Python Memory Management

In this tutorial, we will learn how Python manages the memory or how Python handles our date internally. We will dive deep into this topic to understand internal working of Python and how it handles the memory.

This tutorial will give a deep understanding of Python memory management. When we execute our Python script, there are so many logic runs behind in Python memory to make the code efficient.

Introduction

Memory management is very important for software developers to work efficiently with any programming language. As we know, Python is a famous and widely used programming language. It is used almost in every technical domain. In contrast to a programming language, memory management is related to writing memory-efficient code. We cannot overlook the importance of memory management while implementing a large amount of data. Improper memory management leads to slowness on the application and the server-side components. It also becomes the reason of improper working. If the memory is not handled well, it will take much time while preprocessing the data.

In Python, memory is managed by the Python manager which determines where to put the application data in the memory. So, we must have the knowledge of Python memory manager to write efficient code and maintainable code.

Let's assume memory looks like an empty book and we want to write anything on the book's page. Then, we write data any data the manager find the free space in the book and provide it to the application. The procedure of providing memory to objects is called **allocation.**

On the other side, when data is no longer use, it can be deleted by the Python memory manager. But the question is, how? And where did this memory come from?

Python Memory Allocation

Memory allocation is an essential part of the memory management for a developer. This process basically allots free space in the computer's virtual memory, and there are two types of virtual memory works while executing programs.

- Static Memory Allocation
- Dynamic Memory Allocation

Static Memory Allocation -

Static memory allocation happens at the compile time. For example - In C/C++, we declare a static array with the fixed sizes. Memory is allocated at the time of compilation. However, we cannot use the memory again in the further program.

```
static int a=10;
```

Stack Allocation

The Stack data structure is used to store the static memory. It is only needed inside the particular function or method call. The function is added in program's call stack whenever we call it. Variable assignment inside the function is temporarily stored in the function call stack; the function returns the value, and the call stack moves to the text task. The compiler handles all these processes, so we don't need to worry about it.

Call stack (stack data structure) holds the program's operational data such as subroutines or function call in the order they are to be called. These functions are popped up from the stack when we called.

Dynamic Memory Allocation

Unlike static memory allocation, Dynamic memory allocates the memory at the runtime to the program. For example - In C/C++, there is a predefined size of the integer of float data type but there is no predefine size of the data types. Memory is allocated to the objects at the run time. We use the **Heap** for implement dynamic memory management. We can use the memory throughout the program.

```
int *a;
p = new int;
```

As we know, everything in Python is an object means dynamic memory allocation inspires the Python memory management. Python memory manager automatically vanishes when the object is no longer in use.

Heap Memory Allocation

Heap data structure is used for dynamic memory which is not related to naming counterparts. It is type of memory that uses outside the program at the global space. One of the best advantages of heap memory is to it freed up the memory space if the object is no longer in use or the node is deleted.

In the below example, we define how the function's variable store in the stack and a heap.

Default Python Implementation

Python is an open-source, object-oriented programming language which default implemented in the C programming language. That's very interesting fact - A language which is most popular written in another language? But this is not a complete truth, but sort of.

Basically, Python language is written in the English language. However, it is defined in the reference manual that isn't useful by itself. So, we need an interpreter based code on the rule in the manual.

The benefit of the default implementation, it executes the Python code in the computer and it also converts our Python code into instruction. So, we can say that Python's default implementation fulfills the both requirements.

> Note - Virtual Machines are not the physical computer, but they are instigated in the software.

The program that we write using Python language first converts into the computer-relatable instructions **bytecode**. The virtual machine interprets this bytecode.

Python Garbage Collector

As we have explained earlier, Python removes those objects that are no longer in use or can say that it frees up the memory space. This process of vanish the unnecessary object's memory space is called the Garbage Collector. The Python garbage collector initiates its execution with the program and is activated if the reference count falls to zero.

When we assign the new name or placed it in containers such as a dictionary or tuple, the reference count increases its value. If we reassign the reference to an object, the reference counts decreases its value if. It also decreases its value when the object's reference goes out of scope or an object is deleted.

As we know, Python uses the dynamic memory allocation which is managed by the Heap data structure. Memory Heap holds the objects and other data structures that will be used in the program. Python memory manager manages the allocation or de-allocation of the heap memory space through the API functions.

Python Objects in Memory

As we know, everything in Python is object. The object can either be simple (containing numbers, strings, etc.) or containers (dictionary, lists, or user defined classes). In Python, we don't need to declare the variables or their types before using them in a program.

Let's understand the following example.

Example -

```
a= 10
print(a)
 del a
print(a)
```

Output:

```
10
Traceback (most recent call last):
  File "", line 1, in
    print(x)
NameError : name 'a' is not defined
```

As we can see in the above output, we assigned the value to object x and printed it. When we remove the object x and try to access in further code, there will be an error that claims that the variable x is not defined.

Hence, Python garbage collector works automatically and the programmers doesn't need to worry about it, unlike C.

Reference Counting in Python

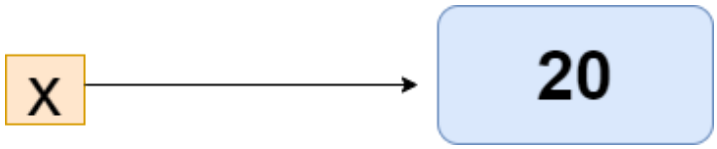
Reference counting states that how many times other objects reference an object. When a reference of the object is assigned, the count of object is incremented one. When references of an object are removed or deleted, the count of object is decremented. The Python memory manager performs the de-allocation when the reference count becomes zero. Let's make it simple to understand.

Example -

Suppose, there is two or more variable that contains the same value, so the Python virtual machine rather creating another object of the same value in the private heap. It actually makes the second variable point to that the originally existing value in the private heap.

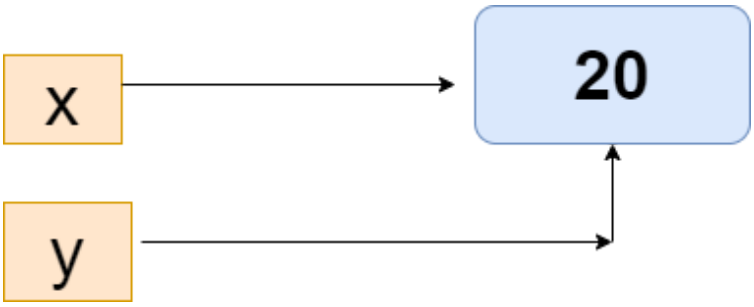
This is highly beneficial to preserve the memory, which can be used by another variable.

```
x = 20
```



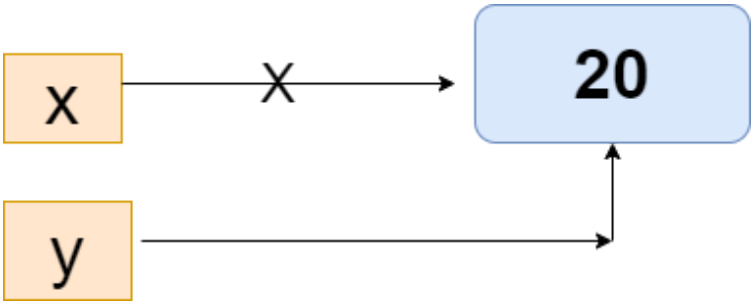
When we assign the value to the x. the integer object 10 is created in the Heap memory and its reference is assigned to x.

```
x = 20
y = x
if id(x) == id(y):
   print("The variables x and y are referring  to the same object")
```



In the above code, we have assigned y = x, which means the y object will refer to the same object because Python allocated the same object reference to new variable if the object is already exists with the same value.

Now, see another example.



Example -

```
x = 20
y = x
x += 1
If id(x) == id(y):
    print("x and y do not refer to  the same object")
```

Output:

```
x and y do not refer to the same object
```

The variables x and y are not referring the same object because x is incremented by one, x creates the new reference object and y still referring to 10.

Transforming the Garbage Collector

The Python Garbage collector has classified the objects using its generation. Python Garbage collector has the three-generation. When we define the new object in the program, its life cycle is handled by the garbage collector's first generation. If the object has use in a different program, it will be stimulated up to the next generation. Every generation has a threshold.

The garbage collector comes into action if the threshold of the number of allocations minus the number of de-allocation is exceeded.

We can modify the threshold value manually using the **GC** module. This module provides the **get_threshold()** method to check the threshold value of a different generation of the garbage collector. Let's understand the following example.

Example -

```
Import GC
print(GC.get_threshold())
```

Output:

```
(700, 10, 10)
```

In the above output, the threshold value 700 is for the first generation and other values for the second and third generation.

The threshold value for trigger the garbage collector can be modified using the **set_threshold()** method.

Example - 2

```
import gc
gc.set_threshold(800, 20, 20)
```

In the above example, the value of the threshold increased for all three generations. It will affect the frequency of running the garbage collector. Programmers don't need to worry about the garbage collector, but it plays essential role in optimizing the Python runtime for the target system.

Python garbage collector handles the low-level details for the developer.

Importance of Performing Manual Garbage Collection

As we have discussed earlier, the Python interpreter handles the reference to object used in the program. It automatically frees the memory when the reference count becomes zero. This is a classical approach for reference counting, if it fails to work when the program has referenced cycles. The reference cycle occurs when one or more objects are referenced to each other. Hence the reference count never becomes zero.

Let's understand the following example -

```
def cycle_create():
   list1 = [18, 29, 15]
   list1.append(list1)
   return list1

cycle_create()
[18, 29, 15, [...]]
```

We have created the reference cycle. The list1 object is referring the object list1 itself. When the function returns object list1, the memory for the object list1 is not freed up. So that reference counting is not suitable for solving the reference cycle. But, we can solve it by altering the garbage collector or performance of the garbage collector.

To accomplish that, we will use the **gc.collect()** function for the gc module.

```
import gc
n = gc.collect()
print("Number of object:", n)
```

The above code will give the number of collected and de-allocated objects.

We can perform the manual garbage collector using the two methods - time-based or event-based garbage collection.

The **gc.collect()** method is used to perform the time-based garbage collection. This method is called after a fixed time interval to perform time-based garbage collection.

In the even-based garbage collection, the **gc.collect()** function calls after an event occur. Let's understand the following example.

Example -

```
import sys, gc

def cycle_create():
    list1 = [18, 29, 15]
    list1.append(list1)

def main():
    print("Here we are creating garbage...")
    for i in range(10):
        cycle_create()

    print("Collecting the object...")
    num = gc.collect()
    print("Number of unreachable objects collected by GC:", num)
    print("Uncollectable garbage:", gc.garbage)

if __name__ == "__main__":
    main()
    sys.exit()
```

Output:

```
Here, we are creating garbage...
Collecting the object...
Number of unreachable objects collected by GC: 10
Uncollectable garbage: []
```

In the above code, we have created the list1 object referred by the list variable. The first element of the list object refers to itself. The list object's reference count is always greater than zero, even it is deleted or out of scope in the program.

C Python Memory Management

In this section, we will discuss the C Python memory architecture in detail.

As we discussed earlier, there is a layer of abstraction from the physical hardware to Python. Various application or Python access the virtual memory that is created by the operating system.

Python uses a portion of the memory for internal use and non-object memory. Another part of the memory is used for Python object such as int, dict, list, etc.

CPython contains the object allocator that allocates memory within the object area. The object allocator gets a call every time the new object needs space. The allocator primary designs for small amount of data because Python doesn't involve too much data at a time. It allocates the memory when it is absolutely required.

There are three main components of the CPython memory allocation strategy.

Arena - It is the largest chunks of memory and aligned on a page boundary in memory. The operating system uses the page boundary which is the edge of a fixed-length contiguous chuck of memory. Python assumes the system's page size is 256 kilobytes.

Pools - It is composed of a single size class. A pool of the same size manages a double-linked list. A pool must be - used, full, or empty. A **used** pool consists of memory blocks for data to be stored. A full **pool** has all the allocated and contain data. An empty pool doesn't have any data and can be assigned any size class for the block when needed.

Blocks - Pools contains a pointer to their "free" block of the memory. In the pool, there is a pointer, which indicates the free block of memory. The allocator doesn't touch these block until it's actually needed.

Common Ways to Reduce the Space Complexity

We can follow some best practices to reduce the space complexity. These techniques are supposed to save quite space and make the program efficient. Below are a few practices in Python for memory allocators.

- **Avoid List Slicing**

We define a list in Python; the memory allocator allocates the Heap's memory according to the list indexing, respectively. Suppose we need a sub-list to the given list, then we perform the list slicing. It is a straightforward way to get the sublist from the original list. Somehow, it is suitable for the small amount of data but not for the large data.

Hence, list slicing generates the copies of the object in the list. It just copies the reference to them. As a result, the Python memory allocator creates a copy of the object and allocates it. So we need to avoid the list slicing.

The best way to avoid the developer should try to use the separate variable to track indices instead of slicing a list.

- **Use List Indexing Carefully**

The developer should try to use the ***"for item in array"*** instead of **"for index in range(len(array))"** to save space and time. If our program doesn't need the indexing of the list element, then don't use it.

- **String Concatenation**

String concatenation is not suitable for saving space and time complexity. When possible, we should avoid using '+' for the string concatenation because strings are immutable. When we add the new string to the existing string, Python creates the new string and allocates it to a new address.

Each string needs a fixed size of memory based on the character and its length. When we change the string, it needs a different amount of memory and requires reallocating.

Let's run the following example.

```
a = Mango
print(a)
a = a + " Ice-cream"
print (a)
```

Output:

```
Mango
Mango Ice-cream
```

It will create variable **a** to refer to the string object, which is the string value information.

Then we add the new sting in it using the '+' operator. Python reallocates the new string in the memory based on its size and length. Suppose the original string's memory size is n byte, then the new string will be the m bytes.

Instead of using the string concatenation, we can use the" **.join(iterable_object)" or format or %.** This makes a huge impact on saving memory and time.

- **Use Iterators and Generators**

Iterators are very helpful for both time and memory when working on a large set of data. Working with the large dataset, we need data processing immediately and can wait for the program to process the entire data set first.

Generators are the special functions used to create the iterator function.

In the following example, we implement an iterator that calls the special generator function. The **yield** keyword returns the current value, moving to the next value only on the loop's next iteration.

Example -

```
def __iter__(self):
   ''' This function allows are set to be iterable. Element can be looped over using the for loop'''
   return self. _generator()

def _generator(self):
   """ This function is used to implement the iterable. It stores the data we are currently on and gi
   for i in self.items():
      yield i
```

- **Use the Built-in Library when Possible**

If we use methods that have already been predefined in the Python library, then import the corresponding library. It would save a lot of space and time. We can also create a module to define the function and import it to the current working program.

Conclusion

In this tutorial, we have discussed working of the memory internally in Python. We have learned how Python manages memory and also discussed default Python implementation. CPython is written in the C programming language. Python is a dynamic type of language that used the Heap data structure to store the memory.

← Prev

Next →

For Videos Join Our Youtube Channel: Join Now

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share