

Object Oriented Programming – Python

So, here we will be learning about Object Oriented Programming (OOPs) in Python and know how to deal with those. We have a lot of Concepts to Cover like,

Classes, Objects, Encapsulation, Polymorphism, Abstraction and a lot more So, let's deep dive into it.

Object

Now, Objects are a way of defining things we work with. So, Everything we work with in Python or another language or maybe even in real world everything is an Object.

Class

Classes are a way of categorizing objects into different forms. In simple words Classes are a blueprint of what we do and Objects define that. Let's take an Analogy if we have a Samsung or iPhone obviously of any particular model and that's obviously not the only model you have. There are millions of thousands of units of that phone so all of those are Objects and they are of the Class Phone. We can also say that Objects are an instance of a particular Class.

Now let's begin with Programming.

Now to define a class in Python we use the keyword `Class`. Let's say we want to work with computer, now obviously we don't have any computer data type so we create a class of our own for that and make an instance or object of that class which will help us retrieve information from that class.

Note: Functions inside a Class are called Methods.

Let's See how we play around with the Code.

```
1 # Ways of defining a Class in Python
2 class Computer:
3     # In a Class we can put in two things behaviours (Methods)
4     # and Attributes (variables)
5     def config(self): # Here we define the configuration of Computer
6         print("i5, 16 GB, 1TB")
7
8 # Since the Variable contains an int datatype
9 # So it will be of <class 'int'>
10 a = 9
11 print(type(a))
12 |
13 # Obviously just like any other Variable we don't
14 # have a type for this variable. So we need to mention
15 # that it is an instance of the Class.
16 com1 = Computer()
17
18 # Let's Check the type of the Object of Computer
19 print(type(com1))
20
21
```

```
hell <
>>> %Run 'OOPs Tutorials.py'
<class 'int'>
<class '__main__.Computer'>
```

Now, let's see how we call a method from a Class. We simply first create an instance or Object of the class and then we mention the Class and finally pass the Object into the Called method. Let's See how that works.

```

20
21 class Computer: # class is defined
22     def config(self): # while working with instances we use self in the methods
23         print("i5, 16GB, 1TB")
24
25 com1 = Computer() # Initializing com1 as the Object of the Class Computer
26 Computer.config(com1)
27
28
29

```

The Output as we Predict is,

```

>>> %Run 'OOPs Tutorials.py'
i5, 16GB, 1TB

```

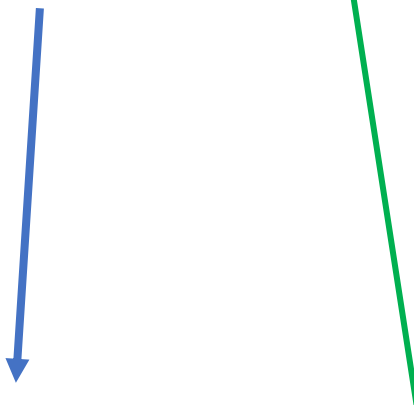
Technically when we work with classes then inside the methods we pass self as the argument of that particular method.


Now, let's see if we call the Class with a different object and with other way. How to do it?

```

9
0 class Computer: # Class gets initialized
1
2     def config(self): # Registering the first method with the Self argument
3         print("i5, 16GB, 1TB")
4
5
6 # Initializing the First Object
7 com1 = Computer()
8
9 #Initializing the Second Object
0 com2 = Computer()
1
2 # Here we have both the Objects of the Same Class.
3 # Now, let' call each of those
4
5 Computer.config(com1)
6 Computer.config(com2)
7
8 # Let's See a Different Way of calling a method which is uded generally.
9 # In this method we use the Object itself to call the method of the Class.
0 com1.config()
1 com2.config()
2

```





Here we call the method by the Object itself. Behind the Scene what is happening is as we have a **self** argument for the method we have in a class. So, the Object we use before the `'.'` gets replaced by **Self**.

The Output Comes out to be,

```
>>> %Run 'OOPs Tutorials.py'
i5, 16GB, 1TB
i5, 16GB, 1TB
i5, 16GB, 1TB
i5, 16GB, 1TB
```

Now, let's talk of Variables with Methods in a Class. To do with that we have a question, where to define the variables? For that we have special method with underscores. The method is `"__init__"`. It is use to initialise Variables it can be treated as a Constructor also. Also, the idea behind `"__init__"` is that it will called automatically every time we have a Object/instance of the Class. Let's see how we do that. We will also se how to pass variables there.

```
class Computer: # Class gets initialized

    def __init__(self): # Constructor/Variable Storage gets initialized
        print("in init")

    def config(self): # The First Method with a Self Argument
        print("i5, 16GB, 1TB")

# First and Second Object of the Class is Created
com1 = Computer()
com2 = Computer()

# Let's Call the method using the Objects
# Now, as we know init is called automatically for the number of
# objects created.

com1.config()
com2.config()
```

Now, the Output will be let's see,

```
in init
in init
i5, 16GB, 1TB
i5, 16GB, 1TB
```

Now, we have seen how “__init__” works. Let’s See how to pass the Variables and how that works.

So, as we know that “__init__” is used to store variables of values passed on to the Class. Now, while doing that what we do is we call the method of the class with the help of the Object and while we create the Object we pass the Values in the “__init__”. But underlying, while we call the method with the help of the Object what happens is that actually $n+1$ number of arguments are passed where n is the number of values you have given in the Object and one extra is of the Self in the Constructor which takes the Object as a Value. Let’s see the Code in action!!

```
5 class Computer: # Class gets initialized
6
7     def __init__(self, cpu, ram): # Constructor/Variable Storage gets initialized
8         # Obviously we have values in parameters but need variables to
9         # store them
10        self.a = cpu # We do that by writing "self.'variable' = Parameter"
11        self.b = ram
12
13    def config(self): # The First Method with a Self Argument
14        # Now we print the Details Passed in by the Object. Again we need to access the
15        # Variables in the Class by Self since they are instance Variables
16        print(f"Config is {self.a} and {self.b}")
17
18    # First and Second Object of the Class is Created, and
19    # Each Object has it's own Configuration.
20
21    com1 = Computer('i5', 16) # Datas of the First Object
22    com2 = Computer('Ryzen 3', 8) # Datas of the Second Object
23
24    # Let's Call the method using the Objects
25    # Now, as we know init is called automatically for the number of
26    # objects created.
27
28    com1.config()
29    com2.config()
```

Now, every Object has it’s own data so the Output will be different for different Objects passing it’s data into the Constructor.

Output

```
@Apurba → Python Programming c:
ata\Local\Programs\Python\Python39
honFiles\lib\python\debugpy\launch
s.py'
Config is i5 and 16
Config is Ryzen 3 and 8
@Apurba → Python Programming
```

*Now, let's explore the **Constructor** and **Self** more.*

In your Computer there is a Concept of Heap Memory. Which stores the Data. So, every time you create an Object it gets allocated a Different Memory Location. So, the Point being every time you create an Object it will get allocated to different Spaces.

```
class Computer:
    pass

c1 = Computer()
c2 = Computer()

print(id(c1))
print(id(c2))
```

Output

```
@Apurba → Python Programming
ata\Local\Programs\Python\Pyth
honFiles\lib\python\debugpy\la
s.py'
1778193871872
1778193871968
```

Now, the Question is how much space or memory do these uses?, Who decides this?

- i) Size of an Object is dependent on number of Variables and Size of each Variables.

And the Constructor decided the amount of memory to be allocated. Let's see an Example

```
class Computer: # Class gets initialized

    def __init__(self): # Constructor with the instance Variables
        self.name = "Navin"
        self.age = 28

# Objects of Class getting initialized
c1 = Computer()
c2 = Computer()

print(c1.name)
print(c2.name)
```

Output

```
@Apurba → Python Programming
ata\Local\Programs\Python\Python
honFiles\lib\python\debugpy\lau
s.py'
Navin
Navin
```

Now, if we wanna change the value of the variable outside the class we do that by,

```

class Computer: # Class gets initialized

    def __init__(self): # Constructor with the instance Variables
        self.name = "Navin"
        self.age = 28

# Objects of Class getting initialized
c1 = Computer()
c2 = Computer()

# Change the Variable Value
c1.name = 'Apurba'

print(c1.name)
print(c2.name)

```

Output

```

@Apurba → Python Programming
ata\Local\Programs\Python\Pyth
honFiles\lib\python\debugpy\la
s.py'
Apurba
Navin

```

Now, let's see why this **Self** is necessary. Let's make another method in the Class and it will update the age.


```

class Computer: # Class getting initialized

    def __init__(self): # Constructor with two Self Variables
        self.name = "Apurba"
        self.age = 18

    def update(self): # Updating the Age with self as parameter
        # Now this Self does an important Job, as soon as you call the method how does
        # the Method knows which Objects age to Update, it is determined by the Self which
        # takes the Object as the Parameter as said before. So, if you have 10 Objects pointing
        # Out you can work with a Single one by just calling the method with that particular
        # Object
        self.age = 30

c1 = Computer()
c2 = Computer()
print("By Default",c1.name, c1.age)

# After we Update the Value
c1.name = 'Navin'
c1.age = 12
print("After Changing",c1.name, c1.age)

# Calling the Update Method and getting it restored with the Age Changed
c1.update()
print("Calling Update",c1.name, c1.age)

```

As you can read above the Comments in the Program has explained it all.

Now, what if I want to compare the age of two Objects, how to do that? Since c1 and c2 contains the Memory Address, how to compare a particular property of that Objects. We do that by declaring a method which we will have to define. Let's See how to do it.

```

class Computer: # Class getting initialized

    def __init__(self): # Constructor with two Self Variables and their default values
        self.name = "Apurba"
        self.age = 18

    def compare(self, other): # Making a Compare Method with two variables
        # Now, important thing is that here the self is getting replaced by the
        # Object which calls it and other is getting replaced by the Object which is
        # Passed as an Argument
        if self.age == other.age: # Comparing age of two Objects and returning results
            return True
        else:
            return False

# Initializing the Objects
c1 = Computer()
# Updating the Age of c1 as by default both of them has age as 18
c1.age = 30
c2 = Computer()

if c1.compare(c2):
    print("They Are Same")
else:
    print("They Are Different")

```

Output

```

@Apurba → Python Programming
ograms\Python\Python39\python.
gpy\launcher' '52017' '--' 'c:
They Are Different

```

So, we can Compare two Objects by defining our Own Methods but yes it will take two Parameters one is **Self** which gets replaced by the **Object who is Calling the method** and the **Other** is a **formal argument** which gets replaced by the **Object** which is passed.

Let's look more at Variables

So, when we talk about variables in a Class we have two types of Variables, One is **Instance Variable** and Other is **Static(Class) Variable**. Now, to define a Variable in a Class we do that in a **"__init__"** method and those are Called Instance Variable, as by default they have some values but they can be changed with each Object as we Know Variables inside a Special Method can be

Changed as they are different for each Object. So, changing them for one Object does not affect the Others. Let's look at an Example,

```
class Car: # Class gets Initialized

    def __init__(self): # Constructor gets Initialized with two Self Variables

        # Now Since these Variables are inside the Constructor so they are called Instance
        # Variables as they might as well have a Default Value but it changes for each Object
        # Separately according to the Object Called and does not affect all the Object's Value
        self.mil = 10 # It tells the Milage of the Car
        self.com = "BMW" # It tells the Company/Brand of the Car

# Objects getting Initialized
c1 = Car()
c2 = Car()

# Changing the Car Brand for the Object 1
c1.com = "Audi"

# Printing the Values for both the Objects
print(f"""          Credentials of C1:
Car Milage: {c1.mil}
Car Brand: {c1.com}
          Credentials of C2:
Car Milage: {c2.mil}
Car Brand: {c2.com}""")
```

Output

```
@Apurba → Python Programming c:
ograms\Python\Python39\python.exe'
gpy\launcher' '52214' '--' 'c:\Use
          Credentials of C1:
Car Milage: 10
Car Brand: Audi
          Credentials of C2:
Car Milage: 10
Car Brand: BMW
```

But what if we wanted a Variable that's same for every Object. That's when we use Class Variable.

Concept:

See, when we work with Variables we have two types of NameSpace in the Program.

What is a NameSpace?

- So, a NameSpace is an area where you create and store Object/Variable.

The two types of NameSpaces are:

- i) Class NameSpace
- ii) Object/Instance NameSpace

Now, let's say we have 4 wheels till now and that's same for all the Object as it's defined not in Special Method but the Class itself and we will check that by printing the Values for all the Objects Separately. So, if we want to suddenly change the number of Wheels to 5 and want to change it for all. Let's see how to do it and then we will check if it's updated.

Let's See a Class Variable of same Value for all Different Objects.

```
class Car: # Class gets Initialized

    # Defining the Class Variable
    wheels = 4 # --> Class Variable

    def __init__(self): # Constructor gets Initialized with two Self Variables

        # Now Since these Variables are inside the Constructor so they are called Instance
        # Variables as they might as well have a Default Value but it changes for each Object
        # Separately according to the Object Called and does not affect all the Object's Value

        self.mil = 10 # It tells the Milage of the Car --> Instance Variable
        self.com = "BMW" # It tells the Company/Brand of the Car --> Instance Variable

# Objects getting Initialized
c1 = Car()
c2 = Car()

# Changing the Car Brand for the Object 1
c1.com = "Audi"

# Printing the Values for both the Objects
print(f"""          Credentials of C1:
Car Milage: {c1.mil}
Car Brand: {c1.com}
Car Wheels: {c2.wheels}

          Credentials of C2:
Car Milage: {c2.mil}
Car Brand: {c2.com}
Car Wheels: {c2.wheels}""")
```

Output:

```

@Apurba → Python Programming
ams\Python\Python39\python.exe'
uncher' '52307' '--' 'c:\Users\
    Credentials of C1:
Car Milage: 10
Car Brand: Audi
Car Wheels: 4
    Credentials of C2:
Car Milage: 10
Car Brand: BMW
Car Wheels: 4

```

Note that in both the Objects the Value of the number of Wheels are Same.

Now, what if we want to change the Value of wheels for all the Objects, for that we need to change the Class Variable and then let's see if it changes really.

```

class Car: # Class gets Initialized

    # Defining the Class Variable
    wheels = 4 # --> Class Variable

    def __init__(self): # Constructor gets Initialized with two Self Variables

        # Now Since these Variables are inside the Constructor so they are called Instance
        # Variables as they might as well have a Default Value but it changes for each Object
        # Separately according to the Object Called and does not affect all the Object's Value

        self.mil = 10 # It tells the Milage of the Car --> Instance Variable
        self.com = "BMW" # It tells the Company/Brand of the Car --> Instance Variable

# Objects getting Initialized
c1 = Car()
c2 = Car()

# Changing the Value for wheels for all the Objects
Car.wheels = 5

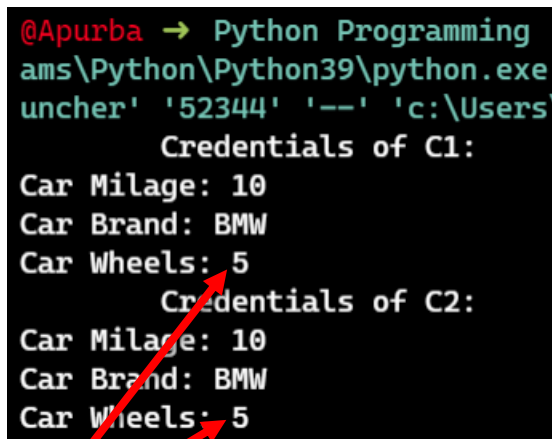
# Printing the Values for both the Objects
print(f"""
    Credentials of C1:
Car Milage: {c1.mil}
Car Brand: {c1.com}
Car Wheels: {c2.wheels}

    Credentials of C2:
Car Milage: {c2.mil}
Car Brand: {c2.com}
Car Wheels: {c2.wheels}""")

```

Note that the value of wheels is changed to 5 from 4 and we do that by defining the <Class name>.<variable name> = value.

Let's See the Output and Verify if the Value of Wheels has changed for all the Objects.



```
@Apurba → Python Programming
ams\Python\Python39\python.exe
uncher' '52344' '--' 'c:\Users'
    Credentials of C1:
Car Milage: 10
Car Brand: BMW
Car Wheels: 5
    Credentials of C2:
Car Milage: 10
Car Brand: BMW
Car Wheels: 5
```

Note that the Value of wheels has changed for both the Objects as we change the value of the Class Variable.

Note: Class Variables are also Called Static Variables.

Types of Methods

Now, when we talk of methods we have three types of Methods. They are:

- i) Instance Method
- ii) Class Method
- iii) Static Method

Note: In Variables Static and Class were same but in methods that's not the Case.

Let's create a Class Student which accepts Marks of three Subjects. And we will calculate average marks. Let's Play with that and understand the different types of Methods.

Note: In Instance Method we have two type of Methods.

- i) Accessors
- ii) Mutators

Ultimately, we are working with Instance Variables. So, if you want to fetch a value we use Accessors and on the Other hand if we wanna change the value we use Mutators.

```
class Student: # class gets Initialized

    # Declaration of Class/Static Variable
    school = 'CodeSpectrum'

    def __init__(self, m1, m2, m3): # Constructor gets Initialized with thee self Variables

        # Instance Variables are stored and deleared as they are working with Objects and can
        # be changed for each Object without disturbing other Objects.
        self.m1 = m1
        self.m2 = m2
        self.m3 = m3

    def avg(self): # Local Method for calculating average
        # This Average Method is an Instance Method as t works with the Objects

        return (self.m1 + self.m2 + self.m3) / 3 # It is returning the average of all the Values

# Objects getting deleared
s1 = Student(34, 47, 42)
s2 = Student(88, 42, 12)

# printing the Values
print(s1.avg(), s2.avg())
```

Output:

```
@Apurba → Python Programming
\Programs\Python\Python39\python.exe
n\debugpy\launcher' '52537' '--
41.0 47.333333333333336
@Apurba → Python Programming
```

Now, let's see how do we work with **Accessors** and **Mutators**

```

class Student: # class gets Initialized

    # Declaration of Class/Static Variable
    school = 'CodeSpectrum'

    def __init__(self, m1, m2, m3): # Constructor gets Initialized with thee self Variables

        # Instance Variables are stored and deleared as they are working with Objects and can
        # be changed for each Object without disturbing other Objects.
        self.m1 = m1
        self.m2 = m2
        self.m3 = m3

    def get_value(self): # Here we treat it as Accessor/Getter --> It fetches or gets the Value
        return f"The Value of m1 is: {self.m1}"

    def set_value(self, value): # Here we treat it as Mutators/Setters --> It sets or Mutates Va
        self.m1 = value

# Objects getting deleared
s1 = Student(34, 47, 42)
s2 = Student(88, 42, 12)

print("Before Setting",s1.get_value()); s1.set_value(45)
print("After Setting", s1.get_value())

```

Output:

```

@Apurba → Python Programming c;; co
\Programs\Python\Python39\python.exe'
n\debugpy\launcher' '52624' '--' 'c:\U
Before Setting The Value of m1 is: 34
After Setting The Value of m1 is: 45

```

Now, let's see how do you work with Classmethod.

Let's Say you wanna Know the info of a Student and it will print the name of the School, and since the Method Works with Class variables so changing the name of the School will affect all the Objects of the Class.

Note, here you are working with a Class Variable so you need to specify a Decorator before the method which you will declare i.e **@classmethod** also since you work with Class Variables here so you will not use **self** as argument in the Method rather you will use **cls**.

Let's see how this works.


```

class Student: # class gets Initialized

    # Declaration of Class/Static Variable
    school = 'CodeSpectrum'

    def __init__(self, m1, m2, m3): # Constructor gets Initialized with thee self Variables

        # Instance Variables are stored and deleared as they are working with Objects and can
        # be changed for each Object without disturbing other Objects.
        self.m1 = m1
        self.m2 = m2
        self.m3 = m3

    @classmethod # Property Decorator, if not deleared it will ask for argument for cls.
    def info(cls): # We use cls here as we work with Class Variables in this method.
        return cls.school

# Objects getting deleared
s1 = Student(34, 47, 42)
s2 = Student(88, 42, 12)

# Calling and getting the Clas Variable, here the School Name
print(f"The School name is: {Student.info()}")

```

Output:

```

@Apurba → Python Programming C
\Programs\Python\Python39\python.
n\debugpy\launcher' '52696' '—'
The School name is: CodeSpectrum
@Apurba → Python Programming

```

Now, let's see how Static Methods work. So, let's say you have a Situation where you don't have anything to do with Class Variables or the Instance Variables, for example factorial of a Number. Now, that what has no relation with both the variables that is when we use StaticMethods and it might be useful when you wanna work with different classes as well. Let's see a Code regarding that.

```

class Student: # class gets Initialized

    # Declaration of Class/Static Variable
    school = 'CodeSpectrum'

    def __init__(self, m1, m2, m3): # Constructor gets Initialized with thee self Variables

        # Instance Variables are stored and deleared as they are working with Objects and can
        # be changed for each Object without disturbing other Objects.
        self.m1 = m1
        self.m2 = m2
        self.m3 = m3

    @classmethod # Property Decorator, if not deleared it will ask for argument for cls.
    def info(cls): # We use cls here as we work with Class Variables in this method.
        return cls.school

    @staticmethod # Property Decorator for the static Method declaration
    def info2():
        print("This is StaticMethod in ABC Module") # Simply Prints the given Statement

# Objects getting deleared
s1 = Student(34, 47, 42)
s2 = Student(88, 42, 12)

# Calling the StaticMethod
print(f"The info is that: ", end= " ");Student.info2()

```

Inner Class

Now, we know that we can have Functions inside a Function, a Class can have a variable and Methods inside a Class. But why do we need Class inside a class? Let's see why

Now, imagine a Scenario where we have a School and then we have some Students and their Credentials are stored inside a Class and the each of them has Laptops as well and we want to know that and not only that but also the Configuration. Now, what we could do is we could ask the user to pass the Details which will be hefty tasks as we will have to create different variables for different Configurations. Now, since every Student has a Laptop so why not create a class Laptop inside a Class Student and then store the configuration there in the constructor. It would be handy. Let's see how we do it.

```

class Student: # Initializing the Class

    def __init__(self, name, roll): # Initializing the constructor with two Instance variables
        self.name = name
        self.roll = roll
        self.lap = self.Laptop() # Object/Instance of the Inner Class

    def show(self): # Instance Method acting kind of a Accessor which prints the details of Student
        print(self.name, self.roll)

    class Laptop(): # Initializing the Inner Class
        def __init__(self):
            self.brand = 'HP'
            self.cpu = 'i5'
            self.ram = '16GB'

# Creating the Objects of the Class
s1 = Student('Apurba', 31)
s2 = Student('John', 4)

# Calling the Inner Class Constructor from the Outer/Parent Class Object
print(s1.lap.brand)

# Calling the Inner Class using a new Object
lap1 = s1.lap
print(lap1.brand)

```

Output:

```

@Apurba → Python Programming
\Programs\Python\Python39\python
n\debugpy\launcher' '52983' '-
HP
HP

```

Now, what if we want to Create an Object of the Inner Class outside the Outer Class, we could do that as well. Let's see how to do it,

```

class Student: # Initializing the Class
    def __init__(self, name, roll): # Initializing the constructor with two Instance variables
        self.name = name
        self.roll = roll

    def show(self): # Instance Method acting kind of a Accessor which prints the details of Student
        print(self.name, self.roll)

    class Laptop(): # Initializing the Inner Class
        def __init__(self):
            self.brand = 'HP'
            self.cpu = 'i5'
            self.ram = '16GB'

# Creating the Objects of the Class
s1 = Student('Apurba', 31)
s2 = Student('John', 4)

# Making an Object/Instance of the Inner Class Outside the Outer Class
lap1 = Student.Laptop()

# Getting the Values
print(lap1.brand, lap1.cpu, lap1.ram)

```

Output:

```

@Apurba → Python Programming
\Programs\Python\Python39\python.exe
n\debugpy\launcher' '53022' '--
HP i5 16GB

```

A better Example is shown below of the Same,

```

class Student: # Initializing the Class
    def __init__(self, name, roll): # Initializing the constructor with two Instance variables
        self.name = name
        self.roll = roll
        self.lap = self.Laptop() # Object/Instance of the Inner Class

    def show(self): # Instance Method acting kind of a Accessor which prints the details of Student
        print(self.name, self.roll)
        self.lap.show()

    class Laptop(): # Initializing the Inner Class
        def __init__(self):
            self.brand = 'HP'
            self.cpu = 'i5'
            self.ram = '16GB'

        def show(self): # Although the name is same as of the Parent Class but the existence is
            # different, it has different Memory Location
            print(self.brand, self.cpu, self.ram)

# Creating the Objects of the Class
s1 = Student('Apurba', 31)
s2 = Student('John', 4)

# Using the Object of the Parent Class to get Data of both Parent and Child/Inner Class
s1.show()

```

Output:

```
@Apurba → Python Programming
\Programs\Python\Python39\python\debugpy\launcher' '53088' '-
Apurba 31
HP i5 16GB
```

Inheritance

So, the Concept of inheritance comes when we inherit some features or Properties from a Outer Class into some other Class and we do that to save ourselves some line of Code. Let's See how we do that.

Now, let's say we have a Class A which has two methods namely f1 and f2 and we can call both the methods by using Objects of the Class.

Now lets say we are working on a big Project and we have a Separate Class called Class B and it has it's individual methods and the situation is we want some more methods and co-incidentally that's present in Class A, so the way I can use that in Class B is inheriting the methods of Class A. Let's see how we do that.

```
class A: # Super/Parent Class is Initialized

    def f1(self): # First Instance is Decleared
        print("Feature 1 Working")

    def f2(self): # Second Instance is Decleared
        print("Feature 2 Working")

class B(A): # Child/Inherited Class is Initialized

    def f3(self): # First Instance is Decleared
        print("Feature 3 Working")

    def f4(self): # Second Instance is Decleared
        print("Feature 4 Working")

# Declaring Objects of the Child/Inherited Class
s1 = B()

# Printing the Values of both the Classes just using the Object of the Child/Inherited Class
s1.f1(), s1.f2(), s1.f3(), s1.f4()
```

Output:

```
@Apurba → Python Programming
\Programs\Python\Python39\python\debugpy\launcher' '53199' '-
Feature 1 Working
Feature 2 Working
Feature 3 Working
Feature 4 Working
```

Now, the above Code we just saw is working on **Single Level Inheritance Model**, where we are Inheriting features of Parent Class from Child Class, but what if we have multiple Child Classes inheriting features from the Parent Classes. Let's see how that works. This is Called **Multilevel Inheritance**.

```
class A: # Super/Parent Class is Initialized

    def f1(self): # First Instance is Declared
        print("Feature 1 Working")

    def f2(self): # Second Instance is Declared
        print("Feature 2 Working")

class B(A): # Child/Inherited Class is Initialized

    def f3(self): # First Instance is Declared
        print("Feature 3 Working")

    def f4(self): # Second Instance is Declared
        print("Feature 4 Working")

class C(B): # Child/Inherited Class is Initialized, It is Child Class of Class B which is Child
            # Class of Class A

    def f5(self): # First Instance is Declared
        print("Feature 5 Working")

# Creating Object of Class C and getting the Values of all the Classes
s1 = C(); s1.f1(), s1.f2(), s1.f3(), s1.f4(), s1.f5()
```

Output:

```
@Apurba → Python Programming
\Programs\Python\Python39\python\debugpy\launcher' '53254' '-
Feature 1 Working
Feature 2 Working
Feature 3 Working
Feature 4 Working
Feature 5 Working
```

Now, there are multiple way in which we can inherit features/Methods from classes to classes. But let's see one more.

Here, we have three classes namely Class A, Class B, Class C and the Class C is inheriting features from Class A and Class B. So, let's see the Code for that. It is Called **Multiple Inheritance**.

```
class A: # Super/Parent Class is Initialized

    def f1(self): # First Instance is Decleared
        print("Feature 1 Working")

    def f2(self): # Second Instance is Decleared
        print("Feature 2 Working")

class B(): # Another Super/Parent Class is Initialized

    def f3(self): # First Instance is Decleared
        print("Feature 3 Working")

    def f4(self): # Second Instance is Decleared
        print("Feature 4 Working")

class C(A,B): # Child/Inherited Class is Initialized, It is Child Class of Class B which is
              # Child Class of Class A,B

    def f5(self): # First Instance is Decleared
        print("Feature 5 Working")

# Creating Object of Class B,C and getting the Values available in eacch classes
s1 = B(); s2 = C();s1.f3(), s1.f4();print(); s2.f1(), s2.f2(), s2.f3(), s2.f4(), s2.f5()
```

Output:

```
@Apurba → Python Programming
\Programs\Python\Python39\pyth
n\debugpy\launcher' '53338' '-
Feature 3 Working
Feature 4 Working

Feature 1 Working
Feature 2 Working
Feature 3 Working
Feature 4 Working
Feature 5 Working
```

As you see since Class B is not inheriting features from Class A so it cannot access those Methods but since Class C is inheriting features from both Class A and Class B so it has access of all the methods.

Constructor in Inheritance

Now, here we will be learning about *Constructor in Inheritance* and *MRO (Method Resolution Order)*.

Note: Subclass can access all the features of Super/Parent Class but that's not Vice-versa.

Now, let's imagine a scenario where we have two Classes Class A and Class B, now Class B is Inheriting features from Class A and now we define a Constructor in Class A or here the Super/Parent Class and now if we make an Object of Class B then we can access all the features but if we make an Object of Class A then we just can access features of Class A only and not Class B, now if call Object A or B then it will Constructor of Class A in both the Cases. But if we have a Constructor of Class B itself then we make an Object of B And call it then the Class B constructor will be called. But is there a possibility that I can call both the constructors of Class A and Class B, yes we have for that we use a Special method/function called `Super()`, let's see all of these in Code.


```

class A: # Super/Parent Class is Initialized

    def __init__(self): # Constructor of Parent Class is Initialized
        print("In A Init")

    def f1(self): # First Instance is Declared
        print("Feature 1 Working")

    def f2(self): # Second Instance is Declared
        print("Feature 2 Working")

class B(A): # Child/Inherited Class is Initialized

    def __init__(self): # Constructor of Child Class is Initialized with a Super method
        super().__init__() # Super() method is called for calling the Parent Class Constructor.
        print("In B Init")

    def f3(self): # First Instance is Declared
        print("Feature 3 Working")

    def f4(self): # Second Instance is Declared
        print("Feature 4 Working")

# Declaring Objects of the Child/Inherited Class
s1 = B()

```

Output:

```

@Apurba → Python Programming
ams\Python\Python39\python.exe'
uncher' '53444' '--' 'c:\Users\
In A Init
In B Init

```

Now, let's say we have three classes Class A, Class B and Class C and then Class C inherits features from Class A and Class B, now we have a Constructor for all three classes and we have an Object for Class C then obviously constructor of Class C is called but since it has two Parent Class and if we mention Super in the C Class Constructor then it calls the Constructor of Class A since we have passed A,B in the **MRO (Method Resolution Order)**, which says if you have multiple inheritance it will start from left → right. Let's see the Code.

```

class A: # Super/Parent Class is Initialized

    def __init__(self): # Constructor of Parent Class is Initialized
        print("In A Init")

    def f1(self): # First Instance is Declared
        print("Feature 1 Working")

class B(): # Class is Initialized

    def __init__(self): # Constructor of the second Parent Class is Initialized
        print("In B Init")

    def f3(self): # First Instance is Declared
        print("Feature 3 Working")

class C(A,B): # Another Super/Parent Class is Initialized.
    # Remember here A is Passed before B as a Parent Class. So, here we will follow MRO
    def __init__(self): # Constructor of the Child Class is Initialized
        super().__init__() # Super() method is called for calling the Parent Classes Constructor.
        print("In C Init")

# Creating an Object for the class C and then testing the Constructor calls
s1 = C()

```

Output:

```

@Apurba → Python Programming
ams\Python\Python39\python.exe'
uncher' '53554' '--' 'c:\Users\A
In A Init
In C Init

```

As you can see due to the MRO rule of instruction Class A constructor is being called rather than calling both A and B.

Now, since the Class C has access to all the methods of A and B but super in Class C has access only to the Class A, so if we want to call a method using super in Class C then we can only call methods of Class A and not Class B. Let's see the Code.

```

class A: # Super/Parent Class is Initialized

    def __init__(self): # Constructor of Parent Class is Initialized
        print("In A Init")

    def f1(self): # First Instance is Declared
        print("Feature 1 Working")

class B(): # Class is Initialized

    def f2(self): # First Instance is Declared
        print("Feature 2 Working")

class C(A,B): # Another Super/Parent Class is Initialized.
    # Remember here A is Passed before B as a Parent Class. So, here we will follow MRO
    def __init__(self): # Constructor of the Child Class is Initialized
        super().__init__() # Super() method is called for calling the Parent Classes Constructor.
        print("In C Init")

    def feat(self):
        super().f1()

# Creating an Object for the class C and then trying to call f1 method from Class C using Super()
s1 = C(); s1.feat()

```

Output:

```

@Aporba → Python Programming
ams\Python\Python39\python.exe'
uncher' '53593' '--' 'c:\Users\
In A Init
In C Init
Feature 1 Working

```

See, the contents for `f1()` method of Class A could be called from Class C using `Super()` method.

Note: To represent a Super/Parent Class we use the *Super Method*.

Polymorphism

Now, talking of Polymorphism breaking the word gives us, poly = many, morph = form, so an Object having different forms, or One form many uses so we use this concept many times in Industry Level Works, like *Loose coupling, Dependence Injection, Interfaces*.

We can achieve Polymorphism in Python in 4 ways,

- i) Duck Typing
- ii) Operator Overloading
- iii) Method Overloading
- iv) Method Overriding

Now, let's talk about all of these in detail.

Duck Typing

So, in Python we have Dynamic Typing and it suggests that we will not have to mention Data Type of the Variables. Because, Variables in Python are just names to a certain Memory Address where the Value can Change so really don't have a Specific type for that Variable. So, let's understand it by an Example.

Now imagine you have a Class called Laptop and there is a method in it which takes an Argument namely your IDE which you use, now as it passes the IDE it gets executed now we don't have a method called Execute nor the IDE so we create a class for that IDE and we have the method execute in there. And then it does some work. Now the Idea is that it is not necessary that the IDE has to be only PyCharm or Vs Code it could be something else as well, so it defines that IDE can change it is not fixed provided that the new IDE class has a method named execute in it. That's what is Duck Typing. The Type can change but the Method Remains the Same.

Let's see the Code for that.

```

class Laptop(): # Parent/Super Class gets Initialized

    def mode(self, ide): # Parent Class method gets created with two Instance Variable
        ide.execute() # Now we don't have a Execute Method in the Laptop Class, but since we know
                       # that if the IDE class will have a method called execute with a Self Argument the IDE will
                       # replace that argument and run whatever is Specified in the Method.

class PyCharm(): # IDE class gets Initialized with Execute method.

    def execute(self):
        print("Working")
        print("Compiling")
        print("Ending")

# Declaring the Object of the IDE Class
ide = PyCharm()

# Declaring the Object of the Parent Class
s1 = Laptop()

# Calling the method of the Parent Class
s1.mode(ide)

```

Output:

```

@Apurba → Python Programming
ams\Python\Python39\python.exe
uncher' '60335' '--' 'c:\Users
Working
Compiling
Ending

```

Now, let's see if we change the IDE what happens with the Code,

```

class Laptop(): # Parent/Super Class gets Initialized

    def mode(self, ide): # Parent Class method gets created with two Instance Variable
        ide.execute() # Now we don't have a Execute Method in the Laptop Class, but since we know
                       # that if the IDE class will have a method called execute with a Self Argument the IDE will
                       # replace that argument and run whatever is Specified in the Method.

class PyCharm(): # IDE class gets Initialized with Execute method.

    def execute(self):
        print("Working")
        print("Compiling")
        print("Ending")

class VsCode(): # Initializing the New IDE Class

    def execute(self):
        print("Working");print("Compiling");print("Checking");print("Ending")

# Declaring the Object of the IDE Class
ide = PyCharm()

# Declaring the Object of the Parent Class
s1 = Laptop()

# Changing the IDE will work on the Same Way provided that the IDE class has the method execute
ide = VsCode(); s1.mode(ide)

```

Output:

```
@Apurba → Python Programming
ams\Python\Python39\python.exe'
uncher' '60386' '--' 'c:\Users\
Working
Compiling
Checking
Ending
```

So, as you see the IDE changes to Vs Code now.

That's all about Duck Typing in Python.

Operator Overloading

So, now we know that we have Operators in Python. We have different of them, we have int, float, bool, etc. Now, let's say you want to add two number so you will simply just add them with the '+' symbol and you get the results and that's really sugary like Synthetic Sugar, where everything is done for you. But, behind the scene something else is happening, so as we all know in Python Everything is an Object and these datatypes are individual classes in itself. So, they are of <class 'int'>, <class 'float'>, <class 'bool'> and etc. So, they should also have some methods. So, in case of Addition when you add two numbers what happens behind the scene the 'int' class calls the "__add__" method in Python. Now, add also takes exactly two arguments as you have added the two numbers and gives you the same result. Let's see it in action ..

```
a = 5 # Declare Variable a
b = 7 # Declare variable b

print(a + b) # Add the following in Conventional Method

print(int.__add__(a,b)) # Does the Same thing of adding but shows us that how the adding is done.
```

Output:

```
@Apurba → Python Programming
ams\Python\Python39\python.exe'
uncher' '60439' '--' 'c:\Users\
12
12
```

Same goes for String as well, Let's see

```
a = '5' # Declare Variable a
b = '7' # Declare variable b

print(a + b) # Add the following in Conventional Method

print(str.__add__(a,b)) # Does the Same thing of adding but shows us that how the adding is done.
```

Output:

```
@Apurba → Python Programming
ams\Python\Python39\python.exe'
uncher' '60459' '--' 'c:\Users\
57
57
```

So, as we see the moment we use '+' in Python behind the scene "`__add__()`" is getting called, the moment we use '-' in Python behind the Scene "`__sub__()`" is getting called and so on. So, we have different methods for all types of Operators and they are being called "Magic Methods".

Now, let's explore the concept of these Operators in a Class defined by us.

So, say we have a Class Student and a constructor taking two values namely the marks or any int value. And we have two Objects of the same Class. Now, if we add the two Objects, let's see what we get,

```
class Student: # Parent/Super class getting Initialized

    def __init__(self, m1, m2):
        self.m1 = m1
        self.m2 = m2

s1 = Student(58, 65)
s2 = Student(50, 69)

s3 = s1 + s2
```

The Output comes like,

```
@Apurba → Python Programming c;; cd 'c:\Users\Apurba\Desktop\Text Folder\Python Programming'; & 'C:\Users\Apurba\AppData\Local\Microsoft\WindowsApps\Python\Python39\python.exe' 'c:\Users\Apurba\.vscode\extensions\ms-python.python-2021.5.842923320\python\python.exe' 'c:\Users\Apurba\Desktop\Text Folder\Python Programming\OOPs Tutorials.py'
Traceback (most recent call last):
  File "c:\Users\Apurba\Desktop\Text Folder\Python Programming\OOPs Tutorials.py", line 645, in <module>
    s3 = s1 + s2
TypeError: unsupported operand type(s) for +: 'Student' and 'Student'
```

It says *Unsupported Operand Types* for +: 'Student' and 'Student', but then we know that when we use '+' the "`__add__()`" method is called in behind. But in our class we don't have that method, so we need to define that ourselves in our class, and that's what we called *Operator Overloading*, where we Overload the Operator by defining our own Magic Method related to it. Let's see the Code for that,

```
class Student: # Parent/Super class getting Initialized

    def __init__(self, m1, m2): # The Parent Class Constructor is getting Declared with two Instance
        self.m1 = m1
        self.m2 = m2

    def __add__(self, other): # The Student class Magic Method is getting declared
        m1 = self.m1 + other.m1 # Adds the first value of the first and second Object respectively
        m2 = self.m2 + other.m2 # Adds the Second value of the first and second Object respectively
        s3 = Student(m1, m2) # Updates the Constructor with the new m1, m2 values.
        return s3 # Returns the upadted values of m1, m2 to the variable and makes it a new Student
        # Object

# Objects of the Parent/Super Class are getting declared
s1 = Student(58, 65)
s2 = Student(50, 69)

# Adds two Objects and finally gets the status of a Student Object with two elements each of which
# is the addition of the first number of two Objects and the Second number of two Objects
s3 = s1 + s2
print(s3.m1, s3.m2) # Prints the First and Second element of the new Student Object respectively
```

The Output of it is,

```
@Apurba → Python Programming
ams\Python\Python39\python.exe
uncher' '49285' '--' 'c:\Users\
108 134
```

Let's see an Example where we compare two Objects of a Class.


```

class Student: # Parent/Super class getting Initialized

    def __init__(self, m1, m2): # The Parent Class Constrcutor is getting Decleared with two Instance
        self.m1 = m1
        self.m2 = m2

    def __gt__(self, other): # The Student class Magic Method is getting decleared
        m1 = self.m1 + self.m2 # Adds the first value of the first Object
        m2 = other.m1 + other.m2 # Aadds the Second value of the second Object
        if m1 > m2: # Checks if the sum of the values of the first Object is greater than the other
            return True # return True if yes
        else:
            return False # else returns False

# Objects of the Parent/Super Class are getting decleared
s1 = Student(1, 20)
s2 = Student(50, 69)

if s1 > s2: # checks which Object is greater
    print("S1 is Greater")
else:
    print("S2 is Greater")

```

And, the Output of the Code is,

```

@Apurba → Python Programming
ams\Python\Python39\python.exe
uncher' '61976' '—' 'c:\Users\
S2 is Greater

```

There is one more thing to Look at. Now, say we have a Variable having Value 9 and if we print the Variable it prints it's value and not the Memory Address of that Variable. But when we print an Object of class it shows us the Memory Address. What happens is that underlying a method named “__str__()” is getting called and it is printing the mentioned values. Now, we don't want to see the Memory Address when we prints the Object. So, we will create our own “__str__()” method i.e overload the existing method. Let's see that in Code.

```

class Student: # Parent/Super class getting Initialized

    def __init__(self, m1, m2): # The Parent Class Constrcutor is getting Decleared with two Instance
        self.m1 = m1
        self.m2 = m2

    def __gt__(self, other): # The Student class Magic Method is getting decleared
        m1 = self.m1 + self.m2 # Adds the first value of the first Object
        m2 = other.m1 + other.m2 # Aadds the Seconds value of the second Object
        if m1 > m2: # Checks if the sum of the values of the first Object is greater than the other
            return True # return True if yes
        else:
            return False # else returns False

# Objects of the Parent/Super Class are getting decleared
s1 = Student(1, 20)
s2 = Student(50, 69)

a = 9 # Variable with a Memory gets decleared
print(a) # prints the value of the Variable, exactly it does a.__str__()

print(s1.__str__()) # Prints in the same way as anything gets print undelying, which gets the Address

```

Output:

```

@Apurba → Python Programming c:; cd 'c:\Users\A
ams\Python\Python39\python.exe' 'c:\Users\Apurba\
uncher' '62036' '--' 'c:\Users\Apurba\Desktop\Text
9
<__main__.Student object at 0x000001EFE1C9E970>

```

Now, we want to Overload/Override the existing Operator and define our own such that if we print an Object, it's values gets print.

Let's see that,

```

class Student: # Parent/Super class getting Initialized

    def __init__(self, m1, m2): # The Parent Class Constrcutor is getting Decleared with two Instance
        self.m1 = m1
        self.m2 = m2

    def __gt__(self, other): # The Student class Magic Method is getting decleared
        m1 = self.m1 + self.m2 # Adds the first value of the first Object
        m2 = other.m1 + other.m2 # Aadds the Second value of the second Object
        if m1 > m2: # Checks if the sum of the values of the first Object is greater than the other
            return True # return True if yes
        else:
            return False # else returns False

    def __str__(self):
        return self.m1, self.m2

# Objects of the Parent/Super Class are getting decleared
s1 = Student(1, 20)
s2 = Student(50, 69)

a = 9 # Variable with a Memory gets decleared
print(a) # prints the value of the Variable, exactly it does a.__str__()

print(s1.__str__()) # Prints in the same way as anything gets print undelying, which gets the Address

```

Let's See the Output,

```

@Apurba → Python Programming
ams\Python\Python39\python.exe
uncher' '62054' '---' 'c:\Users\
9
(1, 20)

```

But, if you see we explicitly mention the “__str__()” method in the print statement, if we remove that it will say ***TypeError: __str__ returned non-string (type tuple)***, now since we can return only str values from str magic method, we need to return it in the form of a String. Let's see that in Code.

```

class Student: # Parent/Super class getting Initialized

    def __init__(self, m1, m2): # The Parent Class Constructor is getting Declared with two Instance
        self.m1 = m1
        self.m2 = m2

    def __gt__(self, other): # The Student class Magic Method is getting declared
        m1 = self.m1 + self.m2 # Adds the first value of the first Object
        m2 = other.m1 + other.m2 # Adds the Second value of the second Object
        if m1 > m2: # Checks if the sum of the values of the first Object is greater than the other
            return True # return True if yes
        else:
            return False # else returns False

    def __str__(self):
        return f"{self.m1}, {self.m2}" # return the Values of the Object in a String Form

# Objects of the Parent/Super Class are getting declared
s1 = Student(1, 20)
s2 = Student(50, 69)

a = 9 # Variable with a Memory gets declared
print(a) # prints the value of the Variable, exactly it does a.__str__()

print(s1) # Prints in the same way as anything gets print undelying, which gets the Address

```

And the Output will be like,

```

@Apurba → Python Programming
ams\Python\Python39\python.exe'
uncher' '62114' '--' 'c:\Users\
9
1, 20

```

So, these were some ways with which we perform Operator Overloading in Python.

Method Overloading & Overriding

Method Overloading

Now, Method Overloading is a process where let's say we have two Methods of same name with different Parameters or Arguments. Like say we have a Student Class with average method one taking two params and the other taking three params. That's what we call Method Overloading. But in Python we generally don't have this Concepts.

Method Overriding

Now, Method Overriding is a Process where let's say we have two methods of same name with same Parameters or Arguments. Like say we have Student Class with avg method one taking 2 params and the other as well. Now Obviously we can't have something like that in Same Class but if we have a concept of Inheritance where the Parent/Super and the Child Class has the same name method with same number of params. That's what we call as Method Overriding.

Let's start with Method Overloading first,

Now, imagine a scenario where we have a method called sum in a Class called Student and we add number using that method. Now, by default we pass 2 values and it accepts that, but then what if I send 3 values or maybe 1 value, So in other languages we can use Method Overloading we can Override the methods. But, Python does not support that so what we can do is we can get the default values of the params to None and then check which param is not None and accordingly return the values. Let's see how it's done in Code,

```
class Student: # Parent Class Student is Initialized

    def sum(self, a = None, b = None, c = None): # Prent Class Method Sum is declared with 4 Instance
        s = 0 # Initializes the default value of the sum variable to 0

        if a != None and b != None and c != None: # Checks if all three are not Nonw
            s = a + b + c
        elif a != None and b != None: # Checks if the first two Parameters are not None
            s = a + b
        else: # Checks if only the first Parameter is not None
            s = a

        return s # returns the Sum according to the Number of Variable Passed into the method

# Object of the Student Class gets deleared
s1 = Student()

# Passing and Printing different values and results respectively, This is a trick we used to sort
# # of achieve Method Overloading in Python Indirectly
print("Two values", s1.sum(4, 5)) # Passed Two Values
print("Three values", s1.sum(4, 5, 8)) # Passed Three Values
```

Output:

```
@Apurba → Python Programming
ams\Python\Python39\python.exe'
uncher' '62269' '--' 'c:\Users\A
Two values 9
Three values 17
```

Let's start with Method Overriding now,

Now, imagine a scenario where we have a Parent Class and it has a method called Show. Now I created a Child Class and it has nothing in it. So if I were to make an Object of the Class Child, and with the help of that Object I call Show method it would show me Error. Let's see that in Code,

```
class Parent: # Parent/Super Class gets Initialized
    def show(self): # Parent Class Method with 1 Instance Param gets declared
        print("In Class Parent")

class Child(): # Child Class gets Initialized
    pass

# Object of the first Class gets Declared
s1 = Parent()

# Call the Show method with the help of the Parent Object
s1.show()

# Object of Child class gets Declared
s2 = Child()

# Call the Show method with the help of Child Object
s2.show()
```

And the Output is pretty Obvious that we will get an error in the `s2.show()` as we don't have any methods written in Our Child Class.

```
@Apurba → Python Programming c:; cd 'c:\Users\Apurba\
ams\Python\Python39\python.exe' 'c:\Users\Apurba\.vscode
uncher' '62355' '--' 'c:\Users\Apurba\Desktop\Text Folders\Python Programming'
In Class Parent
Traceback (most recent call last):
  File "c:\Users\Apurba\Desktop\Text Folders\Python Programming", line 1, in <module>
    s2.show()
AttributeError: 'Child' object has no attribute 'show'
```

As expected we got that,

Now, let's say the case where the Child class is inheriting from Parent Class. Then if we write the same code it would show use the content of

the show method in Parent Class as we have inherited all features of it. Let's see the Code.

```
class Parent: # Parent/Super Class gets Initialized
    def show(self): # Parent Class Method with 1 Instance Param gets declared
        print("In Class Parent")

class Child(Parent): # Child Class gets Initialized by Inheriting from Parent Class
    pass

# Object of the first Class gets Declared
s1 = Parent()

# Call the Show method with the help of the Parent Object
s1.show()

# Object of Child class gets Declared
s2 = Child()

# Call the Show method with the help of Child Object
s2.show()
```

And the Output is we will get "In Class Parent" two times.

```
@Apurba → Python Programming
ams\Python\Python39\python.exe'
uncher' '62376' '--' 'c:\Users\A
In Class Parent
In Class Parent
```

Now, what if we had our own show method in the Child Class. Let's see the Code and what the Output would be,

```

class Parent: # Parent/Super Class gets Initialized
    def show(self): # Parent Class Method with 1 Instance Param gets declared
        print("In Class Parent")

class Child(Parent): # Child Class gets Initialized by Inheriting from Parent Class
    def show(self): # Child Class Instance with 1 Instance Param gets declared
        print("In Class Child")
    pass

# Object of the first Class gets Declared
s1 = Parent()

# Call the Show method with the help of the Parent Object
s1.show()

# Object of Child class gets Declared
s2 = Child()

# Call the Show method with the help of Child Object
s2.show()

```

Output:

```

@Apurba → Python Programming
ams\Python\Python39\python.exe'
uncher' '61140' '--' 'c:\Users\A
In Class Parent
In Class Child

```

So, as we have our own `show()` method in Child Class now. So, the existing `show()` method gets Override which we inherited from the Parent Class and now the `show` method of child class is shown as we call the method using the Object of Child Class. This is what we call **Method Overriding in Python**.

