Polymorphism in Python

Polymorphism is a concept of Object Oriented Programming, which means multiple forms or more than one form. Polymorphism enables using a single interface with input of different datatypes, different class or may be for different number of inputs.

In python as everything is an object hence by default a function can take anything as an argument but the execution of the function might fail as every function has some logic that it follows.

For example,

len("hello") # returns 5 as result

len([1,2,3,4,45,345,23,42]) # returns 8 as result

In this case the function len is polymorphic as it is taking **string** as input in the first case and is taking **list** as input in the second case.

In python, polymorphism is a way of making a function accept objects of different classes if they behave similarly.

**Method overriding** is a type of polymorphism in which a child class which is extending the parent class can provide different definition to any function defined in the parent class as per its own requirements.

Method Overloading

Method overloading or function overloading is a type of polymorphism in which we can define a number of methods with the same name but with a different number of parameters as well as parameters can be of different types. These methods can perform a similar or different function.

Python doesn't support method overloading on the basis of different number of parameters in functions.

Defining Polymorphic Classes

Imagine a situation in which we have a different class for shapes like Square, Triangle etc which serves as a resource to calculate the area of that shape. Each shape has a different number of dimensions which are used to calculate the area of the respective shape.

Now one approach is to define different functions with different names to calculate the area of the given shapes. The program depicting this approach is shown below:

class Square:

side = 5

def calculate\_area\_sq(self):

return self.side \* self.side

class Triangle:

base = 5

height = 4

def calculate\_area\_tri(self):

return 0.5 \* self.base \* self.height

sq = Square()

tri = Triangle()

print("Area of square: ", sq.calculate\_area\_sq())

print("Area of triangle: ", tri.calculate\_area\_tri())

Area of square: 25

Area of triangle: 10.0

The problem with this approach is that the developer has to remember the name of each function separately. In a much larger program, it is very difficult to memorize the name of the functions for every small operation. Here comes the role of method overloading.

Now let's change the name of functions to calculate the area and give them both same name calculate\_area() while keeping the function separately in both the classes with different definitions. In this case the type of object will help in resolving the call to the function. The program below shows the implementation of this type of polymorphism with class methods:

class Square:

side = 5

def calculate\_area(self):

return self.side \* self.side

class Triangle:

base = 5

height = 4

def calculate\_area(self):

return 0.5 \* self.base \* self.height

sq = Square()

tri = Triangle()

print("Area of square: ", sq.calculate\_area())

print("Area of triangle: ", tri.calculate\_area())

Area of square: 25

Area of triangle: 10.0

As you can see in the implementation of both the classes i.e. Square as well as Triangle has the function with same name calculate\_area(), but due to different objects its call get resolved correctly, that is when the function is called using the object sq then the function of class Square is called and when it is called using the object tri then the function of class Triangle is called.

Polymorphism with Class Methods

What we saw in the example above is again obvious behaviour. Let's use a loop which iterates over a tuple of objects of various shapes and call the area function to calculate area for each shape object.

sq = Square()

tri = Triangle()

for(obj in (sq, tri)):

obj.calculate\_area()

Now this is a better example of polymorphism because now we are treating objects of different classes as an object on which same function gets called.

Here python doesn't care about the type of object which is calling the function hence making the class method polymorphic in nature.

Polymorphism with Functions

Just like we used a loop in the above example, we can also create a function which takes an object of some shape class as input and then calls the function to calculate area for it. For example,

find\_area\_of\_shape(obj):

obj.calculate\_area()

sq = Square()

tri = Triangle()

# calling the method with different objects

find\_area\_of\_shape(sq)

find\_area\_of\_shape(tri)

In the example above we have used the same function find\_area\_of\_shape to calculate area of two different shape classes. The same function takes different class objects as arguments and executes perfectly to return the result. This is polymorphism.