**Objects in Python**

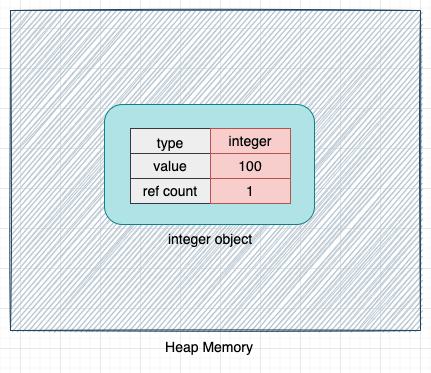
Everything in Python is an object. *Classes*, *functions*, and even simple data types, such as *integers*, *floats*, and *strings*, are objects in Python. When we define an integer in Python, CPython internally creates an object of type *integer*. These objects are stored in heap memory.

Each Python object consists of three fields:

* Value
* Type
* Reference count

Let's consider a simple example:

a = **100**



When the above code is executed, CPython creates an object of type *integer* and allocates memory for this object on the heap memory.

The *type* indicates the type of the object in CPython, and the *value* field, as the name suggests, stores the value of the object (*100* in this case). We will discuss the *ref\_count* field later in the article.

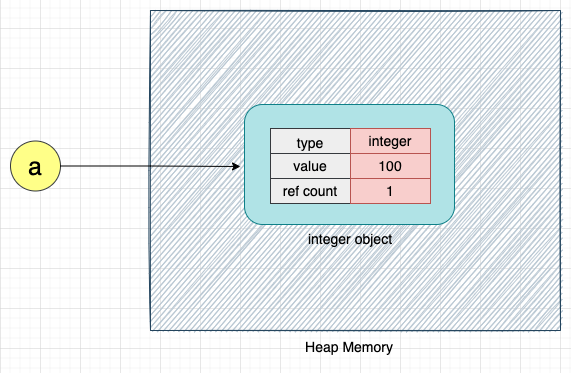
**Variables in Python**

Variables in Python are just references to the actual object in memory. They are like names or labels that point to the actual object in memory. They do not store any value.

Consider the following example:

a = **100**

As discussed earlier, when the above code is executed, CPython internally creates an object of type *integer*. The variable *a* points to this integer object as shown below:

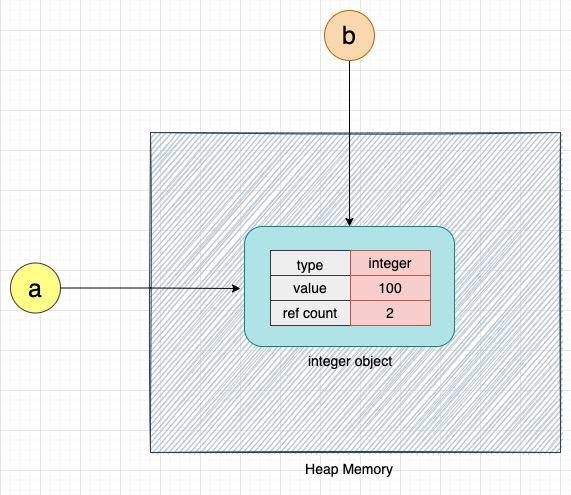


We can access the integer object in the Python program using the variable *a*.

Let's assign this integer object to another variable *b*:

b = a

When the above code is executed, the variables *a* and *b* both point to the same integer object, as shown below:

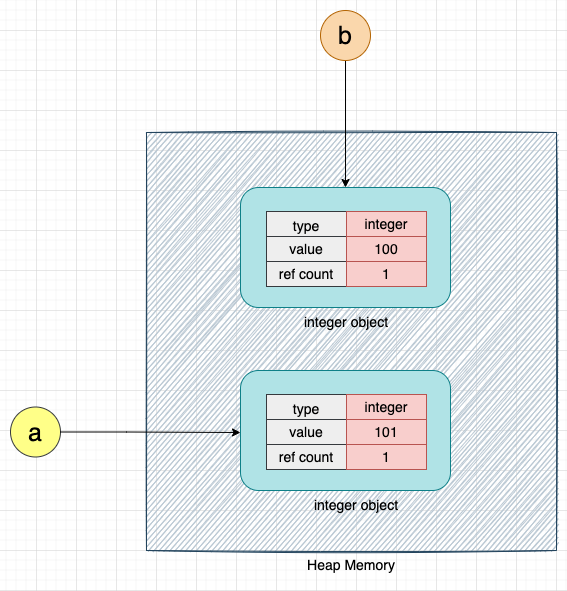


Let's now increment the value of the integer object by 1:

# Increment a by 1

a = a + **1**

When the above code is executed, CPython creates a new integer object with the value *101* and makes variable *a* point to this new integer object. Variable *b* will continue to point to the integer object with the value *100*, as shown below:



Here, we can see that instead of overwriting the value of *100* with *101*, CPython creates a new object with the value *101* because integers in Python are immutable. Once created, they cannot be modified. Please note that floats and string data types are also immutable in Python.

Let's consider a simple Python program to further explain this concept:

i = **0**

**while** i < **100**:

i = i + **1**

The above code defines a simple *while* loop that increments the value of the variable *i* until it is less than *100*. When this code is executed, for every increment of the variable *i*, CPython will create a new integer object with the incremented value, and the old integer object will be deleted (to be more precise, this object would become eligible for deletion) from memory.

CPython calls the *malloc* method for each new object to allocate memory for that object. It calls the *free* method to delete the old object from memory.

Let's convert the above code in terms of *malloc* and *free*:

i = **0** # malloc(i)

**while** i < **100**:

# malloc(i + 1)

# free(i)

i = i + **1**

We can see that CPython creates and deletes a large number of objects, even for this simple program. If we call the *malloc* and *free* methods for each object creation and deletion, it will degrade the program’s execution performance and make the program slow.

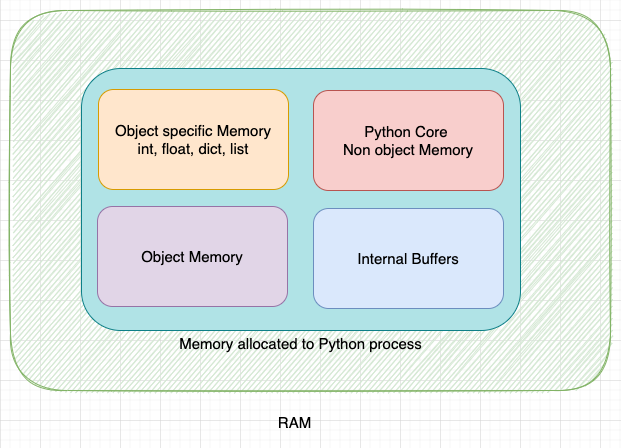
Hence, CPython introduces various techniques to reduce the number of times we have to call *malloc* and *free* for each small object creation and deletion. Let’s now understand how CPython manages memory!

**Memory Management in CPython**

Memory management in Python involves the management of a private heap. A private heap is a portion of memory that is exclusive to the Python process. All Python objects and data structures are stored in the private heap.

The operating system cannot allocate this piece of memory to another process. The size of the private heap can grow and shrink based on the memory requirements of the Python process. The private heap is managed by the Python memory manager defined inside the CPython code.

For representational purposes, the private heap in CPython can be divided into multiple portions as shown below:



Please note that the boundaries of each of these portions are not fixed and can grow or shrink based on the requirement.

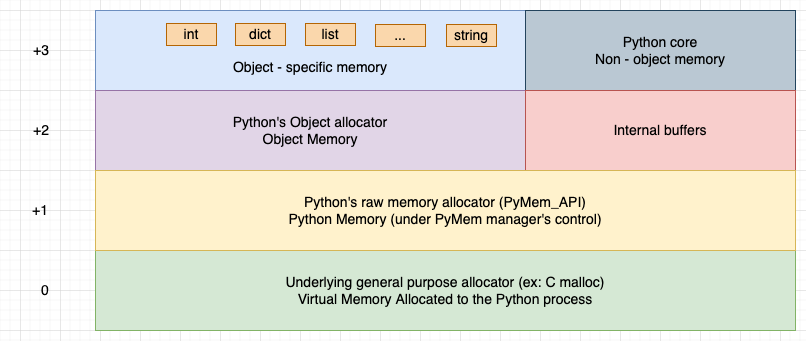
1. **Python Core Non-object memory**: Portion of the memory allocated to python core non-object data.
2. **Internal Buffers**: Portion of the memory allocated to internal buffers.
3. **Object-specific memory** - Portion of the memory allocated to objects that have object-specific memory allocators.
4. **Object Memory**: Portion of the memory allocated for objects.

When the program requests memory, CPython uses the *malloc* method to request that memory from the operating system, and the private heap grows in size.

To avoid calling *malloc* and *free* for each small object creation and deletion, CPython defines multiple allocators and deallocators for different purposes. We’ll discuss each of them in detail in the next section!

**Memory Allocators**

To avoid calling the *malloc* and *free* methods frequently, CPython defines a hierarchy of allocators, as shown below:



Here's the hierarchy of memory allocators from the base level:

* General Purpose Allocator (CPython's malloc method)
* Raw memory Allocator (for objects larger than 512 bytes)
* Object Allocator (for objects smaller than or equal to 512 bytes)
* Object-specific Allocators (specific memory allocators for specific data types)

At the base level is the *general-purpose* allocator. The *general-purpose* allocator is the *malloc* method of the C language for CPython. It is responsible for interacting with the virtual memory manager of the operating system and allocating the required memory to the Python process. This is the only allocator that communicates with the virtual memory manager of the operating system.

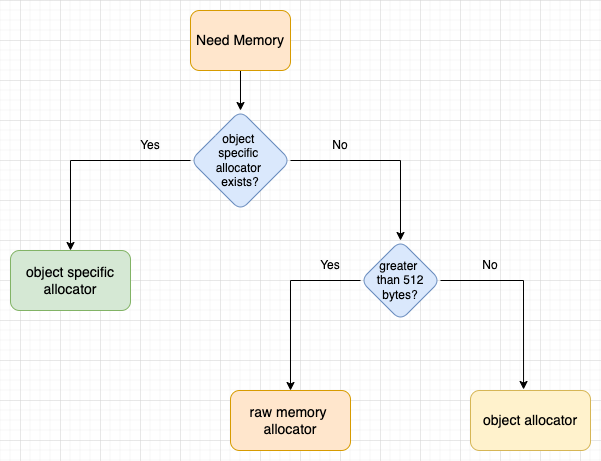
At the top of the *general-purpose* allocator is Python's *raw memory* allocator. The *raw memory* allocator provides an abstraction to the *general-purpose* allocator (i.e., the *malloc* method). When a Python process needs memory, the *raw memory* allocator interacts with the *general-purpose* allocator to provide the required memory. It makes sure that there is enough memory to store all the data of the Python process.

On top of the *raw memory* allocator, we have object allocator. This allocator is used for allocating memory for small objects (smaller than or equal to 512 bytes). If an object needs more than 512 bytes of memory, Python's memory manager directly calls the *raw memory* allocator.

As seen in the above representation, we have object-specific allocators on top of the object allocator. Simple data types, such as *integer*, *float*, *string*, and *list*, have respective object-specific allocators. These object-specific allocators implement memory management policies per the requirement of the object. For example, the object-specific allocator for *integer* has a different implementation than the object-specific allocator for *float*.

Both the object-specific allocators and object allocators work on memory already allocated to the Python process by the raw memory allocator. These allocators never request memory from the operating system. They operate on the private heap. If the object allocator or object-specific allocator needs more memory, Python's raw memory allocator provides it by interacting with the general-purpose allocator.

**Hierarchy of Memory Allocators in Python**



When an object requests memory, and the object has object-specific allocators defined, object-specific allocators are used to allocate the memory.

If the object does not have object-specific allocators, and more than 512 bytes of memory is requested, the Python memory manager directly calls the raw memory allocator to allocate memory.

If the requested memory size is less than 512 bytes, object allocators are used to allocate it.

**Object Allocator**

The object allocator is also called *pymalloc*. It is used to allocate memory for small objects less than 512 bytes in size.

a fast, special-purpose memory allocator for small blocks, to be used on top of a general-purpose malloc.

It is called for every object allocation and deallocation (PyObject\_New/Del) unless the object-specific allocators implement a proprietary allocation scheme (ex.: ints use a simple free list).

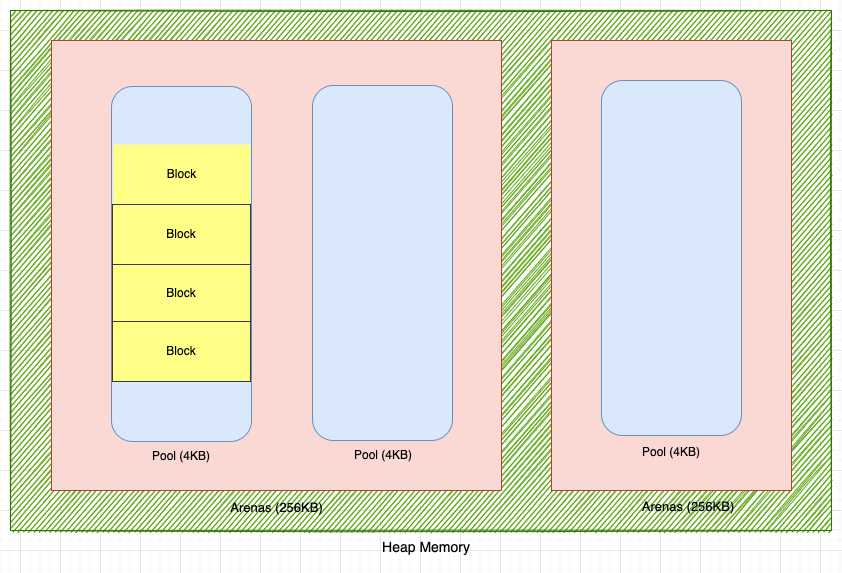
This is also the place where the cyclic garbage collector operates selectively on container objects.

When a small object requests memory, instead of just allocating the memory for that object, the object allocator requests a big block of memory from the operating system. This big block of memory is then later used to allocate memory for other small objects.

This way, the object allocator avoids calling *malloc* for each small object.

The big block of memory that the object allocator allocates is called an *Arena*. Arenas are 256 KB in size.

To use *Arenas* efficiently, CPython divides *Arena* into *Pools*. Pools are 4 KB. So, an *arena* can consists of 64 (256KB / 4KB) pools.



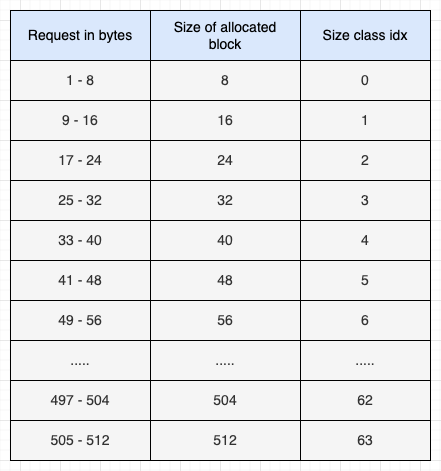
Pools are further divided into *Blocks*.

Next, we’ll discuss each of these components!

**Blocks**

Blocks are the smallest unit of memory that the object allocator can allocate to an object. A block can be allocated to only one object, and an object can only be allocated to one block. It is impossible to place parts of an object in two or more separate blocks.

Blocks are available in different sizes. The smallest size of a block is 8 bytes, while the maximum size of a block is 512 bytes. The size of a block is a multiple of 8, and therefore, block sizes can be 8, 16, 32, ..., 504, or 512 bytes. Each of the block sizes is called a *Size Class*. There are 64 size classes, as shown below:



As seen in the above table, the size class 0 blocks have a size of 8 bytes, while size class 1 blocks have a size of 16 bytes, and so on.

Programs are always allocated either a full block or no blocks at all. So, if a program requests 14 bytes of memory, it is allocated a block of 16 bytes. Similarly, if a program requests 35 bytes of memory, a block of 40 bytes is allocated.

**Pools**

A pool consists of blocks of only one size class. For example, a pool that has a block of size class 0 cannot have blocks of any other size class.

The size of the pool is equal to the size of the virtual memory page. Here' what the term virtual memory page means:

A page, memory page, or virtual page is a fixed-length contiguous block of virtual memory. It is the smallest unit of data for memory management in a virtual memory operating system. - Wikipedia

In most cases, the size of the pool is 4 KB.

Pools are only carved from Arenas when there is no other pool available that has a block of the requested size class.

A pool can be in one of three states:

1. **Used**: A pool is said to be in the *used* state if it has blocks available for allocation.
2. **Full**: A pool is said to be in the *full* state if all the blocks of the pool are allocated.
3. **Empty** A pool is said to be in the *empty* state if all the blocks of the pool are available for allocation. An empty pool has no size class associated with it. It can be used to assign blocks of any size class.

Pools are defined in the CPython code as shown below:

/\* Pool for small blocks. \*/

**struct** pool\_header {

**union** { block \*\_padding;

uint count; } ref; /\* number of allocated blocks \*/

block \*freeblock; /\* pool's free list head \*/

**struct** pool\_header \*nextpool; /\* next pool of this size class \*/

**struct** pool\_header \*prevpool; /\* previous pool "" \*/

uint arenaindex; /\* index into arenas of base adr \*/

uint szidx; /\* block size class index \*/

uint nextoffset; /\* bytes to virgin block \*/

uint maxnextoffset; /\* largest valid nextoffset \*/

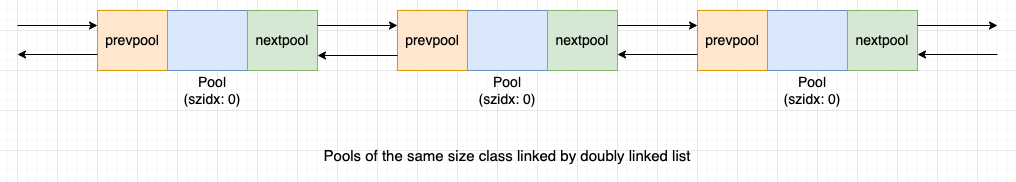
};

The term *szidx* indicates the size class of the pool. If *szidx* is 0 for a pool, it will only have blocks of size class 0 (i.e., blocks of 8 bytes).

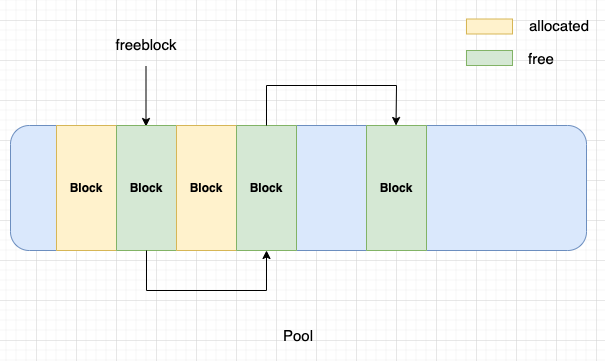
The term *arenaindex* indicates the arena to which the pool belongs.

Pools of the same size class are linked to each other using a doubly linked list. The *nextpool* pointer points to the next pool of the same size class, while the *prevpool* pointer points to the previous pool of the same size class.

Here's how pools of the same size class are connected:



The *freeblock* pointer points to the start of a singly linked list of free blocks within the pool. When an allocated block is freed, it is inserted at the front of the *freeblock* pointer.



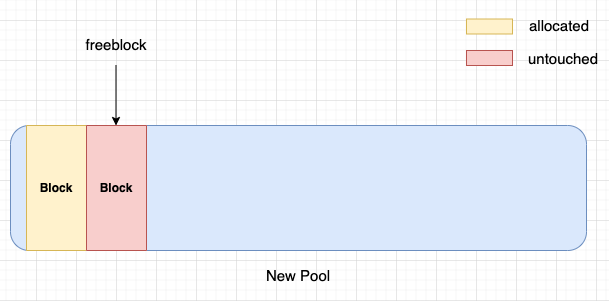
As seen in the above figure, *allocated* blocks are allocated to objects, while *free* blocks were allocated to objects but now are free and can be allocated to new objects.

When a request is made for memory, and there are no pools with a block of the requested size class available, CPython carves a new pool from the Arena.

When a new pool is carved, the entire pool is not immediately fragmented into blocks. Blocks are carved from the pool as and when needed. The *Blue* shaded region of the pool in the above figure indicates that these portions of the pool have not been fragmented into blocks.

An excerpt from the CPython codebase mentions carving blocks from the pool as follows:

The available blocks in a pool are not linked all together when a pool is initialized. Instead, only "the first two" (lowest addresses) blocks are set up, returning the first such block, and setting pool->freeblock to a one-block list holding the second such block. This is consistent with that pymalloc strives at all levels (arena, pool, and block) never to touch a piece of memory until it's actually needed.

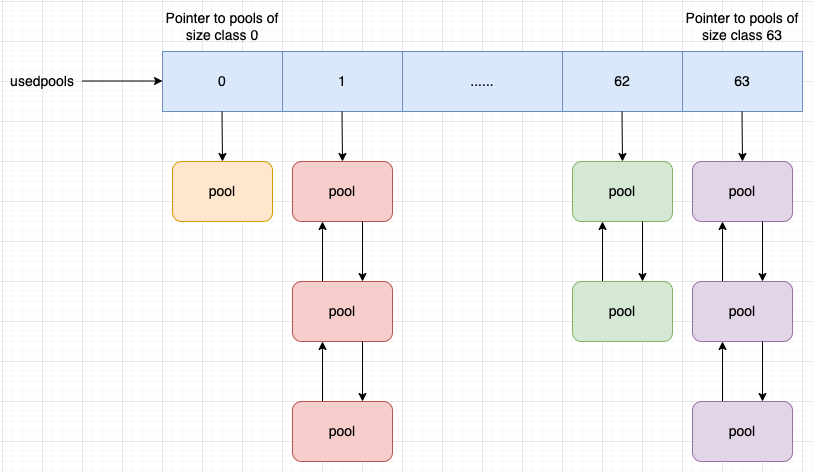


Here, we can see that when a new pool is carved from an arena, only the first two blocks are carved from the pool. One block is allocated to the object that requested the memory, while the other block is free or untouched, and the *freeblock* pointer points to this block.

CPython maintains an array called *usedpools* to keep track of pools in the *used* state (pools available for allocations) of all the size classes.

The index of the *usedpools* array is equal to the size class of the pool. For each index *i* of the *usedpools* array, *usedpools[i]* points to the header of the pools of size class *i*. For example, *usedpools[0]* points to the header of pools of size class *0*, and *usedpools[1]* points to the header of pools of size class *1*.

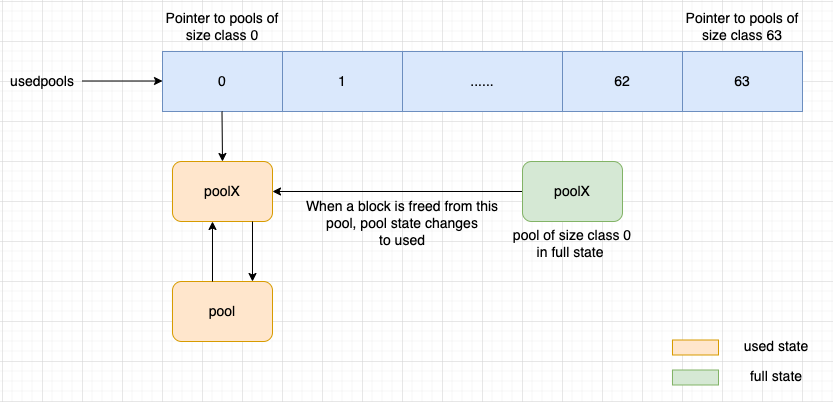
The figure below should make it easier to understand:



As pools of the same size class are linked to each other using a doubly linked list, all the pools in the *used* state of each size class can be traversed using the *usedpools* array.

If *usedpools[i]* points to *null*, it means there are no pools of size class *i* in the *used* state. If an object requests a block of size class *i*, CPython will carve a new pool of size class *i* and update *usedpools[i]* to point to this new pool.

If a block is freed from a pool in *full* state, the pool state changes from *full* to *used*. CPython adds this pool to the front of the doubly linked list of the pools of its size class.



As seen in the above image, *poolX* is of size class *0*, and it is in the *full* state. When a block is freed from *poolX*, its state changes from *full* to *used*. Once the state of *poolX* becomes *used*, CPython adds this pool to the front of the doubly linked list of pools of size class *0*, and *usedpools[0]* will start pointing to this pool.

**Arenas**

Arenas are big blocks of memory used for allocating memory to small objects. They are allocated by the raw memory allocator in sizes of 256 KB.

When a small object requests memory, but there are no existing arenas to handle this request, instead of only requesting memory for this small object, the raw memory allocator requests a big block of memory (256 KB) from the operating system. These big blocks of memory are called arenas.

The pools (4 KB in size) are carved from arenas when needed.

Consider the *arena\_object*, as defined in the CPython codebase:

**struct** arena\_object {

/\* The address of the arena, as returned by malloc \*/

**uintptr\_t** address;

/\* Pool-aligned pointer to the next pool to be carved off. \*/

block\* pool\_address;

/\* The number of available pools in the arena: free pools + never-

\* allocated pools.

\*/

uint nfreepools;

/\* The total number of pools in the arena, whether or not available. \*/

uint ntotalpools;

/\* Singly-linked list of available pools. \*/

**struct** pool\_header\* freepools;

**struct** arena\_object\* nextarena;

**struct** arena\_object\* prevarena;

};

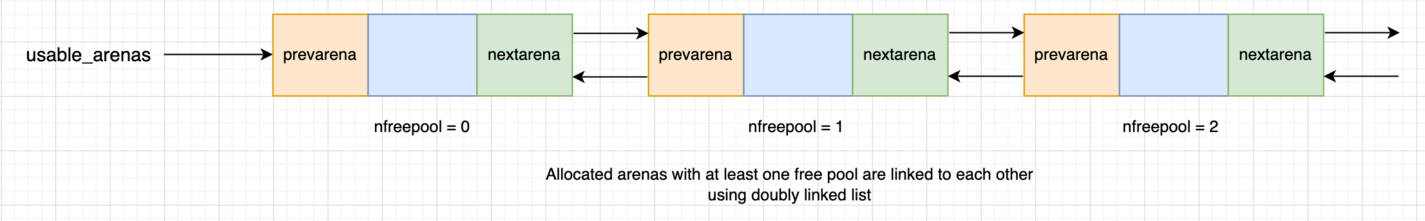
The *freepools* pointer points to the list of free pools, and free pools do not have any of their blocks allocated.

The *nfreepools* term indicates the number of free pools in the arenas.

CPython maintains a doubly linked list called *usable\_arenas* to keep track of all the arenas with *available* pools.

*Available* pools are in either the *empty* or *used* state.

The *nextarena* pointer points to the next usable arena, while the *prevarena* pointer points to the previous usable arena in the *usable\_arenas* doubly linked list. The doubly linked list is sorted in the increasing order of the *nfreepools* value.



As shown above, *usable\_arenas* is sorted on the basis of *nfreepools*. Arenas with 0 free pools is the first item, followed by arenas with 1 free pool, and so on. It means the list is sorted with the most allocated arenas first. We'll explain why this sorting is imported in the next section of the article.

The usable arenas list is sorted based on which ones have the most allocations, so when there is a request for memory allocation, it is served from the arena with the most allocations.

**Does a Python Process Release Memory?**

When an allocated block in a pool is freed, CPython does not return the memory back to the operating system. This memory continues to belong to the Python process, and CPython uses this block to allocate memory to new objects.

Even when all the blocks in a pool are freed, CPython does not return any memory from the pool to the operating system. CPython keeps the memory of the whole pool reserved for its own use.

CPython releases memory back to the operating system at the level of arenas, not at the block or pool level. Also, please note that CPython releases memory of whole arenas at once.

As memory can only be released at the arena level, CPython creates arena from new piece of memory only when it is absolutely necessary! It always tries to allocate memory from previously carved blocks and pools.

This is the reason *usable\_arenas* are sorted in decreasing order of *nfreepools*. The next request for memory would be allocated from arenas with the most allocations. This allows arenas with the least data to become free if the objects they contain are deleted, and the memory occupied by these arenas can be released to the operating system.

**Garbage Collection in Python**

Garbage collection is defined as the process of reclamation or release of allocated memory when it is no longer needed by the program.

Garbage collection (GC) is a form of automatic memory management. The garbage collector attempts to reclaim memory which was allocated by the program, but is no longer referenced — also called garbage. - Wikipedia

In languages like C, the programmer has to manually free the memory for unused objects (garbage collection of unused objects), while garbage collection in Python is automatically taken care of by the language itself.

Python uses two methods for automatic garbage collection:

1. Garbage collection on the basis of reference counting.
2. Generational garbage collection.

Let's first explain what a reference count is, and then we'll learn more about garbage collection on the basis of reference counting!

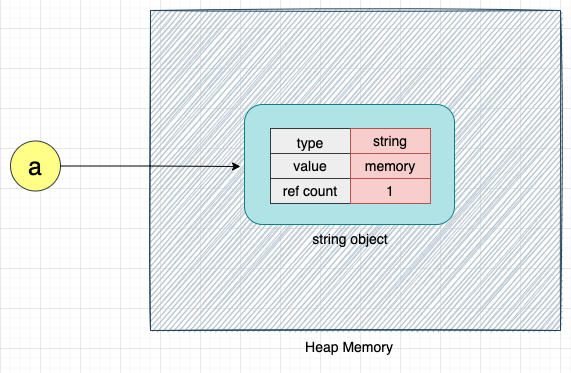
**Reference Count**

As we saw earlier, CPython internally creates the properties *type* and *ref count* for each object.

Let's consider an example to better understand the *ref count* property:

a = "memory"

When the above code is executed, CPython creates an object *memory* of type *string*, as shown below:



The field *ref count* indicates the number of references to the object. We know that in Python, variables are just references to objects. In the above example, the variable *a* is the only reference to the string object *memory*. Hence, the *ref count* value of string object *memory* is *1*.

We can get the reference count of any object in Python using the *getrefcount* method.

Let's get the reference count of the string object *memory*:

**import** **sys**

ref\_count = sys.getrefcount(a)

**print**(ref\_count) # Output: 2

The output of the above code is *2*. This indicates that the string object *memory* is being referenced by two variables. However, we saw earlier that the *memory* object is only being referenced by the variable *a*.

Why is the reference count value *2* for the string object *memory* when using the *getrefcount* method?

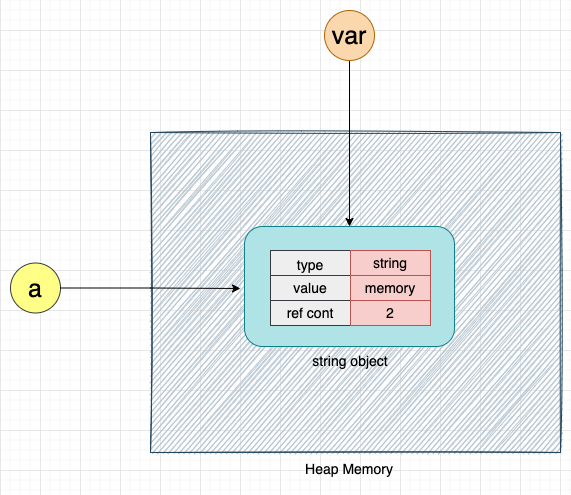
To understand this, let’s consider the definition of the *getrefcount* method:

**def** **getrefcount**(var):

...

Note: The above *getrefcount* definition is only for explanation purposes.

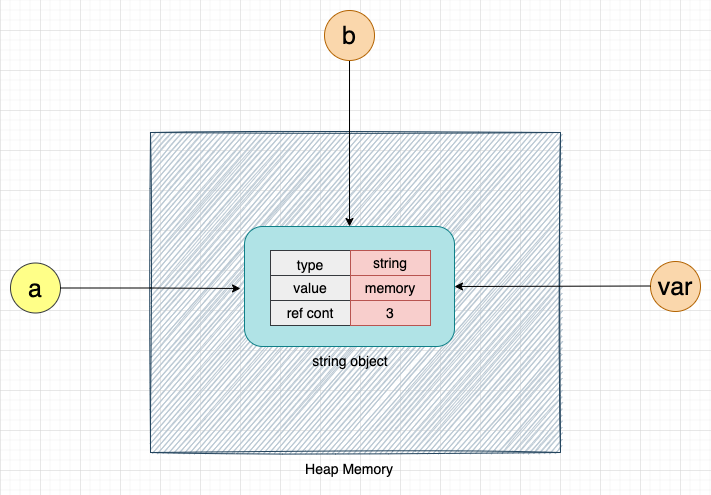
Here, when we pass variable *a* to the method *getrefcount*, the *memory* object is also referred to by the parameter *var* of the *getrefcount* method. Hence, the reference count of the *memory* object is *2*.



Thus, whenever we use the *getrefcount* method to get the reference count of an object, the reference count will always be 1 more than the actual reference count of the object.

Let's create another variable, *b*, that points to the same string object *memory*:

b = a



Variables *a* and *b* are both pointing to the string object *memory*. Hence, the reference count of the string object *memory* will be 2, and the output of the *getrefcount* method will be 3.

**import** **sys**

ref\_count = getrefcount(a)

**print**(ref\_count) # Output: 3

**Decreasing the Reference Count**

To decrease the reference count, we have to remove references to the variable. Here's an example:

b = None

The variable *b* is no longer pointing to the string object *memory*. Hence, the reference count of the string object *memory* will also decrease.

**import** **sys**

ref\_count = getrefcount(a)

**print**(ref\_count) # Output: 2

**Decreasing the Reference Count Using the del Keyword**

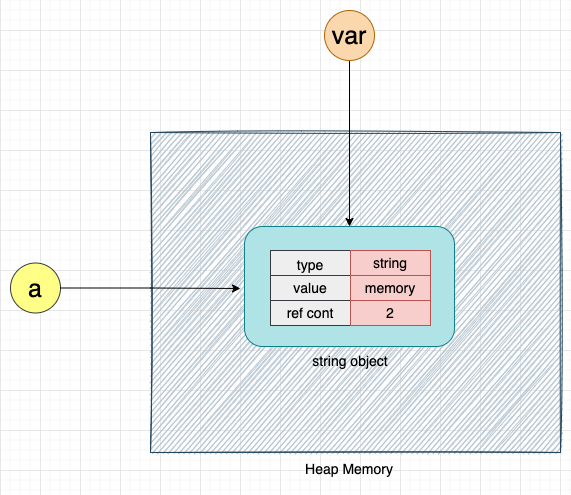
We can also use the *del* keyword to decrease the reference count of an object.

If we assign *None* to the variable *b* (*b = None*), *b* no longer points to the string object *memory*. The *del* keyword works the same way and is used to delete the reference of the object, thereby reducing its reference count.

Please note that the *del* keyword does not delete the object.

Consider the following example:

**del** b



Here, we are just deleting the reference of *b*. The string object *memory* is not deleted.

Let's now get the reference count of the string object *memory*:

**import** **sys**

ref\_count = getrefcount(b)

**print**(ref\_count) # Output: 2

Perfect! The reference count of *b* is now 2.

Let's recap what we learned about the reference count!

The reference count of an object increases if we assign the same object to a new variable. When we dereference the object by using the *del* keyword or by making it point to *None*, the reference count of that object decreases.

Now that we have a better understanding of the concept of reference count in Python, let's learn how garbage collection works on the basis of the reference count.

**Garbage Collection on the Basis of Reference Count**

Garbage collection on the basis of reference count uses the reference count of the object to release or reclaim memory. When the reference count of the object is zero, garbage collection by Python kicks in and removes the object from memory.

When an object is deleted from memory, it may trigger the deletion of other objects.

Consider the following example:

**import** **sys**

x = "garbage"

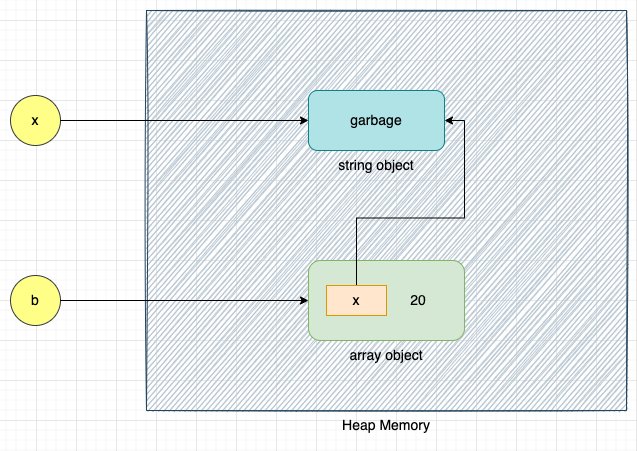
b = [x, **20**]

ref\_count\_x = getrefcount(x)

**print**(ref\_count\_x) # Output: 3

ref\_count\_b = getrefcount(b)

**print**(ref\_count\_b) # Output: 1

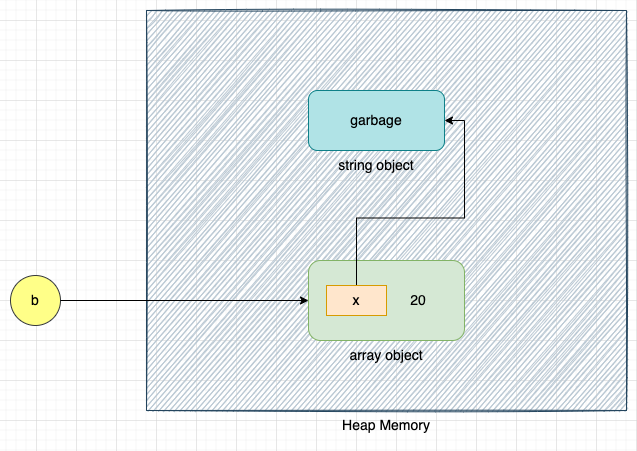


Ignoring the increase in the reference count due to the *getrefcount* method, the reference count of the string object *garbage* is 2 (referenced by variable *x* and referenced from list *b*), and the reference count of the array object is *1*.

Next, let's delete variable *x*:

**del** x

The reference count of the string object *garbage* is 1 due to the array object *[x, 20]*, as shown below:



We can still access the string object *garbage* with the help of variable *b*.

b[**0**]

# Output: garbage

Let's delete the variable *b*:

**del** b

Once the above code is executed, the reference count of the array object becomes 0, and the garbage collector will delete the array object.

Deleting the array object *[x, 20]* will also delete the reference *x* of the string object *garbage* from the array object. This will make the reference count of the *garbage* object 0, and as a result, the *garbage* object will also be collected. Thus, garbage collection of an object may also trigger garbage collection of objects to which the object refers.

Garbage collection based on the reference count is real-time!

Garbage collection gets triggered as soon as the reference count of the object becomes 0. This is the main garbage collection algorithm of Python, and hence, it cannot be disabled.

Garbage collection on the basis of reference count does not work if there are cyclic references. To free or reclaim the memory of the objects that have cyclic references, Python uses a generational garbage collection algorithm.

Next, we’ll discuss cyclic references and then go deeper to understand more about the generational garbage collection algorithm!

**Cyclic References in Python**

A cyclic reference or circular reference is a condition in which an object references itself or when two different objects refer to each other.

Cyclic references are only possible with container objects, such as *list*, *dict*, and user-defined objects. It is not possible with immutable data types, such as *integers*, *floats*, or *strings*.

Consider the following example:

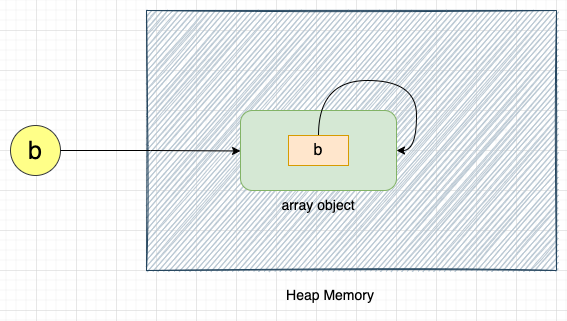
**import** **sys**

b = list()

ref\_count = sys.getrefcount(b) # Output: 2

b.append(b)

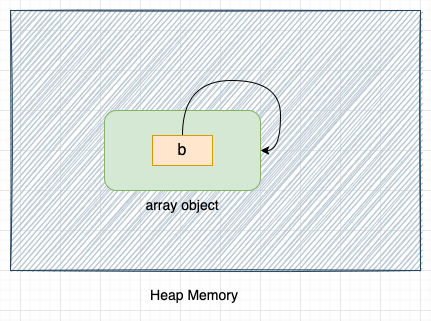
ref\_count = sys.getrefcount(b) # Output: 3



As seen in the above representation, the array object *b* is referencing itself.

Let's delete reference *b*:

**del** b



Once we delete reference *b*, the array object will not be accessible from the Python code, but it will continue to exist in memory, as shown in the above image.

As the array object is referencing itself, the reference count of the array object will never become 0, and it will never be collected by the reference counting garbage collector.

Let's consider another example:

**class** **ClassA**:

**pass**

**class** **ClassB**:

**pass**

A = ClassA()

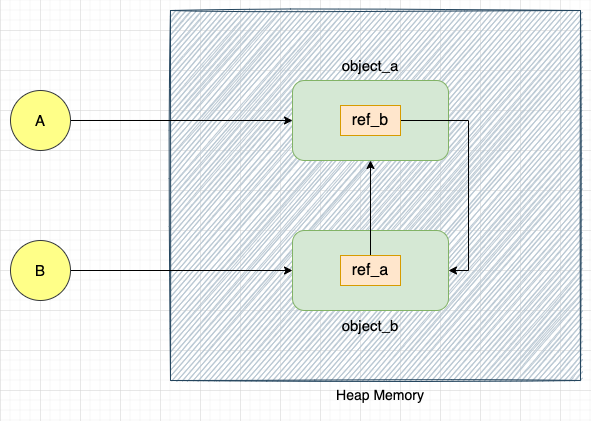
B = ClassB()

# Reference b from a

A.ref\_b = B

# Refer a from b

B.ref\_a = A



Here, we have defined two objects, *object\_a* and *object\_b*, of classes *ClassA* and *ClassB*, respectively. *object\_a* is referred to by variable *A* and *object\_b* is referred to by variable *B*.

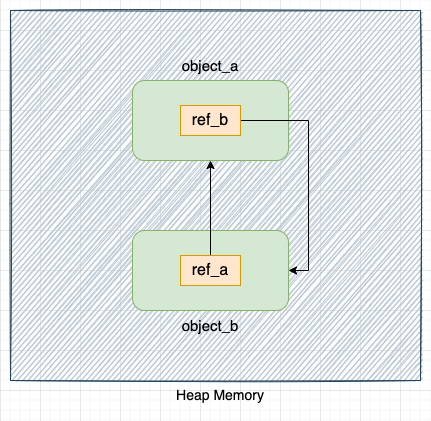
*object\_a* has a property *ref\_b* that points to *object\_b*. Similarly, object *object\_b* has a property *ref\_a* that points to *object\_a*.

*object\_a* and *object\_b* can be accessed in the Python code with the help of variables *A* and *B*, respectively.

Let's delete variables *A* and *B*

**del** A

**del** B



Once we delete variables *A* and *B*, *object\_a* and *object\_b* will not be accessible from the Python codebase, but they will continue to exist in memory.

The reference count of these objects will never become zero, as they point to each other (by properties *ref\_a* and *ref\_b*). Hence, these objects will never be collected by the reference counting garbage collector.

As the reference count of the objects with cyclic references never becomes 0, the reference counting garbage collection method will never clean these objects. For such cases, Python provides another garbage collection algorithm called *Generational Garbage Collection*.

**Generational Garbage Collection in Python**

The generational garbage collection algorithm solves the problem of garbage collecting objects that have circular references. As circular references are only possible with container objects, it scans all the container objects, detects objects with circular references, and removes them if they can be garbage collected.

To reduce the number of objects scanned, generational garbage collection also ignores tuples containing only immutable types (e.g., int and strings).

As scanning objects and detecting cycles is a time-consuming process, the generational garbage collection algorithm is not real-time. It gets triggered periodically. When the generational garbage collection algorithm is triggered, everything else is stopped. Hence, to reduce the number of times generational garbage collection is triggered, CPython classifies container objects into multiple generations and defines the threshold for each of these generations. Generational garbage collection is triggered if the number of objects in a given generation exceeds the defined threshold.

CPython classifies objects into three generations (generations 0, 1, and 2). When a new object is created, it belongs to the first generation. If this object is not collected when CPython runs the generational garbage collection algorithm, it moves to the second generation. If the object is not collected when CPython runs the generational garbage collection again, it is moved to the third generation. This is the final generation, and the object will remain there.

Generally, most objects are collected in the first generation.

When generational garbage collection is triggered for a given generation, it also collects all the younger generations. For example, if garbage collection is triggered for generation 1, it will also collect the objects present in generation 0.

The situation where all the three generations (0, 1, and 2) are garbage collected is called *full collection*. As *full* collection involves scanning and detecting cycles in a large number of objects, CPython tries to avoid *full* collection as much as possible.

We can change the behavior of the generational garbage collector using the *gc* module provided by Python. Generational garbage collection can be disabled using the *gc* module. Hence, it is also called optional garbage collection.

Next, we’ll explain some of the important methods of the *gc* module.

**gc Module in Python**

We can use the following method to get a threshold defined for each generation:

**import** **gc**

gc.get\_threshold()

# Output: (700, 10, 10)

The above output indicates that the threshold for the first generation is *700*, while it is *10* for the second and third generations.

If there are more than 700 objects in the first generation, generational garbage collection will be triggered for all objects in the first generation. If there are more than 10 objects in the 2nd generation, generational garbage collection will be triggered for both generations 1 and 0.

We can also check the number of objects in each generation, as shown below:

**import** **gc**

gc.get\_count()

# Output: (679, 8, 0) # Note: Output will be different on different machines

Here, the number of objects in the first generation is 679, while the number of objects in the second generation is 8, and the number of objects in the third generation is 0.

We can also run the generational garbage collection algorithm manually, as shown below:

**import** **gc**

gc.collect()

*gc.collect()* will trigger the generational garbage collection. By default, it runs a full collection. To run the generational garbage collection for 1st generations we can call this method as shown below:

**import** **gc**

gc.collect(generation=**1**)

Let's check the number of objects that are still alive after the collection process,

**import** **gc**

gc.get\_count()

# Output: (4, 0, 0) # Note: Output will be different on different machines

We can also update the generational threshold for each generation, as shown below:

**import** **gc**

gc.get\_threshold()

gc.set\_threshold(**800**, **12**, **12**)

gc.get\_threshold()

# Output: (800, 12, 12) # Note: Output will be different on different machines

We can disable generational garbage collection using the following command:

gc.disable()

Conversely, we can enable it again using the following command:

gc.enable()