**Python Destructors to Destroy the Object**

Updated on: August 28, 2021 | [4 Comments](https://pynative.com/python-destructor/#comments)

**Destructor is a special method that is called when an object gets destroyed.** On the other hand, a [constructor](https://pynative.com/python-constructors/) is used to create and initialize an object of a class.

**After reading this article, you will learn:**

* How create a destructor in Python
* The use of \_\_del\_\_() method
* Wokring of a destructor

**Table of contents**

* [What is Destructor in Python?](https://pynative.com/python-destructor/#h-what-is-destructor-in-python)
* [Create Destructor using the \_\_del\_\_() Method](https://pynative.com/python-destructor/#h-create-destructor-using-the-__del__-method)
* [Important Points to Remember about Destructor](https://pynative.com/python-destructor/#h-important-points-to-remember-about-destructor)
* [Cases when Destructor doesn’t work Correctly](https://pynative.com/python-destructor/#h-cases-when-destructor-doesn-t-work-correctly)
  + [Circular Referencing](https://pynative.com/python-destructor/#h-circular-referencing)
  + [Exception in \_\_init\_\_ Method](https://pynative.com/python-destructor/#h-exception-in-__init__-method)
* [Summary and Quick Recap](https://pynative.com/python-destructor/#h-summary-and-quick-recap)

**What is Destructor in Python?**

In [object-oriented programming](https://pynative.com/python/object-oriented-programming/), A **destructor is called when an**[**object**](https://pynative.com/python-classes-and-objects/)**is deleted or destroyed**. Destructor is used to perform the clean-up activity before destroying the object, such as closing database connections or filehandle.

Python has a garbage collector that handles memory management automatically. For example, it cleans up the memory when an object goes out of scope.

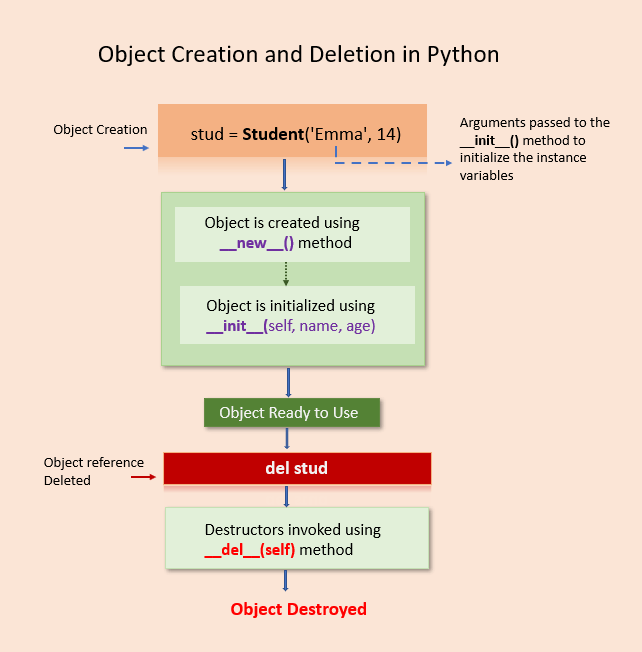
But it’s not just memory that has to be freed when an object is destroyed. We **must release or close the other resources object were using**, such as open files, database connections, cleaning up the buffer or cache. **To perform all those cleanup tasks we use destructor** in Python.

The destructor is the reverse of the [constructor](https://pynative.com/python-constructors/). The constructor is used to initialize objects, while the destructor is used to delete or destroy the object that releases the resource occupied by the object.

In Python, destructor is not called manually but completely automatic. **destructor gets called in the following two cases**

* When an object goes out of scope or
* The reference counter of the object reaches 0.

In Python, The special method \_\_del\_\_() is used to define a destructor. For example, when we execute del object\_name destructor gets called automatically and the object gets garbage collected.

Python destructor to destroy an object

**Create Destructor using the \_\_del\_\_() Method**

The magic method \_\_del\_\_() is used as the destructor in Python. The \_\_del\_\_() method will be implicitly invoked when all references to the object have been deleted, i.e., is when an object is eligible for the garbage collector.

This method is automatically called by Python when the instance is about to be destroyed. It is also called a finalizer or (improperly) a destructor.

**Syntax of destructor declaration**

**def** \_\_del\_\_(self):

# body of a destructor

Where,

* def: The keyword is used to define a method.
* \_\_del\_\_() Method: It is a reserved method. This method gets called as soon as all references to the object have been deleted
* self: The first argument self refers to the current object.

**Note**: The \_\_del\_\_() method arguments are optional. We can define a destructor with any number of arguments.

**Example**

Let’s see how to create a destructor in Python with a simple example. In this example, we’ll create a Class Student with a destructor. We’ll see: –

* How to implement a destructor
* how destructor gets executed when we delete the object.

**class** Student:

# constructor

**def** \_\_init\_\_(self, name):

**print**('Inside Constructor')

self.name = name

**print**('Object initialized')

**def** show(self):

**print**('Hello, my name is', self.name)

# destructor

**def** \_\_del\_\_(self):

**print**('Inside destructor')

**print**('Object destroyed')

# create object

s1 = Student('Emma')

s1.show()

# delete object

**del** s1

[**Run**](https://pynative.com/online-python-code-editor-to-execute-python-code/)

**Output**

Inside Constructor

Object initialized

Hello, my name is Emma

Inside destructor

Object destroyed

**Note**:

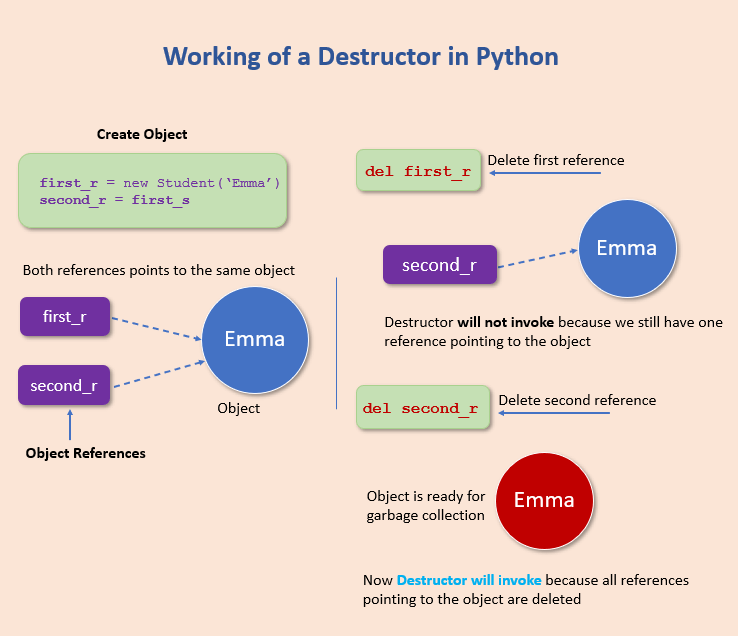
As you can see in the output, the \_\_del\_\_() method get called automatically is called when we deleted the object reference using del s1.

In the above code, we created one object. The s1 is the reference variable that is pointing to the newly created object.

The destructor has called when the reference to the object is deleted or the reference count for the object becomes zero

**Important Points to Remember about Destructor**

* The \_\_del\_\_ method is called for any object when the reference count for that object becomes zero.
* The reference count for that object becomes zero when the application ends, or we delete all references manually using the del keyword.
* The destructor will not invoke when we delete object reference. It will only invoke when all references to the objects get deleted.

Working of destructor

**Example**:

Let’s understand the above points using the example.

* First create object of a student class using s1 = student('Emma')
* Next, create a new object reference s2 by assigning s1 to s2 using s2=s1
* Now, both reference variables s1 and s2 point to the same object.
* Next, we deleted reference s1
* Next, we have added 5 seconds of sleep to the main thread to understand that destructors only invoke when all references to the objects get deleted.

**import** time

**class** Student:

# constructor

**def** \_\_init\_\_(self, name):

**print**('Inside Constructor')

self.name = name

**def** show(self):

**print**('Hello, my name is', self.name)

# destructor

**def** \_\_del\_\_(self):

**print**('Object destroyed')

# create object

s1 = Student('Emma')

# create new reference

# both reference points to the same object

s2 = s1

s1.show()

# delete object reference s1

**del** s1

# add sleep and observe the output

time.sleep(5)

**print**('After sleep')

s2.show()

[**Run**](https://pynative.com/online-python-code-editor-to-execute-python-code/)

**Output**:

Inside Constructor

Hello, my name is Emma

**After Sleep**

After sleep

Hello, my name is Emma

Object destroyed

* As you can see in the output destructors only invoked when all references to the objects get deleted.
* Also, the destructor is executed when the code (application) ends and the object is available for the garbage collector. (I.e., we didn’t delete object reference s2 manually using del s2).

**Cases when Destructor doesn’t work Correctly**

The \_\_del\_\_ is not a perfect solution to clean up a Python object when it is no longer required. In Python, the destructor behave behaves weirdly and doesn’t execute in the following two cases.

1. Circular referencing when two objects refer to each other
2. Exception occured in \_\_init\_\_() method

**Circular Referencing**

The \_\_del()\_\_() doesn’t work correctly in the case of circular referencing. In circular referencing occurs **when two objects refer to each other**.

When both objects go out of scope, Python doesn’t know which object to destroy first. So, to avoid any errors, it doesn’t destroy any of them.

In short, it means that the garbage collector does not know the order in which the object should be destroyed, so it doesn’t delete them from memory.

Ideally, the destructor must execute when an object goes out of scope, or its reference count reaches zero.

But the objects involved in this circular reference will remain stored in the memory as long as the application will run.

**Example**:

In the below example, ideally, both Vehicle and Car objects must be destroyed by the garbage collector after they go out of scope. Still, because of the circular reference, they remain in memory.

I’d recommend using Python’s [with statement](https://www.python.org/dev/peps/pep-0343/) for managing resources that need to be cleaned up.

**import** time

**class** Vehicle():

**def** \_\_init\_\_(self, **id**, car):

self.**id** = **id**;

# saving reference of Car object

self.dealer = car;

**print**('Vehicle', self.**id**, 'created');

**def** \_\_del\_\_(self):

**print**('Vehicle', self.**id**, 'destroyed');

**class** Car():

**def** \_\_init\_\_(self, **id**):

self.**id** = **id**;

# saving Vehicle class object in 'dealer' variable

# Sending reference of Car object ('self') for Vehicle object

self.dealer = Vehicle(**id**, self);

**print**('Car', self.**id**, 'created')

**def** \_\_del\_\_(self):

**print**('Car', self.**id**, 'destroyed')

# create car object

c = Car(12)

# delete car object

**del** c

# ideally destructor must execute now

# to observe the behavior

time.sleep(8)

[**Run**](https://pynative.com/online-python-code-editor-to-execute-python-code/)

**Output**:

Vehicle 12 created

Car 12 created

**Exception in \_\_init\_\_ Method**

In object-oriented programming, A constructor is a special method used to create and initialize an object of a class. using the \_\_init\_\_() method we can implement a constructor to initialize the object.

In OOP, if any exception occurs in the [constructor](https://pynative.com/python-constructors/) while initializing the object, the constructor destroys the object.

Likewise, in Python, if any exception occurs in the **init** method while initializing the object, the method **del** gets called. But actually, an object is not created successfully, and resources are not allocated to it

even though the object was never initialized correctly, the **del** method will try to empty all the resources and, in turn, may lead to another exception.

**Example**:

**class** Vehicle:

**def** \_\_init\_\_(self, speed):

**if** speed > 240:

**raise** Exception('Not Allowed');

self.speed = speed;

**def** \_\_del\_\_(self):

**print**('Release resources')

# creating an object

car = Vehicle(350);

# to delete the object explicitly

**del** car

[**Run**](https://pynative.com/online-python-code-editor-to-execute-python-code/)

**Output**:

Traceback (most recent call last):

Release resources

Exception: Not Allowed

**Summary and Quick Recap**

* In [object-oriented programming](https://pynative.com/python-object-oriented-programming/), A destructor is called when an object is deleted or destroyed.
* Destructor is used to perform the clean-up activity before destroying the object, such as closing database connections or filehandle.
* In Python we use \_\_del\_\_() method to perform clean-up task before deleting the object.
* The destructor will not invoke when we delete object reference. It will only invoke when all references to the objects get deleted.