**Constructors in Python**

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

* How to create a constructor to initialize an object in Python
* Different types of constructors
* Constructor overloading and chaining

**What is Constructor in Python?**

In object-oriented programming, **A constructor is a special method used to create and initialize an object of a class**. This method is defined in the class.

* The constructor is executed automatically at the time of object creation.
* The primary use of a constructor is to declare and initialize data member/ instance variables of a class. The constructor contains a collection of statements (i.e., instructions) that executes at the time of object creation to initialize the attributes of an object.

For example, when we execute obj = Sample(), Python gets to know that obj is an object of class Sample and calls the constructor of that class to create an object.

**Note**: In Python, internally, the \_\_new\_\_ is the method that creates the object, and \_\_del\_\_ method is called to destroy the object when the reference count for that object becomes zero.

In Python, Object creation is divided into two parts in **Object Creation** and **Object initialization**

* Internally, the \_\_new\_\_ is the method that creates the object
* And, using the \_\_init\_\_() method we can implement constructor to initialize the object.

**Syntax** of a constructor

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

# body of the constructor

Where,

* def: The keyword is used to define function.
* \_\_init\_\_() Method: It is a reserved method. This method gets called as soon as an object of a class is instantiated.
* self: The first argument self refers to the current object. It binds the instance to the \_\_init\_\_() method. It’s usually named self to follow the naming convention.

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

**Example: Create a Constructor in Python**

In this example, we’ll create a Class **Student** with an instance variable student name. we’ll see how to use a constructor to initialize the student name at the time of object creation.

**class** Student:

# constructor

# initialize instance variable

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

**print**('Inside Constructor')

self.name = name

**print**('All variables initialized')

# instance Method

**def** show(self):

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

# create object using constructor

s1 = Student('Emma')

s1.show()

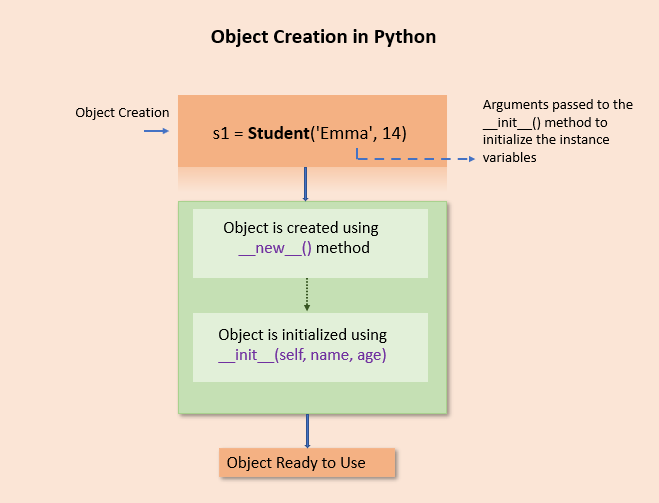
**Output**

Inside Constructor

All variables initialized

Hello, my name is Emma

* In the above example, an object s1 is created using the constructor
* While creating a Student object name is passed as an argument to the \_\_init\_\_() method to initialize the object.
* Similarly, various objects of the Student class can be created by passing different names as arguments.

Create an object in Python using a constructor

**Note**:

* For every object, the constructor will be executed only once. For example, if we create four objects, the constructor is called four times.
* In Python, every class has a constructor, but it’s not required to define it explicitly. Defining constructors in class is optional.
* Python will provide a default constructor if no constructor is defined.

**Types of Constructors**

In Python, we have the following three types of constructors.

* Default Constructor
* Non-parametrized constructor
* Parameterized constructor
* Types of constructor

**Default Constructor**

Python will provide a default constructor if no constructor is defined. Python adds a default constructor when we do not include the constructor in the class or forget to declare it. It does not perform any task but initializes the objects. It is an empty constructor without a body.

If you do not implement any constructor in your class or forget to declare it, the Python inserts a default constructor into your code on your behalf. This constructor is known as the default constructor.

It does not perform any task but initializes the objects. It is an empty constructor without a body.

**Note**:

* The default constructor is not present in the source py file. It is inserted into the code during compilation if not exists. See the below image.
* If you implement your constructor, then the default constructor will not be added.

**Example**:

**class** Employee:

**def** display(self):

**print**('Inside Display')

emp = Employee()

emp.display()

**Output**

Inside Display

As you can see in the example, we do not have a constructor, but we can still create an object for the class because Python added the default constructor during a program compilation.

**Non-Parametrized Constructor**

A constructor without any arguments is called a non-parameterized constructor. This type of constructor is used to initialize each object with default values.

This constructor doesn’t accept the arguments during object creation. Instead, it initializes every object with the same set of values.

**class** Company:

# no-argument constructor

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

self.name = "PYnative"

self.address = "ABC Street"

# a method for printing data members

**def** show(self):

**print**('Name:', self.name, 'Address:', self.address)

# creating object of the class

**cmp** = Company()

# calling the instance method using the object

**cmp**.show()

**Output**

Name: PYnative Address: ABC Street

As you can see in the example, we do not send any argument to a constructor while creating an object.

**Parameterized Constructor**

A constructor with defined parameters or arguments is called a parameterized constructor. We can pass different values to each object at the time of creation using a parameterized constructor.

The first parameter to constructor is self that is a reference to the being constructed, and the rest of the arguments are provided by the programmer. A parameterized constructor can have any number of arguments.

For example, consider a company that contains thousands of employees. In this case, while creating each employee object, we need to pass a different name, age, and salary. In such cases, use the parameterized constructor.

**Example**:

**class** Employee:

# parameterized constructor

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

self.name = name

self.age = age

self.salary = salary

# display object

**def** show(self):

**print**(self.name, self.age, self.salary)

# creating object of the Employee class

emma = Employee('Emma', 23, 7500)

emma.show()

kelly = Employee('Kelly', 25, 8500)

kelly.show()

**Output**

Emma 23 7500

Kelly 25 8500

In the above example, we define a parameterized constructor which takes three parameters.

**Constructor With Default Values**

Python allows us to define a constructor with default values. The default value will be used if we do not pass arguments to the constructor at the time of object creation.

The following example shows how to use the default values with the constructor.

**Example**

**class** Student:

# constructor with default values age and classroom

**def** \_\_init\_\_(self, name, age=12, classroom=7):

self.name = name

self.age = age

self.classroom = classroom

# display Student

**def** show(self):

**print**(self.name, self.age, self.classroom)

# creating object of the Student class

emma = Student('Emma')

emma.show()

kelly = Student('Kelly', 13)

kelly.show()

**Output**

Emma 12 7

Kelly 13 7

As you can see, we didn’t pass the age and classroom value at the time of object creation, so default values are used.

**Self Keyword in Python**

As you all know, the class contains instance variables and methods. Whenever we define instance methods for a class, we use self as the first parameter. Using self, we can **access the instance variable and instance method** of the object.

**The first argument self refers to the current object.**

Whenever we call an instance method through an object, the Python compiler implicitly passes object reference as the first argument commonly known as self.

It is not mandatory to name the first parameter as a self. We can give any name whatever we like, but it has to be the first parameter of an instance method.

**Example**

**class** Student:

# constructor

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

self.name = name

self.age = age

# self points to the current object

**def** show(self):

# access instance variable using self

**print**(self.name, self.age)

# creating first object

emma = Student('Emma', 12)

emma.show()

# creating Second object

kelly = Student('Kelly', 13)

kelly.show()

**Output**

Emma 12

Kelly 13

**Constructor Overloading**

Constructor overloading is a concept of having more than one constructor with a different parameters list in such a way so that each constructor can perform different tasks.

For example, we can create a three constructor which accepts a different set of parameters

**Python does not support constructor overloading**. If we define multiple constructors then, the interpreter will considers only the last constructor and throws an error if the sequence of the arguments doesn’t match as per the last constructor. The following example shows the same.

**Example**

**class** Student:

# one argument constructor

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

**print**("One arguments constructor")

self.name = name

# two argument constructor

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

**print**("Two arguments constructor")

self.name = name

self.age = age

# creating first object

emma = Student('Emma')

# creating Second object

kelly = Student('Kelly', 13)

**Output**

TypeError: \_\_init\_\_() missing 1 required positional argument: 'age'

* As you can see in the above example, we defined multiple constructors with different arguments.
* At the time of object creation, the interpreter executed the second constructor because Python always considers the last constructor.
* Internally, the object of the class will always call the last constructor, even if the class has multiple constructors.
* In the example when we called a constructor only with one argument, we got a type error.

**Constructor Chaining**

Constructors are used for instantiating an object. The task of the constructor is to assign value to data members when an object of the class is created.

Constructor chaining is the process of calling one constructor from another constructor. Constructor chaining is useful when you want to invoke multiple constructors, one after another, by initializing only one instance.

In Python, **constructor chaining is convenient when we are dealing with inheritance**. When an instance of a child class is initialized, the constructors of all the parent classes are first invoked and then, in the end, the constructor of the child class is invoked.

Using the super() method we can invoke the parent class constructor from a child class.

**Example**

**class** Vehicle:

# Constructor of Vehicle

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

**print**('Inside Vehicle Constructor')

self.engine = engine

**class** Car(Vehicle):

# Constructor of Car

**def** \_\_init\_\_(self, engine, max\_speed):

**super**().\_\_init\_\_(engine)

**print**('Inside Car Constructor')

self.max\_speed = max\_speed

**class** Electric\_Car(Car):

# Constructor of Electric Car

**def** \_\_init\_\_(self, engine, max\_speed, km\_range):

**super**().\_\_init\_\_(engine, max\_speed)

**print**('Inside Electric Car Constructor')

self.km\_range = km\_range

# Object of electric car

ev = Electric\_Car('1500cc', 240, 750)

**print**(f'Engine={ev.engine}, Max Speed={ev.max\_speed}, Km range={ev.km\_range}')

**Output**

Inside Vehicle Constructor

Inside Car Constructor

Inside Electric Car Constructor

Engine=1500cc, Max Speed=240, Km range=750

**Counting the Number of objects of a Class**

The constructor executes when we create the object of the class. For every object, the constructor is called only once. So for counting the number of objects of a class, we can add a counter in the constructor, which increments by one after each object creation.

**Example**

**class** Employee:

count = 0

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

Employee.count = Employee.count + 1

# creating objects

e1 = Employee()

e2 = Employee()

e2 = Employee()

**print**("The number of Employee:", Employee.count)

**Output**

The number of employee: 3

**Constructor Return Value**

In Python, the constructor does not return any value. Therefore, while declaring a constructor, we don’t have anything like return type. Instead, a constructor is implicitly called at the time of object instantiation. Thus, it has the sole purpose of initializing the instance variables.

The \_\_init\_\_() is required to return None. We can not return something else. If we try to return a non-None value from the \_\_init\_\_() method, it will raise TypeError.

**Example**

**class** Test:

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

self.**id** = i

**return** True

d = Test(10)

**Output**

TypeError: \_\_init\_\_() should return None, not 'bool'

**Conclusion and Quick recap**

In this lesson, we learned constructors and used them in object-oriented programming to design classes and create objects.

The below list contains the summary of the concepts we learned in this tutorial.

* A constructor is a unique method used to initialize an object of the class.
* Python will provide a default constructor if no constructor is defined.
* Constructor is not a method and doesn’t return anything. it returns None
* In Python, we have three types of constructor default, Non-parametrized, and parameterized constructor.
* Using self, we can access the instance variable and instance method of the object. The first argument self refers to the current object.
* Constructor overloading is not possible in Python.
* If the parent class doesn’t have a default constructor, then the compiler would not insert a default constructor in the child class.
* A child class constructor can also invoke the parent class constructor using the super() method.

**Python Destructors to Destroy the Object**

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

**What is Destructor in Python?**

In 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.

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. 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

**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()

**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 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)

**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 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

**Output**:

Traceback (most recent call last):

Release resources

Exception: Not Allowed

**Summary and Quick Recap**

* In 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