Polymorphism

- 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.

```
len("hello")        # returns 5 as result

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

- In the above 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.

Polymorphism

- 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 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.

area(a,b)
area(a,b,c)

X - Not Allowed in Python

Method Overriding

- When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding.
- We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

```
class Animal:
    def speak(self):
        print("speaking")
class Dog(Animal):
    def speak(self):
        print("Barking")
d = Dog()
d.speak()
```

- Overriding is required when we come to know that in subclass more concrete implementation to a base class method can be given.

Method Overriding

- Given below is a more practical example.

```
class Bank:
    def getroi(self):
        return 10;
class SBI(Bank):
    def getroi(self):
        return 6.5;

class HDFC(Bank):
    def getroi(self):
        return 7.5;
b1 = Bank()
b2 = SBI()
b3 = HDFC()
print("Bank Rate of interest:",b1.getroi());
print("SBI Rate of interest:",b2.getroi());
print("HDFC Rate of interest:",b3.getroi());
```

```
Bank Rate of interest: 10
SBI Rate of interest: 6.5
HDFC Rate of interest: 7.5
>>>
```

- Concrete implementations for getroi() are coming in SBI and HDFC classes.

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
```

Defining Polymorphic Classes

```
sq = Square()
tri = Triangle()
print("Area of square: ", sq.calculate_area_sq())
print("Area of triangle: ", tri.calculate_area_tri())
```

- 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.

Defining Polymorphic Classes

- 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
```

Defining Polymorphic Classes

```
sq = Square()
tri = Triangle()
print("Area of square: ", sq.calculate_area())
print("Area of triangle: ", tri.calculate_area())
```

- 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
- To clarify, 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.

Static Variables and Methods in Python

- Defining static variable and method is a common programming concept and is widely used in C++, Java, Php and many other programming languages for creating class variables and methods that belong to the class and are shared by all the objects of the class.

- In Python, there is no special keyword for creating static variables
- Class or Static variables are the variables that belong to the class and not to objects. Class or Static variables are shared amongst objects of the class.
- All variables which are assigned a value in the class declaration (outside any method or block) are class variables.
- Python allows providing same variable name for a static variable and an instance variable. But it is recommended not to provide same name variables to these variables to avoid confusion.

Static Methods in Python

- Just like static variables, static methods are the methods which are bound to the class rather than an object of the class and hence are called using the class name and not the objects of the class.
- As static methods are bound to the class, they cannot change the state of an object.
- To call a static method we don't need any class object it can be directly called using the class name
- In python there are two ways of defining a static method:
  - Using the staticmethod()
  - Using the @staticmethod

Defining Static Method using staticmethod()

```
class Shape:

    def info(msg):
        # show custom message
        print(msg)
        print("This class is used for representing different shapes.")

# create info static method
Shape.info = staticmethod(Shape.info)

Shape.info("Welcome to Shape class")
```

- In the program above, we declared the info method as static method outside the class using the staticmethod() function approach and after that we were able to call the info() method directly using the class *Shape*.

Defining Static Method using @staticmethod

```
class Shape:

    @staticmethod
    def info(msg):
        # show custom message
        print(msg)
        print("This class is used for representing different shapes.")

Shape.info("Welcome to Shape class")
```

- Using @staticmethod is a more modern and recommended approach of defining static method.

Static Variables and Methods in Python

In nutshell,

- Static variable and methods are used when we want to define some behavior or property specific to the class and which is something common for all the class objects.
- If you look closely, for a static method we don't provide the argument **self** because static methods don't operate on objects.

Operator Overloading in Python

- Operators are used in Python to perform specific operations on the given operands. The operation that any particular operator will perform on any predefined data type is already defined in Python.
- Each operator can be used in a different way for different types of operands. For example, + operator is used for **adding** two integers to give an integer as a result but when we use it with string operands then it **concatenates** the two operands provided.
- This different behavior of a single operator for different types of operands is called *Operator Overloading*.

Can + Operator add anything?

- NO. The + operator can add two integer values, two float values or can be used to concatenate two strings only because these behaviors have been defined in python.
- So, if you want to use the same operator to add two objects of some user defined class then you will have to define that behavior yourself and inform python about that. Hence, Something that goes below will not work:

```
class Complex:
    def __init__(self, r, i):
        self.real = r
        self.img = i

c1 = Complex(5,3)
c2 = Complex(2,4)
print("sum = ", c1+c2)
```

```
Traceback (most recent call last):
  File "C:/User/Python/Python37/oop11.py", line 8, in <module>
    print("sum = ", c1+c2)
TypeError: unsupported operand type(s) for +: 'Complex' and 'Complex'
>>>
```

we can see that the + operator is not supported in a user-defined class. But we can do the same by overloading the + operator for our class Complex .

Special Functions in Python

- Special functions in python are the functions which are used to perform special tasks.
- These special functions have __ as prefix and suffix to their name
- we have seen it in **__init__()** method which is also a special function

Special Functions to overload Mathematical Operators

Name	Symbol	Special Function
Addition	+	__add__(self, other)
Subtraction	-	__sub__(self, other)
Division	/	__truediv__(self, other)
Floor Division	//	__floordiv__(self, other)
Modulus(or Remainder)	%	__mod__(self, other)
Power	**	__pow__(self, other)

Special Functions to overload Assignment Operators

Name	Symbol	Special Function
Increment	+=	__iadd__(self, other)
Decrement	-=	__isub__(self, other)
Product	*=	__imul__(self, other)
Division	/=	__idiv__(self, other)
Modulus	%=	__imod__(self, other)
Power	**=	__ipow__(self, other)

Special Functions to overload Relational Operators

Name	Symbol	Special Function
Less than	<	__lt__(self, other)
Greater than	>	__gt__(self, other)
Equal to	==	__eq__(self, other)
Not equal	!=	__ne__(self, other)
Less than or equal to	<=	__le__(self, other)
Greater than or equal to	> =	__ge__(self, other)

Overloading + Operator

```
class Complex:
    # defining init method for class
    def __init__(self, r, i):
        self.real = r
        self.img = i

    # overloading the add operator using special function
    def __add__(self, sec):
        r = self.real + sec.real
        i = self.img + sec.img
        return complex(r,i)


c1 = Complex(5,3)
c2 = Complex(2,4)
print("sum = ",c1+c2)
```

Overloading + Operator

- In the program, **__add__()** is used to overload the + operator, i.e., when + operator is used with two Complex class objects then the function **__add__()** is called.