In [ ]:
```
#Concept of polymorphism
Poly means many
Morphism means forms
polymorphism == many forms
Entity will be the same but the behaviour of that enitity is different in different conditions.
Example:
    * operator acts as concatenation and addition
    ^ operator acts as repeatition and multiplication
Here the operator is same but the behaviour of both the operators are different in different
scenarios.
Demo:
    "STring1"+"String2" == "STringiString2"
    10**2==30
```

In [ ]:
```
Two Types of Polymorphism
1.Compile Time Polymorphism(overloading)
2.Runtime Polymorphism(Overriding).
```

In [ ]:
```
#Overloading
1.Operator Overloading
2.Method Overloading
3.Constructor Overloading
```

In [ ]:
```
Operator Overloading --> we can use same operator for multiple purposes which is nothing  but
operator overloading.
Python Supports Operator Overloading.
```

In [ ]:
```
Example:
    * operator can be used for both integer addition or string contateneation.
    demo:
        "hello"+"world"==helloworld
        10+20==30
    * operator can be used for both integer multiplication or string repeatation
    demo:
        "hello"*3==("hellohellohello"
        3*3==9
```

In [7]:
```
#Demo program to use + operator for a class object
class Book:
    def __init__(self,pages):
        self.pages=pages
        print("Constructor called")
    def __add__(self,x):
        print("Magic method called")
        return self.pages+x.pages

b1=Book("200")
b2=Book("200")
print(b1+b2)
```

```
Constructor called
Constructor called
Magic method called
200200
```

In [ ]:
```

```

In [ ]:
```
We can overload + operator to work with book object as well for doing this you need to overload
+ operator.
Internally + operator is implemenented vy using a special method named as __add__().
this method is known as Magic method/Special Method/Dunder Method
```

In [9]:
```
#Demo program to use - operator for a class object
class Book:
    def __init__(self,pages):
        self.pages=pages
        print("Constructor called")
    def __sub__(self,x):
        print("Magic method called")
        return self.pages-x.pages

b1=Book(2000)
b2=Book(200)
print(b1-b2)
```

```
Constructor called
Constructor called
1800
```

In [10]:
```
#Demo program to use * operator for a class object
class Book:
    def __init__(self,pages):
        self.pages=pages
        print("Constructor called")
    def __mul__(self,x):
        print("Magic method called")
        return self.pages*x.pages

b1=Book(2000)
b2=Book(200)
print(b1*b2)
```

```
Constructor called
Constructor called
Magic method called
400000
```

In [13]:
```
#Demo program to use // operator for a class object
class Book:
    def __init__(self,pages):
        self.pages=pages
        print("Constructor called")
    def __floordiv__(self,x):
        print("Magic method called")
        return self.pages//x.pages

b1=Book(2000)
b2=Book(200)
print(b1//b2)
```

```
Constructor called
Constructor called
Magic method called
10
```

In [12]:
```
#Demo program to use > and <= operator for a class object
class Student:
    def __init__(self,name,marks):
        self.name=name
        self.marks=marks
        print("Constructor called")
    def __gt__(self,x):
        print("Magic method called")
        return self.marks>x.marks
    def __le__(self,x):
        print("Magic method called")
        return self.marks<x.marks


s1=Student("Om",100)
s2=Student("Shubham",99)
print(s1>s2)
```

```
Constructor called
Constructor called
Magic method called
True
```

In [ ]:
```
#Special/Magic/Dunder Method
There are many internal methods that are already present in our python interpretaor
Whenever we are required we can overload these methods for our use. These methods always
prefix with double underscore and suffix with double underscore.
Example:
    __add__  --> for adding object
    __sub__  --> for subtracting two objects
    __div__  --> for dividing two objects
    __mul__  --> for multiplying two objects
```

In [ ]:
```
#Method Overloading
if two methods having same name but different method signature then that concept is known as
method overloading.
example:
    fun(int a,int b)
    fun(string a)
--> In python method overloading is not possible.
--->> if you are trying to make a method with same name and different arguments then
      python will automatically considers the last one only
```

In [17]:
```
class Test:
    def m1(self):
        print("No argument")
    def m1(self,name):
        print("One argument")
    def m1(self,name1,name2):
        print("Two arguments")
t=Test()
t.m1(10,20)
```

```
Two arguments
```

In [18]:
```
#Demo program to overcame the issue of Method Overloading(Variable length argument)
class Test:
    def m1(self,*a):
        print("Method excecuted")
t=Test()
t.m1()
t.m1(10)
t.m1(10,20)
```

```
Method excecuted
Method excecuted
Method excecuted
```

In [ ]:
```
#Constructor overloading
In python constructor overloading is not possible
if you define multiple constructor then pvm will automatically consider only the
last constructor
```

In [21]:
```
class Test:
    def __init__(self):
        print("No argument constructor")
    def __init__(self,a):
        print("One argument constructor")


    def __init__(self,a,b):
        print("Two argument consstructor")

t=Test(10,20)
```

```
Two argument consstructor
```

In [28]:
```
# We can overcame this problem with the help of variable length argument/Deafult argument
#with default argument
class Test:
    def __init__(self,a=None,b=None,c=None,d=None):
        print("constructor called")

t= Test(10,20,30,40)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [28], in <cell line: 7>()
      4         print("constructor called")
      5
----> 7 t= Test(10,20,30,40,50)

TypeError: __init__() takes from 1 to 5 positional arguments but 6 were given
```

In [33]:
```
#Variable length argument
class test:
    def __init__(self,*a):
        print("Constarucutor called")

t=test(10,20,30,40,60,70)
```

```
Constarucutor called
```

Overriding

In [ ]:
```
Overriding(Run time Polymorphism) --> whatever  memebers are present in the parent class
are by default available to the child class This concept is known as Inheritance.
If the child class is not satisfied with the parent class functionality then child class is allowed to
implement or redefine its own functionality based on certain requirement , this concept is knowns as
overriding.

Runtime polymorphism (Overriding ) can be done in :
    method overriding
    constructor overriding
```

In [ ]:
```
Method Overriding : If two methods are having same name and same number of argument then that
    concept is known as method overriding

Example :
    fun(int a, int b)  --->> Parent class
    fun(int a, int b)  --> class class

Note: In case of overloading the function name is same whereas arguments are different.
    In case of overriding the function name as well as argument both are same.
```

In [35]:
```
#Demo program for method overriding
class Parent:
    def property(self):
        print("Gold+flat+Iphone+Car")

    def study(self):
        print("Btech")
class child(Parent):
    def study(self):
        print("BSC")
c=Parent()
c.property()
c.study()

Gold+flat+Iphone+Car
Btech
```

In [ ]:
```

```

In [ ]:
```
#From overriding the method of child class we can also call parent class method with the help
of super() method
super() --> It is a method that is used to access the parent class method , variables and constructor
```

In [36]:
```
#Demo program for method overriding
class Parent:
    def property(self):
        print("Gold+flat+Iphone+Car")

    def study(self):
        print("Btech")
class child(Parent):
    def study(self):
        super().study()
        print("BSC")
c=child()
c.property()
c.study()

Gold+flat+Iphone+Car
Btech
BSC
```

In [37]:
```
#Constructor Overriding
class parent:
    def __init__(self):
        print("Parent constructir")
class child(parent):
    def __init__(self):
        print("child constructor")
c=child()

child constructor
```

In [38]:
```
class parent:
    def __init__(self):
        print("Parent constructir")
class child(parent):
    pass
c=child()

Parent constructir
```

In [40]:
```
class parent:
    a=20
    def __init__(self):
        print("Parent constructir")
class GrandFather(parent):
    a=30
    def __init__(self):
        print("GrandFather")
class child(GrandFather):
    a=10
    def __init__(self):
        super().__init__()
        print("child constructor")
        print(super().a)
c=child()

GrandFather
child constructor
30
```

In [56]:
```
class cal:
    def add(self,a,b):
        return a+b

#reference_variable=classname(10,20)
x=cal()
x.add(10,20)
print(type(x))
```

```
<class '__main__.cal'>
```

In [57]:
```
x=cal()
x.add(10,20)
print(type(x))
y=list()
y.append(3)
print(type(y))

<class '__main__.cal'>
<class 'list'>
```

In [ ]:
```
#Note if we are not importing module ,class and variable then  the value of
__name__  will always be equal to __main__
```

In [59]:
```
class Test:
    def __init__(self,x=100):
        self.x=x
t=Test(200)
t.y=200
t.z=500
print(t.z*len(t.__dict__))
```

```
1500
```

In [ ]:
```
#We can declare instance variable outside the class with the help of object reference
```