Python

Introduction to Programming

Comp07027

Lecture 12

Object Oriented Programming (OOP)

2

Constructors in Python

And Poly-morphism

Earlier …

… we introduced ***object oriented programming***.

We saw how we could create a ***class***, car, which was a blueprint for all ***instances*** of car we would want to create.

We chose the ***attributes*** registration, colour, make and model.

Constructors in Python

Constructor is a special method used to create and initialize an object of a class. On the other hand, a destructor is used to destroy the object.

Major constructor's concepts to know....

- How to create a constructor to initialize an object in Python
- Different types of constructors

Purpose of constructor

- 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.

- *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`

- `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.

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')

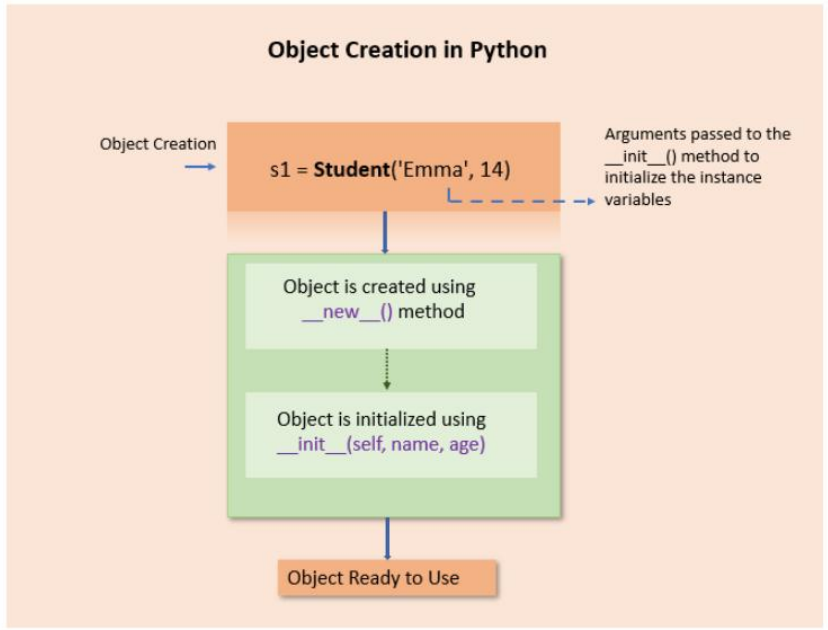
    # instance Method
    def show(self):
        print('Hello, my name is', self.name)


# create object using constructor
s1 = Student('Emma')
s1.show()
```

Output
Inside Constructor
All variables initialized
Hello, my name is Emma

Create a Constructor in Python

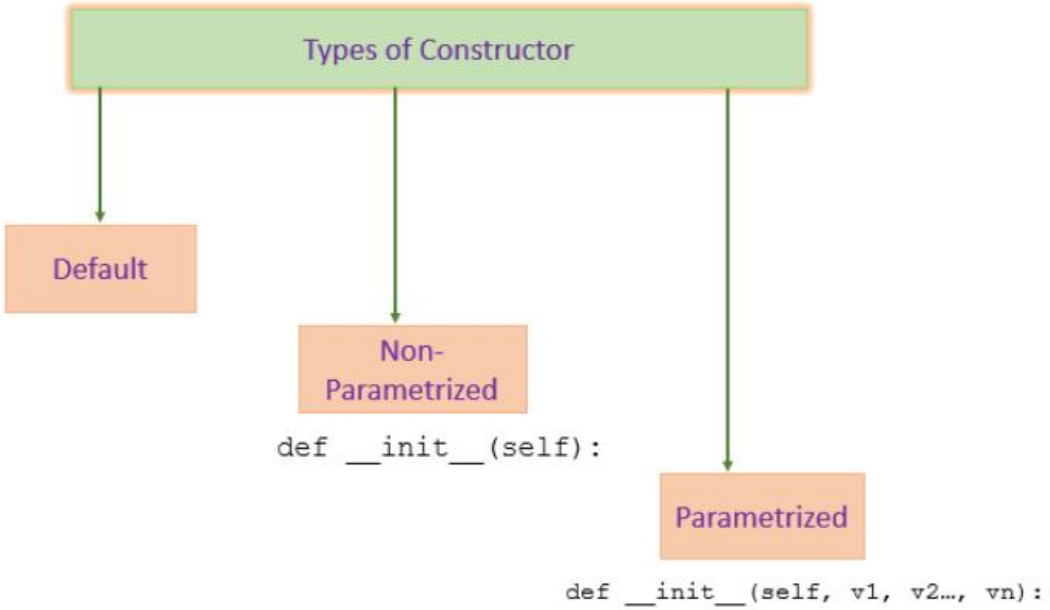
- In the above example, an object `s1` is created using the constructor
- While creating a Student object `name` is passed as an argument to the `__init__()` method to initialize the object.
- Similarly, various objects of the Student class can be created by passing different names



Create an object in Python using a constructor

Types of Constructors

- In Python, we have the following three types of constructors.
- Default Constructor
- Non-parametrized construct…
- 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.

- ```
  class Employee:

      def display(self):
          print('Inside Display')

  emp = Employee()
  emp.display()
  ```

- 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.
  ```

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 = "Scotland Academy"
        self.address = "Wuxi 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()
```

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.
-
```
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()
```

Earlier …

We created the ***class***.

We also created four instances of the car ***class*** (car1, car2, car3, car4).

```
class car:
    # initialisation method
    def __init__(self,registration,colour,make,model):
        self.registration = registration
        self.colour = colour
        self.make = make
        self.model =model


car1 = car("AB01CDE","White","Ford","Focus")
car2 = car("FG02HIJ","Blue","Vauxhall","Corsa")
car3 = car("KL03MNO","Green","Volkswagen","Polo")
car4 = car("PQ04RST","Red","Toyota","Yaris")
```

Earlier …

However, this is only half of the story.

Let’s look at another example.
We will create some classes and define their behaviours.

Creating a square

Here, we define a class called square which has a variable side.

We create a new square object with a side length of 5.

Then we calculate and display the area of the square.

However, is there a better way?

```
class square:
    def __init__(self, side):
        self.side = side


shape1 = square(5)


area = shape1.side ** 2


print("Area = ", area)
```

Methods

Now we define a **method**

calculate_area

which is wholly contained within the class square,

and accessed in the usual way by calling ***object.method*** i.e.

shape1.calculate_area()

```
class square:
    def __init__(self, side):
        self.side = side
    def calculate_area(self):
        return self.side ** 2


shape1 = square(5)

area = shape1.calculate_area()

print("Area = ", area)
```

Methods

So, every time we create an instance of the class square (object) we make calculate_area (method) automatically available using ***shape1.calculate_area()***

```
class square:
    def __init__(self, side):
        self.side = side
    def calculate_area(self):
        return self.side ** 2


shape1 = square(5)

area = shape1.calculate_area()


print("Area = ", area)
```

Adding a circle

Let’s add a class circle, with a method to calculate its area.

We create an instance of circle.

We calculate its area by calling its method.

```
class square:
    def __init__(self, side):
        self.side = side
    def calculate_area(self):
        return self.side ** 2


class circle:
    def __init__(self, radius):
        self.radius = radius
    def calculate_area(self):
        return 3.14 * (self.radius ** 2)



shape2 = circle(3)

area = shape2.calculate_area()


print("Area = ", area)
```

Adding a circle

Notice:

The method to calculate the area in both classes has the same name.

The call to calculate the area is exactly the same for both classes except for the object before the dot.

All the work is done within the object.

```
class square:
    def __init__(self, side):
        self.side = side
    def calculate_area(self):
        return self.side ** 2


class circle:
    def __init__(self, radius):
        self.radius = radius
    def calculate_area(self):
        return 3.14 * (self.radius ** 2)



shape2 = circle(3)

area = shape2.calculate_area()


print("Area = ", area)
```

Methods

So, we can create one of each shape and calculate the area of each one, using its own calculate_area!!

Or …..

we can really up the ante ..

```
class square:
    def __init__(self, side):
        self.side = side
    def calculate_area(self):
        return self.side ** 2

class circle:
    def __init__(self, radius):
        self.radius = radius
    def calculate_area(self):
        return 3.14 * (self.radius ** 2)

shape1 = square(5)
shape2 = circle(3)
area1 = shape1.calculate_area()
area2 = shape2.calculate_area()

print("Area = ", area1)
print("Area = ", area2)
```

Methods

Let’s create some shapes, say two circles and two squares.

In fact, let’s create a list of shapes:

```
list_of_shapes = [circle(3), square(5), circle(4), square(2)]
```

We know that lists in Python are very useful, so let’s use a loop to access each element in the list, one at a time.

```
for each_shape in list_of_shapes:
    print("Area = ", each_shape.calculate_area())
print("All done")
```

```
class square:
    def __init__(self, side):
        self.side = side
    def calculate_area(self):
        return self.side ** 2


class circle:
    def __init__(self, radius):
        self.radius = radius
    def calculate_area(self):
        return 3.14 * (self.radius ** 2)

list_of_shapes = [circle(3), square(5), circle(4), square(2)]

for each_shape in list_of_shapes:
    print("Area = ", each_shape.calculate_area())
print("All done")
```

```
class square:
    def __init__(self, side):
        self.side = side
    def calculate_area(self):
        return self.side ** 2

class circle:
    def __init__(self, radius):
        self.radius = radius
    def calculate_area(self):
        return 3.14 * (self.radius ** 2)

list_of_shapes = [circle(3), square(5), circle(4), square(2)]

for each_shape in list_of_shapes:
    print("Area = ", each_shape.calculate_area())
print("All done")
```

```
"C:\Program Files\Pythc
Area =  28.26
Area =  25
Area =  50.24
Area =  4
All done
```

Wow!!

We have stored different objects in a list,
and found all their areas in exactly the same way i.e.
by calling the method **calculate_area**
which has different definitions in each object.

All the hard work is done by the object itself!

Polymorphism

This ability to treat different types of object in the same way is an example of a powerful concept of OOP called **polymorphism**.

Questions??