```
#Question1: What are the five key concepts of Object-Oriented-Programming(OOP)?
```

In OOP (object-oriented programming) concepts in Python include Class, Object, Method, Inheritance, Polymorphism, Data Abstraction, and Encapsulation

```
#Question2: Write a python class for a car with attributes for 'make', 'model', and 'year', include a method to display the car's informa


class Car():

    # init method or constructor
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def show(self):
        print("Make by", self.make)
        print("Model is", self.model )
        print("Year in", self.year )


audi = Car("TATA","audi a4", "2023")

audi.show()
```

```
Make by TATA
Model is audi a4
Year in 2023
```

```
#Question3: Explain the difference between instance methods and class methods. Provide an example of each.
```

1. ## Instance Method in Python

   Instance methods are the most common type of methods in Python classes. They are associated with instances of a class and operate on the instance's data. When defining an instance method, the method's first parameter is typically named self, which refers to the instance calling the method. This allows the method to access and manipulate the instance's attributes.

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

    def introduce(self):
        return f"Hi, I'm {self.name} and I'm {self.age} years old."


# Creating an instance of the class
person1 = Person("Sanket Vishwakarma", 22)

# Calling the instance method
print(person1.introduce())
```

```
Hi, I'm Sanket Vishwakarma and I'm 22 years old.
```

2. #Class Method in Python Class methods are associated with the class rather than instances. They are defined using the @classmethod decorator and take the class itself as the first parameter, usually named cls. Class methods are useful for tasks that involve the class rather than the instance, such as creating class-specific behaviors or modifying class-level attributes.

```
class MyClass:
    class_variable = 0

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

    @classmethod
    def class_method(cls, x):
        cls.class_variable += x
```

```
        return cls.class_variable

obj1 = MyClass(5)
obj2 = MyClass(10)

print(MyClass.class_method(3))
print(MyClass.class_method(7))
```

```
3
10
```

```
#Question4: How does Python implement method overloading?Give an example.
```

- Method Overloading:

Two or more methods have the same name but different numbers of parameters or different types of parameters, or both. These methods are called overloaded methods and this is called method overloading.

```
# First product method.
# Takes two argument and print their
# product


def product(a, b):
    p = a * b
    print(p)

# Second product method
# Takes three argument and print their
# product


def product(a, b, c):
    p = a * b*c
    print(p)

# Uncommenting the below line shows an error
# product(4, 5)


# This line will call the second product method
product(4, 5, 5)
```

```
100
```

```
#Question5: What are the three type of access modifiers in Python? How are they denoted?
```

1. Access Modifiers can be categorized as Public, Protected and Private in a python.

- Public Access Modifier: Theoretically, public methods and fields can be accessed directly by any class.
- Protected Access Modifier: Theoretically, protected methods and fields can be accessed within the same class it is declared and its subclass
- Private Access Modifier: Theoretically, private methods and fields can be only accessed within the same class it is declared.

```
#Question6: Describe the five types of inheritance in Python. Provide a simple example of multiple inheritance.
```

Inheritance is defined as the mechanism of inheriting the properties of the base class to the child class.

There are five types of Python inheritance are as follow:

- SINGLE INHERITANCE.
- MULTIPLE INHERITANCE.
- MULTILEVEL INHERITANCE
- HIERARCHICAL INHERITANCE
- HYBRID INHERITANCE

```
class Mother:
    mothername = ""

    def mother(self):
        print(self.mothername)

# Base class2


class Father:
    fathername = ""

    def father(self):
        print(self.fathername)

# Derived class


class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)


# Driver's code
s1 = Son()
s1.fathername = "SATYANARAYAN VISHWAKARMA"
s1.mothername = "VIDYA VISHWAKARMA"
s1.parents()
```

```
Father : SATYANARAYAN VISHWAKARMA
Mother : VIDYA VISHWAKARMA
```

```
#Question7: What is the Method Resolution Order(MRO) in Python? How can you retrieve it programmatically?
```

Method Resolution Order :

Method Resolution Order(MRO) it denotes the way a programming language resolves a method or attribute. Python supports classes inheriting from other classes. The class being inherited is called the Parent or Superclass, while the class that inherits is called the Child or Subclass. In python, method resolution order defines the order in which the base classes are searched when executing a method. First, the method or attribute is searched within a class and then it follows the order we specified while inheriting. This order is also called Linearization of a class and set of rules are called MRO(Method Resolution Order).

```
#Question8: Create an abstract base class 'shape' with a abstract method 'area()'. Then create two subclasses 'circle' and 'rectangle' t
```

```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    def describe(self):
        print(f"This shape has an area of {self.area()}")
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):
        import math
        return math.pi * self.radius ** 2



rectangle = Rectangle(10,5)
print(f"Area:{rectangle.area()}")
```

```
Area:50
```

```
circle = Circle(5)
print(f"Area:{circle.area()}")
```

```
Area:78.53981633974483
```

```
#Question9: Demonstrate polymorphism by creating a function that can work with different shape objects to calculate and print their area
```

```
class Shape:
    def draw(self):
        raise NotImplementedError("Subclass must implement abstract method")

    def describe(self):
        print(f"This shape has an area of {self.area()}")
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):
        import math
        return math.pi * self.radius ** 2


rectangle = Rectangle(15,9)
print(f"Area:{rectangle.area()}")
```

```
Area:135
```

```
circle = Circle(8)
print(f"Area:{circle.area()}")
```

```
Area:201.06192982974676
```

```
#Question10: Implement encapsulation in a BankAccount class with privat attributes for balance and account_number. Include methods for d
```

```
class Bank_Account:
    def __init__(self):
        self.balance=0
        print("Hello!!! Welcome to the Deposit & Withdrawal Machine")

    def deposit(self):
        amount=float(input("Enter amount to be Deposited: "))
        self.balance += amount
        print("\n Amount Deposited:",amount)

    def withdraw(self):
        amount = float(input("Enter amount to be Withdrawn: "))
        if self.balance>=amount:
            self.balance-=amount
            print("\n You Withdrew:", amount)
        else:
            print("\n Insufficient balance  ")

    def display(self):
        print("\n Net Available Balance=",self.balance)

# Driver code

# creating an object of class
s = Bank_Account()

# Calling functions with that class object
s.deposit()
s.withdraw()
s.display()
```

```
Hello!!! Welcome to the Deposit & Withdrawal Machine
Enter amount to be Deposited: 50000.0

 Amount Deposited: 50000.0
Enter amount to be Withdrawn: 1000.0

 You Withdrew: 1000.0

 Net Available Balance= 49000.0
```

```
#Question12: Create a decorator that meassures and prints the executation time of a function.


import time

def measure_execution_time(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"Function {func.__name__} took {execution_time:.4f} seconds to execute")
        return result
    return wrapper

# Example usage
@measure_execution_time
def calculate_multiply(numbers):
    tot = 1
    for x in numbers:
        tot *= x
    return tot

# Call the decorated function
result = calculate_multiply([1, 2, 3, 4, 5])
print("Result:", result)
```

```
Function calculate_multiply took 0.0000 seconds to execute
Result: 120
```

```
#Question13: Explain the concept of the Diamond Problem in multiple inheritance. How does python resolve it
```

* Diamond Probem

It refers to an ambiguity that arises when two classes Class2 and Class3 inherit from a superclass Class1 and class Class4 inherits from both Class2 and Class3. If there is a method “m” which is an overridden method in one of Class2 and Class3 or both then the ambiguity arises which of the method “m” Class4 should inherit

```
# Python Program to depict multiple inheritance
# when method is overridden in both classes

class Class1:
    def m(self):
        print("In Class1")

class Class2(Class1):
    def m(self):
        print("In Class2")

class Class3(Class1):
    def m(self):
        print("In Class3")

class Class4(Class2, Class3):
    pass


obj = Class4()
obj.m()
```

```
In Class2
```

```
obj = Class4()
obj.m()

Class2.m(obj)
Class3.m(obj)
Class1.m(obj)
```

```
In Class2
In Class2
In Class3
In Class1
```

- To address this issue, Python provides the super() function. At a basic level, super() is used to call methods in a superclass. In multiple inheritance scenarios, it ensures that the method in the base class gets executed only once.

```
#Question14: Write a class method that keeps track of the number of instances created from a class.


class pwskills:

    # this is used to print the number
    # of instances of a class
    counter = 0

    # constructor of pwskills class
    def __init__(self):

        # increment
        pwskills.counter += 1


# object or instance of pwskills class
g1 = pwskills()
g2 = pwskills()
g3 = pwskills()
print(pwskills.counter)
```

```
3
```

```
#Question15: Implement a static method in a class that checks if a given year is a leap year.


def check_year(year):
    # Leap year condition:
    # 1. If the year is divisible by 4 and not divisible by 100, or
    # 2. If the year is divisible by 400.
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        return True  # It's a leap year
    else:
        return False  # It's not a leap year

year = 2000  # Sample input to test

if check_year(year):
    print("Leap Year")
else:
    print("Not a Leap Year")
```

```
Leap Year
```