Python Programming

RGM College of Engineering & Technology (Autonomous)

Department of Computer Science & Engineering

AY:2021-2022

PYTHON'S OBJECT ORIENTED PROGRAMMING - 10

Guido Van Rossum

Agenda:

1. Introduction to Polymorphism
2. Overloading
3. Overriding
4. Summary on Polymorphism

1. Introduction to Polymorphism

- Poly means Many
- Morphs means Forms.
- Polymorphism means 'Many Forms'

➔ **One Name but multiple forms is the concept of polymorphism.**

Eg 1:

- 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.

Eg 2:

'+' operator acts as concatenation and arithmetic addition Operator.

```
10+20          # Arithmetic Addition          30

'rgm' + 'cet'     # String Concatenation      'rgmcet'
```

Eg 3:

'*' operator acts as multiplication and repetition operator.

```
10 * 2      # Multiplication operator
```

```
20
```

```
'RGMCET' * 3  #Repetetion Operator
```

```
'RGMCETRGMCETRGMCET'
```

Note:

- Example 2 and Example 3 are called as Operator Overloading.

Eg 4:

- We can declare a method with same name in both parent and child classes but with different implementations (i.e., Method Overriding).

I.

```
class P:

    def marry(self):
        print('Appalamma')

class C(P):pass

c = C()
c.marry()
```

```
Appalamma
```

II.

```
class P:

    def marry(self):
        print('Appalamma')

class C(P):

    def marry(self):
        print('Katrina Kaif')

c = C()
c.marry()
```

```
Katrina Kaif
```

Related to polymorphism the following 3 topics are important,

1. **Overloading**
    1. **Operator Overloading**
    2. **Method Overloading**
    3. **Constructor Overloading**
2. **Overriding**
    1. **Method overriding**
    2. **constructor overriding**
3. **Pythonic Behaviour:**
    1. **Duck Typing**
    2. **Easier to Ask Forgiveness than Permission(EAFP)**
    3. **Monkey Patching**

2. Overloading

Related to Overloading, we have to cover the following three types of overloading concepts:

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

1. Operator Overloading:

- We can use same operator for multiple purposes , which is nothing but operator overloading.
- Python supports operator overloading.
- Java won't provide support for operator overloading.

Eg 1: '+' operator can be used for Arithmetic addition and String concatenation.

```
print(20 + 55)     # Arithmetic Operator
```

```
75
```

```
print('RGM' + 'CET')  # String concatenation
```

```
RGMCET
```

Eg 2: '*' operator can be used for multiplication and string repetition purposes.

```
10 * 20
```

```
200
```

```
'RGM' * 3
```

```
'RGMRGMRGM'
```

- So far, we used '+' operator on standard objects like integers and strings. Now we will check that, how '+'operator works on our own class objects. See the following example.

Demo program to use + operator for our class objects:

```
class Book:

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

b1 = Book(100)
b2 = Book(200)
print(b1 + b2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-31638e99fedb> in <module>
      6 b1 = Book(100)
      7 b2 = Book(200)
----> 8 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:

I.

```
class Book:

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

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

b1 = Book(100)
b2 = Book(200)
print('The total number of Pages : ',b1 + b2)
```

```
The total number of Pages :  300
```

II.

```
class Book:

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

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

b1 = Book(100)
b2 = Book(200)
b3 = Book(500)
print('The total number of Pages : ',b1 + b2)
print('The total number of Pages : ',b1 + b3)
print('The total number of Pages : ',b2 + b3)
print('The total number of Pages : ',b1 + b2 + b3)  # TypeError: b1 + b2 result is 'in
```

teger

```
The total number of Pages :  300
The total number of Pages :  600
The total number of Pages :  700

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-0f7b43e52547> in <module>
     13 print('The total number of Pages : ',b1 + b3)
     14 print('The total number of Pages : ',b2 + b3)
---> 15 print('The total number of Pages : ',b1 + b2 + b3)

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

Operator and Corresponding Magic Methods are listed below:

Operator	Corresponding Magic Method
+	__add__(self,other)
-	__sub__(self,other)
*	__mul__(self,other)
/	__div__(self,other)
//	__floordiv__(self,other)
%	__mod__(self,other)
**	__pow__(self,other)
+=	__iadd__(self,other)
-=	__isub__(self,other)
*=	__imul__(self,other)
/=	__idiv__(self,other)

Operator	Corresponding Magic Method
//=	__ifloordiv__(self,other)
%=	__imod__(self,other)
**=	__ipow__(self,other)
<	__lt__(self,other)
<=	__le__(self,other)
>	__gt__(self,other)
>=	__ge__(self,other)
==	__eq__(self,other)
!=	__ne__(self,other)

Note:

- If you want to explore more information about these magic methods, go through with Python documentation available at www.python.org (http://www.python.org)

Demo Program on Overloading > and <= operators for Student class objects:

I.

```
class Student:

    def __init__(self,name,marks):
        self.name=name
        self.marks=marks

s1 = Student('Karthi',88)
s2 = Student('Sahasra',89)
print(s1>s2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-7d2fc1f2c978> in <module>
      7 s1 = Student('Karthi',88)
      8 s2 = Student('Sahasra',89)
----> 9 print(s1>s2)

TypeError: '>' not supported between instances of 'Student' and 'Student'
```

Now, we will make use of magic method of '>' operator in the below implementation.

II.

```
class Student:

    def __init__(self,name,marks):
        self.name=name
        self.marks=marks

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

s1 = Student('Karthi',88)
s2 = Student('Sahasra',89)
print(s1>s2)   #False
```

```
False
```

III.

```
class Student:

    def __init__(self,name,marks):
        self.name=name
        self.marks=marks

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

s1 = Student('Karthi',88)
s2 = Student('Sahasra',89)
s3 = Student('XYZ',35)
print(s1>s3)     # True
```

```
True
```

See the below code,

IV.

```
class Student:

    def __init__(self,name,marks):
        self.name=name
        self.marks=marks

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

s4 = Student('Karthi',88)
s5 = Student('Sahasra',89)

print(s4 < s5)
```

```
True
```

Note:

- Whenever you are implementing the magic method of greater than ('>') in your program, you need not implement the magic method of less than('<'). PVM automatically reverse that functionality.

V.

```
class Student:

    def __init__(self,name,marks):
        self.name=name
        self.marks=marks

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

s4 = Student('Karthi',88)
s5 = Student('Sahasra',89)

print(s4 <= s5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-c63cdbdbd918> in <module>
     11 s5 = Student('Sahasra',89)
     12
---> 13 print(s4 <= s5)

TypeError: '<=' not supported between instances of 'Student' and 'Student'
```

Note: For <=, you need to implement the corresponding magic method.

VI.

```
class Student:

    def __init__(self,name,marks):
        self.name=name
        self.marks=marks

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

    def __le__(self,other):
        return self.marks<=other.marks

s4 = Student('Karthi',88)
s5 = Student('Sahasra',89)
print(s4 <= s5)  # True
```

```
True
```

VII.

```
class Student:

    def __init__(self,name,marks):
        self.name=name
        self.marks=marks

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

    def __le__(self,other):
        return self.marks<=other.marks

s4 = Student('Karthi',88)
s5 = Student('Sahasra',89)
print(s4 >= s5)  # False
```

```
False
```

Note:

- Whenever you are implementing the magic method of greater than or equals to ('>=') in your program, y need not to implement the magic method of less than or equal to('<=').

Demo Program to overload multiplication operator to work on Employee objects.

I.
```
class Employee:

    def __init__(self,name,salaryperday):
        self.name = name
        self.salaryperday = salaryperday


class TimeSheet:

    def __init__(self,name,workingdays):
        self.name = name
        self.workingdays = workingdays


e = Employee('Karthi',500)
t = TimeSheet('Karthi',25)
print('This Month Salary:',e * t)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-39283e959afd> in <module>
     15 e = Employee('Karthi',500)
     16 t = TimeSheet('Karthi',25)
---> 17 print('This Month Salary:',e * t)

TypeError: unsupported operand type(s) for *: 'Employee' and 'TimeSheet'
```

- We got an error, because we didn't implemented magic method for multiplication operator ('*').

Key Point:

- PVM will always call magic method from first argument class. i.e., Here, from Employee class.

II.

```
class Employee:

    def __init__(self,name,salaryPerDay):
        self.name = name
        self.salaryPerDay = salaryPerDay

    def __mul__(self,other):
        return self.salaryPerDay * other.workingDays

class TimeSheet:

    def __init__(self,name,workingDays):
        self.name = name
        self.workingDays = workingDays


e = Employee('Karthi',500)
t = TimeSheet('Karthi',25)
print('This Month Salary:',e * t)
```

```
This Month Salary: 12500
```

III.

```
class Employee:

    def __init__(self,name,salaryPerDay):
        self.name = name
        self.salaryPerDay = salaryPerDay

    def __mul__(self,other):
        return self.salaryPerDay * other.workingDays

class TimeSheet:

    def __init__(self,name,workingDays):
        self.name = name
        self.workingDays = workingDays


e = Employee('Karthi',500)
t = TimeSheet('Karthi',25)
print('This Month Salary:',t * e)     # Order is important
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-666ddfd911e3> in <module>
     17 e = Employee('Karthi',500)
     18 t = TimeSheet('Karthi',25)
---> 19 print('This Month Salary:',t * e)

TypeError: unsupported operand type(s) for *: 'TimeSheet' and 'Employee'
```

IV.

```
class Employee:

    def __init__(self,name,salaryPerDay):
        self.name = name
        self.salaryPerDay = salaryPerDay

    def __mul__(self,other):
        return self.salaryPerDay * other.workingDays

class TimeSheet:

    def __init__(self,name,workingDays):
        self.name = name
        self.workingDays = workingDays

    def __mul__(self,other):
        return self.workingDays * other.salaryPerDay


e = Employee('Karthi',500)
t = TimeSheet('Karthi',25)
print('This Month Salary:',t * e)    # Now works correctly
```

```
This Month Salary: 12500
```

Note:

- Take special care on in which class magic method is implementing. Observe that what is the first argument and that corresponding class must contains magic method.

Importance of __str__() method:

- To cover next example related to Operator Overloading, there is some important concept must be required. Let us discuss about that required concept and then we will go for operator overloading next example.

__str__ () Method:

- Whenever we are trying to print any object reference, internally __str__() method will be called.
- The default implementation of this method returns the string in the following format:

```
<__main__.Student object at 0x000000000280D748>
```

- To provide meaningful string representation for our object, we have to override __str__ () method in our class.

I.

```
class Student:

    def __init__(self,name,rollno,marks):
        self.name = name
        self.rollno = rollno
        self.marks = marks

s1 = Student('Karthi',101,98)
s2 = Student('Sahasra',102,99)

print(s1)
print(s2)
```

```
<__main__.Student object at 0x00000210D153CDF0>
<__main__.Student object at 0x00000210D153CFA0>
```

II.

```
class Student:

    def __init__(self,name,rollno,marks):
        self.name = name
        self.rollno = rollno
        self.marks = marks

    def __str__(self):
        return self.name

s1 = Student('Karthi',101,98)
s2 = Student('Sahasra',102,99)

print(s1)
print(s2)
```

```
Karthi
Sahasra
```

Another form of implementation of __str__() function:

III.

```
class Student:

    def __init__(self,name,rollno,marks):
        self.name = name
        self.rollno = rollno
        self.marks = marks

    def __str__(self):
        return 'Name :{}\t Roll No: {}\tMarks : {}'.format(self.name,self.rollno,self.marks

s1 = Student('Karthi',101,98)
s2 = Student('Sahasra',102,99)

print(s1)
print(s2)
```

```
Name :Karthi     Roll No: 101    Marks : 98
Name :Sahasra    Roll No: 102    Marks : 99
```

Demo Program on Overloading of + operator for Nesting Requirements.

I.

```
class Book:

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


b1 = Book(100)
b2 = Book(200)
b3 = Book(500)
print(b1 + b2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-8a5334b2246c> in <module>
      8 b2 = Book(200)
      9 b3 = Book(500)
---> 10 print(b1 + b2)

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

After implementing corresponding magic method,

II.

```
class Book:

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

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


b1 = Book(100)
b2 = Book(200)
b3 = Book(500)
print(b1 + b2)
print(b2 + b3)
print(b1 + b3)
```

```
300
700
600
```

Now, our requirement is, print(b1 + b2 + b3)

III.

```
class Book:

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

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

```
b1 = Book(100)
b2 = Book(200)
b3 = Book(500)
print(b1 + b2)
print(b2 + b3)
print(b1 + b3)
print(b1 + b2 + b3)
```

```
300
700
600

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-17-0010936a3ca1> in <module>
     14 print(b2 + b3)
     15 print(b1 + b3)
---> 16 print(b1 + b2 + b3)

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

Let us see, what happens, if we write the code in below manner,

IV.

```
class Book:

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

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

```
b1 = Book(100)
b2 = Book(200)
b3 = Book(500)
print(b1 + b2)
print(b2 + b3)
print(b1 + b3)
print(b1 + b2 + b3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-55799aaee07c> in <module>
     11 b2 = Book(200)
     12 b3 = Book(500)
---> 13 print(b1 + b2)
     14 print(b2 + b3)
     15 print(b1 + b3)

TypeError: __add__() missing 1 required positional argument: 'another'
```

Above code won't give required result. See the below code.

V.

```
class Book:

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

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

```
b1 = Book(100)
b2 = Book(200)
b3 = Book(500)
print(b1 + b2)
print(b2 + b3)
print(b1 + b3)
print(b1 + b2 + b3)
```

```
300
700
600

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-19-0010936a3ca1> in <module>
     14 print(b2 + b3)
     15 print(b1 + b3)
---> 16 print(b1 + b2 + b3)

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

VI.

```
class Book:

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

    def __add__(self,other):
        return Book(self.pages + other.pages)


b1 = Book(100)
b2 = Book(200)
b3 = Book(500)
print(b1 + b2)
print(b2 + b3)
print(b1 + b3)
print(b1 + b2 + b3)
```

```
<__main__.Book object at 0x00000210D15DEEB0>
<__main__.Book object at 0x00000210D153CDF0>
<__main__.Book object at 0x00000210D15DEEB0>
<__main__.Book object at 0x00000210D15DEAF0>
```

VII.

```
class Book:

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

    def __add__(self,other):        # return type is Book object
        return Book(self.pages + other.pages)


b1 = Book(100)
b2 = Book(200)
b3 = Book(500)

print(b1 + b2 + b3)
```

```
<__main__.Book object at 0x00000210D153C430>
```

VIII.

```
class Book:

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

    def __add__(self,other):        # return type is Book object
        return Book(self.pages + other.pages)

    def __str__(self):
        return 'The Total Number of Pages :{}'.format(self.pages)


b1 = Book(100)
b2 = Book(200)
b3 = Book(500)
b4 = Book(600)
print(b1 + b2)
print(b2 + b3)
print(b1 + b3)
print(b1 + b2 + b3)
print(b1 + b2 + b3 + b4)
```

```
The Total Number of Pages :300
The Total Number of Pages :700
The Total Number of Pages :600
The Total Number of Pages :800
The Total Number of Pages :1400
```

Suppose, if I want to apply multiplication operation on our Book objects, the see the below code.

IX.

```
class Book:

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

    def __add__(self,other):      # return type is Book object
        return Book(self.pages + other.pages)

    def __str__(self):
        return 'The Total Number of Pages :{}'.format(self.pages)

    def __mul__(self,other):
        return Book(self.pages * other.pages)


b1 = Book(100)
b2 = Book(200)
b3 = Book(500)
b4 = Book(600)
print(b1 + b2)
print(b2 + b3)
print(b1 + b3)
print(b1 + b2 + b3)
print(b1 + b2 + b3 + b4)
print(b1 + b2 * b3 + b4)   # Opertaor Precedence follows.
```

```
The Total Number of Pages :300
The Total Number of Pages :700
The Total Number of Pages :600
The Total Number of Pages :800
The Total Number of Pages :1400
The Total Number of Pages :100700
```

X.

```
class Book:

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

    def __add__(self,other):      # return type is Book object
        print('add method executed...')
        return Book(self.pages + other.pages)

    def __str__(self):
        return 'The Total Number of Pages :{}'.format(self.pages)

    def __mul__(self,other):
        print('mul method executed..')
        return Book(self.pages * other.pages)


b1 = Book(100)
b2 = Book(200)
b3 = Book(500)
b4 = Book(600)

print(b1 + b2 * b3 + b4)   # Operator Precedence follows.
```

```
mul method executed..
add method executed...
add method executed...
The Total Number of Pages :100700
```

XI.

```
class Book:

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

    def __add__(self,other):      # return type is Book object
        print('add method executed...')
        return Book(self.pages + other.pages)

    def __str__(self):
        return 'The Total Number of Pages :{}'.format(self.pages)

    def __mul__(self,other):
        print('mul method executed..')
        return Book(self.pages * other.pages)


b1 = Book(100)
b2 = Book(200)
b3 = Book(500)
b4 = Book(600)

print(b1 + b2 * b3 + b4)   # Operator Precedence follows.
print(b1 * b2 + b3 * b4)
```

```
mul method executed..
add method executed...
add method executed...
The Total Number of Pages :100700
mul method executed..
mul method executed..
add method executed...
The Total Number of Pages :320000
```

2. Method Overloading

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

Eg:

- sqrt(int)
- sqrt(float)

- Java provides support for method overloading.
- But in Python, we cannot declare type explicitly. Based on provided value type will be considered automatically (Dynamically Typed).
- As type concept is not applicable, method overloading concept is not applicable in python.

Note:

- If we are trying to declare multiple methods with same name and different number of arguments then Python will always consider only last method. See the below code for clarification.

Demo Program:

I.

```
class Test:

    def m1(self):
        print('no-arg method')

    def m1(self,x):
        print('one-arg method')

    def m1(self,x,y):
        print('two-arg method')

t=Test()
t.m1()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-e914f311b3a7> in <module>
     11
     12 t=Test()
---> 13 t.m1()

TypeError: m1() missing 2 required positional arguments: 'x' and 'y'
```

II.

```
class Test:

    def m1(self):
        print('no-arg method')

    def m1(self,x):
        print('one-arg method')

    def m1(self,x,y):
        print('two-arg method')
```

```
t=Test()
t.m1(10)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-55a2e45c4333> in <module>
     11
     12 t=Test()
---> 13 t.m1(10)

TypeError: m1() missing 1 required positional argument: 'y'
```

III.

```
class Test:

    def m1(self):
        print('no-arg method')

    def m1(self,x):
        print('one-arg method')

    def m1(self,x,y):
        print('two-arg method')

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

```
two-arg method
```

Why Python won't support Method Overloading?

- Java provides support for method overloading.
- But in Python, we cannot declare type explicitly. Based on provided value type will be considered automatically (Dynamically Typed).
- As type concept is not applicable, method overloading concept is not applicable in python.

In Python, without using multiple methods with the same name with different arguments (just like in Java), we can perform the same thing by make using of single method, which will meet all the requirements. See the below example

I.

```
class Test:

    def m1(self,x):
        print('{} - Argument Method'.format(type(x)))

t = Test()
t.m1(11)
t.m1(11.2)
t.m1('Karthi')
```

```
<class 'int'> - Argument Method
<class 'float'> - Argument Method
<class 'str'> - Argument Method
```

Another way of implementation:

II.

```
class Test:

    def m1(self,x):
        print('{} - Argument Method'.format(x.__class__.__name__))

t = Test()
t.m1(11)
t.m1(11.2)
t.m1('Karthi')
```

```
int - Argument Method
float - Argument Method
str - Argument Method
```

Note:

- Whatever overloading methods are doing in other programming languages, python already has that fact which is shown in the above example program.
- Method Overloading concept is not required in Python programming. Because, whatever is doing by the overloaded methods, python by default is doing the same thing.

How to define a method with variable number of arguments?

- In Python, if a method with variable number of arguments required then, there are two ways available.

  1. With default arguments

  2. With variable number of arguments

Demo Program 1 on using with Default Arguments:

I.
```
class Test:

    def m1(self,a=None,b=None,c=None):
        if a is not None and b is not None and c is not None:
            print('Three Argument Method')
        elif a!=None and b!= None:
            print('Two Argument Method')
        elif a is not None:
            print('One Argument Method')
        else:
            print('No Argument Method')
```

```
t=Test()

t.m1()
t.m1(10)
t.m1(10,20)
t.m1(10,20,30)
```

```
No Argument Method
One Argument Method
Two Argument Method
Three Argument Method
```

II.

```
class Test:

    def m1(self,a=None,b=None,c=None):
        if a is not None and b is not None and c is not None:
            print('Three Argument Method')
        elif a!=None and b!= None:
            print('Two Argument Method')
        elif a is not None:
            print('One Argument Method')
        else:
            print('No Argument Method')

t=Test()

t.m1()
t.m1(10)
t.m1(10,20)
t.m1(10,20,30)
t.m1(10,20,30,40)
```

```
No Argument Method
One Argument Method
Two Argument Method
Three Argument Method

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-771a602dced2> in <module>
     17 t.m1(10,20)
     18 t.m1(10,20,30)
---> 19 t.m1(10,20,30,40)

TypeError: m1() takes from 1 to 4 positional arguments but 5 were given
```

Demo Program 2:

I.

```
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
```

- Suppose, if we want to pass any number of arguments, we can declare that declaration using variable length arguments.

Demo Program 1 on with Variable Number of Arguments:

```
class Test:

    def m1(self,*args):
        print('Variable Length Arguments')

t = Test()
t.m1()
t.m1(10)
t.m1(10,20,30)
t.m1(10,20,30,40,45.50)
t.m1(10,20,30,40,45,50)
```

```
Variable Length Arguments
Variable Length Arguments
Variable Length Arguments
Variable Length Arguments
Variable Length Arguments
```

Demo Program 2 on with Variable Number of Arguments:

(Perform sum of multiple values)

I.

```
class Test:
    def sum(self,*a): # here, a is internally rep. as a tuple
        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 are trying to declare multiple constructors then PVM will always consider last constructor.

Demo Program:

I.
```
class Test:

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

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

    def __init__(self,x,y):
            print('Two-Arg constructor')

t1=Test()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-9dce27397364> in <module>
     10                 print('Two-Arg constructor')
     11
---> 12 t1=Test()

TypeError: __init__() missing 2 required positional arguments: 'x' and 'y'
```

II.

```
class Test:

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

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

    def __init__(self,x,y):
            print('Two-Arg constructor')

t1=Test(10)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-551ab770a73b> in <module>
     10                 print('Two-Arg constructor')
     11
---> 12 t1=Test(10)

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

III.

```
class Test:

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

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

    def __init__(self,x,y):
                print('Two-Arg constructor')

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.

How to define a constructor with default arguments?

Demo Program:

I.

```
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
```

II.

```
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)
t5=Test(10,20,30,40)
```

```
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
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-749b11e1c4d3> in <module>
      7 t3=Test(10,20)
      8 t4=Test(10,20,30)
----> 9 t5=Test(10,20,30,40)

TypeError: __init__() takes from 1 to 4 positional arguments but 5 were give
n
```

How to define a constructor with variable number of arguments?

Demo Program:

I.

```
class Test:
    def __init__(self,*args):
        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
```

Conclusions:

In Overloading, up to this we discussed 3 things:

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

S.no	Concept Name	Python	Java
1	Operator Overloading	✓	**X**
2	Method Overloading	**X**	✓
3	Constructor Overloading	**X**	✓

3. Method overriding and Constructor Overriding

- What ever members available in the parent class are by default 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:

I.

```
class Parent:

    def property(self):
        print('Land + Cash + Gold + Power')

    def marry(self):
        print('Appalamma')

class Child(Parent):
    pass

c = Child()
c.property()
c.marry()
```

```
Land + Cash + Gold + Power
Appalamma
```

- Assume that, from the above example case, child is satisfied with property() method as it is with parent class implementation. But not satisfied with marry() method implementation.
- If you are not satisfied with parent method implementation, happily you are allowed to redefine in the child class based on it's requirement.
- Now, in the below code we are trying to redefine marry() method in child class.

II.

```
class Parent:

    def property(self):
        print('Land + Cash + Gold + Power')

    def marry(self):                    # Overriden Method
        print('Appalamma')

class Child(Parent):

    def marry(self):
        print('Katrina Kaif')       # Overriding Method

c = Child()
c.property()
c.marry()
```

```
Land + Cash + Gold + Power
Katrina Kaif
```

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

III.

```
class Parent:

    def property(self):
        print('Land + Cash + Gold + Power')

    def marry(self):                    # Overriden Method
        print('Appalamma')

class Child(Parent):

    def marry(self):
        super().marry()
        print('Katrina Kaif')       # Overriding Method

c = Child()
c.property()
c.marry()
```

```
Land + Cash + Gold + Power
Appalamma
Katrina Kaif
```

Demo Program for Constructor overriding:

I.

```
class Parent():

    def __init__(self):
        print('Parent Constructor')

class Child(Parent):
    pass

c = Child()  #Child class doesn't contain constructor
```

```
Parent Constructor
```

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

II.

```
class Parent():

    def __init__(self):
        print('Parent Constructor')

class Child(Parent):

    def __init__(self):
        print('Child Constructor')

c = Child()
```

```
Child Constructor
```

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

III.

```
class Parent():

    def __init__(self):
        print('Parent Constructor')

class Child(Parent):
    def __init__(self):
        super().__init__()
        print('Child Constructor')

c = Child()
```

```
Parent Constructor
Child Constructor
```

Overriding Demo Program:

I.

```
class Person:
    def __init__(self,name,age,weight,height):
        self.name = name
        self.age  = age
        self.weight=weight
        self.height=height
    def display(self):
        print('Name :',self.name)
        print('Age :',self.age)
        print('Weight :',self.weight)
        print('Height :',self.Height)

class Employee(Person):
    def __init__(self,name,age,weight,height,eno,esal):
        self.name = name
        self.age  = age
        self.weight=weight
        self.height=height
        self.eno =eno
        self.esal=esal
    def display(self):
        print('Name :',self.name)
        print('Age :',self.age)
        print('Height :',self.height)
        print('Weight :',self.weight)
        print('Employee Number :',self.eno)
        print('Employee Salary :',self.esal)

e = Employee('Karthi',6,4.2,20,872424,50000)
e.display()
```

```
Name : Karthi
Age : 6
Height : 20
Weight : 4.2
Employee Number : 872424
Employee Salary : 50000
```

We can simplify the above code. See the below code:

II.

```
class Person:
    def __init__(self,name,age,weight,height):
        self.name = name
        self.age  = age
        self.weight=weight
        self.height=height
    def display(self):
        print('Name :',self.name)
        print('Age :',self.age)
        print('Weight :',self.weight)
        print('Height :',self.height)

class Employee(Person):
    def __init__(self,name,age,weight,height,eno,esal):
       # self.name = name
       # self.age  = age
       # self.weight=weight
       # self.height=height
        super().__init__(name,age,weight,height)
        self.eno =eno
        self.esal=esal
    def display(self):
       # print('Name :',self.name)
       # print('Age :',self.age)
       # print('Height :',self.height)
       # print('Weight :',self.weight)
        super().display()
        print('Employee Number :',self.eno)
        print('Employee Salary :',self.esal)

e = Employee('Karthi',6,4.2,20,872424,50000)
e.display()
```

```
Name : Karthi
Age : 6
Weight : 4.2
Height : 20
Employee Number : 872424
Employee Salary : 50000
```

III.

```
class Person:
    def __init__(self,name,age,weight,height):
        self.name = name
        self.age  = age
        self.weight=weight
        self.height=height
    def display(self):
        print('Name :',self.name)
        print('Age :',self.age)
        print('Weight :',self.weight)
        print('Height :',self.height)

class Employee(Person):
    def __init__(self,name,age,weight,height,eno,esal):
        super().__init__(name,age,weight,height)
        self.eno =eno
        self.esal=esal
    def display(self):
        super().display()
        print('Employee Number :',self.eno)
        print('Employee Salary :',self.esal)

e = Employee('Karthi',6,4.2,20,872424,50000)
e.display()
```

```
Name : Karthi
Age : 6
Weight : 4.2
Height : 20
Employee Number : 872424
Employee Salary : 50000
```

4. Summary on Polymorphism

1. Polymorphism means, one name but multiple behaviours.

2. Overloading is the best example for Polymorphism.

3. Overloading concept classified into the following categories:

```
1. Operator Overloading

2. Method Overloading        - Not supported in Python

3. Constructor Overloading   - Not supported in Python
```

4. Overriding is the another example for Polymorphism.

5. Overriding concept classified into the following categories:

    1. Method Overloading

    2. Constructor Overloading

The biggest advantage of the Polymorphism is:

Providing more flexibility to the programmer.

Any question?

If you try to practice programs yourself, then you will learn many things automatically

Spend few minutes and then enjoy the study

Thank You