**Polymorphism**

poly means many and morphos means forms. If a variable, object or method perform different behavior according to situation, it is called polymorphism.

In programming, polymorphism means the same function name (but different signatures) being used for different types.

Polymorphism is one of the fundamental cornerstones of [Object Oriented Programming](https://www.scaler.com/topics/what-is-object-oriented-programming-oop/). It is the ability of one function to display multiple functionalities all together. It basically creates a structure that can use many forms of objects.

Polymorphism in Python involves a child class inheriting all the methods from the parent class.

Example:

4+5

= 9

“4”+”5”

=45

Or

# len() being used for a string

print(len("priyanka"))

= 8

# len() being used for a list

**print**(len([10, 20, 30]))

= 3

**How to use polymorphism?**

1. Overloading

I. Operator overloading

II. Method overloading

1. Overriding (polymorphism with inheritance)

Method overriding

1. Duck typing

**Operator overloading:**

If any operator performs additional actions other than what it is meant for, it is called operator overloading.

print("priyanka"\*3)

print(2\*3)

Thus, in the first example, the multiplication operator multiplied two numbers; but, in the second, since

multiplication of a string and an integer is not feasible, the character is displayed three times twice.

Thus, it demonstrates how a single operator may be used in a variety of ways

*#operator overloading Class example*

*# Operator Overloading*

*class A:*

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

*self.x = x*

*def \_\_add\_\_(self, other):*

*return self.x + other.x*

*class B:*

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

*self.x = x*

*a = A(100)*

*b = B(200)*

*print(a+b) #A.\_\_add\_\_(a, b)*

*#10+20 int.\_\_add\_\_(10, 20)*

*#'a'+'b' str.\_\_add\_\_('a', 'b')*

*#a+b A.\_\_add\_\_(a, b)*

***Self keyword?*** *: self represents the instance of the class. By using the “self”  we can access the attributes and methods of the class in python. It binds the attributes with the given arguments.  
The reason you need to use self. is because Python does not use the @ syntax to refer to instance attributes. Python decided to do methods in a way that makes the instance to which the method belongs be passed automatically, but not received automatically: the first parameter of methods is the instance the method is called on.*

*In more clear way you can say that SELF has following Characteristic-*

*Self is always pointing to Current Object.*

*Self is the first argument to be passed in Constructor and Instance Method. If you don’t provide it, it will cause an error.*

**Method Overloading**

When more than one method with the same name is defined in the same class, it is known as method overloading.

Method overloading is not actually possible in pyhton.

If we write same name method in 1 class and once we create object of that method it ll call the last method

Example: wrong method overloading

*class A:*

*def sum(self,a):*

*print("1st sum method",a)*

*def sum(self):*

*print("2nd sum method")*

*obj=A()*

*obj.sum()*

*obj.sum(20)*

it goes to 2nd sum method and gives error A.sum() takes 1 positional argumentbut 2 were gievn

In python, If a method is written such that it can perform more than one task, it is called method overloading.

Python allows classes to have different methods having the same name

*We can resolve method overloading using 3 different ways*

1. **Using multiple conditional statements**

*#resolving method overloading by giving multiple conditional statements*

*class Myclass:*

*def sum(self, a=None, b=None, c=None):*

*if a!=None and b!=None and c!=None:*

*s = a + b + c*

*elif a!=None and b!=None:*

*s = a + b*

*else:*

*s = 'Provide at least Two Numbers'*

*return s*

*obj = Myclass()*

*print(obj.sum(10))*

1. **By taking default arguments**

*#resolving method overloading - by taking default arguments*

*class Poly:*

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

*pass*

*def abc(self,a=10,b=20):*

*print("sum",a+b)*

*obj=Poly()*

*obj.abc()*

*obj.abc(45)*

*obj.abc(22,5)*

1. **By passing variable length arguments**

*#resolving method overloading by passing variable length arguments*

*class Poly:*

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

*pass*

*def abc(self,\*x):*

*print("first",x)*

*obj=Poly()*

*obj.abc(22,34,5)*

**Method Overriding**

If we write method in the both classes, parent class and child class then the parent class’s method is not available to the child class.

In this case only child class’s method is accessible which means child class’s method is replacing parent class’s method.

Method overriding is used when programmer want to modify the existing behavior of a Method.

*# Method Overriding*

*class Add:*

*def result(self, a, b):*

*print('Addition:', a+b)*

*class Multi(Add):*

*def result(self, a, b):*

*print('Multiplication:', a\*b)*

*m = Multi()*

*m.result(10, 20)*

*m = Add()*

*m.result(10, 20)*

**use of super()**

still if we want to use parent class we have to use super method

If we write method in the both classes, parent class and child class then the parent class’s method is not available to the child class.

In this case only child class’s method is accessible which means child class’s method is replacing parent class’s method.

**super ( )** method is used to call parent class’s constructor or methods from the child class.

*Syntax:- super().methodName()*

*# Method Overriding*

*class Add:*

*def result(self, a, b):*

*print('Addition:', a+b)*

*class Multi(Add):*

*def result(self, a, b):*

*super().result(10, 20) # Calling Parent Class's Method*

*print('Multiplication:', a\*b)*

*m = Multi()*

*m.result(10, 20)*

**Duck Typing**

In Python, we follow a principle - If ‘it walks like a duck and talks like a duck, it must be a duck’ which means python doesn’t care about which class of object it is, if it is an object and required behavior is present for that object then it will work. The type of object is distinguished only at runtime. This is called as duck typing.

Python doesn’t care about which class of object it is, in order to call an existing method on an object. If the method is defined on the object, then it will be called.

Example:

*# Duck Typing*

*class Duck:*

*def walk(self):*

*print("thapak thapak thapak thapak")*

*class Horse:*

*def walk(self):*

*print("tabdak tabdak tabdak tabdak")*

*class Cat:*

*def talk(self):*

*print("Meow Meow")*

*def myfunction(obj):*

*obj.walk()*

*d = Duck()*

*myfunction(d)*

*h = Horse()*

*myfunction(h)*

*c = Cat()*

*myfunction(c)*

**Strong Typing**

We can check whether the object passed to the method has the method being invoked or not.

hasattr ( ) Function is used to check whether the object has a method or not.

Syntax:- hasattr(object, attribute)

Where attribute can be a method or variable. If it is found in the object then this method returns True else False.

*#strong typing*

*class Duck:*

*def walk(self):*

*print("thapak thapak thapak thapak")*

*class Horse:*

*def walk(self):*

*print("tabdak tabdak tabdak tabdak")*

*class Cat:*

*def talk(self):*

*print("Meow Meow")*

*def myfunction(obj):*

*if hasattr(obj, 'walk'):*

*obj.walk()*

*if hasattr(obj, 'talk'):*

*obj.talk()*

*d = Duck()*

*myfunction(d)*

*h = Horse()*

*myfunction(h)*

*c = Cat()*

*myfunction(c)*