Session 5

Data structures

Prepared by/ Elzahraa Alaa Tag Eldein

April 6, 2023

Task 1

mention examples for programming languages that have garbage collection and mention examples for those that doesn't have it

What is Garbage Collection?

In general layman's terms, Garbage collection (GC) is nothing but collecting or gaining memory back which has been allocated to objects but which is not currently in use in any part of our program.

Let's get into more detail. Garbage collection is the process in which programs try to free up memory space that is no longer used by objects.

Garbage collection is implemented differently for every language. Most high-level programming languages have some sort of garbage collection built in. Low-level programming languages may add garbage collection through libraries.

As said above, every programming language has their own way of performing GC. In C programming, developers need to take care of memory allocation and deallocation using malloc() and dealloc() functions. But, in the case of C# developers don't need to take care of GC and it’s not recommended either.

In computer science, **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; such memory is called *garbage*. Garbage collection was invented by American computer scientist John McCarthy around 1959 to simplify manual memory management in Lisp.[2]

Garbage collection relieves the programmer from doing manual memory management, where the programmer specifies what objects to de-allocate and return to the memory system and when to do so.[3] Other, similar techniques include stack allocation, region inference, and memory ownership, and combinations thereof. Garbage collection may take a significant proportion of a program's total processing time, and affect performance as a result.

Many programming languages require garbage collection, either as part of the language specification (e.g., RPL, Java, C#, D,[4] Go, and most scripting languages) or effectively for practical implementation (e.g., formal languages like lambda calculus). These are said to be *garbage-collected languages*. Other languages, such as C and C++, were designed for use with manual memory management, but have garbage-collected implementations available. Some languages, like Ada, Modula-3, and C++/CLI, allow both garbage collection and manual memory management to co-exist in the same application by using separate heaps for collected and manually managed objects. Still others, like D, are garbage-collected but allow the user to manually delete objects or even disable garbage collection entirely when speed is required.

Advantages

GC frees the programmer from manually de-allocating memory. This helps avoid some kinds of errors:

- *Dangling pointers*, which occur when a piece of memory is freed while there are still pointers to it, and one of those pointers is dereferenced. By then the memory may have been reassigned to another use, with unpredictable results.
- *Double free bugs*, which occur when the program tries to free a region of memory that has already been freed, and perhaps already been allocated again.
- Certain kinds of *memory leaks*, in which a program fails to free memory occupied by objects that have become unreachable, which can lead to memory exhaustion.[6]

Disadvantages [edit]

GC uses computing resources to decide which memory to free. Therefore, the penalty for the convenience of not annotating object lifetime manually in the source code is overhead, which can impair program performance.[7] A peer-reviewed paper from 2005 concluded that GC needs five times the memory to compensate for this overhead and to perform as fast as the same program using idealised explicit memory management. The comparison however is made to a program generated by inserting deallocation calls using an oracle, implemented by collecting traces from programs run under a profiler, and the program is only correct for one particular execution of the program.[8] Interaction with memory hierarchy effects can make this overhead intolerable in circumstances that are hard to predict or to detect in routine testing. The impact on performance was given by Apple as a reason for not adopting garbage collection in iOS, despite it being the most desired feature.[9]

The moment when the garbage is actually collected can be unpredictable, resulting in stalls (pauses to shift/free memory) scattered throughout a session. Unpredictable stalls can be unacceptable in real-time environments, in transaction processing, or in interactive programs. Incremental, concurrent, and real-time garbage collectors address these problems, with varying trade-offs.

C (and other non-garbage-collecting languages) *has no concept of* ***garbage*** at all, and thus no need to collect it somehow - Either you hold a valid pointer to some allocated memory, then it's considered "valuable memory", or you don't, then your program is just wrong - It's as simple as that.

The latter case is something C doesn't even evaluate any further - There's no point in researching what happens in a program "that's wrong" other than fixing it.

Languages like C and C++ use dynamic heap allocation through dedicated functions/operators like `malloc` and `new`. This allocates memory on the heap, in RAM. If such a program fails to free the memory once done using it, then the programmer has managed to create a certain kind of bug called *memory leak*. Meaning that the program now consumes heap memory that cannot be used, since there is nothing in the program pointing at it any longer.

However, all memory allocated by a process is freed by the OS when the process is done executing. If the process failed to clean up its own heap allocations, the OS will do it. It is still good practice to manually clean up the memory though, but for other reasons (exposes latent bugs).

Therefore the only concern with memory leaks is that they cause programs to consume too much RAM while they execute. Once the process is done executing, all memory - including leaked memory - is freed.

There is no relation between the heap and your hard drive, just as there is no relation between the stack and your hard drive. The hard drive is used for storing the executable part of your program, nothing else. The heap, stack and other such memory areas are for storing data when your program is executing. Since they are allocated in RAM, all info in such areas is lost when the program is done executing.

Garbage Collection:

Garbage collection is to release memory when the object is no longer in use. This system destroys the unused object and *reuses* its memory slot for new objects. You can imagine this as a recycling system in computers.

Python has an automated garbage collection. It has an algorithm to deallocate objects which are no longer needed. Python has two ways to delete the unused objects from the memory.

tds Published in Towards Data Science

Seyma Tas
Oct 2, 2020 · 6 min read · Listen

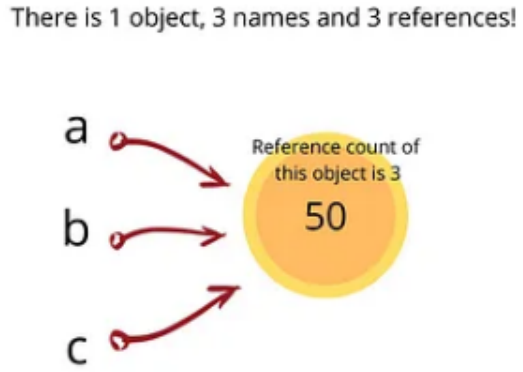
1. Reference counting:

The references are always counted and stored in memory.

In the example below, we assign c to 50. Even if we assign a new variable, the object is the same, the reference count increases by 1! Because every object has its own ID, we print the IDs of objects to see if they are the same or different.

```
a = 50
b = a
c = 50
print(id(a))
print(id(b))
print(id(c))
```

```
4364962480
4364962480
4364962480
```




```
print(a is b)
print(c is b)
print(a is c)
```

```
True
True
True
```

Image by author made with Canva

tds Published in Towards Data Science

Seyma Tas
Oct 2, 2020 · 6 min read · Listen

When we change the value of **a** like in below, we create a new object. Now, **a** points to 60, **b** and **c** point to 50.

When we change **a** to None, we create a none object. Now the previous integer object has no reference, it is deleted by the garbage collection.

We assign **b** to a boolean object. The previous integer object is not deleted because it still has a reference by **c**.

tds Published in Towards Data Science

Seyma Tas
Oct 2, 2020 · 6 min read · Listen

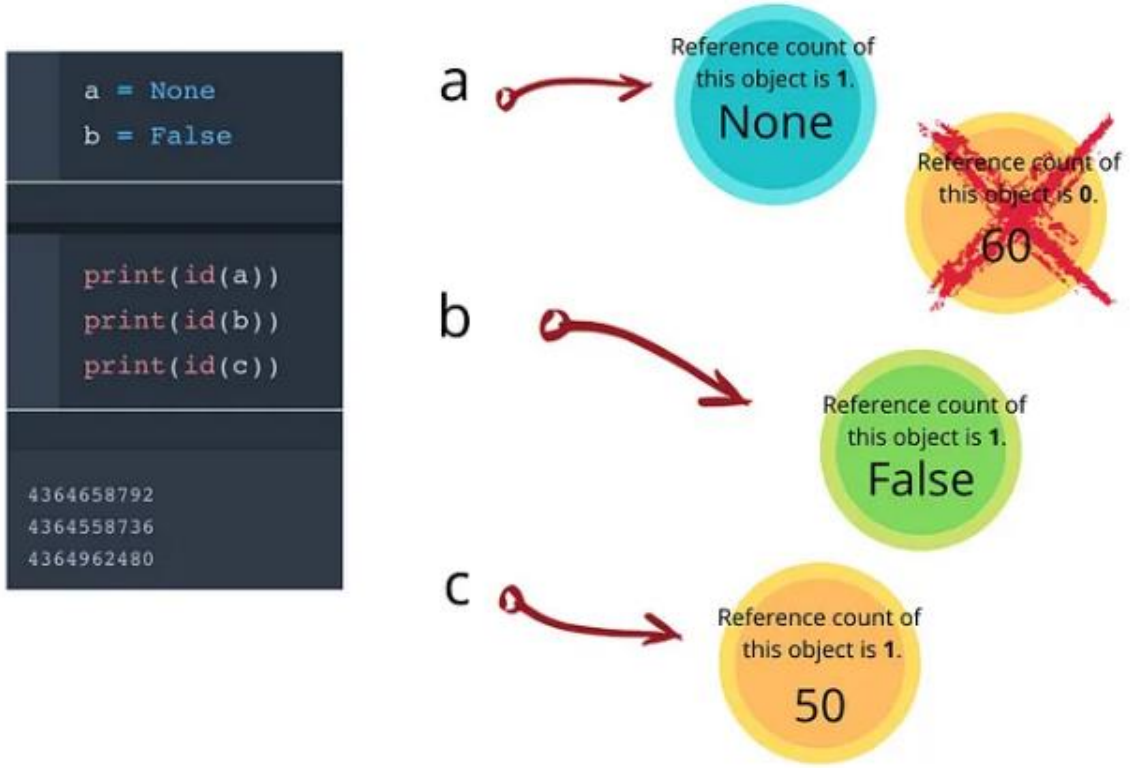


Image by author made with Canva

Now we delete c. We decrease the reference count to c by one.

Seyma Tas

Oct 2, 2020 · 6 min read · Listen

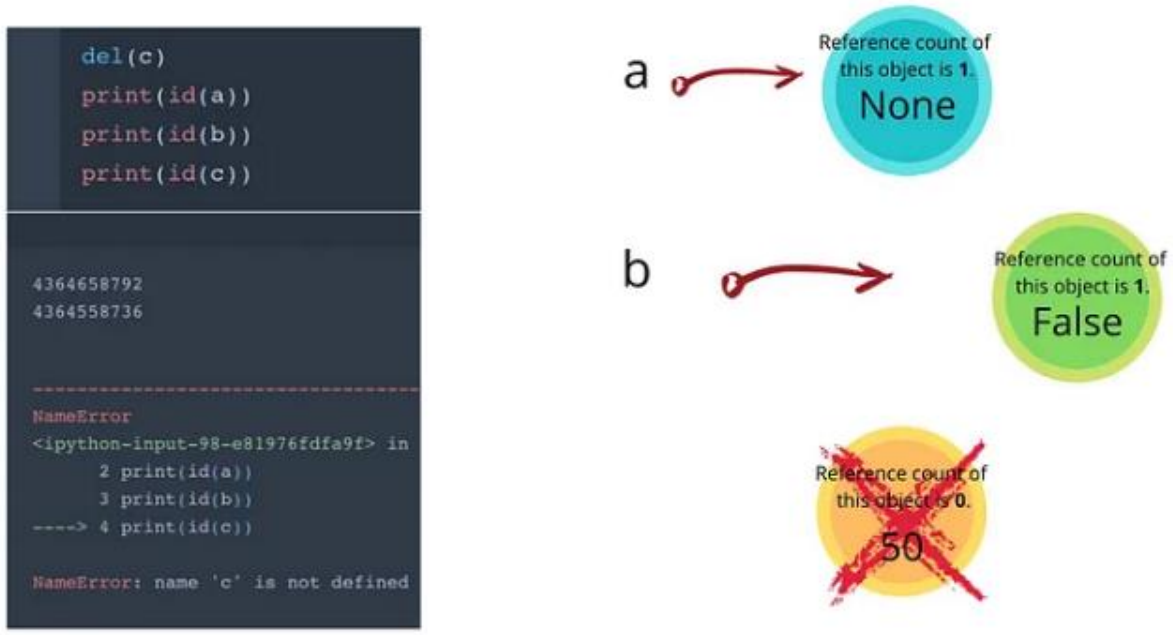


Image by author made with Canva

tds Pub

As you can see above, **del()** statement doesn't delete objects, it removes the name (and reference) to the object. When the reference count is zero, the object is deleted from the system by the garbage collection.

Seyma Tas

Oct 2, 2020 · 6 min read · Listen

Goods and bads of reference counting:

There are advantages and disadvantages of garbage collection by reference counting. For example, it is easy to implement. Programmers don't have to worry about deleting objects when they are no longer used. However, this memory management is bad for memory itself! The algorithm always counts the reference numbers to the objects and stores the reference counts in the memory to keep the memory clean and make sure the programs run effectively.

Everything looks ok until now, but ...

There is a problem!

The most important issue in reference counting garbage collection is that it doesn't work in cyclical references.

tds Published in Towards Data Science

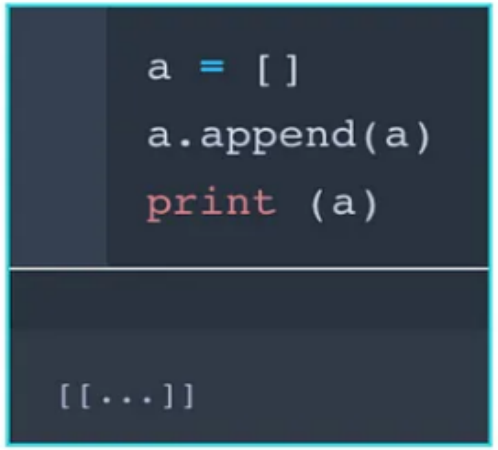
Seyma Tas
Oct 2, 2020 · 6 min read · Listen

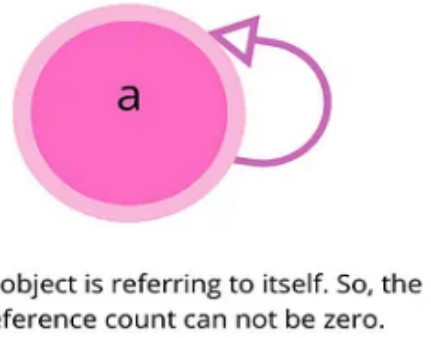
[i]NSTANT

AI Diploma R27

What is a cyclical reference or reference cycle?

It is a situation in which an object refers to itself. The simplest cyclical reference is appending a list to itself.





The simplest cyclical reference. Image by author made with Canva

Reference counting alone can not destroy objects with cyclic references. If the reference count is not zero, the object cannot be deleted.

The solution to this problem is the second garbage collection method.

tds Published in Towards Data Science

Seyma Tas
Oct 2, 2020 · 6 min read · Listen

2. Generational Garbage Collection:

Generational garbage collection is a type of trace-based garbage collection. It can break cyclic references and delete the unused objects even if they are referred by themselves.

How does generational Garbage Collection work?

Python keeps track of every object in memory. 3 lists are created when a program is run. Generation 0, 1, and 2 lists.

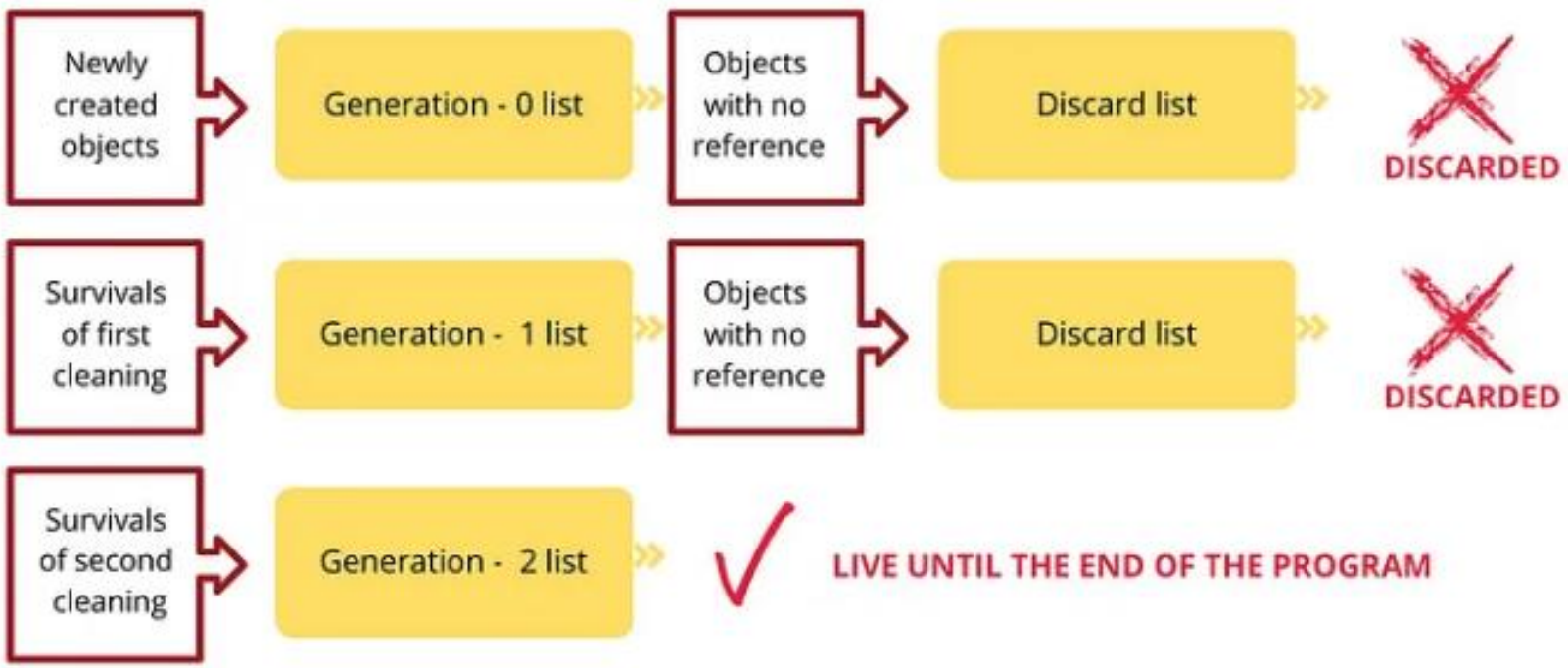
Newly created objects are put in the Generation 0 list. A list is created for objects to discard. Reference cycles are detected. If an object has no outside references it is discarded. The objects who survived after this process are put in the Generation 1 list. The same steps are applied to the Generation 1 list. Survivals from the Generation 1 list are put in the Generation 2 list. The objects in the Generation 2 list stay there until the end of the program execution.

tds Published in Towards Data Science

Seyma Tas
Oct 2, 2020 · 6 min read · Listen

[i]NSTANT

AI Diploma R27



Generational garbage collection. Image by author made with Canva

tds Published in Towards Data Science

Seyma Tas

Oct 2, 2020 · 6 min read · Listen

Conclusion:

Python is a high-level language and we don't have to do the memory management manually. Python garbage collection algorithm is very useful to open up space in the memory. Garbage collection is implemented in Python in two ways: reference counting and generational. When the reference count of an object reaches 0, reference counting garbage collection algorithm cleans up the object immediately. If you have a cycle, reference count doesn't reach zero, you wait for the generational garbage collection algorithm to run and clean the object. While a programmer doesn't have to think about garbage collection in Python, it can be useful to understand what is happening under the hood.

tds Published in Towards Data Science

Seyma Tas

Oct 2, 2020 · 6 min read · Listen

Task 5

mention examples for programming languages that have garbage collection and mention examples for those that doesn't have it

Runtime configuration options for garbage collection

Article • 02/14/2023 • 17 minutes to read • 6 contributors

Feedback

This page contains information about settings for the .NET runtime garbage collector (GC). If you're trying to achieve peak performance of a running app, consider using these settings. However, the defaults provide optimum performance for most applications in typical situations.

Settings are arranged into groups on this page. The settings within each group are commonly used in conjunction with each other to achieve a specific result.

Links and references

https://www.freecodecamp.org/news/a-guide-to-garbage-collection-in-programming/

https://towardsdatascience.com/memory-management-and-garbage-collection-in-python-c١cb٥١d١٦١٢c

https://stackoverflow.com/questions/-gnimmargorp/٤٤٥٨٧١٧٤-egabrag-eht-seod-erehw-rotcelloc-egabrag-tuohtiw-segaugnal%20sihT=txet:~:#ogis%20one%20of%20many,those%20languages%20prioritize%20execution%٢٠speed.

https://learn.microsoft.com/en-us/dotnet/core/runtime-config/garbage-collector