ABHISHEK SAHANI

Assignment_4

1.What are the five key concepts of Object-Oriented Programming (OOP)?

= The five key concepts of Object-Oriented Programming (OOP) are:

1. Encapsulation: This concept involves hiding the implementation details of an object from the outside world and only exposing the necessary information through public methods.

2. Abstraction: Abstraction involves representing complex real-world objects or systems in a simplified way, focusing on essential features and behaviors.

3. Inheritance: Inheritance allows one class to inherit properties and methods from another class, creating a parent-child relationship.

4. Polymorphism: Polymorphism enables objects of different classes to be treated as if they were of the same class, using methods with the same name but different implementations.

5. Composition: Composition involves combining objects or classes to form new objects or classes, promoting modularity and reuse.

These concepts enable developers to design and develop modular, reusable, and maintainable software systems.

Here's an example in Python:

```
class Animal:  # Abstraction
    def __init__(self, name):  # Encapsulation
        self.name = name

    def sound(self):  # Polymorphism
        pass

class Dog(Animal):  # Inheritance
    def sound(self):  # Polymorphism
        return "Woof"

class Cat(Animal):  # Inheritance
    def sound(self):  # Polymorphism
        return "Meow"
```

```
class PetStore:  # Composition
    def __init__(self):
        self.animals = [Dog("Buddy"), Cat("Whiskers")]

    def make_sounds(self):
        for animal in self.animals:
            print(animal.sound())
```

2.Write a Python class for a 'Car with attributes for 'make', 'model', and 'year'. Include a method to display the car's information.

= Here's a Python class for a 'Car' with attributes for 'make', 'model', and 'year', along with a method to display the car's information:

```
class Car:
    def __init__(self, make, model, year):
        """
        Initializes a Car object.

        Args:
            make (str): The car's make.
            model (str): The car's model.
            year (int): The car's year.
        """
        self.make = make
        self.model = model
        self.year = year

    def display_info(self):
        """
        Displays the car's information.
        """
        print(f"Make: {self.make}")
        print(f"Model: {self.model}")
        print(f"Year: {self.year}")

# Example usage:
my_car = Car("Toyota", "Camry", 2022)
my_car.display_info()
```

Output:

```
Make: Toyota
Model: Camry
```

Year: 2022

This class includes:

1. An __init__ method to initialize the Car object with 'make', 'model', and 'year' attributes.
2. A display_info method to print the car's information.

You can create multiple Car objects and call the display_info method to display their information.

3.Explain the difference between instance methods and class methods. Provide an example of each.

= In Python, instance methods and class methods are two types of methods that can be defined within a class.

Instance Methods

Instance methods are bound to an instance of a class. They operate on the instance's attributes and are used to perform actions related to that instance.

Example:

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def honk(self):
        print("Honk honk!")

my_car = Car("Toyota", "Camry", 2022)
my_car.honk()  # Output: Honk honk!
```

In this example, honk is an instance method that operates on the my_car instance.

Class Methods

Class methods are bound to a class rather than an instance. They operate on class attributes and are used to perform actions related to the class.

Example:

```
class Car:
    num_cars = 0
```

```
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        Car.num_cars += 1

    @classmethod
    def count_cars(cls):
        print(f"Total cars: {cls.num_cars}")

my_car1 = Car("Toyota", "Camry", 2022)
my_car2 = Car("Honda", "Civic", 2021)
Car.count_cars()  # Output: Total cars: 2
```

In this example, count_cars is a class method that operates on the num_cars class attribute.

Key differences:

1. Binding: Instance methods are bound to an instance, while class methods are bound to a class.
2. Purpose: Instance methods operate on instance attributes, while class methods operate on class attributes.
3. Syntax: Class methods are decorated with @classmethod and take cls as the first argument.

When deciding between instance and class methods, ask yourself:

- Is the method operating on instance attributes? Use an instance method.
- Is the method operating on class attributes or performing a class-level action? Use a class method.

4.How does Python implement method overloading? Give an example.

= Python does not support method overloading in the classical sense, unlike languages such as Java or C++. However, Python provides alternative ways to achieve similar functionality.

Method Overloading Alternatives:

1. Default Argument Values:

```
```
```
def greet(name, msg='Hello'):
print(f"{msg}, {name}!")

greet('John')  # Output: Hello, John!
greet('John', 'Hi')  # Output: Hi, John!
```

2. **Variable Number of Arguments**:

```
def sum_nums(*numbers):
   return sum(numbers)

print(sum_nums(1, 2, 3))  # Output: 6
print(sum_nums(1, 2, 3, 4, 5))  # Output: 15
```

1. *Keyword Arguments*:

```
def greet(name, **kwargs):
msg = kwargs.get('msg', 'Hello')
print(f"{msg}, {name}!")

greet('John')  # Output: Hello, John!
greet('John', msg='Hi')  # Output: Hi, John!
```

4. *Single Dispatch* (using @singledispatch decorator from functools module):

```
from functools import singledispatch

@singledispatch
def fun(arg):
   return "default"

@fun.register(int)
def _(arg):
   return "int"

@fun.register(list)
def _(arg):
   return "list"

print(fun(1))  # Output: int
print(fun([1, 2]))  # Output: list
print(fun("hello"))  # Output: default
```

1. Polymorphism (using inheritance and method overriding):

```
```

```
class Shape:
def area(self):
pass

class Circle(Shape):
def init(self, radius):
self.radius = radius

def area(self):
    return 3.14 * self.radius ** 2

class Rectangle(Shape):
def init(self, width, height):
self.width = width
self.height = height

def area(self):
    return self.width * self.height

shapes = [Circle(5), Rectangle(3, 4)]
for shape in shapes:
print(shape.area())
```

5.What are the three types of access modifiers in Python? How are they denoted?

= Python has three types of access modifiers:

1. Public: No specific notation is used. Public members are accessible from anywhere.

2. Private: Denoted by double underscore (__) prefix. Private members are accessible only within the class.

3. Protected: Denoted by single underscore (_) prefix. Protected members are accessible within the class and its subclasses.

Here's an example:

```
class MyClass:
    def __init__(self):
        self.public_var = 10  # Public variable
        self._protected_var = 20  # Protected variable
        self.__private_var = 30  # Private variable

    def public_method(self):  # Public method
        print("Public method")
```

```
    def _protected_method(self):  # Protected method
        print("Protected method")

    def __private_method(self):  # Private method
        print("Private method")

obj = MyClass()
print(obj.public_var)  # Accessible
print(obj._protected_var)  # Accessible but discouraged
# print(obj.__private_var)  # Raises AttributeError
obj.public_method()  # Accessible
obj._protected_method()  # Accessible but discouraged
# obj.__private_method()  # Raises AttributeError
```

Note:

- Python's private access modifier is not strictly enforced but rather a convention.
- Name mangling (internal renaming) is used to avoid naming conflicts in subclasses.
- Protected access modifier is more of a convention and not enforced by Python.

Best practices:

- Use public access for methods and variables intended for external use.
- Use protected access for methods and variables intended for internal or subclass use.
- Use private access for sensitive or internal implementation details.

6. Describe the five types of inheritance in Python. Provide a simple example of multiple inheritance.

= Python supports the following five types of inheritance:

1. Single Inheritance: A child class inherits from a single parent class.

```
```
```
class Parent:
def method1(self):
print("Parent method")

class Child(Parent):
def method2(self):
print("Child method")

child = Child()
child.method1()  # Output: Parent method
child.method2()  # Output: Child method
```

2. **Multiple Inheritance**: A child class inherits from multiple parent classes.

```
class Parent1:
    def method1(self):
        print("Parent1 method")

class Parent2:
    def method2(self):
        print("Parent2 method")

class Child(Parent1, Parent2):
    pass

child = Child()
child.method1()  # Output: Parent1 method
child.method2()  # Output: Parent2 method
```

1. *Multilevel Inheritance*: A child class inherits from a parent class, which itself inherits from another parent class.

```
class Grandparent:
def method1(self):
print("Grandparent method")

class Parent(Grandparent):
def method2(self):
print("Parent method")

class Child(Parent):
def method3(self):
print("Child method")

child = Child()
child.method1()  # Output: Grandparent method
child.method2()  # Output: Parent method
child.method3()  # Output: Child method
```

4. *Hierarchical Inheritance*: Multiple child classes inherit from a single parent class.

```
class Parent:
```

```
    def method1(self):
        print("Parent method")

class Child1(Parent):
    def method2(self):
        print("Child1 method")

class Child2(Parent):
    def method3(self):
        print("Child2 method")

child1 = Child1()
child1.method1()  # Output: Parent method
child1.method2()  # Output: Child1 method

child2 = Child2()
child2.method1()  # Output: Parent method
child2.method3()  # Output: Child2 method
```

1. Hybrid Inheritance: Combination of multiple inheritance types.

```
```
```
class Grandparent:
def method1(self):
print("Grandparent method")

class Parent1(Grandparent):
def method2(self):
print("Parent1 method")

class Parent2(Grandparent):
def method3(self):
print("Parent2 method")

class Child(Parent1, Parent2):
def method4(self):
print("Child method")

child = Child()
child.method1()  # Output: Grandparent method
child.method2()  # Output: Parent1 method
child.method3()  # Output: Parent2 method
child.method4()  # Output: Child method
```

7. What is the Method Resolution Order (MRO) in Python? How can you retrieve it programmatically?

= The Method Resolution Order (MRO) in Python is the order in which the interpreter searches for methods and attributes in a class hierarchy. It's used to resolve conflicts when multiple classes define the same method or attribute.

Python uses a depth-first left-to-right (DFLR) approach to resolve method calls. The MRO is calculated using the C3 linearization algorithm.

Here's an example:

```
class A:
    def method(self):
        print("A")

class B(A):
    def method(self):
        print("B")

class C(A):
    def method(self):
        print("C")

class D(B, C):
    pass

d = D()
d.method()  # Output: B
```

In this example, the MRO for class D is:

1. D
2. B
3. C
4. A
5. object (the base class of all Python objects)

To retrieve the MRO programmatically, you can use the mro() method:

```
print(D.mro())
# Output: (__main__.D, __main__.B, __main__.C, __main__.A, object)
```

Alternatively, you can use the inspect module:

```
import inspect
print(inspect.getmro(D))
# Output: (__main__.D, __main__.B, __main__.C, __main__.A, object)
```

Understanding the MRO is crucial when working with multiple inheritance in Python, as it helps resolve method conflicts and ensures predictable behavior.

8. Create an abstract base class 'Shape' with an abstract method 'area(). Then create two subclasses "Circle' and 'Rectangle that implement the area()` method.

= Here's how you can define the abstract base class Shape and its subclasses Circle and Rectangle using Python's abc (Abstract Base Classes) module:

```
from abc import ABC, abstractmethod
import math

# Abstract base class
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

# Subclass: Circle
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * (self.radius ** 2)
```

```
# Subclass: Rectangle
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

# Example usage
circle = Circle(5)
rectangle = Rectangle(4, 6)

print(f"Circle area: {circle.area():.2f}")
print(f"Rectangle area: {rectangle.area()}")
```

Output:

```
Circle area: 78.54
Rectangle area: 24
```

In this code:

- We import the ABC and abstractmethod from the abc module.
- We define Shape as an abstract base class using ABC.
- We declare area() as an abstract method using @abstractmethod.
- We create Circle and Rectangle subclasses implementing area().
- We demonstrate instance creation and area() method calls.

This design ensures that any shape subclass must implement the area() method, enforcing polymorphism and code consistency.

9. Demonstrate polymorphism by creating a function that can work with different shape objects to calculate and print their areas

= Here's an example demonstrating polymorphism with shape objects:

```
from abc import ABC, abstractmethod
import math

# Abstract base class
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

# Subclass: Circle
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * (self.radius ** 2)

# Subclass: Rectangle
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
```

```
    def area(self):
        return self.width * self.height

# Subclass: Triangle
class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height

    def area(self):
        return 0.5 * self.base * self.height

# Polymorphic function
def calculate_and_print_area(shape: Shape):
    print(f"Area of {type(shape).__name__}: {shape.area():.2f}")

# Example usage
circle = Circle(5)
rectangle = Rectangle(4, 6)
triangle = Triangle(3, 7)

calculate_and_print_area(circle)
calculate_and_print_area(rectangle)
calculate_and_print_area(triangle)
```"

Output:

Area of Circle: 78.54

Area of Rectangle: 24.00

Area of Triangle: 10.50

In this example:

- We define shape classes (`Circle`, `Rectangle`, `Triangle`) with an `area()` method.
- The `calculate_and_print_area()` function takes a `Shape` object as input.
- Using polymorphism, this function works with different shape objects.
- We demonstrate instance creation and function calls.

This design showcases polymorphism's power in handling diverse objects through a unified interface, making code more flexible and maintainable

10. Implement encapsulation in a 'BankAccount class with private attributes for balance and 'account_number'. Include methods for deposit, withdrawal, and balance inquiry.

= Here's an implementation of encapsulation in a BankAccount class:

```
class BankAccount:
    def __init__(self, account_number, initial_balance):
        self.__account_number = account_number
        self.__balance = initial_balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited ${amount}. Current balance: ${self.__balance:.2f}")
        else:
            print("Invalid deposit amount.")
```

```
    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew ${amount}. Current balance: ${self.__balance:.2f}")
        else:
            print("Insufficient funds or invalid withdrawal amount.")

    def get_balance(self):
        print(f"Account balance: ${self.__balance:.2f}")

    def get_account_number(self):
        return self.__account_number

# Example usage
account = BankAccount("123456789", 1000.0)
print(f"Account Number: {account.get_account_number()}")
account.get_balance()
account.deposit(500.0)
account.withdraw(200.0)
account.get_balance()
```

Output:

```
Account Number: 123456789
Account balance: $1000.00
Deposited $500. Current balance: $1500.00
Withdrew $200. Current balance: $1300.00
Account balance: $1300.00
```

In this implementation:

* We define private attributes `__account_number` and `__balance` using double underscore prefix.
* Public methods `deposit`, `withdraw`, and `get_balance` provide controlled access to private attributes.
* The `get_account_number` method allows read-only access to the account number.
* Validation checks ensure valid deposit and withdrawal amounts.
* Example usage demonstrates encapsulation benefits.

Encapsulation ensures:

* Data hiding: Sensitive data is hidden from external access.
* Data protection: Data modification is controlled through public methods.
* Code organization: Clear separation of concerns between data and behavior.

11. Write a class that overrides the_str_and_add_magic methods. What will these methods allow you to do?

= Here's an example class that overrides the __str__ and __add__ magic methods:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

    def __add__(self, other):
        if isinstance(other, Vector):
```

```
            return Vector(self.x + other.x, self.y + other.y)
        else:
            raise TypeError("Unsupported operand type for +")

# Example usage
v1 = Vector(2, 3)
v2 = Vector(4, 5)

print(v1)  # Output: Vector(2, 3)
print(v1 + v2)  # Output: Vector(6, 8)
```

In this example:

* The `__str__` method returns a string representation of the `Vector` object.
* The `__add__` method allows vector addition using the `+` operator.

Overriding these magic methods enables:

* **Human-readable representation**: `__str__` provides a clear, readable string representation of the object.
* **Operator overloading**: `__add__` allows using the `+` operator for vector addition.

Other commonly overridden magic methods include:

* `__eq__` (equality comparison)
* `__lt__` (less than comparison)
* `__gt__` (greater than comparison)
* `__mul__` (multiplication)
* `__truediv__` (division)

These methods enhance the usability and readability of your classes.

Additional Example

To demonstrate the usefulness of these magic methods, consider a simple `ComplexNumber` class:

python

```
class ComplexNumber:
    def _init_(self, real, imag):
        self.real = real
        self.imag = imag

    def _str_(self):
        return f"{self.real} + {self.imag}i"

    def _add_(self, other):
        if isinstance(other, ComplexNumber):
            return ComplexNumber(self.real + other.real, self.imag + other.imag)
        else:
            raise TypeError("Unsupported operand type for +")

    def _mul_(self, other):
        if isinstance(other, ComplexNumber):
            return ComplexNumber(
                self.real * other.real - self.imag * other.imag,
                self.real * other.imag + self.imag * other.real
            )
        else:
            raise TypeError("Unsupported operand type for *")

# Example usage
```

```
c1 = ComplexNumber(3, 4)
c2 = ComplexNumber(2, 1)

print(c1)  # Output: 3 + 4i
print(c1 + c2)  # Output: 5 + 5i
print(c1 * c2)  # Output: 2 + 11i
```

12. Create a decorator that measures and prints the execution time of a function.

= Here's an example of a decorator that measures and prints the execution time of a function:

```
import time
from functools import wraps

def timer_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"Function '{func.__name__}' executed in {execution_time:.6f} seconds.")
        return result
    return wrapper

# Example usage
@timer_decorator
def example_function():
    time.sleep(1)  # Simulate some work
    print("Function executed.")
```

example_function()

``

Output:

Function executed.

Function 'example_function' executed in 1.000143 seconds.

In this code:

* We import `time` for timing and `wraps` from `functools` for preserving the original function's metadata.
* The `timer_decorator` function takes a function `func` as input and returns a decorated `wrapper` function.
* The `wrapper` function measures the execution time, calls the original function, and prints the execution time.
* We apply the decorator to `example_function` using `@timer_decorator`.
* Example usage demonstrates the decorator's functionality.

This decorator provides:

* Non-intrusive timing measurement
* Flexible application to any function
* Clear reporting of execution time

You can reuse this decorator across your codebase to easily measure function execution times.

13. Explain the concept of the Diamond Problem in multiple inheritance. How does Python resolve it?

=The Diamond Problem is a classic issue in multiple inheritance, where a class inherits conflicting methods or attributes from its parent classes. This problem arises when:

1. Class A inherits from classes B and C.
2. Classes B and C have a common base class, D.
3. Classes B and C override a method or attribute from class D differently.

Here's an example:

```
class D:
    def method(self):
        print("D's method")

class B(D):
    def method(self):
        print("B's method")

class C(D):
    def method(self):
        print("C's method")

class A(B, C):
    pass

a = A()
a.method()  # Which method should be called?
```

In this scenario, class A inherits conflicting methods from classes B and C.

Python resolves the Diamond Problem using:

1. Method Resolution Order (MRO): Python uses a depth-first left-to-right approach to resolve method calls.

2. C3 Linearization Algorithm: Python employs the C3 algorithm to linearize the inheritance graph, ensuring consistent ordering.

Python's MRO resolves the Diamond Problem as follows:

1. Class A's MRO: A -> B -> C -> D -> object

2. When calling a.method(), Python searches for the method in this order.

In the example above, Python will call B's method because it appears first in the MRO.

To visualize the MRO, use the mro() method:

```
print(A.mro())  # Output: (__main__.A, __main__.B, __main__.C, __main__.D, object)
```

By understanding Python's MRO and C3 algorithm, you can effectively manage multiple inheritance and resolve potential conflicts.

Best Practices:

- Avoid deep inheritance hierarchies.
- Use multiple inheritance judiciously.
- Ensure clear method and attribute naming.
- Understand Python's MRO and C3 algorithm.

By following these guidelines, you can harness the power of multiple inheritance while minimizing potential conflicts.

14. Write a class method that keeps track of the number of instances created from a class.

=Here's an example of a class method that keeps track of the number of instances created from a class:

```
class InstanceTracker:
    instances = 0  # Class attribute to store instance count

    def __init__(self):
        InstanceTracker.instances += 1  # Increment instance count

    @classmethod
    def get_instance_count(cls):
        return cls.instances  # Return instance count

# Example usage
print(InstanceTracker.get_instance_count())  # Output: 0

obj1 = InstanceTracker()
print(InstanceTracker.get_instance_count())  # Output: 1

obj2 = InstanceTracker()
print(InstanceTracker.get_instance_count())  # Output: 2

obj3 = InstanceTracker()
print(InstanceTracker.get_instance_count())  # Output: 3
```

In this code:

- We define a class InstanceTracker with a class attribute instances initialized to 0.
- The __init__ method increments the instances count each time a new instance is created.
- The get_instance_count class method returns the current instance count.

This implementation provides:

- Accurate tracking of instance creation
- Easy access to instance count through class method
- Memory efficiency since only one counter is maintained

Alternative Implementation using __new__ method:

```
class InstanceTracker:
    instances = 0

    def __new__(cls):
        cls.instances += 1
        return super().__new__(cls)

    @classmethod
    def get_instance_count(cls):
        return cls.instances

# Example usage remains the same
```

In this version:

- We override the __new__ method to increment instance count before creating a new instance.
- super().__new__(cls) ensures proper instance creation.

Both implementations achieve the same goal but using different approaches.

15. Implement a static method in a class that checks if a given year is a leap year.

= Here's an implementation of a static method in a class that checks if a given year is a leap year:

```
class DateUtil:
    @staticmethod
    def is_leap_year(year):
        """
        Checks if a given year is a leap year.

        Args:
            year (int): The year to check.

        Returns:
            bool: True if the year is a leap year, False otherwise.
        """
        return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)

# Example usage
print(DateUtil.is_leap_year(2020))  # Output: True
print(DateUtil.is_leap_year(2019))  # Output: False
print(DateUtil.is_leap_year(1900))  # Output: False
print(DateUtil.is_leap_year(2000))  # Output: True
```

In this code:

- We define a class DateUtil with a static method is_leap_year.
- The is_leap_year method checks if a given year is divisible by 4, but not by 100, unless it's also divisible by 400.
- This logic follows the Gregorian calendar's leap year rules.

Benefits of using a static method:

- Utility method that doesn't require instance creation.
- Easy to use and understand.
- Encapsulates leap year logic within the DateUtil class.

Note: This implementation assumes the input year is an integer. You may want to add error handling for non-integer inputs.

Alternative Implementation using Python's built-in calendar module:

```
import calendar

class DateUtil:
    @staticmethod
    def is_leap_year(year):
        return calendar.isleap(year)

# Example usage remains the same
```

In this version:

- We import the calendar module and use its isleap function.

- This approach is more concise and leverages Python's built-in functionality.