Polymorphism:- (Many Forms or Shapes)
It is an ability of an object to adopt the code to the type of the data processing.

OR

It refers to the use of a single type entity (Method, Operator, Object) to represent different types and different scenarios

OR

Implementing same thing in different ways or forms

Example:
Area of Polygon ==> Square, Rectangle, Triangle

Points to Remember:
1 Poly means many and morphism means forms
2 Forms means functionalities or logics.
3 The concept of defining multiple logics to perform same operation is known as a polymorphism.
4 Python is implicitly polymorphic

Example:

```
A=5; B=6
print(A+B)

A="Hello";B="PYTHON"
print(A+B)
```

NOTE:
In the above example A, B are Over-ridden, + is Overloaded.

Example:
Using The Same membership operator checking the variable existed in Data Structures.

```
print('x' in {'a','b','c','x'})
print('x' in ['a','b','c','x'])
print('x' in ('a','b','c','x'))
print('x' in {'a':'b','x':'c'})
```

Example:

```
print(len('PYTHON'))
print(len(list(range(1,25,3))))
```

NOTE:
len function is polymorphism because the same function taking different inputs.

Polymorphism different forms:
Dynamic/RunTime Polymorphism(Overriding)
i) Method overriding
ii) Constructor overriding

Static Polymorphism/CompileTime Polymorphism (Overloading)
i) Method Overloading
ii) Constructor Overloading
iii) Operator Overloading

Dynamic/RunTime Polymorphism: Method overriding
Through method overriding a class may "copy" another class, avoiding duplicated code, and at the same time enhance or customize part of it.
OR
It is the ability of a class to change the implementation of method provided by one of its ancestors.

Example:

```
class A():
    def display(self):
        print("Method belongs to Class A")
class B(A):
    pass

b1=B()
b1.display()
```

Example:

```
class A():
    def display(self):
        print("Method belongs to Class A")
class B(A):
    def display(self):
        print("Method belongs to Class B")

b1=B()
b1.display()
```

Example:

```
class ParentClass():
    def Transport(self):
        print("CYCLE")
class ChildClass(ParentClass):
    def Transport(self):
        print("BIKE")
class GrandChildClass(ChildClass):
    def Transport(self):
        print("CAR")

GG=GrandChildClass()
GG.Transport()
```

Example:

```
class Father():
    def Fname(self):
        print("FatherFirstName")
    def Lname(self):
        print("FatherLastName")
class Son(Father):
    def Fname(self):
        print("SonFirstName")

ObjSon=Son()
ObjSon.Fname()
ObjSon.Lname()
```

NOTE:

If method names are different, It never satisfy method overriding concept.

Example:

```
class Person():
    def __init__(self,name,age):
        self.name=name
        self.age=age

    def DisplayData(self):#OverriddenMethod
        print("Parent Class Method")
        print("Name is: ",self.name)
        print("Age is: ",self.age)

class Employee(Person):
    def __init__(self,name,age,Id):
        super().__init__(name,age)
        self.empID=Id

    def DisplayData(self):#OverriddenMethod
        print("Employee Class Display-Data Method")
        print("Name is: ",self.name)
        print("Age is: ",self.age)
        print("Emp ID is: ",self.empID)

class Developer(Employee):
    def __init__(self,name,age,Id,Exp):
        super().__init__(name,age,Id)
        self.Exp=Exp

    def DisplayData(self):#OverriddenMethod
        print("Developer Class Display-Data Method")
        print("Name is: ",self.name)
        print("Age is: ",self.age)
        print("Emp ID is: ",self.empID)
        print("Exp is: ",self.Exp)

#Person Class Insatnce or Object
PP=Person("Rama",45)
PP.DisplayData()
print()
#Employee Class Instance or Object
EE=Employee("Rama",45,101)
EE.DisplayData()
print()
#Developer Class Instance or Object
Dev=Developer("Rama",45,"101","5 Yrs")
Dev.DisplayData()
```

Constructor Overriding:

Example:

```
class PConstructor():
    def __init__(self):
        print('Welcome to Main Constructor')
class CContstructor(PConstructor):
    def __init__(self):
        print('Good Bye Sub Constructor')
```

```
#Creating Instance
HelloCons=CContstructor()
```

NOTE:
If child class does not contain constructor then parent class constructor will be executed

Example:

```
class PConstructor():
    def __init__(self):
        print('Welcome To Main Constructor')
class CContstructor(PConstructor):
    pass

#Creating Instance
HelloCons=CContstructor()
```

Static Polymorphism: Method & Constructor Overloading:
PYTHON does not supports Method Overloading & Constructor Overloading, It is dynamically typed language. If we are trying to declare multiple methods with same name and different number of arguments then Python will always consider only last method.

Example:

```
class CalCulate():
    def add(self,a,b):
        return a+b
    def add(self,a,b,c):
        return a+b+c
obj=CalCulate()
print(obj.add(3,1))
print(obj.add(3,1,3))
```

NOTE:
TypeError: add() missing 1 required positional argument: 'c'
NOTE:
In Python it always calls latest implemention of the method

Example: We can achieve through default arguments.!

```
class MethodOverLoading():
    def Total(self,a,b,c=0,d=0):
        Sum=a+b+c+d
        return Sum

MOL=MethodOverLoading()
Tot1=MOL.Total(7,8)
print(Tot1)

Tot2=MOL.Total(7,8,9)
print(Tot2)

Tot3=MOL.Total(7,8,9,10)
print(Tot3)
```

Constructor Overloading
It is not possible in Python. If we define multiple constructors then

the last constructor will be considered.

Example:

```
class Constructor1():
    def __init__(self):
        print('Welcome to Constructor1')
    def __init__(self):
        print('Welcome To Constructor2')

#Creating Object
CC=Constructor1()
```

Example:

```
class Constructor1():
    def __init__(self,x):
        print('Welcome to Constructor1')
    def __init__(self,x,y):
        print('Welcome To Constructor2')

#Creating Object
CC=Constructor1(1)
```

NOTE:

```
TypeError: __init__() missing 1 required positional argument: 'y'
```

Example:

```
class Constructor1():
    def __init__(self,x):
        print('Welcome to Constructor1')
    def __init__(self,x,y):
        print('Welcome To Constructor2')

#Creating Object
CC=Constructor1(1,2)
```

Python Operator Overloading:
Assigning extra work to operators is called operators overloading
OR
The assignment of more than one function to a particular operator

Example:Basics of Operators

```
x=10; y=20
print(x+y)
print(int.__add__(x,y))
#Addition of Integers it not possible without __add__ method defined
inside 'int' class
```

Example:

```
print(10+20)#30
print('KSraju'+'DataScientist')#KSrajuDataScientist
```

Example:

```
print(10*2)#20
print('Data Science '*4,end=" ")#Data Science Data Science Data
Science Data Science
```

Example:

```
a=4;b=6
print(a+b)
c="Hello ";d="World!"
print(c+d)
```

Example:

```
class EmpSlary():
    def __init__(self, salary):
        self.salary=salary

Emp1=EmpSlary(300000)
Emp2=EmpSlary(200000)
TotalSal=Emp1+Emp2
print("TotalSalaryOfEmployeesIS: ",TotalSal)
```

NOTE:
1 TypeError: unsupported operand type(s) for +: 'EmpSlary' and 'EmpSlary'
2 The program defines + operator is not for objects adding.

Example:

```
class Subject():
    def __init__(self,Course):
        self.Course=Course

Sub1=Subject(89)
Sub2=Subject(56)
print(Sub1+Sub2)
```

O/P:

```
TypeError: unsupported operand type(s) for +: 'Subject' and 'Subject'
```

Points to Remember:
1 In Python every operator has Magic Method.
2 To overload any operator we have to override that Method in our class.
3 Internally + operator is implemented by using add () method.
4 This method is called magic method for + operator

Operator Overloading Special Functions in Python

Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Div	p1 // p2	p1.__floordiv__p2()
Modulus	p1 % p2	p1.__mod__p2()

Example:

```
class EmpSlary():
    def __init__(self, salary):
        self.salary=salary

    def __add__(self,other):
        return self.salary+other.salary
```

```
Emp1=EmpSlary(300000)
Emp2=EmpSlary(200000)
TotalSal=Emp1+Emp2
print("TotalSalaryOfEmployeesIS: ",TotalSal)
```

NOTE: Return statement plays major role here, whatever value you want to return.

Example:

```
class EmpSlary():
    def __init__(self, salary):
        self.salary=salary

    def __add__(self,other):
        return 3000

Emp1=EmpSlary(300000)
Emp2=EmpSlary(200000)
TotalSal=Emp1+Emp2
print("TotalSalaryOfEmployeesIS: ",TotalSal)
```

Example:

```
class Subject():
    def __init__(self,Course):
        self.Course=Course

    def __add__(self,other):
        return self.Course+other.Course

Sub1=Subject(89)
Sub2=Subject(56)
print(Sub1+Sub2)
```

NOTE:
Python supports + operator for , Addition, Concatenation, and Objects Adding, JAVA Never..!!

Example:

```
class Subject():

    def __init__(self,Course):
        self.Course=Course

    def __sub__(self,Other):
        return self.Course-Other.Course

    def __add__(self,Other):
        return self.Course+Other.Course

SubjectOne=Subject(90)
SubjectTwo=Subject(49)
print(SubjectOne-SubjectTwo)
print(SubjectOne-SubjectTwo)
print(SubjectOne+SubjectTwo)
print(SubjectOne+SubjectTwo)
```

Example:

```
class Subject():
    def __init__(self,Course):
        self.Course=Course

    def __add__(self,other):
        return self.Course+other.Course

    def __mul__(self,other):
        return self.Course*other.Course

Sub1=Subject(89)
Sub2=Subject(56)
print(Sub1+Sub2)
print(Sub1*Sub2)
```

Example:

```
class Subject():
    def __init__(self,Sub,Price):
        self.Sub=Sub
        self.Price=Price

    def __gt__(self,other):
        return self.Price>other.Price

Sub1=Subject('Maths',100)
Sub2=Subject('Social',200)
print(Sub2>Sub1)
```

Example:

```
class Subject():
    def __init__(self,Course):
        self.Course=Course

    def __add__(self,other1,other2):
        return self.Course+other1.Course+other2.Course

Sub1=Subject(89)
Sub2=Subject(56)
Sub3=Subject(76)
print(Sub1+Sub2+Sub3)
```

NOTE: TypeError: __add__() missing 1 required positional argument: 'other2'

Duck-Typing

If its looks like a duck and quacks like duck, It is duck.

OR

It is an application of the duck test in type safety. It requires that type checking be deferred to runtime, and is implemented by means of dynamic typing or reflection. It is type of Polymorphism.

OR

It is a concept related to dynamic typting, where the type or the class of an object is less important than the method it defines. We do not check types at all.

Example:

```
def Cal(a,b):
    return a+b
print(Cal(1,2))
print(Cal("Hello","PYTHON"))
print(Cal(1,"PYTHON"))
```

```
Error:  TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Example:

```
class Duck():
    def Sound(self):
        print("Quack-Quack-Quack-Quack!")

class Dog():
    def Sound(self):
        print("Woof Woof!")

class Cat():
    def Sound(self):
        print("Meow Meow Meow!")

def AllSounds(obj):
    obj.Sound()

x=Duck()
AllSounds(x)
```

NOTE:
It doesn't matter to which class or object x belongs, what matter is if that object has method sound define in it.

Example:

```
class Lion():
    def Sound(self):
        print('Roar, Roar, Roaring..!!')

class Cow():
    def Sound(self):
        print('Ammmmmm..!!')

def AnimalSound(obj):
    obj.Sound()

PyList=Lion()
AnimalSound(PyList)
```

Example:

```
class Lion():
    def Sound(self):
        print('Roar, Roar, Roaring..!!')

class Cow():
    def Sound(self):
        print('Ammmmmm..!!')

def MyFun(obj):
    obj.Sound()
```

```
PyList=[Lion(),Cow()]
for obj in PyList:
    MyFun(obj)
```