Garbage Collection in Ruby and Python

## Author: Ekrar Uddin Mohammed Efaz

## **Student ID : 103494172**

## Unit: Introduction to Programming

## 

## 2

## Table of Contents

|  |  |  |
| --- | --- | --- |
|  | Topic | Page |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |
| 7 |  |  |

3

## Objectives

This programming literary paper serves to Introduce Garbage Collection to beginners in an epitomized fashion without getting too deep into complex technicality. This paper introduces Garbage collection process in two popular high-level programming languages Python and Ruby. This paper also introduces the reader to GC Algorithms, advantages and disadvantages of GC, improvements in GC that happened overtime through abstract images, code and psuedo code.

1. Famillarize reader with Basic Memory Managment (Heaps & Stacks).
2. Introduce Garbage Collection Process and it’s History.
3. Clairfy the GC Algorithms used in Ruby and Python.
4. Display Simple Code for practical explaination the GC process.

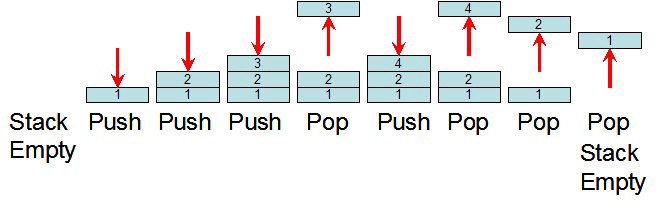
## Understand your Program’s Memory

Memory is divided into multiple segments. Mainly, we talk about two segements. They are:

**1. the stack**.

**2. the heap**.

**Stack:** The stack is a Linear Data structure. It stores the frames that manage the function and procedure calls, as well as the variables that are created. Stack, as the name suggestes piles up new data one top of another. The stack has a set of fixed operations for its work. It is a Last-in-First-Out data structure.[[1]](#footnote-2) When a function is called a stack frame is push on top and when the function returns the stack frame is popped. Whatever is pushed onto the stack is eventually popped after it returns a value.**[[2]](#footnote-3)** The Programmer does not have to worry about the memory allocation in stack. It cleans up the variables created for the programmer.

 **Heap :** The heap is a heirachical data structure. It is like a file cabinet. It has more space but it also takes a lot of time to go through the contents and find the required item.[[3]](#footnote-4) Simillarly, in programming we use heaps to store data which we want to exist for sometime and are fairly larger like custom data-types and objects. Sometimes if a data structure is large enough that it can’t be stored in stack space, that is stored in heap and a pointer to that data is stored in the stack. Heaps are messy compared to neatly organized stacks. All the objects in a program lives in the heap.

### Stack VS Heap in Memory

**Lets raise the question:** *Why do we need stacks , why can’t we keep everything in Heap?*

Lets think of Stacks as a pile of paper on the desk. You can easily reach the most recent work from the top. Stacks provide a more faster and efficient use of memory for smaller data structures like int and str. The programmer doesn’t have to think about the stack space memory management as it is done automatically. But things get more complicated as the size of data structures grow, for example when we have an object. Creating objects on the fly on stack space is very resource intensive thats why objects are created in heap space and a pointer to that space is kept in stack so that it can easily be accessed. Therefore an object can live longer in memory and whenever the object is instantiatied the program stores a pointer to it in stack space. Memory fragmentation is also an issue in Heap space because of inefficient use of memory.[[4]](#footnote-5)



STACK HEAP

## Garbage Collector

#### What is Garbage Collector?

Garbage Collection(GC) is a feature to automatically detect and free unused memory.[[5]](#footnote-6) When Objects are created a lot of memory is consumed. It wouldn’t be a problem if we had unlimited memory,but in reality we have to work with limited memory. So, whenever we aren’t using an object we have to remove it from using memory. In low-level programming languages such as C we manually allocate memory using **malloc()** and free memory segments using **free()[[6]](#footnote-7)**. In object-oriented programming languages memory management is done by Garbage Collector.

#### History of Garbage Collector

Garbage Collector was introduced by John McCarthy in 1950s as a part of automatic memory management for his development of LISP.



*John McCarthy Source:* *https://s.wsj.net/public/resources/images/NA-BN878\_REMMCC\_G\_20111025180845.jpg*

Surprisingly, the algorithm used by John 58 years ago is still in use by ruby for for it’s Garbage Collector. But Python uses a different type of GC Algorithm known as **Reference Counting** which was introduced by George Collins, conincidentally in the same decade as GC by John McCarthy.

### Garbage Collection in Ruby

The first misconception we need to clear is that Garbage Collector on frees up unused memory. Although freeing memory is one of its primary tasks, a Garbage Collector also allocates memory. It is more like a complete memory management system doing everything behind the scenes. At a high-level a Garbage Collector has two key responsibilites:[[7]](#footnote-8)

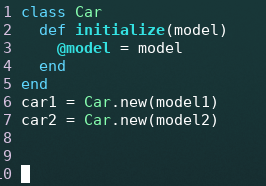
1. Allocating space for new objects

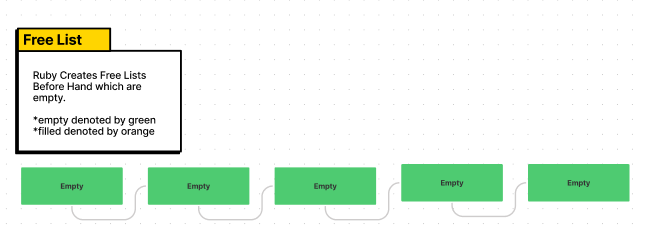
2. Reclaim space from dead objects

Ruby being a High-level modern programming language doesn’t require the programmer to manage most of the memory. It has a built in Garbage Collector to do the dirty work.

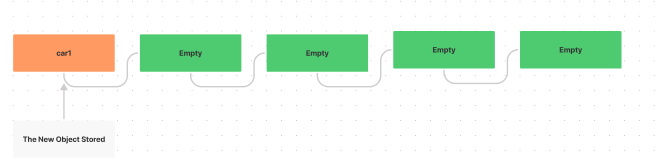
**Ruby Memory Allocation:**

Lets say we create two new objects using the following code below:



Ruby, long before our code starts to run creates objects ahead of time and places them on a linked list[[8]](#footnote-9) formally know as the *free list.* A free list conceptually might look something like this:

Whenever we call the **Car.new** ruby takes one of the precreated objects in free list and stores the created object there. Eventually every new object is put in one of the precreated object spaces by free list.



**Algorithm of Ruby Garbage Collector[[9]](#footnote-10)**

The ruby garbage collector follows the ***mark and sweep*** garbage collection algorithm first introduced by John McCarthy for his programming language Lisp. Any garbage collector has to do two tasks. One is to indentify unreachable objects and the other to free up the space taken by those objects.***Mark and Sweep*** *algorithm* does this in two phases namely:

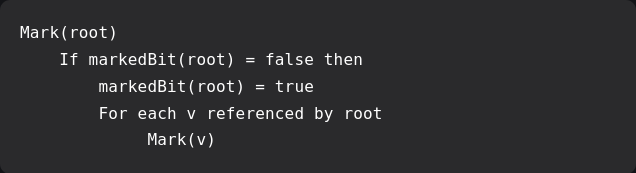
1.Marking Phase

2.Sweeping Phase

**Mark Phase:** When an object is created, its mark bit is set to 0. In *Mark Phase,* the objects an user can refer to has their mark bit set to 1. We perform this operation we simply need to do a graph traversal that is a depth-first approach. Every object is a node and the nodes that are reachable are all visited and their mark bit is set. Marking is a recurvsive approach.

*Psuedo Code For Mark Phase[[10]](#footnote-11)*

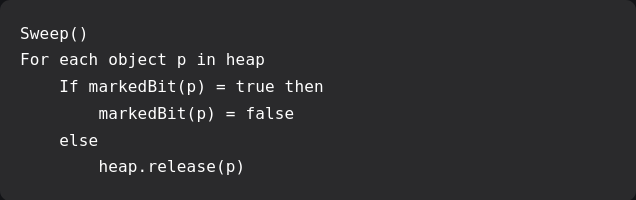
\*root is a variable that refers to an object and we assume that there is a single root



The whole program execution is stopped while the garbage collector is running. This process is called ***stop-the-world*** garbage collection. It has to do this so that your program doesn’t allocate new memory while the collector is working to free up space.

**Sweep Phase:** The sweep phase “sweeps” the unreachable objetcs. It clears the heap memory for all the unreachable objects i.e the objects whoose mark bit is set to 0. And all the reachable object mark is set to 0 too. The algorithm runs continiously we go through the mark and sweep phase again and again.

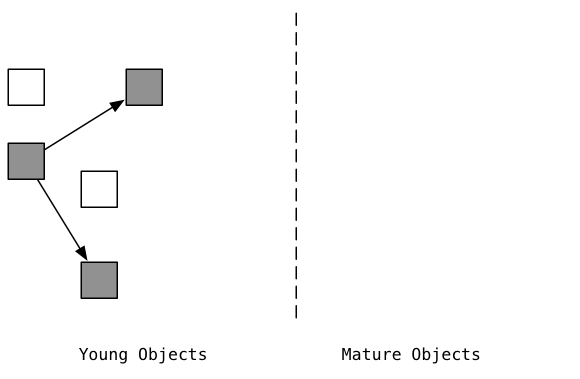
*Psuedo Code For Sweep Phase[[11]](#footnote-12)*



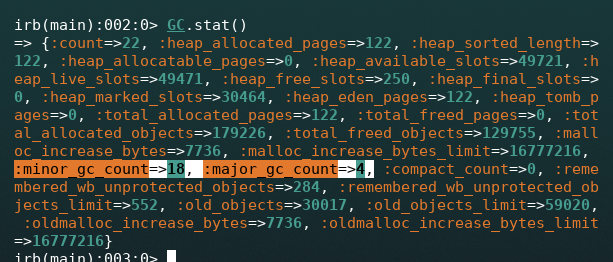
## Improvements in Ruby GC

### Generational Garbage Collection:

*Ruby 2.1 added Generational GC[[12]](#footnote-13).* Genrational Garbage Collection is an improvement to old GC algorithm that has two separate spaces for “young” and “old” objects.

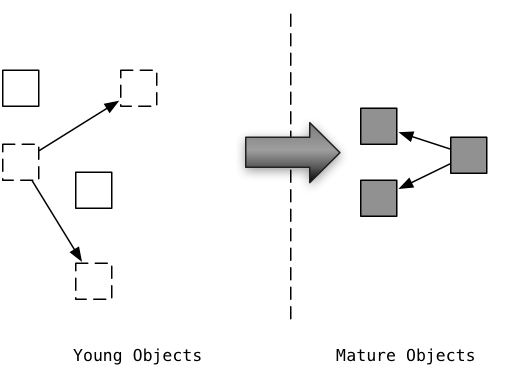


It comes off from the idea that some objects are more likely to be used briefly and aren’t needed as much compared to some objects which are required often. Therefore, the process is divided into a *major phase* and *minor phase.*The “old” objects are checked only during *major phase* and *minor phase* is run more often to check the “young” object.



*:count = number of gc cycles, :minor\_count = major collection, :major\_count = minor collection*

*:heap\_free\_slots = number of free slots in heap*

If an object survives three garbage collection runs it is promoted to an “old” object space.

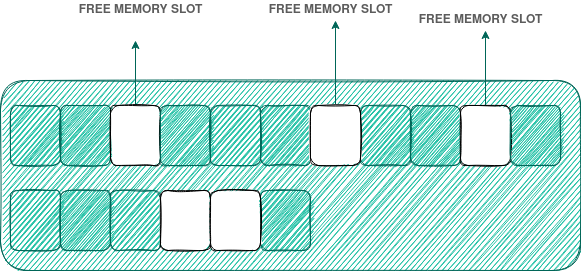
If program cannot find space to create new objects in “young” object space then GC clears space from “old” object space. The minor GC takes less time, so the time spent in garbage collecting is much less than usual. Usually the metric that most GCs are judged on is the pause times[[13]](#footnote-14). Minor GCs have much less pause times compared to Major GCs. Thus an improvement in performance because even if more demanding programs are loaded the Minor GCs has an improved pause time.

### Compaction

To talk about what compaction does, we first have to address the problem.

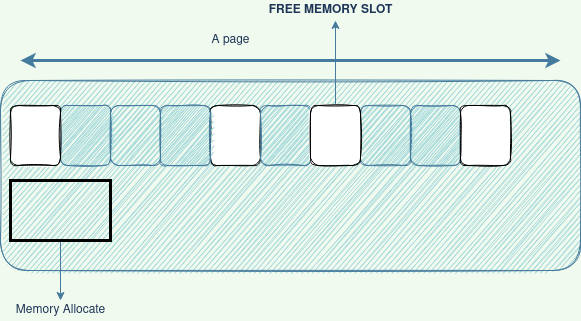
**What is Compaction Actually aiming to solve?**

The problem compaction aims to solve is *Heap Fragmentation.* Heap Fragmentation is popularly known as memory bloat and is a topic of frequent discussion among developers. Memory fragmentation happens when objects allocation in memory is done in a scatterted way. The Garbage Collection marks ruby heap pages as free and when all of its slots are free it is released for allocation. But if some slots are not free that entire page is stuck with empty and filled slots.



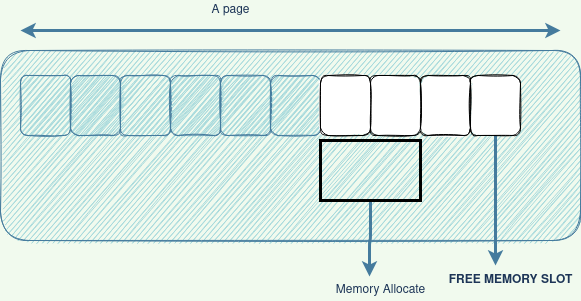
If this case is for large amount of memory pages then memory fragmentation happens where free fragments of memory are scattered rendered unusable. We see compaction as a solution to this problem.

**What is Compaction in GC?[[14]](#footnote-15)**

Compaction is the process of rearranging memory so that the empty slots are usable. Main aim is to have efficient memory usage and avoid bloating. Lets say we want to allocate memory to an object.

But the scattered memory slots are smaller than the required memory so therefore we end up with a situation called “out of memory”. New pages need to be added and all the pages have some unused fragmented memory slots that are reducing memory efficiency.

Thus, Compaction comes into play. Compaction rearranges the fragmented memory in a way that they are all bunched together allowing maximum usage before moving onto new pages.



Compaction in ruby allows more memory to be used efficiently reducing memory bloating also “copy on write friendliness” is something compaction tends to. When a ruby process is forked its child process copies its parent’s page structure which also means it’s got fragmentation if it’s parent got fragmentation leading to more memory waste. Now if a page is compacted it can prevent child processes to inherit fragmentation. Currently, Compaction is being worked on for further improvements by Ruby Core Team.

### Garbage Collection in Python

In Python everything is an object. Allocating memory to an object is simple, but freeing that memory automatically is a complex task that requires two algorithms. Python figures out when your object is not needed and then removes that object and frees memory. This process is done by reference counting and although very efficient it fails to deal with cyclic references where an object makes a reference to itself. That’s where Generational GC kicks in which manages self-referencing.

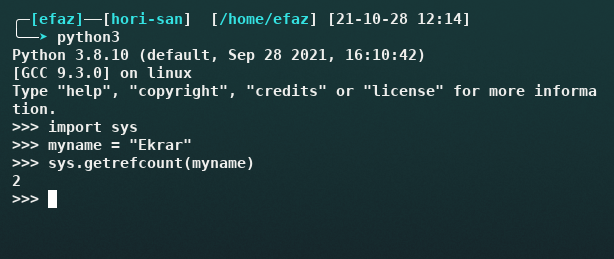
Python basically uses 2 algorithms to make the garbage collection process efficient:

1. Reference Counting

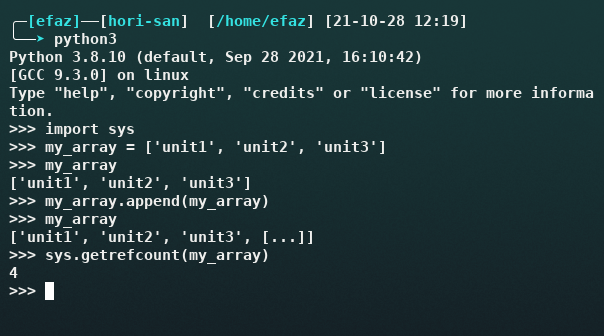
2. Generational GC

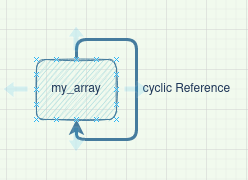
**Algorithm of Python Garbage Collector**

**1. Reference Couning:[[15]](#footnote-16)** InReference counting, python stores an integer value inside an object, known as *reference count,* keeping track of how many references are made to that object. When the reference count reaches 0, the GC removes the object and releases the memory back to the system for allocation.Using Python’s ***sys***module we can ask the program for the reference count of an object.

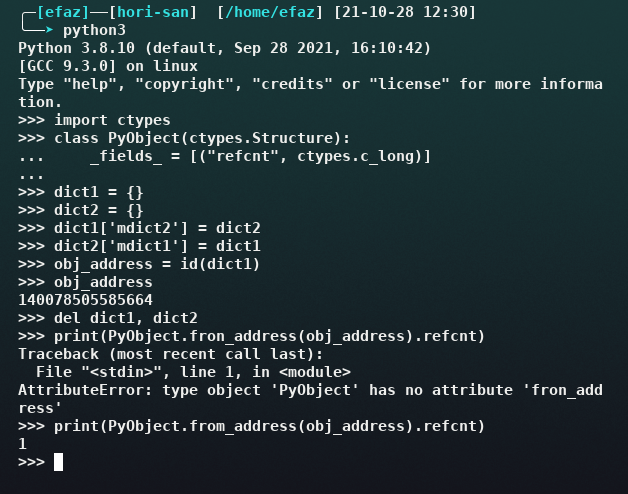


Here we have a string object in python stored in the variable *myname*. When we make a systemcall asking for the reference count of that object, we get 2. Python keeps track of the reference count during the runtime of the program and deletes the objects which have no references to them. This is very efficient except for when objects make a reference to themsleves. In that case as long as the object exists it’s reference count never reaches 0 and as long as it’s reference is not 0 it is never removed. Just like a paradox!. Self-referencing might sound a vague for first-timers but it is a very common task we implement even in simplest of program.





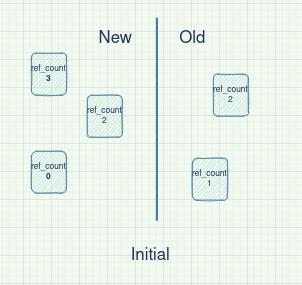
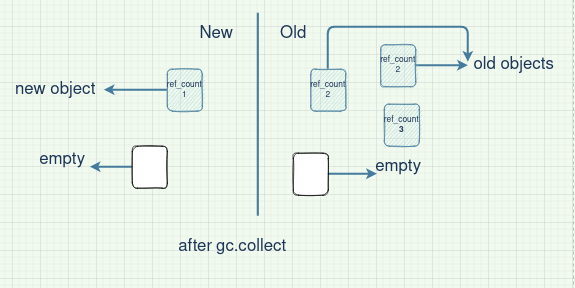
Here, we append an array to itself. It’s reference count never reach 0 because the array makes a reference to itself. In such cases memory is never freed and surprisingly it is a very common situation

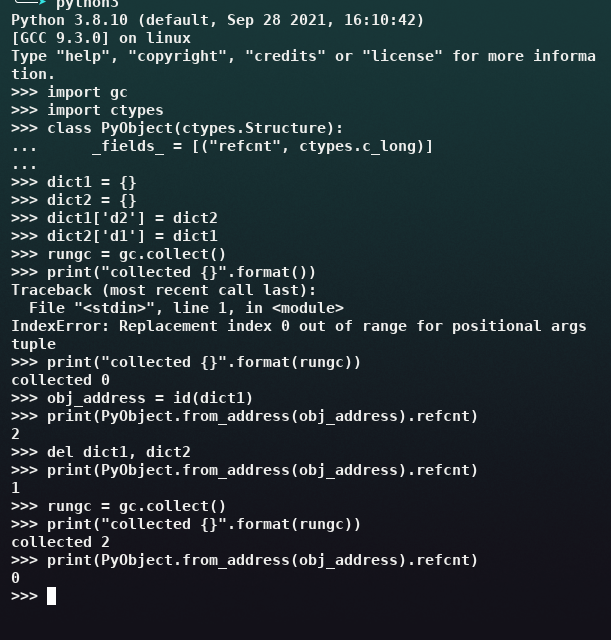


Here although we delete the objects *dict1* and *dict2* the reference count to dict1 is still 1 because of the cyclic references made earlier.

**2. Generational Garbage Collection in Python[[16]](#footnote-17)**

Generational GC in python is simillar to Ruby in many ways. Generational GC looks at objects in two generation based classification. First Generation are newer objects and when an object survives a GC run it is promoted to Older Generation. GC looks at older generation more than newer objects because new objects die young. The collection intiation however depends on GC threshold. To specify the collection behaviour gc module of python provides functionality to custom set thresholds to determine when a GC process should run.



In the Sample code we do a manual GC run to understand the process of Generational GC. First we create objects with reference cycles. Their reference count is 2. Even after we delete the objects their memory has a reference count of 1. Only after we run *gc.collect()* that our memory reference count goes to 0 and the memory is freed.

Generational GC in python is also an improvement over the original. This makes the GC look at lesser object at a time and decreases the pause time of the program while GC runs inturn making the program more efficient.

1. Reference : https://ruby-hacking-guide.github.io/gc.html [↑](#footnote-ref-2)
2. Reference : https://www.freecodecamp.org/news/understand-your-programs-memory-92431fa8c6b/ [↑](#footnote-ref-3)
3. Reference : https://www.freecodecamp.org/news/understand-your-programs-memory-92431fa8c6b/ [↑](#footnote-ref-4)
4. Reference : https://www.guru99.com/stack-vs-heap.html [↑](#footnote-ref-5)
5. Reference : https://ruby-hacking-guide.github.io/gc.html [↑](#footnote-ref-6)
6. Reference : https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/lecture-slides/MIT6\_172F18\_lec11.pdf [↑](#footnote-ref-7)
7. Reference : https://www.educative.io/courses/a-quick-primer-on-garbage-collection-algorithms/jy6v [↑](#footnote-ref-8)
8. Linked list: a linear collection of data elements whose order is not given by their physical placement in memory. [↑](#footnote-ref-9)
9. Reference : https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/ [↑](#footnote-ref-10)
10. Reference : https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/ [↑](#footnote-ref-11)
11. Reference : https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/ [↑](#footnote-ref-12)
12. Reference : https://stackify.com/how-does-ruby-garbage-collection-work-a-simple-tutorial/ [↑](#footnote-ref-13)
13. Pause Times : Pause Times are how the world is stoppped for each GC run [↑](#footnote-ref-14)
14. Reference: https://www.youtube.com/watch?v=H8iWLoarTZc&t=1852s (Talk by a core ruby developer working on compaction) [↑](#footnote-ref-15)
15. Reference: https://www.pythontutorial.net/advanced-python/python-garbage-collection/ [↑](#footnote-ref-16)
16. Reference: https://rushter.com/blog/python-garbage-collector/ [↑](#footnote-ref-17)