5l20awhwm

July 17, 2024

1.explain the important function

1. Importance of Functions Functions are essential in programming for several reasons:

Modularity: They allow the code to be divided into smaller, manageable, and reusable blocks. Readability: Functions make the code easier to read and understand by providing a clear structure. Reusability: Once a function is defined, it can be reused multiple times throughout the code, reducing redundancy. Maintainability: Functions make it easier to update and maintain the code since changes in the logic only need to be made in one place.

2.write a basic python function to greed student

[9]:
```
def greed_student(name):
    print(f"Hello, {name}! welcome to school")
greed_student("malik")
```

```
Hello, malik! welcome to school
```

3.Difference Between print and return Statementsprint:

ans:-This statement outputs the result to the console. It doesn't affect the flow of the program and doesn't return any value to the caller. return: This statement exits the function and sends a value back to the caller. The returned value can be used in further calculations or logic.

4. What are *args and **kwargs*args:

This is used to pass a variable number of non-keyword arguments to a function. It allows you to pass a tuple of arguments. **kwargs: This is used to pass a variable number of keyword arguments to a function. It allows you to pass a dictionary of arguments.

5. Explain the iterator Function

anw:- An iterator is an object that contains a countable number of values and can be iterated upon, meaning you can traverse through all the values. Iterators implement two methods: **iter**() and **next**().

6. Writing a Code to Generate Squares of Numbers from 1 to n Using a Generator

[11]:
```
def generate_squares(n):
    for i in range(1, n + 1):
        yield i ** 2
```

. Writing a Code to Generate Palindromic Numbers up to n Using a Generator

[13]:
```
def generate_palindromes(n):
    for i in range(1, n + 1):
        if str(i) == str(i)[::-1]:
            yield i
```

Writing a Code to Generate Even Numbers from 2 to n Using a Generator python

[14]:
```
def generate_even_numbers(n):
    for i in range(2, n + 1, 2):
        yield i
```

9. Writing a Code to Generate Powers of Two up to n Using a Generator

[15]:
```
def generate_powers_of_two(n):
    for i in range(n + 1):
        yield 2 ** i
```

10. Writing a Code to Generate Prime Numbers up to n Using a Generator

[16]:
```
def generate_primes(n):
    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

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

11. Writing a Code that Uses a Lambda Function to Calculate the Sum of Two Numbers

[17]:
```
sum_two_numbers = lambda x, y: x + y
print(sum_two_numbers(3, 5))  # Output: 8
```

```
8
```

12. Writing a Code that Uses a Lambda Function to Calculate the Square of a Given Number

[77]:
```
square = lambda x: x ** 2
print(square(4))
```

```
16
```

13. Writing a Code that Uses a Lambda Function to Check Whether a Given Number is Even or Odd

[78]:
```
is_even = lambda x: x % 2 == 0
print(is_even(4))
print(is_even(5))
```

```
True
False
```

14. Writing a Code that Uses a Lambda Function to Concatenate Two Strings

[79]:
```
concat_strings = lambda s1, s2: s1 + s2
print(concat_strings("Hello", "World"))
```

```
HelloWorld
```

15. Writing a Code that Uses a Lambda Function to Find the Maximum of Three Given Numbers

[80]:
```
max_of_three = lambda x, y, z: max(x, y, z)
print(max_of_three(3, 5, 1))
```

```
5
```

16. Writing a Code that Generates the Squares of Even Numbers from a Given List

[81]:
```
def generate_even_squares(numbers):
    return (x ** 2 for x in numbers if x % 2 == 0)

numbers = [1, 2, 3, 4, 5, 6]
for square in generate_even_squares(numbers):
    print(square)  # Output: 4, 16, 36
```

```
4
16
36
```

17. Writing a Code that Calculates the Product of Positive Numbers from a Given List

[82]:
```
from functools import reduce

def product_of_positives(numbers):
    positives = filter(lambda x: x > 0, numbers)
    return reduce(lambda x, y: x * y, positives)

numbers = [-1, 2, 3, -4, 5]
print(product_of_positives(numbers))
```

```
30
```

18. Writing a Code that Doubles the Values of All Numbers from a Given List

[83]:
```
double_values = lambda numbers: [x * 2 for x in numbers]
numbers = [1, 2, 3, 4]
```

```
print(double_values(numbers))
```

```
[2, 4, 6, 8]
```

19. Writing a Code that Calculates the Sum of Cubes of Numbers from a Given List

[84]:
```
def sum_of_cubes(numbers):
    return sum(x ** 3 for x in numbers)

numbers = [1, 2, 3, 4]
print(sum_of_cubes(numbers))
```

```
100
```

20. Writing a Code that Filters Out Prime Numbers from a Given List

[85]:
```
def filter_primes(numbers):
    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

    return list(filter(is_prime, numbers))

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(filter_primes(numbers))
```

```
[2, 3, 5, 7]
```

21. Writing a Code that Uses a Lambda Function to Calculate the Sum of Two Numbers

[86]:
```
sum_two_numbers = lambda x, y: x + y
print(sum_two_numbers(3, 5))
```

```
8
```

22. Writing a Code that Uses a Lambda Function to Calculate the Square of a Given Number

[87]:
```
square = lambda x: x ** 2
print(square(4))
```

```
16
```

23. Writing a Code that Uses a Lambda Function to Check Whether a Given Number is Even or Odd

[88]:
```
is_even = lambda x: x % 2 == 0
print(is_even(4))
```

```
print(is_even(5))
```

```
True
False
```

24. Writing a Code that Uses a Lambda Function to Concatenate Two Strings

[89]:
```
concat_strings = lambda s1, s2: s1 + s2
print(concat_strings("Hello", "World"))
```

```
HelloWorld
```

25. Writing a Code that Uses a Lambda Function to Find the Maximum of Three Given Numbers

[90]:
```
max_of_three = lambda x, y, z: max(x, y, z)
print(max_of_three(3, 5, 1))
```

```
5
```

26. What is Encapsulation in OOP?

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It refers to the bundling of data (attributes) and methods (functions) that operate on the data into a single unit or class, and restricting access to some of the object's components. This is typically done using access modifiers to control the visibility of the class members.

27. Explain the Use of Access Modifiers in Python Classes

Python uses the following conventions for access modifiers: Public: Members are accessible from outside the class. No underscore is used. Protected: Members are intended to be accessible within the class and its subclasses. A single underscore _ is used. Private: Members are intended to be accessible only within the class. A double underscore ___ is used.

28. What is Inheritance in OOP?

Inheritance is a mechanism in OOP that allows one class (child class) to inherit attributes and methods from another class (parent class). This promotes code reusability and establishes a relationship between classes.

29. Define Polymorphism in OOP

Polymorphism allows objects of different classes to be treated as objects of a common super class. It is typically achieved through method overriding and interfaces, allowing one interface to be used for a general class of actions.

30. Explain Method Overriding in Python

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The method in the subclass should have the same name, signature, and parameters as the one in the superclass.

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!

[91]:
```
class Animal:
    def make_sound(self):
        print("Generic animal sound")

class Dog(Animal):
    def make_sound(self):
        print("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”.

[92]:
```
class Animal:
    def move(self):
        print("Animal moves")

class Dog(Animal):
    def move(self):
        print("Dog runs")
```

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.

[93]:
```
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!”

[94]:
```
class GermanShepherd(Dog):
    def make_sound(self):
        print("Bark!")
```

35. Define Constructors in Both the Animal and Dog Classes with Different Initialization Parameters

[95]:
```
class Animal:
    def __init__(self, species):
        self.species = species

class Dog(Animal):
    def __init__(self, name):
        super().__init__("Dog")
        self.name = name
```

36. What is Abstraction in Python? How is it Implemented?

Abstraction is the concept of hiding the complex implementation details and showing only the necessary features of an object. It is implemented in Python using abstract classes and interfaces, which are created using the abc module.

37. Explain the Importance of Abstraction in Object-Oriented Programming

Abstraction allows for simpler and more efficient coding by allowing programmers to focus on the interface rather than the implementation details. It helps in managing complexity by providing a clear separation between what an object does and how it does it.

38. How are Abstract Methods Different from Regular Methods in Python?

Abstract methods are declared in an abstract class and do not contain any implementation. Subclasses are required to provide implementations for these methods. Regular methods contain complete implementations and can be used directly.

39. How Can You Achieve Abstraction Using Interfaces in Python?

In Python, interfaces can be achieved using abstract base classes (ABCs) from the abc module. An abstract class can have abstract methods that must be implemented by its subclasses.

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?

[96]:
```python
from abc import ABC, abstractmethod

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

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

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

41. How Does Python Achieve Polymorphism Through Method Overriding?

Polymorphism in Python is achieved through method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass.

42. Define a Base Class with a Method and a Subclass that Overrides the Method

[97]:
```python
class BaseClass:
    def display(self):
        print("Base class display method")

class SubClass(BaseClass):
    def display(self):
```

```
        print("Subclass display method")
```

43. Define a Base Class and Multiple Subclasses with Overridden Methods

[98]:
```
class BaseClass:
    def display(self):
        print("Base class display method")

class SubClass1(BaseClass):
    def display(self):
        print("Subclass 1 display method")

class SubClass2(BaseClass):
    def display(self):
        print("Subclass 2 display method")
```

44. How Does Polymorphism Improve Code Readability and Reusability?

Polymorphism allows for the writing of more generic and reusable code. It enables a single function to work with different types of objects, thus improving readability and reducing redundancy.

45. Describe How Python Supports Polymorphism with Duck Typing

Python supports polymorphism through duck typing, which means that the type or class of an object is less important than the methods it defines. If an object implements the necessary methods, it can be used in place of another object.

46. How Do You Achieve Encapsulation in Python?

Encapsulation in Python is achieved using access modifiers to restrict access to the internal state of an object. Attributes can be made private by prefixing them with double underscores ___.

47. Can Encapsulation Be Bypassed in Python? If So, How?

Yes, encapsulation can be bypassed in Python using name mangling. Private attributes can still be accessed using _ClassName___attributeName.

48. Implement a Class BankAccount with a Private balance Attribute. Include Methods to Deposit, Withdraw, and Check the Balance.

[99]:
```
class BankAccount:
    def __init__(self):
        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 check_balance(self):
        return self.__balance
```

49. Develop a Person Class with Private Attributes name and email, and Methods to Set and Get the Email.

[100]:
```
class Person:
    def __init__(self, name, email):
        self.__name = name
        self.__email = email

    def get_email(self):
        return self.__email

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

50. Why is Encapsulation Considered a Pillar of Object-Oriented Programming (OOP)?

Encapsulation is considered a pillar of OOP because it helps in bundling data with the methods that operate on that data, thus restricting direct access to some of an object's components. This improves data integrity and security by preventing unintended interference and misuse.

51. Creating a Python Decorator to Print Messages Before and After Function Execution

[101]:
```
def message_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before execution")
        result = func(*args, **kwargs)
        print("After execution")
        return result
    return wrapper

@message_decorator
def example_function():
    print("Function is executing")

example_function()
```

```
Before execution
Function is executing
After execution
```

52. Modifying the Decorator to Accept Arguments and Print Function Name

[102]:
```
def message_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Before execution of {func.__name__}")
```

```
        result = func(*args, **kwargs)
        print(f"After execution of {func.__name__}")
        return result
    return wrapper

@message_decorator
def example_function(arg):
    print(f"Function is executing with argument: {arg}")

example_function("Hello")
```

```
Before execution of example_function
Function is executing with argument: Hello
After execution of example_function
```

53. Creating Two Decorators and Applying Them to a Single Function

[103]:
```
def first_decorator(func):
    def wrapper(*args, **kwargs):
        print("First decorator - Before execution")
        result = func(*args, **kwargs)
        print("First decorator - After execution")
        return result
    return wrapper

def second_decorator(func):
    def wrapper(*args, **kwargs):
        print("Second decorator - Before execution")
        result = func(*args, **kwargs)
        print("Second decorator - After execution")
        return result
    return wrapper

@first_decorator
@second_decorator
def example_function():
    print("Function is executing")

example_function()
```

```
First decorator - Before execution
Second decorator - Before execution
Function is executing
Second decorator - After execution
First decorator - After execution
```

54. Modifying the Decorator to Accept and Pass Function Arguments

[104]:
```python
def message_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Before execution of {func.__name__} with args: {args} and␣
 ↪kwargs: {kwargs}")
        result = func(*args, **kwargs)
        print(f"After execution of {func.__name__}")
        return result
    return wrapper

@message_decorator
def example_function(a, b):
    print(f"Function is executing with arguments: {a}, {b}")
    return a + b

example_function(5, 10)
```

```
Before execution of example_function with args: (5, 10) and kwargs: {}
Function is executing with arguments: 5, 10
After execution of example_function
```

[104]:
```
15
```

55. Creating a Decorator that Preserves the Metadata of the Original Function

[106]:
```python
from functools import wraps

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

@message_decorator
def example_function():
    """This is an example function."""
    print("Function is executing")

print(example_function.__name__)
print(example_function.__doc__)
```

```
example_function
This is an example function.
```

57. Creating a Calculator Class with a Static Method add

[107]:
```python
class Calculator:
    @staticmethod
    def add(a, b):
        return a + b

print(Calculator.add(5, 10))
```

```
15
```

58. Creating an Employee Class with a Class Method get_employee_count

[108]:
```python
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")

print(Employee.get_employee_count())
```

```
2
```

59. Creating a StringFormatter Class with a Static Method reverse_string

[109]:
```python
class StringFormatter:
    @staticmethod
    def reverse_string(s):
        return s[::-1]

print(StringFormatter.reverse_string("hello"))
```

```
olleh
```

60. Creating a Circle Class with a Class Method calculate_area

[110]:
```python
class Circle:
    @classmethod
    def calculate_area(cls, radius):
        from math import pi
        return pi * (radius ** 2)

print(Circle.calculate_area(5))
```

```
78.53981633974483
```

61. Creating a TemperatureConverter Class with a Static Method celsius_to_fahrenheit

[111]:
```python
class TemperatureConverter:
    @staticmethod
    def celsius_to_fahrenheit(celsius):
        return (celsius * 9/5) + 32

print(TemperatureConverter.celsius_to_fahrenheit(25))
```

```
77.0
```

62:- urpose of the **str**() Method in Python Classes

The **str**() method is used to define a string representation of an object. It is called by the str() built-in function and by the print function to produce a string representation of an object.

[112]:
```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name}, {self.age} years old"

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

```
Alice, 30 years old
```

63. How Does the **len**() Method Work in Python? Provide an Example

The **len**() method is used to define the behavior of the len() function for an object.

[113]:
```python
class CustomList:
    def __init__(self, items):
        self.items = items

    def __len__(self):
        return len(self.items)

custom_list = CustomList([1, 2, 3, 4])
print(len(custom_list))
```

```
4
```

64. Explain the Usage of the **add**() Method in Python Classes

The **add**() method allows customization of the behavior of the + operator.

[114]:
```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

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

v1 = Vector(1, 2)
v2 = Vector(3, 4)
print(v1 + v2)
```

```
Vector(4, 6)
```

65. Purpose of the **getitem**() Method in Python? Provide an Example

The **getitem**() method is used to access elements of a container object using square bracket notation.

[115]:
```
class CustomDict:
    def __init__(self):
        self.data = {}

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

    def __setitem__(self, key, value):
        self.data[key] = value

custom_dict = CustomDict()
custom_dict["name"] = "Alice"
print(custom_dict["name"])
```

```
Alice
```

66. Usage of the **iter**() and **next**() Methods in Python

These methods allow an object to be iterable.

[116]:
```
class CustomIterator:
    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

my_iter = CustomIterator([1, 2, 3])
for item in my_iter:
    print(item)
```

```
1
2
3
```

67. Purpose of a Getter Method in Python

[117]:
```
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)
print(person.age)
```

```
Alice
30
```

68. Explain the Role of Setter Methods in Python

Setter methods are used to set the values of private attributes.

[118]:
```
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, name):
        self.__name = name

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        if age >= 0:
            self.__age = age

person = Person("Alice", 30)
person.age = 35
print(person.age)
```

```
35
```

69. Purpose of the @property Decorator in Python

The @property decorator is used to define getter methods in a class.

[119]:
```
class Circle:
    def __init__(self, radius):
        self.__radius = radius

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

    @radius.setter
    def radius(self, radius):
        if radius > 0:
            self.__radius = radius

circle = Circle(5)
print(circle.radius)
circle.radius = 10
print(circle.radius)
```

```
5
10
```

70 Explain the Use of the @deleter Decorator in Python Property Decorators

The @deleter decorator is used to define a method that deletes an attribute.

[120]:
```
class Person:
    def __init__(self, name):
        self.__name = name

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

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

person = Person("Alice")
print(person.name)
del person.name
```

```
Alice
```

71. How Does Encapsulation Relate to Property Decorators in Python?

Encapsulation ensures that the internal representation of an object is hidden from the outside. Property decorators provide a way to implement encapsulation by allowing controlled access to an object's attributes.

[121]:
```
class Person:
    def __init__(self, name):
        self.__name = name

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

    @name.setter
    def name(self, name):
        self.__name = name

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

person = Person("Alice")
print(person.name)
person.name = "Bob"
print(person.name)
del person.name
```

```
Alice
Bob
```

[ ]: