Q1:- Explain the importance of functions

ans:-Functions are fundamental in many areas of mathematics, programming, and science. Here's a breakdown of their importance:

1. **Modularity and Reusability**:
   - In programming, functions allow you to encapsulate code into reusable blocks. This makes your code cleaner, easier to manage, and avoids duplication. For example, if you have a function to calculate the area of a rectangle, you can use it wherever you need to perform that calculation, without rewriting the logic each time.

2. **Abstraction and Simplification**:
   - Functions help abstract complex operations. You can use a function to perform a complex series of steps and then interact with it using a simple name. This makes it easier to understand and work with code or mathematical expressions by breaking them into smaller, more manageable parts.

3. **Mathematical Relationships**:
   - In mathematics, functions describe relationships between variables. They allow us to model real-world phenomena and solve problems by representing how one quantity changes with respect to another. For example, the function \(f(x) = x^2 \) represents the relationship where the output is the square of the input.

4. **Problem Solving**:
   - Functions are crucial for solving problems systematically. In programming, breaking down a problem into smaller functions that each handle a specific task can make it easier to solve complex issues. In mathematics, functions provide a way to express and solve equations and inequalities.

5. **Predictability and Control**:
   - Functions ensure consistent behavior. Once a function is defined, it will perform the same operation every time it is called with the same inputs. This predictability is essential in both programming and mathematical modeling.

Overall, functions are a powerful concept that underpins much of modern computing and mathematical analysis. They enable us to structure complex problems, enhance code efficiency, and maintain clarity in both programming and mathematical contexts.

Q 2:-Write a basic function to greet student

Ans def greet_student(name):
    "
    Function to greet a student with a personalized message.”

# Example usage:
student_name = "Alice"
print(greet_student(student_name))

Q3:-Write a difference between print and return statements

Ans :- `print` **Statement**

- **Purpose**: Outputs information to the console or standard output.
- **Usage**: Used for displaying information to the user or for debugging purposes.
- **Effect**: Does not affect the value of expressions or the control flow of the program beyond showing text.
- **Output**: Directly shows the result on the screen

`return` Statement

- **Purpose**: Sends a value from a function back to the caller.
- **Usage**: Used to provide the result of a function so it can be used elsewhere in the program.
- **Effect**: Terminates the function's execution and optionally sends back a value to the calling context.
- **Output**: Does not display the result directly but allows the function to output a value that can be assigned to a variable or used in further calculations.

Q4:- what are *args and **kwargs?

Ans :`*args`

- **Purpose**: Allows a function to accept an arbitrary number of positional arguments.
- **Syntax**: `*args` is used in a function definition preceded by an asterisk.
- **How It Works**: `*args` collects extra positional arguments as a tuple. This means that any number of positional arguments passed to the function will be stored in a tuple named `args`.
- **Usage**: Useful when you don't know beforehand how many arguments will be passed to the function.

`**kwargs`

- **Purpose**: Allows a function to accept an arbitrary number of keyword arguments.
- **Syntax**: `**kwargs` is used in a function definition preceded by two asterisks.
- **How It Works**: `**kwargs` collects extra keyword arguments as a dictionary. This means that any number of keyword arguments passed to the function will be stored in a dictionary named `kwargs`, where the keys are argument names and the values are the corresponding argument values.
- **Usage**: Useful when you want to accept named arguments that may not be known in advance.

Q5:-explain the iterator function

Ans: an iterator is an object that allows you to traverse through a sequence of elements, such as items in a list or characters in a string, without needing to know the underlying details of how the sequence is stored.

Example :-class NumberIterator:

```
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.end:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

# Example usage
number_iter = NumberIterator(1, 5)

for number in number_iter:
    print(number)
```

Q6:- write a code that generate the square of a number from 1 to n using generator

Ans:- def squares_generator(n):

```
    ""
    A generator function that yields the square of numbers from 1 to n.”’
```

```
    Parameters:
    n (int): The upper limit of the range (inclusive).

    Yields:
    int: The square of the current number in the range.
    """
    for i in range(1, n + 1):
        yield i ** 2

# Example usage
n = 5
for square in squares_generator(n):
    print(square)
```

Q7:-write a code that generate palindromic number upto n using a generator

ans:-
```
def is_palindrome(num):

    num (int): The number to check.

    Returns:
    bool: True if the number is a palindrome, False otherwise.
    """
    return str(num) == str(num)[::-1]

def palindromic_numbers_generator(n):
    for num in range(1, n + 1):
        if is_palindrome(num):
            yield num

# Example usage
n = 100
for palin in palindromic_numbers_generator(n):
    print(palin)
```

Q8:-write a code that generate even number from 2 to n using a generator

Ans:-
```
def even_numbers_generator(n):
    """
    A generator function that yields even numbers from 2 to n (inclusive).

    Parameters:
    n (int): The upper limit of the range (inclusive).

    Yields:
    int: An even number within the specified range.
```

```
    """
    for num in range(2, n + 1, 2):
        yield num

# Example usage
n = 20
for even in even_numbers_generator(n):
    print(even)
```

Q9:-write a code that generate powers of two upto n using a generator

Ans:-

```
def powers_of_two(n):
    power = 1
    while power <= n:
        yield power
        power *= 2

# Example usage:
limit = 100
for value in powers_of_two(limit):
    print(value)
```

Q10:-write a code that generate prime numbers upto n using a generator

Ans:-

```
def is_prime(num):
    """Check if a number is prime."""
    if num <= 1:
        return False
    if num <= 3:
        return True
    if num % 2 == 0 or num % 3 == 0:
        return False
    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:
            return False
        i += 6
    return True

def prime_numbers_up_to(n):
    """Generate prime numbers up to n."""
    for num in range(2, n + 1):
        if is_prime(num):
            yield num
```

```
# Example usage:
limit = 50
for prime in prime_numbers_up_to(limit):
    print(prime)
```

Q11:- write a code using lambda function to calculate sum of two numbers

ans:

```
sum_two_numbers = lambda a, b: a + b
 # Example usage num1 = 5
num2 = 7
result = sum_two_numbers(num1, num2)
 print(f"The sum of {num1} and {num2} is {result}.")
```

Q12:-write a code using lambda function to calculate square of given numbers

ans:-

```
square = lambda x: x ** 2

# Example usage
number = 4
result = square(number)

print(f"The square of {number} is {result}.")
```

Q13:-write a code using lambda function to check whether a given numbers is even or odd

Ans:-

```
# Define the lambda function to check if a number is even
is_even = lambda x: x % 2 == 0

# Define the lambda function to check if a number is odd
is_odd = lambda x: x % 2 != 0

# Example usage
number = 7

if is_even(number):
    print(f"{number} is even.")
elif is_odd(number):
    print(f"{number} is odd.")
```

Q15:--write a code using lambda function to concatenate to string functions

Ans:-

```
# Define the lambda function to concatenate two strings
concatenate_strings = lambda str1, str2: str1 + str2

# Example usage
string1 = "Hello, "
string2 = "world!"
result = concatenate_strings(string1, string2)

print(result)
```

Q16:-write a code using lambda function to find maximum of three given numbers

Ans:

```
# Define the lambda function to find the maximum of three numbers
max_of_three = lambda a, b, c: max(a, b, c)

# Example usage
num1 = 10
num2 = 25
num3 = 15
result = max_of_three(num1, num2, num3)

print(f"The maximum of {num1}, {num2}, and {num3} is {result}.")
```

Q17:-write a code that generates the squares of even numbers from a given list

Ans

```
# Define a list of numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Define a lambda function to check if a number is even
is_even = lambda x: x % 2 == 0

# Use a list comprehension to generate squares of even numbers
squares_of_evens = [x ** 2 for x in numbers if is_even(x)]

# Print the result
print("Squares of even numbers:", squares_of_evens)
```

Q18:-write a code that calculate the product of positive numbers from a given list

Ans:-

```
from functools import reduce

# Define a list of numbers
numbers = [1, -2, 3, 4, -5, 6]

# Filter the positive numbers and calculate their product
product_of_positives = reduce(lambda x, y: x * y, (num for num in numbers if num > 0), 1)

# Print the result
print("Product of positive numbers:", product_of_positives)
```

Q19:-write a code that doubles the value of odd numbers from a given list

Ans

```
# Define a list of numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Double the value of odd numbers using list comprehension
doubled_odds = [x * 2 if x % 2 != 0 else x for x in numbers]

# Print the result
print("List with doubled odd numbers:", doubled_odds)
```

Q20:- write a code that calculate the sum of cube of a given numbers from a given list

Ans :

```
# Define a list of numbers
numbers = [1, 2, 3, 4, 5]

# Calculate the sum of the cubes of the numbers using list comprehension
sum_of_cubes = sum(x ** 3 for x in numbers)

# Print the result
print("Sum of the cubes of the numbers:", sum_of_cubes)
```

Q21:-write a code that filter out prime number from a given list
Ans

```
def is_prime(num):
    """Check if a number is prime."""
    if num <= 1:
```

```
        return False
    if num <= 3:
        return True
    if num % 2 == 0 or num % 3 == 0:
        return False
    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:
            return False
        i += 6
    return True

# Define a list of numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Filter out prime numbers using list comprehension
prime_numbers = [num for num in numbers if is_prime(num)]

# Print the result
print("Prime numbers from the list:", prime_numbers)
```

Q27:-what is encapsulation in OOP?
Ans
Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It refers to the practice of bundling the data (attributes) and methods (functions) that operate on the data into a single unit, known as a class. Encapsulation also involves restricting direct access to some of an object's components, which is often achieved through access modifiers (like private, protected, and public).

Q28:-explain the access modifiers in python classes?

Ans

In Python, access modifiers are used to control the visibility and accessibility of class attributes and methods. While Python does not have formal access modifiers like some other languages (e.g., `private`, `protected`, `public` in Java), it uses naming conventions to achieve similar functionality.

Q29:-what is inheritance in OOP?

Ans

Inheritance is a core principle of Object-Oriented Programming (OOP) that allows one class (known as the **subclass** or **derived class**) to inherit attributes and methods from another class (known as the **superclass** or **base class**). This mechanism promotes code reuse,

helps in organizing code hierarchically, and allows for the creation of a new class based on an existing class with some additional or modified features

Q30:-what is polymorphism in OOP?

Ans

Polymorphism is a core concept in Object-Oriented Programming (OOP) that allows objects of different classes to be treated as objects of a common base class. It enables a single interface to be used for a variety of data types, which can be implemented in different ways. Polymorphism allows for flexibility and the ability to extend and maintain code more easily.

Q31:- explain method of overwriting in python?

Ans

In Python, method overriding refers to the process where a subclass provides a specific implementation of a method that is already defined in its base class. This allows the subclass to modify or extend the behavior of the inherited method, tailoring it to its specific needs while still retaining the general structure and interface of the base class.

Q32:- Define a parent class Animal with a method make_sound that prints "Generic animal sound". Create a child class Dog inheriting from Animal with a method make_sound that prints "Woof!".

Ans:-

```
class Animal:
    def make_sound(self):
        print("Generic animal sound")

# Define the child class Dog inheriting from Animal
class Dog(Animal):
    def make_sound(self):
        print("Woof!")

# Example usage
animal = Animal()
animal.make_sound()  # Output: Generic animal sound

dog = Dog()
dog.make_sound()  # Output: Woof!
```

Q33:-Define a method move in the Animal class that prints "Animal moves". Override the move method in the Dog class to print "Dog runs."

Ans:-

```
# Define the parent class Animal
class Animal:
    def move(self):
        print("Animal moves")

# Define the child class Dog inheriting from Animal
class Dog(Animal):
    def move(self):
        print("Dog runs")

# Example usage
animal = Animal()
animal.move()  # Output: Animal moves

dog = Dog()
dog.move()  # Output: Dog runs
```

34. Create a class Mammal with a method reproduce that prints "Giving birth to live young." Create a class Dog Mammal inheriting from both Dog and Mammal

Ans:

```
# Define the parent class Animal
class Animal:
    def move(self):
        print("Animal moves")

# Define the class Mammal
class Mammal:
    def reproduce(self):
        print("Giving birth to live young.")

# Define the class Dog inheriting from both Animal and Mammal
class Dog(Animal, Mammal):
    def move(self):
        print("Dog runs.")

    def bark(self):
        print("Woof!")

# Example usage
dog = Dog()

# Using methods from Animal
dog.move()  # Output: Dog runs.
```

```
# Using methods from Mammal
dog.reproduce()  # Output: Giving birth to live young.

# Using additional methods from Dog
dog.bark()  # Output: Woof!
```

35. Create a class German Shepherd inheriting from Dog and override the make_sound method to print "Bark!"

Ans

```
# Define the Dog class with a make_sound method
class Dog:
    def make_sound(self):
        print("Woof!")

# Define the GermanShepherd class inheriting from Dog
class GermanShepherd(Dog):
    def make_sound(self):
        print("Bark!")  # Override the method to provide a specific sound

# Example usage
dog = Dog()
dog.make_sound()  # Output: Woof!

german_shepherd = GermanShepherd()
german_shepherd.make_sound()  # Output: Bark!
```

36. Define constructors in both the Animal and Dog classes with different initialization parameters.

Ans:-

```
class Animal:
    def __init__(self, species, age):
        self.species = species
        self.age = age

    def __str__(self):
        return f"Animal(species={self.species}, age={self.age})"

class Dog(Animal):
    def __init__(self, species, age, breed, name):
        # Call the constructor of the parent class
        super().__init__(species, age)
        self.breed = breed
```

```
        self.name = name

    def __str__(self):
        return f"Dog(name={self.name}, breed={self.breed}, species={self.species},
age={self.age})"

# Example usage
animal = Animal(species="Elephant", age=10)
print(animal)  # Output: Animal(species=Elephant, age=10)

dog = Dog(species="Canine", age=5, breed="Golden Retriever", name="Buddy")
print(dog)  # Output: Dog(name=Buddy, breed=Golden Retriever, species=Canine, age=5)
```

37. What is abstraction in Python? How is it implemented?
Ans:

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Example usage
def animal_sound(animal):
    print(animal.make_sound())

dog = Dog()
cat = Cat()

animal_sound(dog)  # Output: Woof!
animal_sound(cat)  # Output: Meow!
```

38. Explain the importance of abstraction in object-oriented programming.
Ans:

Abstraction is crucial in object-oriented programming because it:

1. **Simplifies Complexity**: By hiding complex implementation details and exposing only necessary features, abstraction makes it easier to interact with objects and understand their behavior.
2. **Enhances Maintainability**: Changes in the implementation of a class do not affect code that relies on the abstract interface, reducing the risk of breaking existing code.
3. **Improves Reusability**: Abstract classes and interfaces allow for the creation of reusable code components that can be extended and customized, promoting efficient code reuse and modular design.
4. **Facilitates Code Organization**: It helps in organizing code into logical sections with clear responsibilities, making the design more intuitive and manageable.

39. How are abstract methods different from regular methods in Python?

Ans

Abstract Methods:

- **Definition**: Declared in an abstract class using the `@abstractmethod` decorator from the `abc` module.
- **Implementation**: Do not have an implementation in the abstract class; they define a method signature that must be implemented by subclasses.
- **Purpose**: Serve as a blueprint for subclasses, enforcing a contract that subclasses must follow, thereby promoting a consistent interface.

Regular Methods:

- **Definition**: Defined in regular classes and can be implemented with specific behavior.
- **Implementation**: Have their own implementation directly in the class where they are defined.
- **Purpose**: Provide functionality and behavior for the class instances, which can be directly used or overridden by subclasses if needed.

Example:-

```
from abc import ABC, abstract method

class MyAbstractClass(ABC):

    @abstractmethod

    def abstract_method(self):

        pass  # No implementation here



    def regular_method(self):

        return "This is a regular method."
```

```
class MyConcreteClass(MyAbstractClass):
    def abstract_method(self):
        return "Implemented abstract method."

    def regular_method(self):
        return "Overridden regular method."

# Example usage
obj = MyConcreteClass()
print(obj.abstract_method())  # Output: Implemented abstract method.
print(obj.regular_method())   # Output: Overridden regular method.
```

40. How can you achieve abstraction using interfaces in Python?

Ans

Here's how you can use abstract base classes to achieve abstraction, which simulates the concept of interfaces:

1. **Import the ABC Module**: Use the `abc` module to work with abstract base classes.
2. **Define an Abstract Base Class**: Create a class that inherits from `ABC` and use the `@abstractmethod` decorator to define abstract methods.
3. **Implement the Abstract Base Class**: Subclasses of the abstract base class must provide implementations for all abstract methods.

```
from abc import ABC, abstractmethod

# Define an abstract base class with abstract methods
class Shape(ABC):
    @abstractmethod
    def area(self):
        """Method to calculate the area of the shape"""
```

```
        pass

    @abstractmethod
    def perimeter(self):
        """Method to calculate the perimeter of the shape"""
        pass

# Implement concrete subclasses
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

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

    def perimeter(self):
        return 2 * (self.width + self.height)

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

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

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

# Example usage
def print_shape_info(shape):
    print(f"Area: {shape.area()}")
    print(f"Perimeter: {shape.perimeter()}")

rectangle = Rectangle(3, 4)
circle = Circle(5)

print("Rectangle Info:")
print_shape_info(rectangle)

print("\nCircle Info:")
print_shape_info(circle)
```

41. Can you provide an example of how abstraction can be utilized to create a common interface for a group of related classes in Python?

Ans

Certainly! Abstraction in Python can be effectively utilized to create a common interface for a group of related classes by defining an abstract base class (ABC) that outlines a set of methods. All related classes can then implement this abstract base class, ensuring they adhere to the common interface.

Example:

```
from abc import ABC, abstractmethod
```

```
# Define an abstract base class with a common interface
class Employee(ABC):
    @abstractmethod
    def calculate_salary(self):
        """Calculate the salary of the employee"""
        pass

    @abstractmethod
    def get_details(self):
        """Return a summary of the employee details"""
        pass

# Concrete subclass for a Salaried Employee
class SalariedEmployee(Employee):
    def __init__(self, name, annual_salary):
        self.name = name
        self.annual_salary = annual_salary

    def calculate_salary(self):
        return self.annual_salary

    def get_details(self):
        return f"Salaried Employee: {self.name}, Annual Salary: ${self.annual_salary}"

# Concrete subclass for an Hourly Employee
class HourlyEmployee(Employee):
```

```
    def __init__(self, name, hourly_rate, hours_worked):
        self.name = name
        self.hourly_rate = hourly_rate
        self.hours_worked = hours_worked

    def calculate_salary(self):
        return self.hourly_rate * self.hours_worked

    def get_details(self):
        return f"Hourly Employee: {self.name}, Hourly Rate: ${self.hourly_rate}, Hours Worked: {self.hours_worked}"

# Concrete subclass for a Commissioned Employee
class CommissionedEmployee(Employee):
    def __init__(self, name, base_salary, commission_rate, sales_amount):
        self.name = name
        self.base_salary = base_salary
        self.commission_rate = commission_rate
        self.sales_amount = sales_amount

    def calculate_salary(self):
        return self.base_salary + (self.commission_rate * self.sales_amount)

    def get_details(self):
        return (f"Commissioned Employee: {self.name}, Base Salary: ${self.base_salary}, "
                f"Commission Rate: {self.commission_rate*100}%, Sales Amount: ${self.sales_amount}")
```

```
# Example usage
def print_employee_info(employee):
    print(employee.get_details())
    print(f"Salary: ${employee.calculate_salary()}")

salaried_emp = SalariedEmployee("Alice", 60000)
hourly_emp = HourlyEmployee("Bob", 25, 160)
commissioned_emp = CommissionedEmployee("Charlie", 40000, 0.1, 50000)

print("Employee Information:")
print_employee_info(salaried_emp)
print()
print_employee_info(hourly_emp)
print()
print_employee_info(commissioned_emp)
```

42. How does Python achieve polymorphism through method overriding?

Ans:

Python achieves polymorphism through method overriding by allowing subclasses to provide specific implementations of methods that are defined in their parent classes. This enables a single interface to be used for different underlying data types or classes.

Example:

```
class Animal:
    def make_sound(self):
        """Base method to be overridden by subclasses."""
        return "Some generic animal sound"
```

```
class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Polymorphic function that uses the base class type
def print_animal_sound(animal):
    print(animal.make_sound())

# Create instances of different subclasses
dog = Dog()
cat = Cat()

# Use the polymorphic function
print_animal_sound(dog)  # Output: Woof!
print_animal_sound(cat)  # Output: Meow!
```

Base Class (`Animal`):

- Contains the `make_sound` method with a generic implementation.

Subclasses (`Dog` and `Cat`):

- Override the `make_sound` method to provide specific sounds for dogs and cats.

Polymorphic Function (`print_animal_sound`):

- Accepts an `Animal` type, but can work with any subclass of `Animal`. It calls the overridden `make_sound` method of the actual subclass instance, demonstrating polymorphic behavior.

43. Define a base class with a method and a subclass that overrides the method.

Ans:

```
# Define the base class

class Vehicle:

    def start_engine(self):

        """Base method with a default implementation."""

        return "The engine is starting in a generic way."


# Define the subclass that overrides the base class method

class Car(Vehicle):

    def start_engine(self):

        """Override the base method with a specific implementation for cars."""

        return "The car engine is starting with a roar!"


# Define another subclass that overrides the base class method

class Motorcycle(Vehicle):

    def start_engine(self):

        """Override the base method with a specific implementation for motorcycles."""

        return "The motorcycle engine is starting with a vroom!"


# Example usage

def test_vehicle(vehicle):

    print(vehicle.start_engine())
```

Create instances of the subclasses

car = Car()

motorcycle = Motorcycle()

Use the polymorphic function

test_vehicle(car) # Output: The car engine is starting with a roar!

test_vehicle(motorcycle) # Output: The motorcycle engine is starting with a vroom!

Explanation

1. **Base Class (`Vehicle`)**:
   - Contains a method `start_engine` with a generic implementation. This method can be inherited by subclasses and is intended to be overridden.
2. **Subclass (`Car`)**:
   - Inherits from `Vehicle` and overrides the `start_engine` method to provide a specific implementation suitable for cars.
3. **Subclass (`Motorcycle`)**:
   - Also inherits from `Vehicle` and overrides the `start_engine` method to provide a specific implementation suitable for motorcycles.
4. **Polymorphic Function (`test_vehicle`)**:
   - Accepts any `Vehicle` type but works with instances of `Car` and `Motorcycle` due to method overriding. It demonstrates polymorphism by calling the appropriate overridden method based on the actual object type.

44. Define a base class and multiple subclasses with overridden methods.

Ans:

Base class

class Animal:

  def __init__(self, name):

    self.name = name

  def speak(self):

    raise NotImplementedError("Subclasses must implement this method")

```
    def __str__(self):
        return f"I am an animal named {self.name}"

# Subclass 1
class Dog(Animal):
    def speak(self):
        return "Woof!"

    def __str__(self):
        return f"I am a dog named {self.name}"

# Subclass 2
class Cat(Animal):
    def speak(self):
        return "Meow!"

    def __str__(self):
        return f"I am a cat named {self.name}"

# Subclass 3
class Cow(Animal):
    def speak(self):
        return "Moo!"

    def __str__(self):
        return f"I am a cow named {self.name}"
```

```
# Test the classes
def main():
    animals = [
        Dog("Rex"),
        Cat("Whiskers"),
        Cow("Bessie")
    ]

    for animal in animals:
        print(animal)  # This calls the __str__ method
        print(animal.speak())  # This calls the speak method

if __name__ == "__main__":
    main()
```

Explanation:

1. **Base Class `Animal`**:
    - `__init__`: Initializes the `name` attribute.
    - `speak`: A method that is expected to be overridden by subclasses. It raises a `NotImplementedError` to enforce this.
    - `__str__`: Provides a string representation of the animal. This can be overridden by subclasses for more specific details.
2. **Subclass `Dog`**:
    - Overrides the `speak` method to return "Woof!".
    - Overrides the `__str__` method to provide a specific string for dogs.
3. **Subclass `Cat`**:
    - Overrides the `speak` method to return "Meow!".
    - Overrides the `__str__` method to provide a specific string for cats.
4. **Subclass `Cow`**:
    - Overrides the `speak` method to return "Moo!".

    - Overrides the `__str__` method to provide a specific string for cows.
5. **Testing**:
    - In the `main` function, we create instances of `Dog`, `Cat`, and `Cow`.
    - We iterate over these instances, print their string representations (which uses the overridden `__str__` method), and print what they "say" (which uses the overridden `speak` method).

45. How does polymorphism improve code readability and reusability?

Ans:

Polymorphism enhances code readability and reusability by allowing methods to operate on objects of different classes in a consistent way. By using a common interface (e.g., a method name like `speak`), you can write more generic code that works with any subclass implementing that interface. This reduces the need for multiple conditionals or type checks, making the code cleaner and easier to understand. Additionally, it allows new classes to be added without modifying existing code, fostering extensibility and reducing maintenance efforts.

46. Describe how Python supports polymorphism with duck typing

Ans:

Python supports polymorphism through a concept known as *duck typing*. Duck typing is a style of dynamic typing where the type or class of an object is determined by its behavior (methods and properties) rather than its explicit inheritance or class definition.

Duck Typing Explained

In Python, duck typing is encapsulated by the adage: "If it looks like a duck and quacks like a duck, it probably is a duck." This means that the type of an object is determined by whether it has the required methods and properties, rather than whether it inherits from a specific class.

Example:

```
class Bird:
    def speak(self):
        return "Chirp!"

class Dog:
```

```
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

def make_animal_speak(animal):
    # The function doesn't care about the exact type of 'animal'.
    # It only requires that 'animal' has a 'speak' method.
    print(animal.speak())

# Create instances of different classes
bird = Bird()
dog = Dog()
cat = Cat()

# Pass different objects to the same function
make_animal_speak(bird)  # Output: Chirp!
make_animal_speak(dog)   # Output: Woof!
make_animal_speak(cat)   # Output: Meow!
```

47. How do you achieve encapsulation in Python?

Ans:

In Python, encapsulation is achieved by controlling access to an object's internal state and implementation details. While Python does not enforce strict access control like some other languages (e.g., private or protected keywords in Java), it provides mechanisms to suggest and guide the intended use of an object's attributes and methods. Encapsulation helps in

hiding the internal workings of an object and exposing only what is necessary, thereby promoting modularity and reducing the risk of unintended interference.

EXAMPLE:

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner        # Public attribute
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
        else:
            raise ValueError("Deposit amount must be positive")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
        else:
            raise ValueError("Invalid withdrawal amount")

    def get_balance(self):
        return self.__balance  # Public method accessing private attribute

    def __str__(self):
        return f"Account owner: {self.owner}, Balance: {self.__balance}"

# Example usage
```

```
account = BankAccount("Alice", 1000)

print(account.get_balance())  # Output: 1000


account.deposit(500)

print(account.get_balance())  # Output: 1500


account.withdraw(200)

print(account.get_balance())  # Output: 1300


# Accessing private attribute directly (not recommended)

# print(account.__balance)  # Raises AttributeError


# Accessing private method directly (not recommended)

# print(account.__private_method())  # Raises AttributeError


# Using public method to interact with private attributes

print(account)  # Output: Account owner: Alice, Balance: 1300
```

48. Can encapsulation be bypassed in Python? If so, how?

In Python, encapsulation can be bypassed due to the language's flexible and dynamic nature. While private and protected attributes and methods are suggested through naming conventions, they are not enforced strictly. Developers should use these conventions appropriately and design their classes in a way that encourages proper use and minimizes the need to bypass encapsulation.

EXAMPLE:-

```
class Example:

    def __init__(self):

        self.__private_method()
```

```
    def __private_method(self):
        print("This is a private method")


obj = Example()
obj._Example__private_method()  # Output: This is a private method
```

49. Implement a class BankAccount with a private balance attribute. Include methods to deposit, withdraw, and check the balance.

Ans:-

```
class BankAccount:
    def __init__(self, owner, initial_balance=0):
        self.owner = owner          # Public attribute
        self.__balance = initial_balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
        else:
            raise ValueError("Deposit amount must be positive")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
        else:
            raise ValueError("Invalid withdrawal amount")
```

```
    def get_balance(self):
        return self.__balance

    def __str__(self):
        return f"Account owner: {self.owner}, Balance: {self.__balance}"

# Example usage
if __name__ == "__main__":
    # Create a BankAccount instance
    account = BankAccount("Alice", 1000)

    # Print initial balance
    print(account.get_balance())  # Output: 1000

    # Deposit funds
    account.deposit(500)
    print(account.get_balance())  # Output: 1500

    # Withdraw funds
    account.withdraw(200)
    print(account.get_balance())  # Output: 1300

    # Print account details
    print(account)  # Output: Account owner: Alice, Balance: 1300

    # Attempt to access private attribute directly (will raise an error)
```

```
    # print(acc)
```

50. Develop a Person class with private attributes name and email, and methods to set and get the email

Ans:

```
class Person:
    def __init__(self, name, email):
        self.__name = name         # Private attribute
        self.__email = email       # Private attribute

    def set_email(self, new_email):
        if "@" in new_email and "." in new_email:
            self.__email = new_email
        else:
            raise ValueError("Invalid email address")

    def get_email(self):
        return self.__email

    def __str__(self):
        return f"Name: {self.__name}, Email: {self.__email}"

# Example usage
if __name__ == "__main__":
    # Create a Person instance
    person = Person("John Doe", "john.doe@example.com")
```

```
# Print initial email
print(person.get_email())  # Output: john.doe@example.com

# Set a new email
person.set_email("john.new@example.com")
print(person.get_email())  # Output: john.new@example.com

# Print person details
print(person)  # Output: Name: John Doe, Email: john.new@example.com

# Attempt to set an invalid email (will raise an error)
# person.set_email("invalid-email")  # Raises ValueError
```

51. Why is encapsulation considered a pillar of object-oriented programming (OOP)?

Ans:

1. Data Hiding

- **Purpose**: Encapsulation hides the internal state and implementation details of an object from the outside world. This means that the internal workings of an object are protected from direct access and modification.
- **Benefit**: By hiding the internal state, encapsulation ensures that the object's data can only be changed in controlled ways. This helps in maintaining the integrity of the data and prevents unintended or erroneous modifications.

2. Controlled Access

- **Purpose**: Encapsulation allows objects to control how their data is accessed and modified. Through methods (getters and setters), objects can enforce rules and constraints on how their attributes are used.
- **Benefit**: Controlled access helps enforce business rules and data validation. For instance, a method might ensure that an attribute can only be set to a valid value, thus preventing invalid or inconsistent states.

3. Modularity

- **Purpose**: Encapsulation groups data and methods that operate on the data into a single unit or class. This modular design means that an object's implementation is self-contained.
- **Benefit**: Modularity improves code organization and maintainability. Changes to the internal implementation of a class do not affect other parts of the program as long as the public interface remains unchanged. This makes it easier to update or refactor code.

4. Abstraction

- **Purpose**: Encapsulation is closely related to abstraction, which involves providing a simplified interface to complex underlying functionality. Encapsulation hides complex implementation details and exposes only the necessary parts of an object.
- **Benefit**: Abstraction helps users of a class focus on what the object does rather than how it does it. This makes the class easier to use and understand, as users interact with a well-defined interface.

5. Flexibility and Maintenance

- **Purpose**: Encapsulation provides flexibility by allowing the internal implementation of a class to change without affecting external code. As long as the public interface remains consistent, the internal implementation can be modified or optimized.
- **Benefit**: This flexibility leads to easier maintenance and evolution of code. Developers can make improvements or fix bugs within a class without having to alter other parts of the code that use the class.

6. Reusability

- **Purpose**: Encapsulation encourages the creation of reusable components by clearly defining and controlling their interfaces. Encapsulated classes can be reused in different parts of a program or in different programs with confidence.
- **Benefit**: Reusability reduces duplication of code and increases productivity. Encapsulated classes can be easily integrated into various contexts without needing to understand their internal details.

Summary

Encapsulation is a pillar of OOP because it:

- **Protects** the internal state of an object and ensures data integrity.
- **Provides controlled access** to an object's attributes and methods.
- **Supports modularity**, making code easier to manage and maintain.
- **Facilitates abstraction**, allowing users to interact with objects through simplified interfaces.
- **Enables flexibility**, allowing internal changes without affecting external code.
- **Promotes reusability** by creating well-defined and manageable components.

52. Create a decorator in Python that adds functionality to a simple function by printing a message before and after the function execution

Ans:

Creating the Decorator

1. **Define the Decorator Function**: The decorator function will take a function as an argument and return a new function that adds additional behavior.
2. **Wrap the Original Function**: Inside the decorator, define an inner function that wraps the original function. This inner function will handle the additional behavior (printing messages) before and after calling the original function.
3. **Return the Wrapper Function**: The decorator function should return the inner wrapper function.

Example:

```
def print_messages_decorator(func):

    def wrapper(*args, **kwargs):

        print("Before function execution")

        result = func(*args, **kwargs)

        print("After function execution")

        return result

    return wrapper


# Example usage of the decorator


@print_messages_decorator

def simple_function(message):

    print(f"Function execution with message: {message}")


# Call the decorated function

simple_function("Hello, world!")
```

53. Modify the decorator to accept arguments and print the function name along with the message.

Ans:

```
def print_messages_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before function execution")
        result = func(*args, **kwargs)
        print("After function execution")
        return result
    return wrapper


# Example usage of the decorator


@print_messages_decorator
def simple_function(message):
    print(f"Function execution with message: {message}")


# Call the decorated function
simple_function("Hello, world!")
```

54. Create two decorators, and apply them to a single function. Ensure that they execute in the order they are applied

Ans:

```
def before_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before executing function")
        return func(*args, **kwargs)
    return wrapper
```

```
def after_decorator(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print("After executing function")
        return result
    return wrapper

# Applying decorators in order
@before_decorator
@after_decorator
def my_function(message):
    print(f"Function execution with message: {message}")

# Call the decorated function
my_function("Hello, world!")
```

55. Modify the decorator to accept and pass function arguments to the wrapped function.

Ans:

```
def before_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before executing function")
        return func(*args, **kwargs)
    return wrapper

def after_decorator(func):
    def wrapper(*args, **kwargs):
```

```
        result = func(*args, **kwargs)
        print("After executing function")
        return result
    return wrapper

# Apply decorators in order
@before_decorator
@after_decorator
def my_function(message, count):
    for _ in range(count):
        print(f"Function execution with message: {message}")

# Call the decorated function with arguments
my_function("Hello, world!", 3)
```

56. Create a decorator that preserves the metadata of the original function

Ans:

```
import functools

def preserve_metadata_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print("Executing the decorated function")
        return func(*args, **kwargs)
    return wrapper
```

```
@preserve_metadata_decorator
def example_function(param1, param2):
    """
    This is an example function.

    Args:
        param1: The first parameter.
        param2: The second parameter.

    Returns:
        A string combining both parameters.
    """
    return f"Parameters received: {param1} and {param2}"

# Call the decorated function
result = example_function("Hello", "World")
print(result)

# Accessing metadata
print(f"Function name: {example_function.__name__}")
print(f"Function docstring: {example_function.__doc__}")
```

57. Create a Python class `Calculator with a static method `add` that takes in two numbers and returns their sum.

Ans:

```
class Calculator:
    @staticmethod
```

```
    def add(a, b):
        """
        Return the sum of two numbers.

        Args:
            a (int or float): The first number.
            b (int or float): The second number.

        Returns:
            int or float: The sum of the two numbers.
        """
        return a + b

# Example usage
result = Calculator.add(5, 3)
print(result)  # Output: 8
```

58. Create a Python class `Employee` with a class `method get_employee_count` that returns the total number of employees created.

Ans:

```
class Employee:
    # Class variable to keep track of the number of employees
    _employee_count = 0

    def __init__(self, name, position):
        self.name = name
        self.position = position
```

```
        # Increment the employee count each time a new instance is created
        Employee._employee_count += 1

    @classmethod
    def get_employee_count(cls):
        """
        Return the total number of employees created.

        Returns:
            int: The total number of employee instances.
        """
        return cls._employee_count

# Example usage
emp1 = Employee("Alice", "Engineer")
emp2 = Employee("Bob", "Manager")
emp3 = Employee("Charlie", "Technician")

print(Employee.get_employee_count())  # Output: 3
```

59. Create a Python class `StringFormatter` with a static method `reverse_string` that takes a string as input and returns its reverse.

Ans:

```
class StringFormatter:
    @staticmethod
    def reverse_string(s):
        """
```

```
        Reverse the input string and return it.

        Args:
            s (str): The string to be reversed.

        Returns:
            str: The reversed string.
        """
        return s[::-1]

# Example usage
original_string = "hello"
reversed_string = StringFormatter.reverse_string(original_string)
print(reversed_string)  # Output: "olleh"
```

60. Create a Python class `Circle` with a class method `calculate_area` that calculates the area of a circle Type

Ans:

```
import math

class Circle:
    @classmethod
    def calculate_area(cls, radius):
        """
        Calculate the area of a circle given its radius.

        Args:
```

```
        radius (float): The radius of the circle.

    Returns:
        float: The area of the circle.
    """
    if radius < 0:
        raise ValueError("Radius cannot be negative")
    return math.pi * (radius ** 2)

# Example usage
radius = 5
area = Circle.calculate_area(radius)
print(f"The area of the circle with radius {radius} is {area:.2f}")
```

61. Create a Python class `TemperatureConverter` with a static method `celsius_to_fahrenheit` that converts Celsius to Fahrenheit.

Ans:

```
class TemperatureConverter:
    @staticmethod
    def celsius_to_fahrenheit(celsius):
        """
        Convert Celsius to Fahrenheit.

        Args:
            celsius (float): The temperature in Celsius.

        Returns:
```

```
            float: The temperature in Fahrenheit.
        """
        return (celsius * 9/5) + 32

# Example usage
celsius_temp = 25
fahrenheit_temp = TemperatureConverter.celsius_to_fahrenheit(celsius_temp)
print(f"{celsius_temp}°C is equal to {fahrenheit_temp}°F")
```

62. What is the purpose of the __str__() method in Python classes? Provide an example.

Ans:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        """
        Return a human-readable string representation of the Person object.

        Returns:
            str: A string describing the Person.
        """
        return f"Person(Name: {self.name}, Age: {self.age})"

# Example usage
```

```
person = Person("Alice", 30)

print(person)
```

63. How does the ____len___() method work in Python? Provide an example.

Ans:

```
class CustomList:
    def __init__(self, *elements):
        self.elements = list(elements)


    def __len__(self):
        """
        Return the number of elements in the CustomList.


        Returns:
            int: The number of elements in the list.
        """
        return len(self.elements)


# Example usage
my_list = CustomList(1, 2, 3, 4, 5)
print(len(my_list))  # Output: 5
```

64. Explain the usage of the ____add__() method in Python classes. Provide an example.

Ans:

the __add__() method in Python is a special (or "dunder") method used to define the behavior of the addition operator (+) for instances of a class. When you use the + operator with objects of your class, Python internally calls the `__add__()` method to perform the addition.

Purpose of `__add__()`

- **Custom Addition Behavior**: Allows you to define how objects of your class should be added together. This is useful for custom types where the addition operation has a specific meaning.

How `__add__()` Works

- **Definition**: Implement the `__add__()` method to specify the behavior of the `+` operator for your class. It should take two parameters: `self` (the instance on the left of the `+` operator) and `other` (the instance or value on the right).
- **Usage**: Python calls `__add__()` when you use the `+` operator with instances of your class.

Example Code

```
class Vector:

    def __init__(self, x, y):

        self.x = x

        self.y = y


    def __add__(self, other):

        """

        Define the addition operation for Vector objects.


        Args:

            other (Vector): Another Vector object to add.


        Returns:

            Vector: A new Vector instance representing the sum.

        """

        if not isinstance(other, Vector):

            return NotImplemented
```

```
        return Vector(self.x + other.x, self.y + other.y)


    def __repr__(self):
        """
        Provide a string representation of the Vector object.

        Returns:
            str: A string representing the Vector.
        """
        return f"Vector({self.x}, {self.y})"


# Example usage
v1 = Vector(1, 2)
v2 = Vector(3, 4)
v3 = v1 + v2
print(v3)  # Output: Vector(4, 6)
```

65. What is the purpose of the __getitem___() method in Python? Provide an example

Ans:

The `__getitem__()` method in Python is a special (or "dunder") method that allows you to define how objects of a class handle indexing and key access, similar to how dictionaries and lists work. This method is called when you use square brackets to access an item from an instance.

Example Code:-

```
class CustomList:
    def __init__(self, *elements):
```

```
        self.elements = list(elements)

    def __getitem__(self, index):
        """
        Retrieve an element from the CustomList at the specified index.

        Args:
            index (int): The index of the element to retrieve.

        Returns:
            The element at the specified index.
        """
        if index < 0 or index >= len(self.elements):
            raise IndexError("Index out of range")
        return self.elements[index]

    def __repr__(self):
        return f"CustomList({self.elements})"

# Example usage
my_list = CustomList(10, 20, 30, 40, 50)
print(my_list[2])  # Output: 30

# Attempting to access an out-of-range index
```

```
try:
    print(my_list[10])
except IndexError as e:
    print(e)  # Output: Index out of range.))
```

66. Explain the usage of the ____iter___() and ____next__() methods in Python. Provide an example using iterators.

Ans:

In Python, the __iter__() and __next__() methods are fundamental for creating iterators. They enable objects to be iterable and work with constructs like loops, list comprehensions, and the next() function. Here’s a breakdown of these methods and how they work together to implement an iterator

Example:

```
class Countdown:
    def __init__(self, start):
        self.start = start
        self.current = start

    def __iter__(self):
        # The iterator is the object itself
        return self

    def __next__(self):
        if self.current <= 0:
            raise StopIteration
```

```
        self.current -= 1
        return self.current + 1

# Example usage
countdown = Countdown(5)

for number in countdown:
    print(number)
```

67. What is the purpose of a getter method in Python? Provide an example demonstrating the use of a getter method using property decorators

Ans:

In Python, a getter method is used to access the value of a private attribute of a class. The primary purpose of a getter is to provide controlled access to private or protected attributes while allowing you to implement additional logic if needed. Using the @property decorator, you can create getter methods in a more Pythonic way, making the attribute access look like a regular attribute access, while still encapsulating the underlying logic

Example:-

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        """
```

```
        Getter method for the radius attribute.

        Returns:
            float: The radius of the circle.
        """
        return self._radius

    @property
    def diameter(self):
        """
        Calculate the diameter based on the radius.

        Returns:
            float: The diameter of the circle.
        """
        return self._radius * 2

    @property
    def area(self):
        """
        Calculate the area of the circle based on the radius.

        Returns:
            float: The area of the circle.
```

```
        """
        import math
        return math.pi * (self._radius ** 2)

# Example usage
circle = Circle(5)
print(f"Radius: {circle.radius}")  # Output: Radius: 5
print(f"Diameter: {circle.diameter}")  # Output: Diameter: 10
print(f"Area: {circle.area:.2f}")  # Output: Area: 78.54
```

68. Explain the role of setter methods in Python. Demonstrate how to use a setter method to modify a class attribute using property decorators.

Ans:

Setter methods in Python are used to define how an attribute's value should be modified. They allow you to control the assignment of values to private attributes, enabling validation or transformation of data before actually storing it. By using the `@property` decorator along with the `@<propertyname>.setter` decorator, you can create a getter and setter for an attribute, providing controlled access to it

Example:

```
class Person:
    def __init__(self, name, age):
        self._name = name
        self._age = age

    @property
    def name(self):
        """
```

```
        Getter method for the name attribute.

        Returns:
            str: The name of the person.
        """
        return self._name

    @name.setter
    def name(self, value):
        """
        Setter method for the name attribute.

        Args:
            value (str): The new name to set.
        """
        if not value or not value.strip():
            raise ValueError("Name cannot be empty.")
        self._name = value

    @property
    def age(self):
        """
        Getter method for the age attribute.

        Returns:
            int: The age of the person.
```

```
        """
        return self._age

    @age.setter
    def age(self, value):
        """
        Setter method for the age attribute.

        Args:
            value (int): The new age to set.
        """
        if value < 0:
            raise ValueError("Age cannot be negative.")
        self._age = value

    def __repr__(self):
        return f"Person(Name: {self._name}, Age: {self._age})"

# Example usage
person = Person("Alice", 30)
print(person)  # Output: Person(Name: Alice, Age: 30)

# Modify attributes using setter methods
person.name = "Bob"
person.age = 35
print(person)  # Output: Person(Name: Bob, Age: 35)
```

Attempting to set invalid values

try:

    person.name = ""

except ValueError as e:

    print(e) # Output: Name cannot be empty.

try:

    person.age = -5

except ValueError as e:

    print(e) # Output: Age cannot be negative.

69. What is the purpose of the @property decorator in Python? Provide an example illustrating its usage.

Ans:

the `@property` decorator in Python is used to define a method that can be accessed like an attribute. It allows you to create managed attributes in a class, providing a way to access computed values or encapsulate logic behind attribute access, while maintaining a clean and intuitive syntax. This decorator is particularly useful when you want to expose methods that behave like attributes but also want to include additional logic or validation.

Example:

```
class Rectangle:

    def __init__(self, width, height):

        self._width = width

        self._height = height


    @property
```

```
def width(self):
    """
    Getter method for the width attribute.

    Returns:
        float: The width of the rectangle.
    """
    return self._width

@width.setter
def width(self, value):
    """
    Setter method for the width attribute.

    Args:
        value (float): The new width to set.
    """
    if value <= 0:
        raise ValueError("Width must be positive.")
    self._width = value

@property
def height(self):
    """
```

```
    Getter method for the height attribute.

    Returns:
        float: The height of the rectangle.
    """
    return self._height

@height.setter
def height(self, value):
    """
    Setter method for the height attribute.

    Args:
        value (float): The new height to set.
    """
    if value <= 0:
        raise ValueError("Height must be positive.")
    self._height = value

@property
def area(self):
    """
    Calculate the area of the rectangle.
```

```
        Returns:
            float: The area of the rectangle.
        """
        return self._width * self._height

    @property
    def perimeter(self):
        """
        Calculate the perimeter of the rectangle.

        Returns:
            float: The perimeter of the rectangle.
        """
        return 2 * (self._width + self._height)

    def __repr__(self):
        return f"Rectangle(width={self._width},
height={self._height})"

# Example usage
rect = Rectangle(5, 3)
print(rect.width)       # Output: 5
print(rect.height)      # Output: 3
print(rect.area)        # Output: 15
print(rect.perimeter)   # Output: 16
```

```
# Modify dimensions using setters
rect.width = 7
rect.height = 4
print(rect.area)        # Output: 28
print(rect.perimeter)   # Output: 22

# Attempting to set invalid dimensions
try:
    rect.width = -1
except ValueError as e:
    print(e)  # Output: Width must be positive.

try:
    rect.height = 0
except ValueError as e:
    print(e)  # Output: Height must be positive.
```

70. Explain the use of the @deleter decorator in Python property decorators. Provide a code example demonstrating its application.

Ans:-

The `@deleter` decorator in Python is used in conjunction with the `@property` decorator to define a method that deletes an attribute from an instance. This allows you to provide custom behavior when an attribute is deleted using the `del` statement. The `@deleter` decorator is part of the property management system, enabling you to control

attribute deletion and implement cleanup or other logic when an attribute is removed.

Purpose of the `@deleter` Decorator

- **Custom Deletion Logic**: Provides a way to specify what should happen when an attribute is deleted, allowing for cleanup or validation.
- **Controlled Attribute Management**: Ensures that attribute deletion can be managed or restricted according to specific rules or conditions.

Example:

```
class Person:

    def __init__(self, name):

        self._name = name



    @property

    def name(self):

        """

        Getter method for the name attribute.



        Returns:

            str: The name of the person.

        """

        return self._name



    @name.setter

    def name(self, value):

        """
```

```
        Setter method for the name attribute.

        Args:
            value (str): The new name to set.
        """
        if not value or not value.strip():
            raise ValueError("Name cannot be empty.")
        self._name = value

    @name.deleter
    def name(self):
        """
        Deleter method for the name attribute.

        Raises:
            AttributeError: If the name attribute is not set.
        """
        if self._name is None:
            raise AttributeError("Cannot delete an unset name
attribute.")
        print(f"Deleting name: {self._name}")
        del self._name

    def __repr__(self):
        return f"Person(name={self._name})"
```

```
# Example usage

person = Person("Alice")

print(person.name)  # Output: Alice


# Modify the name

person.name = "Bob"

print(person.name)  # Output: Bob


# Delete the name

del person.name  # Output: Deleting name: Bob


# Attempting to access the deleted name

try:

    print(person.name)

except AttributeError as e:

    print(e)  # Output: 'Person' object has no attribute '_name'
```

71. How does encapsulation relate to property decorators in Python? Provide an example showcasing encapsulation using property decorators.

Ans:

Encapsulation in Python refers to the practice of bundling the data (attributes) and methods (functions) that operate on the data into a single unit or class, and restricting direct access to some of the object's components. This is often done to protect the internal state of an object and to prevent unintended interference.

Property decorators (@property, @<propertyname>.setter, and @<propertyname>.deleter) are a key tool for implementing encapsulation in Python. They allow you to control access to an attribute by defining methods that manage how the attribute is read, written, or deleted. This provides a way to enforce validation, maintain invariants, or compute values dynamically while still using a clean and intuitive attribute-like syntax.

Example:

```
class BankAccount:

    def __init__(self, owner, balance):

        self.owner = owner

        self._balance = balance  # Private attribute


    @property

    def balance(self):

        """

        Getter method for the balance attribute.


        Returns:

            float: The current balance of the bank account.

        """

        return self._balance


    @balance.setter

    def balance(self, amount):

        """
```

```
        Setter method for the balance attribute. Ensures that the
balance
        cannot be set to a negative value.

        Args:
            amount (float): The new balance to set.

        Raises:
            ValueError: If the amount is negative.
        """
        if amount < 0:
            raise ValueError("Balance cannot be negative.")
        self._balance = amount

    def deposit(self, amount):
        """
        Deposit money into the bank account.

        Args:
            amount (float): The amount to deposit.

        Raises:
            ValueError: If the deposit amount is negative.
        """
        if amount <= 0:
```

```
            raise ValueError("Deposit amount must be positive.")
        self._balance += amount

    def withdraw(self, amount):
        """
        Withdraw money from the bank account.

        Args:
            amount (float): The amount to withdraw.

        Raises:
            ValueError: If the withdrawal amount is negative or
exceeds the balance.
        """
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive.")
        if amount > self._balance:
            raise ValueError("Insufficient funds.")
        self._balance -= amount

    def __repr__(self):
        return f"BankAccount(owner={self.owner},
balance={self._balance:.2f})"

# Example usage
```

```
account = BankAccount("Alice", 1000)
print(account.balance)  # Output: 1000

# Deposit and withdraw using methods
account.deposit(500)
print(account.balance)  # Output: 1500

account.withdraw(200)
print(account.balance)  # Output: 1300

# Attempting to set balance directly (not allowed)
try:
    account.balance = -500
except ValueError as e:
    print(e)  # Output: Balance cannot be negative.

# Attempting to withdraw more than available balance
try:
    account.withdraw(2000)
except ValueError as e:
    print(e)  # Output: Insufficient funds.
```