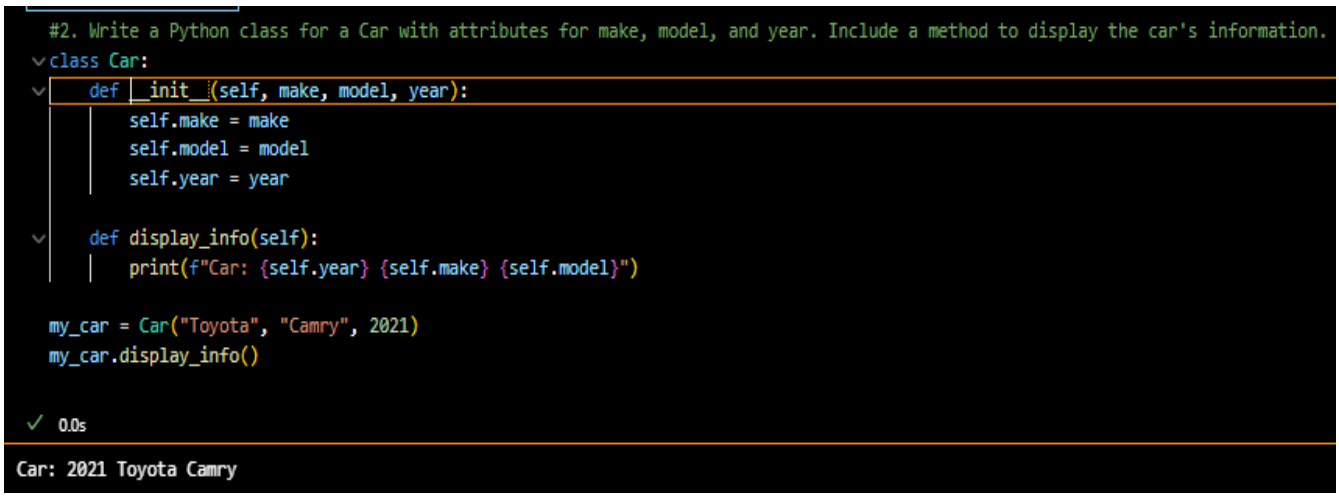
PYTHON ASSIGNMENT 4

GitHub : GitHub/Bha411

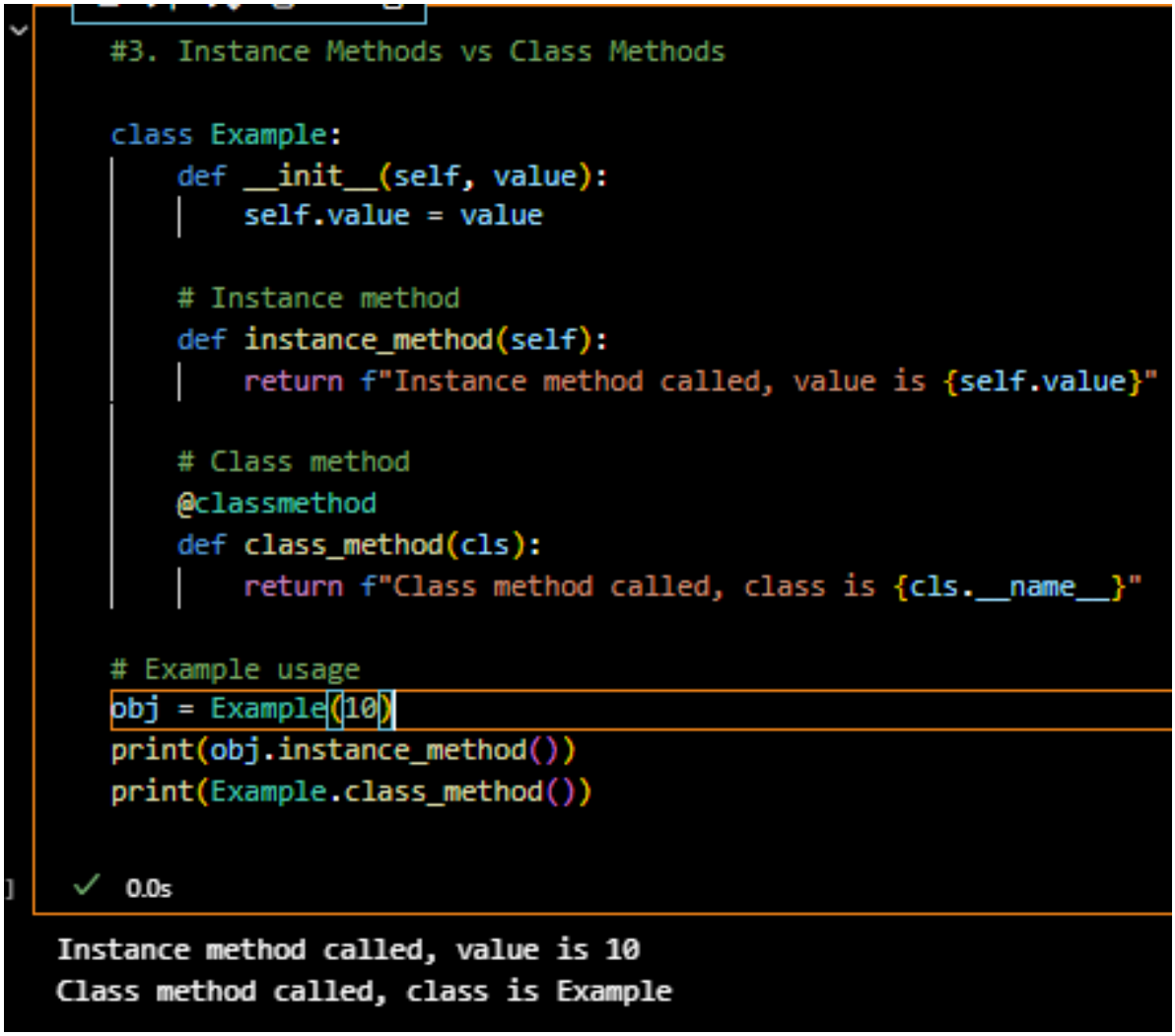
1. What are the five key concepts of Object-Oriented Programming (OOP)?

    - Encapsulation: Bundling of data (attributes) and methods (functions) that operate on the data into a single unit or class.
    - Abstraction: Hiding complex implementation details and exposing only essential features to the user.
    - Inheritance: Allowing a new class to inherit properties and behaviors (methods) from an existing class.
    - Polymorphism: Ability to define methods that behave differently based on the object that calls them.
    - Object: Instances of a class that contain attributes and methods representing real-world entities.

2. Write a Python class for a Car with attributes for make, model, and year. Include a method to display the car's information.

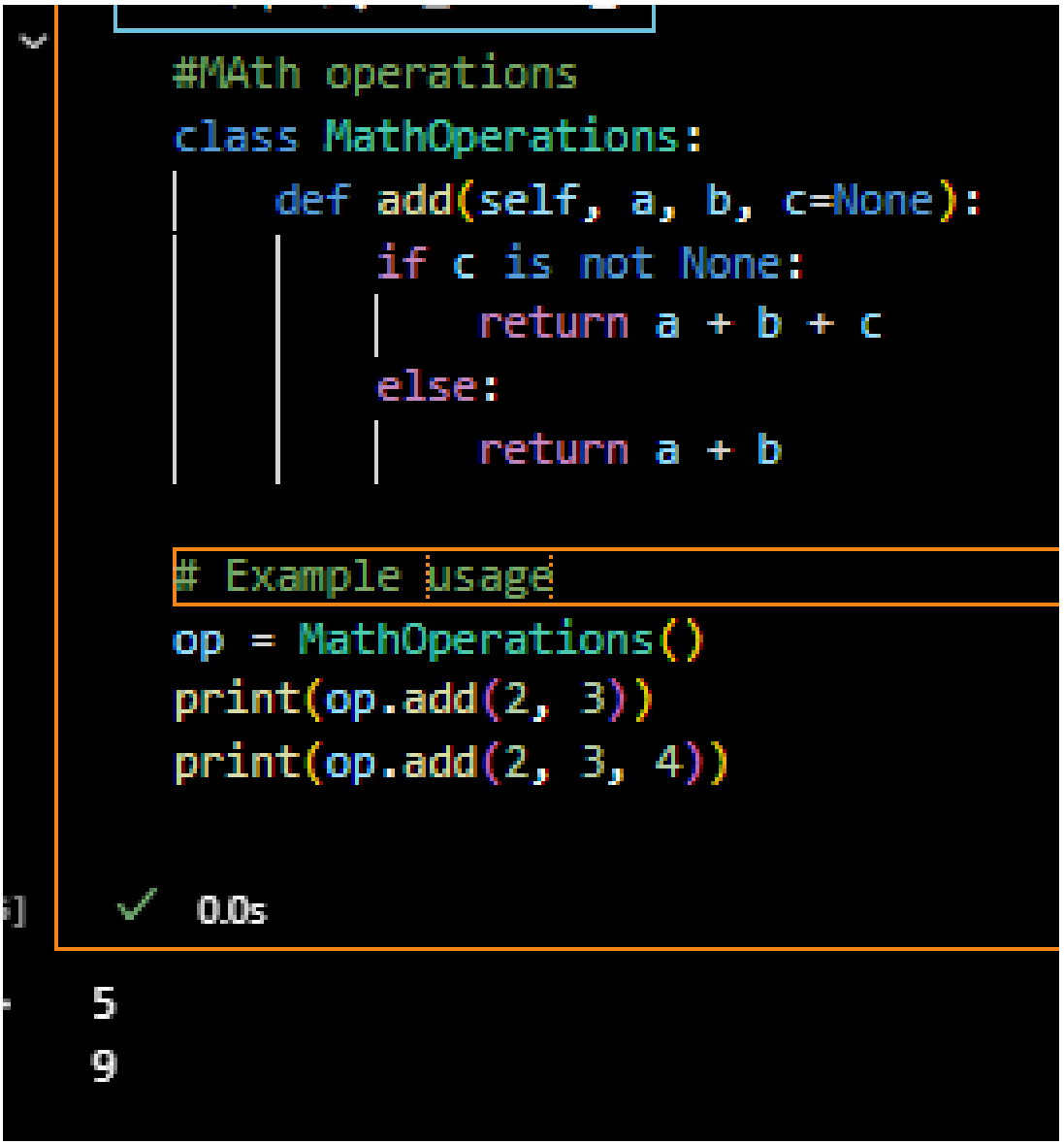


3. Explain the difference between instance methods and class methods. Provide an example of each.
    - Instance methods: Operate on an instance of the class and can access or modify instance attributes. They require self as the first parameter.
    - Class methods: Operate on the class itself rather than instances. They are defined using @classmethod and take cls as the first parameter.



4. How does Python implement method overloading? Give an example.

Python does not support traditional method overloading (multiple methods with the same name but different signatures). Instead, you can achieve similar behavior using default arguments or by checking argument types within a method.



5. What are the three types of access modifiers in Python? How are they denoted?

   Public: Accessible from anywhere. Variables or methods with no underscore (_) are public.

   Protected: Shouldn't be accessed directly outside the class, but still possible. Denoted by a single underscore (_).

   Private: Not accessible outside the class. Denoted by a double underscore (__).

```
#5. Access Modiefiers
class Example:
    public_var = "I am public"
    _protected_var = "I am protected"
    __private_var = "I am private"

    def show_vars(self):
        return self.public_var, self._protected_var, self.__private_var

# Example usage
obj = Example()
print(obj.public_var)
print(obj._protected_var)
print(obj.__private_var)
```

[7] ⊗ 0.0s

```
I am public
I am protected

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[7], line 14
     12 print(obj.public_var)
     13 print(obj._protected_var)
---> 14 print(obj.__private_var)

AttributeError: 'Example' object has no attribute '__private_var'
```

6. Describe the five types of inheritance in Python. Provide a simple example of multiple inheritance.

Single Inheritance: One class inherits from one parent class.

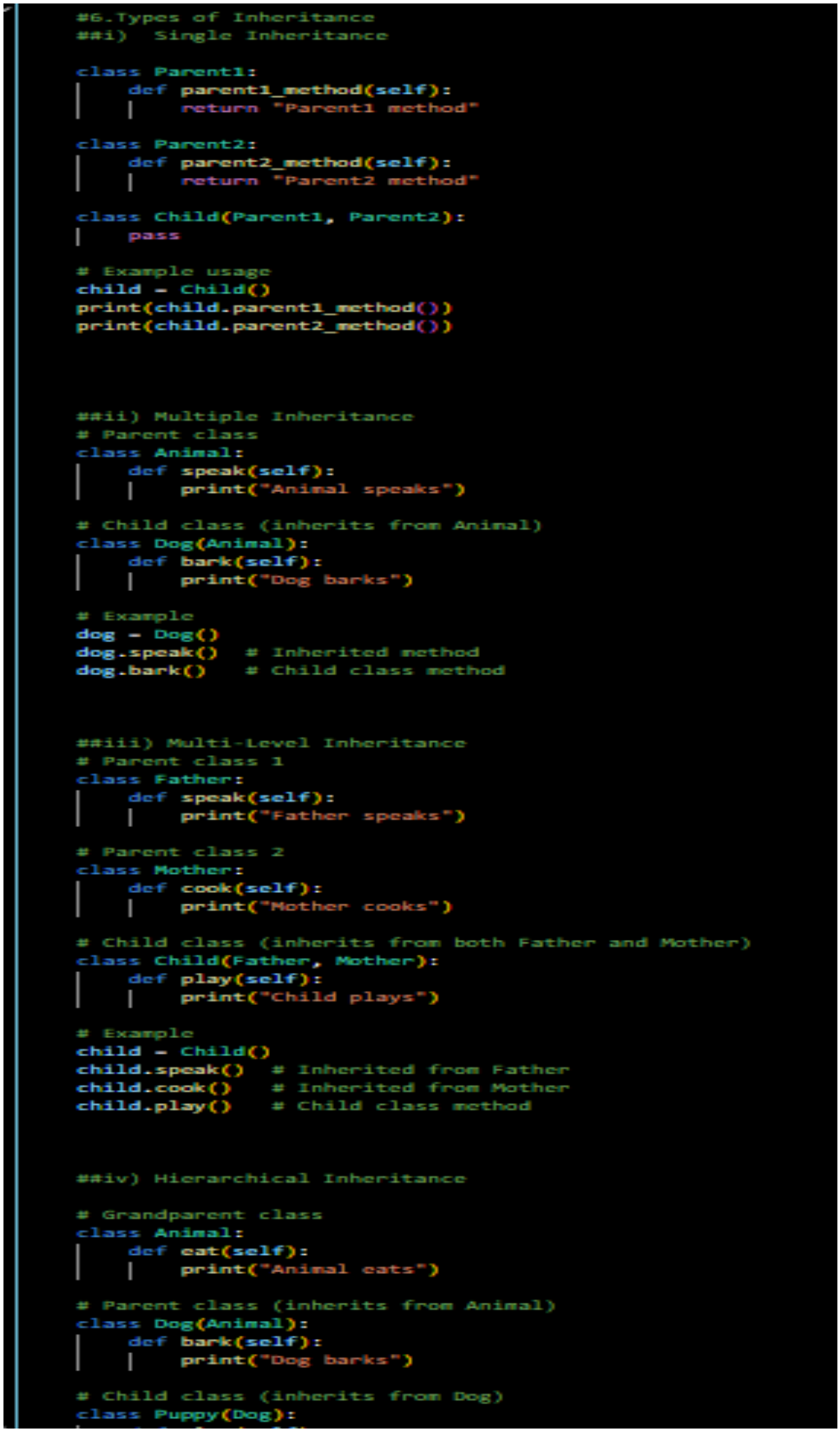
Multiple Inheritance: One class inherits from more than one parent class.

Multilevel Inheritance: A class is derived from a class that is also derived from another class.

Hierarchical Inheritance: Multiple classes inherit from the same parent class.

Hybrid Inheritance: A combination of two or more types of inheritance.

Multiple Inheritance Example:



```python
#6.Types of Inheritance
##i)  Single Inheritance

class Parent1:
    def parent1_method(self):
        return "Parent1 method"

class Parent2:
    def parent2_method(self):
        return "Parent2 method"

class Child(Parent1, Parent2):
    pass

# Example usage
child = Child()
print(child.parent1_method())
print(child.parent2_method())



##ii) Multiple Inheritance
# Parent class
class Animal:
    def speak(self):
        print("Animal speaks")

# Child class (inherits from Animal)
class Dog(Animal):
    def bark(self):
        print("Dog barks")

# Example
dog = Dog()
dog.speak()  # Inherited method
dog.bark()   # Child class method



##iii) Multi-Level Inheritance
# Parent class 1
class Father:
    def speak(self):
        print("Father speaks")

# Parent class 2
class Mother:
    def cook(self):
        print("Mother cooks")

# Child class (inherits from both Father and Mother)
class Child(Father, Mother):
    def play(self):
        print("Child plays")

# Example
child = Child()
child.speak()  # Inherited from Father
child.cook()   # Inherited from Mother
child.play()   # Child class method



##iv) Hierarchical Inheritance

# Grandparent class
class Animal:
    def eat(self):
        print("Animal eats")

# Parent class (inherits from Animal)
class Dog(Animal):
    def bark(self):
        print("Dog barks")

# Child class (inherits from Dog)
class Puppy(Dog):
```

*** Rest in the Jupyter Notebook Uploaded on GITHUB

7. What is the Method Resolution Order (MRO) in Python? How can you retrieve it programmatically?

   The Method Resolution Order (MRO) is the order in which Python looks for a method in a class hierarchy during multiple inheritance. It follows the C3 Linearization algorithm.

   We can retrieve it using the __mro__ attribute or mro() method:

```
#7. Method resolution Order
class A: pass
class B(A): pass
class C(B): pass

print(C.__mro__)
```

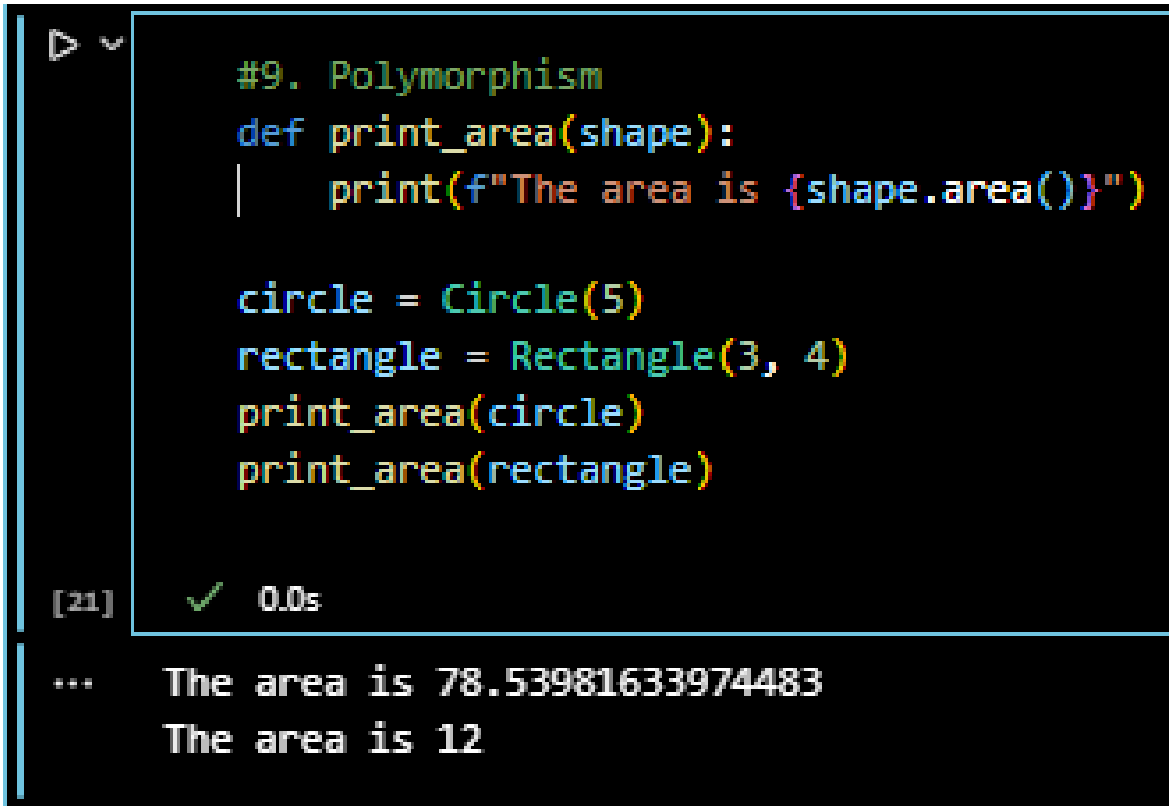
```
7]  ✓ 0.0s
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

8. Create an abstract base class Shape with an abstract method area(). Then create two subclasses Circle and Rectangle that implement the area() method.

```
#8. Abstract base class Shape.
from abc import ABC, abstractmethod
import math

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

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

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

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

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

circle = Circle(5)
rectangle = Rectangle(3, 4)
print(circle.area())
print(rectangle.area())
```

```
[19]  ✓ 0.0s
...  78.53981633974483
     12
```

9. Demonstrate polymorphism by creating a function that can work with different shape objects to calculate and print their areas.

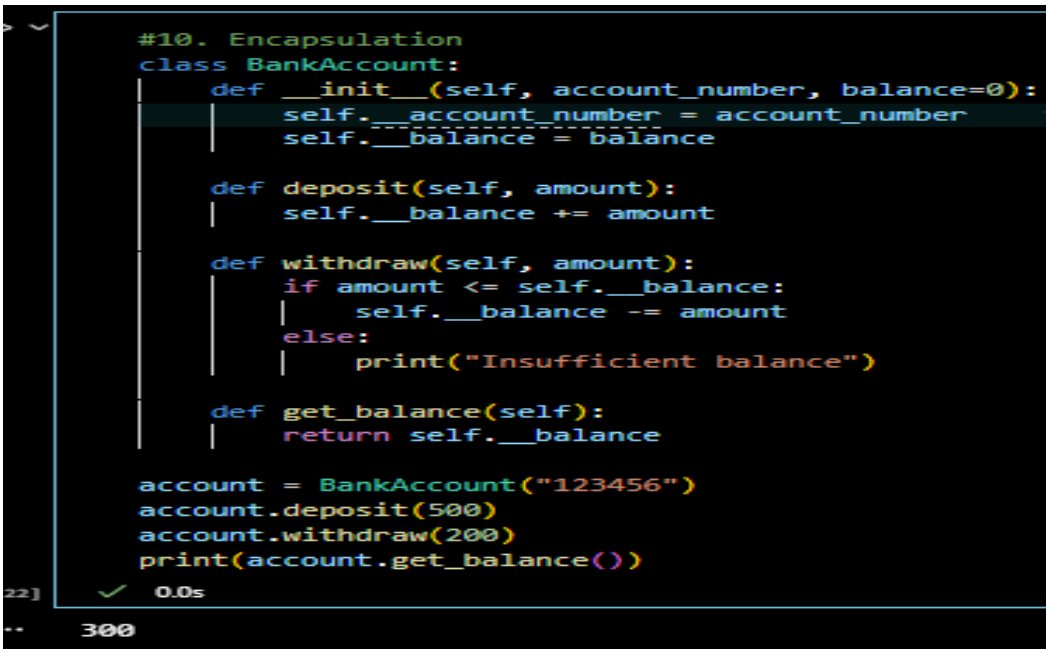


```
#9. Polymorphism
def print_area(shape):
    print(f"The area is {shape.area()}")

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

[21] ✓ 0.0s

```
The area is 78.53981633974483
The area is 12
```

10. Implement encapsulation in a BankAccount class with private attributes for balance and account_number. Include methods for deposit, withdrawal, and balance inquiry.

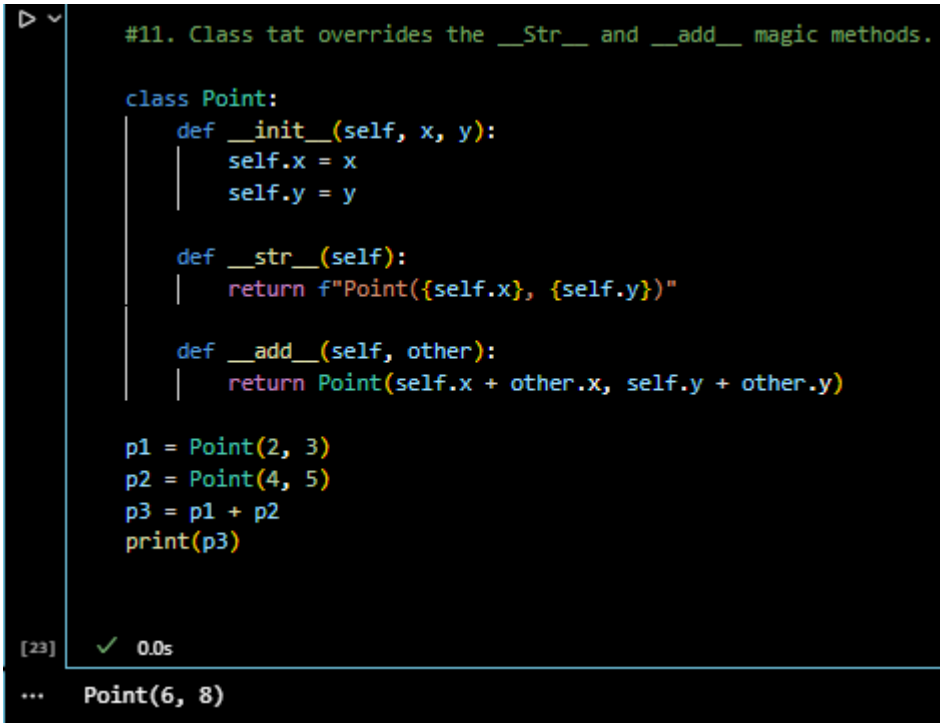


```
#10. Encapsulation
class BankAccount:
    def __init__(self, account_number, balance=0):
        self.__account_number = account_number
        self.__balance = balance

    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

account = BankAccount("123456")
account.deposit(500)
account.withdraw(200)
print(account.get_balance())
```

22] ✓ 0.0s

```
300
```

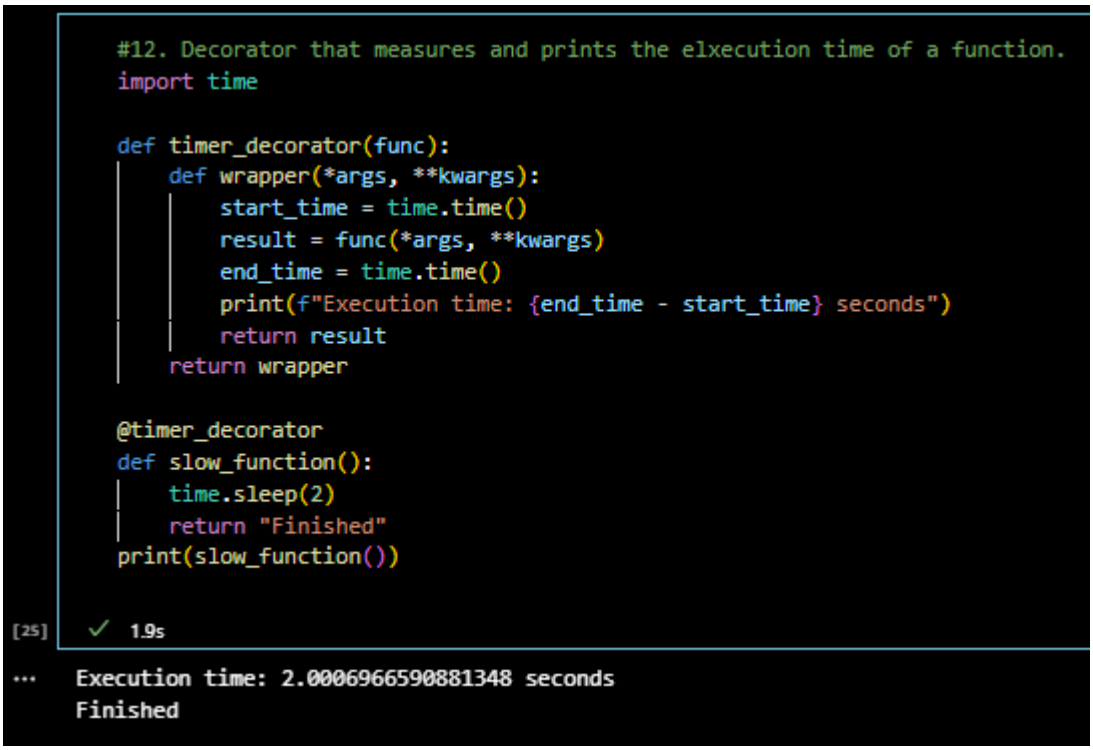
11. Write a class that overrides the __str__ and __add__ magic methods. What will these methods allow you to do?



```python
#11. Class tat overrides the __Str__ and __add__ magic methods.

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

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

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

p1 = Point(2, 3)
p2 = Point(4, 5)
p3 = p1 + p2
print(p3)
```

```
[23] ✓ 0.0s
...  Point(6, 8)
```

12. Create a decorator that measures and prints the execution time of a function.



```python
#12. Decorator that measures and prints the elxecution time of a function.
import time

def timer_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Execution time: {end_time - start_time} seconds")
        return result
    return wrapper

@timer_decorator
def slow_function():
    time.sleep(2)
    return "Finished"
print(slow_function())
```

```
[25] ✓ 1.9s
...  Execution time: 2.0006966590881348 seconds
     Finished
```

13. Explain the concept of the Diamond Problem in multiple inheritance. How does Python resolve it?

The Diamond Problem occurs in multiple inheritance when a class inherits from two classes that have a common ancestor. It creates ambiguity about which path the child class should take to inherit a method or attribute from the common ancestor.

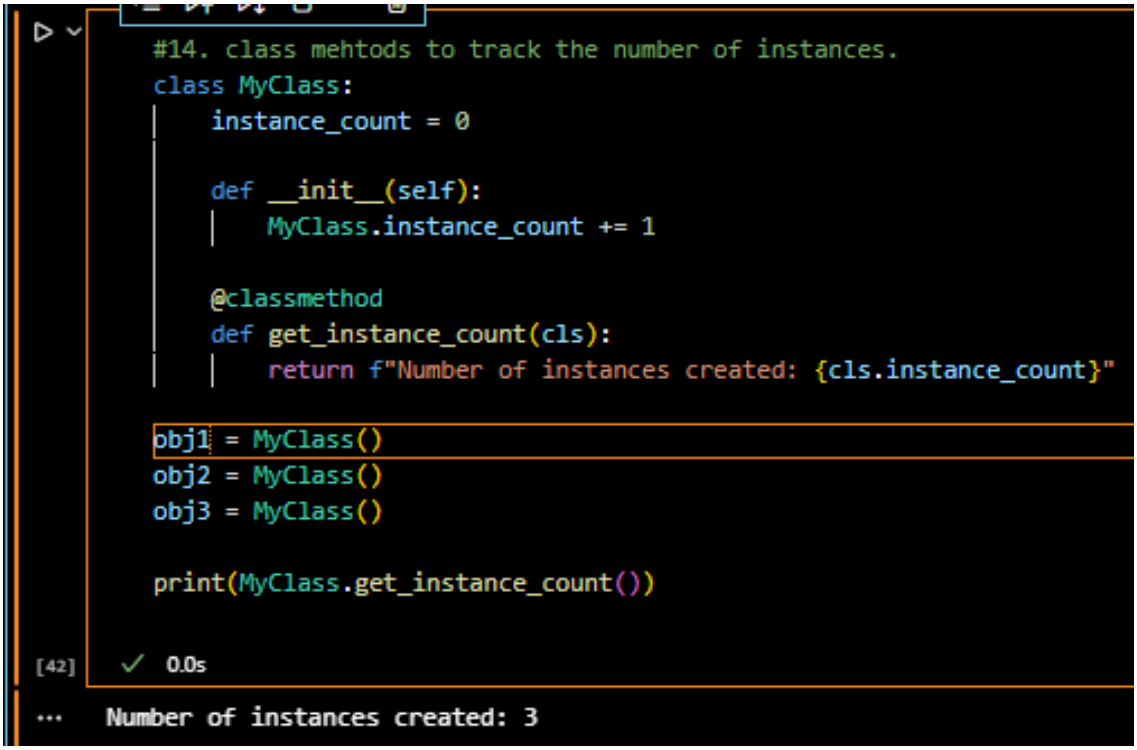
Python resolves the Diamond Problem using the Method Resolution Order (MRO), which ensures a consistent and clear inheritance path. MRO determines the order in which classes are searched when executing a method. Python uses the C3 linearization algorithm to compute the MRO.

```
class A:
    def display(self):
        print("Class A")

class B(A):
    def display(self):
        print("Class B")

class C(A):
    def display(self):
        print("Class C")

class D(B, C):
    pass

a = A()
b = B()
d = D()
c = C()
a.display()
b.display()
c.display()
d.display()
```

[41] ✓ 0.0s

```
Class A
Class B
Class C
Class B
```

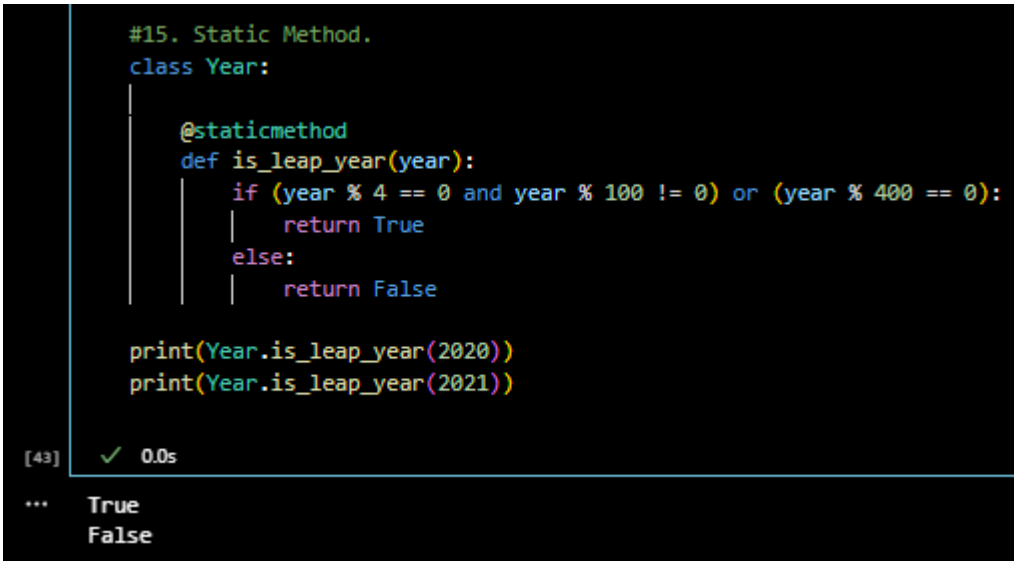
14. Write a class method that keeps track of the number of instances created from a class.



```
#14. class mehtods to track the number of instances.
class MyClass:
    instance_count = 0

    def __init__(self):
        MyClass.instance_count += 1

    @classmethod
    def get_instance_count(cls):
        return f"Number of instances created: {cls.instance_count}"

obj1 = MyClass()
obj2 = MyClass()
obj3 = MyClass()

print(MyClass.get_instance_count())
```

```
[42] ✓ 0.0s
...  Number of instances created: 3
```

15. Implement a static method in a class that checks if a given year is a leap year.



```
#15. Static Method.
class Year:

    @staticmethod
    def is_leap_year(year):
        if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
            return True
        else:
            return False

print(Year.is_leap_year(2020))
print(Year.is_leap_year(2021))
```

```
[43] ✓ 0.0s
...  True
     False
```