## **Memory Management In Python**

## **The Default Python Implementation**

## The default Python implementation, CPython, is actually written in the C programming language.

## When I first heard this, it blew my mind. A language that’s written in another language?! Well, not really, but sort of.

## The Python language is defined in a [reference manual](https://docs.python.org/3/reference/index.html) written in English. However, that manual isn’t all that useful by itself. You still need something to interpret written code based on the rules in the manual.

## You also need something to actually execute interpreted code on a computer. The default Python implementation fulfills both of those requirements. It converts your Python code into instructions that it then runs on a virtual machine.

## Note: Virtual machines are like physical computers, but they are implemented in software. They typically process basic instructions similar to [Assembly instructions](https://en.wikipedia.org/wiki/Assembly_language).

## Python is an interpreted programming language. Your Python code actually gets compiled down to more computer-readable instructions called [bytecode](https://docs.python.org/3/glossary.html#term-bytecode). These instructions get interpreted by a virtual machine when you run your code.

## Have you ever seen a .pyc file or a \_\_pycache\_\_ folder? That’s the bytecode that gets interpreted by the virtual machine.

## It’s important to note that there are implementations other than CPython. [IronPython](http://ironpython.net/) compiles down to run on Microsoft’s Common Language Runtime. [Jython](http://www.jython.org/) compiles down to Java bytecode to run on the Java Virtual Machine. Then there’s [PyPy](https://pypy.org/), but that deserves its own entire article, so I’ll just mention it in passing.

## For the purposes of this article, I’ll focus on the memory management done by the default implementation of Python, CPython.

## Disclaimer: While a lot of this information will carry through to new versions of Python, things may change in the future. Note that the referenced version for this article is the current latest version of Python, [3.7](https://realpython.com/python37-new-features/).

## Okay, so CPython is written in C, and it interprets Python bytecode. What does this have to do with memory management? Well, the memory management algorithms and structures exist in the CPython code, in C. To understand the memory management of Python, you have to get a basic understanding of CPython itself.

## CPython is written in C, which does not natively support object-oriented programming. Because of that, there are quite a bit of interesting designs in the CPython code.

## You may have heard that everything in Python is an object, even types such as int and str. Well, it’s true on an implementation level in CPython. There is a struct called a PyObject, which every other object in CPython uses.

## Note: A struct, or structure, in C is a custom data type that groups together different data types. To compare to object-oriented languages, it’s like a class with attributes and no methods.

## The PyObject, the grand-daddy of all objects in Python, contains only two things:

## ob\_refcnt: reference count

## ob\_type: pointer to another type

## The reference count is used for garbage collection. Then you have a pointer to the actual object type. That object type is just another struct that describes a Python object (such as a dict or int).

## Each object has its own object-specific memory allocator that knows how to get the memory to store that object. Each object also has an object-specific memory deallocator that “frees” the memory once it’s no longer needed.

## However, there’s an important factor in all this talk about allocating and freeing memory. Memory is a shared resource on the computer, and bad things can happen if two different processes try to write to the same location at the same time.

## **The Global Interpreter Lock (GIL)**

The GIL is a solution to the common problem of dealing with shared resources, like memory in a computer. When two threads try to modify the same resource at the same time, they can step on each other’s toes. The end result can be a garbled mess where neither of the threads ends up with what they wanted.

Consider the book analogy again. Suppose that two authors stubbornly decide that it’s their turn to write. Not only that, but they both need to write on the same page of the book at the same time.

They each ignore the other’s attempt to craft a story and begin writing on the page. The end result is two stories on top of each other, which makes the whole page completely unreadable.

One solution to this problem is a single, global lock on the interpreter when a thread is interacting with the shared resource (the page in the book). In other words, only one author can write at a time.

Python’s GIL accomplishes this by locking the entire interpreter, meaning that it’s not possible for another thread to step on the current one. When CPython handles memory, it uses the GIL to ensure that it does so safely.

## **Garbage Collection**

Let’s revisit the book analogy and assume that some of the stories in the book are getting very old. No one is reading or referencing those stories anymore. If no one is reading something or referencing it in their own work, you could get rid of it to make room for new writing.

That old, unreferenced writing could be compared to an object in Python whose reference count has dropped to 0. Remember that every object in Python has a reference count and a pointer to a type.

The reference count gets increased for a few different reasons. For example, the reference count will increase if you assign it to another variable:

numbers = [1, 2, 3]

# Reference count = 1

more\_numbers = numbers

# Reference count = 2

It will also increase if you pass the object as an argument:

total = sum(numbers)

As a final example, the reference count will increase if you include the object in a list:

matrix = [numbers, numbers, numbers]

Python allows you to inspect the current reference count of an object with the sys module. You can use sys.getrefcount(numbers), but keep in mind that passing in the object to getrefcount() increases the reference count by 1.

In any case, if the object is still required to hang around in your code, its reference count is greater than 0. Once it drops to 0, the object has a specific deallocation function that is called which “frees” the memory so that other objects can use it.

But what does it mean to “free” the memory, and how do other objects use it? Let’s jump right into CPython’s memory management.