OOP [PYTHON] - By Prem Sharma

CLASS , OBJECTS , ENCAPSULATION , POLYMORPHISM , INHERITANCE , ABSTRACTION

=>>Object-Oriented Design
Restructuring the class data (if necessary),
Implementation of methods, i.e., internal data structures and algorithms,
Implementation of control, and
Implementation of associations.

=>>Key Concepts in OOP:

Classes:-

Classes serve as blueprints or templates(define structure) for creating objects.
They define the properties (attributes) and behaviours (methods) that objects will have.
We treat classes as custom data types.

OR

A class is a collection of objects. A class contains the blueprints or the prototypes from which the objects are being created. it is a logical entity that contains some attributes and methods
->Classes are created by keyword class.
->Attributes are the variables that belong to a class.
->Attributes are always public and can be accessed using the dot(.) operator.
Eg=>
Myclass.Myattribute

OBJECTS AND CLASSES

Objects:-

Objects are instances of classes. They represent individual entities(real-world entity) that possess
characteristics (attributes) and can perform actions (methods) defined in their class.
An object can be a variable.

OR

objects is an entity that has a state and behaviour associated with it. it may be any real world objects like keyboard , chair , table , etc. & Integers , string floating-points numbers , even arrays and dictionaries all are objects. like number 12 an int , "Hello" an string etc.

Attributes:-

Attributes, also known as properties or fields, are variables that store data within an object.
These attributes define the object's state.

Methods:-

Methods are functions or procedures associated with objects.
They represent the actions or behaviours that objects can perform.

```
# Syntex:
class prem:
    '''This is prem sharma !!!'''
print(prem.__doc__)
help(prem)
```

OUTPUT:-

```
This is Prem sharma !!!
Help on class prem in module __main__:

class prem(builtins.object)
 |  This is Prem sharma !!!
 |
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

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

    def talk(self):
        return f'''
        name:{self.name}
        age:{self.age}
        marks:{self.marks}'''
        # print('Hello i am :',self.name)-------}
        # print('My Age is :',self.age)---------}=>commented is an another type to print
        # print('My marks are :',self.marks)----}
obj1=Student("Prem",18,76)
# obj1.talk()
obj2=Student("Manish",18,82)
# obj2.talk()

out = obj1.talk()
print(out)
print("----------------------")
out = obj2.talk()
print(out)
```

OUTPUT:-

```
        name:Prem
        age:18
        marks:76
----------------------
        name:Manish
        age:18
        Marks:82
```

Self variable:-

self is the default variable which is always pointing to current object
(like this keyword in Java)
By using self we can access instance variables and instance methods of objects.
Note:
1. self should be first parameter inside constructor
def __init__(self):
2. self should be first parameter inside instance methods
def talk(self):

CONSTRUCTOR AND DESTRUCTORS

Constructor Concept:-

☕Constructor is a special method in python.
☕The name of the constructor should be __init__(self)
☕Constructor will be executed automatically at the time of object creation.
☕The main purpose of constructor is to declare and initialise instance variables.
☕Per object constructor will be executed only once.
☕Constructor can take at least one argument(at least self)
☕Constructor is optional and if we are not providing any constructor then python will provide default constructor.
-> In Python name of the constructor is "__init__()".

```
# Constructor example
def __init__(self,name,rollno,marks):
    self.name=name
    self.rollno=rollno
    self.marks=marks
   ———————————
# Question with the Constructor.
# constructor is a special method which is used to initialise the members of the class

class Employee:
    def __init__(self):
        self.name = "NA"
        self.age = 0
        self.salary = 0
    def setDetails(self,name,age,salary):
        self.name = name
        self.age = age
        self.salary = salary
    def getDetails(self):
        print("Name : ",self.name)
        print("Age : ",self.age)
        print("Salary : ",self.salary)

# main driven code
emp1 = Employee()
emp2 = Employee()
emp3 = Employee()
emp1.setDetails("John",30,30000)
emp2.setDetails("Mike",40,40000)
emp1.getDetails()
print("----------")
emp2.getDetails()
print("----------")
emp3.getDetails()  # use the default info as we have not set any values for emp3
```

OUTPUT:-

```
Name :  John
Age :  30
Salary :  30000
-------------
Name :  Mike
Age :  40
Salary :  40000
-------------
Name :  NA
Age :  0
Salary :  0
——————————
Name :  John
Age :  30
Salary :  30000
destructor called John object deleted
-------------
Name :  Mike
Age :  40
Salary :  40000
```

_____

=>>What is constructor overloading?

Constructor overloading is a concept where we can have more than one constructor in a class

```
class Employee:
    def __init__(self,name,age,salary):
        self.name = name
        self.age = age
        self.salary = salary
    # def __init__(self,name,age):
    #     self.name = name
    #     self.age = age
    #     self.salary = 0
    def getDetails(self):
        print("Name : ",self.name)
        print("Age : ",self.age)
        print("Salary : ",self.salary)
# main driven code
emp1 = Employee("John",30, 30000)
emp2 = Employee("Mike",40, 10000)
emp1.getDetails()
print("-------------")
emp2.getDetails()
print("-------------")
```

OUTPUT:-

```
Name :  John
Age :  30
Salary :  30000
-------------
Name :  Mike
Age :  40
Salary :  10000
-------------
```

Destructors:-

Destructor is a special method and the name should be __del__
Just before destroying an object Garbage Collector always calls destructor to perform clean up activities (Resource deallocation activities like close database connection etc).
Once destructor execution is completed then Garbage Collector automatically destroys that object.

Note:

The job of a destructor is not to destroy objects and it is just to perform clean up activities.

```
# a default method use to simulate the object deletion
class test:
    def __init__(self):
        print('This is Constructor')

    def __del__(self):
        print('This is destructor')
obj1 = test()
obj2 = test()
del obj1
print('hello')
```

OUTPUT:-

```
This is Constructor
This is Constructor
This is destructor
Hello
```

OR

```
class test:
    def __init__(self):
        print('This is Constructor')

    def details(self, name):
        self.name = name

    def __del__(self):
        print(f'{self.name} object is deleted')

# main code
obj1 = test()
obj2 = test()
obj2.details('ravi')
obj1.details('saket')
del obj2
```

OUTPUT:-
This is Constructor
This is Constructor
This is destructor
ravi object is deleted
——----------

=>>Question with Destructor

destructor is a special method which is used to delete the members of the class

```
class Employee:
    def __init__(self):
        self.name = "NA"
        self.age = 0
        self.salary = 0
    def setDetails(self,name,age,salary):
        self.name = name
        self.age = age
        self.salary = salary
    def getDetails(self):
        print("Name : ",self.name)
        print("Age : ",self.age)
        print("Salary : ",self.salary)
    def __del__(self):
        print(f"destructor called {self.name} object deleted")

# main driven code
emp1 = Employee()
emp2 = Employee()
emp1.setDetails("John",30,30000)
emp2.setDetails("Mike",40,40000)
emp1.getDetails()
del emp1
print("-------------")
emp2.getDetails()
print("-------------")
```

OUTPUT:-
Name : John
Age : 30
Salary : 30000
destructor called John object deleted

Name : Mike
Age : 40
Salary : 40000

TALKING ABOUT BOTH IN SINGLE QUESTION :-

```
class Test:
    # constructor
    def __init__(self):
        print('This is constructor')
    def disp(self):
        print('hello world')
    # destructor
    def __del__(self):
        print('This is destructor')

obj1 = Test()
obj2 = Test()

print('Object Created')
obj1.disp()
obj1.disp()
del obj1
```

OUTPUT;-

```
This is constructor
This is constructor
This is destructor
Object Created
hello world
hello world
This is destructor
```

```
class Test:
    _a = 10
    def change(self):
        self.ab = 10
    def readClassvar(self):
        return self._a

obj = Test()
# print(_Test__a)
print(obj.readClassvar())
print(obj._a)
```

OUTPUT:-

```
10
10
```

CLASS VARIABLE AND INSTANCE VARIABLE

=>>What is the class variable?

Class variable is a variable that is shared by all instances of a class

---

=>>What is an instance variable?

Instance variable is a variable that is unique to each instance

EXAMPLE:-

```
# class variable
class Test:
    # class variable
    # x = 10
     def __init__(self):
        # instance variable
        self.y = 20
    def m1(self):
        # instance variable
        self.z = 30
obj = Test()
obj.m1()
obj.a = 40
print(obj.__dict__)
```

OUTPUT:-

```
{'y': 20, 'z': 30, 'a': 40}
```

```
# examples
class Employee:
    # class variable
    raise_amount = 1.04
    num_of_emps = 0
   # instance variable
    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay
        # we don't want to change the number of employees when we create an instance
        # so we use Employee.num_of_emps instead of self.num_of_emps
        Employee.num_of_emps += 1

    # regular method
    def fullname(self):
        return '{} {}'.format(self.first, self.last)
    # regular method
    def apply_raise(self):
```

```
        # we can access class variable through class itself or instance
        # self.raise_amount is better because we can change the raise_amount for a single
instance
        # Employee.raise_amount is better because we can change the raise_amount for
all instances
        self.pay = int(self.pay * self.raise_amount)
```

```
class Test:
    x = 10
    def add(self):
        self.var = 20
obj = Test()
obj.add()
print(obj.__dict__)
```

OUTPUT:-
{'var': 20}

```
# class Variable : shared by all instances.
# Instance Variable: unique to each instance.
```

- - - - - - - - - - - - - - - - - -

```
class Test:
    # class Variable
    val = 100
    def __init__(self, a):
        # instance variable
        self.a = a
obj1 = Test(20)
obj2 = Test(30)
print(obj1.val)
```

OUTPUT:-
100

- - - - - - - - - - - - - - - - - -

SOME THINGS WHICH CAN HELPS IN OOP

```
print(dir(obj1))
```

OUTPUT:-

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'get', 'tmp', 'val']

```
# how to create dictionary in oops
out = getattr(obj1 , 'val' , 120)

print(obj1.__dict__)
OUTPUT:-
{'val': 20}
```

```
# repr work as a print item as it is....
x = '12345'
print(repr(x))
# -----------------
x = '''
hello
prem'''
print(repr(x))
OUTPUT:-
'12345'
'\nhello\nprem'
```

INHERITANCE

=>>One of the core concepts in object-oriented programming (OOP) languages is inheritance. It is a mechanism that allows you to create a hierarchy of classes that share a set of properties and methods by deriving a class from another class. Inheritance is the capability of one class to derive or inherit the properties from another class.

=> Benefits of inheritance are:

-> Inheritance allows you to inherit the properties of a class, i.e., base class to another, i.e., derived class. The benefits of Inheritance in Python are as follows:
-> It represents real-world relationships well.
-> It provides the reusability of a code.
-> We don't have to write the same code again and again.
Also, it allows us to add more features to a class without modifying it.
-> It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.
-> Inheritance offers a simple, understandable model structure.
-> Less development and maintenance expenses result from an inheritance.
-> Python Inheritance Syntax
-> The syntax of simple inheritance in Python is as follows:

```
class BaseClass:
    # {Body}
class DerivedClass(BaseClass):
    # {Body}
```

```
# a = 12
# isinstance(a , int)
class Task:
    tmp = 0
    def __init__(self , val):
        self.val = val
    def get(self):
        return self.val

    # def get(self , v):
    #     self.val = v

obj1 = Task(20)
obj1 = Task(30)
obj1.val = 20
# obj1.get(100)
print(obj1.get())
```
OUTPUT:-
20

```
Task(30).val
```
OUTPUT:-
30

```
out = obj1.__dict__
print(out)
```
OUTPUT:-
{'val': 20}

```
isinstance(Task(20),Task)
```
OUTPUT:-
True
```
# single inheritance
class A:
    def __init__(self):
        self.a = 10
class B(A):
    def __init__(self):
        super().__init__(self)
```

```
-----------------------------------------------
# multiple inheritance
class A:
    def __init__(self):
        self.a = 10
    def disp(self):
        return "This is disp in class A"

class B:
    def __init__(self):
        self.b = 100
    def disp2(self):
        return "This is disp in class B"

class C(A , B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        self.c = 1000

# a1 = A()
# print(a1.__dict__)

# b1 = B()
# print(b1.__dict__)

c1 = C()
print(c1.__dict__)
print(c1.disp())
print(c1.disp2())
OUTPUT:-
{'a': 10, 'b': 100, 'c': 1000}
This is disp in class A
This is disp in class B
```

ABSTRACT METHOD

=>>Working on Python Abstract classes

By default, Python does not provide abstract classes. Python comes with a module that provides the base for defining Abstract Base classes(ABC) and that module name is ABC.

ABC works by decorating methods of the base class as an abstract and then registering concrete classes as implementations of the abstract base. A method becomes abstract when decorated with the keyword @abstractmethod.

```
# HOW TO  USE ABSTRACT METHOD
from abc import ABC, abstractmethod

class shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass
```

——--

```
EXAMPLE ON ABSTRACT METHOD
from abc import ABC , abstractmethod
class test(ABC):
    @abstractmethod
    def add(self , a , b):
        return a + b

    @abstractmethod
    # if i not use [@abstractmethod] and write below area there is an error
    def sub(self , a , b):
        return a - b

class Sample(test):
    def add(self , a , b):
        return f' Addition of {a} and {b} is {a + b}'

    def sub(self , a , b):
        return f' Subtraction of {a} and {b} is {a - b}'

obj = Sample()
print(obj.add(2 , 4))
print(obj.sub(2 , 4))
```

OUTPUT:-

```
Addition of 2 and 4 is 6
Subtraction of 2 and 4 is -2
```

——--

```
# Python program showing
# abstract base class work
from abc import ABC, abstractmethod
class Polygon(ABC):
    @abstractmethod
    def no_of_sides(self):
        pass

class Triangle(Polygon):
    # overriding abstract method
    def no_of_sides(self):
        print("I have 3 sides")

class Pentagon(Polygon):
    # overriding abstract method
    def no_of_sides(self):
        print("I have 5 sides")

class Hexagon(Polygon):
    # overriding abstract method
    def no_of_sides(self):
        print("I have 6 sides")

class Quadrilateral(Polygon):
    # overriding abstract method
    def no_of_sides(self):
        print("I have 4 sides")

# Driver code
R = Triangle()
R.no_of_sides()

K = Quadrilateral()
K.no_of_sides()

R = Pentagon()
R.no_of_sides()

K = Hexagon()
K.no_of_sides()
```

ENCAPSULATION

=>> Meaning of encapsulation ??
It describes the idea of bundling data and methods that work on that data within one unit , e.g. a class in Python
This concept is also often used to hide the internal representation ,

```
class test:
    __a = 0 # private class variable
    def __init__(self , val):
        self.__val = val  # private instance variable
    def change_val(self , new_val):
        self.__val = new_val

obj = test(10)
print(obj._test__val)  #-->> mungling process
# print(obj.__val) -->>error
```

OUTPUT:-
10

=>>Encapsulation hides the internal state of an object the outside world and only exposes the necessary

```
# encapsulation -- public  , protected , private
# Can be public-->>accessible from any where
# protected ->> accessible from within the class and its subclass
# private ->> accessible only from within the class
```

```
# abstract class
from abc import ABC #, abstractmethod
class Rect(ABC):  #-IF WE USE ABSTRACT METHOD Rect(ABC)-->>does not work
    # @abstractmethod
    def area(self , l , b):
        return l*b
    # @abstractmethod
    def perimeter(self , l , b):
        return 2 * (l + b)

obj = Rect()
print(obj.area(2 , 5))
print(obj.perimeter(2 , 5))
```

OUTPUT:-
10
14

```
class Rectangle(Rect):
    def area(self , l , b):
        print('hi')

obj = Rectangle()
print(obj.area(3 , 4))
```

OUTPUT:-

```
hi
None
```

Example=>> 01

```
class Car:
    def __init__(self, speed, color):
        self.__speed = speed # private attribute
        self.color = color # public attribute

    def set_speed(self, value):
        self.__speed = value

    def get_speed(self):
        return self.__speed

car1 = Car(200, 'red')
car2 = Car(250, 'blue')

car1.set_speed(300)
print(car1.get_speed())
print(car1.color)

# print(car1.__speed) # AttributeError: 'Car' object has no attribute '__speed'
```

OUTPUT:-

```
300
red
```

Example=>> 02

```
class shape:
    def __init__(self):
        print('shape class constructor')
    def area(self):gg
        print('this is area in shape')

class rect(shape):
    def __init__(self):
        print('rect class ')
        shape.__init__(self)
    def area(self):
        print('this is area in rect')
        shape.area(self)
   def peri(self):
        print('perimeter in rect')

obj1 = rect()
obj1.area()
```

OUTPUT:-

```
rect class
shape class constructor
this is area in rect
this is area in shape
```

Example=>> 03

```
class test:
    def __init__(self) :
        self.__val = 100
    def read_val(self):
        return self.__val

    def change_val(self, new_val):
        self.__val = new_val
obj = test()
print(obj.read_val())
obj.change_val(23)
print(obj.read_val())
```

OUTPUT:-

```
100
23
```

POLYMORPHISM

==>> Polymorphism is a key concept in object-oriented programming (OOP) that allows objects of different types to be treated as objects of a common superclass.

— — — — — — — — — — — — — — — —

```
# example of polymorphism
class Duck:
    def sound(self):
        print("Quack")

class Human:
    def sound(self):
        print("Hello")

# Duck typing example
def make_sound(entity):
    entity.sound()

duck = Duck()
human = Human()

make_sound(duck)   # Output: "Quack"
make_sound(human)  # Output: "Hello"
```

OUTPUT:-

Quack
Hello

——----------

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):
        print("Dog barks")

class Cat(Animal):
    def speak(self):
        print("Cat meows")
```

```
# Example of method overriding
animal = Animal()
animal.speak()  # Output: "Animal speaks"

dog = Dog()
dog.speak()     # Output: "Dog barks"

cat = Cat()
cat.speak()     # Output: "Cat meows"
```

OUTPUT:-
Animal speaks
Dog barks
Cat meows

---

1. Compile-Time Polymorphism (Static Binding / Early Binding):
    a)Method (Function) Overloading
    b)Operator Overloading

_ _ _ _ _ _ _

2. Run-Time Polymorphism (Dynamic Binding / Late Binding):
a)Function Overriding.

In Python, polymorphism is mainly achieved through run-time polymorphism (method overriding) since the language does not
directly support function overloading of operator overloading as seen in statically-typed languages like C++ or Java. However,

---

Python does support operator overloading through magic methods (e.g., __add__, __sub__, etc.) and function overloading
can be emulated using default arguments or variable-length argument lists.

Method (Function) Overloading.

⇒> Function Overloading is defined as the process of having two or more functions with the same name,but different in parameters. Function overloading can be considered as an example of polymorphism in Python.
Python does not support function overloading. We may overload the methods but can only use the latest defined method.

```
# EXAMPLES
class ABC:
    def add(self , a , b , c=0):
        return a + b + c
    # decide before compiling
obj = ABC()
print(obj.add(3 , 4))
```

OUTPUT:-
7

```
class ABC:
    def add(self , *c):
        return sum(c)
obj = ABC()
print(obj.add(3 , 4))
```

OUTPUT:-
7

Operator Overloading.

=>> Operator Overloading means giving extended meaning beyond their predefined operational meaning.

For example operator + is used to add two integers as well as join two strings and merge two lists.

```
# EXAMPLE
class ABC:
    def add(self , *c):
        return sum(c)
    def __add__(self , other): # " __xxx__ " is known as magic methods.
        return 'This is addition'
obj1 = ABC()
obj2 = ABC()

print(obj1 + obj2) # when we comparing or subtracting we use another __xxx___  , not using this __add__
```

Function Overriding.

=>> Function overriding means that a derived class function is redefining the base class function.

——————————

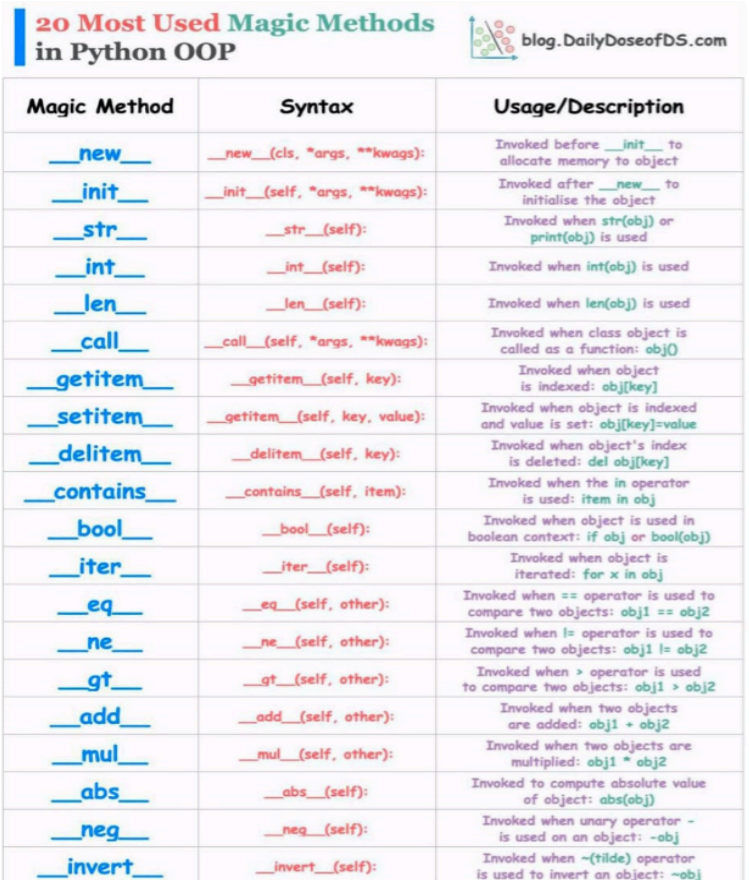
EXAMPLE

```
class abc:
    def add(self , a , b):
        return a + b

    def sub(self , a , b):
        return a - b

class xyz(abc):
    def add(self , a , b):
        return f' Addition of {a} and {b} is {a + b}'
    def div(self , a , b):
        return f' Division of {a} by {b} is {a / b}'
obj = xyz()
print(obj.add(5 , 2))
print(obj.div(5 , 2))
```

OUTPUT:-

Addition of 5 and 2 is 7

Division of 5 by 2 is 2.5



20 Most Used Magic Methods in Python OOP

blog.DailyDoseofDS.com

Magic Method	Syntax	Usage/Description
__new__	__new__(cls, *args, **kwags):	Invoked before __init__ to allocate memory to object
__init__	__init__(self, *args, **kwags):	Invoked after __new__ to initialise the object
__str__	__str__(self):	Invoked when str(obj) or print(obj) is used
__int__	__int__(self):	Invoked when int(obj) is used
__len__	__len__(self):	Invoked when len(obj) is used
__call__	__call__(self, *args, **kwags):	Invoked when class object is called as a function: obj()
__getitem__	__getitem__(self, key):	Invoked when object is indexed: obj[key]
__setitem__	__getitem__(self, key, value):	Invoked when object is indexed and value is set: obj[key]=value
__delitem__	__delitem__(self, key):	Invoked when object's index is deleted: del obj[key]
__contains__	__contains__(self, item):	Invoked when the in operator is used: item in obj
__bool__	__bool__(self):	Invoked when object is used in boolean context: if obj or bool(obj)
__iter__	__iter__(self):	Invoked when object is iterated: for x in obj
__eq__	__eq__(self, other):	Invoked when == operator is used to compare two objects: obj1 == obj2
__ne__	__ne__(self, other):	Invoked when != operator is used to compare two objects: obj1 != obj2
__gt__	__gt__(self, other):	Invoked when > operator is used to compare two objects: obj1 > obj2
__add__	__add__(self, other):	Invoked when two objects are added: obj1 + obj2
__mul__	__mul__(self, other):	Invoked when two objects are multiplied: obj1 * obj2
__abs__	__abs__(self):	Invoked to compute absolute value of object: abs(obj)
__neg__	__neg__(self):	Invoked when unary operator - is used on an object: -obj
__invert__	__invert__(self):	Invoked when ~(tilde) operator is used to invert an object: ~obj

GUI Graphical User Interface

Graphical User Interface(GUI) is a form of user interface which allows users to interact with computers through visual indicators using items such as icons, menus, windows, etc. It has advantages over the Command Line Interface(CLI) where users interact with computers by writing commands using keyboard only and whose usage is more difficult than GUI.

What is Tkinter?

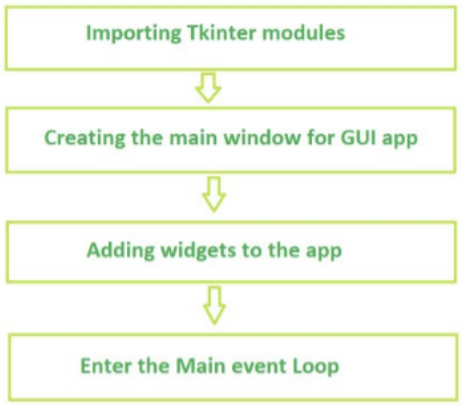
⇒Tkinter is the inbuilt python module that is used to create GUI applications. It is one of the most commonly used modules for creating GUI applications in Python as it is simple and easy to work with. You don't need to worry about the installation of the Tkinter module separately as it comes with Python already. It gives an object-oriented interface to the Tk GUI toolkit.

Some other Python Libraries available for creating our own GUI applications are
Kivy
Python Qt
wxPython
Among all Tkinter is most widely used

=>Here are some common use cases for Tkinter:

1) Creating windows and dialog boxes: Tkinter can be used to create windows and dialog boxes that allow users to interact with your program. These can be used to display information, gather input, or present options to the user.
2) Building a GUI for a desktop application: Tkinter can be used to create the interface for a desktop application, including buttons, menus, and other interactive elements.
3) Adding a GUI to a command-line program: Tkinter can be used to add a GUI to a command-line program, making it easier for users to interact with the program and input arguments.
4) Creating custom widgets: Tkinter includes a variety of built-in widgets, such as buttons, labels, and text boxes, but it also allows you to create your own custom widgets
5) Prototyping a GUI: Tkinter can be used to quickly prototype a GUI, allowing you to test and iterate on different design ideas before committing to a final implementation.

6) In summary, Tkinter is a useful tool for creating a wide variety of graphical user interfaces, including windows, dialog boxes, and custom widgets. It is particularly well-suited for building desktop applications and adding a GUI to command-line programs.




```
# Tkinter event loop
from tkinter import *
from tkinter.ttk import *

# writing code needs to
# create the main window of
# the application creating
# main window object named root
root = Tk()

# giving title to the main window
root.title("First_Program")

# Label is what output will be
# show on the window
label = Label(root, text ="Hello World !").pack()

# calling mainloop method which is used
# when your application is ready to run
# and it tells the code to keep displaying
root.mainloop()
```

What are Widgets?

=> Widgets in Tkinter are the elements of GUI application which provides various controls (such as Labels, Buttons, ComboBoxes, CheckBoxes, MenuBars, RadioButtons and many more) to users to interact with the application. Fundamental structure of tkinter program.

—Basic Tkinter Widgets:

Tkinter is the GUI library of Python, it provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called Widgets. The list of commonly used Widgets are mentioned below –

Widget Description:-

Label	The Label widget is used to provide a single-line caption for other widgets. It can also contain images.
Button	The Button widget is used to display buttons in your application.
Entry	The Entry widget is used to display a single-line text field for accepting values from a user.
Menu	The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton.
Canvas	The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application.
Checkbutton	The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time.
Frame	The Frame widget is used as a container widget to organise other widgets.
Listbox	The Listbox widget is used to provide a list of options to a user.
Menubutton	The Menubutton widget is used to display menus in your application.
Message	The Message widget is used to display multiline text fields for accepting values from a user.
Radio Button	The Radio button widget is used to display a number of options as radio buttons. The user can select only one option at a time.
Scale	The Scale widget is used to provide a slider widget.
Scroll	The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes.
Text	The Text widget is used to display text in multiple lines.
Toplevel	The Toplevel widget is used to provide a separate window container.
LabelFrame	A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts.

tkMessageBox	This module is used to display message boxes in your applications.
Spinbox	The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values.
PanedWindow	A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically.

```
# Example :
from tkinter import *

 # create root window
root = Tk()

# frame inside root window
frame = Frame(root)

# geometry method
frame.pack()

# button inside frame which is
# inside root
button = Button(frame, text ='Prem')
button.pack()
```

Geometry Management

=> Creating a new widget doesn't mean that it will appear on the screen. To display it, we need to call a special method: either grid, pack(example above), or place.

Method Description

pack()	The Pack geometry manager packs widgets in rows or columns.
grid()	The Grid geometry manager puts the widgets in a 2-dimensional table.
place()	The Place geometry manager is the simplest of the three general geometry managers provided in Tkinter.

- It allows you to explicitly set the position and size of a window, either in absolute terms, or relative to another window.

- Using the Entry() class we will create a text box for user input. To display the user input text, we'll make changes to the function clicked(). We can get the user entered text using the get() function. When the Button after entering of the text, a default text concatenated with the user text. Also change button grid location to column 2 as Entry() will be column 1.

```
# Import Module
from tkinter import *

# create root window
root = Tk()

# root window title and dimension
root.title("Welcome PREM SHARMA")
# Set geometry(widthxheight)
root.geometry('350x200')

# adding a label to the root window
lbl = Label(root, text = "Are you a Human")
lbl.grid()

# adding Entry Field
txt = Entry(root, width=10)
txt.grid(column =1, row =0)


# function to display user text when
# button is clicked
def clicked():

    res = "You wrote , " + txt.get()
    lbl.configure(text = res)

# button widget with red color text inside
btn = Button(root, text = "Click me" ,
             fg = "red", command=clicked)
# Set Button Grid
btn.grid(column=2, row=0)

# Execute Tkinter
root.mainloop()
```