Python Garbage Collection: Key Concepts and Mechanisms

Learn how Python automatically manages memory with reference counting and generational garbage collection, and understand how to manually control garbage collection using the gc module.

Sep 14, 2024 · 9 min read

Samuel Shaibu
Data Scientist | Microsoft Certified Data Analyst Associate | Technical Writer

TOPICS

Python

Data Science

Memory management is an important yet often overlooked aspect of programming. If not handled properly, it can lead to slow applications, unexpected crashes, or even memory leaks. Fortunately, Python takes care of this through a process known as garbage collection, which is a built-in system that automatically manages memory.

But how does Python's garbage collection work, and why should you care? Understanding this process is key to writing efficient, bug-free code that performs well, even as your projects grow in complexity.

By the end of this article, you will know the basics of garbage collection in Python and why it matters, understand how Python uses reference counting and generational garbage collection to manage memory and learn some tips for avoiding memory leaks. I encourage you at the end to also enroll in our **Writing Efficient Python Code** course, which will teach you how to allocate resources in your environment and consider overhead, larger topics of which garbage collection is just a part.

What is Garbage Collection in Python?

Garbage collection is Python's way of automatically managing memory, ensuring that your applications runs smoothly by freeing up memory that is no longer in use. In simpler terms, it is like having an invisible janitor who cleans up after your code, disposing of objects that are no longer needed.

“Python collecting garbage.” Image by Dall-E 2

Why do we need garbage collection in Python?

In any application, objects are created to store data, perform calculations, or manage tasks. However, once these objects have served their purpose, they continue to occupy memory space until they are explicitly removed. If these unused objects accumulate, they can cause your application to use more memory than necessary, leading to slower performance and potential crashes.

Garbage collection prevents such scenarios. It automatically detects when an object is no longer needed (no part of your code references it), and then safely removes it from memory. This process helps to:

- **Prevent Memory Leaks:** By automatically cleaning up unused objects, garbage collection reduces the risk of memory leaks, where memory that is no longer needed is not released.
- **Optimize Performance:** By freeing up memory, garbage collection helps maintain the performance of your application, especially in long-running programs or those that handle large amount of data.
- **Simplify Development:** Since Python handles memory management automatically, developers can focus on writing code rather than managing memory.

How Python Implements Garbage Collection Automatically

Python’s garbage collection is a complex system designed to manage memory automatically, allowing developers to focus on writing code rather than worrying about memory management. Python primarily uses two mechanisms to implement garbage collection: **reference counting** and **generational garbage collection**. These mechanisms work together to ensure that memory is managed efficiently, minimizing the chances of memory leaks and optimizing the performance of your application.

Before looking into these mechanisms, it is important to note that this garbage collection process is most relevant to CPython, the most used implementation of Python. While other implementations like PyPy or Jython might handle garbage collection differently, CPython relies heavily on these two methods to keep your applications running smoothly.

Reference counting

Reference counting is the foundational method Python uses to manage memory. At its core, reference counting involves keeping track of the number of references (or pointers) to an object in memory. Each time a new reference to an object is created, Python increases that object’s reference count. Conversely, when a reference is removed or goes out of scope, Python decreases the reference count. Here is how it works:

- **Tracking References:** Every object in Python has a reference count, which is updated whenever the object is referenced or dereferenced. For instance, assigning an object

to a variable or passing it to a function increases its reference count, while deleting the variable decreases the count.

- **Deallocating Memory:** When an object's reference count drops to zero, meaning no part of your code is using the object, Python automatically deallocates the memory it occupies.

Despite its efficiency, reference counting has limitations. The most significant limitation is its inability to handle **cyclic references**, which occur when two or more objects reference each other, forming a cycle. In these cases, reference counts never reach zero, preventing memory from being reclaimed. This is where **generational garbage collection** comes in.

Generational garbage collection

To overcome the limitations of reference counting, Python also employs generational garbage collection. This advanced method is designed to handle cyclic references and improve memory management efficiency. The core idea behind generational garbage is based on the observation that most objects are either short-lived (temporary) or long-lived (persistent). By categorizing objects based on their age, Python optimizes the garbage collection process. Here is how generational garbage collection works:

- **Generations:** Python's garbage collector organizes objects into three generations: Generation 0 (youngest), Generation 1 (middle-aged), and Generation 2 (oldest). New objects are placed in Generation 0, and if they survive garbage collection, they move to the next generation.
- **Prioritizing Younger Objects:** The garbage collector runs more frequently on younger objects (Generation 0) because these objects are more likely to become unused quickly. As objects age and are moved to higher generations, they are collected less frequently. This approach reduces the overhead of garbage collection by focusing more on objects that are likely to be discarded soon.
- **Handling Cyclic References:** Generational garbage collection is particularly effective at identifying and collecting objects involved in cyclic references. During the collection process, Python's garbage collector can detect these cycles and reclaim the memory, preventing memory leaks caused by cyclic references.

How You Can Trigger Python Garbage Collection Manually

While Python's garbage collection system is designed to handle memory management automatically, there are situations where manually managing garbage collection can be helpful. Let's take a look.

Initiating garbage collection in Python

In most cases, Python's garbage collector runs automatically, cleaning up unused objects without any intervention. However, there are situations where you might want to manually trigger garbage collection to free up memory. This is particularly useful in long-running applications or memory-intensive processes where memory usage needs to be managed more closely.

To manually trigger garbage collection in Python, you can use the `gc.collect()` function. This function forces the garbage collector to run immediately, reclaiming memory that is no longer in use. Here is how it works:

```
import gc
# Trigger garbage collection manually
gc.collect()
```


When you call `gc.collect()`, Python's garbage collector will perform a full collection, examining all objects in memory and deallocating those that are no longer referenced. This can be particularly useful in scenarios such as:

- **Memory-Intensive Applications:** In applications that process large amounts of data or create many objects, manually triggering garbage collection can help free up

memory at critical points, reducing the risk of memory bloat.

- **Long-Running Processes:** In services or applications that run for extended periods, like servers or background tasks, manually managing garbage collection can help maintain a steady memory footprint, ensuring that the application remains responsive and efficient over time.

However, it is important to use `gc.collect()` judiciously. Frequent manual garbage collection can introduce unnecessary overhead, as the process can be resource-intensive. It is typically best used in specific scenarios where you have identified potential memory issues or need to free up memory at a precise moment.

Disabling garbage collection

In some cases, you might find it beneficial to disable Python's automatic garbage collection temporarily. This can be useful in scenarios where the overhead of garbage collection could negatively impact performance, such as in real-time applications, performance-critical sections of code, or during operations where you want to minimize interruptions.

To disable the garbage collector, you can use the `gc.disable()` function:

```
import gc
# Disable automatic garbage collection
gc.disable()
```


When garbage collection is disabled, Python will stop automatically collecting and deallocating unused objects. This can lead to more predictable performance in certain situations, as it prevents the garbage collector from running unexpectedly during critical operations.

However, disabling garbage collection comes with its risks:

- **Memory Leaks:** Without garbage collection, unused objects remain in memory until the process ends, which can lead to memory leaks. This is particularly problematic in long-running applications, where memory usage can grow unchecked.
- **Cyclic References:** Since cyclic references are not automatically resolved without garbage collection, disabling the garbage collector can exacerbate memory issues if cycles are present in your code.

For these reasons, it is essential to re-enable garbage collection after the performance-critical section of your code has executed. You can re-enable the garbage collector using `gc.enable()`:

```
# Re-enable automatic garbage collection
gc.enable()
```


Let's take a look at some best practices for manual garbage collection:

- **Use Sparingly:** Manual garbage collection should be applied only when necessary. Overuse can result in performance degradation.
- **Combine with Profiling:** Before manually triggering or disabling garbage collection, consider profiling your application to understand its memory usage patterns. This can help you determine whether manual intervention is needed and where it will have the most impact.
- **Re-enable When Needed:** If you disable garbage collection, remember to re-enable it as soon as the critical section of code is complete. This ensures that memory is managed efficiently over the long term.

By understanding how and when to use manual garbage collection, you can gain finer control over your application's memory management. This not only helps optimize

performance but also prevents memory-related issues that could affect the stability of your application.

In the next section, we will explore common garbage collection issues in Python and provide practical tips for debugging and resolving them.

Practical Considerations for Python Developers

Managing memory efficiently is critical for writing performant Python applications. While Python's garbage collection system handles most memory management tasks automatically, there are practical steps developers can take to avoid common pitfalls and optimize memory usage. In this section, we will explore strategies for avoiding memory leaks, dealing with cyclic references, and managing large numbers of objects in Python applications.

Avoiding memory leaks

Memory leaks occur when objects that are no longer needed are not properly deallocated, causing the application to consume more memory over time. This can lead to performance degradation or maybe even cause the application to crash. To monitor objects in memory, you can use Python's `gc` module. By looking at the number of objects in memory, you can see unexpected increases, which might indicate a memory leak:

```
import gc

# Get a list of all objects tracked by the garbage collector
all_objects = gc.get_objects()

print(f"Number of tracked objects: {len(all_objects)}")
```


If you suspect that an object is not being garbage collected, the `gc.get_referrers()` function helps identify what is keeping the object in memory. This function returns a list of objects that reference the given object, allowing you to determine whether these references are necessary or can be removed:

```
some_object = ...
referrers = gc.get_referrers(some_object)

print(f"Object is being referenced by: {referrers}")
```


Another approach to avoid memory leaks is by using weak references, particularly in caching scenarios. The `weakref` module lets you create references that do not increase the reference count of the object:

```
import weakref

class MyClass:
    pass
obj = MyClass()
weak_obj = weakref.ref(obj)

print(weak_obj())  # Access the object

del obj  # Delete the strong reference
print(weak_obj())  # None, as the object is now collected
```


Cyclic references occur when two or more objects reference each other, creating a loop that prevents their reference counts from ever reaching zero. This can lead to memory leaks if the garbage collector is unable to detect and collect these objects.

Python's generational garbage collector is designed to handle cyclic references, but it's still best not to create unnecessary cycles in the first place. Using the `gc` module, you can detect objects that are part of reference cycles by triggering a collection and checking the `gc.garbage` list, which contains objects that are part of cycles and could not be collected:

```
import gc

# Trigger garbage collection and get the number of uncollectable objects
gc.collect()

uncollectable_objects = gc.garbage
print(f"Number of uncollectable objects: {len(uncollectable_objects)}")
```


To prevent cyclic references, consider breaking the cycle by explicitly removing references when they are no longer needed. You can do this by setting the reference to `None` or by using weak references, which are particularly good for avoiding strong reference cycles. Also, simplifying the design of your data structures and object relationships can decrease the chances of creating cycles.

Managing a large number of objects

Creating and destroying large numbers of objects in a short period can strain Python's garbage collector. Think about applications that handle large datasets, process real-time data, or perform complex situations.

One idea is to batch object creation and deletion. Instead of creating and destroying objects one at a time, batching these operations can reduce the frequency of garbage collection and allow the garbage collector to work more efficiently. For example:

```
objects = []

for i in range(1000):
    obj = SomeClass()
    objects.append(obj)

# Process all objects at once, then delete them
del objects[:]
gc.collect()  # Optionally trigger a manual garbage collection
```


Another consideration is optimizing object lifetimes by ensuring that objects are either short-lived or long-lived. Short-lived objects are quickly collected, and long-lived objects are moved to higher generations, where they are collected less frequently.

For scenarios where objects are frequently created and destroyed, an object pool can be a useful technique. An object pool reuses a fixed number of objects, reducing the strain on the garbage collector and improving performance in memory-constrained environments. Here is an example of an object pool implementation:

```
class ObjectPool:
    def __init__(self, size):
        self.pool = [SomeClass() for _ in range(size)]
    def get(self):
        return self.pool.pop()
    def release(self, obj):
        self.pool.append(obj)
```


Object pools are particularly beneficial in real-time applications or games, where performance is critical, and the overhead of frequent object creation and destruction can be significant.

Conclusion

In this article, we have explored Python's garbage collection system, covering everything from how it automatically manages memory to manual interventions that you can can take to make the performance better. The combination of reference counting and generational garbage collection enables Python to manage memory effectively, though there are times when manual intervention is beneficial. By following the best practices outlined in this guide, you can avoid memory leaks, enhance performance, and take greater control over how your applications handle memory.

If you are looking to explore more advanced topics in memory management and overall code optimization, there are several great DataCamp resources available. You can read our **How to Write Memory-Efficient Classes in Python** and **Memory Profiling in Python** tutorials, both of which are helpful for pinpointing performance bottlenecks. Moreover, learning how to write more **efficient Python code** is always helpful for any career.

AUTHOR

Samuel Shaibu

Experienced data professional and writer who is passionate about empowering aspiring experts in the data space.

Python Garbage Collection FAQs

How can I monitor memory usage in Python applications?

By using tools like `memory_profiler`, `tracemalloc`, and `objgraph`. These tools help track memory consumption, identify memory leaks, and optimize memory usage.

How can I manually trigger garbage collection in Python?

Can Python's garbage collection be disabled?

TOPICS

Python Data Science

Learn Python with DataCamp

COURSE

Python Toolbox

4 hr 280K

Continue to build your modern Data Science skills by learning about iterators and list comprehensions.