GC, GIL

Created	@October 2, 2023 12:02 PM
Owner	Din Lester
Tags	python
Status	Done

Sources:

- https://rushter.com/blog/python-garbage-collector/
- https://tenthousandmeters.com/blog/python-behind-the-scenes-13-the-gil-and-its-effects-on-python-multithreading/
- http://arctrix.com/nas/python/gc/
- https://realpython.com/python-gil/
- https://devguide.python.org/internals/garbage-collector/
- https://jenkov.com/tutorials/java-concurrency/concurrency-vs-parallelism.html

---

As we know everything in python is an object. Knowing when to allocate them is easy. Python does it when you need to define a new object. Unlike allocation, automatic deallocation is tricky. Python needs to know when your object is no longer needed. Removing objects prematurely will result in a program crash.

Garbage collections algorithms track which objects can be deallocated and pick an optimal time to deallocate them. Standard CPython's garbage collector has two components, the reference counting collector and the generational **garbage collector**, known as gc module.

> *Reference count incredibly efficient and straightforward but cannot detect reference cycles. So, here GC plays out.*

Reference count

Variables in python don’t actualy store a value, it rather stores a pointer. So, that every object in python has additional field with reference count.

```
a = [1, 2, 3]
b = a
```

This code creates two references to a single object.

Global variables usually live until the end of the Python process, so the reference count never drops to zero. This is achieved by handling those variables in a special dictionary structure, which you can see by calling `globals()`.

As for local variables (those declared in functions, objects, etc.), they exist until the Python executor leaves this code block or until the **del** function is called.

```
import sys

foo = []

# 2 references, 1 from the foo var and 1 from getrefcount
print(sys.getrefcount(foo))


def bar(a):
    # 4 references
    # from the foo var, function argument, getrefcount and Python's function stack
    print(sys.getrefcount(a))


bar(foo)
# 2 references, the function scope is destroyed
print(sys.getrefcount(foo))
```

> *In Jupiter all cells are using global scope, so to free memory use* ***del*** - interesting fact 🙂

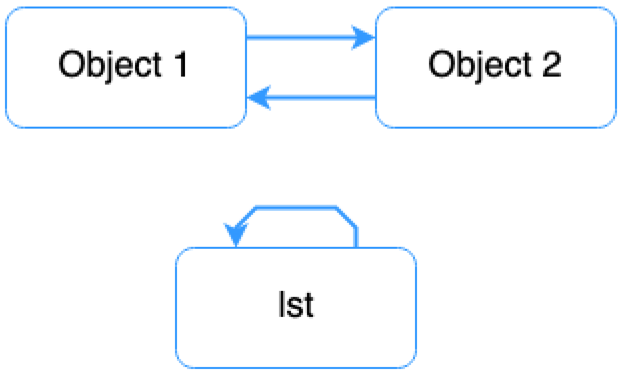
> *The reference counting algorithm has a lot of issues, such as circular references, thread locking, and memory and performance*

> *overhead. Reference counting is one of the reasons why Python can't get rid of the* ***GIL*****.** **- important**

The main advantage of such an approach is that the objects can be immediately and easily destroyed after they are no longer needed.

Generational garbage collector

As mentioned above, RC has problem with detecting circular references. So, let’s review the new mechanism for memory deallocation.



```
import gc

# We use ctypes module  to access our unreachable objects by memory address.
class PyObject(ctypes.Structure):
    _fields_ = [("refcnt", ctypes.c_long)]


gc.disable()  # Disable generational gc

lst = []
lst.append(lst)

# Store address of the list
lst_address = id(lst)

# Destroy the lst reference
del lst

object_1 = {}
object_2 = {}
object_1['obj2'] = object_2
object_2['obj1'] = object_1
```

```
obj_address = id(object_1)

# Destroy references
del object_1, object_2

# Uncomment if you want to manually run garbage collection process
# gc.collect()

# Check the reference count
print(PyObject.from_address(obj_address).refcnt)
print(PyObject.from_address(lst_address).refcnt)
```

Reference cycles can only occur in container objects (i.e., in objects that can contain other objects), such as lists, dictionaries, classes, tuples. The garbage collector algorithm does not track all immutable types except for a tuple. Tuples and dictionaries containing only immutable objects can also be untracked depending on certain conditions. Thus, the reference counting technique handles all non-circular references.

When does the generational GC trigger ?

GC runs periodically. It classifies container objects into three generations. Every new object starts in the first generation. If an object survives a garbage collection round, it moves to the older (higher) generation. Lower generations are collected more often than higher.

> *Each generation stores 2 values: count and threesholds. Every time you create a new object, CPython checks this values and if count > threeshold, it runs GC for specific generation.*

> ***IMPORTANT.*** *For 3 generation there are additional check for optimization purposes. [LINK]*

The standard threshold values are set to (700, 10, 10) respectively, but you can always check them using the `gc.get_threshold` function. You can also adjust them for your particular workload by using the `gc.set_threshold` function.

How to find reference cycles ? [GOING HARD]

Traditional garbage collection (eg. mark and sweep or stop and copy) usually works as follows:

1. Find the *root* objects of the system. These are things like the global environment (like the **main** module in Python) and objects on the stack.
2. Search from these objects and find all objects reachable from them. This objects are all "alive".
3. Free all other objects.

Unfortunately this approach cannot be used in the current version of Python. Because of the way extension modules work, Python can never fully determine the root set. If the root set cannot be determined accurately we risk freeing objects still referenced from somewhere. Even if extension modules were designed differently, the is no portable way of finding what objects are currently on the C stack.

> *Also, reference count is working fine, so we need somehow combine this 2 approches.*

How to we find reference cycles? First we add another field to container objects in addition to the two link pointers. We will call this field gc_refs. Here are the steps to find reference cycles:

1. For each container object, set **gc_refs** equal to the object's reference count.
2. For each container object, find which container objects it references and decrement the referenced container's **gc_refs** field.
3. All container objects that now have a **gc_refs** field greater than one are referenced from outside the set of container objects. We cannot free these objects so we move them to a different set.
4. Any objects referenced from the objects moved also cannot be freed. We move them and all the objects reachable from them too.
5. Objects left in our original set are referenced only by objects within that set (ie. they are inaccessible from Python and are garbage). We can now go about freeing these objects.

> *Basically, GC iterates over each container object and temporarily removes all references to all container objects it references. After full iteration, all objects which reference count lower than two are unreachable from Python's code and thus can be collected.*

> **Immortal objects.** *Python interpreter can now skip reference counting for some objects that live until the Python process terminates (e.g., for `None`, `True` and `False`). All objects that have the refcount set to `0xFFFFFFFF` (`4294967295`) are now considered immortal. Such a change allows some Python objects to be completely static in memory, which is a good optimization for Python applications that use multiprocessing and copy-on-write mechanism.*

Big and scary GIL

As we already know, python uses reference count to realize memory from unused objects. However, when we utilize multiple threads, there can be a race condition, where two threads may potentially modify the reference count simultaneously. So, it can cause for memory leak or incorrectly release the memory while a reference to that object **still exists**.

> *So, to solve this problem there is a special structure - GIL (simple mutex lock on thread).*

We have next code:

```
import time
from threading import Thread

COUNT = 50000000

def countdown(n):
    while n>0:
        n-=1

start = time.time()
countdown(COUNT)
end = time.time()

print(f"Time taken in seconds --- {end-start}")
```

On MacBook Pro 2019 it tooked 2.4 s.

Let’s modify this code to multithreading

```
import time
from threading import Thread

COUNT = 50000000

def countdown(n):
    while n>0:
        n -= 1

t1 = Thread(target=countdown, args=(COUNT//2,))
t2 = Thread(target=countdown, args=(COUNT//2,))

start = time.time()
t1.start()
t2.start()
t1.join()
t2.join()
end = time.time()

print(f"Time taken in seconds --- {end-start}")
```

IT TOOKED 2.7s.

What the heck ? Why don’t we just drop this GIL ?

As Guido Van Rossum said:

> *“I’d welcome a set of patches into Py3k only if the performance for a single-threaded program (and for a multi-threaded but I/O-bound program) does not decrease”*

A lot of extensions were being written for the existing C libraries whose features were needed in Python. To prevent inconsistent changes, these C extensions required a thread-safe memory management which the GIL provided. C libraries that were not thread-safe became easier to integrate. And these C extensions became one of the reasons why Python was readily adopted by different communities.

<u>So, main reasons are: single-threaded and I/O bound tasks are slower without GIL, many libraries would be broken without it.</u>