CONSTRUCTOR

• What are Constructors in Python?

• Types of Constructors in Python

• Rules of Python Constructor

• Multiple Constructors in Single Class

**Definatation:**

Constructors in Python is a special method of a class for creating and initializing an object instance at that class.

In python every class has a constructor, it's not required to define explicitly.

2)Constructors are generally used for instantiating an object.

3)The task of constructors is to initialize(assign values) to the data members of the class when an object of class is created.

The purpose of the constructor is to construct an object and assign a value to the object's members.

4)In Python the \_\_init\_\_() method is called the constructor and is always called when an object is created.

In Python, a constructor is a special method that is called when an object is created. The purpose of a python constructor is to assign values to the data members within the class when an object is initialized. The name of the constructor method is always \_\_init\_\_.

Here is an example of a simple class with a constructor:

class Person:

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

self.name = name

self.age = age

person = Person("John", 30)

print(person.name)

print(person.age)

Output:

John

30

In this example, the \_\_init\_\_ method is called when the Person object is created, and it sets the name and age attributes of the object.

The \_\_init\_\_ method is commonly referred to as the “constructor” because it is responsible for constructing the object. It is called automatically when the object is created, and it is used to initialize the object’s attributes.

**Types of Python Constructor**

In Python, there are two types of constructors:

• Default Constructor: A default constructor is a constructor that takes no arguments. It is used to create an object with default values for its attributes.

• Parameterized Constructor: A parameterized constructor is a constructor that takes one or more arguments. It is used to create an object with custom values for its attributes.

• Non-Parameterized Constructor: A non-parameterized constructor is a constructor that does not take any arguments. It is a special method in Python that is called when you create an instance of a class. The non-parameterized constructor is used to initialize the default values for the instance variables of the object.

**Default Constructors**

Default constructors are useful when you want to create an object with a predefined set of attributes, but you don’t want to specify the values of those attributes when the object is created.

Here is an example of a default constructor:

class Person:

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

self.name = "John"

self.age = 30

person = Person()

print(person.name)

print(person.age)

Output:

John

30

In this example, the \_\_init\_\_ method is the default constructor for the Person class. It is called automatically when the object is created, and it sets the default values for the name and age attributes.

**Parameterized Constructors**

Parameterized constructors are useful when you want to create an object with custom values for its attributes. They allow you to specify the values of the object’s attributes when the object is created, rather than using default values.

Here is an example of a class with a parameterized constructor:

class Person:

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

self.name = name

self.age = age

person = Person("Alice", 25)

print(person.name)

print(person.age)

Output:

Alice

25

In this example, the \_\_init\_\_ method is the parameterized constructor for the Person class. It takes two arguments, name and age, and it sets the values of the name and age attributes of the object to the values of these arguments.

**Non-Parameterized Constructors**

There is not necessarily a need for a non-parameterized constructor in Python. It is up to the programmer to decide whether to include a non-parameterized constructor in a class.

However, a non-parameterized constructor can be useful in the following cases:

• When you want to initialize the default values for the instance variables of an object.

• When you want to perform some operations when an object is created, such as opening a file or establishing a connection to a database.

• When you want to create a “skeleton” object that can be used as a template for creating other objects.

For example, consider the following class:

class MyClass:

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

self.arg1 = 10

self.arg2 = 20

In this case, the non-parameterized constructor is used to initialize the default values for the instance variables arg1 and arg2. If you create an instance of the MyClass class without passing any arguments, the default values will be used.

class MyClass:

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

self.arg1 = 10

self.arg2 = 20

obj = MyClass()

print(obj.arg1)

print(obj.arg2)

Output:

10

20

Note: If you do not define a non-parameterized constructor in your class, Python will automatically provide one for you. However, it is a good practice to define a constructor for your class, even if it is a non-parameterized constructor so that you have control over the initialization of your objects.

**Rules of Python Constructor**

Here are some rules for defining constructors in Python:

• The constructor method must be named \_\_init\_\_. This is a special name that is recognized by Python as the constructor method.

• The first argument of the constructor method must be self. This is a reference to the object itself, and it is used to access the object’s attributes and methods.

• The constructor method must be defined inside the class definition. It cannot be defined outside the class.

• The constructor method is called automatically when an object is created. You don’t need to call it explicitly.

• You can define both default and parameterized constructors in a class. If you define both, the parameterized constructor will be used when you pass arguments to the object constructor, and the default constructor will be used when you don’t pass any arguments.

Multiple Constructors in Single Class

You can have more than one constructor in a single Python class. This is known as “method overloading”. To do this, you will need to use the same method name (in this case, the name of the method will be the same as the name of the class) but define the method with different numbers or types of arguments.

Here is an example of a class with two constructors:

class MyClass:

def \_\_init\_\_(self, arg1, arg2):

self.arg1 = arg1

self.arg2 = arg2

def \_\_init\_\_(self, arg1):

self.arg1 = arg1

self.arg2 = None

The first constructor takes two arguments and sets them as instance variables, while the second constructor takes only one argument and sets it as an instance variable. When you create an instance of the MyClass class, Python will use the appropriate constructor based on the number of arguments that you pass.

For example:

class MyClass:

def \_\_init\_\_(self, arg1, arg2):

self.arg1 = arg1

self.arg2 = arg2

def \_\_init\_\_(self, arg1):

self.arg1 = arg1

self.arg2 = None

obj1 = MyClass(10, 20)

obj2 = MyClass(30)

n the first case, the first constructor will be called and arg1 will be set to 10 and arg2 will be set to 20. In the second case, the second constructor will be called and arg1 will be set to 30 and arg2 will be set to None.

Note: Python does not have true method overloading like some other programming languages. When you define multiple methods with the same name in a single class, only the last one will be used. However, you can use default values for arguments to achieve a similar effect.

**Python built-in class functions**

The built-in functions defined in the class are described in the following table.

SN Function Description

1 getattr(obj,name,default) It is used to access the attribute of the object.

2 setattr(obj, name,value) It is used to set a particular value to the specific attribute of an object.

3 delattr(obj, name) It is used to delete a specific attribute.

4 hasattr(obj, name) It returns true if the object contains some specific attribute.

Example

1. class Student:

2. def \_\_init\_\_(self, name, id, age):

3. self.name = name

4. self.id = id

5. self.age = age

6.

7. # creates the object of the class Student

8. s = Student("John", 101, 22)

9.

10. # prints the attribute name of the object s

11. print(getattr(s, 'name'))

12.

13. # reset the value of attribute age to 23

14. setattr(s, "age", 23)

15.

16. # prints the modified value of age

17. print(getattr(s, 'age'))

18.

19. # prints true if the student contains the attribute with name id

20.

21. print(hasattr(s, 'id'))

22. # deletes the attribute age

23. delattr(s, 'age')

24.

25. # this will give an error since the attribute age has been deleted

26. print(s.age)

**Output:**

John

23

True

AttributeError: 'Student' object has no attribute 'age'

**Built-in class attributes**

Along with the other attributes, a Python class also contains some built-in class attributes which provide information about the class.

The built-in class attributes are given in the below table.

SN Attribute Description

1 \_\_dict\_\_ It provides the dictionary containing the information about the class namespace.

2 \_\_doc\_\_ It contains a string which has the class documentation

3 \_\_name\_\_ It is used to access the class name.

4 \_\_module\_\_ It is used to access the module in which, this class is defined.

5 \_\_bases\_\_ It contains a tuple including all base classes.

Example

1. class Student:

2. def \_\_init\_\_(self,name,id,age):

3. self.name = name;

4. self.id = id;

5. self.age = age

6. def display\_details(self):

7. print("Name:%s, ID:%d, age:%d"%(self.name,self.id))

8. s = Student("John",101,22)

9. print(s.\_\_doc\_\_)

10. print(s.\_\_dict\_\_)

11. print(s.\_\_module\_\_)

Output:

None

{'name': 'John', 'id': 101, 'age': 22}

\_\_main\_\_