Name – Gathram Kesava Murthy

Registered mail id- kesavagathram.murthy@outlook.com

Course Name – Master Generative AI: Data Science Course

Assignment Name –OOPS

Submission Date – 30th October 2024

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

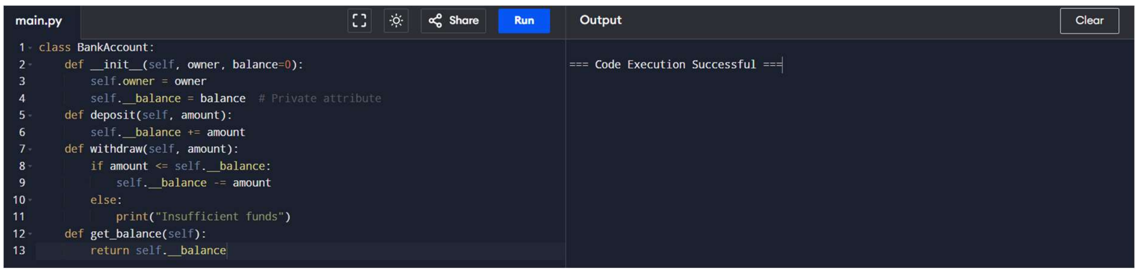
The five key concepts of Object-Oriented Programming (OOP) are:

1. Encapsulation

Definition: Encapsulation is the bundling of data (attributes) and methods (functions) that manipulate that data into a single unit called a class. This principle restricts direct access to some of an object's components, which helps prevent unauthorized access and modifications.

Key Aspects:

- **Data Hiding:** Encapsulation allows certain data to be hidden from outside interference and misuse. This is often achieved by declaring attributes as private (or protected) and providing public methods (getters and setters) to access or modify those attributes.




```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.__balance = balance  # Private attribute
    def deposit(self, amount):
        self.__balance += amount
    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient funds")
    def get_balance(self):
        return self.__balance
```

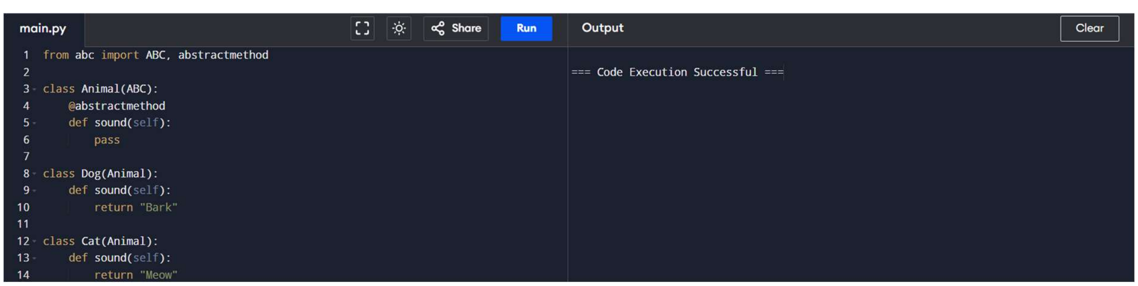
2. Abstraction

Definition: Abstraction is the process of simplifying complex systems by modeling classes based on the essential properties and behaviors an object should have, while ignoring the irrelevant details.

Key Aspects:

- **High-Level Interfaces**: Abstraction allows users to interact with objects through high-level interfaces. For instance, when you drive a car, you use the steering wheel, pedals, and gear shift without needing to understand the engine's inner workings.

- **Abstract Classes and Interfaces**: In many OOP languages, you can define abstract classes or interfaces that specify methods that must be implemented by derived classes. This enforces a contract for subclasses, ensuring they adhere to certain functionalities.




```
from abc import ABC, abstractmethod

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

class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"
```

3. Inheritance

Definition: Inheritance is a mechanism where a new class (subclass or derived class) inherits attributes and methods from an existing class (superclass or base class). This promotes code reuse and establishes a hierarchical relationship between classes.

Key Aspects:

- **Reusability**: By inheriting from existing classes, you can reuse code, reducing redundancy. If a method is defined in the superclass, the subclass can use it directly or override it.
- **Polymorphic Behavior**: Inheritance enables polymorphism, where a derived class can be treated as an instance of its base class. This is useful for writing more generic and reusable code.

main.py

```python
class Vehicle:
    def start(self):
        print("Vehicle started")

class Car(Vehicle):
    def drive(self):
        print("Car is driving")

my_car = Car()
my_car.start()  # Inherited method
my_car.drive()
```

Output

```
Vehicle started
Car is driving

=== Code Execution Successful ===
```

4. Polymorphism

Definition: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different underlying forms (data types).

Key Aspects:

- **Method Overriding**: Subclasses can provide specific implementations of methods defined in their superclass, allowing for behavior that is specific to the subclass.
- **Method Overloading**: In some languages, you can define multiple methods with the same name but different parameters. Python does not support method overloading in the same way, but you can achieve similar functionality using default parameters or variable-length arguments.

main.py

```python
class Shape:
    def area(self):
        pass

class Circle(Shape):
    def area(self, radius):
        return 3.14 * radius * radius

class Rectangle(Shape):
    def area(self, length, width):
        return length * width

def print_area(shape, *args):
    print(f"Area: {shape.area(*args)}")

circle = Circle()
rectangle = Rectangle()
print_area(circle, 5)  # Area of the circle
print_area(rectangle, 4, 5)  # Area of the rectangle
```

Output

```
Area: 78.5
Area: 20

=== Code Execution Successful ===
```

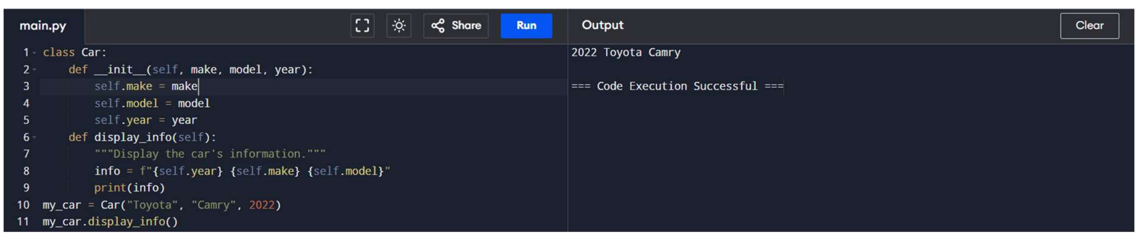
5. Composition

Definition: Composition is a design principle where complex objects are built from simpler objects. It emphasizes creating relationships between classes through the inclusion of instances of other classes.

Key Aspects:

- **Has-A Relationship**: In composition, an object contains references to other objects. This contrasts with inheritance, which establishes an is-a relationship.
- **Flexibility**: Composition allows you to change the behavior of a class at runtime by altering its components, promoting more flexible and maintainable code.

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.

In Python, instance methods and class methods are two types of methods that serve different purposes and have different access levels to the class and its attributes. Here's a breakdown of their differences, along with examples of each:

Instance Methods

- **Definition**: Instance methods are functions defined inside a class that operate on an instance of that class (an object). They can access and modify instance attributes and are called on instances of the class.
- **First Parameter**: The first parameter of an instance method is typically named self, which refers to the instance invoking the method.
- **Usage**: Instance methods are used when you need to access or modify the state of an instance.

Example of an Instance Method

main.py

```python
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
    def display_info(self):
        """Display the car's information."""
        return f"{self.year} {self.make} {self.model}"
my_car = Car("Toyota", "Camry", 2022)
print(my_car.display_info())
```

Output

```
2022 Toyota Camry

=== Code Execution Successful ===
```

Class Methods

- **Definition**: Class methods are functions defined within a class that are bound to the class itself rather than instances of the class. They can be called on the class itself, and they are not dependent on instance attributes.
- **First Parameter**: The first parameter of a class method is conventionally named cls, which refers to the class itself.
- **Usage**: Class methods are used when you need to perform an action that is relevant to the class as a whole rather than a specific instance. They can also be used as factory methods to create instances of the class.
- **Decorator**: Class methods are defined using the @classmethod decorator.

Example of a Class Method

main.py

```python
class Car:
    total_cars = 0
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        Car.total_cars += 1
    @classmethod
    def get_total_cars(cls):
        return cls.total_cars
car1 = Car("Toyota", "Camry", 2022)
car2 = Car("Honda", "Civic", 2023)
print(Car.get_total_cars())
```

Output

```
2

=== Code Execution Successful ===
```

Differences:

Feature	Instance Method	Class Method
Definition	Operates on an instance of the class	Operates on the class itself
First Parameter	`self` (reference to the instance)	`cls` (reference to the class)
Access to Attributes	Can access and modify instance attributes	Can access class attributes, not instance attributes
Invocation	Called on an instance (e.g., `obj.method()`)	Called on the class (e.g., `Class.method()`)
Use Case	When instance-specific behavior is needed	When class-wide behavior is needed or when creating instances

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

Python does not support traditional method overloading in the same way as some other programming languages (like Java or C++). In those languages, you can define multiple methods with the same name but different parameter lists. However, Python allows you to achieve similar behavior through default parameters, variable-length arguments, or by using conditional statements within a single method.

Implementing Method Overloading in Python

1. **Default Parameters**: You can provide default values for parameters, allowing the method to be called with different numbers of arguments.
2. **Variable-Length Arguments**: You can use *args and **kwargs to accept a variable number of positional and keyword arguments.
3. **Conditional Logic**: You can implement logic within a single method to handle different types or numbers of parameters.

Example of Method Overloading using Default Parameters

Here’s an example that demonstrates method overloading using default parameters:

main.py

```
class Calculator:
    def add(self, a, b=0, c=0):
        return a + b + c
calc = Calculator()
result1 = calc.add(5)
result2 = calc.add(5, 10)
result3 = calc.add(5, 10, 15)
print(result1)
print(result2)
print(result3)
```

Output

```
5
15
30

=== Code Execution Successful ===
```

Example of Method Overloading using Variable-Length Arguments

Here's another example that uses *args to accept a variable number of arguments:

main.py

```
class Multiplier:
    def multiply(self, *args):
        result = 1
        for num in args:
            result *= num
        return result
mult = Multiplier()
result1 = mult.multiply(2)
result2 = mult.multiply(2, 3)
result3 = mult.multiply(2, 3, 4, 5)
print(result1)
print(result2)
print(result3)
```

Output

```
2
6
120

=== Code Execution Successful ===
```

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

In Python, access modifiers are used to define the visibility and accessibility of class attributes and methods. There are three main types of access modifiers:

1. Public Access Modifier

- **Definition**: Public members (attributes and methods) are accessible from anywhere in the code. There are no restrictions on accessing these members.
- **Denotation**: Public members are defined without any special prefix.

Example:

main.py

```python
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model
    def display_info(self):
        return f"{self.make} {self.model}"
my_car = Car("Toyota", "Camry")
print(my_car.display_info())
```

Output

```
Toyota Camry

=== Code Execution Successful ===
```

2. Protected Access Modifier

- **Definition**: Protected members are intended to be accessible within the class and by subclasses. They are not meant to be accessed directly from outside the class.
- **Denotation**: Protected members are defined with a single underscore prefix (_).

Example:

main.py

```python
class Car:
    def __init__(self, make, model):
        self._make = make
        self._model = model
    def _display_info(self):
        return f"{self._make} {self._model}"
class ElectricCar(Car):
    def display_info(self):
        return self._display_info()
my_electric_car = ElectricCar("Tesla", "Model S")
print(my_electric_car.display_info())
```

Output

```
Tesla Model S

=== Code Execution Successful ===
```

3. Private Access Modifier

- **Definition**: Private members are intended to be accessible only within the class itself. They cannot be accessed directly from outside the class, including subclasses.
- **Denotation**: Private members are defined with a double underscore prefix (__).

Example:

main.py

```python
class Car:
    def __init__(self, make, model):
        self.__make = make
        self.__model = model
    def __display_info(self):
        return f"{self.__make} {self.__model}"
    def public_info(self):
        return self.__display_info()
my_car = Car("Honda", "Civic")
print(my_car.public_info())
```

Output

```
Honda Civic

=== Code Execution Successful ===
```

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

In Python, inheritance allows a class (derived or child class) to inherit attributes and methods from another class (base or parent class). There are five main types of inheritance in Python:

1. Single Inheritance

In single inheritance, a derived class inherits from only one base class. This is the simplest form of inheritance.

Example:

```python
class Animal:
    def speak(self):
        return "Animal speaks"
class Dog(Animal):
    def bark(self):
        return "Dog barks"
dog = Dog()
print(dog.speak())
print(dog.bark())
```

Output

```
Animal speaks
Dog barks

=== Code Execution Successful ===
```

2. Multiple Inheritance

In multiple inheritance, a derived class can inherit from more than one base class. This allows the derived class to combine functionalities from multiple sources.

Example:

```python
class Parent1:
    def method1(self):
        return "Method from Parent1"
class Parent2:
    def method2(self):
        return "Method from Parent2"
class Child(Parent1, Parent2):
    def method3(self):
        return "Method from Child"
child = Child()
print(child.method1())
print(child.method2())
print(child.method3())
```

Output

```
Method from Parent1
Method from Parent2
Method from Child

=== Code Execution Successful ===
```

3. Multilevel Inheritance

In multilevel inheritance, a class derives from another derived class, forming a chain of inheritance.

Example:

```python
class Animal:
    def speak(self):
        return "Animal speaks"
class Dog(Animal):
    def bark(self):
        return "Dog barks"
class Puppy(Dog):
    def weep(self):
        return "Puppy weeps"
puppy = Puppy()
print(puppy.speak())
print(puppy.bark())
print(puppy.weep())
```

Output

```
Animal speaks
Dog barks
Puppy weeps

=== Code Execution Successful ===
```

4. Hierarchical Inheritance

In hierarchical inheritance, multiple derived classes inherit from a single base class. This allows different derived classes to share common functionality.

Example:

```python
class Animal:
    def speak(self):
        return "Animal speaks"
class Dog(Animal):
    def bark(self):
        return "Dog barks"
class Cat(Animal):
    def meow(self):
        return "Cat meows"
dog = Dog()
cat = Cat()
print(dog.speak())
print(cat.speak())
print(dog.bark())
print(cat.meow())
```

Output

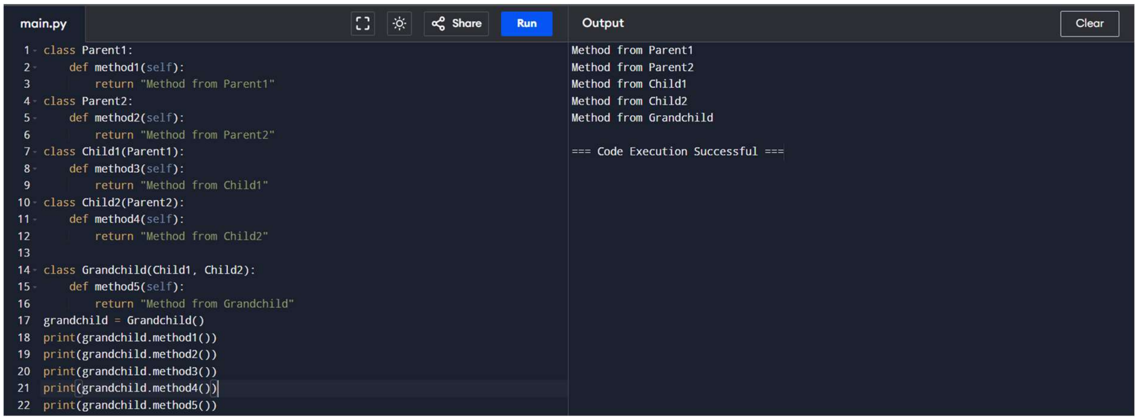
```
Animal speaks
Animal speaks
Dog barks
Cat meows

=== Code Execution Successful ===
```

5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance. This is usually a complex structure where multiple inheritance and other types coexist.

Example:



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

Method Resolution Order (MRO) in Python:

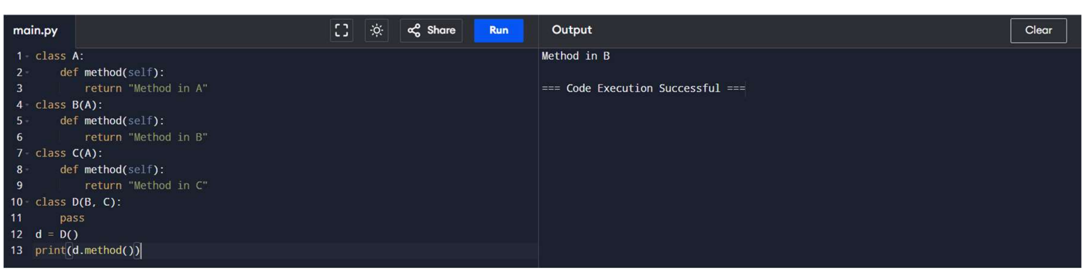
Method Resolution Order (MRO) is the order in which Python looks for methods and attributes in a hierarchy of classes. This becomes especially important in the context of multiple inheritance, where a class can inherit from more than one parent class. The MRO ensures that Python can unambiguously determine which method to invoke when a method is called on an instance of a class.

Python uses the C3 linearization algorithm (or C3 superclass linearization) to compute the MRO, which maintains a consistent order that respects the inheritance hierarchy.

MRO Rules:

1. **Depth-First Search**: Python searches for a method in a depth-first manner, starting from the class itself, and then moving to its parent classes.
2. **Left-to-Right**: If multiple base classes are present, they are considered in the order they are listed in the class definition.
3. **Consistency**: The order of base classes is preserved to avoid confusion and ensure predictable behavior.

Example of MRO:



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

main.py

```python
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)
print(f"Area of Circle: {circle.area():.2f}")
rectangle = Rectangle(4, 6)
print(f"Area of Rectangle: {rectangle.area()}")
```

Output

```
Area of Circle: 78.54
Area of Rectangle: 24

=== Code Execution Successful ===
```

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

Polymorphism in Python allows objects of different classes to be treated as objects of a common superclass. In the context of shapes, this means you can create a function that takes different shape objects and calculates their areas without needing to know their specific types.

Here is how you can demonstrate polymorphism by creating a function that works with Circle and Rectangle objects to calculate and print their areas:

Implementation:

main.py

```python
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
def print_area(shape: Shape):
    print(f"The area of the shape is: {shape.area():.2f}")
circle = Circle(5)
rectangle = Rectangle(4, 6)
print_area(circle)
print_area(rectangle)
```

Output

```
The area of the shape is: 78.54
The area of the shape is: 24.00

=== Code Execution Successful ===
```

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

main.py

```python
class BankAccount:
    def __init__(self, account_number, initial_balance=0):
        self.__account_number = account_number
        self.__balance = initial_balance
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited: ${amount:.2f}. New balance: ${self.__balance:.2f}")
        else:
            print("Deposit amount must be positive.")
    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew: ${amount:.2f}. New balance: ${self.__balance:.2f}")
        else:
            print("Withdrawal amount must be positive and less than or equal to the balance.")
    def get_balance(self):
        return self.__balance
    def get_account_number(self):
        return self.__account_number
account = BankAccount("123456789", 1000)
account.deposit(500)
account.withdraw(200)
print(f"Current balance: ${account.get_balance():.2f}")
account.withdraw(1500)
```

Output

```
Deposited: $500.00. New balance: $1500.00
Withdrew: $200.00. New balance: $1300.00
Current balance: $1300.00
Withdrawal amount must be positive and less than or equal to the balance.

=== Code Execution Successful ===
```

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

In Python, magic methods (also known as dunder methods, for "double underscore") are special methods that allow you to define the behavior of your objects for built-in functions and operators. The __str__ method is used to define how an object should be represented as a string, while the __add__ method allows you to define how two objects of a class can be added together using the + operator.

Here is an example of a class that overrides both the __str__ and __add__ magic methods:

Implementation of a Vector Class:

main.py

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return f"Vector({self.x}, {self.y})"
    def __add__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        return NotImplemented
v1 = Vector(2, 3)
v2 = Vector(4, 5)
print(v1)
print(v2)
v3 = v1 + v2
print(v3)
```

Output

```
Vector(2, 3)
Vector(4, 5)
Vector(6, 8)

=== Code Execution Successful ===
```

Explanation

1. **Class Definition**:
    - The Vector class represents a 2D vector with x and y coordinates.
2. **Constructor (__init__)**:
    - Initializes the vector with x and y values.
3. **String Representation (__str__)**:

- The __str__ method returns a human-readable string representation of the Vector object, allowing you to customize how the object is displayed when printed or converted to a string. For instance, calling print(v1) outputs Vector(2, 3).

4. **Addition (__add__):**
    - The __add__ method is used to define the behavior of the + operator. It allows you to add two Vector objects together by returning a new Vector whose x and y values are the sum of the corresponding values of the two vectors. If the other object is not a Vector, it returns NotImplemented, which allows Python to handle the error gracefully.

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

main.py

```python
import time
def timeit(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"Execution time of '{func.__name__}': {execution_time:.6f} seconds
        return result
    return wrapper
@timeit
def compute_sum(n):
    total = sum(range(n + 1))
    return total
@timeit
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
result_sum = compute_sum(1000000)
print(f"Sum: {result_sum}")
result_fact = factorial(10)
print(f"Factorial: {result_fact}")
```

Output

```
Execution time of 'compute_sum': 0.012131 seconds
Sum: 500000500000
Execution time of 'factorial': 0.000000 seconds
Execution time of 'factorial': 0.000008 seconds
Execution time of 'factorial': 0.000020 seconds
Execution time of 'factorial': 0.000026 seconds
Execution time of 'factorial': 0.000030 seconds
Execution time of 'factorial': 0.000037 seconds
Execution time of 'factorial': 0.000042 seconds
Execution time of 'factorial': 0.000046 seconds
Execution time of 'factorial': 0.000050 seconds
Execution time of 'factorial': 0.000058 seconds
Execution time of 'factorial': 0.000063 seconds
Factorial: 3628800

=== Code Execution Successful ===
```

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

The Diamond Problem, also known as the Deadly Diamond of Death, is a classic issue in object-oriented programming that arises when a class inherits from two or more classes that have a common ancestor. This situation creates ambiguity regarding which method or attribute to inherit from the shared ancestor class.

Illustration of the Diamond Problem

To understand the Diamond Problem, consider the following class hierarchy:

```
   A
  / \
 B   C
  \ /
   D
```

- **Class A** is the base class.
- **Class B** and **Class C** both inherit from **Class A**.

- **Class D** inherits from both **Class B** and **Class C**.

If **Class A** has a method (let's say method_a()), when **Class D** calls method_a(), it can be ambiguous whether it should use the implementation from **Class B** or **Class C**.

How Python Resolves the Diamond Problem

Python uses the **C3 linearization algorithm** (or C3 superclass linearization) to resolve the Diamond Problem. This method provides a clear order of method resolution, ensuring that:

1. **Depth-First Search**: It performs a depth-first left-to-right search for the method.
2. **Preserving Order**: It respects the order of base classes as specified in the class definition.
3. **Single Inheritance**: It avoids duplicate calls by ensuring that each class appears only once in the method resolution order.

Example of the Diamond Problem in Python

main.py

```python
class A:
    def method(self):
        print("Method from A")
class B(A):
    def method(self):
        print("Method from B")
class C(A):
    def method(self):
        print("Method from C")
class D(B, C):
    pass
d = D()
d.method()
print(D.__mro__)
```

Output

```
Method from B
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'
    >, <class 'object'>)

=== Code Execution Successful ===
```

Explanation of the Example

1. **Class Definitions**:
   - A is the base class with a method method().
   - B and C both inherit from A and override the method().
   - D inherits from both B and C.
2. **Method Call**:
   - When you call d.method(), Python looks for method() in D, then B, and finds it there first. Therefore, it prints "Method from B".
3. **Method Resolution Order (MRO)**:
   - You can see the MRO of D using D.__mro__, which shows the order in which Python resolves method calls:
     - First, it checks D
     - Then B
     - Then C
     - Finally A

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

You can keep track of the number of instances created from a class in Python using a class variable along with a class method to manage this count. Below is an implementation of a class called InstanceCounter that keeps track of how many instances have been created.

Implementation of the InstanceCounter Class:

main.py

```python
class InstanceCounter:
    instance_count = 0
    def __init__(self):
        InstanceCounter.instance_count += 1
    @classmethod
    def get_instance_count(cls):
        return cls.instance_count
obj1 = InstanceCounter()
obj2 = InstanceCounter()
obj3 = InstanceCounter()
print(f"Number of instances created: {InstanceCounter.get_instance_count()}")
```

Output

```
Number of instances created: 3

=== Code Execution Successful ===
```

Explanation

1. **Class Variable (instance_count)**:
   - This variable is defined at the class level and is shared among all instances of the class. It keeps track of how many instances have been created.
2. **Constructor (__init__)**:
   - Each time an instance of InstanceCounter is created, the constructor increments the instance_count by 1.
3. **Class Method (get_instance_count)**:
   - This method is defined with the @classmethod decorator, which allows it to access the class variable. It returns the current count of instances created.

Example Usage

- When instances obj1, obj2, and obj3 are created, the instance_count is incremented accordingly.
- Finally, calling get_instance_count() returns the total number of instances created, which is printed to the console.

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

A static method in a class can be used to perform a function that doesn't require access to instance or class variables. In this case, you can create a static method that checks if a given year is a leap year based on the rules of the Gregorian calendar.

Implementation of a YearUtils Class:

main.py

```python
class YearUtils:
    @staticmethod
    def is_leap_year(year):
        if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
            return True
        return False
year = 2024
if YearUtils.is_leap_year(year):
    print(f"{year} is a leap year.")
else:
    print(f"{year} is not a leap year.")
print(YearUtils.is_leap_year(1900))
print(YearUtils.is_leap_year(2000))
print(YearUtils.is_leap_year(2023))
```

Output

```
2024 is a leap year.
False
True
False

=== Code Execution Successful ===
```

Explanation

1. **Static Method (is_leap_year)**:
   - The method is defined with the @staticmethod decorator, indicating that it does not require access to any instance or class-specific data.
   - The method checks if the given year is a leap year using the following rules:
     - A year is a leap year if it is divisible by 4.
     - However, if the year is divisible by 100, it is not a leap year unless it is also divisible by 400.
2. **Example Usage**:
   - The method is called directly on the class YearUtils without needing to create an instance.
   - The example checks if the year 2024 is a leap year and prints the result.
   - Additional checks are performed for the years 1900, 2000, and 2023, demonstrating the method's functionality.