Memory Allocation & Garbage Collection

Approach 1: I surrender my soul to the God of Python, who will protect me, allocate my memory, and collect my unused memory when no longer needed.

Approach 2: Understand things at a deeper level, because it will help you at some point down the road.

Concept 1: Different areas of memory

① Stack (like a stack of objects, in a known order)

(like a pile of laundry,

② Heap (like a pile of [illegible] in the corner, just thrown randomly in there)

Stack ⇒ nicely ordered, FAST

Heap ⇒ no so ordered, SLOW

Concept 2: The Python Virtual Machine (CPython) which actual interprets, translates, compiles, and runs your code does indeed handle most of the memory management.

This model (also used in C#, Java, Perl, Ruby, Lua, etc.) is called Garbage Collection.

In CPython implementation, all Python objects (even int, float, etc)

live on the HEAP.

Concept 3: CPython does it like this: when an object is created, it allocates some memory on the HEAP for that object. It then "tags" that area of memory with a sign "In use". It keeps track of the "in use" states by something called a reference count.

Concept 4: When the reference count for an object falls to ZERO, then it may be de-allocated. This means that this area of memory is now free to use for

memory

other things. The Garbage Collector handles this de-allocation

```
class Test:
    def __init__(self, name):
        self.name = name

    def __del__(self):
        print(f"deleting {self.name}")


if "__name__" == "__main__":
    print("Discarded object creation")
    Test("hello1")

    print("Saved object creation")
    myList = Test("hello2")

    print("program done")
```

What Happens :

① Test("hello1")

→ object is created on the HEAP

→ but, it is not used for anything, so it gets garbage collected immediately

Reference Count = 0

② myList = Test("hello2")

→ object is created on the HEAP

→ since it is linked to myList, it stays allocated.

Reference Count = 1

③ program ends

→ only then does the memory associated with the myList object get deallocated.

Another Example :

s1 = "Python"

s2 = "Computer Science"

S2 = "Computer"

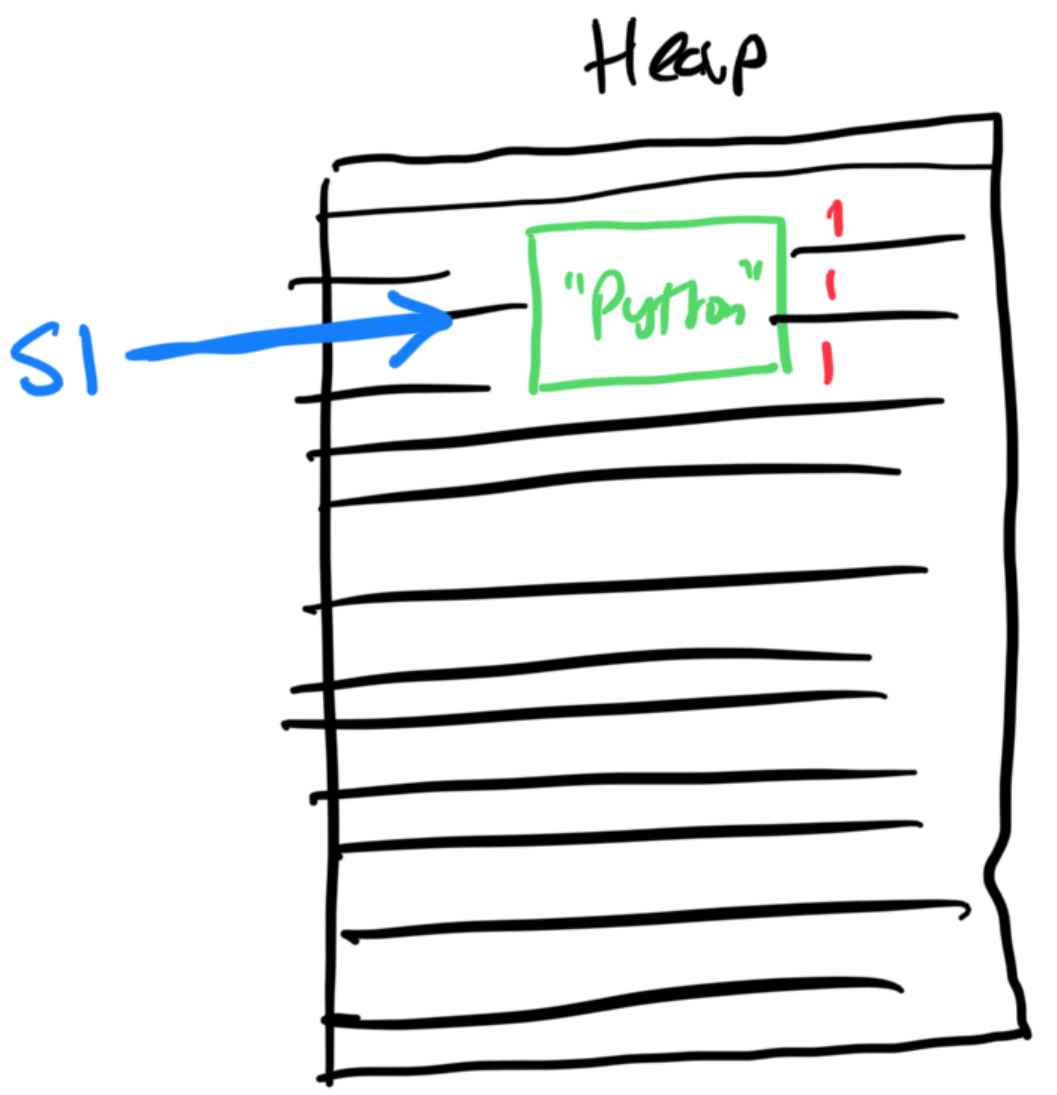
S3 = S2

S1 = "Zyboobs"

S2 = S1

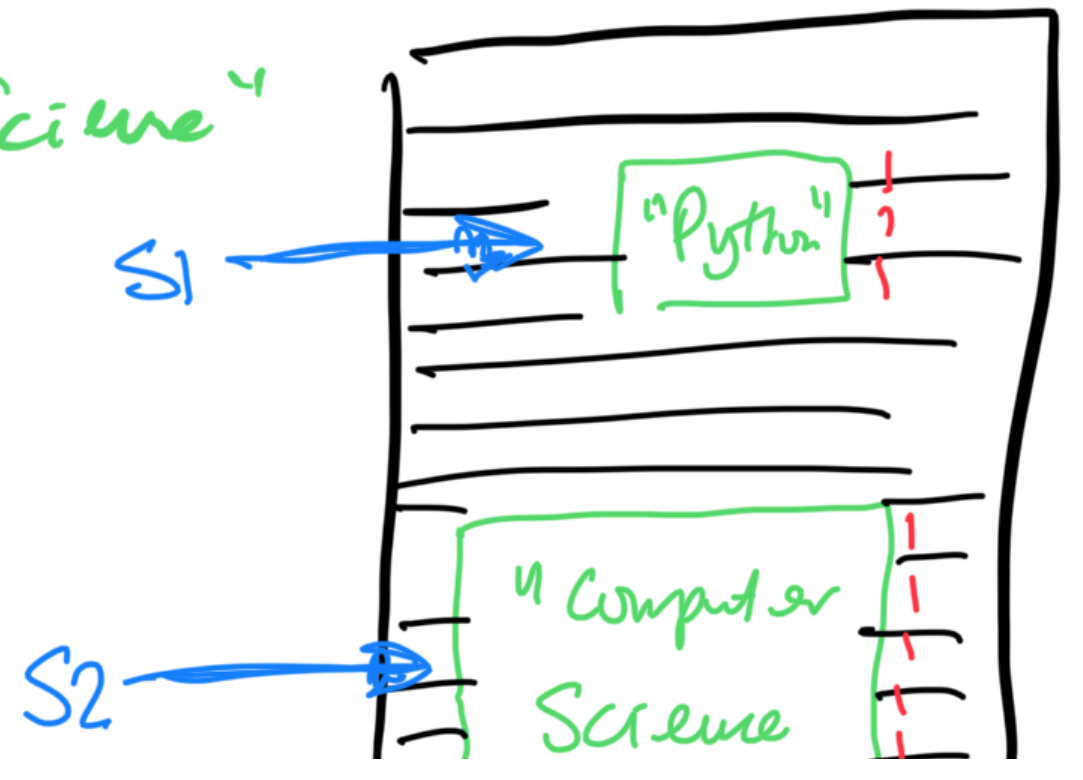
What happens

S1 = "Python"



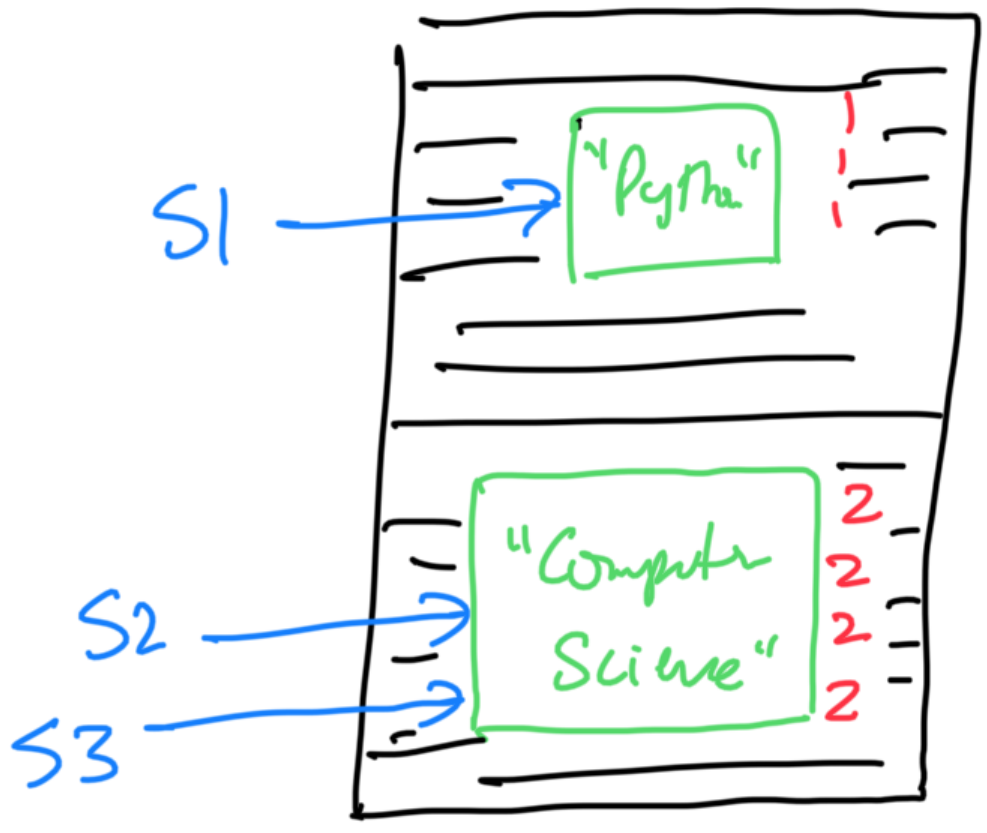
Reference Count for the "Python" area of the HEAP set to 1.

S2 = "Computer Science"



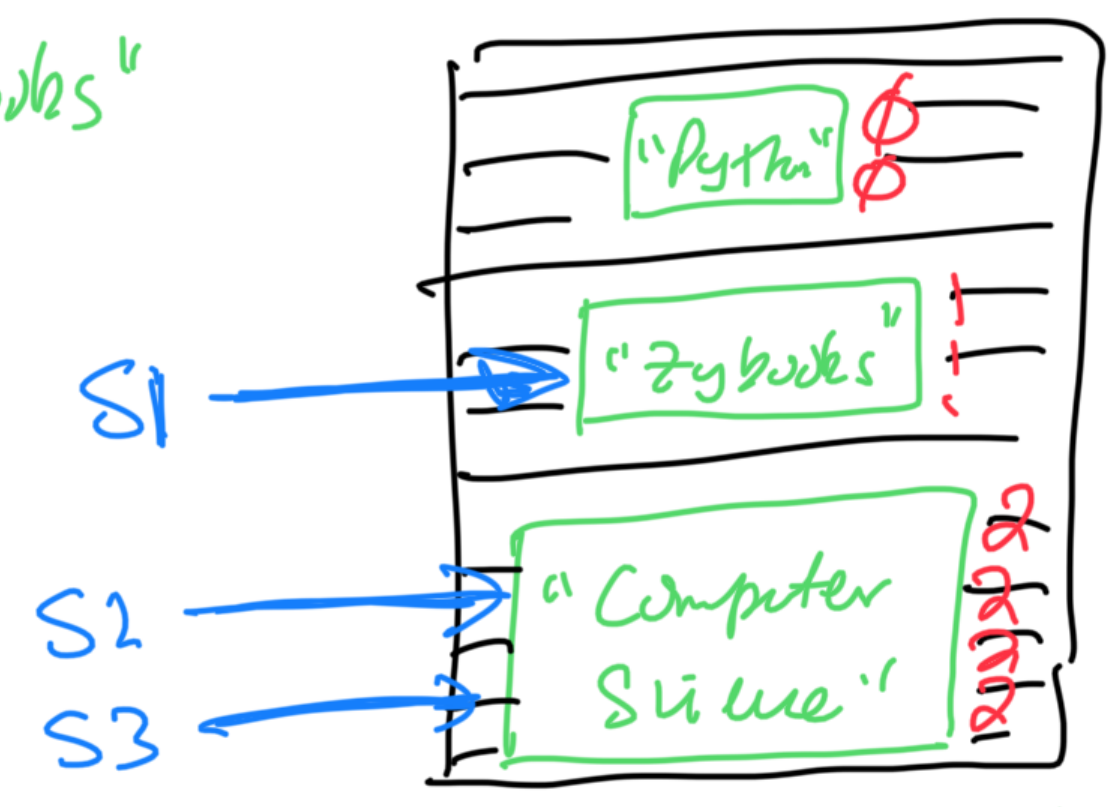
Reference Count for the "Computer Science" area of the HEAP set to 1.

S3 = S2



Reference count for the "Computer Science" area of the HEAP raised to 2 !!

S1 = "Zybooks"



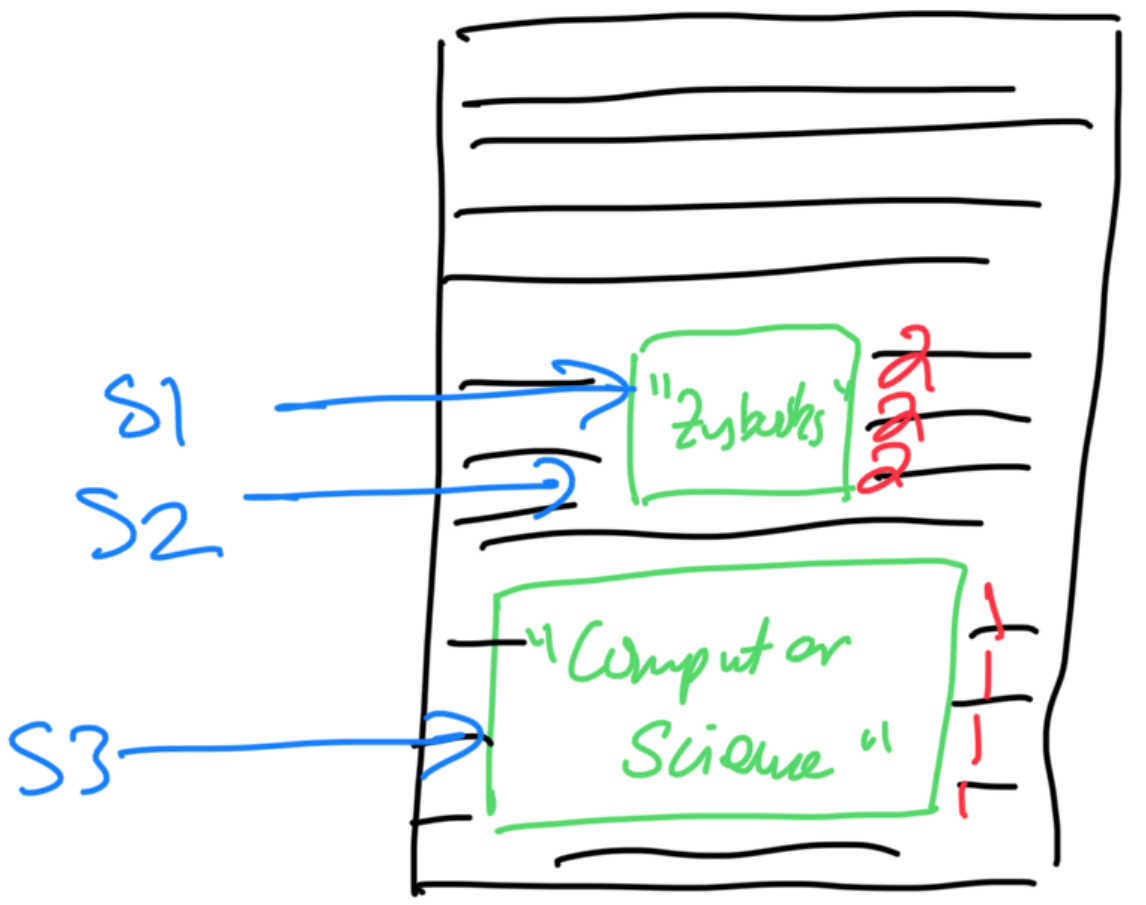
A) Reference count for a new "Zybooks" area of the HEAP set to 1.

B) Reference count for the "Python"

5) Reference count of the HEAP lowered to zero!!

Subject to GARBAGE Collection.

S2 = S1



A) Reference count for "Zybooks" area of the HEAP raised to 2.

B) Reference count for "Computer Science" area of HEAP lowered to 1.

There are actually a couple of very useful Python libraries that we

can use to get more information on memory usage and reference count.

```
import gc
```

← methods for garbage collection inspection

```
import ctypes
```

↖ contains some cool C functions which allow to inspect the value at a certain memory location (amongst other things)

```
gc_objs = gc.get_objects()
print(f"BEFORE: {len(gc_objs)}")
```

→ this will tell us how many objects have been deallocated by the garbage collector process.

See memory_test2.py for

see memory [illegible]

an example of how we can use this to follow garbage collection as the program executes.

So, at this point, you're probably saying, "Please, Lord, let him stop with this.... who CARES?"

Please stay with me here, just for one more example. ☺

Let's modify the code of memory_test2.py a bit.

```
class Dog:
    def __init__(self, name, cousin=None)
        self.name = name        ← string
        self.cousin = cousin    ← object
```

self.cousin = cousin (Cat)

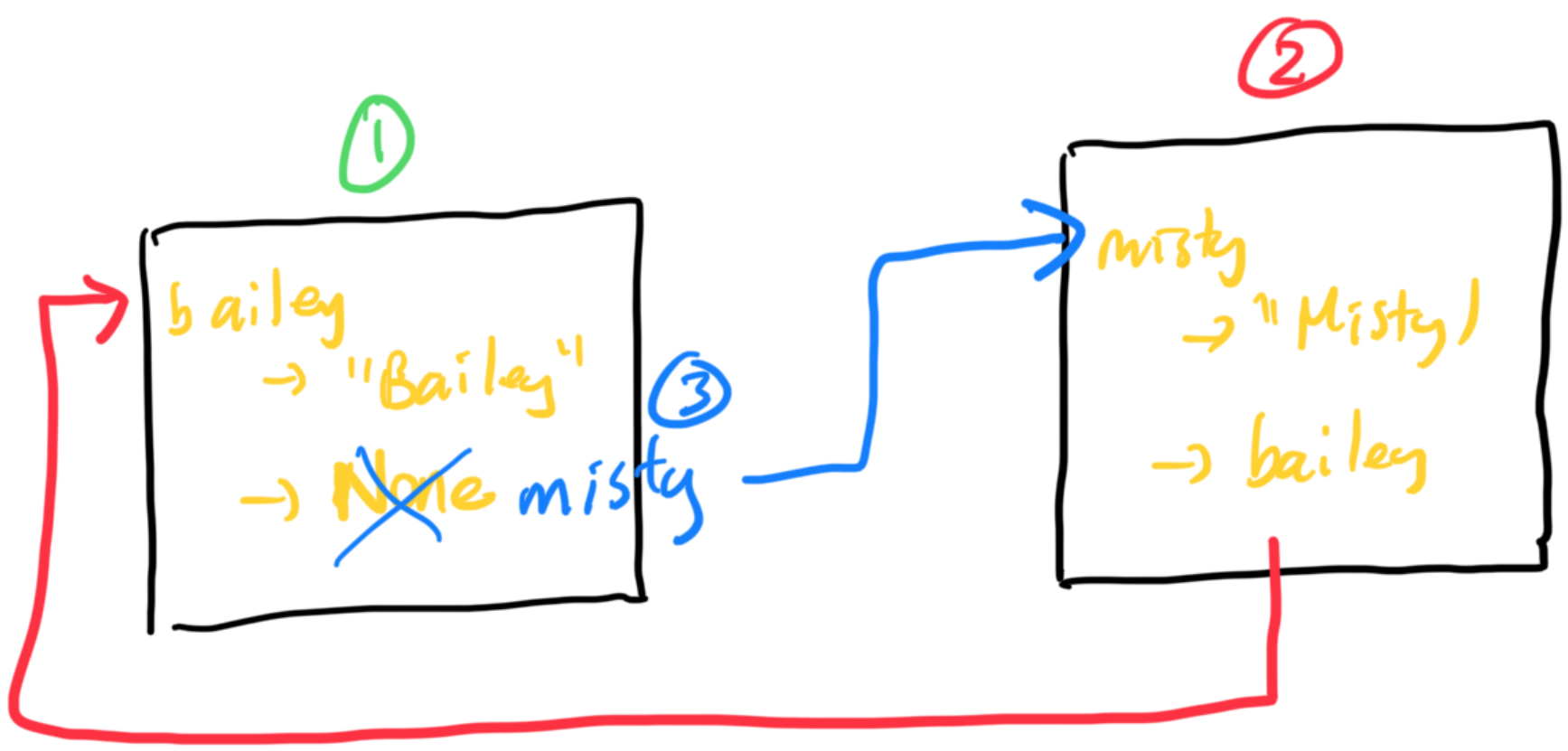
```
class Cat:
    def __init__(self, name, cousin=None)
        self.name = name          ← string
        self.cousin = cousin      ← object (Dog)
```

① bailey = Dog("Bailey")

② misty = Cat("Misty", bailey)

③ bailey.cousin = misty

→ What's happening?



→ This is called a circular reference.

→ This is called

Bailey reference count → 2 (self, misty)

Misty reference count → 2 (self, bailey)

→ Now, let's delete bailey

del bailey

→ Misty's reference count is still 2 !!

This is called a MEMORY LEAK

→ misty still thinks that bailey is its cousin, and might want to access that part of memory that it thinks is allocated to it. But it's gone.

→ How do we fix it?

If we choose to delete bailey, then we have to remember to also [illegible] references to bailey elsewhere;

fix references to dangling [illegible]

by hand.

$$\boxed{\text{misty.cousin} = \text{None}}$$

→ one could argue that we created the problem in the first place, by deleting the dangling object. But, we should be able to delete objects no longer being used, in order to free up memory and make our code more compact, efficient, and faster.

→ there is, however, a price to pay!! We have to work hard to make sure there are no memory leaks.

→ Fun fact: When an app has a very high

"crashes", there is
probability that it was a memory
leak!!