

SELF, INSTANCE AND STATIC VARIABLE IN PYTHON

SELF VARIABLE IN PYTHON

Self represents the instance of the class. By using the “self” keyword we can access the attributes and methods of the class in python. It binds the attributes with the given arguments. Python uses the self-parameter to refer to instance attributes and methods of the class. The self-variable in Python can also be used to access a variable field within the class definition.

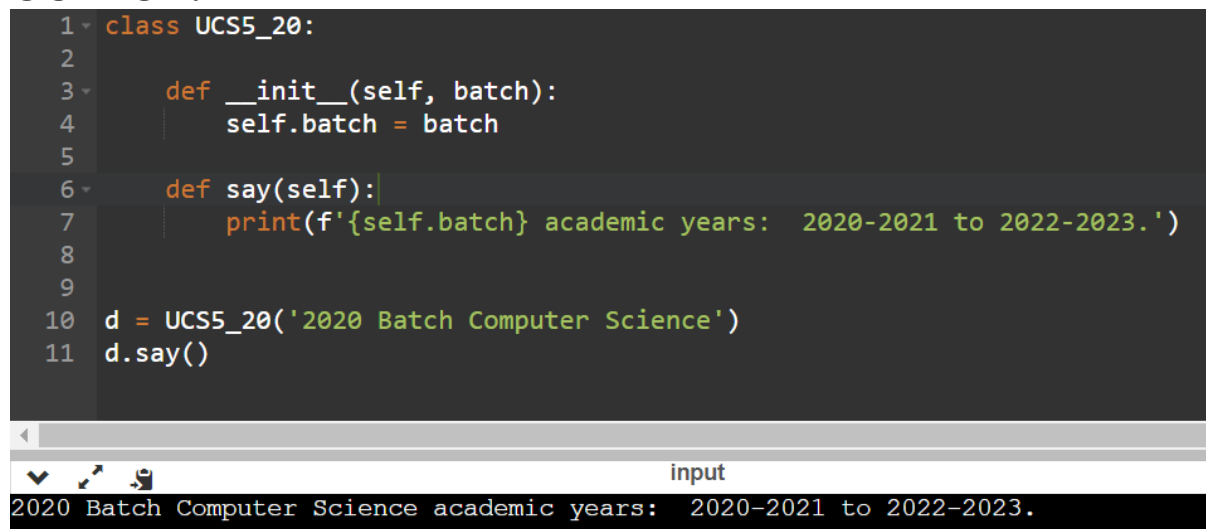
Self is the first argument to be passed in Constructor and Instance Method. **Self** must be provided as a First parameter to the Instance method and constructor. **Self** is a convention and not a Python keyword. **Self** is parameter in Instance Method and user can use another parameter name in place of it.

EXAMPLE 1:

```
class UCS5_20:
    def __init__(self, batch):
        self.batch = batch

    def say(self):
        print(f'{self.batch} academic years: 2020-2021 to 2022-2023.')
d = UCS5_20('2020 Batch Computer Science')
d.say()
```

OUTPUT:



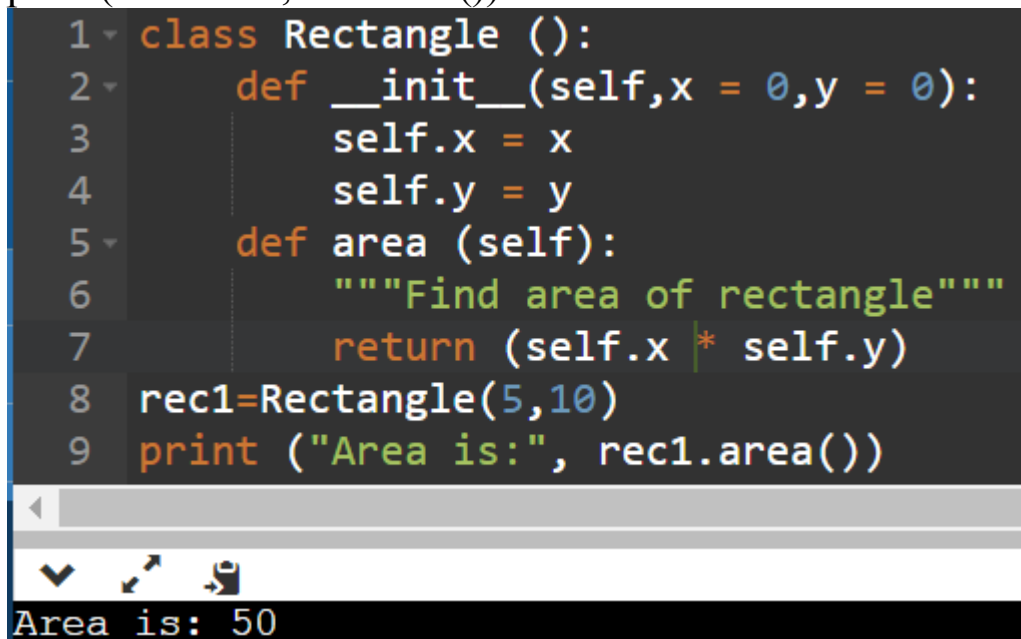
```
1 class UCS5_20:
2
3     def __init__(self, batch):
4         self.batch = batch
5
6     def say(self):
7         print(f'{self.batch} academic years: 2020-2021 to 2022-2023.')
8
9
10 d = UCS5_20('2020 Batch Computer Science')
11 d.say()
```

input

2020 Batch Computer Science academic years: 2020-2021 to 2022-2023.

EXAMPLE 2:

```
class Rectangle ():
    def __init__(self,x = 0,y = 0):
        self.x = x
        self.y = y
    def area (self):
        """Find area of rectangle"""
        return (self.x * self.y)
rec1=Rectangle(5,10)
print ("Area is:", rec1.area())
```

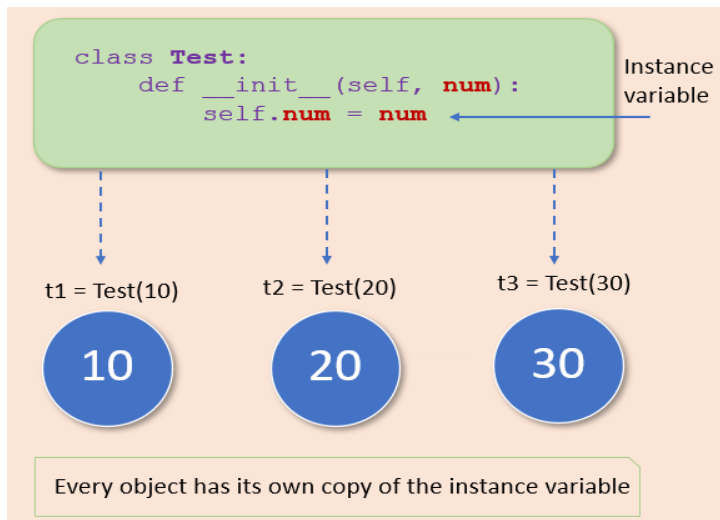


```
1 class Rectangle ():
2     def __init__(self,x = 0,y = 0):
3         self.x = x
4         self.y = y
5     def area (self):
6         """Find area of rectangle"""
7         return (self.x * self.y)
8 rec1=Rectangle(5,10)
9 print ("Area is:", rec1.area())
```

Area is: 50

INSTANCE VARIABLE IN PYTHON

- **Instance variables** in a class are called fields or attributes of an object.
- If the value of a variable varies from object to object, then such variables are called instance variables.
- For every object, a separate copy of the instance variable will be created.
- Instance variables are not shared by objects. Every object has its own copy of the instance attribute. This means that for each object of a class, the instance variable value is different.
- Instance variables are used within the instance method.
- We can access the instance variable using the object and dot (.) operator.
- Instance variables are declared inside a method using the **self** keyword.



EXAMPLE:

class Student:

```

# constructor
def __init__(self, name, age):
    # Instance variable
    self.name = name
    self.age = age
# create first object
s1 = Student("Ajita", 12)
# access instance variable
print('Object 1')
print('Name:', s1.name)
print('Age:', s1.age)
# create second object
s2 = Student("Jancy", 10)
# access instance variable
print('Object 2')
print('Name:', s2.name)
print('Age:', s2.age)
  
```

OUTPUT:

```

Object 1
Name: Ajita
Age: 12
Object 2
Name: Jancy
Age: 10
  
```

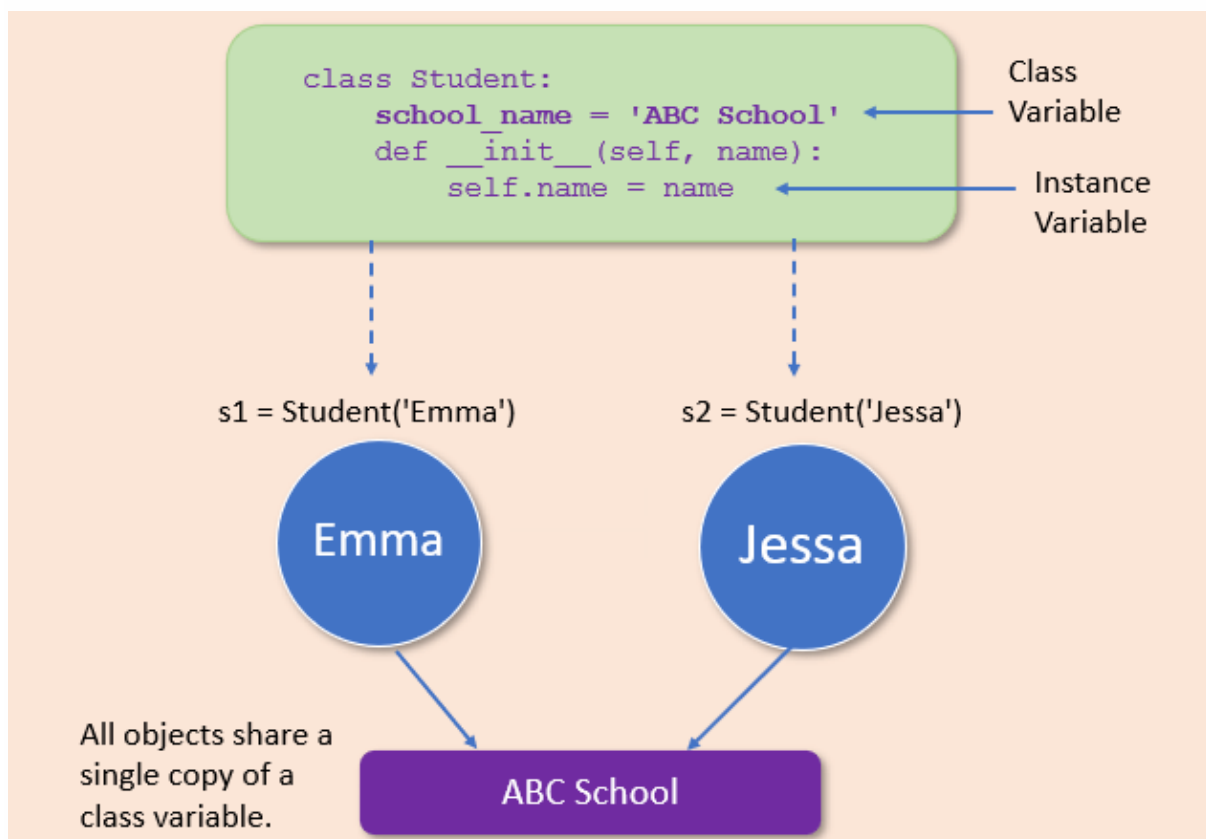
CLASS VARIABLE IN PYTHON

A Python class variable is **shared by all object instances of a class**. Class variables are declared when a class is being constructed. They are not defined inside any methods of a class. Because a class variable is shared by instances of a class, the Python class owns the variable.

A class variable is a variable that is declared inside of class, but outside of any instance method or `__init__()` method.

If the **value of a variable is not varied from object to object**, such types of variables are called class variables or static variables.

Class variables are **shared by all instances of a class**. Class variables are declared when a class is being constructed.



EXAMPLE:

```
class Student:
    # Class variable
    College_name = 'SXC '

    def __init__(self, name, roll_no):
        self.name = name
```

```

        self.roll_no = roll_no
# create first object
s1 = Student('Arthi', 10)
print(s1.name, s1.roll_no, Student.College_name)
# access class variable
# create second object
s2 = Student('Jhony', 20)
# access class variable
print(s2.name, s2.roll_no, Student.College_name)

```

```

Arthi 10 SXC
Jhony 20 SXC

```

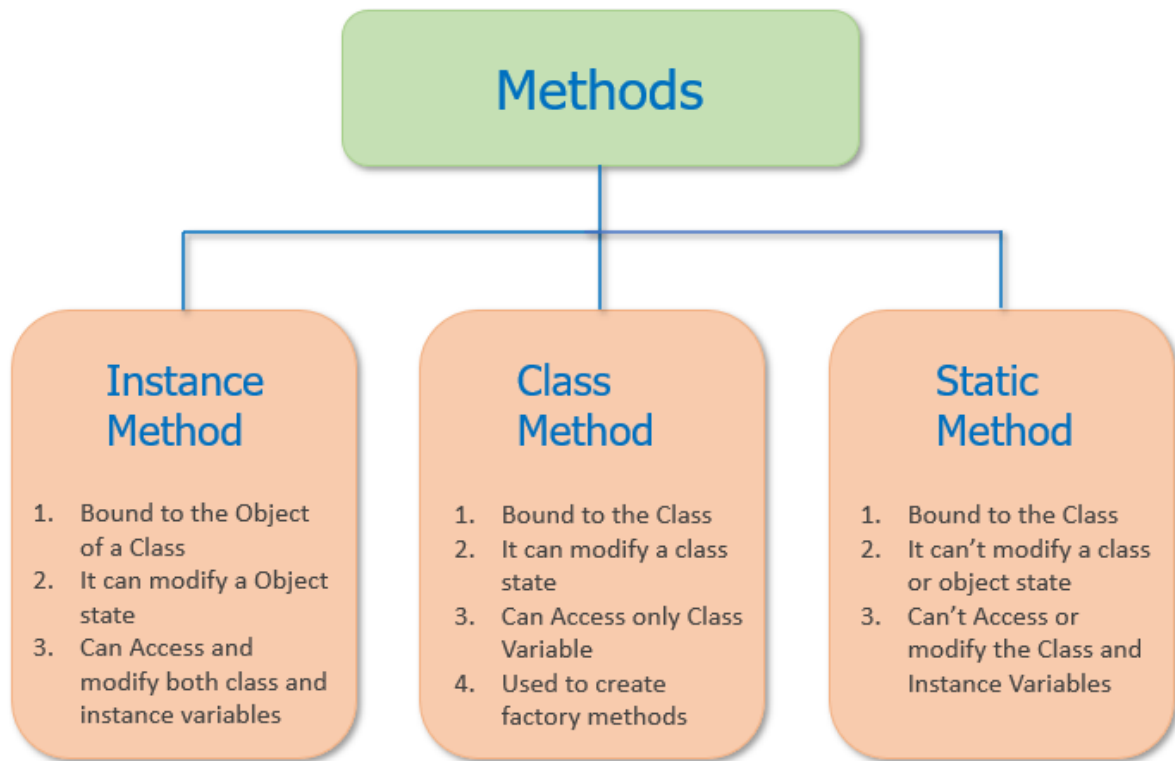
INSTANCE METHOD, CLASS METHOD AND STATIC METHOD

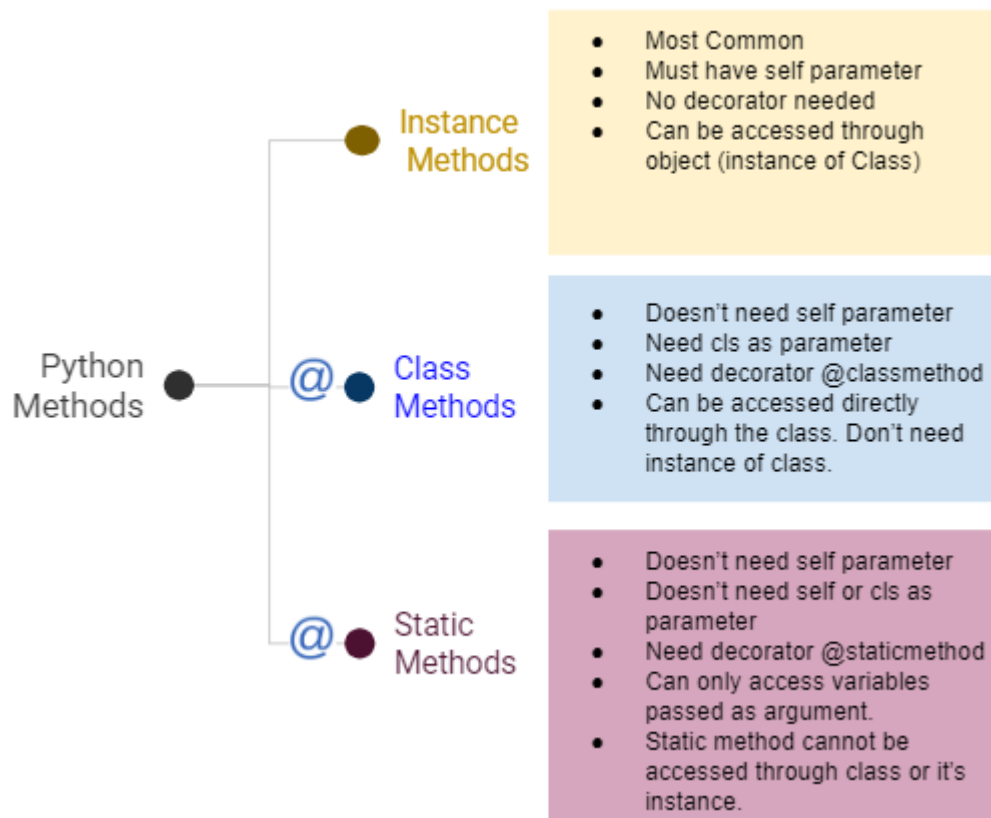
Instance method performs a set of actions on the data/value provided by the instance variables. If we use instance variables inside a method, such methods are called instance methods. The instance method acts on an object's attributes. It can modify the object state by changing the value of instance variables. The instance method receives the caller object as the first parameter, and it requires no decorator.

Class method is method that is called on the class itself, not on a specific object instance. Therefore, it belongs to a class level, and all class instances share a class method. Class method Used to access or modify the class state. It can modify the class state by changing the value of a class variable that would apply across all the class objects. The class method receives the caller class as the first parameter, and it requires the `@classmethod` decorator.

Static method is a general utility method that performs a task in isolation. This method doesn't have access to the instance and class variable. Static methods have limited use because they don't have access to the attributes of an object (instance variables) and class attributes (class variables). However, they can be helpful in utility such

as conversion form one type to another. A static method does not receive an implicit first argument. The static method does not take any necessary parameter, and it requires the `@staticmethod` decorator.





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

```
@staticmethod
def is_adult(age):
    if age >= 18:
        return True
    else:
        return False
```

@classmethod

```
def create(cls, name, age):  
    if cls.is_adult(age) == True:  
        return cls(name, age, "Yes ,can Vote")  
    else:  
        return cls(name, age, "No,can't Vote")  
  
st1 = Person.create("Antro", 15)  
st2 = Person.create("Ajina", 20)  
print("Can", st1.name, "vote?", st1.can_vote)  
print("Can", st2.name, "vote?", st2.can_vote)
```

```
Can Antro vote? No,can't Vote  
Can Ajina vote? Yes ,can Vote
```

Instance Method Example:

```
class Student():  
    def __init__(self, name):  
        self.name = name  
    def display(self):  
        return self.name  
st1 = Student("Antro")  
st2 = Student("Ajina")  
print(st1.display())  
print(st2.display())
```

```
Antro  
Ajina
```


Super() METHOD IN PYTHON

- The **super() method** is used to give access to methods and properties of a parent or sibling class. The super() function returns an object that represents the parent class. Allows us to avoid using the base class name explicitly. The super() function in Python implements code reusability and modularity as there is no need for us to rewrite the whole function again and again. The super() function in Python is known as dynamical function. The arguments are given in the super() function and the arguments in the function that we have called should match.

Syntax:

```
super(type, object)
```

Parameters:

1. type: (Optional) The class name whose base class methods needs to be accessed
2. object: (Optional) An object of the class or self.

EXAMPLE for super()

```
class Rectangle:
```

```
    def __init__(self, length, width):
```

```
        self.length = length
```

```
        self.width = width
```

```
    def area(self):
```

```
        return self.length * self.width
```

```
    def perimeter(self):
```

```
        return 2 * self.length + 2 * self.width
```

```
class Square(Rectangle):  
    def __init__(self, length):  
        super().__init__(length, length)  
  
sqr = Square(4)  
print("Area of Square is:", sqr.area())  
  
rect = Rectangle(2, 4)  
print("Area of Rectangle is:", rect.area())
```

```
Area of Square is: 16  
Area of Rectangle is: 8
```

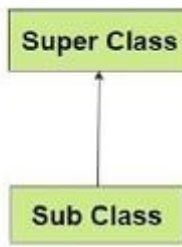
INHERITANCE IN PYTHON

Inheritance is the capability of one class to derive or inherit the properties from another class. Inheritance relationship defines the classes that inherit from other classes as derived, subclass, or sub-type classes.

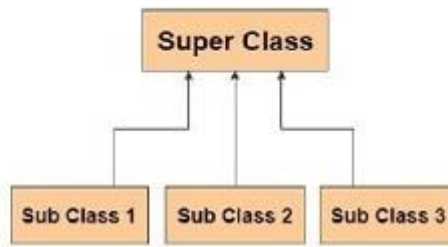
Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

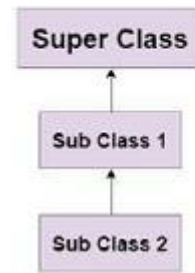
Single Inheritance



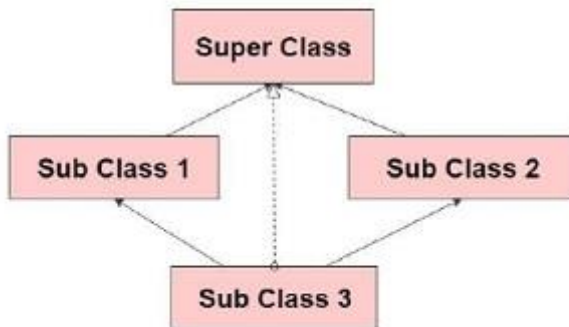
Hierarchical Inheritance



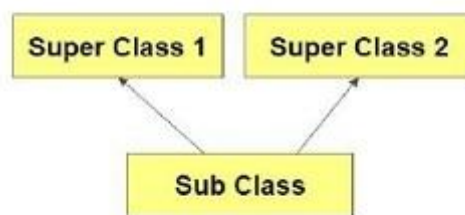
MultiLevel Inheritance



Hybrid Inheritance

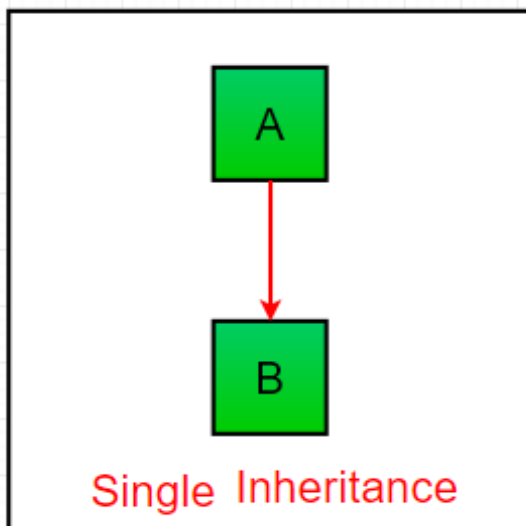


Multiple Inheritance



SINGLE INHERITANCE

Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.



SINGLE INHERITANCE: PROGRAM

class Polygon:

```
def __init__(self, no_of_sides):  
    self.n = no_of_sides  
    self.sides = [0 for i in range(no_of_sides)]
```

def inputSides(self):

```
self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]
```

def dispSides(self):

```
for i in range(self.n):  
    print("Side",i+1,"is",self.sides[i])
```

class Triangle(Polygon):

def __init__(self):

```
Polygon.__init__(self,3)
```

def findArea(self):

```
a, b, c = self.sides  
# calculate the semi-perimeter  
s = (a + b + c) / 2  
area = (s*(s-a)*(s-b)*(s-c)) ** 0.5  
print('The area of the triangle is %0.2f' %area)
```

```
t = Triangle()
```

```
t.inputSides()
```

```
t.dispSides()
```

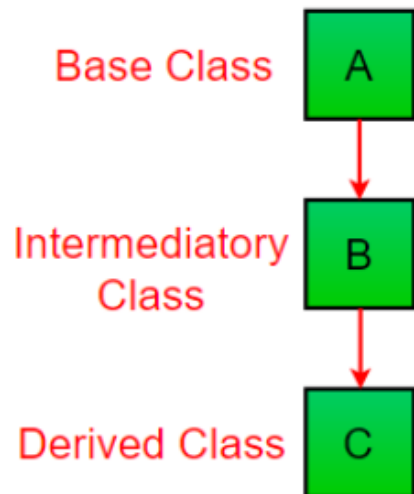
```
t.findArea()
```

OUTPUT:

```
Enter side 1 : 3  
Enter side 2 : 4  
Enter side 3 : 2  
Side 1 is 3.0  
Side 2 is 4.0  
Side 3 is 2.0  
The area of the triangle is 2.90
```

MULTI LEVEL INHERITANCE

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and grandfather.



Multilevel Inheritance

MULTI-LEVEL INHERITANCE: PROGRAM 1

```
class Employees():
    def Name(self):
        print ("Employee Name: Jeba")

class salary(Employees):
    def Salary(self):
        print ("Salary: 85000")

class Designation(salary):
    def desig(self):
        print( "Designation: System Engineer")
```

```
call = Designation()
call.Name()
call.Salary()
```

```
Salary: 85000
Designation: System Engineer
```

MULTI-LEVEL INHERITANCE: PROGRAM 2

```
class student:
    def getStudent(self):
        self.name = input("Name: ")
        self.age = input("Age: ")
        self.gender = input("Gender: ")
class test(student):
    # Method
    def getMarks(self):
        self.stuClass = input("Class: ")
        print("Enter the marks of the respective subjects")
        self.c = int(input("C Programming: "))
        self.cpp = int(input("C++ Programming: "))
        self.java = int(input("Java Programming: "))
        self.py = int(input("Python Programming: "))
class marks(test):
    # Method
    def display(self):
        print("\n\nName: ",self.name)
        print("Age: ",self.age)
        print("Gender: ",self.gender)
        print("Study in: ",self.stuClass)
        print("Total Marks: ", self.c + self.cpp + self.java + self.py)

obj = marks()
obj.getStudent()
obj.getMarks()
print("*****")
obj.display()
print("*****")
```

OUTPUT:

```

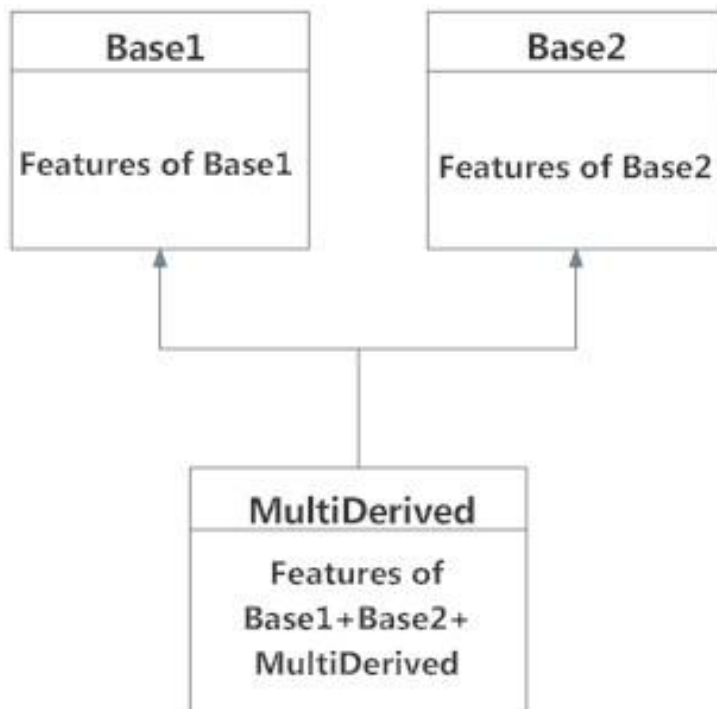
Name: George
Age: 20
Gender: Male
Class: II B.Sc
Enter the marks of the respective subjects
C Programming: 90
C++ Programming: 80
Java Programming: 92
Python Programming: 98
*****

Name: George
Age: 20
Gender: Male
Study in: II B.Sc
Total Marks: 360
*****

```

MULTIPLE INHERITANCE:

When a class can be derived from more than one base class this type of inheritance is called multiple inheritance. In multiple inheritance, all the features of the base classes are inherited into the derived class.



MULTIPLE INHERITANCE: PROGRAM 1

```

class Car():
    def Benz(self):
        print(" This is a Benz Car ")
class Bike():
    def Bmw(self):
        print(" This is a BMW Bike ")
class Bus():
    def Volvo(self):

```

```

    print(" This is a Volvo Bus ")
class Truck():
    def Eicher(self):
        print(" This is a Eicher Truck ")
class Plane():
    def Indigo(self):
        print(" This is a Indigo plane ")
class Transport(Car,Bike,Bus,Truck,Plane):
    def Main(self):
        print("This is the Main Class:ENJOY TRAVELLING")
B=Transport()
B.Benz()
B.Bmw()
B.Volvo()
B.Eicher()
B.Indigo()
B.Main()

```

OUTPUT:

```

This is a Volvo Bus
This is a Eicher Truck
This is a Indigo plane
This is the Main Class:ENJOY TRAVELLING

```

MULTIPLE INHERITANCE: PROGRAM 2

```

class Addition:
    def Sum(self,a,b):
        return a+b;
class Multiplication:
    def Mul(self,a,b):
        return a*b;
class Divide(Addition,Multiplication):
    def Div(self,a,b):
        return a/b;
ans = Divide()
num1=int(input("Enter first Number  :"))
num2=int(input("Enter second Number  :"))
print("*****")

```



```

print("ARITHMETIC OPERATIONS:MULTIPLE INHERITANCE")
print("*****")
print("Number 1 =",num1)
print("Number 2=",num2)
print("*****")

print("ADDITION      : ",ans.Sum(num1,num2))
print("MULTIPLICATION : ",ans.Mul(num1,num2))
print("DIVISION       : ",ans.Div(num1,num2))

```

OUTPUT:

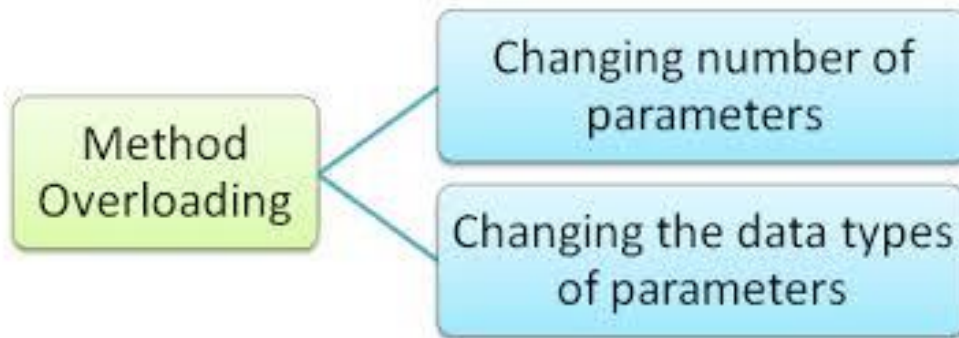
```

Enter first Number :45
Enter second Number :5
*****
ARITHMETIC OPERATIONS:MULTIPLE INHERITANCE
*****
Number 1 = 45
Number 2= 5
*****
ADDITION      : 50
MULTIPLICATION : 225
DIVISION       : 9.0

```

METHOD OVERLOADING

- Method Overloading is the class having methods that are the same name with different arguments.
- Arguments different will be based on a number of arguments and types of arguments.
- It is used in a single class.
- It is also used to write the code clarity as well as reduce complexity.



Advantages of using overload are:

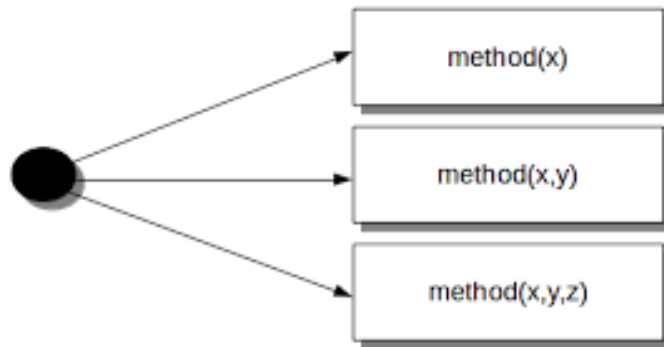
- Overloading a method fosters reusability. For example, instead of writing multiple methods that differ only slightly, we can write one method and overload it.
- Overloading also improves code clarity and eliminates complexity.

Without Method Overloading

```
int add2(int x, int y)
{
    return(x+y);
}
int add3(int x, int y, int z)
{
    return(x+y+z);
}
int add4(int w, int x, int y, int z)
{
    return(w+x+y+z);
}
```

With Method Overloading

```
int add(int x, int y)
{
    return(x+y);
}
int add(int x, int y, int z)
{
    return(x+y+z);
}
int add(int w, int x, int y, int z)
{
    return(w+x+y+z);
}
```



```
class Person:  
def Hello(self, name=None):  
if name is not None:  
    print('Hello ' + name)  
else:  
    print('Hello SXC')  
# Create instance  
obj = Person()  
# Call the method  
obj.Hello()  
# Call the method with a parameter  
obj.Hello('S')
```

OUTPUT:
Hello
Hello SXC

```
# class
class Compute:
# area method
def area(self, x = None, y = None):
if x != None and y != None:
return x * y
elif x != None:
return x * x
else:
return 0
# object
obj = Compute()
print("Area Value:", obj.area())
# one argument
print("Area Value:", obj.area(4))
# two argument
print("Area Value:", obj.area(3, 5))
```

Area Value: 0
Area Value: 16
Area Value: 15

OVERRIDING IN PYTHON

- Overriding is the ability of a class to change the implementation of a method provided by one of its ancestors.
- Overriding is a very important part of OOP since it is the feature that makes inheritance exploit its full power. Through method overriding a class may "copy" another class, avoiding duplicated code, and at the same time enhance or customize part of it. Method overriding is thus a strict part of the inheritance mechanism.
- Method Overriding is the method having the same name with the same arguments.

- It is implemented with inheritance also.
- It mostly used for memory reducing processes.
- Following conditions must be met for overriding a function:
- **Inheritance** should be there. Function overriding cannot be done within a class. We need to derive a child class from a parent class.
- The function that is redefined in the child class should have the same signature as in the parent class i.e. same **number of parameters**.

Method overloading	Method overriding
1. More than one method with same name, different prototype in same scope is called method overloading.	1. More than one method with same name, same prototype in different scope is called method overriding.
2. In case of method overloading, parameter must be different.	2. In case of method overriding, parameter must be same.
3. Method overloading is the example of compile time polymorphism.	3. Method overriding is the example of run time polymorphism.
4. Method overloading is performed within class.	4. Method overriding occurs in two classes.
5. In case of method overloading, Return type can be same or different.	5. In case of method overriding Return type must be same.
6. Static methods can be overloaded which means a class can have more than one static method of same name.	6. Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class.
7. <u>Static binding</u> is being used for overloaded methods	7. <u>dynamic binding</u> is being used for overridden/overriding methods.

EXAMPLE:

```
class CSE:
    def message(self):
        print('This message is from CSE')

class IIBSC(CSE):
    def message(self):
        print('This message is from II BSC')

class IIMSC(CSE):
    def message(self):
        print('This message is from II MSC')

c1 = CSE()
c1.message()
print('-----')
c2 = IIBSC()
c2.message()
print('-----')
c3 = IIMSC()
c3.message()
```

```
This message is from CSE
-----
This message is from II BSC
-----
This message is from II MSC
```

EXAMPLE:

```
class Arith:
    def add(self, a, b):
        print('The Sum of Two = ', a + b)

class ArithThree(Arith):
    def add(self, a, b, c):
        print('The Sum of Three = ', a + b + c)

e = Arith()
e.add(10, 50)

print('-----')
e2 = ArithThree()
e2.add(50, 150, 200)
```

```
The Sum of Two = 60
-----
The Sum of Three = 400
```

OPERATOR OVERLOADING IN PYTHON

Operator overloading in Python is **the ability of a single operator to perform more than one operation based on the class (type) of operands**. The operator overloading in Python means provide extended meaning beyond their predefined operational meaning.

Such as, we use the "+" operator for adding two integers as well as joining two strings or merging two lists.

```
print (10 + 10)
```

```
print ("ii" + "bsc")
print (23 * 10)
print ("bsc " * 3)
```

OUTPUT

20
iibsc
220
Bscbscbsc

EXAMPLE 1:

```
class Sample:
    def __init__(self ,a):
        self.a=a
    def __mul__(self,obj):
        return self.a * obj.a
```

```
obj1 =Sample(5)
obj2 =Sample(4)
obj3=Sample("JREXY&")
print(obj1 *obj2)
print(obj3 * obj2)
```

20

20
JREXY&JREXY&JREXY&JREXY&

EXAMPLE 2:

```
class Arithmetic:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __add__(self, obj): #overloading '+' operator
        a = self.a + obj.a
        b = self.b + obj.b
```

```

    return(Arithmetic(a, b))
def __sub__(self, obj): #overloading '-' operator
    a = self.a - obj.a
    b = self.b - obj.b
    return(Arithmetic(a, b))

def __mul__(self, obj): #overloading '*' operator
    a = self.a * obj.a
    b = self.b * obj.b
    return(Arithmetic(a, b))
def display(self):
    print("a =", self.a, " b =", self.b)

```

```

c1 = Arithmetic(4, 9)
c1.display()

```

```

c2 = Arithmetic(2, 3)
c2.display()

```

```

print("*****")
addition = c1 + c2
addition.display()

```

```

subtraction = c1 - c2
subtraction.display()

```

```

multiplication = c1 * c2
multiplication.display()
print("*****")

```

```

a = 4  b = 9
a = 2  b = 3
*****
a = 6  b = 12
a = 2  b = 6
a = 8  b = 27
*****

```

```

*****

```


EXAMPLE 3:

```
class X:  
    def __init__(self, x):  
        self.x = x
```

```
    # adding two objects  
    def __add__(self, y):  
        return self.x + y.x
```

```
ob1 = X(20)  
ob2 = X(25)  
ob3 = X("Rexy")  
ob4 = X("Jeba")
```

```
print(ob1 + ob2)  
print(ob3 + ob4)
```

OUTPUT:

45

RexyJeba

EXAMPLE 4:

```
class Student:  
    def __init__(self, m1, m2):  
        self.m1 = m1  
        self.m2 = m2  
    def __add__(self, m1, m2): #adding the two objects  
        m1 = self.m1 + other.m1  
        m2 = self.m2 + other.m2  
        s3 = student (m1,m2)  
        return s3  
    def __gt__(self, other): #comparingthe two objects  
        r1 = self.m1 + self.m2  
        r2 = other.m1 + other.m2  
        if(r1 > r2):  
            return True
```

```
    else:
        return False
arthi = Student(100, 75)
anisha = Student(90, 80)
if (arthi > anisha):
    print ("arthi wins")
else:
    print ("anisha wins")
```

OUTPUT:arthi wins

METHOD RESOLUTION ORDER

- MRO is a concept used in inheritance. It is the order in which a method is searched for in a classes hierarchy
- MRO is from bottom to top and left to right
- This order is called linearization of class Child, and the set of rules applied are called MRO (Method Resolution Order).
- it plays vital role in the context of multiple inheritance as single method may be found in multiple super classes.
- The Python Method Resolution Order defines the class search path used by Python to search for the right method to use in classes having multi-inheritance. It as evolved since Python 2.2 to 2.3.

CASE 1:

```
1 class Apple:
2     def eat(self):
3         print("This is Apples Eat method" )
4
5 class Orange:
6     pass
7
8 class Grapes(Apple, Orange):
9     pass
10 obj = Grapes()
11 obj.eat()
12 print(Grapes.mro()) # print MRO for class Grapes
```

input

```
This is Apples Eat method
[<class '__main__.Grapes'>, <class '__main__.Apple'>, <class '__main__.Orange'>, <class 'object'>]
```

From MRO of class Grapes, we get to know that Python looks for a method first in class **Grapes**. Then it goes to **Apple** and then to **Orange**. So, first it goes to super class given first in the list then second super class, from left to right order. Then finally Object class, which is a super class for all classes.

CASE 2:

```
1 class Apple:
2     def eat(self):
3         print("this is Apple's eat")
4 class Orange:
5     def eat(self):
6         print("this is Orange's eat")
7
8 class Grapes(Apple, Orange):
9     pass
10 obj = Grapes()
11 obj.eat()
12 print(Grapes.mro())
```

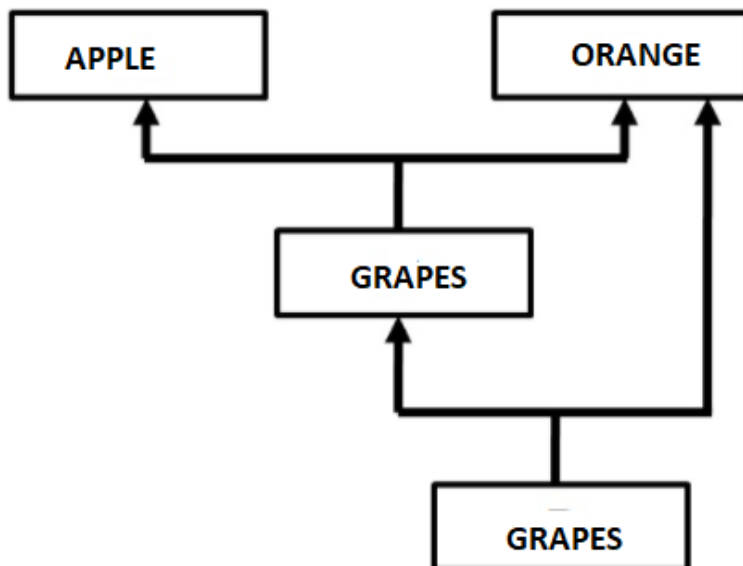
input

```
this is Apple's eat
[<class '__main__.Grapes'>, <class '__main__.Apple'>, <class '__main__.Orange'>, <class 'object'>]
```

- Python calls eat() method in class Apple. According to MRO, it searches Apple first and then Orange. So if method is found in Apple then it calls that method.
- However, if we remove eat() method from class Apple then eat() method in class Orange will be called as it is the next class to be searched according to MRO.

CASE 3:

- create Cherry from Grapes and Orange. Classes Grapes and Orange have eat() method and as expected MRO chooses method from Grapes. Remember it goes from left to right. So it searches Grapes first and all its super classes of Grapes and then Orange and all its super classes. We can observe that in MRO of the output given below.



```

1 class Apple:
2     def eat(self):
3         print("This is Apple's eat")
4 class Orange:
5     def eat(self):
6         print("This is Orange's eat")
7 class Grapes(Apple, Orange):
8     def eat(self):
9         print("This is Grapes's eat")
10 class Cherry(Grapes, Orange):
11     pass
12 obj = Cherry()
13 obj.eat()
14 print(Cherry.mro())

```

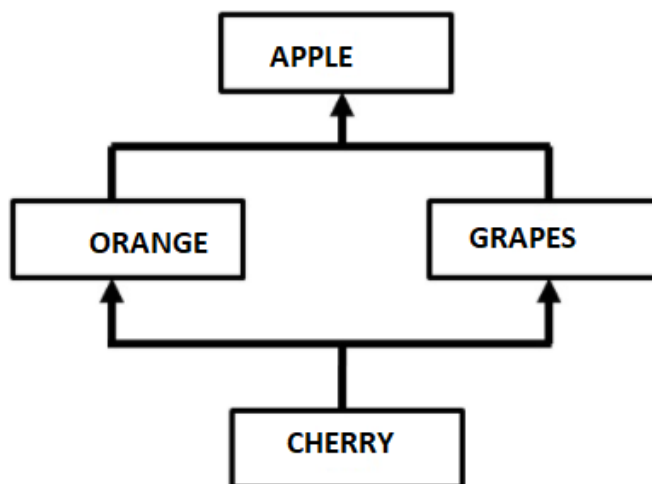
```

This is Grapes's eat
[<class '_main_.Cherry'>, <class '_main_.Grapes'>, <class '_main_.Apple'>, <class '_main_.Orange'>, <class 'object'>]

```

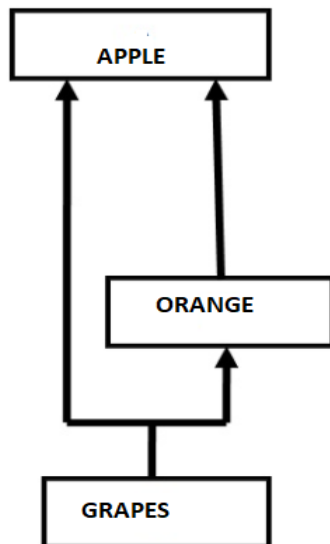
CASE 4:

Method eat() is present in both APPLE and GRAPES.



CASE 5:

- There are cases when Python cannot construct MRO owing to complexity of hierarchy. In such cases it will throw an error as demonstrated by the following code.



```
1 class Apple:
2     def eat(self):
3         print("APPLE")
4 class Orange(Apple):
5     def eat(self):
6         print("ORANGE")
7 class Grapes(Apple, Orange):
8     pass
9 obj = Grapes()
10 obj.eat()
```

input

```
Traceback (most recent call last):
  File "main.py", line 7, in <module>
    class Grapes(Apple, Orange):
TypeError: Cannot create a consistent method resolution
order (MRO) for bases Apple, Orange
```