Python Garbage Collection

Summary: in this tutorial, you'll learn how Python garbage collection works and how to interact with the garbage collector programmatically.

Introduction to Python garbage collection

In C/C++, you're fully responsible for managing the memory of the program. However, in Python, you don't have to manage the memory yourself because Python does it for you automatically.

In the references tutorial (https://www.pythontutorial.net/advanced-python/python-references/) , you've learned that Python Memory Manager keeps track of references of objects. The Memory Manager destroys the object and reclaims the memory once the reference count of that object reaches zero.

However, reference counting doesn't work properly all the time. For example, when you have an object that references itself or two objects reference each other. This creates something called circular references.

When the Python Memory Manager cannot remove objects with circular references, it causes a memory leak.

This is why the garbage collector comes into play to fix the circular references.

Python allows you to interact with the garbage collector via the built-in `gc` module.

Interacting with Python garbage collector

In this example, we'll first create a circular reference between two instances of class A and class B. Then, we use the garbage collector to destroy the objects in the circular reference.

First, import the `gc` and `ctypes` modules and defines two functions for counting references and check if an object exists in the memory:

```
import gc
import ctypes


def ref_count(address):
    return ctypes.c_long.from_address(address).value


def object_exists(object_id):
    for object in gc.get_objects():
        if id(object) == object_id:
            return True

    return False
```

In this code, the `ref_count()` returns the reference count of an object specified by its memory address. And the `object_exists()` function returns `True` if an object exists in the memory.

Second, create two classes (https://www.pythontutorial.net/python-oop/python-class/) `A` and `B` that have a reference each other:

```
class A:
    def __init__(self):
        self.b = B(self)
```

```
        print(f'A: {hex(id(self))}, B: {hex(id(self.b))}')


class B:
    def __init__(self, a):
        self.a = a
        print(f'B: {hex(id(self))}, A: {hex(id(self.a))}')
```

Third, disable the garbage collector by calling the `disable()` function:

```
gc.disable()
```

Fourth, create a new instance of class A that also automatically creates a new instance of B:

```
a = A()
```

Output:

```
B: 0x20fccd148e0, A: 0x20fccd75c40
A: 0x20fccd75c40, B: 0x20fccd148e0
```

Fifth, define two variables to hold the memory addresses of the instances of A and B. These variables keep track of the memory addresses of instances of A and B when the variable `a` references another object.

```
a_id = id(a)
b_id = id(a.b)
```

Sixth, show the reference counts of instances of A and B:

```
print(ref_count(a_id))  # 2
print(ref_count(b_id))  # 1
```

The instance of A has two references which is the variable `a` and the instance of B. And the instance of B has one reference which is the instance of A.

Seventh, check if both instances of A and B are in the memory:

```
print(object_exists(a_id))  # True
print(object_exists(b_id))  # True
```

Both of them exist.

Eighth, set a variable to `None` :

```
a = None
```

Ninth, get the reference counts of the instance of A and B:

```
print(ref_count(a_id))  # 1
print(ref_count(b_id))  # 1
```

Now, both reference count of the instance of A and B is 1.

Tenth, check if the instances exist:

```
print(object_exists(a_id))  # True
print(object_exists(b_id))  # True
```

Both of them still exist as expected.

Eleventh, start the garbage collector:

```
gc.collect()
```

When the garbage collector runs, it can detect the circular reference, destroy the objects, and reclaim the memory.

Twelveth, check if the instances of A and B exist:

```
print(object_exists(a_id))  # False
print(object_exists(b_id))  # False
```

Both of them don't exist anymore due to the garbage collector.

Thirteenth, get the reference counts of the instances of A and B:

```
print(ref_count(a_id))  # 0
print(ref_count(b_id))  # 0
```

Put it all together.

```
import gc
import ctypes


def ref_count(address):
    return ctypes.c_long.from_address(address).value


def object_exists(object_id):
    for object in gc.get_objects():
        if id(object) == object_id:
            return True

    return False


class A:
    def __init__(self):
        self.b = B(self)
```

```
        print(f'A: {hex(id(self))}, B: {hex(id(self.b))}')


class B:
    def __init__(self, a):
        self.a = a
        print(f'B: {hex(id(self))}, A: {hex(id(self.a))}')


# disable the garbage collector
gc.disable()

a = A()

a_id = id(a)
b_id = id(a.b)

print(ref_count(a_id))  # 2
print(ref_count(b_id))  # 1

print(object_exists(a_id))  # True
print(object_exists(b_id))  # True


a = None
print(ref_count(a_id))  # 1
print(ref_count(b_id))  # 1

print(object_exists(a_id))  # True
print(object_exists(b_id))  # True

# run the garbage collector
gc.collect()
```

```
# check if object exists
print(object_exists(a_id))  # False
print(object_exists(b_id))  # False

# reference count
print(ref_count(a_id))  # 0
print(ref_count(b_id))  # 0
```

Summary

- Python automatically manages memory for you using reference counting and garbage collector.
- Python garbage collector helps you avoid memory leaks by detecting circular references and destroy objects appropriately.
- Never disable the garbage collector unless you have a good reason to do so.