**OOPs (Object-Oriented Programming) – 10 Questions**

1. What is the difference between self and cls in Python classes?

- self refers to the object of a class

- cls refers to the class itself

2. How does inheritance work in Python? Give an example with method overriding.

Inheritance allows a child class to inherit properties and behaviors from parent class. It promotes code reusability and supports method overriding and polymorphism.

```
class Animal:
    def sound(self):
        print("Animals make sound")
class Dog(Animal):
    def sound(self):
        print("Dogs Bark!")
class Cat(Animal):
    def sound(self):
        print("Cats Meow!")
a = Animal()
d = Dog()
c = Cat()
a.sound()
d.sound()
c.sound()
```

**Output :**

```
Animals make sound
Dogs Bark!
Cats Meow!
```

3. What is method overloading in Python? Is it supported natively?

Python doesn’t support method overloading natively. It can be mimicked using default arguments or *args.

4. Define constructor and destructor in Python. When are they called?

A constructor is a special method used to initialize objects when a class is instantiated. Constructor is called when object is created.

A destructor is a special method used to clean up resources before the object is destroyed. Destructor is called when object is deleted or program ends.

5. What is the difference between instance method, class method, and static method?

(i) Instance Method

- It is the most common type of method in a class.
- The first parameter is always self, which refers to the object calling the method.
- It can access and modify instance variables and call other instance methods.
- It is called using an object of the class, like obj.method().

(ii) Class Method

- A class method works at the class level rather than the instance level.
- It is defined using the @classmethod decorator.
- The first parameter is cls, which refers to the class itself, not an instance.
- It can access and modify class variables shared by all instances.
- It can be called using the class name or an object.

(iii) Static Method

- It is defined using the @staticmethod decorator.
- It does not take self or cls as a parameter.
- It cannot access or modify instance or class variables directly.
- It is like a regular function placed inside a class for logical grouping.

6. How do you restrict access to class attributes in Python (pseudo-private)?

We put two underscores __ before the variable name to make it private. It hides it from outside access.

7. Write a Python class to demonstrate encapsulation with getter/setter methods.

```
class ColorPalette:
    def __init__(self, color):
        self.__favorite_color = color  # private attribute
    def get_favorite_color(self):
        return self.__favorite_color
    def set_favorite_color(self, new_color):
        allowed_colors = ["red", "green", "blue", "yellow", "purple"]
        if new_color.lower() in allowed_colors:
            self.__favorite_color = new_color.lower()
            print(f"Favorite color set to {self.__favorite_color}")
        else:
            print("Invalid color! Choose from:", allowed_colors)
palette = ColorPalette("blue")
print("Current favorite color:", palette.get_favorite_color())
palette.set_favorite_color("green")
palette.set_favorite_color("orange")
print("Final favorite color:", palette.get_favorite_color())
```

**Output :**

```
Current favorite color: blue
Favorite color set to green
Invalid color! Choose from: ['red', 'green', 'blue', 'yellow', 'purple']
Final favorite color: green
```

8. What is polymorphism in Python? Show it with two unrelated classes using the same method name.

Polymorphism allows objects of different classes to be treated through a common interface, typically by defining methods with the same name but with behavior specific to each class.

```
class Cat:
    def speak(self):
        return "Cats Meow!"
class Dog:
    def speak(self):
        return "Dogs Bark!"
def make_speak(obj):
    print(obj.speak())
make_speak(Cat())
make_speak(Dog())
```

**Output :**

```
Cats Meow!
Dogs Bark!
```

9. What is a magic method? Name a few commonly used ones and their purpose.

Magic methods are special methods in Python with names that start and end with double underscores. They help control how objects behave.

__init__ - Constructor — called when an object is created

__str__ - string representation

__add__ - operator overloading Defines behavior for + operator

__del__() - Destructor — called when an object is about to be deleted

10. How do you use isinstance() and issubclass() functions?

- The isinstance() function is used to check if an object is an instance of a specific class.

- The issubclass() function checks if a class is a subclass of another class.

**Decorators – 10 Questions**

1. What is a decorator in Python and what is its typical use case?

A decorator is essentially a function that takes another function as an argument and returns a new function with enhanced functionality. Decorators are often used in scenarios such as logging, authentication and memorization, allowing us to add additional functionality to existing functions or methods in a clean, reusable way.

2. Write a simple decorator that logs when a function is called.

```
def log_function_name(func):
    def wrapper(*args, **kwargs):
        print(f"Calling function: {func.__name__}")
        return func(*args, **kwargs)
    return wrapper
@log_function_name
def welcome():
    print("Welcome!")
welcome()
```

**Output :**

```
Calling function: welcome
Welcome!
```

3. Can you apply more than one decorator to a function? In what order are they applied?

Yes, we can use more than one decorator. They are applied from bottom to top.

4. What is the use of functools.wraps() in a decorator?

The functools.wraps() decorator is used within a custom decorator to preserve the metadata of the original function that is being decorated. When a function is wrapped by another function, the attributes of the original function are typically replaced by those of the wrapper function.

5. Convert the following decorator to one that accepts arguments (parameterized decorator).

```
from functools import wraps
def logger(prefix):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(f"{prefix} calling {func.__name__} with {args} {kwargs}")
            return func(*args, **kwargs)
        return wrapper
    return decorator
@logger("LOG:")
def add(a, b, multiplier=1):
    return (a + b) * multiplier
print(add(100, 200, multiplier=2))
```

Output :

```
LOG: calling add with (100, 200) {'multiplier': 2}
600
```

6. How can you write a decorator to check if the user is logged in before accessing a function?

```
ALLOWED_USERS = ['admin' , 'manager']
def access_file(username, filename, lines_to_read):
    if username in ALLOWED_USERS:
        print(f"Access granted to '{username}'. Reading first {lines_to_read} lines from '{filename}':")
        with open(filename, 'r') as file:
            for i in range(lines_to_read):
                line = file.readline()
                if not line:
```

```
                break
            print(line.strip())
    else:
        print(f"Access denied for '{username}'.")
with open('important_data.txt', 'w') as f:
    for i in range(1, 6):
        f.write(f"Important Line {i}\n")
access_file('admin', 'important_data.txt', 3)
access_file('guest', 'important_data.txt', 3)
```

**Output :**

```
Access granted to 'admin'. Reading first 3 lines from 'important_data.txt':
Important Line 1
Important Line 2
Important Line 3
Access denied for 'guest'.
```

7. How does the @property decorator work? Give an example.

The @property decorator is used to define a method that can be accessed like an attribute. It allows getter, setter functionality while maintaining encapsulation.

```
import math
class Circle:
    def __init__(self, radius):
        self._radius = radius
    @property
    def radius(self):
        return self._radius
    @radius.setter
    def radius(self, value):
        if value <= 0:
            raise ValueError("Radius must be positive")
        self._radius = value
    @property
    def area(self):
        return math.pi * (self._radius ** 2)
c = Circle(5)
print("Radius:", c.radius)
print("Area:", c.area)
c.radius = 10
print("Updated Area:", c.area)
```

**Output :**

```
Radius: 5
Area: 78.53981633974483
Updated Area: 314.1592653589793
```

8. Write a decorator that catches and logs any exceptions in a function.

```
from functools import wraps
def catch_exceptions(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        try:
            return func(*args, **kwargs)
        except Exception as e:
            print(f"[ERROR] An exception occurred in '{func.__name__}': {e}")
            return None
    return wrapper
```

```
@catch_exceptions
def divide(a, b):
    return a / b
print(divide(10, 2))
print(divide(10, 0))
```

Output :

```
5.0
[ERROR] An exception occurred in 'divide': division by zero
None
```

9. What is the difference between function decorator and class decorator?

- Function decorators wrap functions.
- Class decorators wrap or modify class definitions.

10. Can decorators be used with class methods or static methods?

Yes, decorators can be used with class methods and static methods

**Generators – 10 Questions**

1. What is a generator function? How is it different from a normal function?

A generator function is a special type of function that can pause its execution, yield a value, and then resume execution from where it left off. This allows it to produce a sequence of values over time, rather than computing and returning all values at once.

Generator functions use the yield keyword instead of return. A normal function uses return to send back a value and terminate its execution.

2. Write a generator function to yield even numbers up to 20.

```
def even_numbers():
    for num in range(1, 21):
        if num % 2 == 0:
            yield num
for i in even_numbers():
    print(i)
```

**Output :**

```
2
4
6
8
10
12
14
16
18
20
```

3. What happens if you call next() on an exhausted generator?

We get StopIteration error when no more values are left.

4. What is the use of yield? How does it help in memory efficiency?

yield pauses the function and saves memory because it doesn't return the full list at once.

5. How do you use a generator expression? How is it different from list comprehension?

Generator uses ( ) and gives values one at a time. List comprehension uses [ ] and creates the whole list at once.

```
# List comprehension
squares_list = [x*x for x in range(5)]
print(squares_list)
```

```
# Generator expression
squares_gen = (x*x for x in range(5))
print(next(squares_gen))
print(next(squares_gen))
print(next(squares_gen))
print(next(squares_gen))
print(next(squares_gen))
```

**Output :**

```
[0, 1, 4, 9, 16]
0
1
4
9
16
```

6. Convert a normal function that returns a list into a generator.

```
def get_squares_list(n):
    squares = []
    for i in range(n):
        squares.append(i * i)
    return squares
def get_squares_generator(n):
    for i in range(n):
        yield i * i
print("Using normal function (returns list):")
square_list = get_squares_list(5)
```

```
print(square_list)
print("\nUsing generator function (yields values):")
for square in get_squares_generator(5):
    print(square)
```

**Output :**

```
Using normal function (returns list):
[0, 1, 4, 9, 16]
Using generator function (yields values):
0
1
4
9
16
```

7. How would you read a large file using a generator to process it line by line?

```
def read_file(filename):
    with open(filename) as f:
        for line in f:
            yield line
```

8. How does the generator maintain its state between calls?

The function pauses at yield and resumes on next call.

9. What is the difference between return and yield inside a function?

- return ends function.
- yield pauses and resumes function.

10. What is the output of list(generator_function()) and how does it differ from a list-returning function?

Generator gives items one by one, then list() collects them. Normal list-returning gives all values at once.

**Iterators – 10 Questions**

1. What is the difference between an iterable and an iterator?

An iterable is an object that can be iterated over, meaning it can return its members one at a time. Examples include lists, tuples, strings, and dictionaries in Python. An iterable implements the __iter__() method, which returns an iterator.

An iterator is an object that represents a stream of data. It is used to iterate over an iterable and keep track of the current position during iteration. An iterator implements both the __iter__() method and the __next__() method.

2. How do you make a class iterable using __iter__() and __next__()?

```
class Counter:
    def __init__(self, start, end):
        self.current = start
        self.end = end
    def __iter__(self):
        return self
    def __next__(self):
        if self.current <= self.end:
            value = self.current
            self.current += 1
            return value
        else:
            raise StopIteration
for i in Counter(1, 5):
    print(i)
```

**Output :**

```
1
2
3
```

```
4
5
```

3. Explain what happens when StopIteration is raised.

The StopIteration exception is automatically raised by an iterator to signal that there are no more items to return.

- StopIteration - it stops the iteration.

4. Give an example of using iter() with a sentinel value.

```
def get_input():
    return input("Enter something (type 'stop' to quit): ")
for item in iter(get_input, 'stop'):
    print(f"You entered: {item}")
```

**Output :**

```
Enter something (type 'stop' to quit): k
You entered: k
Enter something (type 'stop' to quit): stop
```

5. How does a for loop work internally with iterators?

It gets an iterator using iter(), then keeps calling next() until StopIteration.

- Calls iter() on the iterable to get an iterator object.
- Repeats:
  Calls next() on the iterator to get the next value.
  Assigns it to item.
  Executes the loop body.
- Stops when StopIteration is raised.

6. What built-in functions rely on iterators (e.g., map, zip, filter)?

Functions like map(), filter(), zip(), next() return iterators.

7. How to manually loop over an iterator using next()?

```
items = iter([1, 2, 3])
print(next(items))
print(next(items))
print(next(items))
```

**Output :**

```
1
2
3
```

8. Write a custom iterator that returns square of numbers from 1 to 5.

```
class Squares:
    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
        else:
            raise StopIteration
s = Squares()
for square in s:
    print(square)
```

**Output :**

```
1

4

9

16

25
```

9. What happens when you try to iterate over an already exhausted iterator?

An iterator is exhausted when it has gone through all its values and has raised StopIteration. Calling next() again will immediately raise StopIteration.

10. What is the use of the itertools module in python with itertools.

The itertools module in Python provides a collection of functions for creating and manipulating iterators efficiently. It is part of the standard library and is designed to be fast and memory-efficient, particularly when dealing with large datasets or complex iteration patterns.

Provides advanced iterators like count(), cycle(), combinations(), permutations(), etc.