**Day-9 Assessment**

**OOPs (Object-Oriented Programming)**

1. self refers to the instance of the class. It is used in instance methods to access instance-specific data. cls refers to the class itself. It is used in class methods to access class-level data.  
  
2. Inheritance allows a class to inherit attributes and methods from another class.  
class Animal:  
   def speak(self):  
       print("Animal speaks")  
  
class Dog(Animal):  
   def speak(self):     
       print("Dog barks")  
  
a = Animal()  
d = Dog()  
a.speak()    
d.speak()

o/p:

Animal speaks

Dog barks  
  
3.Method overloading means defining multiple methods with the same name but different parameters. Python does not support method overloading.

4. Constructor (\_\_init\_\_): Initializes the object, called automatically when an object is created.  
Destructor (\_\_del\_\_): Cleans up the object, called when the object is about to be destroyed.  
  
  
5.Instance method first arg is self, Class method first arg is cls and Static method is none.

class Demo:  
   def instance\_method(self):  
       print("Instance method")  
  
   @classmethod  
   def class\_method(cls):  
       print("Class method")  
  
   @staticmethod  
   def static\_method():  
       print("Static method")

6. By using single or double underscores we can restrict access.  
\_var = protected (convention)  
\_\_var = private (name mangling)  
  
7. class BankAccount:  
   def \_\_init\_\_(self):  
       self.\_\_balance = 0  
  
   def set\_balance(self, amount):  
       if amount < 0:  
           print("Invalid balance")  
       else:  
           self.\_\_balance = amount  
  
   def get\_balance(self):  
       return self.\_\_balance  
  
acc = BankAccount()  
acc.set\_balance(1000)  
print(acc.get\_balance())

o/p: 1000

8. Polymorphism allows objects of different classes to be treated using the same interface.  
class Cat:  
   def speak(self):  
       print("Meow")  
  
class Robot:  
   def speak(self):  
       print("Beep")  
  
def make\_speak(entity):  
   entity.speak()  
  
make\_speak(Cat())      
make\_speak(Robot())

o/p: Meow

Beep

9. Magic methods start and end with double underscores (\_\_). They provide special functionality.

Purpose:  
\_\_init\_\_  
Constructor  
\_\_str\_\_  
String representation  
\_\_len\_\_  
Length of object  
\_\_add\_\_  
Add operator overloading  
\_\_eq\_\_  
Equality check  
\_\_del\_\_  
Destructor  
  
  
10. isinstance(obj, Class): Checks if object is an instance of the given class or subclass.  
issubclass(SubClass, ParentClass): Checks if one class is a subclass of another.  
class Animal: pass  
class Dog(Animal): pass  
  
d = Dog()  
print(isinstance(d, Dog))         
print(isinstance(d, Animal))     
print(issubclass(Dog, Animal))   
print(issubclass(Animal, Dog))

o/p:

True

True

True

False

**Decorators**

1. A decorator is a function that modifies the behavior of another function or method without changing its source code.  
Use cases:  
Logging  
Access control  
Caching  
Timing execution  
Authorization checks  
  
2. def log\_decorator(func):  
   def wrapper(\*args, \*\*kwargs):  
       print(f"Function '{func.\_\_name\_\_}' was called")  
       return func(\*args, \*\*kwargs)  
   return wrapper  
  
@log\_decorator  
def say\_hello():  
   print("Hello!")  
  
say\_hello()

o/p:

Function 'say\_hello' was called

Hello!  
  
3.Yes, you can stack multiple decorators. Order of application is bottom-up (nearest decorator is applied first).  
  
4. functools.wraps() preserves the original function's metadata when it’s wrapped by a decorator.  
  
   
5. Original decorator:  
def log(func):  
   def wrapper(\*args, \*\*kwargs):  
       print("Logging")  
       return func(\*args, \*\*kwargs)  
   return wrapper  
Parameterized decorator:  
def log(prefix):  
   def decorator(func):  
       def wrapper(\*args, \*\*kwargs):  
           print(f"{prefix} - Logging")  
           return func(\*args, \*\*kwargs)  
       return wrapper  
   return decorator  
  
@log("DEBUG")  
def do\_task():  
   print("Doing task")  
  
6. def login\_required(func):  
   def wrapper(\*args, \*\*kwargs):  
       user = kwargs.get('user')  
       if user and user.get('logged\_in'):  
           return func(\*args, \*\*kwargs)  
       else:  
           print("Access Denied: User not logged in")  
   return wrapper  
  
@login\_required  
def view\_dashboard(\*args, \*\*kwargs):  
   print("Welcome to Dashboard")  
  
view\_dashboard(user={'logged\_in': True})     
view\_dashboard(user={'logged\_in': False})

o/p:

Welcome to Dashboard

Access Denied: User not logged in

7. The @property decorator allows you to access a method like an attribute, used for getter/setter behavior.  
Example:  
class Circle:  
   def \_\_init\_\_(self, radius):  
       self.\_radius = radius  
  
   @property  
   def area(self):  
       return 3.14 \* self.\_radius \*\* 2  
  
c = Circle(5)  
print(c.area)

o/p: 78.5  
  
8. def catch\_exceptions(func):  
   def wrapper(\*args, \*\*kwargs):  
       try:  
           return func(\*args, \*\*kwargs)  
       except Exception as e:  
           print(f"Exception in {func.\_\_name\_\_}: {e}")  
   return wrapper  
  
@catch\_exceptions  
def divide(a, b):  
   return a / b  
  
divide(5, 0)

o/p: 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. Yes, decorators work with both class methods and static methods.

**Generators**

1. A generator function is a special kind of function that uses \*\*yield\*\* instead of \*\*return\*\* to produce values one at a time, pausing between each.  
Generator function returns one value at a time but normal function returns entire result at once.

2. def even\_numbers():  
   for i in range(0, 21, 2):  
       yield i  
  
for num in even\_numbers():  
   print(num, end=' ')

o/p: 0 2 4 6 8 10 12 14 16 18 20  
  
3. It raises a StopIteration exception.  
gen = (x for x in range(2))  
print(next(gen))     
print(next(gen))     
print(next(gen))     
  
4. yield pauses function execution and remembers its state.  
Helps in memory efficiency by not storing the entire result in memory.  
def count\_up\_to(n):  
   for i in range(n):  
       yield i    
  
5. Generator Expression:  
gen = (x\*x for x in range(5))  
List Comprehension:  
lst = [x\*x for x in range(5)]  
Generator expression Uses() and list comprehension Uses[].  
  
6. Normal function:  
def get\_squares(n):  
   return [x\*x for x in range(n)]  
Generator version:  
def get\_squares(n):  
   for x in range(n):  
       yield x\*x  
  
7. To read a large file efficiently using a generator, you can create a function that uses yield to return one line at a time. This approach avoids loading the entire file into memory, making it suitable for big files.

8. Generators maintain state using a frame: Current position in code, Local variable and Execution context. Each call to next() resumes from the last yield, keeping previous values.  
  
9. return exits the function and it gets one time result, whereas yield pauses and returns a value and it gets multiple results over time.  
  
10. def gen\_func():  
   for i in range(3):  
       yield i  
print(list(gen\_func()))

o/p: [0, 1, 2]

list-generator function values generated on demand and it uses one time unless recalled, whereas In list-returning function all values are created at once and can use returned list multiple times.

**Iterators**

1. Iterable is an object that can return an iterator and can be looped using for. Iterator is an object with \_\_next\_\_() and \_\_iter\_\_() is used to fetch elements one at a time.

2. class CountToThree:  
   def \_\_init\_\_(self):  
       self.num = 1  
  
   def \_\_iter\_\_(self):  
       return self  
  
   def \_\_next\_\_(self):  
       if self.num > 3:  
           raise StopIteration  
       val = self.num  
       self.num += 1  
       return val  
  
for i in CountToThree():  
   print(i)

o/p:

1

2

3  
  
3. It signals that the iterator is exhausted. Internally, a for loop catches this to stop iterating. If you manually use next(), it will raise the exception.  
it = iter([1])  
print(next(it))       
print(next(it))       
  
4. import sys  
for line in iter(input, 'quit'):  
   print("You typed:", line)  
  
5. for x in iterable:  
      it = iter(iterable)  
   while True:  
       try:  
           x = next(it)  
       except StopIteration:  
           break  
It calls iter() to get an iterator, then repeatedly calls next() until StopIteration is raised.  
  
6. These return lazy iterators:  
-map(func, iterable)  
-filter(func, iterable)  
-zip(iter1, iter2)  
-enumerate(iterable)  
-reversed(seq)  
-iter(obj)  
-next(iterator)

it = map(lambda x: x \* 2, [1, 2, 3])  
print(next(it))     
o/p:2

7. nums = iter([10, 20, 30])  
while True:  
   try:  
       print(next(nums))  
   except StopIteration:  
       break

o/p:

10

20

30  
  
8. class SquareIterator:  
   def \_\_init\_\_(self):  
       self.num = 1  
  
   def \_\_iter\_\_(self):  
       return self  
  
   def \_\_next\_\_(self):  
       if self.num > 5:  
           raise StopIteration  
       val = self.num \*\* 2  
       self.num += 1  
       return val  
  
for n in SquareIterator():  
   print(n)

o/p:

1

4

9

16

25  
  
9. It will immediately raise StopIteration on every next() call.  
  
10. itertools provides fast, memory-efficient tools for working with iterators.  
It includes infinite iterators, combinatorics, filtering, grouping, etc.