```
#1.Explain the importance of Functions
""" ans: functions are the block of code that do specific job and can
b+ ca +d again and again as per requirment of code.
there are certain reson for which function is important-1:Function is
a small , reusable  program witin  big program. we can give it a name
and tell it what to do.
2.Instead of writing  same code again and again, we can use a
function,to make our code short .
3.Functions help to organize our code.by using function we can make
our code short,neat and clean"""
```

```
{"type":"string"}
```

```
#2Write a basic function to greet students.
def greeting(a):

     print("hello welcome to pw skills ",a)
greeting("abhijit")
```

```
hello welcome to pw skills  abhijit
```

```
#3. What is the difference between print and return statements?
"""ans= in python we use both print and return statement.the main
difference between print and return are:
1:in python we use print statement to display output on the console
but we use return statement to returns a final value of a function
execution which may be used further in
 the code. """
```

```
{"type":"string"}
```

```
#4. What are *args and **kwargs?
"""ans= *args (Non-Keyword Arguments) and **kwargs (Keyword Arguments)
 *args =*args in function  is used to pass a variable number of
arguments to a function. It is used to pass a non-keyworded, variable-
length argument list.
  *args allows us to take in more arguments.
 **kwargs=**kwargs in function is used to pass a keyworded, variable-
length argument list.We use the name kwargs with the double star.
  The reason is that the double star allows us to pass through keyword
arguments"""
```

```
{"type":"string"}
```

```
#5. Explain the iterator function
"""ans=an iterator is an object that is used to iterate over iterable
objects like lists, tuples, dict andn set  iterators object is
initialized using the iter() method after that
it uses the next() method for iteration
example:string = "GFG"
char_iterator = iter(string)
```

```
print(next(char_iterator))
print(next(char_iterator))
print(next(char_iterator))"""

{"type":"string"}

#6.Write a code that generates the squares of numbers from 1 to n
using a generator
def square_num(n):
  for i in range(1,n+1):
    yield i**2
for square in square_num(10):
  print(square)

1
4
9
16
25
36
49
64
81
100

#7. Write a code that generates palindromic numbers up to n using a
generator
def is_palindrom(n):
  for i in range(1,n+1):
    if str(i)==str(i)[::-1]:
      yield i
for palindrom_generator in is_palindrom(100):
  print(palindrom_generator)


1
2
3
4
5
6
7
8
9
11
22
33
44
55
66
```

```
77
88
99

#8. Write a code that generates even numbers from 2 to n using a
generator
def even_num(n):
  for i in range(1,n+1):
    if i%2==0:
      yield i
for even_finder in even_num(100):
  print(even_finder)

2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
42
44
46
48
50
52
54
56
58
60
62
64
66
68
70
72
74
```

```
76
78
80
82
84
86
88
90
92
94
96
98
100
```

```
#9.Write a code that generates powers of two up to n using a generator
def power_num(n):
  for i in range(1,n+1):
    yield i**2
for power_generator in power_num(10):
  print(power_generator)
```

```
1
4
9
16
25
36
49
64
81
100
```

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


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 gen_primes(n):
    for num in range(2, n + 1):
        if is_prime(num):
            yield num

# Example usage:
```

```
n = 200
print(list(gen_primes(n)))
```

```
[5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73,
75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99, 101, 103, 105,
107, 109, 111, 113, 115, 117, 119, 121, 123, 125, 127, 129, 131, 133,
135, 137, 139, 141, 143, 145, 147, 149, 151, 153, 155, 157, 159, 161,
163, 165, 167, 169, 171, 173, 175, 177, 179, 181, 183, 185, 187, 189,
191, 193, 195, 197, 199]
```

```
#11.Write a code that uses a lambda function to calculate the sum of
two numbers
sum = lambda x, y: x + y
result = sum(2, 9)
print(result)
```

```
11
```

```
#12. Write a code that uses a lambda function to calculate the square
of a given number
square = lambda x: x ** 2
print(square(3))
```

```
9
```

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

check_number = lambda num: "even" if num % 2 == 0 else "odd"
number = int(input("Enter a number: "))
result = check_number(number)
print(result)
```

```
Enter a number: 6
even
```

```
#14. Write a code that uses a lambda function to concatenate two
strings
concatenate_strings = lambda str1, str2: str1 + str2
result = concatenate_strings("Hello, ", "world!")
print(result)
```

```
Hello, world!
```

```
#15.Write a code that uses a lambda function to find the maximum of
three given numbers
find_max = lambda a, b, c: max(a, b, c)
result = find_max(10, 5, 8)
print("Maximum Number:", result)
```

```
Maximum Number: 10
```

```
#16.Write a code that generates the squares of even numbers from a
given list
def get_even_squares(lst):
    even_squares = []
    for num in lst:
        if num % 2 == 0:
            even_squares.append(num ** 2)
    return even_squares

my_list = [1, 2, 3, 4, 5, 6, 7, 8]
result = get_even_squares(my_list)
print(result)
```

```
[4, 16, 36, 64]
```

```
#17. Write a code that calculates the product of positive numbers from
a given list
def calculate_product(lst):
    product = 1
    for num in lst:
        if num > 0:
            product *= num
    return product
my_list = [2, 3, 5, 4, 6, 7]
result = calculate_product(my_list)
print(result)
```

```
5040
```

```
#18def double_odd_values(lst):
def double_odd_values(lst):
    return [x * 2 for x in lst if x % 2 != 0]
n = [3, 5, 7]
result = double_odd_values(n)
print(result)
```

```
[6, 10, 14]
```

```
#19write a code to calculate the sum of cubes of numbers from a given
list
def sum_of_cubes(numbers):
    return sum(num ** 3 for num in numbers)
my_list = [1, 2, 3, 4, 5]
result = sum_of_cubes(my_list)
print(result)
```

```
225
```

```
#20. Write a code that filters out prime numbers from a given list


#21.Write a code that uses a lambda function to calculate the sum of two numbers
add_numbers = lambda x, y: x + y
result = add_numbers(5, 3)
print( result)
```

```
8
```

```
#22.Write a code that uses a lambda function to calculate the square of a given number
square = lambda x: x**2

number = 5
result = square(number)
print(result)
```

```
25
```

```
#23. Write a code that uses a lambda function to check whether a given number is even or odd
check_number = lambda num: "even" if num % 2 == 0 else "odd"
number = int(input("Enter a number: "))
result = check_number(number)
print(result)
```

```
Enter a number: 8
even
```

```
#24.  Write a code that uses a lambda function to concatenate two string
words = ['Hello', 'world']
result = ''.join(words)
print(result)
```

```
Helloworld
```

```
#25.Write a code that uses a lambda function to find the maximum of three given numbers
find_max = lambda a, b, c: max(a, b, c)

# Example usage
num1, num2, num3 = 10, 45, 15
maximum = find_max(num1, num2, num3)
print(maximum)
```

```
45
```

```
#26.What is encapsulation in OOP?
"""answer- Encapsulation is one of the fundamental concepts in object-oriented programming. It describes the idea of wrapping data and the methods that work on data within one unit.
This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data.
To prevent accidental change, an object’s variable can only be changed by an object’s method. Those types of variables are known as private variables."""
```

```
{"type":"string"}
```

```
#27.Explain the use of access modifiers in Python classes
"""answer-A Class in Python has three types of access modifiers:

1-Public Access Modifier
2-Protected Access Modifier
3-Private Access Modifier
public access modifier-The members of a class that are declared public are easily accessible from any part of the program.
 All data members and member functions of a class are public by default.
Protected Access Modifier-The members of a class that are declared protected are only accessible to a class derived from it.
Data members of a class are declared protected by adding a single underscore ‘_’ symbol before the data member of that class.
Private Access Modifier-The members of a class that are declared private are accessible within the class only, private access modifier is the most secure access modifier.
 Data members of a class are declared private by adding a double underscore ‘__’ symbol before the data member of that class"""
```

```
{"type":"string"}
```

```
#28. What is inheritance in OOP
"""answer-One of the core concepts in object-oriented programming languages is inheritance.
 It is a mechanism that allows you to create a hierarchy of classes that share a set of properties and methods by deriving a class from another class.
 Inheritance is the capability of one class to derive or inherit the properties from another class. """
```

```
{"type":"string"}
```

```
#29.Define polymorphism in OOP
""" answer-The word polymorphism means having many forms. In programming, polymorphism means the same function name being used for different types.
 The key difference is the data types and number of arguments used in function."""
```

```
{"type":"string"}
```

```
#30. Explain method overriding in Python
"""answer-Method overriding is an ability of any object-oriented
programming language that allows a subclass or child class to provide
a specific implementation of a method that is already provided by one
of its super-classes or parent classes.
When a method in a subclass has the same name, same parameters or
signature and same return type as a method in its super-class, then
the method in the subclass is said to override the method in the
super-class."""
```

```
{"type":"string"}
```

```
#31.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!")

a = Dog()

a.make_sound()
```

```
Woof!
```

```
#32.Define a method move in the Animal class that prints "Animal
moves". Override the move method in the Dog class to print "Dog runs.
class Animal:
    def move(self):
        print("Animal moves")

class Dog(Animal):
    def move(self):
a=Dog()
a.move()
```

```
  File "<ipython-input-20-0738abef1e30>", line 8
    a=Dog()
    ^
IndentationError: expected an indented block after function definition
on line 7
```

```python
#33. 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

#34. Create a class GermanShepherd inheriting from Dog and override the make_sound method to print "Bark!
class Dog:
    def make_sound(self):
        print("Woof!")

class GermanShepherd(Dog):
    def make_sound(self):
        print("Bark!")
dog = GermanShepherd()
dog.make_sound()
```

```
Bark!
```

```python
#35.Define constructors in both the Animal and Dog classes with different initialization parameters
class Animal:
    def __init__(self, height, age, weight):
        self.height = height
        self.age = age
        self.weight = weight
class Dog(Animal):
    def __init__(self, height, age, weight, dogType):
      self.height = height
      self.age = age
      self.dogType = dogType
      self.weight = weight
my_dog = Dog(height=5, age=3, weight=2, dogType="golden")

print(my_dog)
```

```
<__main__.Dog object at 0x7953962c74f0>
```

```python
#36 What is abstraction in Python? How is it implemented
"""answer-Data abstraction is one of the most essential concepts of Python OOPs which is used to hide irrelevant details from the user and show the details that are relevant to the users.
For example, the readers of geeksforgeeks only know that a writer can write an article on geeksforgeeks, and when it gets published readers can read the articles but the reader is not aware of the background process of publishing the article."""
from abc import ABC, abstractmethod
```

```
class Base(ABC):
    @abstractmethod
    def method_1(self):
        # Empty body
        pass

#37. Explain the importance of abstraction in object-oriented
programming
"""answer- the importance of abstraction in object oriented
programming is :1:Complexity Management: Abstraction helps handle
complexity by hiding unnecessary details from the user.
2:Code Reusability: By abstracting away low-level details, you create
reusable code. Objects in OOP languages provide abstractions that hide
internal implementation.
3:Data Abstraction: Deals with manipulating complex objects while
hiding their underlying structure.
4:limination of Redundancy: Abstraction eliminates redundancy by
grouping related classes"""

{"type":"string"}

#38.How are abstract methods different from regular methods in Python?
"""answer-abstract method is different from regular method in python
as both are serves in different purpose like:1:in abstraction method
we use @abstractmethod but in regular method we do not use such thing.
2:regular method are used without any decorator
3:Regular methods can be overridden in derived classes but abstract
class do not override """

{"type":"string"}

#39.How can you achieve abstraction using interfaces in Python?
"""
In Python, abstraction can be achieved using abstract classes or
interfaces provided by the “abc” module"""

class Base(ABC):
    @abstractmethod
    def method_1(self):
        # Empty body
        pass

#40. Can you provide an example of how abstraction can be utilized to
create a common interface for a group of related classes in Python?
from abc import ABC, abstractmethod

class Animal(ABC):
    def __init__(self, name):
        self.name = name
```

```
    @abstractmethod
    def walk(self):
        pass

class Dog(Animal):
    def walk(self):
        print(f"{self.name} walks like a dog.")


my_dog = Dog("Buddy")
my_dog.walk()
```

```
Buddy walks like a dog.
```

```
#41.How does Python achieve polymorphism through method overriding
"""answer-Polymorphism in Python is achieved through method
overriding, which allows subclasses to provide specific
implementations for methods that are defined in their superclass.
 This enables a single interface to be used for different underlying
form"""
 #example
class Animal:
    def make_sound(self):
        return "Some generic sound"

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

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

def animal_sound(animal):
    print(animal.make_sound())

# Example usage:
dog = Dog()
cat = Cat()
generic_animal = Animal()

animal_sound(dog)
animal_sound(cat)
animal_sound(generic_animal)
```

```
Woof!
Meow!
Some generic sound
```

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

```python
generic_animal = Animal()
dog = Dog()
print(generic_animal.make_sound())
print(dog.make_sound())
```

```
Some generic sound
Woof!
```

```python
#43. Define a base class and multiple subclasses with overridden
methods
class Animal:
    def speak(self):
        print("Animal speaks something...")

class Dog(Animal):
    def speak(self):
        print("Dog barks!")

class Cat(Animal):
    def speak(self):
        print("Cat meows!")


dog = Dog()
dog.speak()

cat = Cat()
cat.speak()
```

```
Dog barks!
Cat meows!
```

```python
#44.How does polymorphism improve code readability and reusability
"""polymorphism improve code readability:-Polymorphism enables
developers to write generic code that operates on superclass objects,
 unaware of the specific subclasses being used.
By creating a common interface or superclass, you can abstract away
differences between various types of objects.
This promotes reusability by allowing the same code to be applied to
different objects and scenarios.
polymorphism improvs code reuauability:-Polymorphism promotes cleaner,
more concise code by reducing redundancy.
You can reuse code written for the base class, making it easier to
understand and maintain.
When you use polymorphic methods, you reduce code duplication and make
your code more intuitive.
It becomes easier to reason about behavior without worrying about
specific implementations.
By adhering to the principle of “one interface, multiple
implementations,” polymorphism fosters elegant and efficient code."""
```

```
{"type":"string"}

#45. Describe how Python supports polymorphism with duck typing
"""Duck typing is the principle that allows for the implicit
implementation of polymorphism in Python.
 There's no need to explicitly declare interfaces or return types. If
an object has the necessary method or attribute, you can use it,
regardless of the object's class or inheritance hierarchy."""

{"type":"string"}

#46.How do you achieve encapsulation in Python
"""Encapsulation in Python refers to the practice of hiding the
implementation details of an object or class, and only exposing a
public interface for interacting with it.
Here are some key points about encapsulation in Python:

Wrapping Data and Methods:
Encapsulation involves bundling data (attributes) and methods
(functions) within a single unit (usually a class).
When you create a class, you are implementing encapsulation.
Access Modifiers:
Access modifiers control the visibility of class members (variables
and methods).
In Python, we have three access levels:
Public: Members accessible from anywhere (no special syntax required).
Protected: Members prefixed with a single underscore (_). Accessible
within the class and its subclasses.
Private: Members prefixed with double underscores (__). Accessible
only within the class itself.
"""

{"type":"string"}

#47.Can encapsulation be bypassed in Python? If so, how?
"""Can encapsulation be bypassed in Python? If so, how?
ChatGPT
Yes, encapsulation can be bypassed in Python, though it's generally
not recommended to do so. Encapsulation is the practice of restricting
access to certain parts of an object to protect the integrity of its
data and ensure it is used as intended. In Python, this is typically
implemented using naming conventions and the language's support for
private and protected attributes.

Private and Protected Attributes
Protected Attributes: Indicated by a single underscore (_), suggesting
that these are intended for internal use within the class or its
subclasses. However, this is only a convention and not enforced by the
Python interpreter.

Private Attributes: Indicated by a double underscore (__), triggering
```

```
name mangling, which makes it harder (but not impossible) to access
from outside the class.

Example of Encapsulation"""

class MyClass:
    def __init__(self, value):
        self._protected_attribute = value
        self.__private_attribute = value

    def get_private_attribute(self):
        return self.__private_attribute

    def set_private_attribute(self, value):
        self.__private_attribute = value

obj = MyClass(10)
print(obj._protected_attribute)
print(obj.get_private_attribute())
```

```
10
10
```

```
#48. Implement a class BankAccount with a private balance attribute.
Include methods to deposit, withdraw, and check the balance
class BankAccount:
    def __init__(self, account_holder_name):
        self.account_holder_name = account_holder_name
        self.balance = 0

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

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
        else:
            print("Insufficient balance.")

    def get_balance(self):
        return self.balance
ac_no_1 = BankAccount( "Toninho Takeo")
ac_no_1.deposit(500)
ac_no_1.withdraw(200)
print (ac_no_1.check_balance() )
```

```
---------------------------------------------------------------------------
-----
AttributeError                            Traceback (most recent call
last)
<ipython-input-52-ad45e4c31a36> in <cell line: 21>()
```

```
     19 ac_no_1.deposit(500)
     20 ac_no_1.withdraw(200)
---> 21 print (ac_no_1.check_balance() )

AttributeError: 'BankAccount' object has no attribute 'check_balance'
```

```python
#49.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):
        if "@" in email and "." in email:
            self.__email = email
            print("Email updated successfully.")
        else:
            print("Invalid email format.")

    def get_email(self):
        return self.__email

    def get_name(self):
        return self.__name

# Example usage
person = Person("abhijit ", "abhijitmaharana@.com")

print(person.get_email())


person.set_email("abhijitmaharana@gmail.com")
print(person.get_email())


person.set_email("jane.doe")
print(person.get_name())
```

```
abhijitmaharana@.com
Email updated successfully.
abhijitmaharana@gmail.com
Invalid email format.
abhijit
```

```python
#50.Why is encapsulation considered a pillar of object-oriented
programming (OOP)?
"""encapsulation is considered a pillar of object oriented programming
because Encapsulation is a key principle of object-oriented
programming that promotes data hiding, abstraction, and modularity.
```

```
 By encapsulating data and behavior within a class, developers can
create more secure, reusable, and maintainable code."""

{"type":"string"}

#51.Create a decorator in Python that adds functionality to a simple
function by printing a message before and after the function execution
def my_decorator(func):
    def wrapper():
        print("Before function execution")
        func()
        print("After function execution")
    return wrapper

@my_decorator
def my_function():
    print("Inside the function")
my_function()
```

```
Before function execution
Inside the function
After function execution
```

```
#52. Modify the decorator to accept arguments and print the function
name along with the message
def print_info(message):
    def decorator(func):
        def wrapper(*args, **kwargs):
            print(f"{message} - Executing function: {func.__name__}")
            result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator
@print_info("Running")
def add(a, b):
    return a + b

@print_info("Processing")
def multiply(a, b):
    return a *b
print(say_hello("Alice"))

print(add(2, 3))

print(multiply(4, 5))
```

```
---------------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
last)
<ipython-input-58-e48f64094e2b> in <cell line: 17>()
```

```
     15 def multiply(a, b):
     16     return a *b
---> 17 print(say_hello("Alice"))
     18
     19 print(add(2, 3))

NameError: name 'say_hello' is not defined
```

```
#53. Create two decorators, and apply them to a single function.
Ensure that they execute in the order they are applied
def decorator_one(func):
    def wrapper(*args, **kwargs):
        print("Decorator One: Before function execution")
        result = func(*args, **kwargs)
        print("Decorator One: After function execution")
        return result
    return wrapper
def decorator_two(func):
    def wrapper(*args, **kwargs):
        print("Decorator Two: Before function execution")
        result = func(*args, **kwargs)
        print("Decorator Two: After function execution")
        return result
    return wrapper
@decorator_one
@decorator_two
def my_function():
    print("Inside the function")


my_function()
```

```
Decorator One: Before function execution
Decorator Two: Before function execution
Inside the function
Decorator Two: After function execution
Decorator One: After function execution
```

```
#54.Modify the decorator to accept and pass function arguments to the
wrapped function

#55.Create a decorator that preserves the metadata of the original
function
import functools

def my_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):

        print("Calling decorated function")
        return func(*args, **kwargs)
```

```
    return wrapper


@my_decorator
def my_function(a, b):

    return a + b

print(my_function.__name__)
print(my_function.__doc__)
print(my_function.__annotations__)
```

```
my_function
None
{}
```

```
#56.Create a Python class `Calculator` with a static method `add` that
takes in two numbers and returns their sum
class Calculator:
    @staticmethod
    def add(a, b):

        return a + b


result = Calculator.add(3, 5)
print(result)
```

```
8
```

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

    _employee_count = 0

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

        Employee._employee_count += 1

    @classmethod
    def get_employee_count(cls):

        return cls._employee_count
emp1 = Employee("Alice")
emp2 = Employee("Bob")
emp3 = Employee("Charlie")

print(Employee.get_employee_count())
```

```
3
```

```
#58.Create a Python class `StringFormatter` with a static method
`reverse_string` that takes a string as input and returns its reverse
class StringFormatter:
    @staticmethod
    def reverse_string(input_string):
        return input_string[::-1]

reversed_str = StringFormatter.reverse_string("Hello, World!")
print(reversed_str)
```

```
!dlroW ,olleH
```

```
#59. Create a Python class `Circle` with a class method
`calculate_area` that calculates the area of a circle given its radius
import math

class Circle:
    @classmethod
    def calculate_area(cls, radius):

        return math.pi * radius * radius
radius = 5.0
area = Circle.calculate_area(radius)
print(f"The area of a circle with radius {radius} is: {area:.2f}")
```

```
The area of a circle with radius 5.0 is: 78.54
```

```
#60.Create a Python class `TemperatureConverter` with a static method
`celsius_to_fahrenheit` that converts Celsius to Fahrenheit
class TemperatureConverter:
    @staticmethod
    def celsius_to_fahrenheit(celsius):

        return (celsius * 9/5) + 32

celsius_temperature = 25.0
fahrenheit_temperature =
TemperatureConverter.celsius_to_fahrenheit(celsius_temperature)
print(f"{celsius_temperature} degrees Celsius is equal to
{fahrenheit_temperature:.2f} degrees Fahrenheit.")
```

```
25.0 degrees Celsius is equal to 77.00 degrees Fahrenheit.
```

```
#61.What is the purpose of the __str__() method in Python classes?
Provide an example
class Book:
    def __init__(self, title, author):
        self.title = title
```

```
        self.author = author

    def __str__(self):
        return f"Book: '{self.title}' by {self.author}"

book1 = Book("1984", "George Orwell")
book2 = Book("To Kill a Mockingbird", "Harper Lee")

print(book1)
print(book2)

book_str = str(book1)
print(book_str)
```

```
Book: '1984' by George Orwell
Book: 'To Kill a Mockingbird' by Harper Lee
Book: '1984' by George Orwell
```

```
#62. How does the __len__() method work in Python? Provide an example
class MyList:
    def __init__(self, initial_list=None):
        if initial_list is None:
            initial_list = []
        self.data = initial_list

    def __len__(self):
        return len(self.data)
my_list = MyList([1, 2, 3, 4, 5])
print(len(my_list))

empty_list = MyList()
print(len(empty_list))
```

```
5
0
```

```
#63. Explain the usage of the __add__() method in Python classes.
Provide an example
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):

        new_x = self.x + other.x
        new_y = self.y + other.y
        return Vector(new_x, new_y)

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

```
v1 = Vector(2, 3)
v2 = Vector(5, 7)

v3 = v1 + v2
print(v3)

#64. What is the purpose of the __getitem__() method in Python?
Provide an 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[0])
print(my_list[3])
print(my_list[1:4])

1
4
[2, 3, 4]

#65. Explain the usage of the __iter__() and __next__() methods in
Python. Provide an example using iterators
"""__iter__() Method:
The __iter__() method is used to create an iterator for a given object
(such as a list, tuple, or custom object).
It returns an iterator object that allows us to access elements one at
a time.
Syntax: iter(object) or iter(callable, sentinel)
object: The collection or custom object for which we want to create an
iterator.
callable, sentinel: Optional parameters for customizing the iteration
process.
If we call the iterator after all elements have been iterated, a
StopIterationError is raised.
__next__() Method:
The __next__() method returns the next element of the iteration.
If there are no more elements, it raises a StopIteration exception.
It's part of the iterable and iterator interface, allowing us to
create custom iterable objects.
"""
listA = ['a', 'e', 'i', 'o', 'u']
iter_listA = iter(listA)
try:
    print(next(iter_listA))
    print(next(iter_listA))
```

```
    print(next(iter_listA))
    print(next(iter_listA))
    print(next(iter_listA))
    print(next(iter_listA))
except StopIteration:
    pass


lst = [11, 22, 33, 44, 55]
iter_lst = iter(lst)
while True:
    try:
        print(iter_lst.__next__())
    except StopIteration:
        break
```

```
a
e
i
o
u
11
22
33
44
55
```

```
#66. What is the purpose of a getter method in Python? Provide an
example demonstrating the use of a getter method using property
decorators
class Temperature:
    def __init__(self, celsius):
        self._celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @property
    def fahrenheit(self):
        return self._celsius * 9/5 + 32


temp = Temperature(25)
print(f"Temperature in Celsius: {temp.celsius}")
print(f"Temperature in Fahrenheit: {temp.fahrenheit}")
```

```
Temperature in Celsius: 25
Temperature in Fahrenheit: 77.0
```

```python
#67.Explain the role of setter methods in Python. Demonstrate how to use a setter method to modify a class attribute using property decorators
import math

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

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, new_radius):
        if new_radius <= 0:
            raise ValueError("Radius must be positive")
        self._radius = new_radius

    @property
    def diameter(self):
        return 2 * self._radius

    @property
    def area(self):
        return math.pi * (self._radius ** 2)


circle = Circle(5)
print(f"Radius: {circle.radius}")
print(f"Diameter: {circle.diameter}")
print(f"Area: {circle.area:.2f}")
```

```
Radius: 5
Diameter: 10
Area: 78.54
```

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

@property decorator is a built-in decorator in Python which is helpful in defining the properties effortlessly without manually calling the inbuilt function property(). Which is used to return the property attributes of a class from the stated getter, setter and deleter as parameters. Now, lets see some examples to illustrate the use of @property decorator in Python: Example 1:

```python
# Python program to illustrate the use of
# @property decorator
```

```python
# Defining class
class Portal:

    # Defining __init__ method
    def __init__(self):
        self.__name =''

    # Using @property decorator
    @property

    # Getter method
    def name(self):
        return self.__name

    # Setter method
    @name.setter
    def name(self, val):
        self.__name = val

    # Deleter method
    @name.deleter
    def name(self):
       del self.__name

# Creating object
p = Portal();

# Setting name
p.name = 'GeeksforGeeks'

# Prints name
print (p.name)

# Deletes name
del p.name

# As name is deleted above this
# will throw an error
print (p.name)

#69.Explain the use of the @deleter decorator in Python property
decorators. Provide a code example demonstrating its application
"""In Python, the @deleter decorator is not directly used within
property decorators. Instead, it is typically referred to as @property
for defining getter methods, @<property_name>.setter for defining
setter methods, and @<property_name>.deleter for defining deleter
methods.
 Together, these decorators allow you to define properties with custom
behavior for getting, setting, and deleting attributes of an
object."""
```

```python
 import math

class Circle:
    def __init__(self, radius):
        self._radius = radius  # Private attribute

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, new_radius):
        if new_radius <= 0:
            raise ValueError("Radius must be positive")
        self._radius = new_radius

    @property
    def diameter(self):
        return 2 * self._radius

    @property
    def area(self):
        return math.pi * (self._radius ** 2)

    @radius.deleter
    def radius(self):
        print("Deleting radius...")
        del self._radius
circle = Circle(5)
print(f"Radius: {circle.radius}")
print(f"Diameter: {circle.diameter}")
print(f"Area: {circle.area:.2f}")
del circle.radius


circle.radius = 7
print(f"New Radius: {circle.radius}")

#70.How does encapsulation relate to property decorators in Python?
Provide an example showcasing encapsulation using property decorators.
"""Encapsulation in object-oriented programming refers to the bundling
of data (attributes) and methods (functions) that operate on the data
into a single unit, typically a class.
 It allows you to control access to the data and hide implementation
details, thus preventing direct modification of the internal state
from outside the class.
  In Python, property decorators (@property, @<property_name>.setter,
@<property_name>.deleter) play a crucial role in achieving
encapsulation by providing controlled access to class attributes."""
class BankAccount:
```

```
    def __init__(self, initial_balance=0):
        self._balance = initial_balance  # Private attribute

    @property
    def balance(self):
        return self._balance

    @balance.setter
    def balance(self, new_balance):
        if new_balance < 0:
            raise ValueError("Balance cannot be negative")
        self._balance = new_balance

    @balance.deleter
    def balance(self):
        print("Deleting account...")
        del self._balance
account = BankAccount(1000)
print(f"Current Balance: {account.balance}")

account.balance = 1500
print(f"Updated Balance: {account.balance}")
del account.balance


Current Balance: 1000
Updated Balance: 1500
Deleting account...
```