic memory allocation, automatic garbage collection, and reference counting mechanisms.

Programs need memory to run
Memory is a limited resource

Memory Management Features in Python

- **Automatic Memory Management**: Python handles memory allocation/deallocation
- **Memory Pooling**: Reuse of previously allocated memory blocks
- **Memory Fragmentation Prevention**: Efficient organization of memory blocks
- Benefits: -
  - Prevents memory leaks
  - Optimizes memory usage
  - Reduces development complexity
  - Improves application performance

Poor memory management leads to crashes and slow programs

Understanding Python Memory Management: Names, References, and Objects

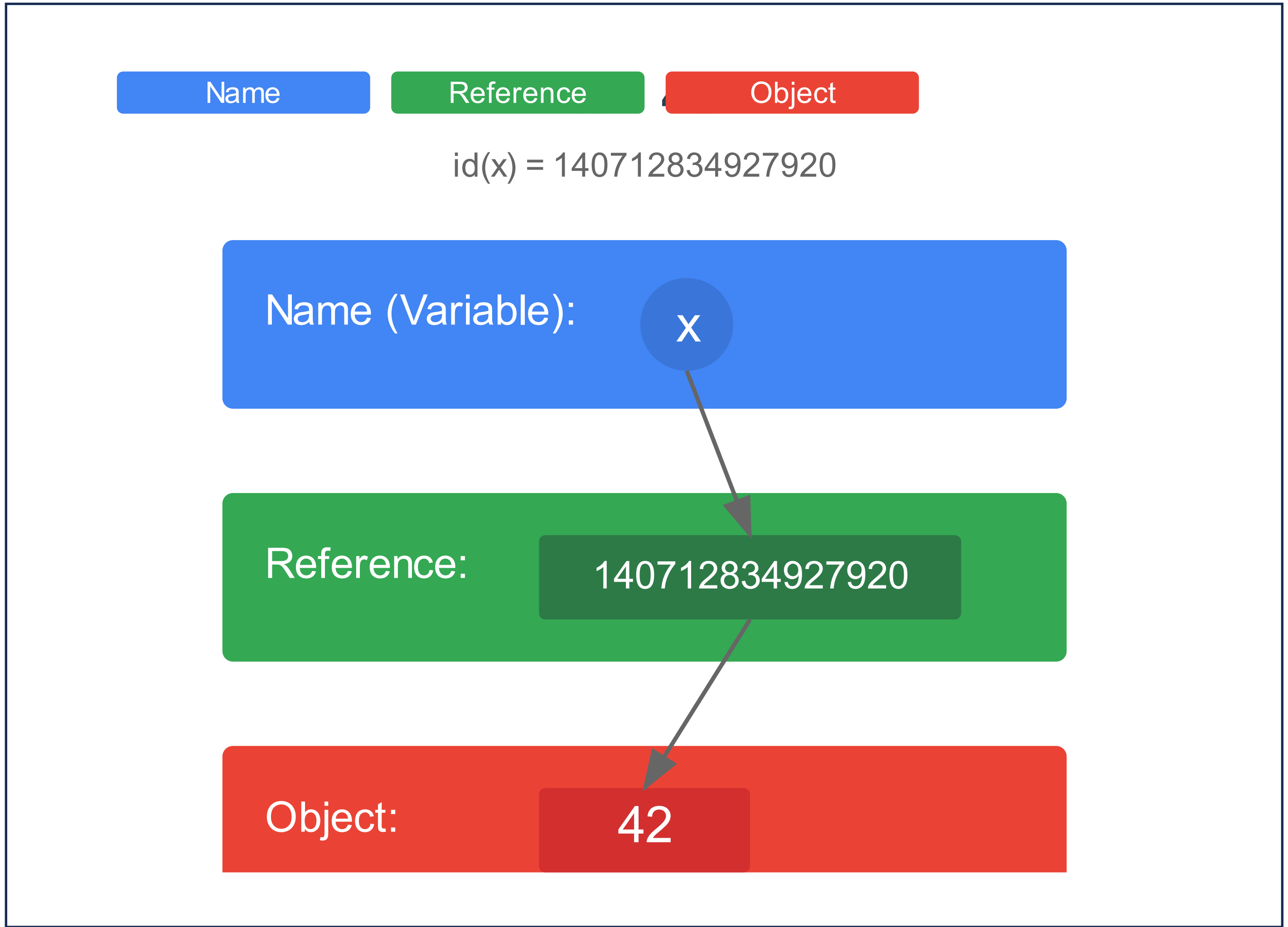
- **Names**: In Python, variables are essentially names that reference or point to objects in memory. When you create a variable, it doesn't directly hold the object itself but acts as a label or identifier.

- **References**: A reference is the link between the name and the object. This reference allows Python to manage and access the object in memory without directly storing it in the variable.

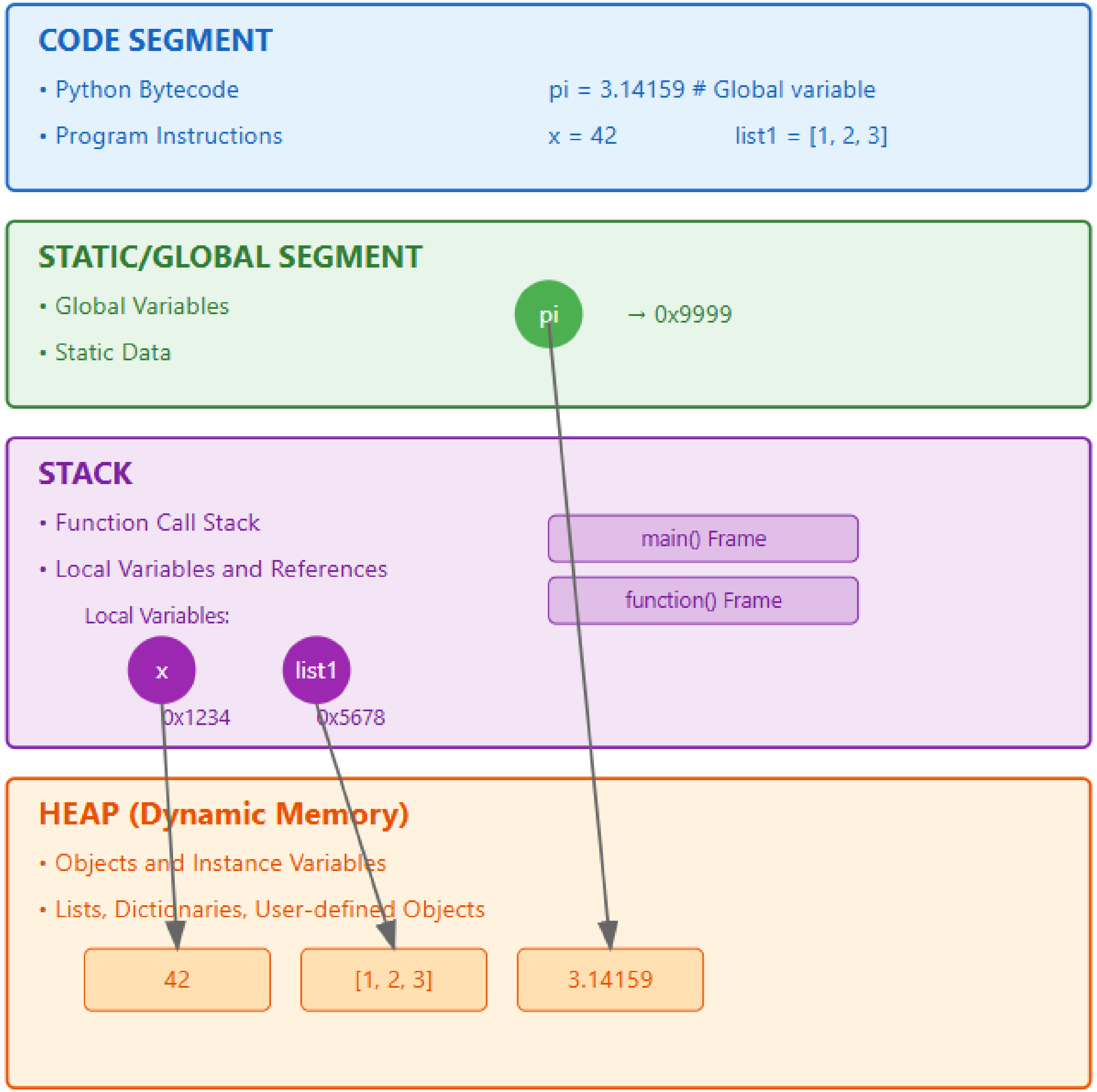
- **Objects**: Objects are the actual entities stored in memory. These can be data types such as integers, strings, lists, or user-defined objects. Every object has its own memory address, and Python keeps track of them using references.

Understanding Python Memory Management: Names, References, and Objects

```python
x = 42
print(id(x)) # e.g., 140712834927920
```



Python Memory Layout: Code, Stack, and Heap Segments



Code Segment: This segment contains the compiled Python bytecode and program instructions.

Static/Global Segment: This segment stores global variables and static data that are available throughout the program's execution.

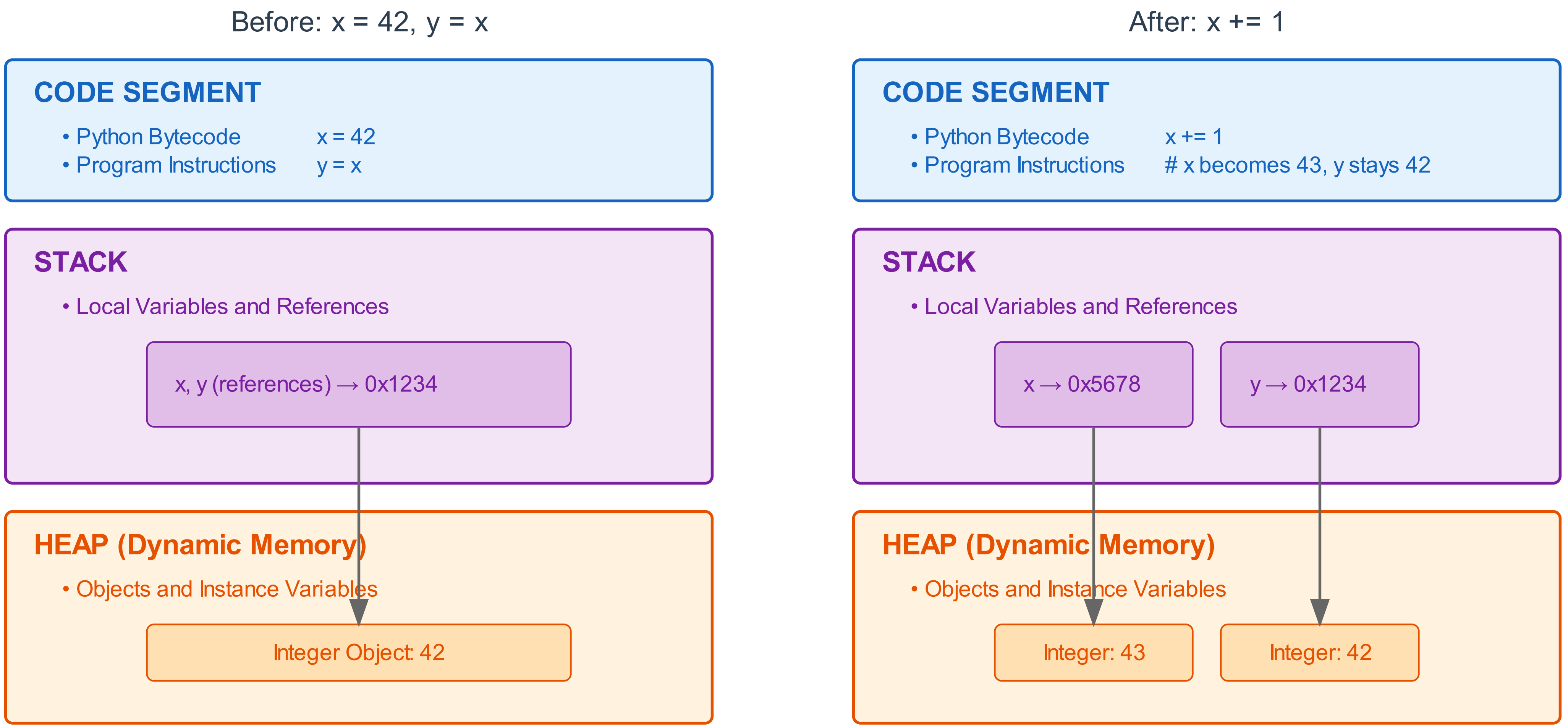
Stack: The stack is used for function call frames, local variables, and references.

Heap: The heap is used to store dynamically allocated objects, such as lists, dictionaries, or user-defined objects. Managed dynamically by Python’s garbage collector.

Memory Characteristics of Immutable Objects (Integers)

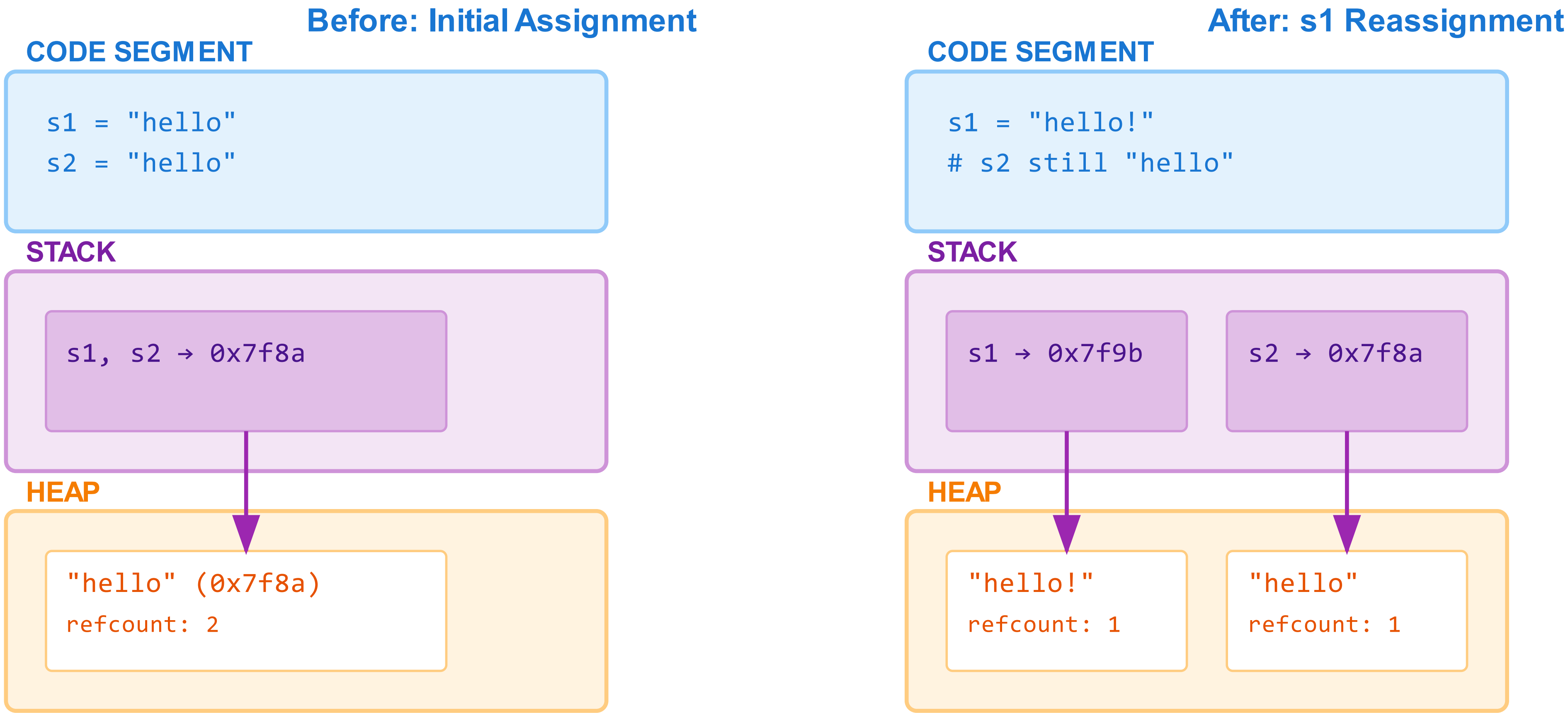
```python
x = 42
y = x
```

```python
x = 42
y = x
x += 1 # x becomes 43, but y stays 42
```



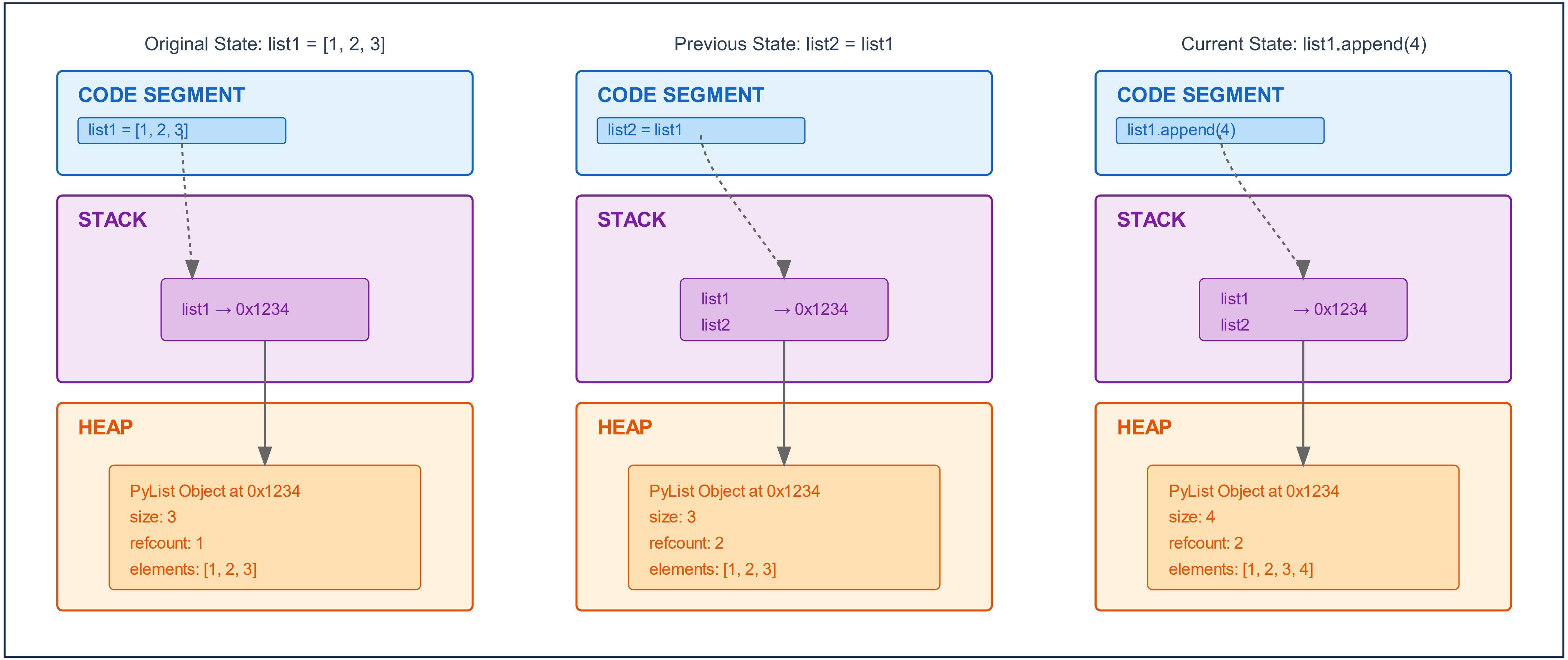
Memory Characteristics of Immutable Objects (Strings)

```
s1 = "hello"
s2 = "hello" # Same object reused
s1 = "hello!" # # Creates new object, s2 keeps old reference
```



Mutable Objects (list, dict, set)Memory Characteristics

```
list1 = [1, 2, 3] # Create list, list1 points to it
list2 = list1 # list2 points to same list
list1.append(4) # Modify the list object
```



Introduction to Memory Management

- Memory management in Python is handled automatically
- Multiple mechanisms work together to manage memory efficiently
- Understanding these concepts helps write better Python code
- Core Memory Management Mechanisms
  - Reference Counting
  - Garbage collection
  - Memory Pooling
  - Memory Interning
  - Manual Memory Management

1. Reference Counting

Reference Counting

- Primary memory management mechanism in Python
- Each object maintains a count of references pointing to it
- When count reaches zero, object is automatically deallocated
- Immediate memory reclamation
- Handles most common memory management scenarios

Reference Counting: Advantages

1. **Predictable Performance**
    - Immediate cleanup of unused objects
    - No unexpected pauses in execution
    - Memory freed as soon as it's no longer needed
2. **Deterministic Resource Management**
    - Resources released promptly
    - Efficient for memory-intensive applications
    - Good for real-time systems
3. **Simple Implementation**
    - Easy to understand and maintain
    - Transparent to developers
    - Automatic memory management
4. **Memory Efficiency**
    - No need to wait for garbage collector
    - Reduced memory footprint
    - Better cache utilization

Python Reference Counting Example

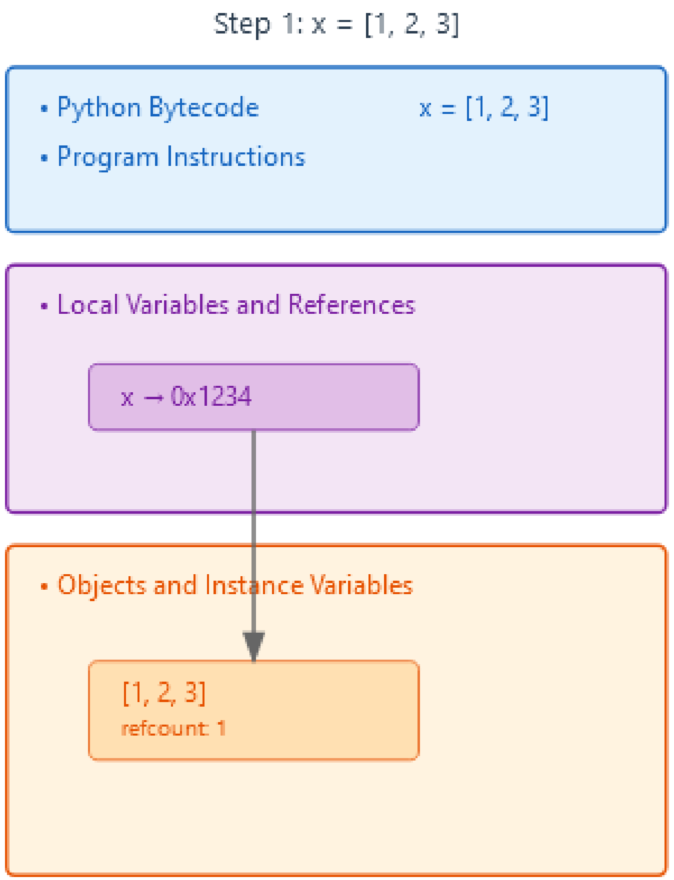
- When a list, such as [1, 2, 3], is assigned to a variable (e.g., x), its reference count is initially set to 1.
- If additional references are created, such as through the assignments y = x and z = x, the reference count increases to 3.
- Conversely, as references are removed using the del statement, the reference count decreases.
- Once the reference count reaches zero, Python's memory management system automatically deallocates the memory associated with the list. This mechanism ensures efficient memory usage and prevents memory leaks in applications.

Python Reference Counting Example

```python
# Import required modules
import sys      # For checking reference counts

# Create list and first reference
x = [1, 2, 3]
print(sys.getrefcount(x) - 1)

# Add second reference
y = x
print(sys.getrefcount(y) - 1)

# Add third reference
z = x
print(sys.getrefcount(z) - 1)

# Remove first reference
del x

# Remove second reference
del y

# Remove final reference
del z

# Object now eligible for garbage collection
# Force garbage collection (optional)
```

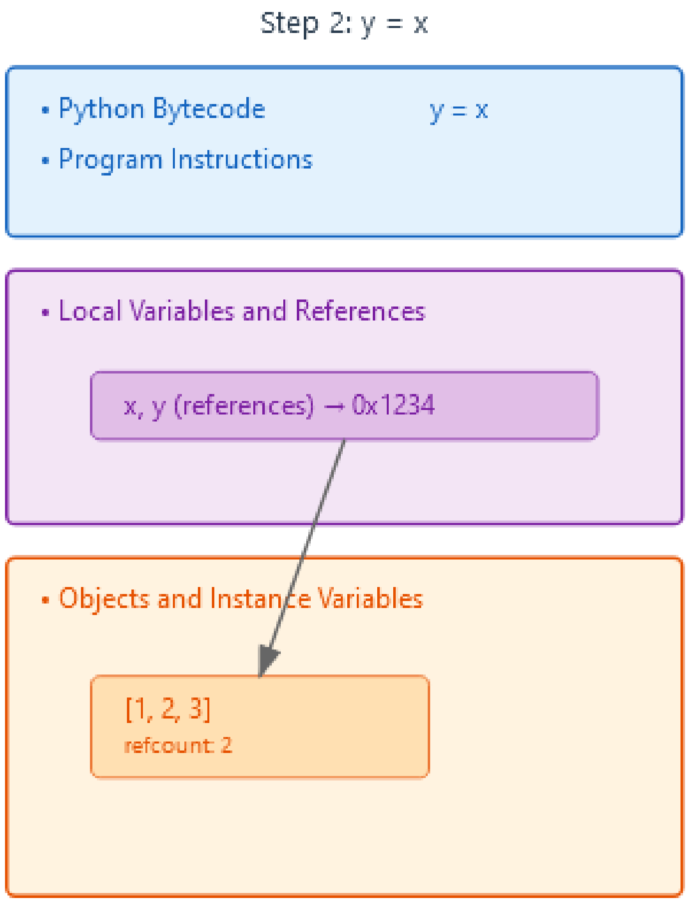
Python Reference Counting Example

```python
# Import required modules
import sys      # For checking reference counts

# Create a list and assign it to variable x
x = [1, 2, 3]  # Reference count = 1
# Print current reference count minus 1
# (subtract 1 because getrefcount() creates a temporary reference)
print(sys.getrefcount(x) - 1) # Output: 1
# Memory state: One reference (x) pointing to [1, 2, 3]
```



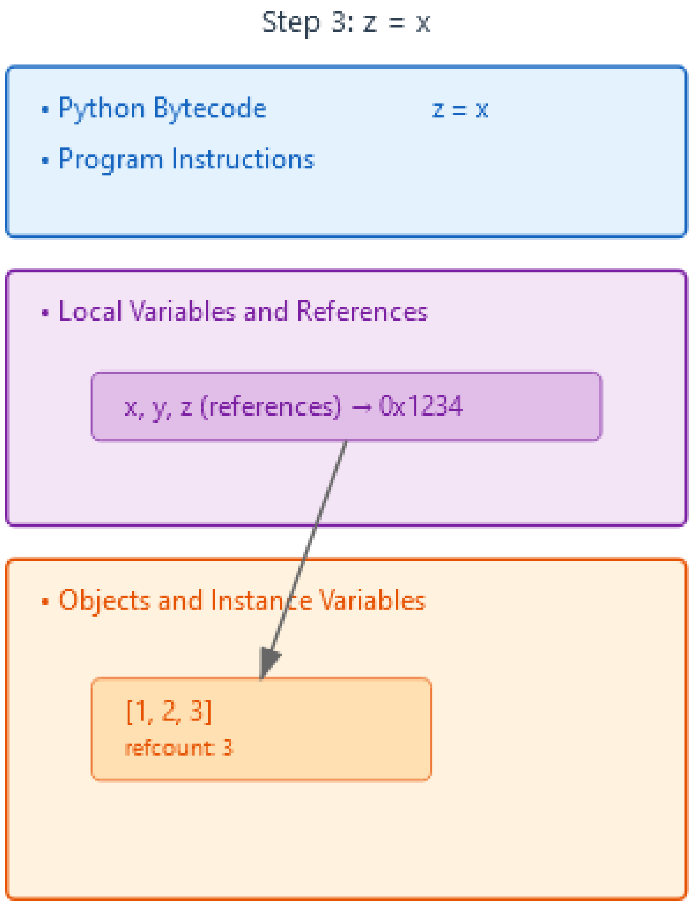
Python Reference Counting Example

```python
# Import required modules
import sys      # For checking reference counts

# Create a list and assign it to variable x
x = [1, 2, 3]  # Reference count = 1
# Print current reference count minus 1
# (subtract 1 because getrefcount() creates a temporary reference)
print(sys.getrefcount(x) - 1) # Output: 1
# Memory state: One reference (x) pointing to [1, 2, 3]

# Create another reference y pointing to the same list object
y = x # Reference count = 2
print(sys.getrefcount(y) - 1) # Output: 2
# Memory state: Two references (x, y) pointing to [1, 2, 3]
```



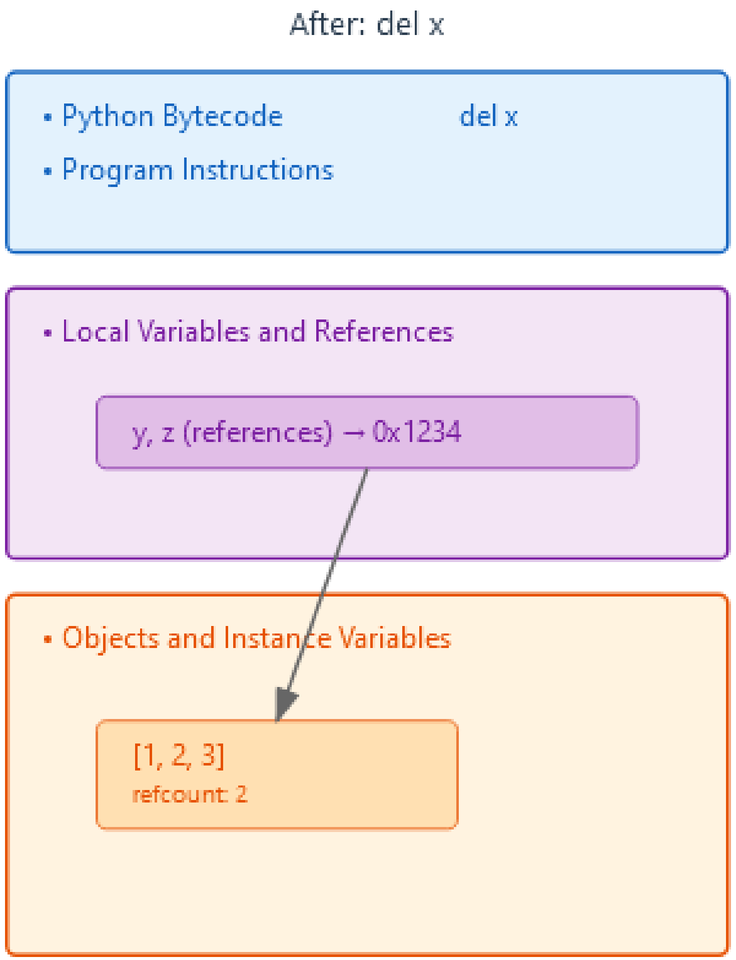
Python Reference Counting Example

```python
# Import required modules
import sys      # For checking reference counts

# Create a list and assign it to variable x
x = [1, 2, 3]  # Reference count = 1
# Print current reference count minus 1
# (subtract 1 because getrefcount() creates a temporary reference)
print(sys.getrefcount(x) - 1) # Output: 1
#Memory state: One reference (x) pointing to [1, 2, 3]

# Create another reference y pointing to the same list object
z = x  # Reference count = 2
print(sys.getrefcount(y) - 1) # Output: 2
# Memory state: Two references (x, y) pointing to [1, 2, 3]

# Create third reference z pointing to the same list object
z = x # Reference count = 3
print(sys.getrefcount(z) - 1) # Output: 3
# Memory state: Three references (x, y, z) pointing to [1, 2, 3]
```



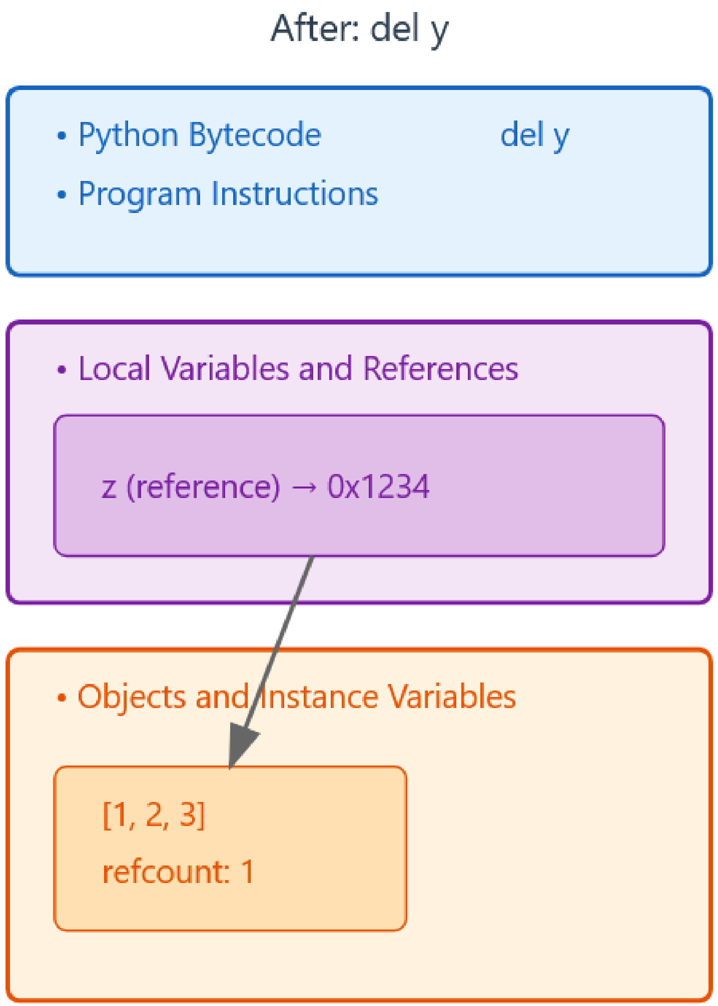
Python Reference Counting Example

```python
# Delete x reference
del x # Reference count = 2
# Memory state: refcount decreases to 2
# Two references remain (y, z) pointing to [1, 2, 3]
```



Python Reference Counting Example

```python
# Delete x reference
del x # Reference count = 2
# Memory state: refcount decreases to 2
# Two references remain (y, z) pointing to [1, 2, 3]

# Delete y reference
del y # Reference count = 1
# Memory state: refcount decreases to 1
# One reference remains (z) pointing to [1, 2, 3]
```



Python Reference Counting Example

```python
# Delete x reference
del x  # Reference count = 2
# Memory state: refcount decreases to 2
# Two references remain (y, z) pointing to [1, 2, 3]

# Delete y reference
del y  # Reference count = 1
# Memory state: refcount decreases to 1
# One reference remains (z) pointing to [1, 2, 3]

# Delete z reference
del z # Reference count = 0
# Memory state: refcount becomes 0
# No references remain, object becomes eligible for garbage collection
# At this point, when refcount = 0:
# Object is marked as eligible for garbage collection
```

After: del z

- Python Bytecode — del z
- Program Instructions

- Local Variables and References

No references remain

- Objects and Instance Variables

[1, 2, 3]
refcount: 0
Eligible for GC

Python Reference Counting Example

```
# Import required modules
import sys                    # For checking reference counts

# Create list and first reference
x = [1, 2, 3]                 # Creates list object in heap # Initial reference count = 1
print(sys.getrefcount(x) - 1)    # Output: 1

# Add second reference
y = x                         # y points to same object as x # Increases reference count to 2
print(sys.getrefcount(x) - 1)    # Output: 2 # Shows two references (x and y)

# Add third reference
z = x                         # z points to same object as x and y # Increases reference count to 3
print(sys.getrefcount(x) - 1)    # Output: 3 # Shows three references (x, y, and z)

# Remove first reference
del x                         # Removes x's reference # Decreases count to 2 # Object still exists in memory

# Remove second reference
del y                         # Removes y's reference # Decreases count to 1 # Object still exists in memory

# Remove final reference
del z                         # Removes final reference # Reference count becomes 0

# Object now eligible for garbage collection
# Force garbage collection (optional)
# Not typically needed # Python automatically handles cleanup # Returns number of objects collected
```

Reference Counting: Disadvantages

1. **Overhead Costs**
   - Additional memory for reference counts
   - CPU cycles needed for updating counters
2. **Circular References**
   - Cannot handle circular references alone
   - Requires additional garbage collector
   - Potential memory leaks if not managed
3. **Threading Complication**
   - Reference counting must be thread-safe
   - Requires atomic operations
   - Can impact performance in multi-threaded apps

Reference Counting: Circular References

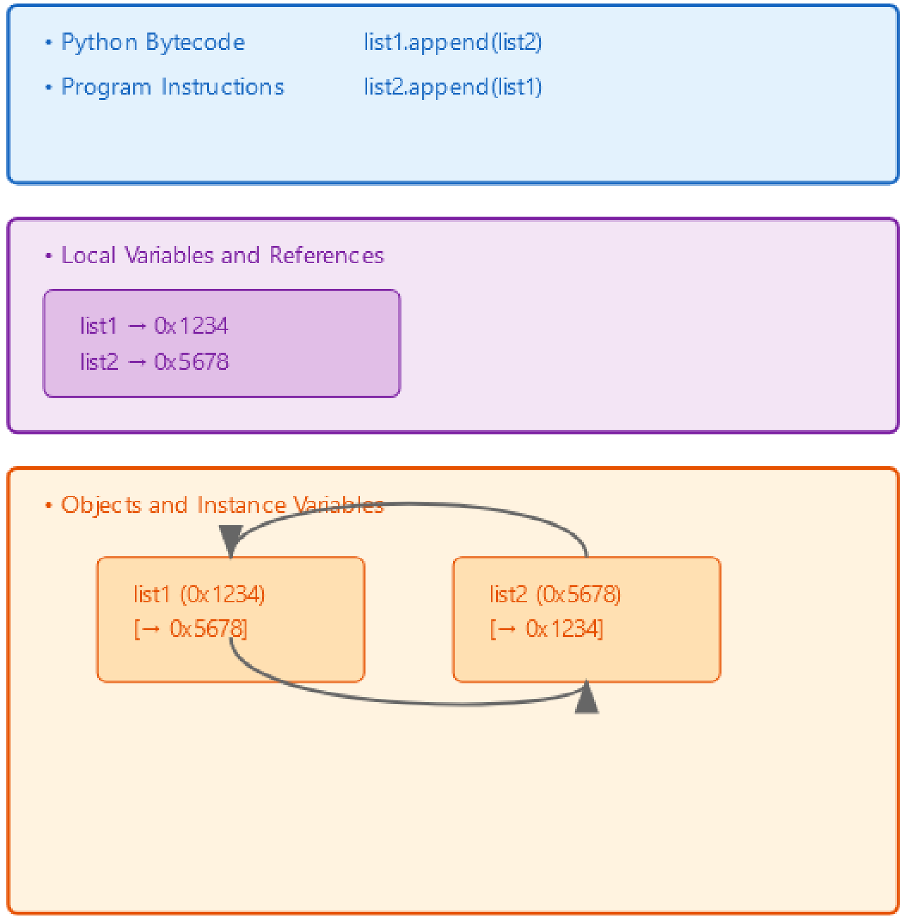
Reference Counting with Circular References occurs when two or more objects or data structures reference each other in a cycle. The problem arises because each object maintains a non-zero reference count due to the circular dependency, even when they become unreachable from the rest of the program.

- Lists referencing each other create cycles
- Memory stays allocated due to internal references
- Reference count never reaches zero
- Garbage collection needed to detect and free circular references

Reference Counting: Circular References

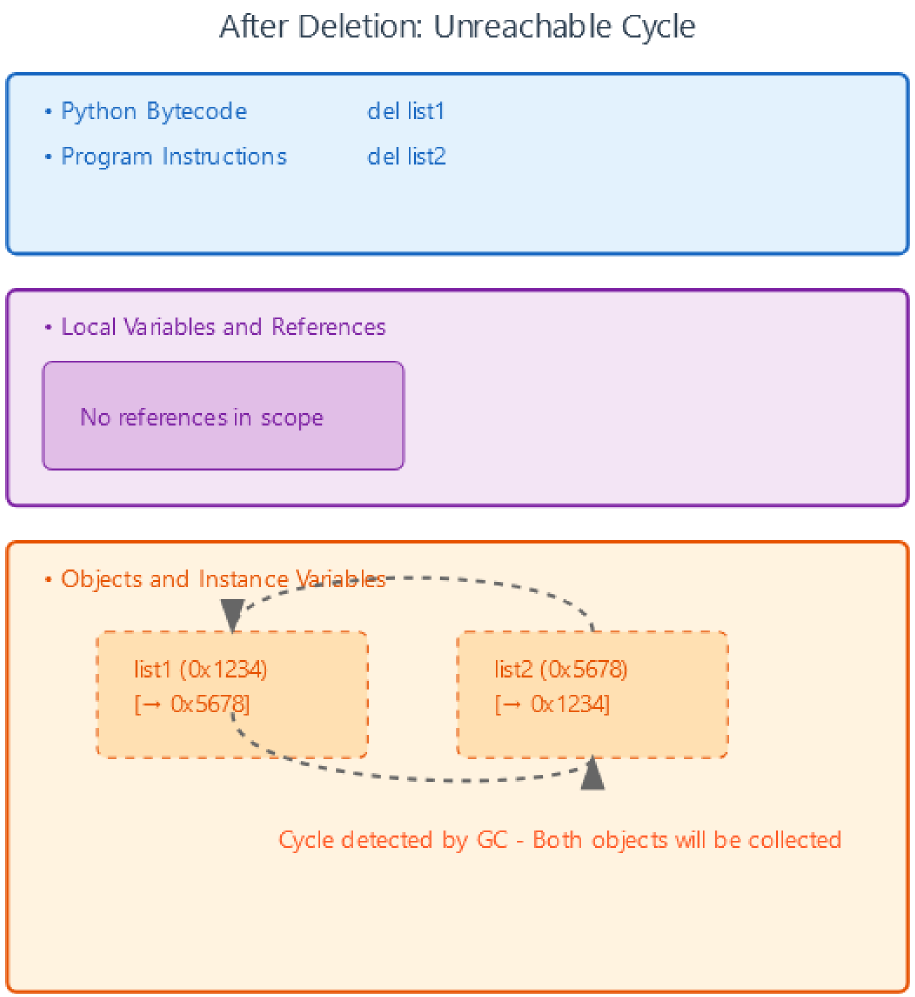
```python
# Create two lists
list1 = []
list2 = []

# Create circular reference
list1.append(list2) # list2 referenced by list1
list2.append(list1) # list1 referenced by list2
```

Before Deletion: Circular Reference



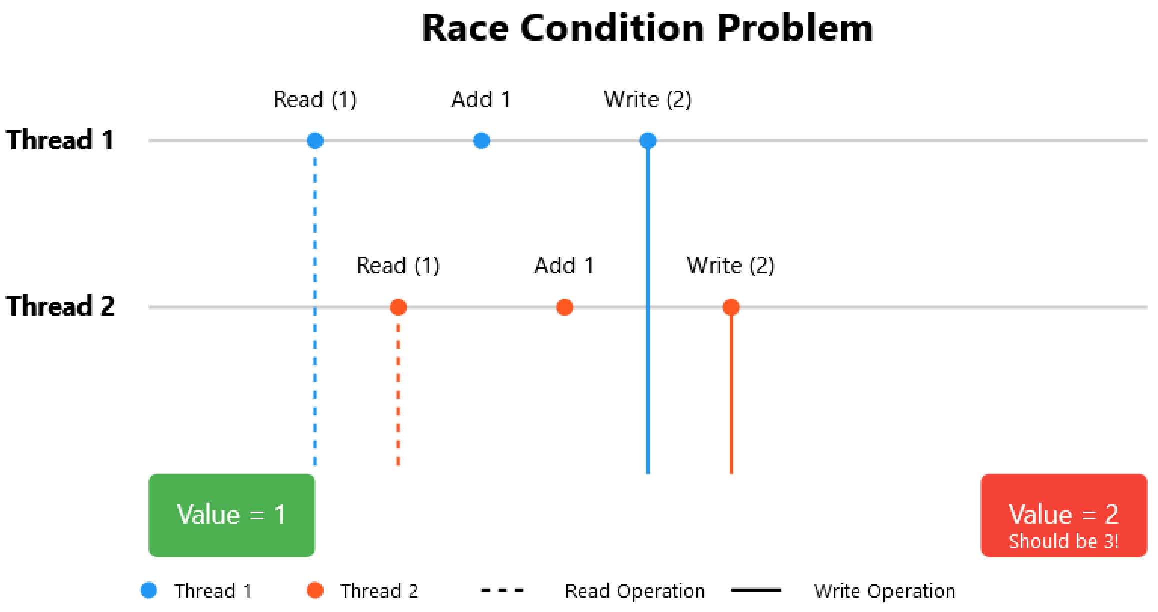
Reference Counting: Circular References

```python
# Create two lists
list1 = []
list2 = []

# Create circular reference
list1.append(list2) # list2 referenced by list1
list2.append(list1) # list1 referenced by list2

# Even after deletion, lists still reference each other
del list1 # Still referenced by list2[0]
del list2 # Still referenced by list1[0]

# Garbage collector detects these objects are unreachable
# Memory will be freed in next GC cycle
```



Reference Counting: Threading Complication

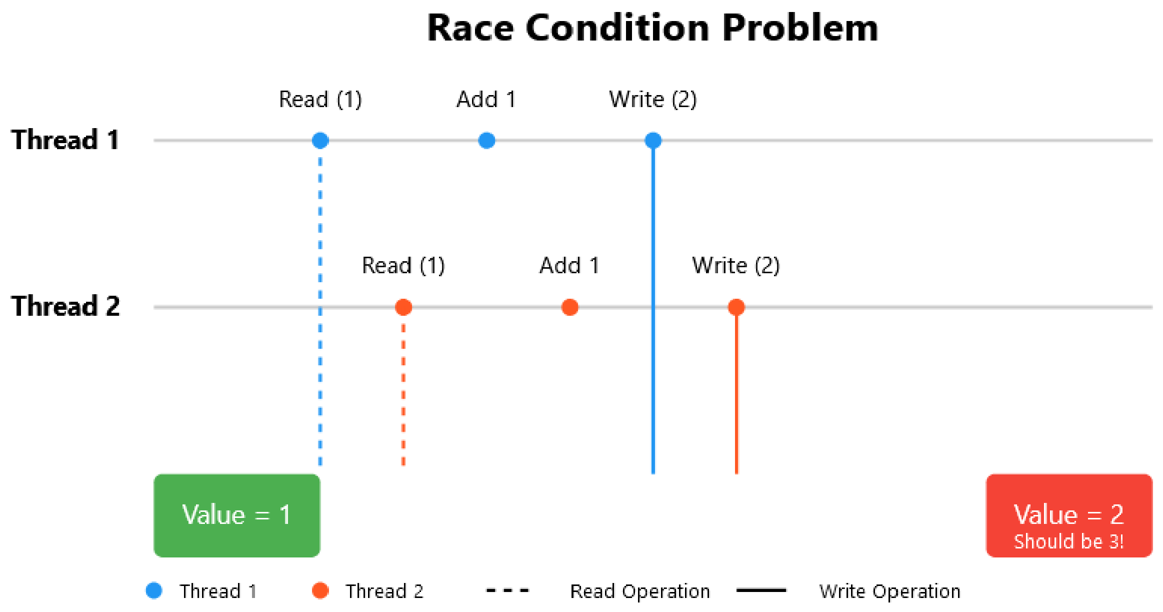
- A race condition occurs when two threads simultaneously try to modify shared data without proper synchronization.



Reference Counting: Threading Complication

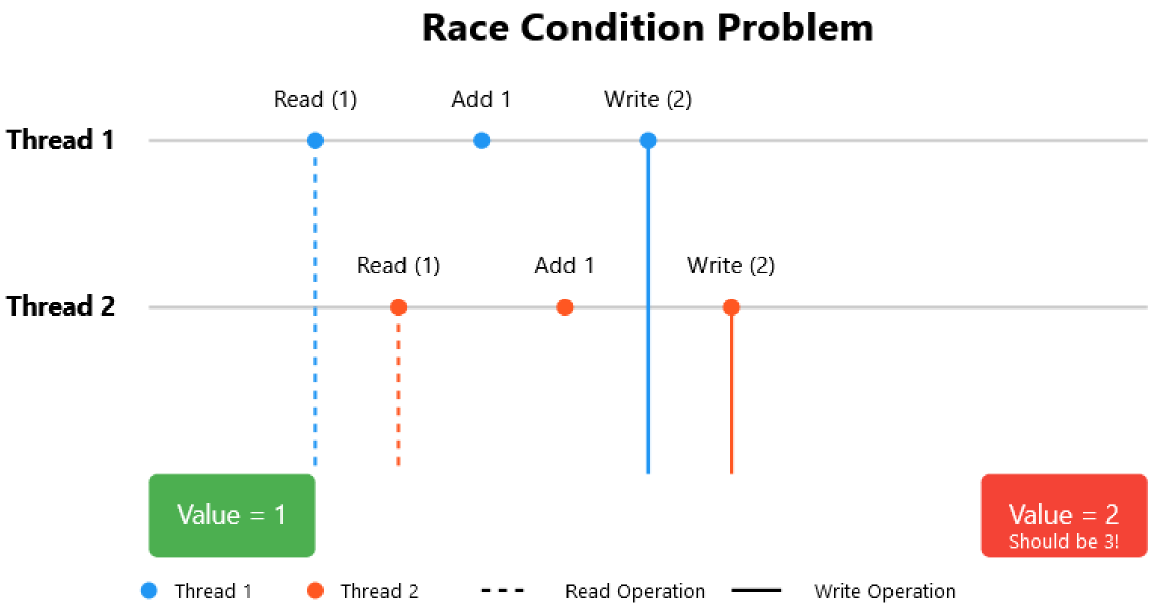
- A race condition occurs when two threads simultaneously try to modify shared data without proper synchronization.

- In this diagram, two threads attempt to increment a counter starting at 1. Both threads read the initial value (1), add 1, and write back 2.



Reference Counting: Threading Complication

- A race condition occurs when two threads simultaneously try to modify shared data without proper synchronization.
- In this diagram, two threads attempt to increment a counter starting at 1. Both threads read the initial value (1), add 1, and write back 2.
- Due to the lack of synchronization, the final value is **2** instead of the expected **3**, as one increment operation is effectively lost.



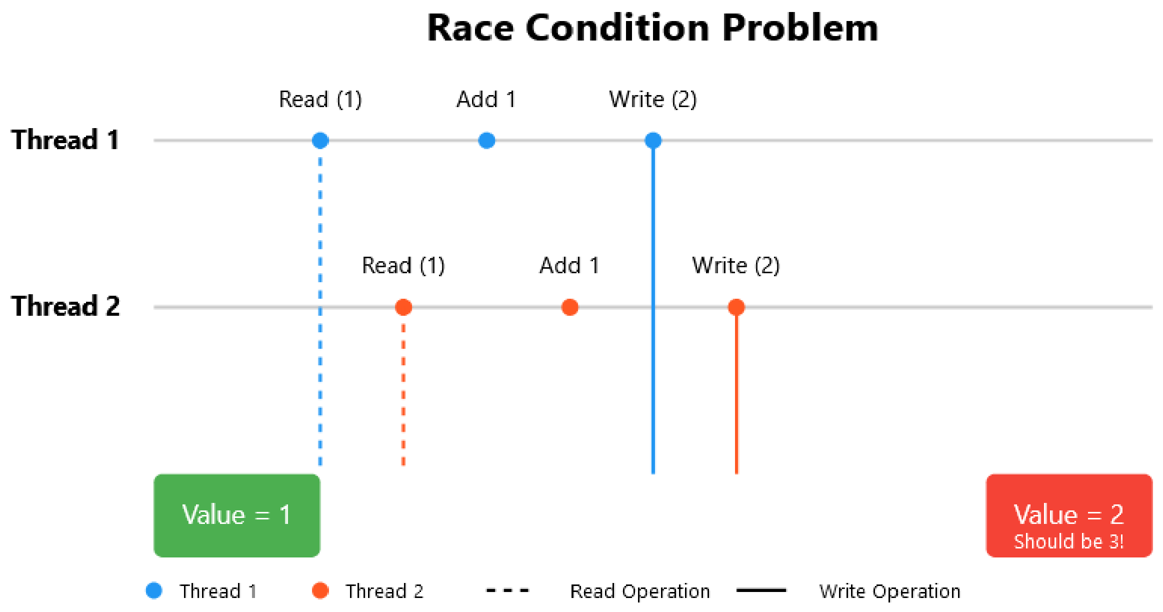
Reference Counting: Threading Complication

- A race condition occurs when two threads simultaneously try to modify shared data without proper synchronization.

- In this diagram, two threads attempt to increment a counter starting at 1. Both threads read the initial value (1), add 1, and write back 2.

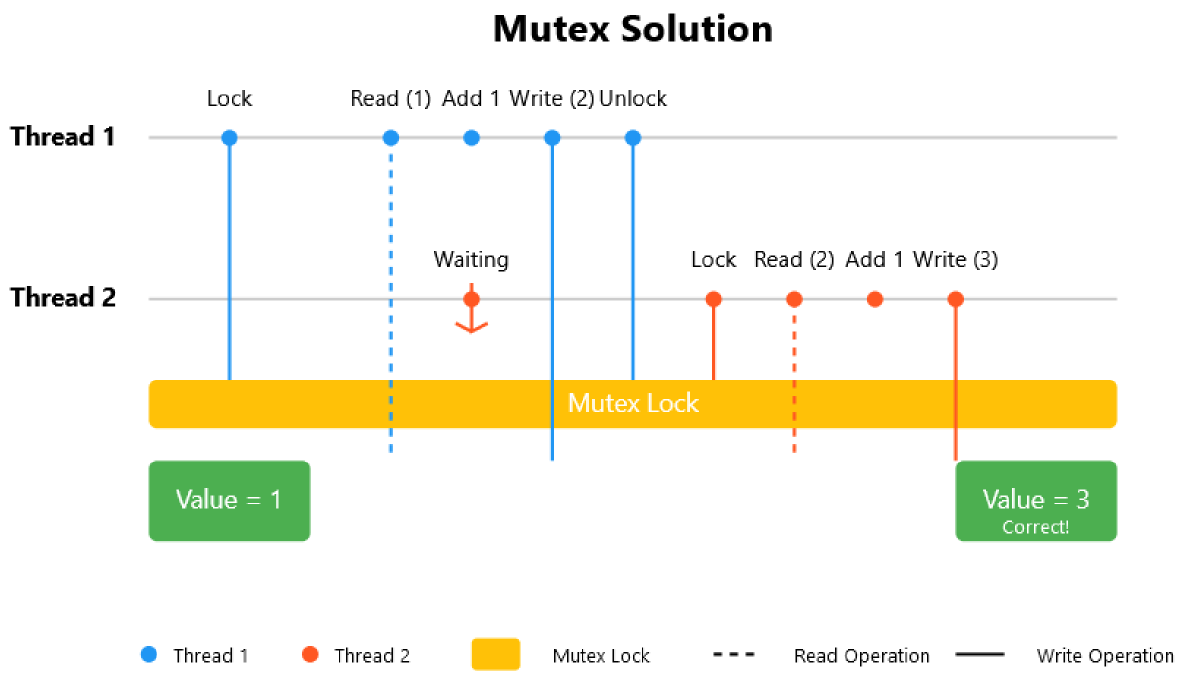
- Due to the lack of synchronization, the final value is **2** instead of the expected 3, as one increment operation is effectively lost.

- The **blue** and red lines represent **Thread 1** and Thread 2 respectively, with dashed lines showing read operations and solid lines showing write operations.



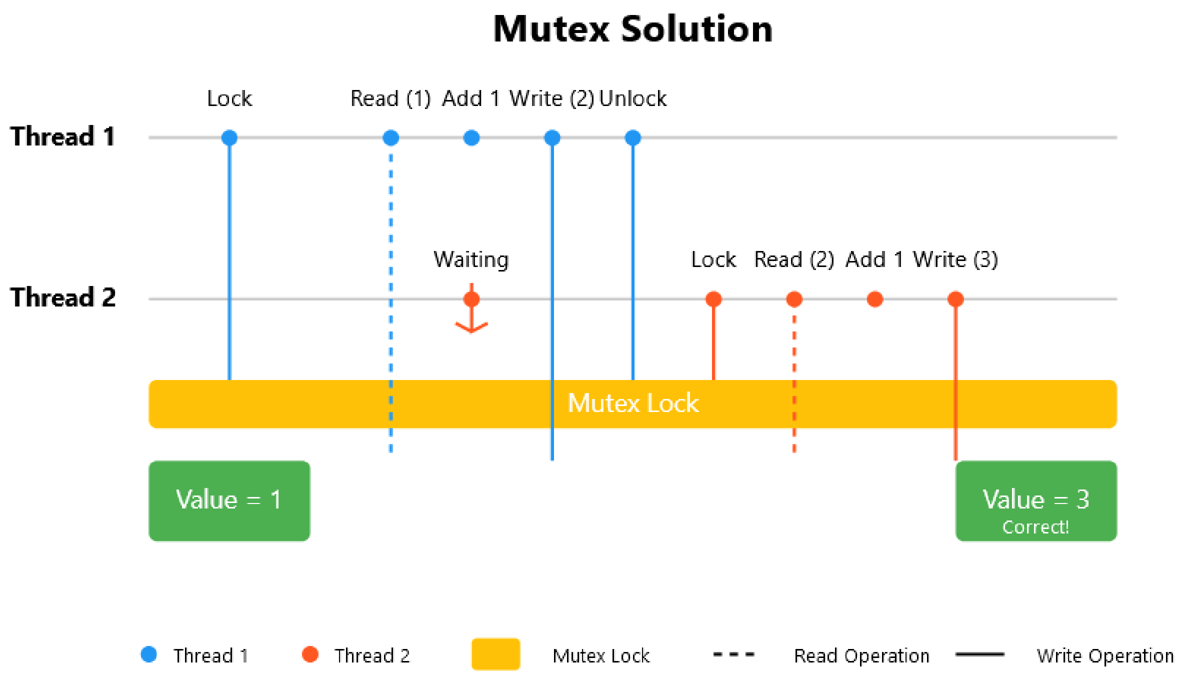
Reference Counting: Mutex Solution

- The Mutex Solution diagram shows how to prevent race conditions using mutual exclusion locks.



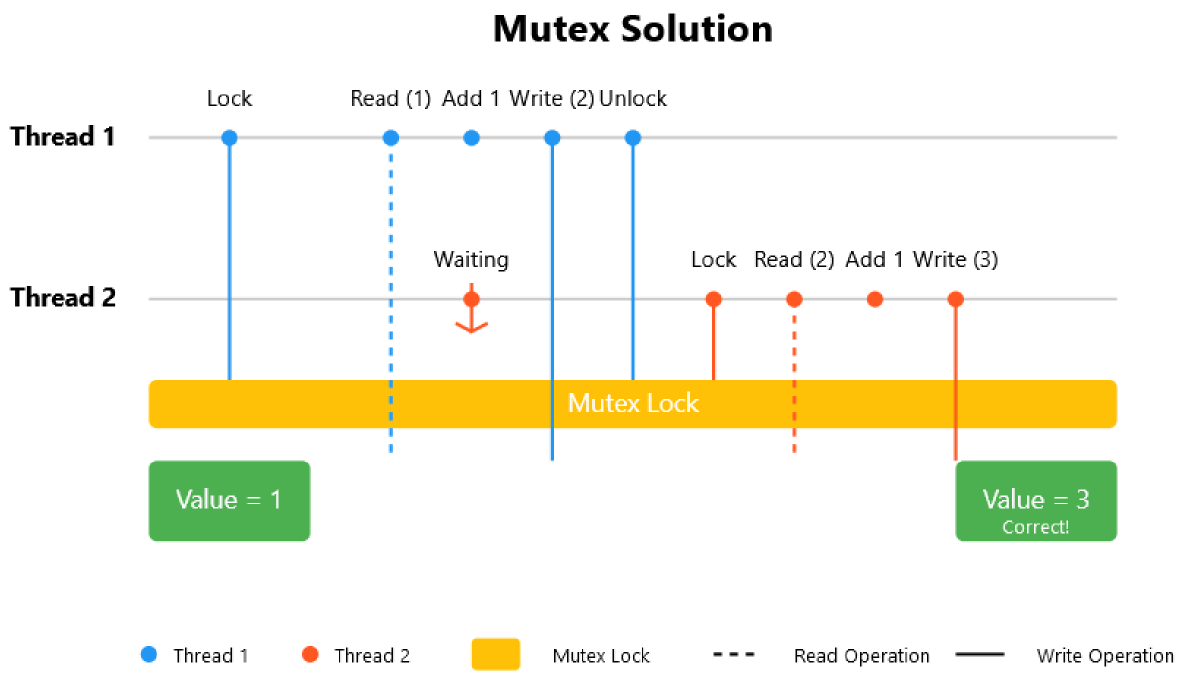
Reference Counting: Mutex Solution

- The Mutex Solution diagram shows how to prevent race conditions using mutual exclusion locks.
- Thread 1 (**blue**) first acquires the lock, reads value 1, adds 1, writes 2, then releases the lock. During this time, Thread 2 (red) must wait.



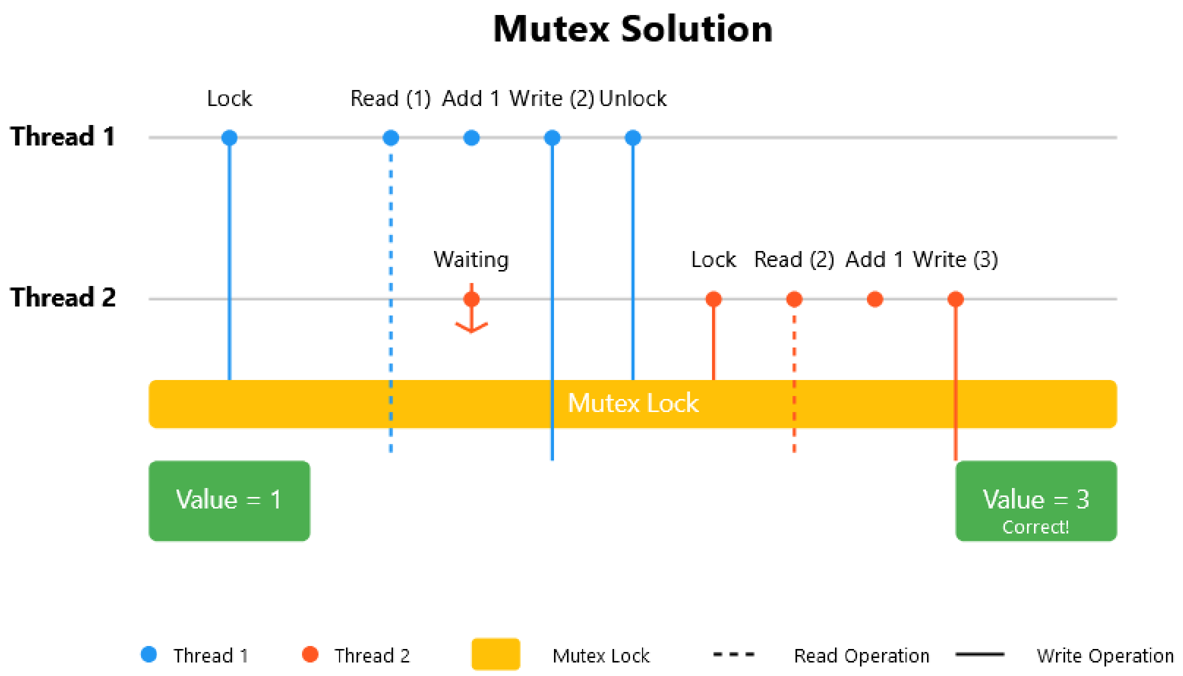
Reference Counting: Mutex Solution

- The Mutex Solution diagram shows how to prevent race conditions using mutual exclusion locks.
- Thread 1 (**blue**) first acquires the lock, reads value 1, adds 1, writes 2, then releases the lock. During this time, Thread 2 (red) must wait.
- Only after Thread 1 releases the lock can Thread 2 acquire it, read value 2, add 1, and write 3.



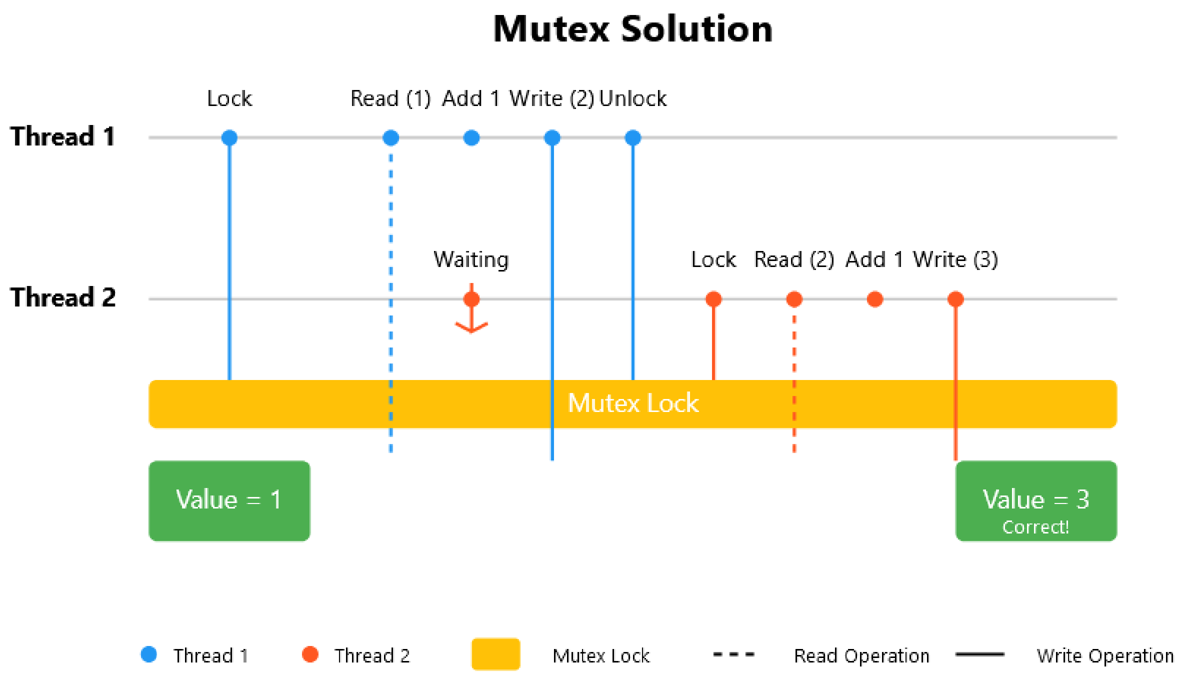
Reference Counting: Mutex Solution

- The Mutex Solution diagram shows how to prevent race conditions using mutual exclusion locks.
- Thread 1 (**blue**) first acquires the lock, reads value 1, adds 1, writes 2, then releases the lock. During this time, Thread 2 (red) must wait.
- Only after Thread 1 releases the lock can Thread 2 acquire it, read value 2, add 1, and write 3.
- The yellow bar represents the mutex lock that ensures only one thread can access the shared resource at a time.



Reference Counting: Mutex Solution

- The Mutex Solution diagram shows how to prevent race conditions using mutual exclusion locks.
- Thread 1 (**blue**) first acquires the lock, reads value 1, adds 1, writes 2, then releases the lock. During this time, Thread 2 (red) must wait.
- Only after Thread 1 releases the lock can Thread 2 acquire it, read value 2, add 1, and write 3.
- The **yellow bar** represents the mutex lock that ensures only one thread can access the shared resource at a time.
- This synchronization leads to the correct final value of **3**, solving the race condition problem.



2. Garbage Collection

Garbage Collection

- **Automatic Memory Management**: Detects and removes unused objects to optimize memory usage, eliminating manual memory handling.
- **Reference Counting**: Primary mechanism to track object usage through a count of active references.
- **Cyclic Garbage Collector**: Handles circular references effectively, preventing memory leaks from interdependent objects.
- **Three Generations**:
  - **Gen 0**: Newly created objects, collected most frequently
  - **Gen 1**: Objects surviving one collection cycle, collected less often
  - **Gen 2**: Long-lived objects, collected least frequently

Garbage Collection

1. **Generation 0 (Young)**
    - Contains newly created objects
    - Most frequently collected (every 700 objects)
    - Quick collection cycle
    - High turnover rate
2. **Generation 1 (Middle-Aged)**
    - Objects that survive Generation 0
    - Collected after 10 Gen 0 collections
    - Medium collection frequency
    - More stable objects
3. **Generation 2 (Old)**
    - Long-lived objects that survive Gen 1
    - Least frequently collected
    - Collected after 10 Gen 1 collections
    - Most stable objects

Generation 0 (Young): In function `calculate_totals()`, temporary variables like `temp_sum` and loop calculations are created and quickly discarded. These objects are short-lived and immediately collected after use. Think of a calculator that clears its memory after each calculation, efficiently managing temporary values.

Generation 1 (Middle-Aged): A shopping cart example: `cart_items` list survives multiple operations as items are added and removed. It persists longer than temporary variables but isn't permanent. Like a shopping basket that stays active during your shopping session but gets cleaned up after checkout.

Generation 2 (Old): Program settings stored in `SETTINGS` dictionary live throughout the entire program's life. Created once at startup with essential configurations like API URLs and timeouts. These objects rarely change and stay in memory until the program ends, similar to a restaurant's business hours - set once, used throughout.

Garbage Collection: Advantages and Disadvantages

Advantages

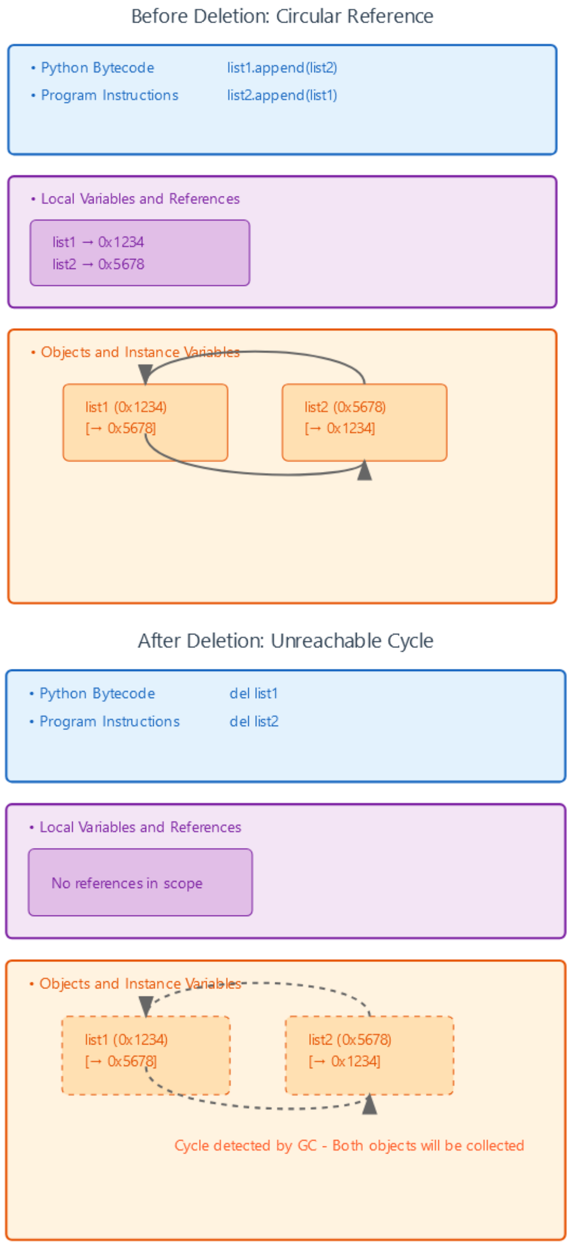
- Automatic memory handling
- No manual allocation/deallocation needed
- Prevents memory leaks
- Efficient memory utilization
- Developer-friendly
- Safer memory management
- Can be manually controlled (gc module)

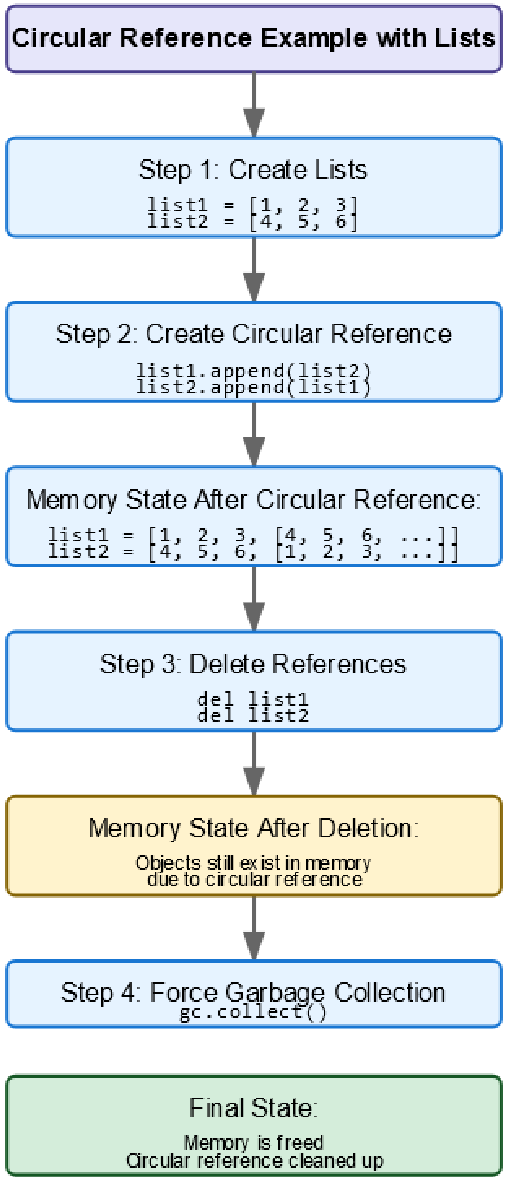
Disadvantages

- Less control over memory
- Performance overhead
- Memory usage can be higher
- Timing of cleanup not predictable
- Resource intensive for large applications

Garbage Collection: Force garbage collection

```python
# Import garbage collector module
import gc

list1 = [1, 2, 3]
list2 = [4, 5, 6]

# Create circular reference
list1.append(list2) # list1 references list2
list2.append(list1) # list2 references list1

# Remove references
del list1 # Reference count doesn't reach 0
del list2 # Due to circular reference #

Force garbage collection
gc.collect() # Cleans up circular reference
```





3. Memory Pooling

3. Memory Pooling

- An optimization technique used to manage memory allocation for small, frequently used objects.

- Pre-allocation of memory blocks for efficient reuse.

- Focuses on small objects like integers, strings, and frequently used types.

Memory Pooling: Key Characteristics

1. **Pre-allocation**: Memory blocks are reserved ahead of time to minimize the cost of repeated allocation.

2. **Fixed-size Pools**: Memory blocks are allocated in pools tailored to object sizes.

3. **Efficient Reuse**: Reuses blocks of memory instead of reallocating and deallocating memory repeatedly.

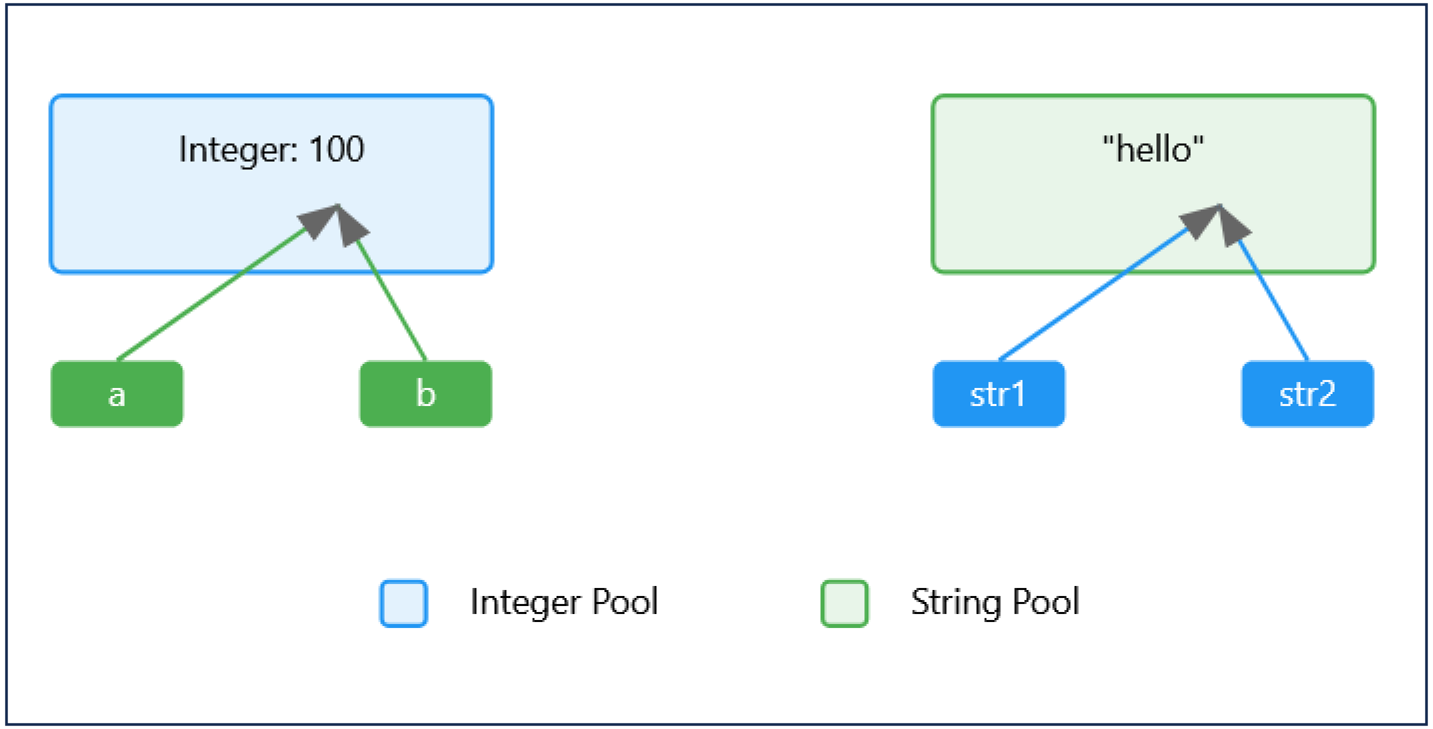
4. **Automatic Management**: Python handles memory pooling internally, no developer intervention needed.

Memory Pooling: Key Benefits

- **Reduces Overhead**: Pre-allocating memory reduces the need for frequent memory allocation and deallocation.
- **Minimizes Fragmentation**: Reuses memory blocks efficiently, reducing gaps in memory.
- **Improves Performance**: Faster memory allocation due to pre-allocation and reuse.
- **Optimized for Small Objects**: Especially beneficial for small integers, strings, and frequently used types.

Memory Pooling: Example

```python
# Memory Pooling Examples
# Integers (small numbers)
a = 100
b = 100
print("Same integers:")
print(a is b)  # True

# Short strings
str1 = "hello"
str2 = "hello"
print("\nShort strings:")
print(str1 is str2)  # True
```



4. Memory Interning

Memory Interning

- Python uses an optimization technique called **interning** to save memory and speed up execution.
- Immutable objects like **strings** and **integers** are cached for reusability.
- Interned objects with the same value share the same memory address.
- Reduces memory overhead by avoiding duplication of immutable objects.
- Enhances performance when comparing frequently used objects.

Memory Interning: : Advantages & Disadvantages

Advantages:

- Memory Efficiency: Saves memory by storing a single copy of immutable objects.

- Performance Boost: Faster comparison using object identity (is) instead of value equality (==).

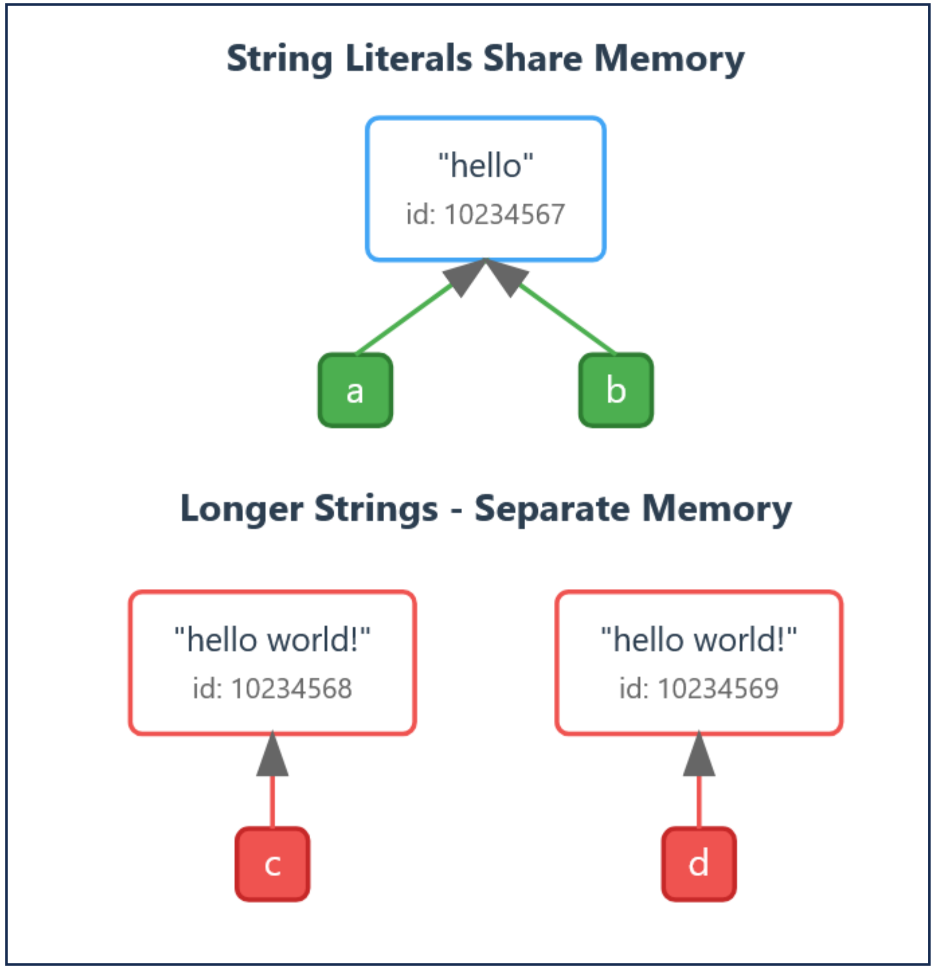
Disadvantages:

- Limited Scope: Works only for certain cases (e.g., small integers, short strings).

- Manual Control Required: Larger strings or non-standard cases need explicit interning.

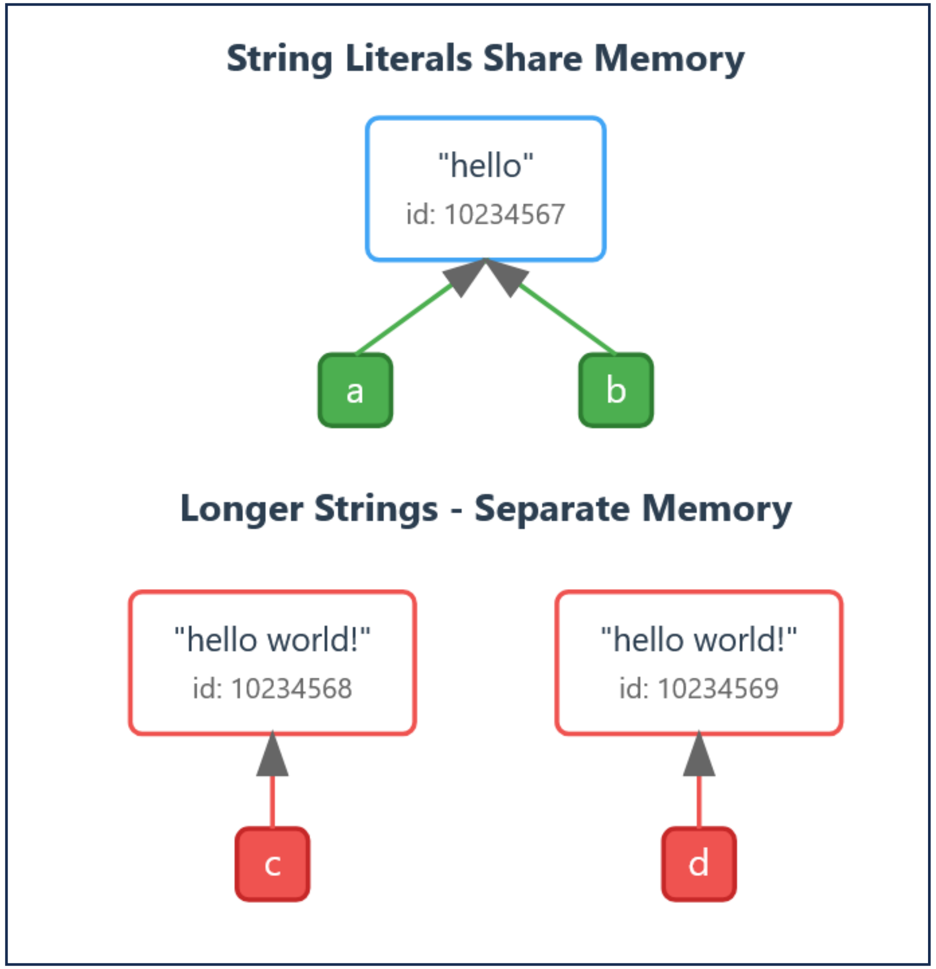
Memory Interning: : Example

```python
# String literals share memory location
a = "hello"
b = "hello"

print(id(a)) # Memory location for 'a' (e.g., 140712834927872)
print(id(b)) # Memory location for 'b' (same as 'a’)
print(a is b) # True - same memory location

# Longer strings - separate memory locations
c = "hello world!"
d = "hello world!"

print(id(c)) # Memory location for 'c' (e.g., 140712834928192)
print(id(d)) # Memory location for 'd' (different from 'c’)
print(c is d) # False - different memory locations
```



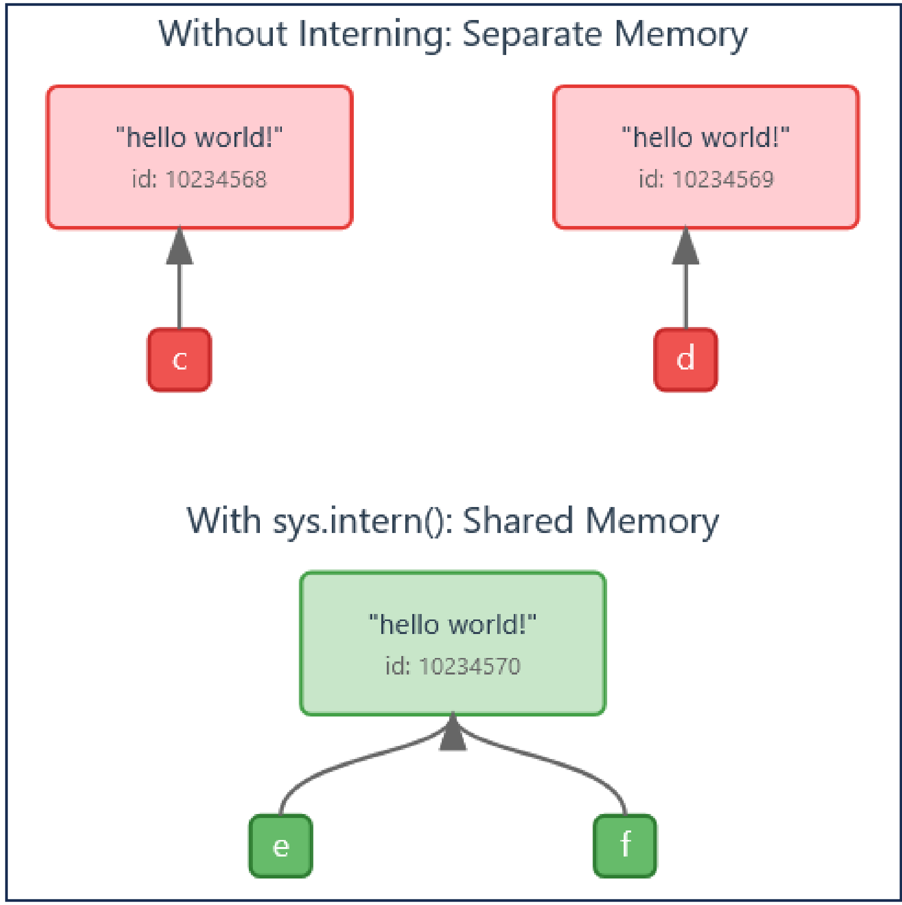
Memory Interning: : Example

```python
# String literals share memory location
a = "hello"
b = "hello"

print(id(a)) # Memory location for 'a' (e.g., 140712834927872)
print(id(b)) # Memory location for 'b' (same as 'a’)
print(a is b) # True - same memory location

# Longer strings - separate memory locations
c = "hello world!"
d = "hello world!"

print(id(c)) # Memory location for 'c' (e.g., 140712834928192)
print(id(d)) # Memory location for 'd' (different from 'c’)
print(c is d) # False - different memory locations
```



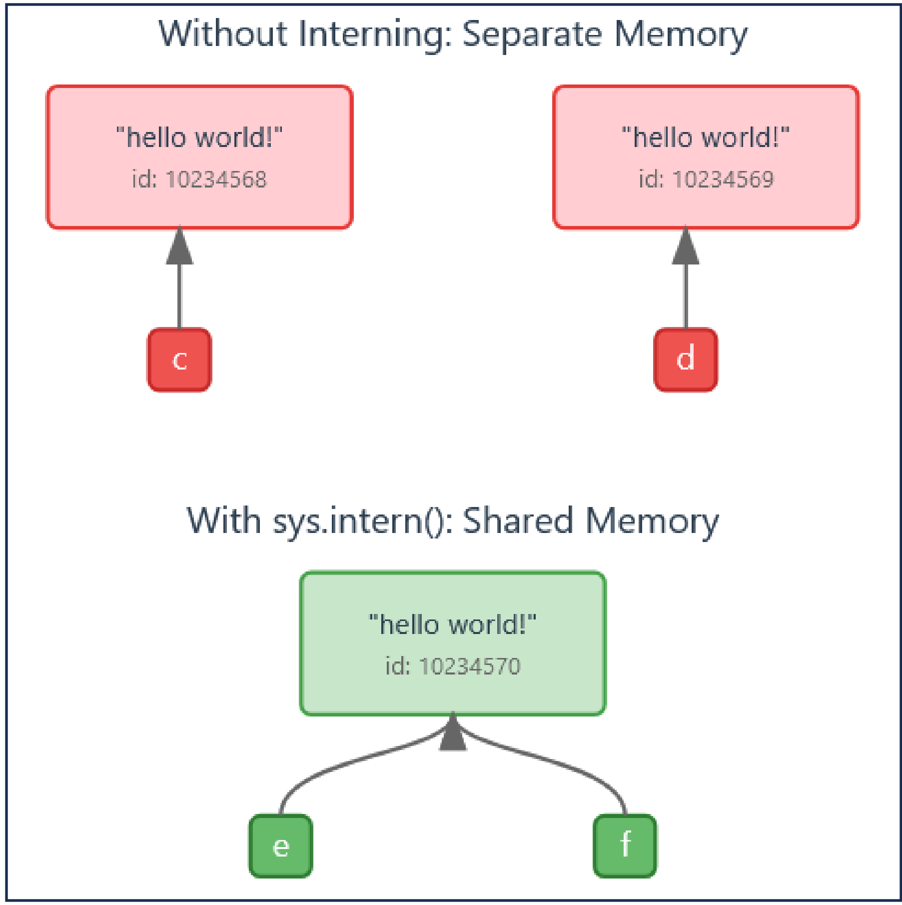
Memory Interning: : Manual Interning Example

```python
import sys
# Longer strings - separate memory locations
c = "hello world!"
d = "hello world!"

print(id(c)) # Memory location for 'c' (e.g., 140712834928192)
print(id(d)) # Memory location for 'd' (different from 'c’)
print(c is d) # False - different memory locations

# Manual interning for large strings
e = sys.intern("hello world!")
f = sys.intern("hello world!")
print(id(e))  # Memory location for 'e'
print(id(f))  # Memory location for 'f'
print(e is f)  # True - same memory location due to manual interning
```



Memory Interning: : Manual Interning Example

```
import sys
# Longer strings - separate memory locations
c = "hello world!"
d = "hello world!"

print(id(c)) # Memory location for 'c' (e.g., 140712834928192)
print(id(d)) # Memory location for 'd' (different from 'c’)
print(c is d) # False - different memory locations

# Manual interning for large strings
e = sys.intern("hello world!")
f = sys.intern("hello world!")
print(id(e))  # Memory location for 'e'
print(id(f))  # Memory location for 'f'
print(e is f)  # True - same memory location due to manual interning
```



5. Manual Memory Management

Manual Memory Management

Overview

- Python uses automatic memory management via a garbage collector.
- Manual memory management is rarely required but can improve performance and control.

When Manual Memory Management is Needed

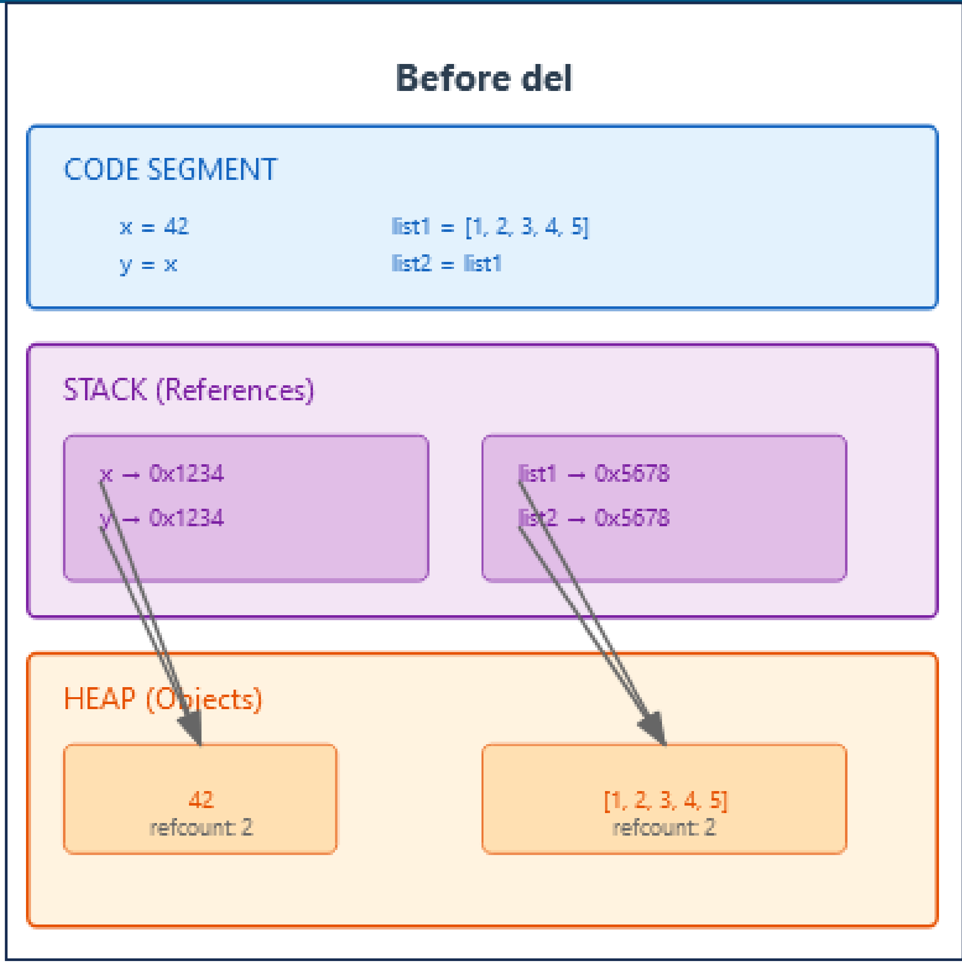
- **Optimizing Memory Usage**: For applications with large datasets or memory-heavy operations.
- **Controlling Object Lifecycle**: In resource-constrained systems requiring precise control. [creation (allocating memory efficiently), usage (optimizing resource consumption), and release (ensuring timely deallocation of memory to avoid resource exhaustion)].

Key Techniques

- **del Keyword**: Explicitly removes object references to free memory.
- **Weak References**: Use the weakref module to manage references without increasing the reference count.
- **Garbage Collection** (gc): Disable, enable, or manually trigger garbage collection.

Using del for Manual Object Deletion

```python
# Create variables
x = 42
y = x
list1 = [1, 2, 3, 4, 5]
list2 = list1

# Delete all references
del x, y # Integer references deleted
del list1, list2 # List references deleted
# All objects now eligible for garbage collection
```

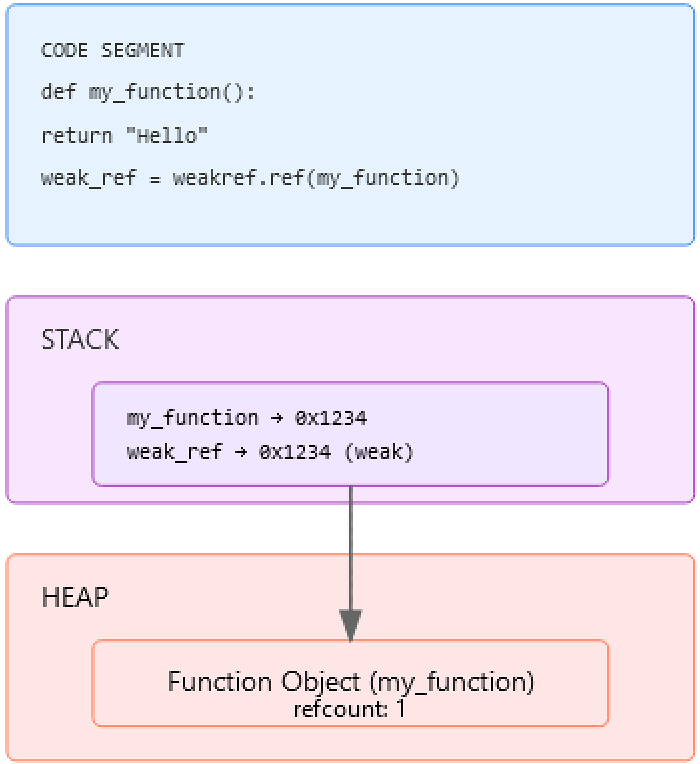


The **del** keyword allows developers to free up memory immediately by explicitly removing object references, improving performance, preventing memory leaks, and optimizing resource management in memory-intensive applications.

Using Weak References

```python
import weakref
import sys  # To check the reference count

def my_function():
    return "Hello‟

# Check the reference count before creating a weak reference
print(f"Reference count Before: {sys.getrefcount(my_function) - 1}")

# Create a weak reference to the function
weak_ref = weakref.ref(my_function)

# Check the reference count after creating a weak reference
print(f"Reference count after: {sys.getrefcount(my_function) - 1}")

# Access the function through the weak reference and call it
print(f"Accessing function through weak reference: {weak_ref()()}")  # Prints: Hello

# Delete the original reference to the function
del my_function

# Check the weak reference after the original function is deleted
print(f"Weak reference after deletion: {weak_ref()}")  # Prints: None
```

Before Deletion



Using Weak References

```python
import weakref
import sys  # To check the reference count

def my_function():
    return "Hello"

# Check the reference count before creating a weak reference
print(f"Reference count Before: {sys.getrefcount(my_function) - 1}")

# Create a weak reference to the function
weak_ref = weakref.ref(my_function)

# Check the reference count after creating a weak reference
print(f"Reference count after: {sys.getrefcount(my_function) - 1}")

# Access the function through the weak reference and call it
print(f"Accessing function through weak reference: {weak_ref()()}")  # Prints: Hello

# Delete the original reference to the function
del my_function

# Check the weak reference after the original function is deleted
print(f"Weak reference after deletion: {weak_ref()}")  # Prints: None
```

After deletion

CODE SEGMENT

```
del my_function
print(weak_ref()) # None
```

STACK

```
my_function → X (deleted)
weak_ref → None
```

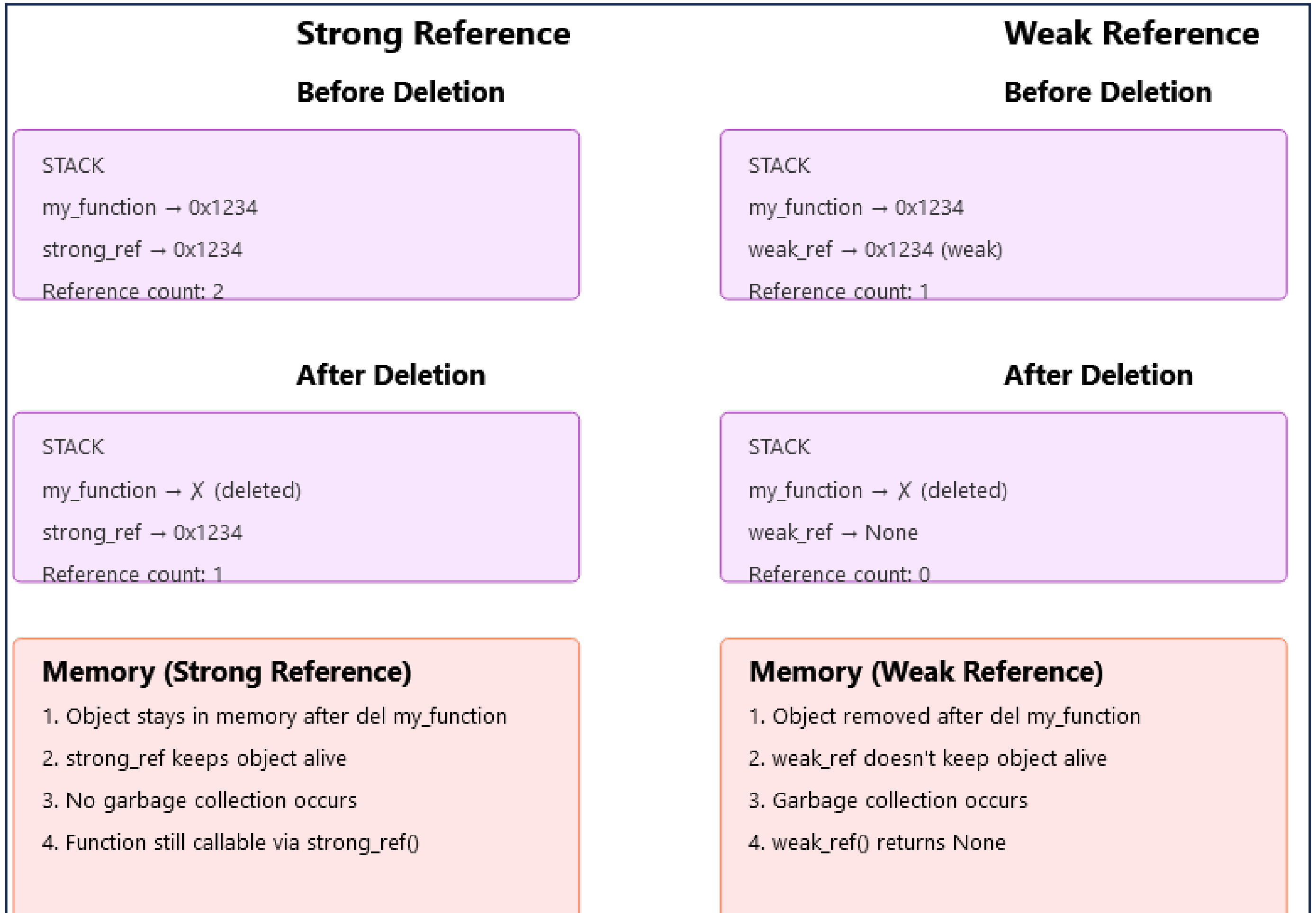
HEAP

Function Object (Garbage Collected)
refcount: 0

Using Weak References

Key Differences:

- Normal references increase reference count (Left refcount = 2).
- Weak references don't increase reference count (Right refcount = 1).
- Normal references prevent garbage collection, weak references don't.
- Normal references keep object alive, weak references allow collection when no strong references remain.



Using Garbage Collection (gc)

Manual garbage collection gives developers direct control over Python's garbage collector, allowing them to manage memory cleanup explicitly instead of relying on Python's automatic process.

Key Functions in gc

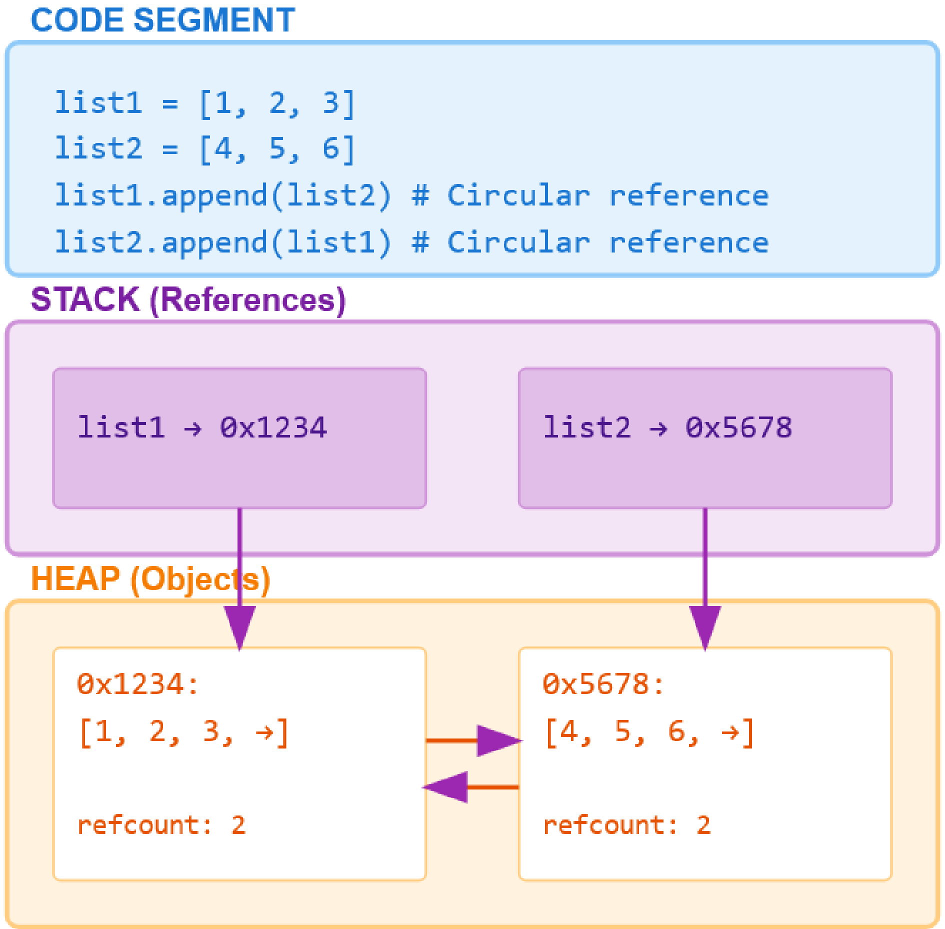
1. **`gc.disable()`**
   - Turns off automatic garbage collection.
   - Prevents interruptions during memory-intensive or time-sensitive operations.

1. **`gc.collect()`**
   - Triggers garbage collection manually.
   - Immediately frees up unused memory to avoid memory overflow or improve performance.

2. **`gc.enable()`**
   - Re-enables automatic garbage collection.
   - Restores Python's default memory management when manual control is no longer needed.

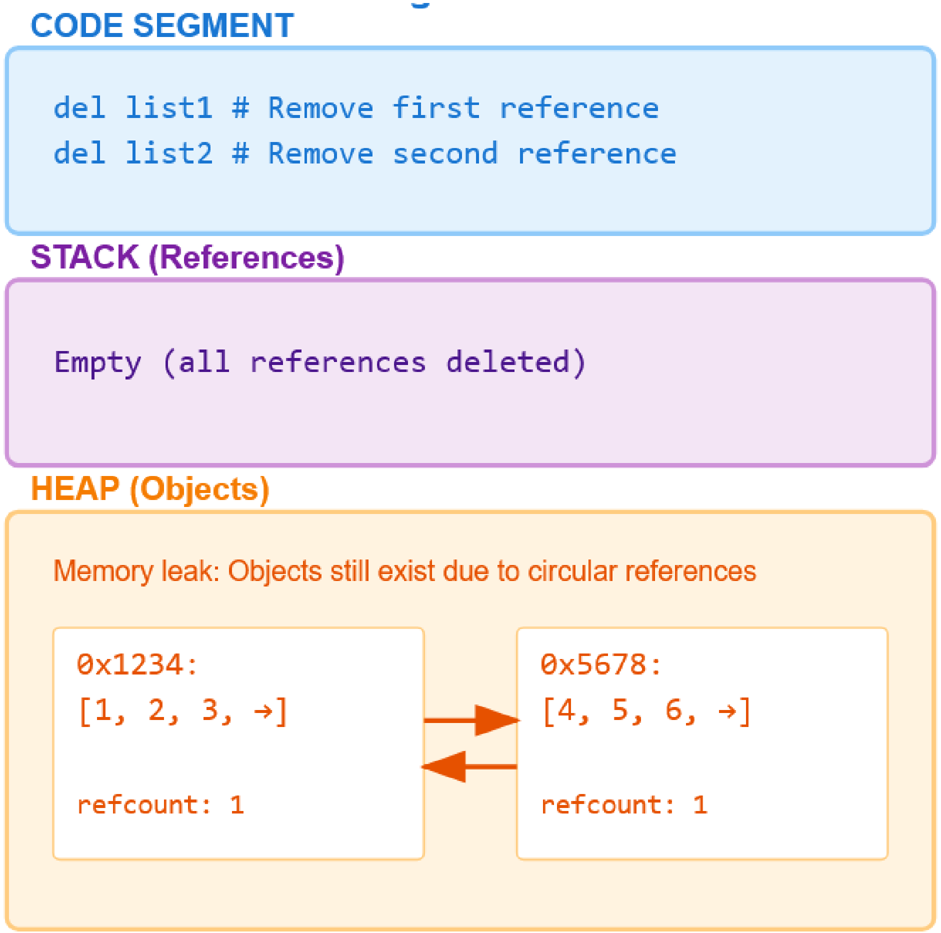
Using Garbage Collection (gc)

```python
# Import garbage collector module
import gc

# Disable automatic garbage collection
gc.disable()
print("Automatic garbage collection disabled.")

list1 = [1, 2, 3]
list2 = [4, 5, 6]
list1.append(list2) # list1 references list2
list2.append(list1) # list2 references list1

# Remove references
del list1 # Reference count doesn't reach 0
del list2 # Due to circular reference #

# Manually trigger garbage collection
gc.collect()  # Cleans up circular references
print("Garbage collection manually triggered and circular references cleaned up.")

# Re-enable automatic garbage collection
gc.enable()
print(" Automatic garbage collection re-enabled.")
```



Using Garbage Collection (gc)

```python
# Import garbage collector module
import gc

# Disable automatic garbage collection
gc.disable()
print("Automatic garbage collection disabled.")

list1 = [1, 2, 3]
list2 = [4, 5, 6]
list1.append(list2) # list1 references list2
list2.append(list1) # list2 references list1

# Remove references
del list1 # Reference count doesn't reach 0
del list2 # Due to circular reference #

# Manually trigger garbage collection
gc.collect()  # Cleans up circular references
print("Garbage collection manually triggered and circular references cleaned up.")

# Re-enable automatic garbage collection
gc.enable()
print(" Automatic garbage collection re-enabled.")
```



Using Garbage Collection (gc)

```python
# Import garbage collector module
import gc

# Disable automatic garbage collection
gc.disable()
print("Automatic garbage collection disabled.")

list1 = [1, 2, 3]
list2 = [4, 5, 6]
list1.append(list2) # list1 references list2
list2.append(list1) # list2 references list1

# Remove references
del list1 # Reference count doesn't reach 0
del list2 # Due to circular reference #

# Manually trigger garbage collection
gc.collect()  # Cleans up circular references
print("Garbage collection manually triggered and circular references cleaned up.")

# Re-enable automatic garbage collection
gc.enable()
print(" Automatic garbage collection re-enabled.")
```