Python Memory Management 101

Deeping in Garbage collector

José Manuel Ortega
@jmortegac
About me

- @jmortegac
- http://jmortega.github.io
- https://www.linkedin.com/in/jmortega1/
Agenda

- Introduction to memory management
- Garbage collector and reference counting with python
- Review the gc module for configuring the python garbage collector
- Best practices for memory management
Introduction to memory management

- Memory management is the process of efficiently allocating, de-allocating, and coordinating memory so that all the different processes run smoothly and can optimally access different system resources.
Python Objects in Memory

- **Heap**
  - Free Memory (Dynamic)

- **Stack**
  - Function Calls and Local Variable
  - Global Variable

- **Static/Global**
  - Instructions

- **Code**
Python Memory Manager

- Garbage Collector
- Private Heap Space (Python objects and data structures)
- Memory Manager
- Python Program
- Interpreter
Heap allocation
Heap allocation

def main():
    x=300
    print(id(x))
    w=fun(x)
    print(id(w))

def sqr(x):
    print (id(x))
    z=x*x
    print(id(z))
    return z

if __name__ == '__main__':
    main()
Python Objects in Memory

- Each variable in Python acts as an object
- Python is a *dynamically typed language* which means that we do not need to declare types for variables.
Python 3.8.1 (default, Feb 2 2020, 08:37:37)
>>> x=5
>>> print(x)
5
>>> del(x)
>>> print(x)
Traceback (most recent call last):
  File "<stdin>"", line 1, in <module>
NameError: name 'x' is not defined

Python 3.8.1 (default, Feb 2 2020, 08:37:37)

```python
x=50
y=x
x+=1
print("x,y:",x,y)
x,y: 51 50
```
Python Objects in Memory

```python
Python 3.8.1 (default, Feb 2 2020, 08:37:37)
>>> x = [1, 2, 3]
>>> y = x
>>> x[2] = 4
>>> print("x,y:", x, y)
```
x,y: [1, 2, 4] [1, 2, 4]
```python
Python 3.8.1 (default, Feb 2 2020, 08:37:37)
x = ['foo', [1, 2, 3], 10.4]
y = x
print(x, y)
['foo', [1, 2, 3], 10.4] ['foo', [1, 2, 3], 10.4]
y[1][0] = 4
print(x, y)
['foo', [4, 2, 3], 10.4] ['foo', [4, 2, 3], 10.4]
x[1][0] = 5
print(x, y)
['foo', [5, 2, 3], 10.4] ['foo', [5, 2, 3], 10.4]
```
id() method

```
Python 3.8.1 (default, Feb 2 2020, 08:37:37)
>>> list = [1,2,3]
>>> id(list)
140352728911232
>>> list2 = [1,2,3]
>>> id(list2)
140352725718848
>>> list2 = list
>>> id(list2)
140352728911232
>>> id(list)
140352728911232
```
id() method

Python 3.8.1 (default, Feb 2 2020, 08:37:37)
	x=10
	y=10
>>> print(id(x))
139786009870880
>>> print(id(y))
139786009870880
>>> print(x is y)
True
>>> print(y is x)
True
>>> z=x
>>> print(id(z))
139786009870880

id() method

```python
ten
z = w

print(id(w))
139785921940720

print(id(z))
139785921940720

print(x is z)
False

print(z is x)
False```
is Operator

```
x = 1234
y = x
id(x)
id(y)
140395389327632
id(y)
140395389327632
x is y
True
y is x
True
```
Reference counting

- Python manages objects by using **reference counting**
- Reference counting works by counting the number of times an object is referenced by other objects in the application.
- When references to an object are removed, the reference count for an object is decremented.
Reference counting

- A reference is a container object pointing at another object.

- Reference counting is a simple technique in which objects are allocated when there is reference to them in a program.
Reference counting

- when reference count increases?
  - x=1
  - def(x):
  - list.append(x)
Reference counting

```python
Python 3.8.1 (default, Feb 2 2020, 08:37:37)
>>> import sys
>>> list = [1,2,3,4]
>>> sys.getrefcount(list)
2
>>> list2 = list
>>> sys.getrefcount(list)
3
```
Reference counting

```python
Python 3.8.1 (default, Feb 2 2020, 08:37:37)
>>> l1 = [1, 2, 3, 4]
>>> l2 = l1
>>> import sys
>>> sys.getrefcount(l1)
3
>>> l2 = None
>>> sys.getrefcount(l1)
2
>>> sys.getrefcount(l1) = None
>>> l1 = None
>>> l1 = None
>>> sys.getrefcount(l1)
4268
>>> l1 = None
```
import sys
def print_value(value):
...    print("Value is ",value)
...    print("RefCount is ",sys.getrefcount(value))

value=7
sys.getrefcount(value)
20
print_value(value)
Value is 7
RefCount is 22
sys.getrefcount(value)
20
Reference counting

- Easy to implement
- Objects are immediately deleted when reference counter is 0

✗ Not thread-safe
✗ Doesn’t detect cyclic references
✗ space overhead - reference count is stored for every object
Garbage collector (GC) module

```python
Python 3.8.1 (default, Feb 2 2020, 08:37:37)
$ list = [1,2,3,4]
$ list2 = [list,[5,6,7,8]]
$ dict = {'key':list2,'key2':'value'}
$ import gc
$ gc.get_referents(dict)
[[[1, 2, 3, 4], [5, 6, 7, 8]], 'value']
```
Python Garbage collector

- Reference Counting + Generational GC
- RefCount reaches zero, immediate deletion
- Deleted objects with cyclic references are deleted with Tracing GC
Garbage collector (GC) reference cycle
>>> def ref_cycle():
...     list = [1, 2, 3, 4]
...     list.append(list)
...     return list
...     return list
import gc
for i in range(8):
    ref_cycle()

n = gc.collect()

print("Number of unreachable objects collected by GC:", n)
print("Uncollectable garbage:", gc.garbage)
print("Number of unreachable objects collected by GC:", gc.collect())
Creating garbage...
Collecting...
Number of unreachable objects collected by GC: 8
Uncollectable garbage: []
Number of unreachable objects collected by GC: 0
import objgraph
x = "hello"
y = [x, [x], list(x), dict(x=x)]
objgraph.show_refs([y], filename='sample-graph.png')
Best practices for memory management

- Using `gc.collect()` carefully

```python
print("Collecting...")
n = gc.collect()
print("Number of unreachable objects collected:", n)
print("Uncollectable garbage:", gc.garbage)
```
Garbage collector (GC) methods

**gc — Garbage Collector interface**

This module provides an interface to the optional garbage collector. It provides the ability to disable the collector, tune the collection frequency, and set debugging options. It also provides access to unreachable objects that the collector found but cannot free. Since the collector supplements the reference counting already used in Python, you can disable the collector if you are sure your program does not create reference cycles. Automatic collection can be disabled by calling `gc.disable()`. To debug a leaking program call `gc.set_debug(gc.DEBUG_LEAK)`. Notice that this includes `gc.DEBUG_SAVEALL`, causing garbage-collected objects to be saved in `gc.garbage` for inspection.

The `gc` module provides the following functions:

- **gc.enable()**
  Enable automatic garbage collection.

- **gc.disable()**
  Disable automatic garbage collection.

- **gc.isenabled()**
  Return `True` if automatic collection is enabled.

- **gc.collect(generation=2)**
  With no arguments, run a full collection. The optional argument `generation` may be an integer specifying which generation to collect (from 0 to 2). A `ValueError` is raised if the generation number is invalid. The number of unreachable objects found is returned.
Best practices for memory management

- Using **with context manager** for working with files

```python
with open('data.txt', 'r') as file:
    data = ','.join(line.strip() for line in file)
```
Best practices for memory management

- Avoid **List Slicing with [:]**

```python
list= [1,2,3,4]
list[1:3]
list[slice(1,3)]
```
Best practices for memory management

- String Concatenation

```python
string = "hello"
string += "world"

wordList = ("hello", "world")
string = " ".join(wordList)
```
Best practices for memory management

- Use Iterators and Generators

```python
def __iter__(self):
    """This function allows are set to be iterable. Elements can be looped over using 'for item in set'""
    return self._generator()

def _generator(self):
    """This function is a helper for iterable. It stores the data we are currently on and gives the next item at each iteration of the loop.""
    for item in self.items():
        yield item
```
References

- https://realpython.com/python-memory-management
- https://rushter.com/blog/python-garbage-collector
- https://pythonchb.github.io/PythonTopics/weak_references.html
https://www.youtube.com/c/JoseManuelOrtegaDev