Polymorphism

Poly means many. Morphs means forms.
Polymorphism means 'Many Forms'.

Eg1: Yourself is best example of polymorphism.In front of Your parents You will have one type of behaviour and with friends another type of behaviour.Same person but different behaviours at different places,which is nothing but polymorphism.

Eg2: + operator acts as concatenation and arithmetic addition

Eg3: * operator acts as multiplication and repetition operator

Eg4: The Same method with different implementations in Parent class and child classes.(overriding)

Related to polymorphism the following 4 topics are important

1. Duck Typing Philosophy of Python

2. Overloading
- **1. Operator Overloading**
- **2. Method Overloading**
- **3. Constructor Overloading**

3. Overriding
- **1. Method overriding**
- **2. constructor overriding**

1. Duck Typing Philosophy of Python:

In Python we cannot specify the type explicitly. Based on provided value at runtime the type will be considered automatically. Hence Python is considered as Dynamically Typed Programming Language.

```
def f1(obj):
    obj.talk()
```

What is the type of obj? We cannot decide at the beginning. At runtime we can pass any type.Then how we can decide the type?
At runtime if 'it walks like a duck and talks like a duck,it must be duck'. Python follows this principle. This is called Duck Typing Philosophy of Python.

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Demo Program:

```
1)  class Duck:
2)      def talk(self):
3)          print('Quack.. Quack..')
4)
5)  class Dog:
6)      def talk(self):
7)          print('Bow Bow..')
8)
9)  class Cat:
10)     def talk(self):
11)         print('Moew Moew ..')
12)
13) class Goat:
14)     def talk(self):
15)         print('Myaah Myaah ..')
16)
17) def  f1(obj):
18)     obj.talk()
19)
20) l=[Duck(),Cat(),Dog(),Goat()]
21) for obj in l:
22)     f1(obj)
```

Output:

```
Quack.. Quack..
Moew Moew ..
Bow Bow..
Myaah Myaah ..
```

The problem in this approach is if obj does not contain talk() method then we will get AttributeError

Eg:

```
1)  class Duck:
2)      def talk(self):
3)          print('Quack.. Quack..')
4)
5)  class Dog:
6)      def bark(self):
7)          print('Bow Bow..')
8)  def  f1(obj):
9)      obj.talk()
10)
11) d=Duck()
12) f1(d)
13)
```

```
14) d=Dog()
15) f1(d)
```

Output:

```
D:\durga_classes>py test.py
Quack.. Quack..
Traceback (most recent call last):
  File "test.py", line 22, in <module>
    f1(d)
  File "test.py", line 13, in f1
    obj.talk()
AttributeError: 'Dog' object has no attribute 'talk'
```

But we can solve this problem by using hasattr() function.

hasattr(obj,'attributename')
attributename can be method name or variable name

Demo Program with hasattr() function:

```
1)   class Duck:
2)       def talk(self):
3)           print('Quack.. Quack..')
4)
5)   class Human:
6)       def talk(self):
7)           print('Hello Hi...')
8)
9)   class Dog:
10)      def bark(self):
11)          print('Bow Bow..')
12)
13) def  f1(obj):
14)      if hasattr(obj,'talk'):
15)          obj.talk()
16)      elif hasattr(obj,'bark'):
17)          obj.bark()
18)
19) d=Duck()
20) f1(d)
21)
22) h=Human()
23) f1(h)
24)
25) d=Dog()
26) f1(d)
27) Myaah Myaah Myaah...
```

Overloading:

We can use same operator or methods for different purposes.

Eg1: + operator can be used for Arithmetic addition and String concatenation
print(10+20)#30
print('durga'+'soft')#durgasoft

Eg2: * operator can be used for multiplication and string repetition purposes.
print(10*20)#200
print('durga'*3)#durgadurgadurga

Eg3: We can use deposit() method to deposit cash or cheque or dd
deposit(cash)
deposit(cheque)
deposit(dd)

There are 3 types of overloading
1. Operator Overloading
2. Method Overloading
3. Constructor Overloading

1. Operator Overloading:

We can use the same operator for multiple purposes, which is nothing but operator overloading.

Python supports operator overloading.

Eg1: + operator can be used for Arithmetic addition and String concatenation
print(10+20)#30
print('durga'+'soft')#durgasoft

Eg2: * operator can be used for multiplication and string repetition purposes.
print(10*20)#200
print('durga'*3)#durgadurgadurga

Demo program to use + operator for our class objects:

```
1)  class Book:
2)      def __init__(self,pages):
3)          self.pages=pages
4)
5)  b1=Book(100)
6)  b2=Book(200)
7)  print(b1+b2)
```

```
D:\durga_classes>py test.py
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    print(b1+b2)
TypeError: unsupported operand type(s) for +: 'Book' and 'Book'
```

We can overload + operator to work with Book objects also. i.e Python supports Operator Overloading.

For every operator Magic Methods are available. To overload any operator we have to override that Method in our class.
Internally + operator is implemented by using __add__() method.This method is called magic method for + operator. We have to override this method in our class.

Demo program to overload + operator for our Book class objects:

```
1)  class Book:
2)      def __init__(self,pages):
3)          self.pages=pages
4)
5)      def __add__(self,other):
6)          return self.pages+other.pages
7)
8)  b1=Book(100)
9)  b2=Book(200)
10) print('The Total Number of Pages:',b1+b2)
```

Output: The Total Number of Pages: 300

The following is the list of operators and corresponding magic methods.

```
+   ---> object.__add__(self,other)
-   ---> object.__sub__(self,other)
*   ---> object.__mul__(self,other)
/   ---> object.__div__(self,other)
//   ---> object.__floordiv__(self,other)
%   ---> object.__mod__(self,other)
**   ---> object.__pow__(self,other)
+=   ---> object.__iadd__(self,other)
-=   ---> object.__isub__(self,other)
*=   ---> object.__imul__(self,other)
/=   ---> object.__idiv__(self,other)
//=   ---> object.__ifloordiv__(self,other)
%=   ---> object.__imod__(self,other)
**=   ---> object.__ipow__(self,other)
<   ---> object.__lt__(self,other)
<=   ---> object.__le__(self,other)
>   ---> object.__gt__(self,other)
>=   ---> object.__ge__(self,other)
```

== ---> object.__eq__(self,other)
!= ---> object.__ne__(self,other)

Overloading > and <= operators for Student class objects:

```
1)  class Student:
2)      def __init__(self,name,marks):
3)          self.name=name
4)          self.marks=marks
5)      def __gt__(self,other):
6)          return self.marks>other.marks
7)      def __le__(self,other):
8)          return self.marks<=other.marks
9)
10)
11) print("10>20 =",10>20)
12) s1=Student("Durga",100)
13) s2=Student("Ravi",200)
14) print("s1>s2=",s1>s2)
15) print("s1<s2=",s1<s2)
16) print("s1<=s2=",s1<=s2)
17) print("s1>=s2=",s1>=s2)
```

Output:

```
10>20 = False
s1>s2= False
s1<s2= True
s1<=s2= True
s1>=s2= False
```

Program to overload multiplication operator to work on Employee objects:

```
1)  class Employee:
2)      def __init__(self,name,salary):
3)          self.name=name
4)          self.salary=salary
5)      def __mul__(self,other):
6)          return self.salary*other.days
7)
8)  class TimeSheet:
9)      def __init__(self,name,days):
10)         self.name=name
11)         self.days=days
12)
13) e=Employee('Durga',500)
14) t=TimeSheet('Durga',25)
15) print('This Month Salary:',e*t)
```

Output: This Month Salary: 12500

2. Method Overloading:

If 2 methods having same name but different type of arguments then those methods are said to be overloaded methods.

Eg: m1(int a)
m1(double d)

But in Python Method overloading is not possible.
If we are trying to declare multiple methods with same name and different number of arguments then Python will always consider only last method.

Demo Program:

```
1)  class Test:
2)      def m1(self):
3)          print('no-arg method')
4)      def m1(self,a):
5)          print('one-arg method')
6)      def m1(self,a,b):
7)          print('two-arg method')
8)
9)  t=Test()
10) #t.m1()
11) #t.m1(10)
12) t.m1(10,20)
```

Output: two-arg method

In the above program python will consider only last method.

How we can handle overloaded method requirements in Python:

Most of the times, if method with variable number of arguments required then we can handle with default arguments or with variable number of argument methods.

Demo Program with Default Arguments:

```
1)  class Test:
2)      def sum(self,a=None,b=None,c=None):
3)          if a!=None and b!= None and c!= None:
4)              print('The Sum of 3 Numbers:',a+b+c)
5)          elif a!=None and b!= None:
6)              print('The Sum of 2 Numbers:',a+b)
7)          else:
8)              print('Please provide 2 or 3 arguments')
9)
10) t=Test()
```

```
11) t.sum(10,20)
12) t.sum(10,20,30)
13) t.sum(10)
```

Output:
The Sum of 2 Numbers: 30
The Sum of 3 Numbers: 60
Please provide 2 or 3 arguments

Demo Program with Variable Number of Arguments:

```
1)  class Test:
2)      def sum(self,*a):
3)          total=0
4)          for x in a:
5)              total=total+x
6)          print('The Sum:',total)
7)
8)
9)  t=Test()
10) t.sum(10,20)
11) t.sum(10,20,30)
12) t.sum(10)
13) t.sum()
```

3. Constructor Overloading:

Constructor overloading is not possible in Python.
If we define multiple constructors then the last constructor will be considered.

```
1)  class Test:
2)      def __init__(self):
3)          print('No-Arg Constructor')
4)
5)      def __init__(self,a):
6)          print('One-Arg constructor')
7)
8)      def __init__(self,a,b):
9)          print('Two-Arg constructor')
10) #t1=Test()
11) #t1=Test(10)
12) t1=Test(10,20)
```

Output: Two-Arg constructor

In the above program only Two-Arg Constructor is available.

But based on our requirement we can declare constructor with default arguments and variable number of arguments.

Constructor with Default Arguments:

```
1)  class Test:
2)      def __init__(self,a=None,b=None,c=None):
3)          print('Constructor with 0|1|2|3 number of arguments')
4)
5)  t1=Test()
6)  t2=Test(10)
7)  t3=Test(10,20)
8)  t4=Test(10,20,30)
```

Output:

```
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
```

Constructor with Variable Number of Arguments:

```
1)  class Test:
2)      def __init__(self,*a):
3)          print('Constructor with variable number of arguments')
4)
5)  t1=Test()
6)  t2=Test(10)
7)  t3=Test(10,20)
8)  t4=Test(10,20,30)
9)  t5=Test(10,20,30,40,50,60)
```

Output:

```
Constructor with variable number of arguments
Constructor with variable number of arguments
Constructor with variable number of arguments
Constructor with variable number of arguments
Constructor with variable number of arguments
```

Method overriding:

What ever members available in the parent class are bydefault available to the child class through inheritance. If the child class not satisfied with parent class implementation then child class is allowed to redefine that method in the child class based on its requirement. This concept is called overriding.
Overriding concept applicable for both methods and constructors.

Demo Program for Method overriding:

```
1)  class P:
2)      def property(self):
3)          print('Gold+Land+Cash+Power')
4)      def marry(self):
5)          print('Appalamma')
6)  class C(P):
7)      def marry(self):
8)          print('Katrina Kaif')
9)
10) c=C()
11) c.property()
12) c.marry()
```

Output:
Gold+Land+Cash+Power
Katrina Kaif

From Overriding method of child class,we can call parent class method also by using super() method.

```
1)  class P:
2)      def property(self):
3)          print('Gold+Land+Cash+Power')
4)      def marry(self):
5)          print('Appalamma')
6)  class C(P):
7)      def marry(self):
8)          super().marry()
9)          print('Katrina Kaif')
10)
11) c=C()
12) c.property()
13) c.marry()
```

Output:
Gold+Land+Cash+Power
Appalamma
Katrina Kaif

Demo Program for Constructor overriding:

```
1)  class P:
2)      def __init__(self):
3)          print('Parent Constructor')
4)
5)  class C(P):
6)      def __init__(self):
7)          print('Child Constructor')
8)
9)  c=C()
```

Output: Child Constructor

In the above example,if child class does not contain constructor then parent class constructor will be executed

From child class constuctor we can call parent class constructor by using super() method.

Demo Program to call Parent class constructor by using super():

```
1)  class Person:
2)      def __init__(self,name,age):
3)          self.name=name
4)          self.age=age
5)
6)  class Employee(Person):
7)      def __init__(self,name,age,eno,esal):
8)          super().__init__(name,age)
9)          self.eno=eno
10)         self.esal=esal
11)
12)     def display(self):
13)         print('Employee Name:',self.name)
14)         print('Employee Age:',self.age)
15)         print('Employee Number:',self.eno)
16)         print('Employee Salary:',self.esal)
17)
18) e1=Employee('Durga',48,872425,26000)
19) e1.display()
20) e2=Employee('Sunny',39,872426,36000)
21) e2.display()
```

Output:

```
Employee Name: Durga
Employee Age: 48
Employee Number: 872425
Employee Salary: 26000
Employee Name: Sunny
Employee Age: 39
```

Employee Number: 872426
Employee Salary: 36000