DAY-9 MORNING ASSESSMENT

# OOPS

1.difference between self and cls in python:

Self: self is used in instance method ,represents the object that is calling the method

Cls: cls is used in class method,allows access to class-level variables and methods.

2.Inheritance allows to inherit one classes properties and methods to another class. This promotes code reusability and makes your code more organized.

Example:

class Animal:  
 def sound(self):  
 print("Animal makes sound")  
class Dog(Animal):  
 #overriding the sound method  
 def sound(self):  
 print("Dog barks")  
  
class Cat(Animal):  
 #overriding the sound method  
 def sound(self):  
 print("Cat meows")  
  
a=Animal()  
a.sound()  
  
b=Dog()  
b.sound()  
  
c=Cat()  
c.sound()

o/p: Animal makes sound

Dog barks

Cat meows

3.Method overloading means defining multiple methods with the same name but different parameters(number,type,order)

No,the python does not support method overloading ,if you define multiple methods with the same name in the class , only the last one will be used.

Example:

class Demo:  
 def greet(self):  
 print("Hello")  
 def greet(self,name):  
 print("Hello",name)  
  
d=Demo()  
d.greet("Vidya") #"Hello Vidya"  
d.greet() #error ,requires one positional argument

4.-->A constructor is a special method in python used to initialize objects when they are created.

\_\_init\_\_ automatically called when you create an object of the class.

🡪Destructor(\_\_del\_\_)

A destructor is a special method that is called when an object is deleted or goes out of scope.(i.e garbage collected)

Class Myclass:

Def \_\_del\_\_(self):

Print(“destructor called”)

🡪called when the object is about to be destroyed and no reference remained.

5.INSTANCE METHOD:

Takes self as self as first argument. Can access and modify instance variables. Called using object of the class.

CLASS METHOD:

Takes cls as first argument. Can access or modify class variables not instance variables. Declared using @classmethod decorator.

Called using class name and object.

STATIC METHOD:

Doesn’t take self or cls as arguments. Declared using @staticmethod decorator. cannot access class or instance class directly.

6.using name conventions we can declare python variables as private(\_\_) and protected(\_) .

Protected: an attribute or method is used for internal purpose only.

Private: python mangles the attribute harder to access anywhere.

7. class BankAccount:  
   def \_\_init\_\_(self, owner, balance):  
       self.owner = owner           # public attribute  
       self.\_\_balance = balance     # private attribute (encapsulated)  
  
   # Getter method for balance  
   def get\_balance(self):  
       return self.\_\_balance  
  
   # Setter method for balance  
   def set\_balance(self, amount):  
       if amount < 0:  
           print("Balance cannot be negative.")  
       else:  
           self.\_\_balance = amount  
  
   # Method to deposit money  
   def deposit(self, amount):  
       if amount > 0:  
           self.\_\_balance += amount  
           print(f"Deposited ₹{amount}. New balance: ₹{self.\_\_balance}")  
       else:  
           print("Invalid deposit amount.")  
  
   # Method to withdraw money  
   def withdraw(self, amount):  
       if 0 < amount <= self.\_\_balance:  
           self.\_\_balance -= amount  
           print(f"Withdrew ₹{amount}. New balance: ₹{self.\_\_balance}")  
       else:  
           print("Insufficient funds or invalid amount.")

acc = BankAccount("Vidya", 5000)  
  
# Accessing public attribute  
print("Owner:", acc.owner)  
  
# Trying to access private attribute directly  
# print(acc.\_\_balance)  AttributeError  
  
# Using getter and setter methods  
print("Balance:", acc.get\_balance())      # 5000  
acc.set\_balance(6000)  
print("Updated Balance:", acc.get\_balance())  
  
acc.deposit(1000)  
acc.withdraw(2000)  
  
# Name mangling access (not recommended)  
print("Accessing with name mangling:", acc.\_BankAccount\_\_balance)

8.In python it allows methods to have different behaviours with the same name.

 Example: class Dog:  
   def speak(self):  
       print("Dog says: Woof!")  
  
class Human:  
   def speak(self):  
       print("Human says: Hello!")  
  
# Using polymorphism  
def make\_it\_speak(entity):  
   entity.speak()  # Calls the speak method of whatever object is passed  
  
# Unrelated classes, but same method name  
d = Dog()  
h = Human()  
  
make\_it\_speak(d)  # Output: Dog says: Woof!  
make\_it\_speak(h)  # Output: Human says: Hello!

 9.MAGIC METHOD:

Magic methods are automatically invoked by python in specific situations , allowing you to customize the behaviour of your objects.

* \_\_init\_\_(self)-initializes object
* \_\_del\_\_(self)-destructor
* \_\_str\_\_(self) – string representation for print

10.these two are built in functions used for type checking and inheritance checking.

Isinstance(obj,ClassName)

Issubclass(derivedclass,subclass)

# DECORATORS

1.decorator modifies the function without modifying the behaviour of source code using nested functions.

Typical use cases are:

Logging, authorization

2. def log\_decorator(func):  
 def wrapper(\*args, \*\*kwargs):  
 print(f"[LOG] {func.\_\_name\_\_} started.")  
 return func(\*args, \*\*kwargs)  
 return wrapper  
  
@log\_decorator  
def say\_hello():  
 print("Hello, world!")  
  
say\_hello()

3.yes multiple decorators can be applied.

Ex: @decorator1  
@decorator2  
def my\_function():  
 pass

This is equal to

My\_function=decorator1(decorator2(my\_function))

Decorator2 is applied first and then second one.

4. Preserve the original function’s metadata when it’s wrapped by a decorator.  
🡪Copy attributes like \_\_name\_\_, \_\_doc\_\_, \_\_module\_\_, etc., from the original function to the wrapper function.Preserve the original function’s metadata when it’s wrapped by a decorator.  
🡪Copy attributes like \_\_name\_\_, \_\_doc\_\_, \_\_module\_\_, etc., from the original function to the wrapper function.

5.to convert an argument into an parameterized decorator , you need to add an extra outer function to accept argument.

Before:

def log\_function(func):  
   def wrapper(\*args, \*\*kwargs):  
       print(f"Calling function: {func.\_\_name\_\_}")  
       return func(\*args, \*\*kwargs)  
   return wrapper  
  
@log\_function  
def greet(name):  
   print(f"Hello, {name}!")

after:

def log\_function(level):  # Outer function to accept argument  
   def decorator(func):  # Actual decorator  
       def wrapper(\*args, \*\*kwargs):  # Wrapper around the function  
           print(f"[{level}] Calling function: {func.\_\_name\_\_}")  
           return func(\*args, \*\*kwargs)  
       return wrapper  
   return decorator  
  
@log\_function("INFO")  
def greet(name):  
   print(f"Hello, {name}!")

6. user\_logged\_in = False # Simulated login status  
def login\_required(func):  
 def wrapper():  
 if user\_logged\_in:  
 func()  
 else:  
 print("Please log in to access this function.")  
 return wrapper  
@login\_required  
def show\_profile():  
 print("Welcome to your profile!")  
# Try without logging in  
show\_profile()  
# Simulate login  
user\_logged\_in = True  
show\_profile()

7.the property decorator is used to define getter method. encapsulates the internal/private data.

class Circle:  
   def \_\_init\_\_(self, radius):  
       self.\_radius = radius  # convention: \_radius is "protected"  
  
   @property  
   def area(self):  
       return 3.14 \* self.\_radius \*\* 2  
  
   @property  
   def radius(self):  
       return self.\_radius  
  
   @radius.setter  
   def radius(self, value):  
       if value > 0:  
           self.\_radius = value  
       else:  
           print("Radius must be positive.")  
  
# Usage  
c = Circle(5)  
print(c.area)      # No parentheses: acts like a variable  
print(c.radius)    # Get radius  
c.radius = 10      # Set radius using setter  
print(c.area)      # Area updated  
c.radius = -3      # Triggers warning

8. def exception\_logger(func):  
   def wrapper(\*args, \*\*kwargs):  
       try:  
           return func(\*args, \*\*kwargs)  
       except Exception as e:  
           print(f"Exception in {func.\_\_name\_\_}: {e}")  
   return wrapper  
  
@exception\_logger  
def divide(a, b):  
   return a / b  
  
# Usage  
print(divide(10, 2))  # 5.0  
print(divide(10, 0))  # Logs: Exception in divide: division by zero.

9. Function Decorator decorates functions or methods and it has function object and it uses for logging and validation.

Whereas class decorator decorates classes and it has class object and used for registering and modifying class behaviour.

10. class MyClass:  
   @staticmethod  
   @my\_decorator  
   def static\_method():  
       pass  
  
   @classmethod  
   @my\_decorator  
   def class\_method(cls):  
       pass

Yes can also apply decorators to instance methods normally

# GENERATORS

1. A generator uses yield to return a value one at a time and maintains state between calls.Normal function uses return and exit after returning.

2. def even\_gen():  
 for i in range(2,21,2):  
 yield i

3.raises a StopIteration

4.yield allows producing one value at a time, avoiding loading all data into memory.

5. gen=(x\*x for x in range(10))  
lst=[x\*x for x in range(10)]  
print(list(gen))  
print(lst)

o/p:

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

6. #list - returning  
def square\_list(n):  
 return [i\*i for i in range(n)]  
  
def square\_gen(n):  
 for i in range(n):  
 yield i\*i  
  
7. def read\_file\_gen(filename):  
 with open(filename,'r') as f:  
 for line in f:  
 yield line.strip()

8.it saves the execution context at each yield, and resumes from there on the next next() call.

9.return exits the function while yield pauses the function and continues if wanted.

10. def gen\_func():  
 yield 1  
 yield 2  
  
def list\_func():  
 return [1,2]  
  
print(list(gen\_func()))  
print(list\_func())

# ITERATORS

1.iterable can be looped over where as iterator is a object with next() and created using iter()

2. class Counter:  
   def \_\_init\_\_(self, max):  
       self.max = max  
       self.current = 1  
  
   def \_\_iter\_\_(self):  
       return self  
  
   def \_\_next\_\_(self):  
       if self.current <= self.max:  
           val = self.current  
           self.current += 1  
           return val  
       raise StopIteration  
  
for i in Counter(3):  
   print(i)

o/p: 1 2 3

3.it means the end of iteration to a for loop or next call.

4. with open("file.txt") as f:  
   for line in iter(f.readline, ''):  
       print(line)

5. It calls iter(obj) to get an iterator, then repeatedly calls next() until StopIteration.

6. Built-in functions using iterators:  
map(), filter(), zip(), enumerate(), reversed()It calls iter(obj) to get an iterator, then repeatedly calls next() until StopIteration.

7. nums = iter([1, 2, 3])  
print(next(nums))  # 1  
print(next(nums))  # 2

8.class SquareGen:  
   def \_\_init\_\_(self):  
       self.num = 1  
  
   def \_\_iter\_\_(self):  
       return self  
  
   def \_\_next\_\_(self):  
       if self.num <= 5:  
           val = self.num \*\* 2  
           self.num += 1  
           return val  
       raise StopIteration  
  
for x in SquareGen():  
   print(x)

9.it raises stop iteration.You must recreate it to use it again.

10. the itertools mudule provides a collection of fast, memory-efficient tools for handling iterators. It has infinite iterators like count(),cycle(),repeat() and combinators and also used for filtering.