# Classes and Objects

Introduction

Problem solving using Advance Python

B.Tech 2$^{nd}$ semester

# Content

- Introduction: Python classes and objects
- User-defined classes
- Encapsulation
- Data hiding
- Class variable and Instance variables
- Instance methods
- Class methods
- Static methods
- Constructor in python
- Parametrized constructor
- Magic methods in python
- Object as an argument
- Instances as Return Values, namespaces

# Course Objective

- To learn the Object Oriented Concepts in Python.
- To learn the concept of reusability through inheritance and polymorphism.
- To impart the knowledge of functional programming.
- To learn the concepts of designing graphical user interfaces.
- To explore the knowledge of standard Python libraries.

# Course Outcome

**At the end of course student will be able:**

- Define classes and create instances in python.
- Implement concept of inheritance and polymorphism using python.
- Implement functional programming in python.
- Create GUI based Python application.
- Apply the concept of Python libraries to solve real world problems.

# CO-PO and PSO Mapping

| Course Outcome ( CO) | At the end of course , the student will be able : | Bloom's Knowledge Level (KL) |
|---|---|---|
| CO1 | Define classes and create instances in python. | K1, K2 |
| CO2 | Implement concept of inheritance and polymorphism using python. | K3 |
| CO3 | Implement functional programming in python. | K2 |
| CO4 | Create GUI based Python application. | K3 |
| CO5 | Apply the concept of Python libraries to solve real world problems. | K3, K6 |

# CO-PO and PSO Mapping

## Mapping of Course Outcomes and Program Outcomes:

| Programming in Python | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CO.K | PO1 | PO2 | PO 3 | PO 4 | PO5 | PO6 | PO 7 | PO8 | PO 9 | PO10 | PO1 1 | PO12 |
| CO1 | | | | | | | | | | | | |
| CO2 | | | | | | | | | | | | |
| CO3 | | | | | | | | | | | | |
| CO4 | | | | | | | | | | | | |
| CO5 | | | | | | | | | | | | |
| Average | | | | | | | | | | | | |

## Mapping of Course Outcomes and Program Specific Outcomes:

| Programming in Python | | | | |
|---|---|---|---|---|
| CO.K | PSO1 | PSO2 | PSO3 | PSO4 |
| CO1 | | | | |
| CO2 | | | | |
| CO3 | | | | |
| CO4 | | | | |
| CO5 | | | | |
| Average | | | | |

# Unit Objective

# Prerequisite and Recap

- Python conditional statements:

  -If-else ,elif and nested if else

- Python loop statement:

  -while loop

  -for in loop

- Python Data structures:

  -list, tuple ,string ,set and dictionary

Let's

Recap

# Introduction : OOPs concept

- Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviours are bundled into individual objects.
- object-oriented programming is an approach for modelling concrete, real-world things, like cars, as well as relations between things, like companies and employees, students and teachers, and so on.
- In this approach , data and functions are combined to form a class.

For example:

| Object | Data or attribute | Functions/methods |
|---|---|---|
| Person | Name, age ,gender | Speak(),walk() etc. |
| Polygon | Vertices ,border ,color | Draw(),erase() etc. |
| Computer | Brand, resolution,price | Display(),printing() etc. |

# Introduction : Python Classes and Objects

## Python Classes and Objects

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.
- A class creates  a new type and object is an instance of the class.
- The python library is based on the concept of classes and objects.

# Introduction : Python Classes and Objects

## Create a Class

•Starts with a keyword class followed by the class_name and a colon:

•Similar to function definition.

Syntax:

```
class  class_name:
              <statement-1>
               <statement-2>
              ……
               <statement-n>
```

**Example:**

Create a class named MyClass, with a property named x:

```
class MyClass:
               x = 5
print(MyClass)
```

**Output:**

```
<class '__main__.MyClass'>
```

# Introduction : Python Classes and Objects

## Create Object:

- Creating an object or instance of a class is known as class instantiation
- We can use the class name to create objects:

**Syntax:**

```
Object_name =class_name()
# using this syntax ,an empty object of a class is created.
```

## Accessing a class member:

- The object can access class variables and class methods using dot(.) operator.

## Example:

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

**Output:**

```
5
```

# Introduction : Python Classes and Objects

## The __init__() Method:

- The example above are classes and objects in their simplest form, and are not really useful in real life applications.
- All classes have a method called __init__(), which is always executed when the class is being initiated.
- Use the __init__() Method to assign values to object properties, or other operations that are necessary to do when the object is being created:
- The __init__() Method is called automatically every time the class is being used to create a new object.
- The __init__() method is useful to initialize the variables of the class object.

# Introduction : Python Classes and Objects

## Example:

Create a class named Person, use the __init__() method to assign values for name and age:

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

**Output:**

```
John
36
```

**Another way:**

```
class Person:
    def __init__(self, name, age):
        print(“In class method”)
        self.name = name
        self.age = age
       print(“the values are”,name,age)
p1=Person(“john”,36)
```

**Output:**

```
In class method
The values are:
John 36
```

# Class Variables and Instance Variables

Variables are essentially symbols that stand in for a value we're using in a program. Object-oriented programming allows for variables to be used at the class level or the instance level.

**Class Variables**

- Declared inside the class definition (but outside any of the instance methods).
- They are not tied to any particular object of the class, hence shared across all the objects of the class.
- Modifying a class variable affects all objects instance at the same time.

# Class Variables and Instance Variables

**Instance Variable or object variable** —

- Object variable or instance variable owned by each object created for a class.
- Declared inside the constructor method of class (the __init__ method).
- They are tied to the particular object instance of the class, hence the contents of an instance variable are completely independent from one object instance to the other.
- The object variable is not shared between objects

# Class Variables and Instance Variables

```
class Car:
        wheels = 4    # <- Class variable
        def __init__(self, name):
            self.name  = name    # <- Instance variable
```

Above is the basic, no-frills *Car* class defined. Each instance of it will have class variable *wheels* along with the instance variable *name*.

**Let’s instantiate the *class* to access the variables:**

```
mercedes=Car("mercedes")
print(mercedes.wheel)# access of class variable through object
print(Car.wheel) )# access of class variable through class
print(mercedes.name) )# access of instance variable through object
```

# Difference between Class Variables and Instance Variables

- Class variable:

1. Class variable owned by class.
2. All the objects of the class will share the class variable.
3. Any change made to the class variable will be reflected in all other objects.

- Instance variable:

1. Instance variable owned by object.
2. The instance variable is not shared between objects.
3. Any change made to the Instance variable will not be reflected in all other objects.

**Important point:**

1. **Generally , Class variable is used to define constant with a particular class or provide default attribute.**
2. **Another use of class variable is to count the number of objects created.**

# Program to demonstrate the use of class or static variable

```
class Fruits(object):
    count = 0
def __init__(self, name, count):
    self.name = name
    self.count = count
Fruits.count = Fruits.count + count

def main():
    apples = Fruits("apples", 3);
    pears = Fruits("pears", 4);
    print (apples.count)
    print (pears.count)
    print (Fruits.count)
    print (apples.__class__.count) # This is Fruit.count
    print (type(pears).count) # So is this
if __name__ == '__main__':
main()
```

Output:

```
3
4
7
7
7
```

# Demonstrate the use of variable defined on the class level

```
class example:
        staticVariable = 9 # Access through class

print (example.staticVariable) # Gives 9

#Access through an instance
instance = example()
print(instance.staticVariable) #Again gives 9

#Change within an instance
instance.staticVariable = 12
print(instance.staticVariable) # Gives 12
print(example.staticVariable) #Gives 9
```

Output:

```
9
9
12
9
```

# Introduction : Python Classes and Objects(Self Parameter)

**The self Parameter or argument:**

- The self argument refers to the object itself.
- The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
- It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class.
- We can have .__init__() any number of parameters, but the first parameter will always be a variable called *self*. When a new class instance is created, the instance is automatically passed to the self parameter in .__init__() so that new **attributes** can be defined on the object.

# Introduction : Python Classes and Objects(Self Parameter)

- Creating object using self argument:

Example:

```
Class Myclass:
        a=10
        def func(self):
                return “using self”
# instance an object
ob=Myclass()
print(ob.func())  # output : using self
print(Myclass.func(ob))  # output : using self
```

# Introduction : Python Classes and Objects

## Example:

Use the words mysillyobject and abc instead of self:

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

    def myfunc(abc):
        print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

**Output:**

```
Hello my name is John
```

# User Defined Classes or problems:

**Q1.program modifying a mutable type attributes:**

```
Class Number:
        evens=[]
        odd=[]
        def __init__(self,num):
                self.num=num
                if num%2==0:
                        Number.evens.append(num)
                else:
                        Number.odds.append(num)
N1=Number(21)
N1=Number(32)
N1=Number(43)
N1=Number(54)
N1=Number(65)
print(“even Numbers are:”,Number.evens)
print(“odd Numbers are:”,Number.odds)
```

Output:

```
Even Numbers are: [32,54]
Odd Numbers are: [21,43,65]
```

# User Defined Classes or problems:

**Q1. Write a program that uses class to store the name and marks of students . Use list to store the marks in three subjects.**

**Q2. Write a program with class Employee that keeps a track of the number of employees in an organization and also stores their name , designation and salary details.**

**Q3. Write a program that has a class Circle . Use a class variable to define the value of constant PI. Use this class variable to calculate are and circumference of a circle with specified radius.**

**Q4. Write a program that has a class student that stores roll number, name and marks(in three subjects) of the students. Display the information(roll number, name and total marks)stored about the student.**

**Q5.Write a class that stores a string and all its status details such as number of uppercase characters , vowels ,consonants ,space etc.**

# Python Encapsulation

## Python Encapsulation

- Using OOP in Python, we can restrict access to methods and variables.
- This prevents data from direct modification which is called encapsulation.
- In Python, we denote private attributes using underscore as the prefix i.e single _ or double __.

# Python Encapsulation

**Example : Data Encapsulation in Python**

```
class Computer:
    def __init__(self):
        self.__maxprice = 900
    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))
    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()
```

# Python Encapsulation

## Example : Data Encapsulation in Python

```
# change the price
c.__maxprice = 1000
c.sell()
# using setter function
c.setMaxPrice(1000)
c.sell()
```

**Output:**

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

# Python Encapsulation

## Example : Data Encapsulation in Python

- In the above program, we defined a Computer class.
- We used __init__() method to store the maximum selling price of Computer. We tried to modify the price. However, we can't change it because Python treats the __maxprice as private attributes.
- As shown, to change the value, we have to use a setter function i.e setMaxPrice() which takes price as a parameter.

# Data Hiding

**Data Hiding:**

- An object’s attributes may or may not be visible outside the class definition.
- In Python, we use double underscore before the attributes name to make them inaccessible/private or to hide them.
- The attributes with prefix double underscore not visible outside the class.

# Data Hiding

**Example:**

```
class MyClass:
    __hiddenVar = 12
    def add(self, increment):
        self.__hiddenVar += increment
        print (self.__hiddenVar)
myObject = MyClass()
myObject.add(3)
myObject.add (8)
print(Myobject.__hiddenVar)# Error as not accessible outside
print (myObject._MyClass__hiddenVar)
```

**Output:**

```
15
23
23
```

# Data Hiding

**Example:**

```
class Car:
   __maxspeed  = 0
   __name = ""
      def __init__(self):
           self.__maxspeed  = 200
          self.__name  = "Supercar"
      def drive(self):
          print('driving. maxspeed ' + str(self.__maxspeed))
redcar = Car()
redcar.drive()
redcar.__maxspeed  = 10  # will not change variable because its private
redcar.drive()
```

*Note:* To change the value of a private variable, a setter method is used

# Data Hiding

**Example:**

```
class Car:
    __maxspeed = 0
    __name = ""
       def __init__(self):
             self.__maxspeed = 200
            self.__name = "Supercar"
       def drive(self):
            print('driving. maxspeed ' + str(self.__maxspeed))
       def setMaxSpeed(self,speed):
            self.__maxspeed = speed
redcar = Car()
redcar.drive()
redcar.setMaxspeed = 320  # will change variable
redcar.drive()
```

# Introduction to Methods:

## Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

**Example:**

Insert a function that prints a greeting, and execute it on the p1 object:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def myfunc(self):
        print("Hello my name is " + self.name)
p1 = Person("John", 36)
p1.myfunc()
```

**Output:**

```
Hello my name is John
```

# Instance methods, Class method, static methods

Python's data model, Python offers three types of methods namely *instance*, *class* and *static methods.*

***Instance or object methods:***

- They are most widely used methods.
- Instance method receives the instance of the class as the first argument, which by convention is called self, and points to the instance of class
- However it can take any number of arguments. Using the self parameter, we can access the other attributes and methods on the same object and can change the object state.
- Also, using the self.__class__ attribute, we can access the class attributes, and can change the class state as well.
- **Therefore, instance methods gives us control of changing the object as well as the class state.**
- A built-in example of an instance method is str.upper()

```
>>> "welcome".upper() # called on the str object
'WELCOME'
```

# Example :Instance Methods

Q. Write a program to deposit or withdraw money in a bank account.

```
Class Account:
    def __init__(self):
        self.balance=0
        print(“New Account Created”)
    def deposit(self):
        amount=float(input(“enter amount to deposit:”))
        self.balance+=amount
    def withdraw(self):
        amount=float(input(“enter amount to withdraw:”))
        if amount> self.balance:
            print(“Insufficient balance”)
        else:
            self.balance-=amount
            print(New Balance”, self.balance)
    def enquiry(self):
        print(“Balance”,self.balance)
```

```
account=Account()
account.deposit()
account.withdraw()
account.enquiry()
```

**Output:**

```
New Account created
enter amount to deposit: 1000
New Balance:1000.000000
enter amount to withdraw:25.23
New Balance:974.770000
Balance:974.770000
```

# Decorators Before :Class and static Methods

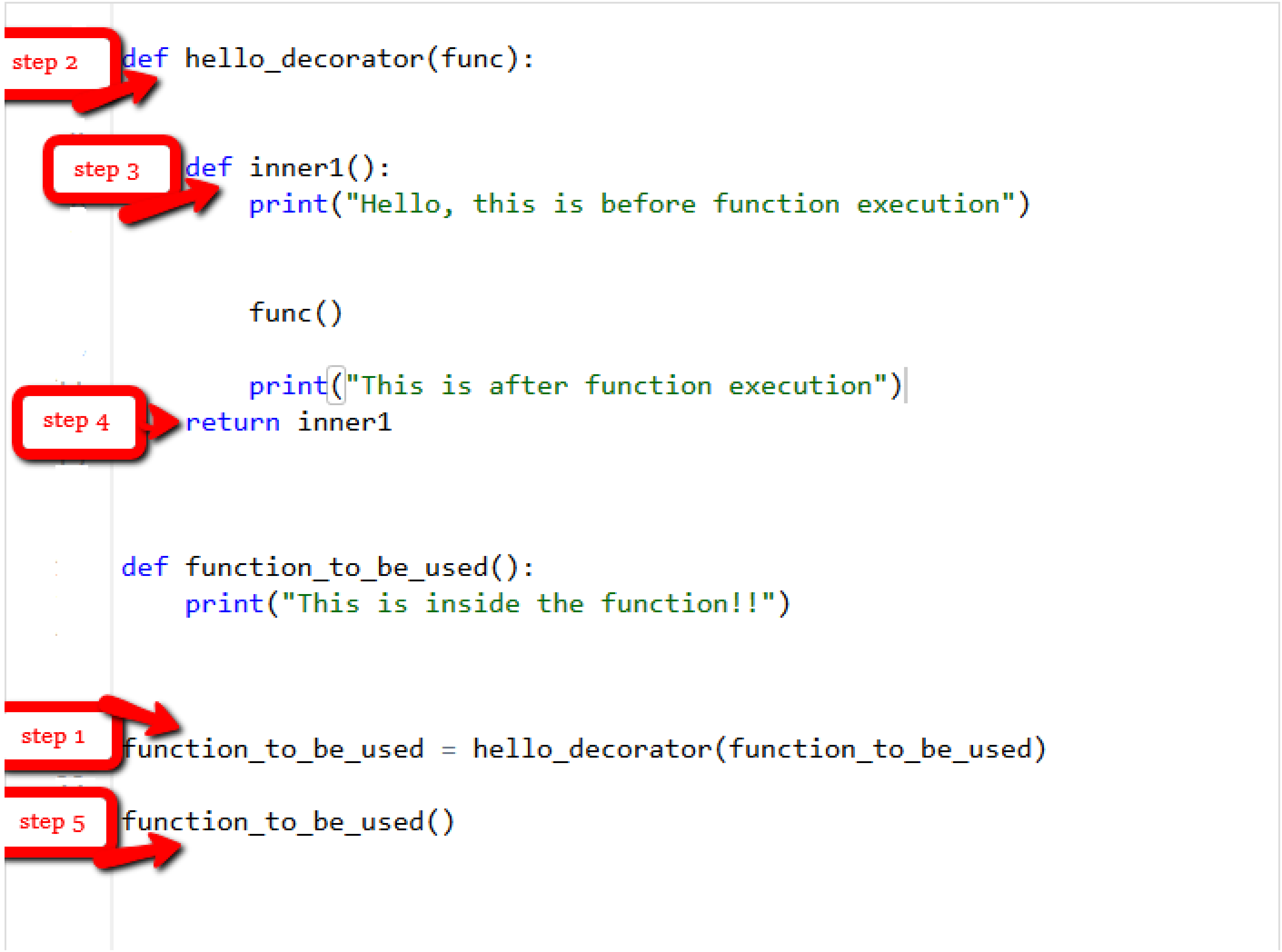
- Decorators are very powerful and useful tool in Python since it allows programmers to modify the behaviour of function or class.
- Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it.
- Before diving deep into decorators let us understand some concepts that will come in handy in learning the decorators.

# Decorators Before :Class  and static Methods

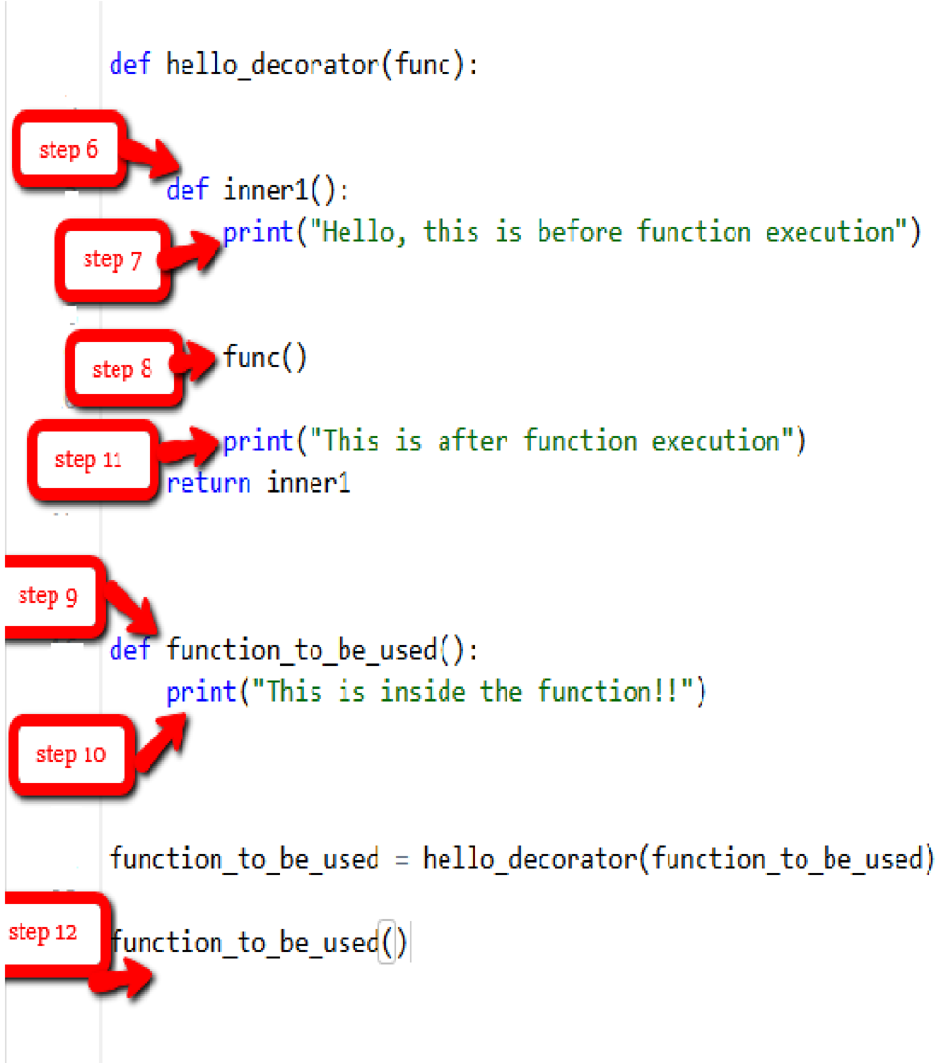
```
# defining a decorator
def hello_decorator(func):
    # inner1 is a Wrapper function in
    # which the argument is called
    # inner function can access the outer local
    # functions like in this case "func"
    def inner1():
        print("Hello, this is before function execution")
        # calling the actual function now
        # inside the wrapper function.
        func()
        print("This is after function execution")
    return inner1
```

```
# defining a function, to be called inside wrapper

def function_to_be_used():
        print("This is inside the function !!")

# passing 'function_to_be_used' inside the
# decorator to control its behavior

function_to_be_used=hello_decorator(function_to_be_used)

# calling the function

function_to_be_used()
```

# Decorators Before :Class  and static Methods



```
def hello_decorator(func):

    def inner1():
        print("Hello, this is before function execution")


        func()

        print("This is after function execution")
    return inner1



def function_to_be_used():
    print("This is inside the function!!")



function_to_be_used = hello_decorator(function_to_be_used)

function_to_be_used()
```



```
def hello_decorator(func):


    def inner1():
        print("Hello, this is before function execution")


        func()

        print("This is after function execution")
    return inner1



def function_to_be_used():
    print("This is inside the function!!")



function_to_be_used = hello_decorator(function_to_be_used)

function_to_be_used()
```

# Decorators Before :Class  and static Methods

```
# defining a decorator
def hello_decorator(func):
        def inner1():
        print("Hello, this is before function execution")
        func()
        print("This is after function execution")
    return inner1
```

```
# defining a function, to be called inside wrapper

@hello_decorator
def function_to_be_used():
        print("This is inside the function !!")

# passing 'function_to_be_used' inside the
# decorator to control its behavior


# calling the function

function_to_be_used()
```

# Class Methods

- A class method accepts the class as an argument to it which by convention is called cls. It take the cls parameter, which points to the class instead of the object of it. It is declared with the @classmethod decorator.
- Class methods are bound to the class and not to the object of the class. **They can alter the class state that would apply across all instances of class but not the object state.**

**Note: 1.class methods are called by class .**

**2. First argument of the class method cls , not the self**

- Syntax for Class Method.

```
class my_class:
   @classmethod
  def function_name(cls, arguments):
      #Function Body
      return value
```

# Class Methods

**#Normal Calling**

```
class Rectangle:
        def __init__(self,length,breadth):
                self.length=length
                self.breadth=breadth
        def area(self):
                return self.length * self.breadth
        def Square(cls,side):
                return cls(side,side)
Rectangle.Square=classmethod(Rectangle.Square)
S=Rectangle.Square(10)
print(“Area=“,S.area())
```

**Output:**

**Area=100**

**# Using Decorators**

```
class Rectangle:
        def __init__(self,length,breadth):
                self.length=length
                self.breadth=breadth
        def area(self):
                return self.length * self.breadth
        @classmethod
        def Square(cls,side):
                return cls(side,side)
S=Rectangle.Square(10)
print(“Area=“,S.area())
```

**Output:**

**Area=100**

# Static Methods

- A static method is marked with a @staticmethod decorator to flag it as *static*.
- It does not receive an implicit first argument (neither self nor cls).
- It can also be put as a method that "does't know its class".
  Hence a static method is merely attached for convenience to the class object.
- A static method can be called either on the class or an instance.
- **Hence static methods can neither modify the object state nor class state. They are primarily a way to namespace our methods.**
- Syntax for Static Method

```
class my_class:
    @staticmethod
    def function_name(arguments):
        #Function Body
        return value
```

# Static Methods

```
class Choice:
        def __init__(self, subjects):
                self.subjects=subjects
        @static method
        def validate_subject(subjects):
                if “CSA” in subjects:
                        print(“This option is no longer available”)
                else:
                        return True
Subjects=[“DS”,”CSA”,”Foc”,”OS”,”TOC”]
if all(Choice.validate_subjects(i) for I in subjects):
   ch=Choice(subjects)
   print(“You have been alloted the subjects:”,subjects)
```

**OUTPUT:**

```
This option is no longer available.
```

# What are the differences between Classmethod and StaticMehtod?

| Class Method | Static Method |
|---|---|
| The class method takes cls (class) as first argument. | The static method does not take any specific parameter. |
| Class method can access and modify the class state. | Static Method cannot access or modify the class state. |
| The class method takes the class as parameter to know about the state of that class. | Static methods do not know about class state. These methods are used to do some utility tasks by taking some parameters. |
| @classmethod decorator is used here. | @staticmethod decorator is used here. |

# Constructor (CO1)

- Constructors are generally used for instantiating an object.
- The task of constructors is to initialize(assign values) to the data members of the class when an object of class is created.
- In Python the **__init__()** method is called the constructor and is always called when an object is created.
- def __init__(self):

  # body of the constructor

# Constructor (CO1)

Types of constructors :

**Default constructor** :The default constructor is simple constructor which doesn’t accept any arguments.

It’s definition has only one argument which is a reference to the instance being constructed.

**Parameterized constructor** :Constructor with parameters is known as parameterized constructor.

The parameterized constructor take its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

# Constructor (CO1)

```
class default_Constructor:
# default constructor
    def __init__(self):
        self.dC = “Hello Python Default Constructor"
# a method for printing data members
    def print_ default_Constructor(self):
        print(self.dC)
# creating object of the class
obj = default_Constructor()
# calling the instance method using the object obj
obj. print_ default_Constructor()
```

# Constructor (CO1)

```
class Addition:
    first = 0
    second = 0
    answer = 0
# parameterized constructor
    def __init__(self, f, s):
        self.first = f
        self.second = s

    def display(self):
        print("First number = " + str(self.first))
        print("Second number = " + str(self.second))
        print("Addition of two numbers = " + str(self.answer))
    def calculate(self):
        self.answer = self.first + self.second
```

```
# creating object of the class
# this will invoke parameterized constructor
obj = Addition(1000, 2000)

# perform Addition
obj.calculate()

# display result
obj.display()
```

# Magic Methods

- Magic methods in Python are the special methods that start and end with the double underscores.
- They are also called **dunder** methods.
- Magic methods are not meant to be invoked directly by you, but the invocation happens internally from the class on a certain action.

For example, when you add two numbers using the + operator, internally, the **__add__()** method will be called.

# Magic Methods

- Built-in classes in Python define many magic methods.
- Use the **dir()** function to see the number of magic methods inherited by a class.

For example, the following lists all the attributes and methods defined in the int class.

```
>>> dir(int)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
'__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__',
'__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__',
'__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__',
'__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__',
'__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__',
'__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__',
'__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate',
'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

# Magic Methods

the __add__ method is a magic method which gets called when we add two numbers using the + operator.

Consider the following example.

```
>>> num=10
>>> num + 5
15
>>> num.__add__(5)
15
```

# Magic Methods

## __new__() method

Languages such as Java and C# use the new operator to create a new instance of a class. In Python the __new__() magic method is implicitly called before the __init__() method. The __new__() method returns a new object, which is then initialized by __init__().

```
class Employee:
    def __new__(cls):
        print ("__new__ magic method is called")
        inst = object.__new__(cls)
                return inst
    def __init__(self):
        print ("__init__ magic method is called")
        self.name=Python'
```

# Magic Methods

**__str__() method**

•Another useful magic method is __str__().

•It is overridden to return a printable string representation of any user defined class.

•We have seen str() built-in function which returns a string from the object parameter.

For example, str(12) returns '12'. When invoked, it calls the __str__() method in the int class.

# Magic Methods

```
>>> num=12
>>> str(num)
'12'
>>> #This is equivalent to
>>> int.__str__(num)
'12’
```

# Magic Methods

Override the __str__() method in the Employee class to return a string representation of its object.

```
class Employee:
    def __init__(self):
        self.name='Swati'
        self.salary=10000
    def __str__(self):
        return 'name='+self.name+' salary=$'+str(self.salary)
```

# Magic Methods

See how the str() function internally calls the __str__() method defined in the Employee class.

This is why it is called a magic method!

```
>>> e1=Employee()
>>> print(e1)
name=Swati salary=$10000
```

# Magic Methods

**__add__() method**

In following example, a class named distance is defined with two instance attributes - ft and inch.

The addition of these two distance objects is desired to be performed using the overloading + operator.

To achieve this, the magic method __add__() is overridden, which performs the addition of the ft and inch attributes of the two objects.

The __str__() method returns the object's string representation.

# Magic Methods

```
class distance:
    def __init__(self, x=None,y=None):
        self.ft=x
        self.inch=y
    def __add__(self,x):
        temp=distance()
        temp.ft=self.ft+x.ft
        temp.inch=self.inch+x.inch
        if temp.inch>=12:
            temp.ft+=1
            temp.inch-=12
            return temp
    def __str__(self):
        return 'ft:'+str(self.ft)+'  in: '+str(self.inch)
```

```
>>> d1=distance(3,10)
>>> d2=distance(4,4)
>>> print("d1= {} d2={}".format(d1, d2))
d1= ft:3  in: 10 d2=ft:4 in: 4
>>> d3=d1+d2
>>> print(d3)
ft:8 in: 2
```

# Magic Methods

**__ge__() method**

The following method is added in the distance class to overload the >= operator.

```
class distance:
    def __init__(self, x=None,y=None):
        self.ft=x
        self.inch=y
    def __ge__(self, x):
        val1=self.ft*12+self.inch
        val2=x.ft*12+x.inch
        if val1>=val2:
            return True
        else:
            return False
```

# Magic Methods

This method gets invoked when the >= operator is used and returns True or False. Accordingly, the appropriate message can be displayed.

```
>>> d1=distance(2,1)

>>> d2=distance(4,10)

>>> d1>=d2

False
```