(7.5) Polymorphism , Duck Typing

1. Introduction

Polymorphism

Overloading

Over-riding

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

Eg1: + operator acts as concatenation and arithmetic addition

Eg2: * operator acts as multiplication and repetition operator

Eg3: The Same method with different implementations in Parent class and child
classes.(overriding)
```

Related to polymorphism the following 3 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
```

2. Duck Typing Philosophy of Python

Methods are more important than class type

```
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.
```

Program - 1

In [3]:

```
class Duck:
    def talk(self):
        print('Quack.. Quack..')

class Dog:
    def talk(self):
        print('Bow Bow..')

class Cat:
    def talk(self):
        print('Moew Moew ..')

class Goat:
    def talk(self):
        print('Myaah Myaah ..')

def f1(obj):
    obj.talk()

l=[Duck(),Cat(),Dog(),Goat()]
for obj in l:
    f1(obj)
```

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

Program - 2

In [4]:

```
class Parrot:
    def fly(self):
        print("Fly high in  the sky")

class Sparrow:
    def fly(self):
        print("Fly low in the sky")

s = Sparrow()
p = Parrot()

for obj in [ s, p]:
    obj.fly()
```

```
Fly low in the sky
Fly high in  the sky
```

Program - 3

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

In [5]:

```
class Duck:
    def talk(self):
        print('Quack.. Quack..')

class Dog:
    def bark(self):
        print('Bow Bow..')

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

```
d=Duck()
f1(d)

d=Dog()
f1(d)
```

```
Quack.. Quack..
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_2256/4002753860.py in <module>
     14 
     15 d=Dog()
---> 16 f1(d)

C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_2256/4002753860.py in f1(obj)
      8 
      9 def f1(obj):
---> 10     obj.talk()
     11 
     12 d=Duck()

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
```

In [6]:

```
class Duck:
    def talk(self):
        print('Quack.. Quack..')

class Human:
    def talk(self):
        print('Hello Hi...')

class Dog:
    def bark(self):
        print('Bow Bow..')

def f1(obj):
    if hasattr(obj,'talk'):
        obj.talk()
    elif hasattr(obj,'bark'):
        obj.bark()

d=Duck()
f1(d)

h=Human()
f1(h)

d=Dog()
f1(d)
```

```
Quack.. Quack..
Hello Hi...
Bow Bow..
```

3. Over-riding

```
When Child class method replace(redefine) parent class method in child class is
called method over-riding

Overriding concept applicable for both methods and constructors.
```

Case - 1

In [9]:

```
class Parent:
    def bike(self):
        print("Parent has splendra bike")
    def car(self):
        print("Marutii-800")

class Child(Parent):
    def bike(self): # over-riding bike method of parent class
        print("I have royal enfield Himalyan")

b = Child()
b.car()
b.bike()
```

```
Marutii-800
I have royal enfield Himalyan
```

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

In [10]:

```
class Parent:
    def bike(self):
        print("Parent has splendra bike")
    def car(self):
        print("Marutii-800")

class Child(Parent):
    def bike(self): # Over-riding bike method of parent class
        print("I have royal enfield Himalyan")
        super().bike()

b = Child()
b.car()  # ?
b.bike()  # ?
```

```
Marutii-800
I have royal enfield Himalyan
Parent has splendra bike
```

Case - 2

Constructor Over-riding

In [12]:

```
class A:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return self.name

class B(A):
    # A.__init__ has been over-rided by B.__init__
    def __init__(self, name, age):
        self.age = age
        super().__init__(name)  # super --> A class
        # A.__init__(self, name)


b = B('Pankaj', 20)
print(b)
print(b.age)
```

```
Pankaj
20
```

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

4. Overloading

Introduction

```
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

```
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.
```

Magic Methods

__init__	initlizer
__str__	string representation of an object (Stdout)

__repr__	raw representation of an object (Shell & Stdout)
__len__	return integer value
__add__	+
__sub__	-
__mul__	*
__truediv__	/
__floordiv__	//
__pow__	**
__mod__	%
__lt__	<
__gt__	>
__ge__	>=
__le__	<=
__eq__	==
__ne__	!=
__iadd__	+=
__isub__	-=

Problem - 1

In [19]:

```python
class Person:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return self.name.title()
    def get_name(self):
        return self.name
    def set_name(self, name):
        self.name = name
    def __len__(self):
        return len(self.name)
    def __add__(self, other):
        name = f"{self.name} & {other.name}"
        return Person(name) # instantiate
```

In [20]:

```python
a = Person("Pankaj Yadav")
b = Person("Sachin Yadav")
```

```
print(a, len(a))
print(b, len(b))
```

```
Pankaj Yadav 12
Sachin Yadav 12
```

In [24]:

```
d = a + b # operator overloading
print(d, type(d))
```

```
Pankaj Yadav & Sachin Yadav <class '__main__.Person'>
```

Problem - 2: Vector Addition and substraction Operator overloading

In [25]:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return f"vector({self.x}, {self.y})"


v1 = Vector(4, 5)
v2 = Vector(6, 3)
print(v1)
print(v2)
```

```
vector(4, 5)
vector(6, 3)
```

In [26]:

```
v3 = v1 + v2 # + operator is not overloaded
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_2256/2026195404.py in <module>
----> 1 v3 = v1 + v2 # + operator is not overloaded

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

In [27]:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return f"Vector({self.x}, {self.y})"
    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y) # creating a new object of vector class
    def __sub__(self, other):
        x = self.x - other.x
        y = self.y - other.y
        return Vector(x, y) # creating a new object of vector class
```

In [28]:

```
v1 = Vector(4, 5)
v2 = Vector(6, 3)
print(v1)
print(v2)
```

```
Vector(4, 5)
Vector(6, 3)
```

In [30]:

```
v3 = v1 + v2
print(v3)

v4 = v1 - v2
print(v4)

print(type(v3),type(v4))
```

```
Vector(10, 8)
Vector(-2, 2)
<class '__main__.Vector'> <class '__main__.Vector'>
```

2. Method Overloading

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

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.

i) Normal Method Overloading

In [31]:

```
def area(a):
    "area of Circle"
def area(a, b):
    "area of rectangle"
def area(a, b, c):
    "area of Triangle"
```

In [32]:

```
area(10)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_2256/2905087139.py in <module>
----> 1 area(10)

TypeError: area() missing 2 required positional arguments: 'b' and 'c'
```

But we can logically develope overloading concept by using default argument

In [35]:

```
def area(a, b=None, c=None):
    if b is None and c is None:
        print("Circle")
    elif b is not None and c is None:
        print("Rectangle")
    elif b is not None and c is not None:
        print("Trianle")
    else:
        print("!! Invalid Argument !!")
```

In [37]:

```
area(10)
area(10,20)
area(10,20,30)
```

```
Circle
Rectangle
Trianle
```

ii) Class Method Overloading

In [40]:

```python
class Test:
    def m1(self):
        print('no-arg method')
    def m1(self,a):
        print('one-arg method')
    def m1(self,a,b):
        print('two-arg method')

t=Test()
# t.m1()
#t.m1(10)
t.m1(10,20)
```

```
two-arg method
```

Program with default argument

In [41]:

```python
class Test:
    def sum(self,a=None,b=None,c=None):
        if a!=None and b!= None and c!= None:
            print('The Sum of 3 Numbers:',a+b+c)
        elif a!=None and b!= None:
            print('The Sum of 2 Numbers:',a+b)
        else:
            print('Please provide 2 or 3 arguments')

t=Test()
t.sum(10,20)
t.sum(10,20,30)
t.sum(10)
```

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

Program with Variable Number of Arguments

In [42]:

```python
class Test:
    def sum(self,*a):
        total=0
        for x in a:
            total=total+x
        print('The Sum:',total)


t=Test()
t.sum(10,20)
t.sum(10,20,30)
t.sum(10)
t.sum()
```

```
The Sum: 30
The Sum: 60
The Sum: 10
The Sum: 0
```

3. Constructor Overloading

Constructor overloading is not possible in Python.

If we define multiple constructors then the last constructor will be considered.

In [43]:

```
class Test:
    def __init__(self):
        print('No-Arg Constructor')

    def __init__(self,a):
        print('One-Arg constructor')

    def __init__(self,a,b):
        print('Two-Arg constructor')

#t1=Test()
#t1=Test(10)
t1=Test(10,20)
```

```
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

Program with default argument

In [44]:

```
class Test:
    def __init__(self,a=None,b=None,c=None):
        print('Constructor with 0|1|2|3 number of arguments')

t1=Test()
t2=Test(10)
t3=Test(10,20)
t4=Test(10,20,30)
```

```
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:

In [45]:

```
class Test:
    def __init__(self,*a):
        print('Constructor with variable number of arguments')

t1=Test()
t2=Test(10)
t3=Test(10,20)
t4=Test(10,20,30)
t5=Test(10,20,30,40,50,60)
```

```
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
```

In [ ]:

```

```