**Class and object:**

-> Class is a entity or blueprint which contains properties.

-> Object is a it's reference or instance of a class

-> 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.

**ex:**

class Person:

def \_init\_(self, name, age):

self.name = name

self.age = age

p1 = Person("John", 36)

print(p1.name)

print(p1.age)

**Abstraction:**

-> Abstraction is used to hiding the internal data.

-> By we used abc(abstract base classes) class method with in a function to abstract method.

-> It's specially used for security purpose.

**syntax:**

from abc import ABC,abstractmethod

**ex:**

from abc import Abc,abstractmethod

class colleges(ABC):

def types(self):

pass

class eng\_colleges(colleges):

def types(self):

print("I am studying in engineering")

a=eng\_colleges

a.types

**Polymorphism:**

-It means the ability to take various forms(some objects having different behaviors)

-it made-up by two terms

-its simply we can call like one name many forms.

**example:**

class draw:

def \_init\_(self):

self.image="jpg"

def display(self):

print(self.image)

class main(draw):

def \_init\_(self):

self.image="png"

def display(self):

print(self.image)

obb1=draw()

obb1.display()

obb2=main()

obb2.display()

ootput:

jpg

png

**ENCAPSULATION:**

wrapping up of data and functiond into single unit is known as encapsulation

Encapsulation provides the basic property to hide data, thereby providing security to user data.

it is also known as INFORMATION HIDING CONCEPT

class A:

\_a=10 #protected

\_\_b=20 #private

def show(self):

print("a=",self.\_a)

print("b=",self.\_\_b)

obj1=A()

obj1.show()#show a=10,b=20

print("OUTSIDE OF CLASS",obj1.\_a) #show out put a=10

print("OUTSIDE OF CLASS",obj1.\_\_b) # show attribute error

**DUNDER:-**

* Dunder methods, also known as magic methods or special methods, are predefined methods in Python that provide functionality for certain operations and behaviors.
* These methods are always surrounded by double underscores (i.e., dunder) both at the beginning and end of their names.
* Let's look at an example of a dunder method and its real-time usage. One commonly used dunder method is \_\_str\_\_, which is used to define the string representation of an object.

EXAMPLE:-

class Car:

def \_\_init\_\_(self, make, model, year):

self.make = make

self.model = model

self.year = year

def \_\_str\_\_(self):

return f"{self.make} {self.model} ({self.year})"

my\_car = Car("Toyota", "Camry", 2022)

print(my\_car)

• In this example, we define a Car class with attributes make, model, and year. The \_\_str\_\_ method is overridden to provide a custom string representation of the Car object. When we call print(my\_car), the \_\_str\_\_ method is automatically invoked and returns the desired string representation: "Toyota Camry (2022)".

• By using \_\_str\_\_, we can control how an object is printed or converted to a string. This can be useful for debugging, logging, or any situation where we want a customized textual representation of an object.

• Dunder methods allow us to implement various behaviors and operations in our classes, such as comparison, mathematical operations, attribute access, and more. They provide a powerful way to define the behavior of objects in Python and make our code more intuitive and readable.

**Types of DUNDER METHOD:-**

Object Initialization:

* \_\_init\_\_(self, ...): Initializes an object.

String Representation:

* \_\_str\_\_(self): Returns a string representation of the object.
* \_\_repr\_\_(self): Returns a string representation that can be used to recreate the object.

Container Methods:

•\_\_len\_\_(self): Returns the length of the object.

•\_\_getitem\_\_(self, key): Gets an item using the subscript notation ([]).

•\_\_setitem\_\_(self, key, value): Sets an item using the subscript notation ([]).

•\_\_delitem\_\_(self, key): Deletes an item using the subscript notation ([]).

Comparison Operators:

* \_\_eq\_\_(self, other): Implements the equality operator (==).
* \_\_ne\_\_(self, other): Implements the not equal operator (!=).
* \_\_lt\_\_(self, other): Implements the less than operator (<).
* \_\_gt\_\_(self, other): Implements the greater than operator (>).
* \_\_le\_\_(self, other): Implements the less than or equal to operator (<=).
* \_\_ge\_\_(self, other): Implements the greater than or equal to operator (>=).

Mathematical Operators:

* \_\_add\_\_(self, other): Implements the addition operator (+).
* \_\_sub\_\_(self, other): Implements the subtraction operator (-).
* \_\_mul\_\_(self, other): Implements the multiplication operator (\*).
* \_\_divmod\_\_(self, other): Implements the division and modulus operator (divmod()).
* \_\_pow\_\_(self, other): Implements the exponentiation operator (\*\*).
* \_\_floordiv\_\_(self, other): Implements the floor division operator (//).
* \_\_mod\_\_(self, other): Implements the modulus operator (%).

Attribute Access:

* \_\_getattr\_\_(self, name): Gets called when an attribute is not found.
* \_\_setattr\_\_(self, name, value): Gets called when an attribute is set.
* \_\_delattr\_\_(self, name): Gets called when an attribute is deleted.

#object init

class MyClass:

def \_\_init\_\_(self, value):

self.value = value

def \_\_str\_\_(self):

return f"MyClass object with value: {self.value}"

obj = MyClass(42)

print(obj)

#attribute

class MyClass:

def \_\_getattr\_\_(self, name):

return f"Attribute '{name}' not found."

def \_\_setattr\_\_(self, name, value):

print(f"Setting attribute '{name}' to {value}.")

self.\_\_dict\_\_[name] = value

def \_\_delattr\_\_(self, name):

print(f"Deleting attribute '{name}'.")

del self.\_\_dict\_\_[name]

obj = MyClass()

print(obj.attribute) # Output: Attribute 'attribute' not found. -gettatr

obj.new\_attribute = 42 # Output: Setting attribute 'new\_attribute' to 42. -setattr

print(obj.new\_attribute) # Output: 42

del obj.new\_attribute # Output: Deleting attribute 'new\_attribute'. -delatrr

print(obj.new\_attribute) # Output: Attribute 'new\_attribute' not found.

class MyList:

def \_\_init\_\_(self, items):

self.items = items

def \_\_len\_\_(self):

return len(self.items)

def \_\_getitem\_\_(self, index):

return self.items[index]

def \_\_setitem\_\_(self, index, value):

self.items[index] = value

def \_\_delitem\_\_(self, index):

del self.items[index]

my\_list = MyList([1, 2, 3, 4, 5])

print(len(my\_list)) # Output: 5

print(my\_list[2]) # Output: 3

my\_list[1] = 10

print(my\_list[:]) #[1, 10, 3, 4, 5]

del my\_list[3]

print(len(my\_list)) # Output: 4