Python and Memory

PETER NORTON

Peter works on automating cloud environments. He loves using Python to solve problems. He has contributed to books on Linux and Python, helped with the New York Linux Users Group, and helped to organize past DevOpsDays NYC events. In addition to Python, Peter is slowly improving his knowledge of Rust, Clojure, and maybe other fun things. Even though he is a native New Yorker, he is currently living in and working from home in the northeast of Brazil. pcnorton@rbox.co.

I found myself thinking about an interesting problem I ran into a few years ago. I was wondering why an open source metrics collection system seemed to have a relatively low performance ceiling when relaying metrics.

After much troubleshooting, I found that the performance issue resided in attempts at splitting a list: when it had thousands of messages, it would pull off some messages from the front of a list, then split the list, and the way it was doing this was inefficient. In the end I found some improvement in using a deque, but the problem has left me with continuing questions about some of these oddities in how Python does its memory management.

Since then I have found myself with an imperfect and incomplete understanding of Python's memory management, and I thought it would be interesting to take a quick look at the standard library to see if it can help tell us what Python is doing—how we allocate and manage memory. Let's start with a very light-on-details version of how we got here.

Back in the Day...

In the days before Python 2.0, Python's memory management simply consisted of a method called *reference counting*, that is, most objects referenced from somewhere—within the global scope in a file, function, or object or class from a module you've loaded—will maintain a field that the interpreter will increase when there is a new reference to the object and will decrease when a reference is removed (e.g., a context manager going out of scope or a function finishing up). For many data types, that's pretty foolproof, and it's a very simple system for a language runtime to implement.

Circular Data Structures

The well-known weakness in this simple solution is that there are lots of situations where, deliberately or incidentally, we can create circular references—object A contains a reference to object B, and object B also contains a reference to object A. In case you're having trouble visualizing the situation, something like the following serves as a trivial example:

```
dict_A = {
    "a": "this is an A",
    "b": dict_B
}
dict_B = {
    "a": dict_A,
    "b": "this is a B"
}
```

When we create situations like this, it's called a *reference loop*, and since both objects will never have a reference count that goes to zero upon going out of scope, it is now memory that can't be collected automatically by a reference-counting garbage collector (GC).

As you can see in this example, you need a so-called container type to do this. Lists and dictionaries are primary examples of container types, which are so-called because they contain

references to other objects. Because they're more complex than simple non-container types like a bool, integer, float, or string, and because they are essentially a way to create memory pointers, by their nature they can break reference counting if not used with extreme care.

Python Needed More than Reference Counting

Python made it to version 2 with the reference counter as its only garbage collection mechanism, and by then its limits were well known. Starting in version 2.0, an additional garbage collection mechanism, called a *generational collector,* was added that could clean up where the reference counting couldn't. The principle is that it looks for objects that are allocated but possibly not reachable by the reference counter and makes sure that any objects that aren't in use are cleaned up. When they are found to be in use, it makes the sensible decision that something that's in use is likely to remain in use, and "promotes" that object to a more senior generation, which doesn't get checked as often.

Reference counting is very fast and easy to implement, while the generational collector is more complex: because it scans memory and needs the state of memory to not change while it's running, it has more of an impact on performance because every time it runs it must stop the main thread, grab the Global Interpreter Lock, and do its work.

The more objects that the generational garbage collection needs to manage, the longer it will take to scan memory. The idea is that most objects will be immediately handled by the reference counter, making the more expensive alternative the one that will be used less.

How Active Is the GC, Really?

You can get a precise idea of how much work the GC is doing by using the `gc` module.

From version 3.3 on, the Python's `gc` module has provided hooks that will invoke a callback to notify the program whenever a GC event occurs. With these hooks, you can gather the data you need to describe what the collector is doing and to help you infer why it's doing it. Most of the work that the GC is doing is usually invisible, but this module helps with the important feature of letting you see how fast the interpreter thinks it needs to clean up after itself, and how actively it is doing so.

```
import gc
print(gc.get_stats())
def print_hook(phase, info):
    print(f"The gc hook is in phase {phase}")
    print(f"And the gc hook provided this info: {info}")

gc.callbacks.append(print_hook)
foo = list(range(500))
del(foo)
print(gc.get_stats())
```

And at the end you should see a summary provided by `gc.get_stats` that roughly matches what you see as having been collected and shown by the `print_hook`.

Once you are able to know more about the garbage collection patterns that code is creating, you may find yourself wondering what kind of controls you have over how the garbage collection actually works—for example, can you schedule the garbage collection for times that you prefer? Or are there other ways to control the garbage collection behavior at all?

There are tunable knobs that allow you to manage your programs' garbage collection. For instance, you have the `gc.disable` and `gc.enable` to guarantee that your code runs uninterrupted for a span. Or if you know for sure that you won't need it, you can just disable automatic garbage collection at the start—for example, if you have a job that loads a lot of data, then outputs some data, and exits after a short time, who bothers with garbage collection at all? In some cases it can be a huge advantage to not clean up garbage before exiting.

Similarly, if you find you need to manually manage when collections are run, you can in fact make the garbage collector run a collection using `gc.collect()`, which will run either a full collection (check everything) or you can tell it to work on a particular generation by specifying 0, 1, or 2.

If it turns out that there are sections of your code that legitimately will never change, or if a lot of work is done, then a fork-exec is done: the Python `gc` module provides the `gc.freeze()` and `gc.unfreeze()` functions, whose main use is documented as being for forking a process and minimizing churn in the VM subsystem having to map child processes to new pages if the parent process decides to collect something that it didn't need to (after all, often it will be doing nothing but using `wait()` in a loop).

Thresholds

Thresholds are another interesting setting. If you think you want to collect more or less frequently but still have collection be automatic, you set thresholds, which represent how many times container objects are allocated until the GC kicks in and does a run over first or second generation objects to see what needs collection. The thresholds set how many objects there are in a particular generation before a garbage collection run is initiated. The default thresholds of 700, 10, 10 show the expectation that there will normally be a lot more small objects created that will be collected than there are that will survive, and that only a few will be tenured and survive to be shifted from the 0th generation to the 1st and 2nd. The 2nd generation is the highest and shouldn't have very many unreachable but live objects.

The idea with thresholds is that if you discover that you've created a lot of garbage that is getting tenured, and should remain live, you can adjust those thresholds to prevent the garbage collector from doing unnecessary work.

Tracemalloc—Seeing Which of Your Files Are Allocating Memory

On the other side of the coin, what's also easier to observe in Python 3.3+ is what's happening on the allocation side. It's conceptually clear that whenever you create an object, there will be memory allocated, but the layers of modules, objects, iterators, etc. can often make it hard to just take a glance at your code and have a good idea as to how much work is being put into creation and allocation of memory.

To do that there is an interesting module called `tracemalloc` [1], which can be used to show you where your allocations are happening. And while tracing, you can even pick up individual objects and see where their allocations took place, so you can figure out where in your code you may be exercising the allocator. If you're doing enough small allocations, this module can help you confirm whether memory is being used where you expect it. This seems like it would be most useful when understanding code from outside modules but could still be useful in code you've written for yourself.

The sample here is a variation on the standard library documentation and will show you how many allocations were made in each file that was recorded (though the limit of `[0:10]` when extracting from the traceback will only show us the top 10 in this case):

```
import tracemalloc
import requests

tracemalloc.start(25)
resp = requests.get('https://google.com/')
t = tracemalloc.take_snapshot()
tracemalloc.stop()
print("\n".join([str(s) for s in t.statistics('traceback')
[0:10]]))
```

This second example will do something very similar but shows you what the stack looked like when a particular object was allocated its memory:

```
import tracemalloc
import requests

tracemalloc.start(25)
resp = requests.get('https://google.com/')
objinfo = tracemalloc.get_object_traceback(resp)
tracemalloc.stop()
print("\n".join([str(l) for l in objinfo]))
```

In this case, the output should look something like this:

```
$ /usr/bin/Python3 gc-test-obj-traceback.py
gc-test-obj-traceback.py:5
/usr/lib/Python3/dist-packages/requests/api.py:75
/usr/lib/Python3/dist-packages/requests/api.py:60
/usr/lib/Python3/dist-packages/requests/sessions.py:533
/usr/lib/Python3/dist-packages/requests/sessions.py:668
/usr/lib/Python3/dist-packages/requests/sessions.py:668
/usr/lib/Python3/dist-packages/requests/sessions.py:247
/usr/lib/Python3/dist-packages/requests/sessions.py:646
/usr/lib/Python3/dist-packages/requests/adapters.py:533
/usr/lib/Python3/dist-packages/requests/adapters.py:265
```

And sure enough, in my Python installation adapters.py on line 265 is the beginning of the `build_response()` function in the requests module, and that is where the object was created. I was very pleasantly surprised to find this particular behavior. It seems to be a great tool to help with code spelunking to discover where an object actually came from, which can be very dependent on runtime conditions when you're creating objects in complicated situations.

The existence of the `tracemalloc` module is interesting, and it would be fun to explore more of its API and expected behaviors by pointing it at live code.

The More State-of-the-Art Allocators

However fun it is to be able to look under the hood, it wouldn't be fair to not mention a bit more about what the current state of the art is outside of Python.

Since the addition of the generational collector, a lot of research has been done in the garbage collection field. Most of the practical research that I'm familiar with has focused on the use case of Java and the JVM. So I'll give an overview of my incomplete understanding of the progress of Java's last few generations of garbage collection from the point of view of someone who's had to struggle with it.

Most of us have probably used the Concurrent Mark+Sweep collector, which was the primary Java garbage collector for a long time. However, it didn't age well—as applications and platforms switched from 32-bit pointers to 64-bit pointers, developers of applications that were memory intensive found it was fairly common to experience multisecond stop-the-world GC pauses at the most inconvenient times. In addition, the more memory that was in use, the worse the impact of the pauses and the longer the pauses. And, as the available memory in 64-bit systems has grown, the CMS collector has shown its age and was not able to keep up.

In the past decade a new garbage collector known as the G1 GC matured in the Sun/Oracle Java 8 with the goal of reducing pause times and enabling the JVM to be able to handle larger memory heaps, and it no longer became a very tricky proposition to request and use a heap of greater than eight GB.

Currently, JVM-hosted applications are growing to use more resources and more memory in specific. In the OpenJDK era there are two brand new garbage collection systems that have miraculously grown to handle more memory with smaller, less-noticeable pauses. One is called Shenandoah [2] and the other is called `zgc` [3]. Both are incredibly ambitious and featureful. Users of Java will be very happy with this change.

In short, the state-of-the-art of memory management and garbage collection has continued to advance. From my perspective, the driver in garbage collection research has been Java, which has made steady gains. Python is already used in projects that have large memory footprints, but anecdotally I haven't heard that it's great, and I wonder whether advances in the language, like the multiple interpreters in PEP 554 [4] planned for Python 3.9, are likely to have an impact in memory usage by making high-performance multithreaded Python applications start to seem more tractable.

In any case, please go and try out these modules on your own code, and enjoy!

References

[1] https://docs.Python.org/3/library/tracemalloc.html.

[2] https://wiki.openjdk.java.net/display/shenandoah/Main.

[3] https://wiki.openjdk.java.net/display/zgc/Main.

[4] https://www.Python.org/dev/peps/pep-0554/.