# Garbage collection in Python: things you need to know

Last updated on July 15, 2019, in Python

This article describes garbage collection (GC) in Python 3.6.

Usually, you do not need to worry about memory management. When objects are no longer needed, Python automatically reclaims memory from them. However, understanding how GC works can help you write better and faster Python programs.

## Memory management

Unlike many other languages, Python does not necessarily release the memory back to the Operating System. Instead, it has a dedicated object allocator for objects smaller than 512 bytes, which keeps some chunks of already allocated memory for further use in the future. The amount of memory that Python holds depends on the usage patterns. In some cases, all allocated memory could be released only when Python process terminates.

If a long-running Python process takes more memory over time, it does not necessarily mean that you have memory leaks. If you are interested in Python's memory model, you can read my article on memory management.

## Garbage collection algorithms

Standard CPython's garbage collector has two components, the reference counting collector and the generational **garbage collector**, known as gc module.

The **reference counting** algorithm is incredibly efficient and straightforward, but it cannot detect reference cycles. That is why Python has a supplemental algorithm called generational cyclic GC, that specifically deals with reference cycles.

The reference counting module is fundamental to Python and can't be disabled, whereas the cyclic GC is optional and can be invoked manually.

## Reference counting

Reference counting is a simple technique in which objects are deallocated when there is no reference to them in a program.

Every variable in Python is a reference (a pointer) to an object and not the actual value itself. For example, the assignment statement just adds a new reference to the right-hand side.

To keep track of references, every object (even integer) has an extra field called reference count that is increased or decreased when a pointer to the object is created or deleted. See Objects, Types and Reference Counts section, for a detailed explanation.

**Examples, where the reference count increases:**

- assignment operator
- argument passing
- appending an object to a list (object's reference count will be increased).

If reference counting field reaches zero, CPython automatically calls the object-specific deallocation function. If an object contains references to other objects, then their reference count is decremented too. Thus other objects may be deallocated in turn. For example, when a list is deleted the reference count for all its items is decreased.

Variables, which are declared outside of functions, classes, and blocks are called globals. Usually, such variables live until the end of the Python's process. Thus, the reference count of objects, which are referred by global variables, never drops to 0.

Variables, which are defined inside blocks (e.g., in a function or class) have a local scope (i.e., they are local to its block). If Python interpreter exits from the block, it destroys all references created inside the block.

You can always check the number of current references using `sys.getrefcount` function.

Here is a simple example:

```
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))
```

The main reason why CPython uses reference counting is historical. There are a lot of debates nowadays about the weaknesses of such technique. Some people claim that modern garbage collection algorithms can be more efficient without reference counting at all. The reference counting algorithm has a lot of issues, such as circular references, thread locking and memory and performance overhead.

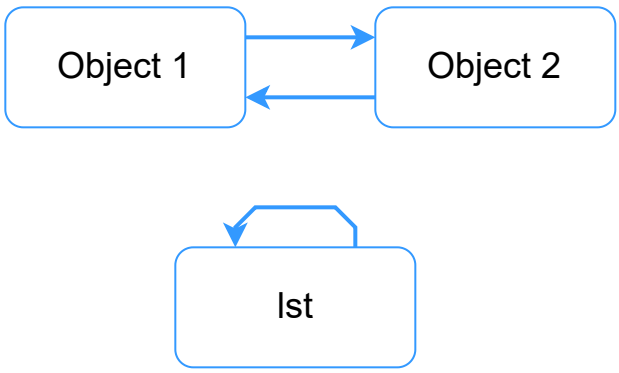
The main advantage of such approach is that the objects can be immediately destroyed after they are no longer needed.

## Generational garbage collector

Why do we need additional garbage collector when we have reference counting?

Unfortunately, classical reference counting has a fundamental problem — it cannot detect reference cycles. A reference cycle occurs when one or more objects are referencing each other.

Here are two examples:




As we can see, the 'lst' object is pointing to itself, moreover, `object 1` and `object 2` are pointing to each other. The reference count for such objects is always at least 1.

To get a better idea you can play with a simple Python example:

```
import gc

# We are using ctypes 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)
```

In the example above, the `del` statement removes the references to our objects (i.e., decreases reference count by 1). After Python executes the `del` statement, our objects are no longer accessible from Python code. However, such objects are still sitting in the memory, that's because they are still referencing each other and the reference count of each object is 1. You can visually explore such relations using objgraph module.

To resolve this issue, the additional cycle-detecting algorithm was introduced in Python 1.5. The gc module is responsible for this and exists only for dealing with such a problem.

Reference cycles can only occur in container objects (i.e., in objects that can contain other objects), such as lists, dictionaries, classes, tuples. Garbage collector 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

Unlike the reference counting, the cyclic GC does not work in real-time and runs periodically. To reduce the frequency of GC calls and pauses CPython uses various heuristics.

The GC 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. Because most of the newly created objects die young, it improves GC performance and reduces the GC pause time.

In order to decide when to run, each generation has an individual counter and threshold. The counter stores the number of object allocations minus deallocations since the last collection.

Every time you allocate a new container object, CPython checks whenever the counter of the first generation exceeds the threshold value. If so Python initiates the collection process.

If we have two or more generations that currently exceed the threshold, GC chooses the oldest one. That is because oldest generations are also collecting all previous (younger) generations. To reduce performance degradation for long-living objects the third generation has additional requirements in order to be chosen.

The standard threshold values are set to (700, 10, 10) respectively, but you can always check them using the `gc.get_threshold` function.

### How to find reference cycles

It is hard to explain the reference cycle detection algorithm in a few paragraphs. But basically, GC iterates over each container object and temporarily removes all references to 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.

To fully understand the cycle-finding algorithm I recommend you to read an original proposal from Neil Schemenauer and collect function from CPython's source code. Also, the Quora answers and The Garbage Collector blog post can be helpful.

Note that, the problem with finalizers, which was described in the original proposal, has been fixed since Python 3.4. You can read about it in the PEP 442.

## Performance tips

Cycles can easily happen in real life. Typically you encounter them in graphs, linked lists or in structures, in which you need to keep track of relations between objects. If your program has an intensive workload and requires low latency, you need to avoid reference cycles as possible.

To avoid circular references in your code, you can use weak references, that are implemented in the `weakref` module. Unlike the usual references, the `weakref.ref` doesn't increase the reference count and returns `None` if an object was destroyed.

In some cases, it is useful to disable GC and use it manually. The automatic collection can be disabled by calling `gc.disable()`. To manually run the collection process, you need to use `gc.collect()`.

## How to find and debug reference cycles

Debugging reference cycles can be very frustrating especially when you use a lot of third-party libraries.

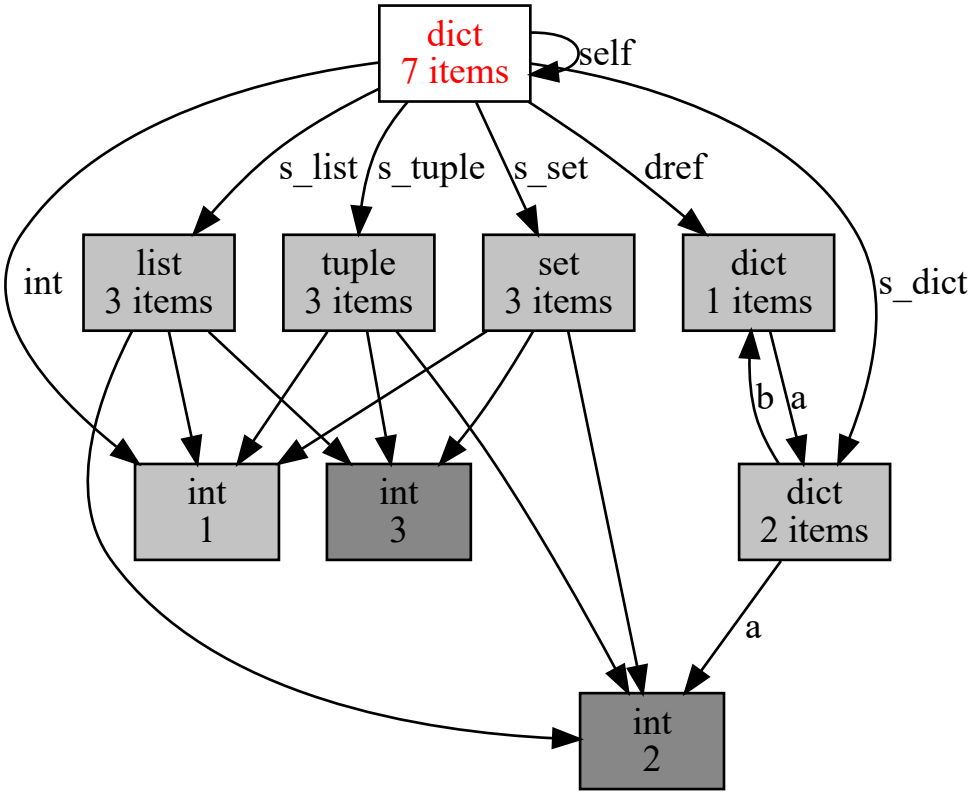
The standard gc module provides a lot of useful helpers that can help in debugging. If you set debugging flags to `DEBUG_SAVEALL`, all unreachable objects found will be appended to `gc.garbage` list.

```
import gc

gc.set_debug(gc.DEBUG_SAVEALL)

print(gc.get_count())
lst = []
lst.append(lst)
list_id = id(lst)
del lst
gc.collect()
for item in gc.garbage:
    print(item)
    assert list_id == id(item)
```

Once you have identified a problematic spot in your code you can visually explore object's relations using objgraph.




## Conclusion

Most of the garbage collection is done by reference counting algorithm, which we cannot tune at all. So, be aware of implementation specifics, but don't worry about potential GC problems prematurely.

Hopefully, you have learned something new. If you have any questions left, I will be glad to answer them in the comments below.

> Popular posts in Python category

Python, cpython internals, advanced python, memory

RSS

Want a monthly digest of these blog posts?

Email

Subscribe

No spam. No unnecessary emails.

Load disqus comments

Back to top