Assignment- Object Oriented Programming

1) Explain the importance of function

   Ans: Functions are essential in programming because they allow for code reuse, modularity, and organization. They help break down complex problems into smaller, manageable pieces, making code easier to understand, test, and maintain. Functions also allow for the abstraction of functionality, meaning you can define a process once and use it repeatedly throughout your program.

2) Write a basic function to greet a student

   Ans :

```
def greet_student(name):

    print(f"Hello, {name}! Welcome to the class.")

greet_student("Alice")
```

3) What is the difference between print and return statements ?

   Ans:

   `print` Statement: Outputs a message to the console. It doesn't affect the flow of the program or return any value. It's mainly used for debugging or providing user feedback.
   Example:

```
def say_hello():

    print("Hello!")

say_hello()  # This prints "Hello!" to the console.
```

   `return` Statement: Ends the function execution and optionally returns a value to the caller. The returned value can be used in further computations or stored in a variable.
   Example:

```
def add(a, b):

    return a + b

result = add(2, 3)  # This returns the value 5, which can be stored
or used
```

4) What are *args and **kwargs ?

Ans

`*args`: Allows a function to accept any number of positional arguments. Inside the function, `*args` is treated as a tuple.
Example:

```
def add(*args):

    return sum(args)

print(add(1, 2, 3, 4))  # Outputs: 10
```

`**kwargs`: Allows a function to accept any number of keyword arguments. Inside the function, `**kwargs` is treated as a dictionary.
Example:

```
def print_info(**kwargs):

    for key, value in kwargs.items():

        print(f"{key}: {value}")

print_info(name="Alice", age=25)  # Outputs: name: Alice, age: 25
```

5) Explain the iterator function

Ans

An iterator is an object that allows a programmer to traverse through all the elements of a collection (like a list or a tuple) without needing to know the underlying structure. It implements two methods: `__iter__()` and `__next__()`.

Example:

```
class MyIterator:

    def __init__(self, data):

        self.data = data

        self.index = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.index < len(self.data):
            result = self.data[self.index]
            self.index += 1
            return result
        else:
            raise StopIteration
iterator = MyIterator([1, 2, 3])
for item in iterator:
    print(item)
```

6) Writes a code that generates the squares of numbers from 1 to n using a generator

Ans

```
def square_generator(n):
    for i in range(1, n+1):
        yield i * i
for square in square_generator(5):
    print(square)
```

7) Writes a code that generates palindromic numbers up to n using a generator

Ans:

```
def palindromic_generator(n):
    for i in range(1, n+1):
        if str(i) == str(i)[::-1]:
            yield i

# Example usage
for palindrome in palindromic_generator(100):
    print(palindrome)
```

8) Writes a code that generates even numbers from 2 to n using a generator

Ans

```
def even_number_generator(n):
    for i in range(2, n+1, 2):
        yield i

# Example usage
for even_number in even_number_generator(10):
    print(even_number)
```

9) Writes a code that generates power of two up to n using generator

Ans

```
def power_of_two_generator(n):
    i = 1

    while i <= n:

        yield i

        i *= 2


# Example usage

for power in power_of_two_generator(32):

    print(power)
```

10) Write a code that generates prime numbers up to n using a generator.

Ans

```
def is_prime(num):

    if num < 2:

        return False

    for i in range(2, int(num**0.5) + 1):

        if num % i == 0:

            return False

    return True


def prime_generator(n):

    for i in range(2, n + 1):

        if is_prime(i):

            yield i


# Example usage
```

```
for prime in prime_generator(50):
    print(prime)
```

11) Write a code that uses a lambda function to calculate the sum of two numbers.

Ans

```
sum_numbers = lambda a, b: a + b
print(sum_numbers(3, 5))  # Outputs: 8
```

12) Write a code that uses a lambda function to calculate the square of a given number.

Ans

```
square = lambda x: x * x
print(square(4))  # Outputs: 16
```

13) Write a code that uses a lambda function to check whether a given number is even or odd.

Ans:

```
is_even = lambda x: "Even" if x % 2 == 0 else "Odd"
print(is_even(5))  # Outputs: Odd
```

14) No question

15) Write a code that uses a lambda function to concatenate two strings.

Ans:

```
concat_strings = lambda s1, s2: s1 + s2

# Example usage
print(concat_strings("Hello, ", "World!"))  # Outputs: Hello, World!
```

16. Write a code that uses a lambda function to find the maximum of three given numbers.

```
max_lambda = lambda x, y, z: max(x, y, z)

print(max_lambda(3, 7, 5))  # Output: 7
```

---

17. Write a code that generates the squares of even numbers from a given list.

```
numbers = [1, 2, 3, 4, 5, 6]

even_squares = [x ** 2 for x in numbers if x % 2 == 0]

print(even_squares)  # Output: [4, 16, 36]
```

---

18. Write a code that calculates the product of positive numbers from a given list.

```
from functools import reduce


numbers = [-1, 2, 3, -4, 5]

product = reduce(lambda x, y: x * y, (num for num in numbers if num
> 0))

print(product)  # Output: 30
```

---

19. Write a code that doubles the values of odd numbers from a given list.

```
numbers = [1, 2, 3, 4, 5]

doubled_odds = [x * 2 for x in numbers if x % 2 != 0]

print(doubled_odds)  # Output: [2, 6, 10]
```

---

20. Write a code that calculates the sum of cubes of numbers from a given list.

```
numbers = [1, 2, 3, 4]

sum_of_cubes = sum(x ** 3 for x in numbers)

print(sum_of_cubes)  # Output: 100
```

---

21. Write a code that filters out prime numbers from a given list.

```
def is_prime(num):

    if num < 2:

        return False

    return all(num % i != 0 for i in range(2, int(num**0.5) + 1))


numbers = [2, 4, 5, 9, 11, 15]

primes = list(filter(is_prime, numbers))

print(primes)  # Output: [2, 5, 11]
```

22. Write a code that uses a lambda function to calculate the sum of two numbers.

```
sum_lambda = lambda x, y: x + y

print(sum_lambda(3, 7))  # Output: 10
```

---

23. Write a code that uses a lambda function to calculate the square of a given number.

```
square_lambda = lambda x: x ** 2

print(square_lambda(5))  # Output: 25
```

---

24. Write a code that uses a lambda function to check whether a given number is even or odd.

```
even_odd_lambda = lambda x: "Even" if x % 2 == 0 else "Odd"

print(even_odd_lambda(8))  # Output: Even

print(even_odd_lambda(7))  # Output: Odd
```

---

25. Write a code that uses a lambda function to concatenate two strings.

```
concat_lambda = lambda a, b: a + b

print(concat_lambda("Hello, ", "World!"))  # Output: Hello, World!
```

---

26. Write a code that uses a lambda function to find the maximum of three given numbers.

```
max_lambda = lambda x, y, z: max(x, y, z)

print(max_lambda(4, 9, 2))  # Output: 9
```

---

27. What is encapsulation in OOP?

Answer:
Encapsulation is a principle in **Object-Oriented Programming (OOP)** that restricts direct access to the internal data of a class and only allows controlled access through methods (getters and setters). This is done by using access modifiers like **private** and **protected**.

Example:

```
class Student:

    def __init__(self, name, grade):

        self.name = name          # Public attribute

        self.__grade = grade      # Private attribute


    def get_grade(self):          # Getter method

        return self.__grade


    def set_grade(self, grade):  # Setter method

        if 0 <= grade <= 100:

            self.__grade = grade

        else:
            print("Invalid grade.")


student = Student("Alice", 85)

print(student.get_grade())  # Output: 85

student.set_grade(95)

print(student.get_grade())  # Output: 95
```

---

28. Explain the use of access modifiers in Python classes.

Answer:
Access modifiers in Python control the visibility of class attributes and methods:

- **Public**: Accessible from anywhere (default in Python).
- **Protected**: Prefix with a single underscore `_`. Suggests internal use.
- **Private**: Prefix with double underscores `__`. Restricts access to within the class.

Example:

```
class Demo:

    public_var = "Public"

    _protected_var = "Protected"

    __private_var = "Private"


    def access_private(self):

        return self.__private_var


obj = Demo()
```

```
print(obj.public_var)       # Output: Public

print(obj._protected_var)   # Output: Protected

# print(obj.__private_var)  # AttributeError

print(obj.access_private()) # Output: Private
```

---

29. What is inheritance in OOP?

Answer:
Inheritance is an OOP feature that allows a class (child) to inherit properties and methods from another class (parent). It enables code reuse and hierarchy.

Example:

```
class Animal:  # Parent class

    def sound(self):

        print("Animals make sounds.")


class Dog(Animal):  # Child class

    def sound(self):

        print("Dog barks.")


dog = Dog()

dog.sound()  # Output: Dog barks.
```

---

30. Define polymorphism in OOP.

Answer:
Polymorphism allows different classes to define methods with the same name, enabling a unified interface. It includes method overriding and overloading.

Example:

```
class Animal:

    def sound(self):

        print("Some generic sound.")



class Dog(Animal):

    def sound(self):

        print("Bark!")



class Cat(Animal):

    def sound(self):

        print("Meow!")



animals = [Dog(), Cat()]

for animal in animals:

    animal.sound()

# Output: Bark!

#         Meow!
```

---

31. Explain method overriding in Python.

Answer:
Method overriding occurs when a child class provides a new implementation for a method already defined in its parent class.

Example:

```
class Parent:

    def display(self):

        print("This is Parent class.")


class Child(Parent):

    def display(self):

        print("This is Child class.")


obj = Child()

obj.display()  # Output: This is Child class.
```

---

32. 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!".

```
class Animal:

    def make_sound(self):

        print("Generic animal sound")


class Dog(Animal):
    def make_sound(self):

        print("Woof!")


dog = Dog()

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

---

33. Define a method `move` in the Animal class that prints "Animal moves". Override the method in the Dog class to print "Dog runs".

```
class Animal:

    def move(self):

        print("Animal moves.")


class Dog(Animal):

    def move(self):

        print("Dog runs.")


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 DogMammal inheriting from both Dog and Mammal.

```
class Mammal:

    def reproduce(self):

        print("Giving birth to live young.")


class DogMammal(Dog, Mammal):

    pass


dog_mammal = DogMammal()

dog_mammal.make_sound()  # Output: Woof!

dog_mammal.reproduce()   # Output: Giving birth to live young.
```

---

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

```
class GermanShepherd(Dog):

    def make_sound(self):

        print("Bark!")


gs = GermanShepherd()

gs.make_sound()  # Output: Bark!
```

---

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

```
class Animal:

    def __init__(self, species):

        self.species = species


class Dog(Animal):

    def __init__(self, species, breed):

        super().__init__(species)

        self.breed = breed


dog = Dog("Mammal", "Labrador")

print(f"Species: {dog.species}, Breed: {dog.breed}")

# Output: Species: Mammal, Breed: Labrador
```

---

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

Answer:
Abstraction is the process of hiding implementation details while exposing functionality. It is implemented using abstract classes and methods with the `abc` module.

Example:

```
from abc import ABC, abstractmethod


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


class Dog(Animal):
    def sound(self):
        print("Woof!")


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

---

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

Answer:
Abstraction helps in:

- Hiding complex implementation details.
- Promoting modularity and separation of concerns.
- Providing a clear interface for users.

---

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

Answer:

- **Abstract methods** are declared in an abstract class and do not have an implementation. Subclasses must override them.
- **Regular methods** have a concrete implementation and can be directly used.

Example:

```python
from abc import ABC, abstractmethod


class AbstractClass(ABC):

    @abstractmethod

    def abstract_method(self):

        pass


class ConcreteClass(AbstractClass):

    def abstract_method(self):

        print("Implemented abstract method.")


obj = ConcreteClass()

obj.abstract_method()  # Output: Implemented abstract method.
```

Question 40:

How can you achieve abstraction using interfaces in Python?

Answer:

Python does not have explicit interfaces like Java or C++. However, we can achieve abstraction using abstract base classes (ABCs) provided by the `abc` module.

Here's how you can use ABCs to define an abstract interface:

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```

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

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

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

    def area(self):
        return 3.14159 * self.radius * self.radius
```

In this example:

- `Shape` is an abstract base class with an abstract method `area`.
- `Rectangle` and `Circle` are concrete subclasses that implement the `area` method.
-
- This creates a common interface for `Rectangle` and `Circle` objects, allowing them to be treated as `Shape` objects where the `area` method is needed.

Question 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?

Answer:

See the example in Question 40.

Question 42:

How does Python achieve polymorphism through method overriding?

Answer:

Python achieves polymorphism through method overriding and duck typing.

Method Overriding:

- Subclasses can override methods inherited from their parent classes.

- When a method is called on an object, the appropriate implementation based on the object's type is invoked.
-

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

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

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

Question 43:

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

Answer:

In Python, a subclass can override a method from the base class to provide a more specific or different implementation. Here's an example:

```
# Base class

class Vehicle:

    def description(self):

        print("This is a vehicle.")



# Subclass that overrides the method

class Car(Vehicle):

    def description(self):

        print("This is a car, a type of vehicle.")
```

```
# Creating objects

vehicle = Vehicle()

car = Car()


# Calling the method

vehicle.description()  # Output: This is a vehicle.

car.description()      # Output: This is a car, a type of vehicle.
```

Explanation:

1. **Base Class (`Vehicle`):**
   - **It has a method `description()` that prints a general description.**
2. **Subclass (`Car`):**
   - **It inherits from the `Vehicle` class.**
   - **It overrides the `description()` method to provide a more specific message.**
3. **Output:**
   - **When calling `description()` on an object of the `Vehicle` class, the base class method runs.**
   - **When calling `description()` on an object of the `Car` class, the overridden method runs.**

Question 44:

Define a base class and multiple subclasses with overridden methods.

Answer:

In Python, you can define a base class with a method and have multiple subclasses that override the method with their specific implementations. Here's an example:

```
# Base class
```

```
class Animal:

    def sound(self):

        print("Animals make sounds.")


# Subclass 1: Dog

class Dog(Animal):

    def sound(self):

        print("Dog barks.")


# Subclass 2: Cat

class Cat(Animal):

    def sound(self):

        print("Cat meows.")


# Subclass 3: Cow

class Cow(Animal):

    def sound(self):

        print("Cow moos.")


# Creating objects of each class

animal = Animal()

dog = Dog()

cat = Cat()
```

```
cow = Cow()


# Calling the overridden methods

animal.sound()  # Output: Animals make sounds.

dog.sound()     # Output: Dog barks.

cat.sound()     # Output: Cat meows.

cow.sound()     # Output: Cow moos.
```

Explanation:

1. **Base Class (`Animal`):**
   - **It contains a general `sound()` method.**
2. **Subclasses (`Dog`, `Cat`, and `Cow`):**
   - **Each subclass inherits from the `Animal` base class.**
   - **Each subclass overrides the `sound()` method to provide its specific implementation.**
3. **Output:**
   - **Each subclass displays a different message when the `sound()` method is called.**

Question 45:

How does polymorphism improve code readability and reusability?

Answer:

Polymorphism improves code readability and reusability by:

- **Promoting code flexibility:** Polymorphic code can work with different object types, making it easier to write generic functions and algorithms.
- **Simplifying code structure:** Polymorphic code can avoid complex conditional logic to handle different object types.

- **Encouraging code extensibility:** New subclasses can be added without modifying existing code, as long as they implement the necessary methods.

Question 46:

Describe how Python supports polymorphism with duck typing.

Answer:

Duck Typing:

- In Python, objects don't need to explicitly inherit from a common base class to be used polymorphically.
- As long as an object has the necessary methods, it can be used where that interface is expected.

```
Python
def make_sound(animal):
    animal.make_sound()  # Calls the make_sound() method of the object

make_sound(Dog())  # Works because Dog has a make_sound() method
make_sound(Cat())  # Works because Cat also has a make_sound() method
```

Question 47:

How do you achieve encapsulation in Python?

Answer:

Encapsulation in Python is achieved by using private attributes and methods:

- **Private attributes:** Attributes prefixed with `__` are considered private and are not directly accessible from outside the class.
- **Private methods:** Methods prefixed with `__` are also private and cannot be called directly from outside the class.

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

    def get_name(self):
        return self.__name
    def get_age(self):
        return self.__age
```

Question 48:

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

Answer:

Yes, encapsulation can be bypassed in Python using `_classname__attributename` syntax.

```
person = Person("Alice", 30)
print(person._Person__name)  # Accesses the private attribute
```

However, this is generally discouraged as it violates the principle of encapsulation.

Question 49:

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

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

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if amount > self.__balance:
            print("Insufficient funds")
        else:
            self.__balance -= amount

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

Question 50:

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

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

    def set_email(self, email):
        self.__email = email

    def get_email(self):
        return self.__email
```

Question 51:

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

Encapsulation is a pillar of OOP because it:

- **Protects data integrity:** By hiding implementation details, encapsulation prevents accidental modification of internal state.
- **Improves code maintainability:** Changes to the internal implementation can be made without affecting the public interface.
- **Enhances code reusability:** Encapsulated classes can be reused in different contexts without worrying about their internal workings.
- **Promotes modularity:** Encapsulation helps break down complex systems into smaller, self-contained modules.

Question 52:

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

Answer:

```
def my_decorator(func):

    def wrapper(*args, **kwargs):

        print("Before function execution")

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

@my_decorator
def my_function():
    print("Inside my_function")

my_function()
```

Question 53:

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

Answer:

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Before executing {func.__name__}")
        result = func(*args, **kwargs)
        print(f"After executing {func.__name__}")
        return result
    return wrapper

@my_decorator
def my_function(name):
    print(f"Hello, {name}!")


my_function("Alice")
```

Question 54:

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

Answer:

```
def decorator1(func):
    def wrapper(*args, **kwargs):
        print("Decorator 1 before")
        result = func(*args, **kwargs)
        print("Decorator 1 after")
        return result
    return wrapper


def decorator2(func):
    def wrapper(*args, **kwargs):
        print("Decorator 2 before")
        result = func(*args, **kwargs)
        print("Decorator 2 after")
        return result
    return wrapper
```

```
@decorator1
@decorator2
def my_function():
    print("Inside my_function")


my_function()
```

Question 55:

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

Answer:

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


@my_decorator
def my_function(x, y):
    return x + y


result = my_function(5, 3)
```

```
print(result)
```

Question 56:

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

Answer:

```
import functools


def my_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print("Before function execution")
        result = func(*args, **kwargs)
        print("After function execution")
        return result
    return wrapper


@my_decorator
def my_function():
    print("Inside my_function")


print(my_function.__name__)
```

Question 57:

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

Answer:

```
class Calculator:
    @staticmethod
    def add(x, y):
        return x + y
```

Question 58:

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

Answer:

```
class Employee:
    count = 0

    def __init__(self, name):
        self.name = name
        Employee.count += 1

    @classmethod
    def get_employee_count(cls):
        return cls.count
```

Question 59:

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

Answer:

```
class StringFormatter:

    @staticmethod

    def reverse_string(string):

        return string[::-1]
```

Question 60:

Create a Python class 'Circle' with a class method calculate_area that calculates the area of a circle given its radius.

Answer:

```
import math



class Circle:

    @classmethod

    def calculate_area(cls, radius):

        return math.pi * radius ** 2
```

Question 61:

Create a Python class Temperature Converter with a static method 'celsius_to_fahrenheit that converts Celsius to Fahrenheit.

Answer:

```
class TemperatureConverter:
    @staticmethod
    def celsius_to_fahrenheit(celsius):
        return (celsius * 9/5) + 32
```

Question 62:

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

Answer:

The `__str__()` method is a special method in Python that defines how an object of a class should be represented as a string. When you use the `print()` function or convert an object to a string using `str()`, Python calls this method to determine the string representation.

Example:

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

    def __str__(self):
        return f"Person(name='{self.name}', age={self.age})"

person = Person("Alice", 30)
print(person)  # Output: Person(name='Alice', age=30)
```

Question 63:

How does the `__len__()` method work in Python? Provide an example.

Answer:

The `__len__()` method defines the behavior of the `len()` function when applied to an object of the class. It should return an integer representing the length or size of the object.

Example:

```
class MyList:

    def __init__(self, data):

        self.data = data


    def __len__(self):

        return len(self.data)


my_list = MyList([1, 2, 3, 4, 5])

print(len(my_list))  # Output: 5
```

Question 64:

Explain the usage of the `__add__()` method in Python classes. Provide an example.

Answer:

The `__add__()` method defines the behavior of the `+` operator when used with objects of the class. It allows you to implement custom addition operations for your objects.

Example:

```
class Number:

    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return Number(self.value + other.value)

num1 = Number(10)
num2 = Number(20)
result = num1 + num2
print(result.value)  # Output: 30
```

Question 65:

What is the purpose of the `__getitem__()` method in Python? Provide an example.

Answer:

The `__getitem__()` method enables you to implement indexing and slicing for your custom objects. It is called when you use the `[]` operator on an object of the class.

Example:

```
class MyList:
    def __init__(self, data):
        self.data = data

    def __getitem__(self, index):
        return self.data[index]

my_list = MyList([1, 2, 3, 4, 5])
```

```
print(my_list[2])  # Output: 3
```

Question 66:

Explain the usage of the `__iter__()` and `__next__()` methods in Python. Provide an example using iterators.

Answer:

The `__iter__()` method returns an iterator object, and the `__next__()` method is called on the iterator object to get the next value. This allows you to implement custom iterators for your classes.

Example:

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

    def __iter__(self):
        return self

    def __next__(self):
        if self.start < self.end:
            value = self.start
            self.start += 1
            return value
        else:
            raise StopIteration
```

```
for num in MyRange(1, 5):
    print(num)  # Output: 1 2 3 4
```

Question 67:

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

Answer:

Getter methods are used to retrieve the value of an attribute in a controlled way. They can be used to perform validation or calculations before returning the value.

Example:

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

    @property
    def name(self):
        return self._name

    @property
    def age(self):
        return self._age
```

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

print(person.name)  # Output: Alice
```

Question 68:

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

Answer:

Setter methods are used to modify the value of an attribute in a controlled way. They can be used to perform validation or calculations before setting the value.

Example:

```
class Person:

    def __init__(self, name, age):

        self._name = name

        self._age = age


    @property

    def name(self):

        return self._name


    @name.setter

    def name(self, new_name):

        self._name = new_name


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

```
person.name = "Bob"

print(person.name)  # Output: Bob
```

Question 69:

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

Answer:

The `@property` decorator in Python is used to define a getter method for a class attribute, allowing you to access the attribute like a property (without parentheses). It is part of Python's property mechanism and provides a way to encapsulate data by allowing controlled access.

The `@property` decorator enables:

1. **Read-only properties.**
2. **Validation or computation when accessing attributes.**
3. **Creating attributes that act like methods but are accessed like attributes.**

Syntax and Example

Here's an example illustrating the usage of the `@property` decorator:

```
class Circle:

    def __init__(self, radius):

        self._radius = radius  # Private attribute



    @property

    def radius(self):

        """Getter for radius."""

        return self._radius
```

```python
    @radius.setter

    def radius(self, value):

        """Setter for radius with validation."""

        if value < 0:

            raise ValueError("Radius cannot be negative.")

        self._radius = value


    @property

    def area(self):

        """Read-only property for the area of the circle."""

        return 3.14159 * self._radius ** 2



# Creating an object of the Circle class

circle = Circle(5)


# Accessing the radius property

print(f"Radius: {circle.radius}")  # Output: Radius: 5


# Setting a new radius using the setter

circle.radius = 10

print(f"Updated Radius: {circle.radius}")  # Output: Updated Radius: 10
```

```
# Accessing the computed area (read-only property)

print(f"Area: {circle.area}")  # Output: Area: 314.159


# Attempting to set a negative radius (will raise an error)

try:

    circle.radius = -3

except ValueError as e:

    print(e)  # Output: Radius cannot be negative.
```

Explanation of the Code:

1. `@property`:
    - Decorates the `radius` method, making it behave like a property (getter). You can access `circle.radius` without parentheses.
2. `@radius.setter`:
    - This allows setting the value of `radius` while performing validation. It ensures the radius cannot be negative.
3. Read-only Property:
    - The `area` method is decorated with `@property`, but no setter is provided. Hence, `circle.area` is read-only.
4. Encapsulation:
    - The `_radius` attribute is private (indicated by a leading underscore), and access is controlled via the `@property` decorator.

Output of the Code:

```
Radius: 5
```

```
Updated Radius: 10

Area: 314.159

Radius cannot be negative.
```

Question 70:

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

Answer:

The `@deleter` decorator is used to define a deleter method for a property. It allows you to delete the value of an attribute in a controlled way.

Example:

```
class Person:

    def __init__(self, name, age):

        self._name = name

        self._age = age


    @property

    def name(self):

        return self._name


    @name.setter

    def name(self, new_name):

        self._name = new_name


    @name.deleter
    def name(self):
        del self._name


person = Person("Alice", 30)

del person.name

print(person.name)  # Output: AttributeError: 'Person' object has no attribute '_name'
```

Question 71:

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

Answer:

Encapsulation is one of the core principles of **Object-Oriented Programming (OOP)**. It is the process of **hiding internal implementation details** of a class and providing controlled access to the data using methods or properties. In Python, the `@property` decorator plays a key role in encapsulation by enabling **getter**, **setter**, and **deleter** methods to control access to private attributes.

Encapsulation and Property Decorators

1. **Private Attributes**: Class attributes are prefixed with an underscore `_` to indicate they are private.
2. **Property Decorators**: The `@property` decorator allows controlled access to these private attributes.
3. **Validation and Computation**: Using `@property` and its associated setters (`@property.setter`), we can enforce validation or computations when accessing or modifying private attributes.

Example Illustrating Encapsulation with Property Decorators

python

Copy code

```
class BankAccount:

    def __init__(self, balance):

        self._balance = balance  # Private attribute



    @property

    def balance(self):

        """Getter for balance. Controlled access to the private
attribute."""

        return self._balance



    @balance.setter

    def balance(self, value):

        """Setter for balance with validation."""

        if value < 0:

            raise ValueError("Balance cannot be negative.")

        self._balance = value



    @balance.deleter

    def balance(self):

        """Deleter for balance."""

        print("Deleting balance...")

        self._balance = None
```

```
# Creating an object of BankAccount

account = BankAccount(1000)


# Accessing the balance using the getter

print(f"Initial Balance: {account.balance}")  # Output: Initial
Balance: 1000


# Updating balance using the setter

account.balance = 1500

print(f"Updated Balance: {account.balance}")  # Output: Updated
Balance: 1500


# Attempting to set a negative balance (will raise an error)

try:

    account.balance = -500

except ValueError as e:

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


# Deleting the balance

del account.balance

print(f"Balance after deletion: {account.balance}")  # Output:
Balance after deletion: None
```

Explanation of the Code:

1. **Private Attribute**:
   - `_balance` is marked as private to indicate it should not be accessed directly.
2. **`@property` Decorator**:
   - Defines the **getter** for `balance`, allowing read-only access.
3. **`@balance.setter`**:
   - Allows controlled modification of the balance with validation (e.g., preventing negative values).
4. **`@balance.deleter`**:
   - Provides a way to delete the attribute with a custom message.
5. **Encapsulation**:
   - The internal attribute `_balance` is accessed and modified through the getter and setter, ensuring encapsulation and validation.

Output of the Code:

yaml

Copy code

```
Initial Balance: 1000

Updated Balance: 1500

Balance cannot be negative.

Deleting balance...

Balance after deletion: None
```