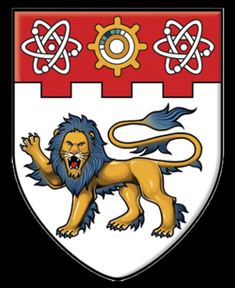


NANYANG TECHNOLOGICAL UNIVERSITY

Data Structures & Algorithms in Python

Memory Management Programs Explanations

Dr. Owen Noel Newton Fernando

College of Engineering

School of Computer Science and Engineering

Circular_References.py

Overview

```
# Create two empty lists
list1 = []                      # Create first empty list
list2 = []                      # Create second empty list

# Create circular reference
list1.append(list2)             # list1 now contains list2 as its element
list2.append(list1)             # list2 now contains list1 as its element

print("After creating circular reference:")
print(f"list1 = {list1}")       # Shows list1 containing list2
print(f"list2 = {list2}")       # Shows list2 containing list1

# Memory leak situation
del list1                       # Delete first reference but object still exists

del list2                       # Delete second reference but object still exists

# At this point:
# - Both lists still exist in memory
# - Each list contains a reference to the other
# - No external references exist to access these lists
# - Only garbage collector can free this memory
# - This is why reference counting alone isn't enough
```

Insights Gained

•Goal: Show how two Python lists can point to each other, forming a cycle (A contains B, B contains A). This creates objects that can’t be reached by your variables even after you del the names.

•Why it matters: Python normally frees memory when an object’s reference count hits zero. In a cycle, each object keeps the other alive, so reference counts never reach zero. Python’s garbage collector (GC) is needed to detect and clean such unreachable cycles.

•What you’ll see: The print lines display nested lists like list1 -> [[...]], revealing the back-and-forth references (often shown with [...]). After del, the lists still occupy memory until the GC runs.

Circular_References.py

Overview

- list1 = [] # Create first empty list
- list2 = [] # Create second empty list

Insights Gained

Make an empty list. The name list1 now references (points to) a list object in memory.

Make another empty list; list2 points to a different list object.

Circular_References.py

Overview

- # Create circular reference
- list1.append(list2) # list1 now contains list2 as its element
- list2.append(list1) # list2 now contains list1 as its element

- print("After creating circular reference:")
- print(f"list1 = {list1}") # Shows list1 containing list2
- print(f"list2 = {list2}") # Shows list2 containing list1

Insights Gained

Put list2 inside list1.

Now list1 contains a reference to list2.

Put list1 inside list2.

Now each list contains the other → a circular reference.

A label so the output is easy to recognize.

Printing a list triggers Python to show its contents.

Because lists point to each other, Python uses [...] to avoid infinite printing.

Same idea for list2.

Circular_References.py

Overview

- # Memory leak situation
- del list1
- # Delete first reference but object still exists
- del list2
- # Delete second reference but object still exists

Insights Gained

Remove the name list1.

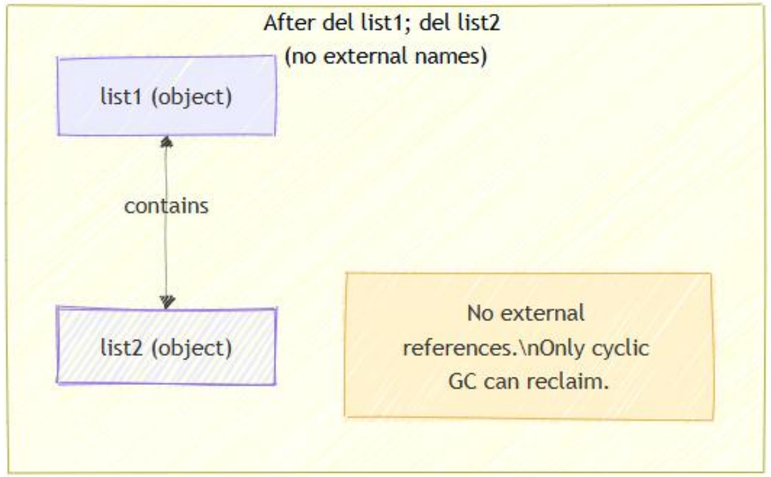
The underlying list object still exists because list2 is still pointing to it.

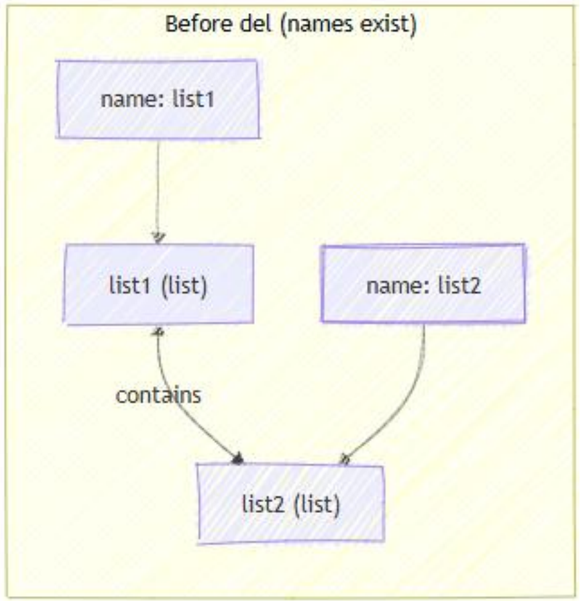
Remove the name list2.

Now no variable names refer to either list, but each list still points to the other.

They’re unreachable by you, but still in memory until GC collects them.

Circular_References.py





Force_Garbage_Collection.py

Overview

```python
# Import garbage collector module for manual collection
import gc

print("Initial garbage count:", gc.get_count())

# Create two lists
list1 = [1, 2, 3]
list2 = [4, 5, 6]
print("\nAfter creating lists garbage count:", gc.get_count())

# Create circular reference
list1.append(list2)
list2.append(list1)
print("\nAfter circular reference garbage count:", gc.get_count())
print("list1:", list1)
print("list2:", list2)

# Remove direct references
del list1

del list2
print("\nAfter deletion garbage count:", gc.get_count())

# Force garbage collection
collected = gc.collect()  # Returns number of objects collected
print("\nGarbage collector collected", collected, "objects.")
print("Final garbage count:", gc.get_count())
```

Insights Gained

GC module: Gives you tools to look at and control Python's automatic garbage collector.

gc.get_count(): Shows how many collection-triggering allocations happened in three "generations" (young → old). Values change as your program creates temporary objects.

Circular reference again: We build a cycle like before. Deleting variable names doesn't free the objects because the lists still reference each other.

gc.collect(): Manually asks Python to scan for unreachable cycles and free them. It returns how many objects were collected.

Force_Garbage_Collection.py

Overview

- # Import garbage collector module for manual collection
- import gcprint("Initial garbage count:", gc.get_count())

Insights Gained

Load Python’s garbage collector tools.

Peek at internal counters for GC generations before we do anything. Useful for comparison.

Force_Garbage_Collection.py

Overview

- # Create two lists
- list1 = [1, 2, 3]
- list2 = [4, 5, 6]
- print("\nAfter creating lists garbage count:", gc.get_count())

Insights Gained

Create two normal lists.

More allocations happened, so counters may rise.

Force_Garbage_Collection.py

Overview

- #Create circular reference
- list1.append(list2)
- list2.append(list1)
- print("\nAfter circular reference garbage count:", gc.get_count())
- print("list1:", list1)
- print("list2:", list2)

Insights Gained

Create a cycle of references between the lists.

Show the nested structure; Python abbreviates cycles with [...].

Force_Garbage_Collection.py

Overview

- # Remove direct references
- del list1
- del list2
- print("\nAfter deletion garbage count:", gc.get_count())

Insights Gained

Remove the variable names.

Objects still exist because they point to each other.

Counters help illustrate that memory hasn’t been reclaimed yet.

Force_Garbage_Collection.py

Overview

- # Force garbage collection
- collected = gc.collect()
- # Returns number of objects collected
- print("\nGarbage collector collected", collected, "objects.")
- print("Final garbage count:", gc.get_count())

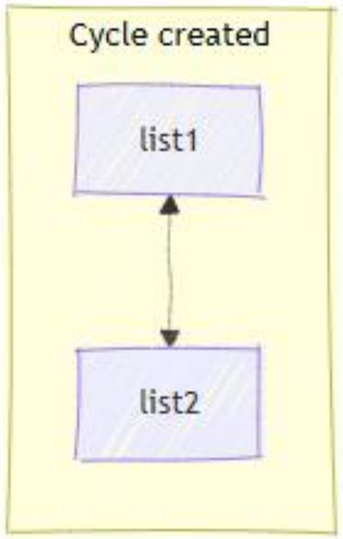
Insights Gained

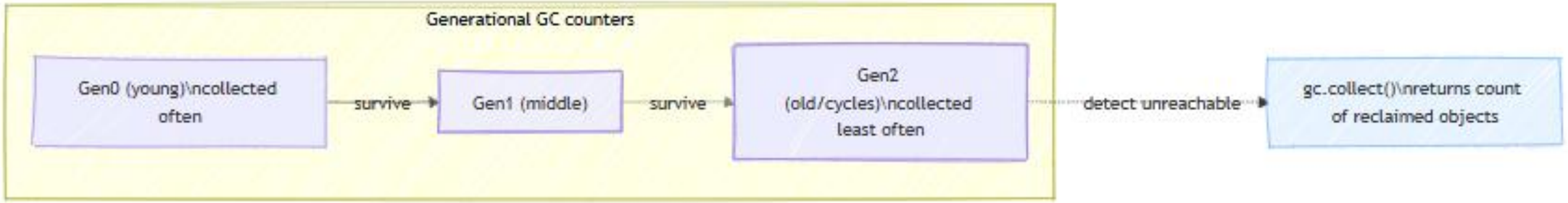
Force a GC cycle that detects unreachable objects (like our cycle) and frees them.

Returns how many were collected.

After collection, counts often drop/stabilize.

Force_Garbage_Collection.py





Gen0-to-Gen2_Garbage_Collection.py

Overview

```python
# Import garbage collector module for manual collection
import gc


print("Initial garbage count:", gc.get_count())
# Shows (gen0, gen1, gen2) counts before creating lists


# Create two lists
list1 = [1, 2, 3]
list2 = [4, 5, 6]
print("
After creating lists garbage count:", gc.get_count())
# Count increases as new objects created


# Create circular reference
list1.append(list2)    # list1 now contains [1, 2, 3, [4, 5, 6]]
list2.append(list1)    # list2 now contains [4, 5, 6, [1, 2, 3, [...]]]
print("
After circular reference garbage count:", gc.get_count())
print("list1:", list1)
print("list2:", list2)
```

Insights Gained

gc.get_count() shows allocation pressure in three generations.

Gen0-to-Gen2_Garbage_Collection.py

Overview

```
# Remove direct references
del list1  # Deleting reference but objects still exist
del list2  # Due to circular reference
print("
After deletion garbage count:", gc.get_count())
# Objects still in memory due to circular reference


# Force garbage collection
collected = gc.collect()  # Returns number of objects collected
print("
Garbage collector collected", collected, "objects.")
print("Final garbage count:", gc.get_count())
# Final count explanation:
# - 41 (or other number) objects in gen0 are Python's internal objects
#   These include:
#   - Frame objects for function calls
#   - Temporary objects from print statements
#   - Internal references for Python's runtime
#   - System-level temporary objects
# - 0 in gen1 means all middle-aged objects collected
# - 0 in gen2 means all old objects and circular references collected
```

Insights Gained

A circular reference remains after del until gc.collect() runs and frees it.

Gen0-to-Gen2_Garbage_Collection.py

Overview

- # Import garbage collector module for manual collection
- import gc

- print("Initial garbage count:", gc.get_count())
- # Shows (gen0, gen1, gen2) counts before creating lists

- # Create two lists
- list1 = [1, 2, 3]
- list2 = [4, 5, 6]
- print("After creating lists garbage count:", gc.get_count())
- # Count increases as new objects created

Insights Gained

Loads garbage-collector tools.

Returns a tuple (gen0, gen1, gen2) of allocation counters.

Allocates objects → counters typically rise.

Gen0-to-Gen2_Garbage_Collection.py

Overview

- # Create circular reference

- list1.append(list2) # list1 now contains [1, 2, 3, [4, 5, 6]]
- list2.append(list1) # list2 now contains [4, 5, 6, [1, 2, 3, [...]]]

- print("After circular reference garbage count:", gc.get_count())

- print("list1:", list1)
- print("list2:", list2)

- # Remove direct references
- del list1 # Deleting reference but objects still exist
- del list2 # Due to circular reference
- print("After deletion garbage count:", gc.get_count())

- # Objects still in memory due to circular reference

Insights Gained

Build a cycle intentionally.

Names removed but objects still mutually reachable → need GC.

Gen0-to-Gen2_Garbage_Collection.py

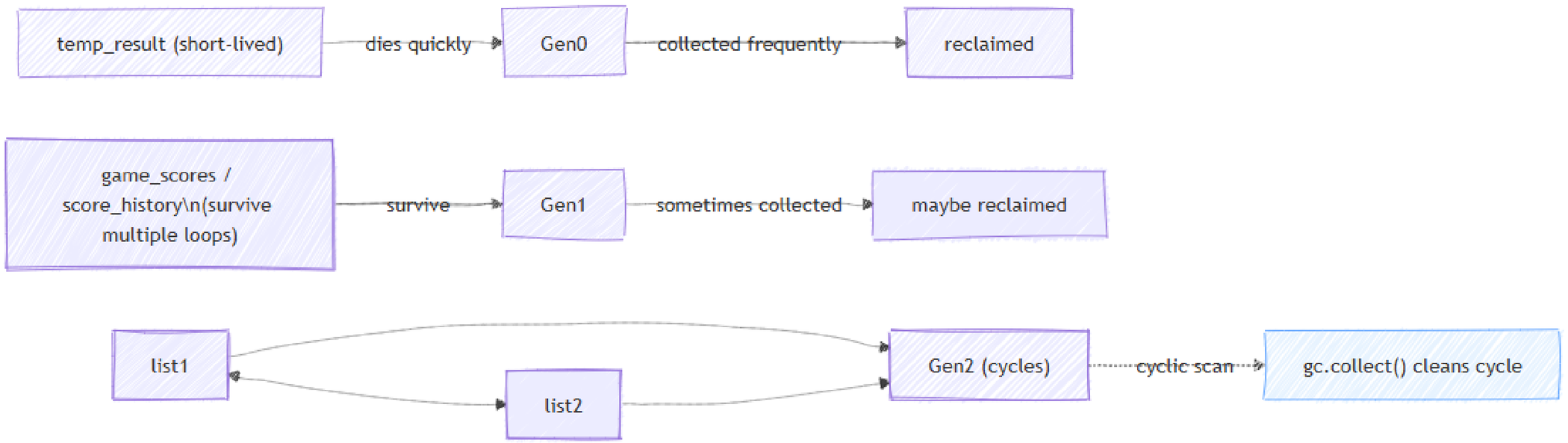
Overview

- # Force garbage collection
- collected = gc.collect()
- # Returns number of objects collected
- print("Garbage collector collected", collected, "objects.")
- print("Final garbage count:", gc.get_count())

Insights Gained

Forces a full collection; returns number of objects reclaimed.

Gen0-to-Gen2_Garbage_Collection.py



Reference_Counting.py

Overview

```python
# Import sys module to access system-specific functions
import sys

x = [1, 2, 3]
print(f"After creating x: {sys.getrefcount(x) - 1} reference(s)")

y = x
print(f"After creating y: {sys.getrefcount(x) - 1} reference(s)")

z = x
print(f"After creating z: {sys.getrefcount(x) - 1} reference(s)")

del x
print(f"After deleting x: {sys.getrefcount(y) - 1} reference(s)")

del y
print(f"After deleting y: {sys.getrefcount(z) - 1} reference(s)")

del z
# Now the object has no references and is eligible for garbage collection
```

Insights Gained

Reference counting is Python’s first line of defense for memory management.
Each object tracks how many references point to it.

sys.getrefcount(obj) shows the current reference count plus one (because the function itself temporarily holds a reference), so we subtract 1 to get the real count.

As we assign more variables to the same list, the count goes up; as we del names, it goes down.

When it hits zero, the object becomes collectible.

Reference_Counting.py

Overview

- # Import sys module to access system-specific functions
- import sysx = [1, 2, 3]
- print(f"After creating x: {sys.getrefcount(x) - 1} reference(s)")

Insights Gained

We need sys.getrefcount() to read reference counts.

Create a list and bind it to x. Refcount starts at 1.

Get the actual number of references to the list (subtract the temporary reference that getrefcount adds).

Reference_Counting.py

Overview

- y = x
- print(f"After creating y: {sys.getrefcount(x) - 1} reference(s)")
- z = x
- print(f"After creating z: {sys.getrefcount(x) - 1} reference(s)")

Insights Gained

y now points to the same list; refcount increases.

Another name to the same list; refcount increases again.

Reference_Counting.py

Overview

- del x
- print(f"After deleting x: {sys.getrefcount(y) - 1} reference(s)")
- del y
- print(f"After deleting y: {sys.getrefcount(z) - 1} reference(s)")
- del z
- # Now the object has no references and is eligible for garbage collection

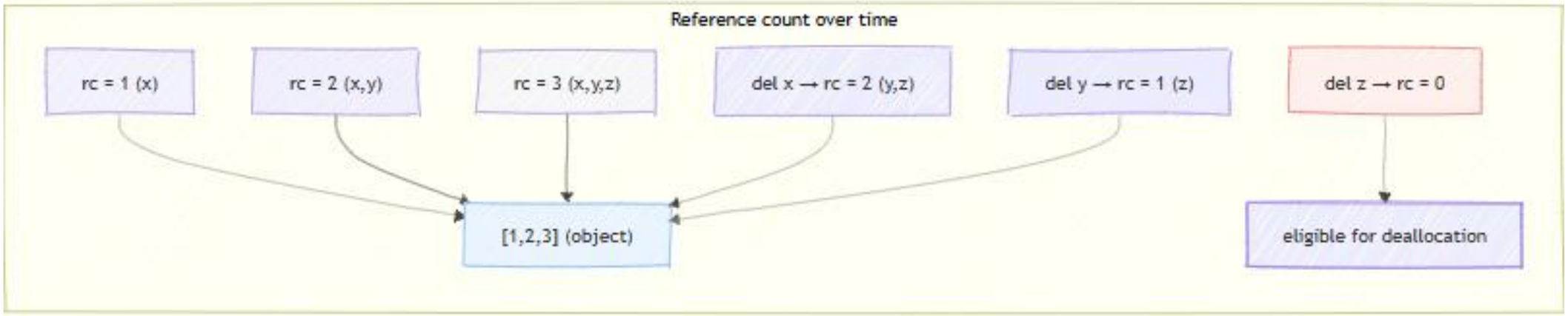
Insights Gained

Remove one reference; list stays alive because y and z still point to it.

Remove another reference; only z remains.

Now refcount becomes 0 → the list is eligible for immediate deallocation (or soon after).

Reference_Counting.py



Immutable_Integer_and_Strings.py

Overview

```python
# Integer immutability
x = 42

y = x
print(f"Before: x = {x}, y = {y}")
print(f"Before: id(x) = {id(x)}, id(y) = {id(y)}")

x += 1  # x becomes 43, y stays 42
print(f"\nAfter: x = {x}, y = {y}")
print(f"After: id(x) = {id(x)}, id(y) = {id(y)}")

# String immutability
print("\n" + "-" * 20 + "\n")

s1 = "hello"
s2 = "hello"
print(f"Initially both strings same: id(s1) = {id(s1)}, id(s2) = {id(s2)}")

s1 = "hello!"  # Creates new string
print(f"After change: id(s1) = {id(s1)}, id(s2) = {id(s2)}")
```

Insights Gained

Immutable objects (like int and str) never change in place.

Any “change” creates a new object with a new identity (id(...)).

Assigning y = x makes both names point to the same integer 42.

After x += 1, Python makes a new integer 43 and rebinds x to it.

y still points to the old 42.

Short identical strings may share memory under the hood (an optimization called interning), so id(s1) and id(s2) can match.
Reassigning s1 = "hello!" gives s1 a new object and a new id.

Immutable_Integer_and_Strings.py

Overview

- # Integer immutability
- x = 42
- y = x
- print(f"Before: x = {x}, y = {y}")
- print(f"Before: id(x) = {id(x)}, id(y) = {id(y)}")

Insights Gained

Two names pointing to the same integer object 42.

id is like an object’s “address”. Same value → same underlying object

Immutable_Integer_and_Strings.py

Overview

- x += 1
- # x becomes 43, y stays 42
- print(f"\nAfter: x = {x}, y = {y}")
- print(f"After: id(x) = {id(x)}, id(y) = {id(y)}")
- # String immutabilityprint("\n" + "-" * 20 + "\n")

Insights Gained

With immutables, this creates a new integer 43 and rebinds x.

y remains 42.

Confirms x and y now have different ids.

Immutable_Integer_and_Strings.py

Overview

- s1 = "hello“
- s2 = "hello“
- print(f"Initially both strings same: id(s1) = {id(s1)}, id(s2) = {id(s2)}")
- s1 = "hello!“
- # Creates new stringprint(f"After change: id(s1) = {id(s1)}, id(s2) = {id(s2)}")

Insights Gained

Short identical strings are often interned (Python may reuse the same object), so ids can match.

New string object; s2 still points to the original "hello".

Immutable_Integer_and_Strings.py

Overview

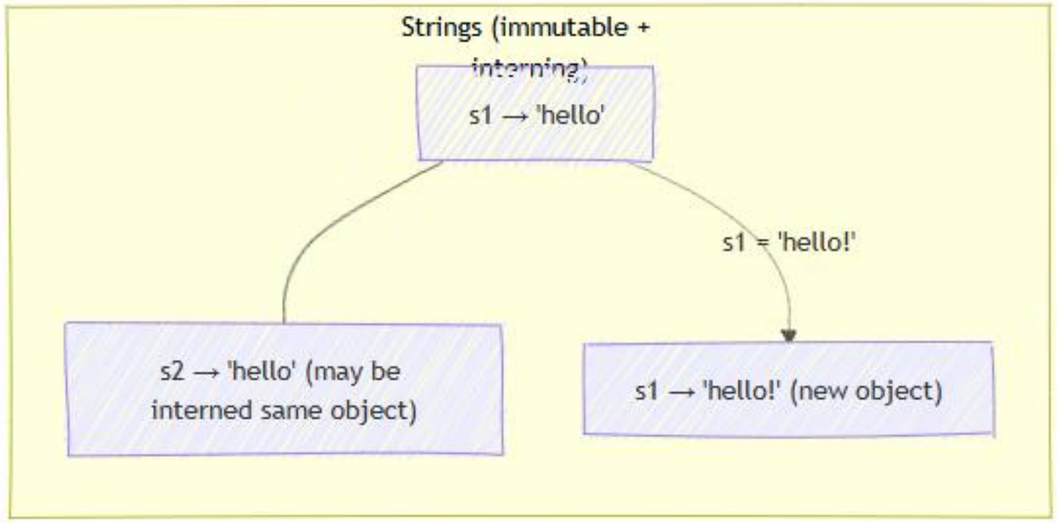
- s1 = "hello“
- s2 = "hello“
- print(f"Initially both strings same: id(s1) = {id(s1)}, id(s2) = {id(s2)}")
- s1 = "hello!“
- # Creates new stringprint(f"After change: id(s1) = {id(s1)}, id(s2) = {id(s2)}")

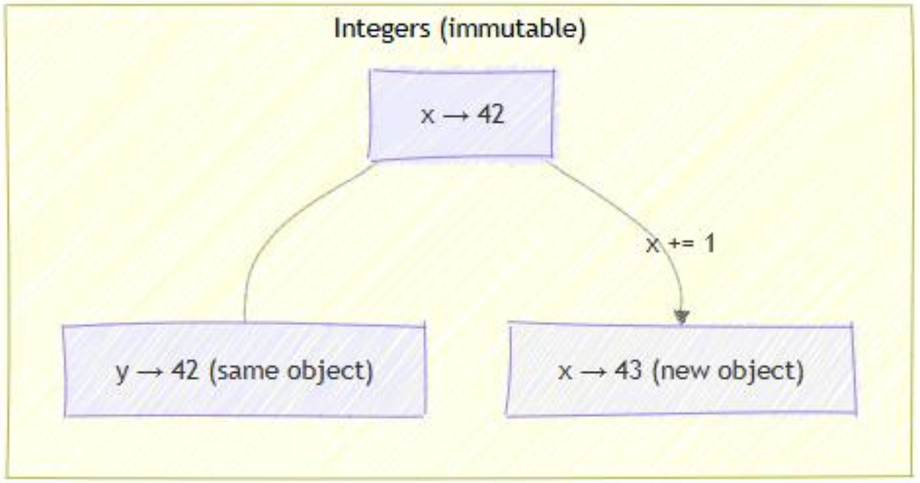
Insights Gained

Short identical strings are often interned (Python may reuse the same object), so ids can match.

New string object; s2 still points to the original "hello".

Immutable_Integer_and_Strings.py





Name-Reference-Object.py

Overview

```
pi = 3.14159  # Global variable with value shown in heap

# STACK SEGMENT
x = 42        # Local variable pointing to 42 in heap
text = "hello"
list1 = [1, 2, 3]  # Local variable pointing to list in heap

print(f"x at memory location: {id(x)}")  # Will show something like 0x1234
print(f"list1 at memory location: {id(list1)}")  # Will show something like
0x5678
print(f"pi at memory location: {id(pi)}")  # Will show something like 0x9999
```

Insights Gained

Names vs Objects: Names (variables) live on the stack of the current scope; they reference objects on the heap (e.g., the integer 42 , list [1,2,3]).

id(obj) shows a unique identity for the object during its lifetime (CPython shows an address-like integer). Same id ⇒ same object.

Mutability: Rebinding a name points to a new object; mutating a container changes the same object.

Name-Reference-Object.py

Overview

- pi = 3.14159 # Global variable with value shown in heap

- # STACK SEGMENT
- x = 42
- # Local variable pointing to 42 in heap
- text = "hello"
- list1 = [1, 2, 3
-] # Local variable pointing to list in heap

Insights Gained

Bind pi to a floating-point object stored on the heap.

Bind x to an integer object 42 in the heap.

Bind text to a string object (immutable).

Create a list object (mutable) and bind list1 to it.

Name-Reference-Object.py

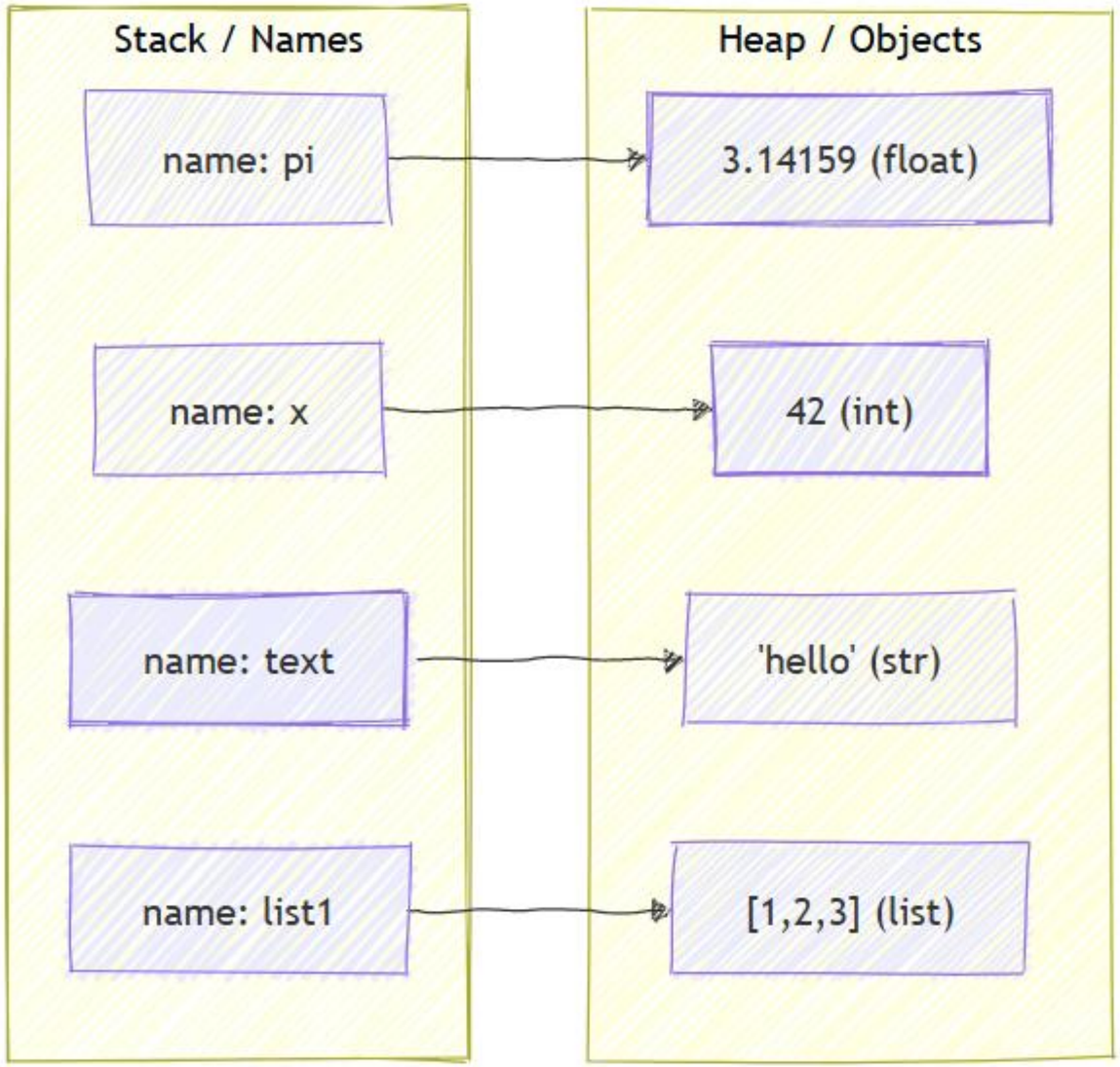
Overview

- print(f"x at memory location: {id(x)}")
- # Will show something like 0x1234
- print(f"list1 at memory location: {id(list1)}")
- # Will show something like 0x5678
- print(f"pi at memory location: {id(pi)}")
- # Will show something like 0x9999

Insights Gained

Show each object’s identity (helps visualize name→object mapping).

Name-Reference-Object.py



Memory_Pooling.py

Overview

```
# Memory Pooling Example
print("Integer Memory Pooling:")
# Python reuses integers between -5 and 256
a = 100
b = 100
print(f"a = {a}, b = {b}")
print(f"Are a and b same object? {a is b}")  # True - same object reused
print(f"Memory location of a: {id(a)}")
print(f"Memory location of b: {id(b)}")  # Same as a's location

print("\nString Memory Pooling (String Interning):")
# Python interns (reuses) short strings
str1 = "hello"
str2 = "hello"
print(f"str1 = {str1}, str2 = {str2}")
print(f"Are str1 and str2 same object? {str1 is str2}")  # True - same string reused
print(f"Memory location of str1: {id(str1)}")
print(f"Memory location of str2: {id(str2)}")  # Same as str1's location

# Counter examples (no pooling)
print("\nNo Pooling Examples:")
# Large integers aren't pooled
x = 257
y = 257
print(f"x = {y}, y = {y}")
print(f"Are large integers (257) pooled? {x is y}")  # False - separate objects

# Long or complex strings aren't automatically pooled
str3 = "hello world!"
str4 = "hello world!"
print(f"str3 = {str3}, str4 = {str4}")
print(f"Are longer strings pooled? {str3 is str4}")  # False - separate objects
```

Insights Gained

Show each object’s identity (helps visualize name→object mapping). Pooling (CPython optimization) reuses some small/common immutable objects to save memory and speed comparisons.

Small ints in [-5, 256] are typically preallocated and reused. Comparing with is often returns True in this range.

Short strings may be interned automatically; longer strings usually aren’t (unless manually interned).

Identity vs equality: is checks same object; == checks same value.

Memory_Pooling.py

Overview

- # Memory Pooling Example
- print("Integer Memory Pooling:")

- # Python reuses integers between -5 and 256
- a = 100
- b = 100

- print(f"a = {a}, b = {b}")
- print(f"Are a and b same object? {a is b}") # True - same object reused
- print(f"Memory location of a: {id(a)}")
- print(f"Memory location of b: {id(b)}") # Same as a's location
- print(“ String Memory Pooling (String Interning):")

Insights Gained

Both names often point to the same small-int object; pooling saves memory.

True → same identity.

Memory_Pooling.py

Overview

- # Python interns (reuses) short strings

- str1 = "hello"
- str2 = "hello"
- print(f"str1 = {str1}, str2 = {str2}")
- print(f"Are str1 and str2 same object? {str1 is str2}")
- # True - same string reused

- print(f"Memory location of str1: {id(str1)}")
- print(f"Memory location of str2: {id(str2)}")
- # Same as str1's location

- # Counter examples (no pooling)
- print("No Pooling Examples:")
- # Large integers aren't pooled

- x = 257
- y = 257
- print(f"x = {y}, y = {y}")
- print(f"Are large integers (257) pooled? {x is y}")
- # False - separate objects

Insights Gained

values outside the small-int pool often create separate objects.

Often False because 257 isn’t pooled.

Memory_Pooling.py

Overview

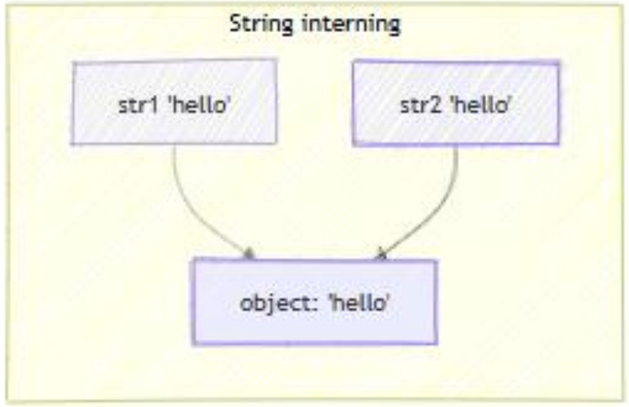
- # Long or complex strings aren't automatically pooled
- str3 = "hello world!"
- str4 = "hello world!"
- print(f"str3 = {str3}, str4 = {str4}")
- print(f"Are longer strings pooled? {str3 is str4}")
- # False - separate objects

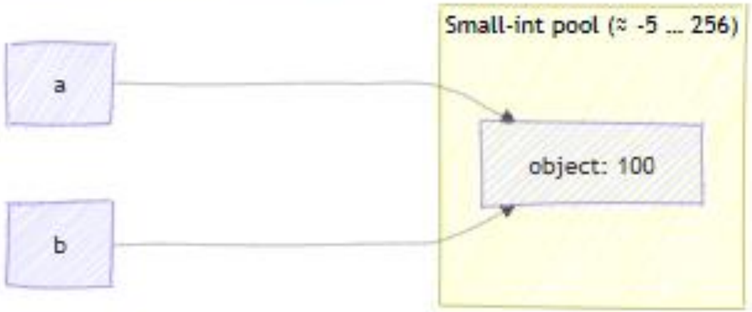
Insights Gained

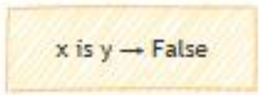
Short literals may be interned → same object.

Longer strings are usually distinct (no automatic interning).

Memory_Pooling.py







Memory_Interning.py

Overview

```python
# Memory Interning Example
import sys

# Demonstrate automatic vs manual string interning
print("Without Manual Interning:")
# Longer strings - Python doesn't automatically intern them
c = "hello world!"
d = "hello world!"

print(f"Memory location of c: {id(c)}")  # First memory location
print(f"Memory location of d: {id(d)}")  # Different memory location
print(f"Are c and d the same object? {c is d}")  # False - different objects
print(f"Do c and d have same value? {c == d}")   # True - same value, different objects

print("\nWith Manual Interning using sys.intern():")
# Manually intern longer strings to force memory sharing
e = sys.intern("hello world!")
f = sys.intern("hello world!")

print(f"Memory location of e: {id(e)}")  # First memory location
print(f"Memory location of f: {id(f)}")  # Same memory location as e
print(f"Are e and f the same object? {e is f}")  # True - same object due to interning
print(f"Do e and f have same value? {e == f}")   # True - same value
```

Insights Gained

Interning stores one shared copy of a string so equal strings can share memory.

Python does some automatic interning (identifiers, some literals), but not for all strings—especially longer ones.

sys.intern(s) forces interning so repeated occurrences of s point to the same object.

Memory_Interning.py

Overview

- # Memory Interning Example
-
- import sys

- # Demonstrate automatic vs manual string interning
  print("Without Manual Interning:")

- # Longer strings - Python doesn't automatically intern them

- c = "hello world!"
- d = "hello world!"

- print(f"Memory location of c: {id(c)}")
- print(f"Memory location of d: {id(d)}")
- print(f"Are c and d the same object? {c is d}")
- print(f"Do c and d have same value? {c == d}")

Insights Gained

We need sys.intern() to manually intern strings.

Two separate string objects with same value → c is d likely False .

Memory_Interning.py

Overview

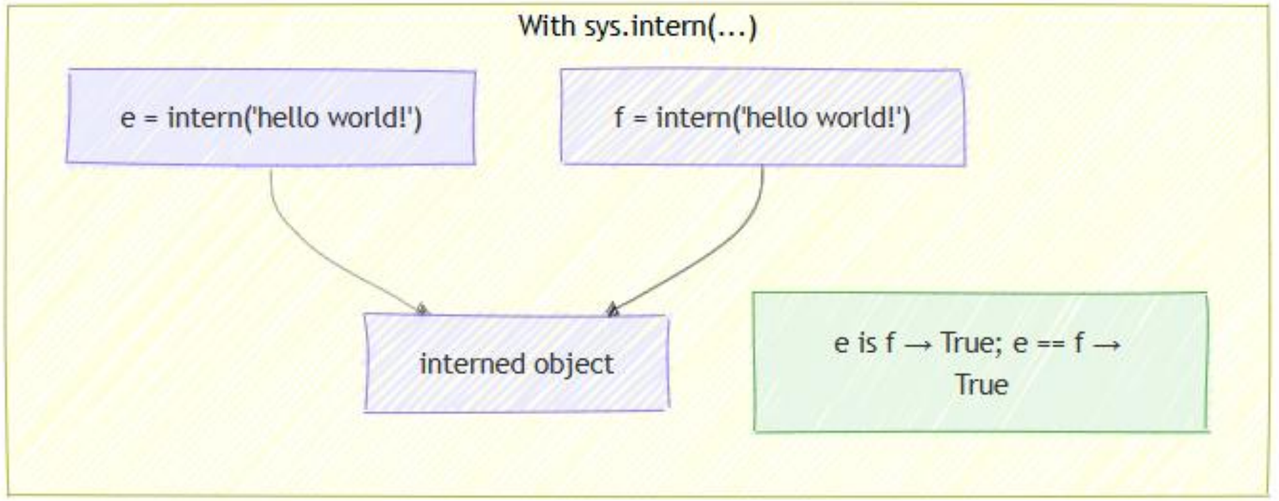
- # Manually intern longer strings to force memory sharing
- e = sys.intern("hello world!")
- f = sys.intern("hello world!")
- print(f"Memory location of e: {id(e)}")
- print(f"Memory location of f: {id(f)}")
- print(f"Are e and f the same object? {e is f}")
- print(f"Do e and f have same value? {e == f}")

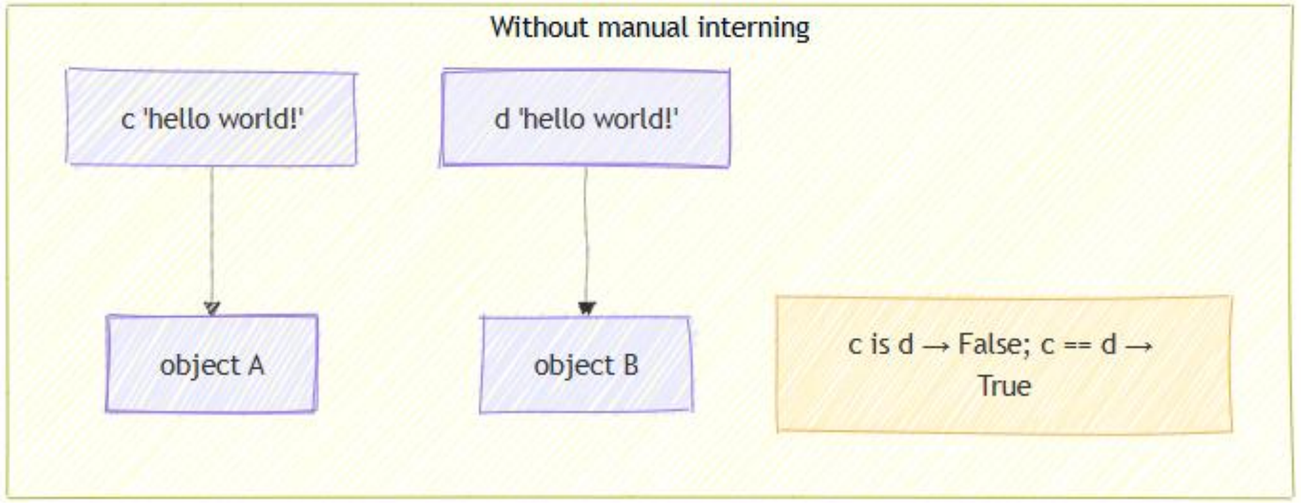
Insights Gained

Returns a shared interned object for that text.

Returns the same interned object as e → e is f
True .

Memory_Interning.py





Mutable Objects_list_dict_set.py

Overview

```
# List demonstration
print("LIST DEMONSTRATION")
print("-" * 20)
list1 = [1, 2, 3]
list2 = list1  # Both point to same list

print(f"Initially:")
print(f"list1 = {list1}, id(list1) = {id(list1)}")
print(f"list2 = {list2}, id(list2) = {id(list2)}")

list1.append(4)  # Modifies the same list
print(f"\nAfter list1.append(4):")
print(f"list1 = {list1}, id(list1) = {id(list1)}")
print(f"list2 = {list2}, id(list2) = {id(list2)}")

print("\n" + "=" * 40 + "\n")

# Dictionary demonstration
print("DICTIONARY DEMONSTRATION")
print("-" * 20)
dict1 = {'a': 1, 'b': 2}
dict2 = dict1  # Both point to same dictionary

print(f"Initially:")
print(f"dict1 = {dict1}, id(dict1) = {id(dict1)}")
print(f"dict2 = {dict2}, id(dict2) = {id(dict2)}")

dict1['c'] = 3  # Modifies the same dictionary
print(f"\nAfter dict1['c'] = 3:")
print(f"dict1 = {dict1}, id(dict1) = {id(dict1)}")
print(f"dict2 = {dict2}, id(dict2) = {id(dict2)}")

print("\n" + "=" * 40 + "\n")

# Set demonstration
print("SET DEMONSTRATION")
print("-" * 20)
set1 = {1, 2, 3}
set2 = set1  # Both point to same set

print(f"Initially:")
print(f"set1 = {set1}, id(set1) = {id(set1)}")
print(f"set2 = {set2}, id(set2) = {id(set2)}")

set1.add(4)  # Modifies the same set
print(f"\nAfter set1.add(4):")
print(f"set1 = {set1}, id(set1) = {id(set1)}")
print(f"set2 = {set2}, id(set2) = {id(set2)}")
```

Insights Gained

Lists, dicts, and sets are mutable: changing them in place affects all names that reference the same object.

Assigning alias = container creates another reference (alias), not a copy.

The id(...) stays the same before and after mutation, proving we changed the same object.

Mutable Objects_list_dict_set.py

Overview

- # List demonstration

- print("LIST DEMONSTRATION")
- print("-" * 20)
- list1 = [1, 2, 3]
- list2 = list1
- # Both point to same list
- print(f"Initially:")

- print(f"list1 = {list1}, id(list1) = {id(list1)}")
- print(f"list2 = {list2}, id(list2) = {id(list2)}")

- list1.append(4) # Modifies the same list

- print(f"After list1.append(4):")
- print(f"list1 = {list1}, id(list1) = {id(list1)}")
- print(f"list2 = {list2}, id(list2) = {id(list2)}")
- print("" + "=" * 40 + "")

Insights Gained

- list2 becomes an alias; both names point to one list object.

- Adds to that same list; both list1 and list2 see the change.

Mutable Objects_list_dict_set.py

Overview

```
# Dictionary demonstration
print("DICTIONARY DEMONSTRATION")
print("-" * 20)
dict1 = {'a': 1, 'b': 2}
dict2 = dict1  # Both point to same dictionary

print(f"Initially:")
print(f"dict1 = {dict1}, id(dict1) = {id(dict1)}")
print(f"dict2 = {dict2}, id(dict2) = {id(dict2)}")

dict1['c'] = 3  # Modifies the same dictionary
print(f"\nAfter dict1['c'] = 3:")
print(f"dict1 = {dict1}, id(dict1) = {id(dict1)}")
print(f"dict2 = {dict2}, id(dict2) = {id(dict2)}")

print("\n" + "=" * 40 + "\n")
```

Insights Gained

Same for dictionaries—update via one name is visible via the other

Mutable Objects_list_dict_set.py

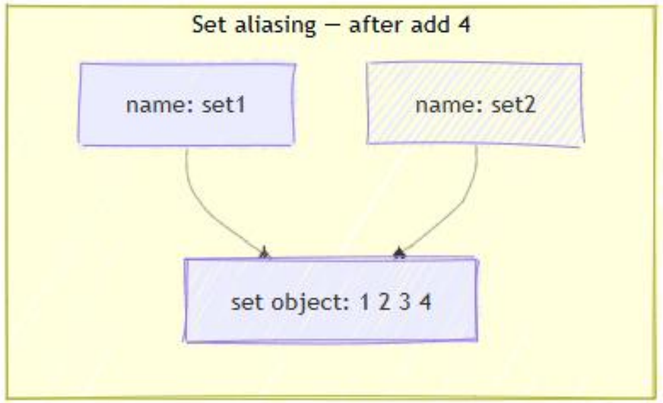
Overview

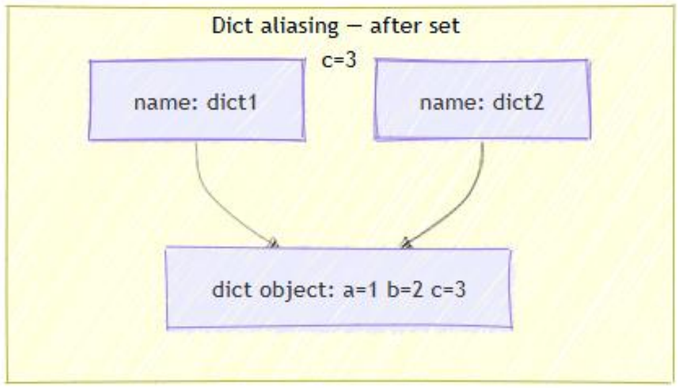
```
# Set demonstration
print("SET DEMONSTRATION")
print("-" * 20)
set1 = {1, 2, 3}
set2 = set1  # Both point to same set

print(f"Initially:")
print(f"set1 = {set1}, id(set1) = {id(set1)}")
print(f"set2 = {set2}, id(set2) = {id(set2)}")

set1.add(4)  # Modifies the same set
print(f"\nAfter set1.add(4):")
print(f"set1 = {set1}, id(set1) = {id(set1)}")
print(f"set2 = {set2}, id(set2) = {id(set2)}")
```

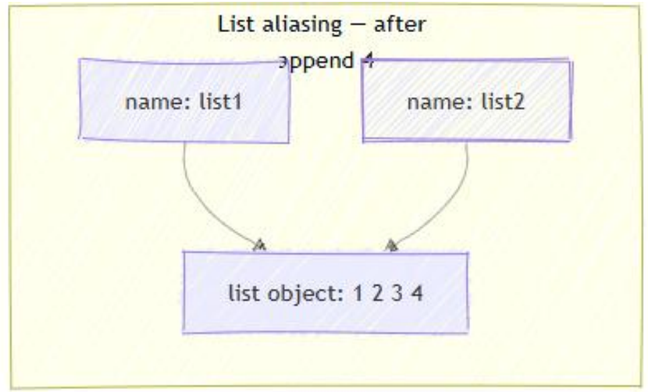
Insights Gained

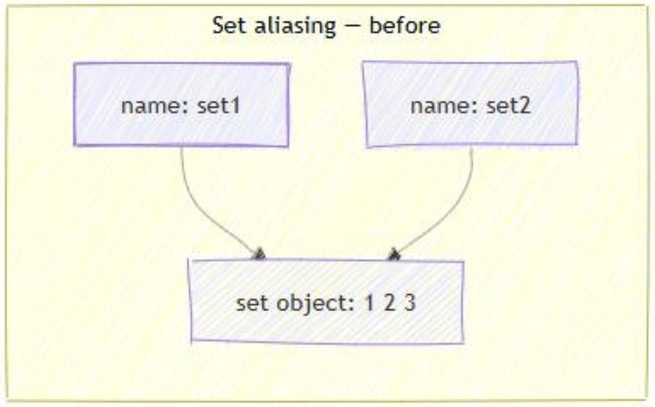
Same for sets. id confirms it’s the same object before/after.

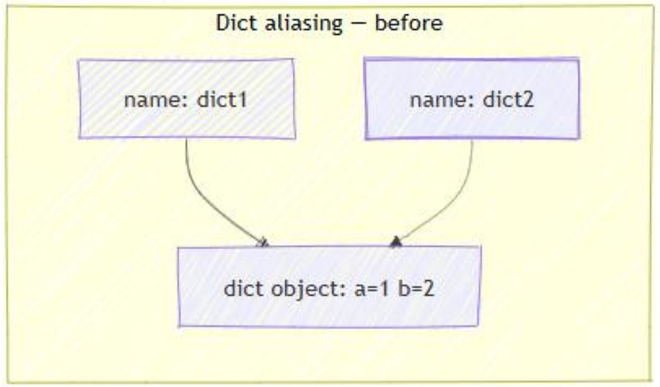
Mutable Objects_list_dict_set.py

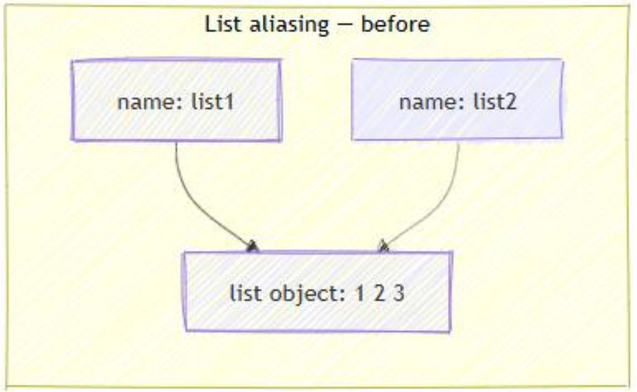












Weak_References.py

Overview

```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()()}")  # Prin : 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
```

Insights Gained

A weak reference refers to an object without increasing its reference count.

When all strong references disappear, the object can be collected; then the weak reference returns None .

Useful in caches/observer patterns to avoid keeping objects alive unintentionally.

Weak_References.py

Overview

- 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}")

Insights Gained

Tools for creating weak references.

Shows refcount + 1 (due to the call).

We subtract 1 to see the real count.

Weak_References.py

Overview

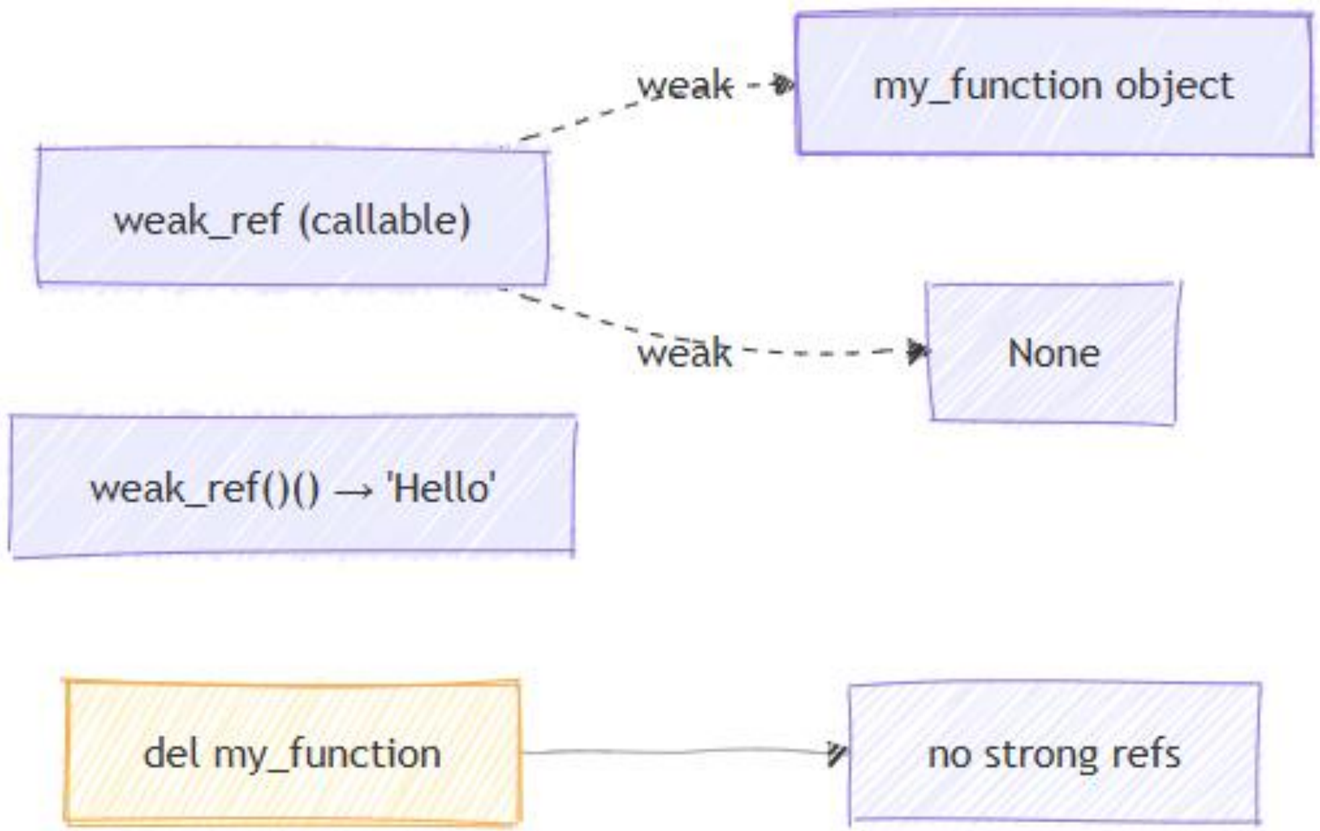
```
# Create a weak reference to the function

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
```

Insights Gained

Creates a weak reference callable; calling weak_ref() returns the object (or None if collected).

Removes the last strong reference; afterwards weak_ref() returns None .

Weak_References.py



Reference_Counting_Threading_Complication.py

Overview

```
import threading
import time
import random  # To add random delays

# Shared resource
shared_counter = 0
expected_count = 0  # Track the total expected increments

# Function that increments the shared counter
def increment_counter(thread_name):
    global shared_counter, expected_count
    for _ in range(5):
        # Read the current value
        temp = shared_counter
        print(f"{thread_name} reads: {temp}")

        # Simulate processing time
        time.sleep(random.uniform(0.0001, 0.001))

        # Increment the value
        temp += 1
        print(f"{thread_name} increments to: {temp}")

        # Simulate processing time
        time.sleep(random.uniform(0.0001, 0.001))

        # Write the updated value back
        shared_counter = temp
        print(f"{thread_name} writes back: {shared_counter}")

        # Track the total expected increments
        expected_count += 1

# Create two threads with names for logging
thread1 = threading.Thread(target=increment_counter, args=("Thread-1",))
thread2 = threading.Thread(target=increment_counter, args=("Thread-2",))

# Start the threads
thread1.start()
thread2.start()

# Wait for both threads to finish
thread1.join()
thread2.join()

# Print results
print(f"\nExpected final count: {expected_count}")
print(f"Actual final count: {shared_counter}")
print(f"Lost updates: {expected_count - shared_counter}")
```

Insights Gained

Demonstrates a race condition: two threads read, modify, and write a shared value without synchronization, so updates can be lost.

Random sleeps widen the race window to make overlap likely.

expected_count shows how many increments we intended; shared_counter shows how many we actually kept.

Reference_Counting_Threading_Complication.py

Overview

```
import threading
import time
import random  # To add random delays

# Shared resource
shared_counter = 0
expected_count = 0  # Track the total expected increments
```

Insights Gained

Import Threading, Time Library
Import Random Library for adding delays

The shared variable both threads update.

Reference_Counting_Threading_Complication.py

Overview

```
# Function that increments the shared counter
def increment_counter(thread_name):
    global shared_counter, expected_count
    for _ in range(5):
        # Read the current value
        temp = shared_counter
        print(f"{thread_name} reads: {temp}")

        # Simulate processing time
        time.sleep(random.uniform(0.0001, 0.001))

        # Increment the value
        temp += 1
        print(f"{thread_name} increments to: {temp}")

        # Simulate processing time
        time.sleep(random.uniform(0.0001, 0.001))

        # Write the updated value back
        shared_counter = temp
        print(f"{thread_name} writes back: {shared_counter}")

        # Track the total expected increments
        expected_count += 1
```

Insights Gained

Worker reads → waits → increments → waits → writes back (racy!).

Adds timing randomness so threads interleave

Reference_Counting_Threading_Complication.py

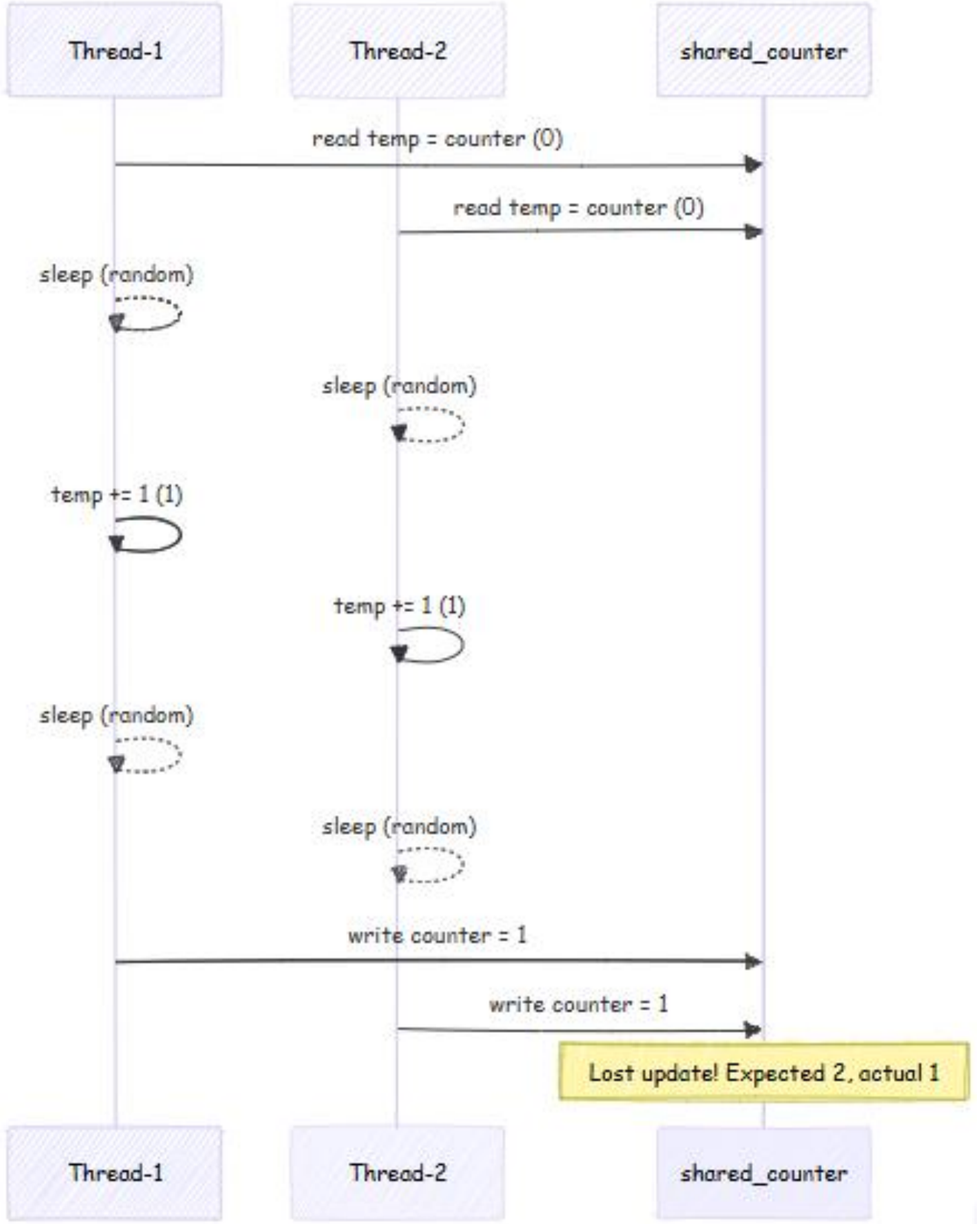
Overview

```
# Create two threads with names for logging
thread1 = threading.Thread(target=increment_counter,
args=("Thread-1",))
thread2 = threading.Thread(target=increment_counter,
args=("Thread-2",))

# Start the threads
thread1.start()
thread2.start()

# Wait for both threads to finish
thread1.join()
thread2.join()

# Print results
print(f"\nExpected final count: {expected_count}")
print(f"Actual final count: {shared_counter}")
print(f"Lost updates: {expected_count - shared_counter}")
```

Insights Gained

Compare intended (expected_count) vs actual (shared_counter) to show lost updates.

Reference_Counting_Threading_Complication.py



Reference_Counting_Mutex_Solution.py

Overview

```
import threading
import time
import random

# Shared resource
shared_counter = 0
expected_count = 0  # To track expected increments
mutex = threading.Lock()  # Create a mutex lock

def increment_counter(thread_name):
    global shared_counter, expected_count

    for i in range(5):  # Reduced iterations to make it clearer
        # Acquire lock before critical section
        mutex.acquire()  # Lock to prevent other threads from accessing
        try:
            # Read
            local_counter = shared_counter
            print(f"{thread_name} reads: {local_counter}")

            # Simulate some processing time
            time.sleep(0.1)

            # Increment
            local_counter += 1
            print(f"{thread_name} increments to: {local_counter}")

            # Simulate more processing time
            time.sleep(0.1)

            # Write back
            shared_counter = local_counter
            print(f"{thread_name} writes back: {shared_counter}")
            print("-" * 50)

            expected_count += 1
        finally:
            mutex.release()  # Always release the lock, even if an error occurs
            time.sleep(0.1)  # Give other thread a chance to acquire lock

# Create two threads
thread1 = threading.Thread(target=increment_counter, args=("Thread-1",))
thread2 = threading.Thread(target=increment_counter, args=("Thread-2",))

start_time = time.time()

# Start threads
thread1.start()
thread2.start()

# Wait for threads to complete
thread1.join()
thread2.join()

end_time = time.time()

print(f"\nExpected final count: {expected_count}")
print(f"Actual final count: {shared_counter}")
print(f"Lost updates: {expected_count - shared_counter}")
print(f"Time taken: {end_time - start_time:.2f} seconds")
```

Insights Gained

A mutex (lock) ensures only one thread changes the shared value at a time, removing the race.

The try/finally pattern guarantees the lock is released even if an error occurs.

Reference_Counting_Mutex_Solution.py

Overview

```
import threading
import time
import random


# Shared resource

shared_counter = 0
expected_count = 0

# To track expected increments


mutex = threading.Lock()

# Create a mutex lock
```

Insights Gained

A lock that provides exclusive access to the critical section.

Reference_Counting_Mutex_Solution.py

Overview

```
def increment_counter(thread_name):
    global shared_counter, expected_count

    for i in range(5):  # Reduced iterations to make it clearer
        # Acquire lock before critical section
        mutex.acquire()  # Lock to prevent other threads from accessing
        try:
            # Read
            local_counter = shared_counter
            print(f"{thread_name} reads: {local_counter}")

            # Simulate some processing time
            time.sleep(0.1)

            # Increment
            local_counter += 1
            print(f"{thread_name} increments to: {local_counter}")

            # Simulate more processing time
            time.sleep(0.1)

            # Write back
            shared_counter = local_counter
            print(f"{thread_name} writes back: {shared_counter}")
            print("-" * 50)

            expected_count += 1
        finally:
            mutex.release()  # Always release the lock, even if an error occurs
            time.sleep(0.1)  # Give other thread a chance to acquire lock
```

Insights Gained

Enter/leave the critical section; only one thread proceeds at once.

Read → compute → write happens atomically relative to other threads.

Reference_Counting_Mutex_Solution.py

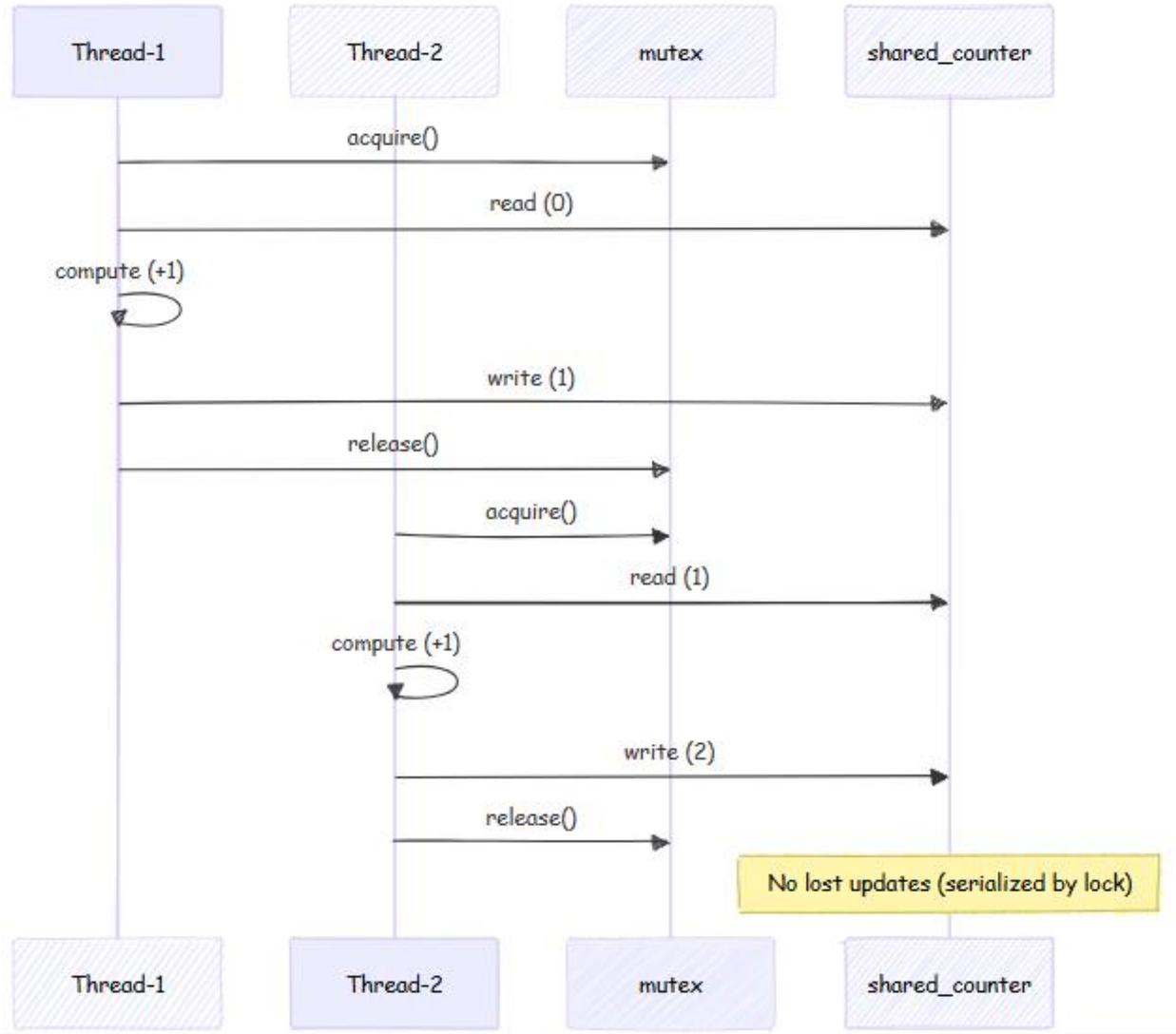
Overview

```
# Create two threads
thread1 = threading.Thread(target=increment_counter, args=("Thread-1",))
thread2 = threading.Thread(target=increment_counter, args=("Thread-2",))

start_time = time.time()

# Start threads
thread1.start()
thread2.start()

# Wait for threads to complete
thread1.join()
thread2.join()

end_time = time.time()

print(f"\nExpected final count: {expected_count}")
print(f"Actual final count: {shared_counter}")
print(f"Lost updates: {expected_count - shared_counter}")
print(f"Time taken: {end_time - start_time:.2f} seconds")
```

Insights Gained

Shows the correct final count; slightly slower due to locking

Reference_Counting_Mutex_Solution.py



Using_Del_Manual_Object_Deletion.py

Overview

```
# Example of manual object deletion using 'del'

# Create variables and show initial state
x = 42
y = x  # y references same integer as x
print(f"Initial value of x: {x}")
print(f"x and y reference same object: {x is y}")
print(f"Memory location of x: {id(x)}")
print(f"Memory location of y: {id(y)}")

list1 = [1, 2, 3, 4, 5]
list2 = list1  # list2 references same list as list1
print(f"\nInitial list1: {list1}")
print(f"list1 and list2 reference same object: {list1 is list2}")
print(f"Memory location of list1: {id(list1)}")
print(f"Memory location of list2: {id(list2)}")

# Delete integer references
print("\nDeleting integer references...")
del x, y  # Removes references to integer 42
# print(x)  # Would raise NameError: x is not defined

# Delete list references
print("Deleting list references...")
del list1, list2  # Removes references to list
# print(list1)  # Would raise NameError: list1 is not defined

print("\nAll objects are now eligible for garbage collection")
# Note: Objects will be collected when Python's garbage collector runs
# We can't access these objects anymore as all references are deleted
```

Insights Gained

del name removes a name binding; the object is freed only when no strong references remain.

With aliases, the object can survive.

For immutable objects like int , multiple names can point to the same object; deleting one name doesn’t delete the object.

For mutable containers, other aliases keep the object alive until all references are gone.

Using_Del_Manual_Object_Deletion.py

Overview

```
# Example of manual object deletion using 'del'

# Create variables and show initial state
x = 42
y = x  # y references same integer as x
print(f"Initial value of x: {x}")
print(f"x and y reference same object: {x is y}")
print(f"Memory location of x: {id(x)}")
print(f"Memory location of y: {id(y)}")

list1 = [1, 2, 3, 4, 5]
list2 = list1  # list2 references same list as list1
print(f"\nInitial list1: {list1}")
print(f"list1 and list2 reference same object: {list1 is list2}")
print(f"Memory location of list1: {id(list1)}")
print(f"Memory location of list2: {id(list2)}")

# Delete integer references
print("\nDeleting integer references...")
del x, y  # Removes references to integer 42
# print(x)  # Would raise NameError: x is not defined
```

Insights Gained

Deletes the names x and y . If no other references exist, the integer object can be released.

Deletes both aliases to the list; now the list has refcount 0 and is collectible.

Using_Del_Manual_Object_Deletion.py

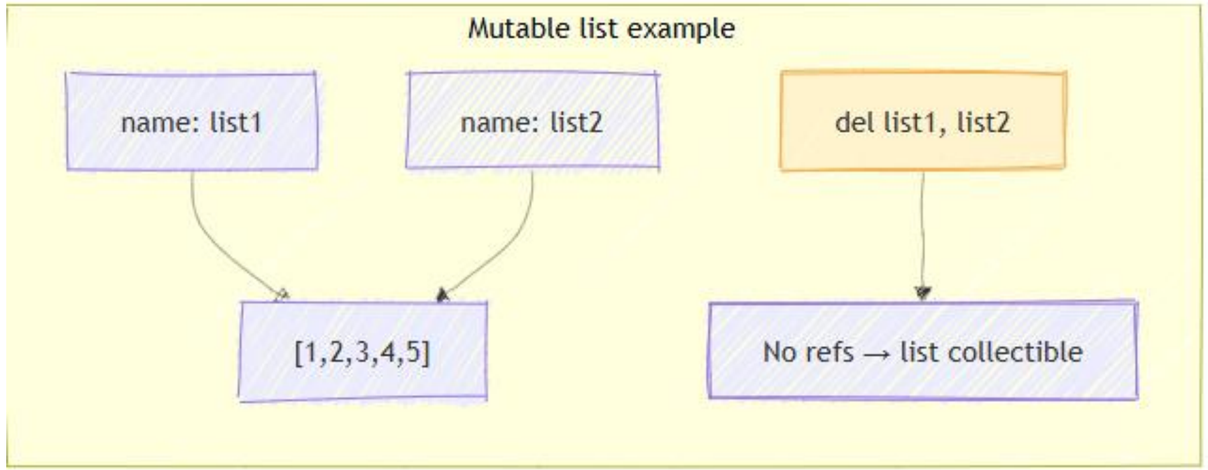
Overview

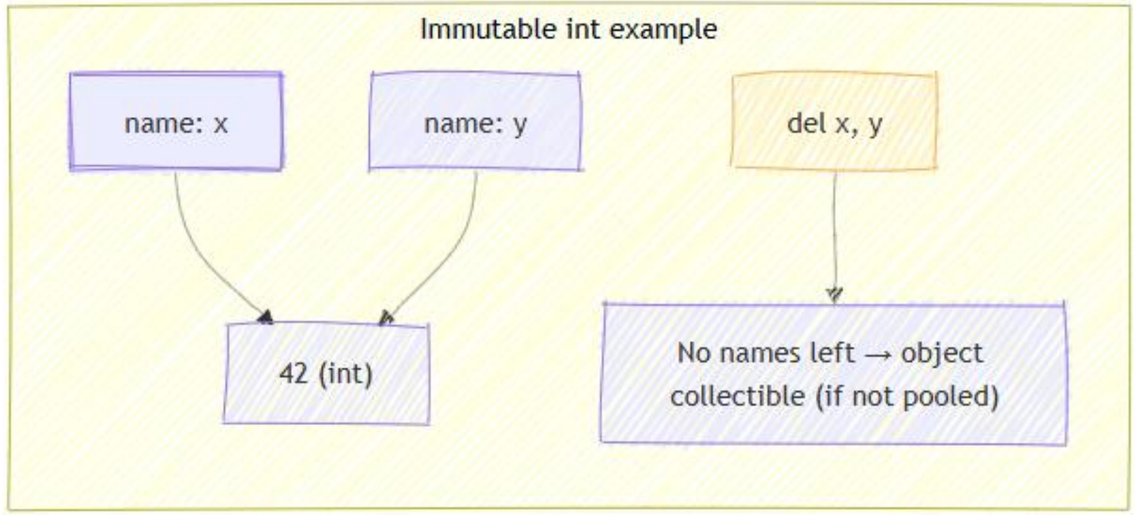
```
# Delete list references

print("Deleting list references...")
del list1, list2  # Removes references to list

# print(list1)
# Would raise NameError: list1 is not defined


print("\nAll objects are now eligible for garbage collection")



# Note: Objects will be collected when Python's garbage collector runs
# We can't access these objects anymore as all references are deleted
```

Insights Gained

Clarify that GC will reclaim memory automatically.

Using_Del_Manual_Object_Deletion.py





Using Garbage Collection (gc).py

Overview

```
# Import garbage collector module
import gc

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

# Create objects with circular references
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 clea ed up.")

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

Insights Gained

- Shows how to disable and re-enable the GC to demonstrate that cycles won't be reclaimed by refcount alone.
- Manually calling gc.collect() performs a full scan and frees the unreachable cycle

Using Garbage Collection (gc).py

Overview

```
# Import garbage collector module

import gc

# Disable automatic garbage collection

gc.disable()

print("Automatic garbage collection disabled.")

# Create objects with circular references

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

list1.append(list2)  # list1 references list2
list2.append(list1)  # list2 references list1
```

Insights Gained

Turns off the background cyclic GC (refcounting still works).

Two lists reference each other → refcount won’t drop to 0.

Using Garbage Collection (gc).py

Overview

```
# 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.")
```

Insights Gained

Detects and frees unreachable cycles.

Restores normal automatic collection.

Using Garbage Collection (gc).py