Python - Polymorphism

What is Polymorphism in Python?

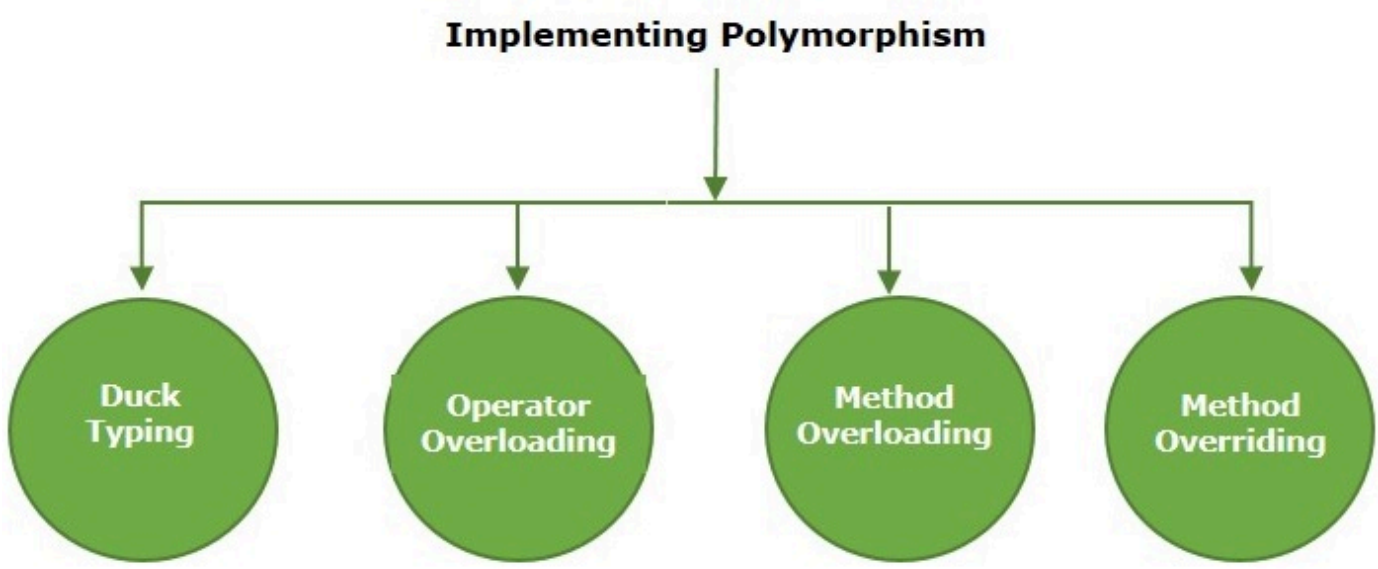
The term **polymorphism** refers to a function or method taking different forms in different contexts. Since Python is a dynamically typed language, polymorphism in Python is very easily implemented.

If a method in a parent class is overridden with different business logic in its different child classes, the base class method is a polymorphic method.

Ways of implementing Polymorphism in Python

There are four ways to implement polymorphism in Python –

- Duck Typing
- Operator Overloading
- Method Overriding
- Method Overloading



Duck Typing in Python

`Duck typing` is a concept where the type or class of an object is less important than the methods it defines. Using this concept, you can call any method on an object without checking its type, as long as the method exists.

This term is defined by a very famous quote that states: Suppose there is a bird that walks like a duck, swims like a duck, looks like a duck, and quaks like a duck then it probably is a duck.

Example

In the code given below, we are practically demonstrating the concept of duck typing.

</> Open Compiler

```
class Duck:
   def sound(self):
      return "Quack, quack!"

class AnotherBird:
   def sound(self):
      return "I'm similar to a duck!"

def makeSound(duck):
   print(duck.sound())

# creating instances
duck = Duck()
anotherBird = AnotherBird()
# calling methods
makeSound(duck)
makeSound(anotherBird)
```

When you execute this code, it will produce the following output –

```
Quack, quack!
I'm similar to a duck!
```

Method Overriding in Python

In **`method overriding`**, a method defined inside a subclass has the same name as a method in its superclass but implements a different functionality.

Example

As an example of polymorphism given below, we have **shape** which is an abstract class. It is used as parent by two classes circle and rectangle. Both classes override parent's draw() method in different ways.

</> Open Compiler

```
from abc import ABC, abstractmethod
class shape(ABC):
   @abstractmethod
   def draw(self):
      "Abstract method"
      return

class circle(shape):
   def draw(self):
      super().draw()
      print ("Draw a circle")
      return

class rectangle(shape):
   def draw(self):
      super().draw()
      print ("Draw a rectangle")
      return

shapes = [circle(), rectangle()]
for shp in shapes:
   shp.draw()
```

Output

When you run the above code, it will produce the following output –

```
Draw a circle
Draw a rectangle
```

The variable **shp** first refers to circle object and calls draw() method from circle class. In next iteration, it refers to rectangle object and calls draw() method from rectangle class. Hence draw() method in shape class is polymorphic.

Overloading Operators in Python

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you.

You could, however, define the **__add__** method in your class to perform vector addition and then the plus operator would behave as per expectation –

Example

</> Open Compiler

```
class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print (v1 + v2)
```

When the above code is executed, it produces the following result –

```
Vector(7,8)
```

Method Overloading in Python

When a class contains two or more methods with the same name but different number of parameters then this scenario can be termed as method overloading.

Python does not allow overloading of methods by default, however, we can use the techniques like variable-length argument lists, multiple dispatch and default parameters to achieve this.

Example

In the following example, we are using the variable-length argument lists to achieve method overloading.

</>

Open Compiler

```
def add(*nums):
   return sum(nums)

# Call the function with different number of parameters
result1 = add(10, 25)
result2 = add(10, 25, 35)

print(result1)
print(result2)
```

When the above code is executed, it produces the following result –

```
35
70
```