In [ ]:

```
#                          Python OOPS Assignment Questions

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


# 2. Write a Python class for a `Car` with attributes for `make`, `model`, and `yea
# the car's information.


# 3. Explain the difference between instance methods and class methods. Provide an


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


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


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


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


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


# 9. Demonstrate polymorphism by creating a function that can work with different s
# and print their areas.


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


# 11. Write a class that overrides the `__str__` and `__add__` magic methods. What
# you to do?


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


# 13. Explain the concept of the Diamond Problem in multiple inheritance. How does


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


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

In [ ]:

```
#  Ques 1

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


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

# 1 Classes and Objects:

# Class: A blueprint or template for creating objects. It defines the properties (a
```

```
# behaviors (methods) that the objects created from the class will have.
# Object: An instance of a class. It represents a specific entity with the characte
# by the class.

# Example:
```

In [1]:
```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

car1 = Car('Toyota', 'Corolla')
```

In [ ]:
```
# 2 Encapsulation:

# Encapsulation is the concept of bundling the data (attributes) and methods that c
# into a single unit, or class. It also involves restricting access to certain deta
# by making some data or methods private.
# Encapsulation ensures data security and hides the internal workings from the outs

# Example:
```

In [2]:
```
class Car:
    def __init__(self, brand, model):
        self.__brand = brand  # Private attribute
        self.model = model

    def get_brand(self):
        return self.__brand

car1 = Car('Toyota', 'Corolla')
print(car1.get_brand())  # Accessing private attribute through a method
```

```
Toyota
```

In [ ]:
```
# 3 Inheritance:

# Inheritance allows a class (called the child or subclass) to inherit properties a
# behaviors from another class (called the parent or superclass). It promotes code

# Example:
```

In [3]:
```
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

class Car(Vehicle):  # Car class inherits from Vehicle
    def __init__(self, brand, model):
        super().__init__(brand)
        self.model = model
```

In [ ]:
```
# 4 Polymorphism:

# Polymorphism allows different classes to define methods with the same name but pe
# different behaviors. This enables a single interface to represent different types
# allowing methods to work on objects of different classes.

# Example:
```

In [4]:
```
class Animal:
    def sound(self):
        pass
```

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

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

animals = [Dog(), Cat()]
for animal in animals:
    print(animal.sound())  # Calls sound() method specific to the object
```

```
Bark
Meow
```

In [ ]:
```
# 5 Abstraction:

# Abstraction involves hiding the complex implementation details and showing only t
# parts of an object to the user. It helps in reducing complexity by using simple i
# to interact with objects.
# In Python, abstraction is achieved using abstract classes and interfaces.

# Example:
```

In [5]:
```
from abc import ABC, abstractmethod

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

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

In [6]:
```
# Ques 2

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

class Car:
    def __init__(self, make, model, year):
        """Initialize the attributes of the car."""
        self.make = make
        self.model = model
        self.year = year

    def display_info(self):
        """Display the car's information."""
        print(f"Car Information: {self.year} {self.make} {self.model}")

# Example usage
my_car = Car('Toyota', 'Corolla', 2021)
my_car.display_info()
```

```
Car Information: 2021 Toyota Corolla
```

In [ ]:
```
#  Ques 3

# Explain the difference between instance methods and class methods. Provide an exa

# In Python, the key difference between instance methods and class methods is in ho
```

```
# 1. Instance Methods:
# Definition: An instance method is a method that is called on an instance of a cla
# and modify the attributes of the specific instance (object) on which it is called

# How it's called: It’s called on an object (instance of a class), and the first pa
# self, which refers to the instance calling the method.

# Example of an Instance Method:
```

In [7]:
```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def display_info(self):
        """Instance method to display car's information."""
        print(f"Car Info: {self.year} {self.make} {self.model}")

# Create an instance of Car
my_car = Car('Toyota', 'Corolla', 2021)

# Call the instance method
my_car.display_info()
```

```
Car Info: 2021 Toyota Corolla
```

In [ ]:
```
# 2. Class Methods:

# Definition: A class method is a method that is bound to the class itself, not the
# It is defined using the @classmethod decorator, and the first parameter is cls, w
# class, not an instance.
# How it's called: It’s called on the class itself or an instance, and it can only
# class-level data, not instance-level data.

# Example of a Class Method:
```

In [8]:
```
class Car:
    total_cars = 0  # Class attribute

    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        Car.total_cars += 1  # Increment the total car count

    @classmethod
    def display_total_cars(cls):
        """Class method to display total cars."""
        print(f"Total cars: {cls.total_cars}")

# Create instances of Car
car1 = Car('Toyota', 'Corolla', 2021)
car2 = Car('Honda', 'Civic', 2020)

# Call the class method using the class
Car.display_total_cars()

# You can also call the class method using an instance
car1.display_total_cars()
```

```
Total cars: 2
Total cars: 2
```

In [ ]:
```
# Key Differences:

# Instance Method:

# Works with instance-specific data (attributes).
# Requires an instance to be called.
# First parameter is self, which refers to the instance.

# Class Method:

# Works with class-level data (attributes shared among all instances).
# Can be called on the class itself.
# First parameter is cls, which refers to the class.
```

In [ ]:
```
# # Ques 4

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

# Method Overloading in Python

# In many programming languages, method overloading allows a class to have multiple
# with the same name but different parameters. However, Python does not support tra
# overloading like some other languages (e.g., Java or C++). In Python, if you defi
# with the same name, the most recent definition will overwrite the previous ones.

# Instead, Python achieves a similar effect using:

# Default arguments
# Variable-length arguments (*args and **kwargs)

# Example Using Default Arguments

# You can simulate method overloading by using default arguments.
# This allows a method to behave differently depending on the number of arguments p
```

In [9]:
```
class MathOperations:
    def add(self, a, b=0, c=0):
        return a + b + c

# Example usage
math_op = MathOperations()

# Call with one argument
print(math_op.add(10))  # Output: 10

# Call with two arguments
print(math_op.add(10, 5))  # Output: 15

# Call with three arguments
print(math_op.add(10, 5, 2))  # Output: 17
```

```
10
15
17
```

In [ ]:
```
#  Ques 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 c
# and methods. Unlike some other languages, Python doesn't enforce strict access co
# follows a convention-based approach for access modifiers, which are essentially g
```

```
# There are three types of access modifiers in Python:

# 1. Public Access Modifier:

# Description: Public members are accessible from anywhere, both inside and outside
# By default, all class members (attributes and methods) in Python are public unles
# How it's denoted: Public members are written as regular names without any leading

# Example:
```

In [10]:
```
class Car:
    def __init__(self, make, model):
        self.make = make   # Public attribute
        self.model = model # Public attribute

    def display_info(self):
        print(f"Car: {self.make} {self.model}")

car1 = Car("Toyota", "Corolla")
print(car1.make)  # Accessing public attribute from outside the class
```

```
Toyota
```

In [ ]:
```
# 2. Protected Access Modifier:

# Description: Protected members are intended to be accessible only within the clas
# subclasses, though they can still be accessed outside the class as a convention.
# They are not strictly private, but a single underscore (_) before the attribute/m
# indicates that it is intended for internal use only.

# How it's denoted: Protected members are preceded by a single underscore (_).

# Example:
```

In [11]:
```
class Car:
    def __init__(self, make, model):
        self._make = make   # Protected attribute

class SportsCar(Car):
    def display_info(self):
        print(f"Sports Car: {self._make}")

car1 = SportsCar("Ferrari", "488")
car1.display_info()

# Though accessible, it's intended to be used within the class and subclasses
print(car1._make)  # Not recommended, but accessible
```

```
Sports Car: Ferrari
Ferrari
```

In [ ]:
```
# 3. Private Access Modifier:

# Description: Private members are meant to be accessible only within the class in
# defined. They cannot be accessed directly from outside the class. Python uses nam
# private members less accessible outside the class. Name mangling changes the name
# private attribute in a way that prevents direct access from outside.

# How it's denoted: Private members are preceded by two underscores (__).

# Example:
```

In [12]:
```
class Car:
    def __init__(self, make, model):
        self.__make = make  # Private attribute

    def display_info(self):
        print(f"Car: {self.__make}")

car1 = Car("Tesla", "Model S")
car1.display_info()

# Attempting to access the private attribute directly will raise an error
# print(car1.__make)  # This will raise an AttributeError

# However, you can still access it using name mangling (not recommended)
print(car1._Car__make)  # Output: Tesla (using name mangling)
```

```
Car: Tesla
Tesla
```

In [ ]:
```
# Summary:

# Public (self.attribute): Accessible from anywhere.

# Protected (self._attribute): Intended for use within the class and its subclasses
# (though still accessible from outside).

# Private (self.__attribute): Only accessible within the class using name mangling
# direct access from outside.
```

In [ ]:
```
# Ques 6

# Describe the five types of inheritance in Python. Provide a simple example of mul

# 1. Single Inheritance:

# In single inheritance, a class inherits from one parent class. The child class ca
# and reuse the functionality of the parent class.

# Example:
```

In [13]:
```
class Parent:
    def parent_method(self):
        print("This is a method from the parent class.")

class Child(Parent):
    def child_method(self):
        print("This is a method from the child class.")

child = Child()
child.parent_method()  # Inherited from the parent class
child.child_method()   # Defined in the child class
```

```
This is a method from the parent class.
This is a method from the child class.
```

In [15]:
```
# 2. Multiple Inheritance:

# In multiple inheritance, a class inherits from more than one parent class.
# This allows the child class to inherit attributes and methods from multiple class

# Example:

class Parent1:
```

```
    def method_parent1(self):
        print("Method from Parent1")

class Parent2:
    def method_parent2(self):
        print("Method from Parent2")

class Child(Parent1, Parent2):
    def method_child(self):
        print("Method from Child")

# Creating an instance of Child class
child = Child()
child.method_parent1()  # Inherited from Parent1
child.method_parent2()  # Inherited from Parent2
child.method_child()    # Defined in Child class
```

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

In [16]:

```
# 3. Multilevel Inheritance:

# In multilevel inheritance, a class inherits from a parent class, and another clas
# from that child class, creating a chain of inheritance.

# Example:

class Grandparent:
    def grandparent_method(self):
        print("Method from Grandparent")

class Parent(Grandparent):
    def parent_method(self):
        print("Method from Parent")

class Child(Parent):
    def child_method(self):
        print("Method from Child")

child = Child()
child.grandparent_method()  # Inherited from Grandparent
child.parent_method()       # Inherited from Parent
child.child_method()        # Defined in Child
```

```
Method from Grandparent
Method from Parent
Method from Child
```

In [17]:

```
# 4. Hierarchical Inheritance:

# In hierarchical inheritance, multiple child classes inherit from the same parent

# Example:

class Parent:
    def parent_method(self):
        print("Method from Parent")

class Child1(Parent):
    def child1_method(self):
        print("Method from Child1")

class Child2(Parent):
    def child2_method(self):
```

```
        print("Method from Child2")

child1 = Child1()
child2 = Child2()

child1.parent_method()  # Inherited from Parent
child2.parent_method()  # Inherited from Parent
```

```
Method from Parent
Method from Parent
```

In [18]:
```
# 5. Hybrid Inheritance:

# Hybrid inheritance is a combination of two or more types of inheritance.
# It can involve multiple inheritance, multilevel inheritance, or hierarchical inhe

# Example:

class Parent:
    def parent_method(self):
        print("Method from Parent")

class Child1(Parent):
    def child1_method(self):
        print("Method from Child1")

class Child2(Parent):
    def child2_method(self):
        print("Method from Child2")

class Grandchild(Child1, Child2):
    def grandchild_method(self):
        print("Method from Grandchild")

grandchild = Grandchild()
grandchild.parent_method()   # Inherited from Parent
grandchild.child1_method()   # Inherited from Child1
grandchild.child2_method()   # Inherited from Child2
grandchild.grandchild_method()  # Defined in Grandchild
```

```
Method from Parent
Method from Child1
Method from Child2
Method from Grandchild
```

In [ ]:
```
# Ques 7

# What is the Method Resolution Order (MRO) in Python? How can you retrieve it prog

# What is Method Resolution Order (MRO) in Python?

# The Method Resolution Order (MRO) in Python is the order in which a method (or at
# searched in a hierarchy of classes. This becomes important when a class inherits
# parent classes, as Python needs to decide the order in which to look for a metho

# Python follows the C3 Linearization Algorithm (also called the C3 superclass line
# for MRO, which ensures that:

# A child class is always checked before its parents.
# The order in which parents are listed is respected.
# No class is checked before its descendants.

# Example of Method Resolution Order:
```

In [19]:
```
class A:
    def show(self):
        print("A class")

class B(A):
    def show(self):
        print("B class")

class C(A):
    def show(self):
        print("C class")

class D(B, C):
    pass

d = D()
d.show()
```

```
B class
```

In [20]:
```
# How to Retrieve MRO Programmatically

# You can retrieve the Method Resolution Order of a class using the following metho

# Using .__mro__ attribute: The .__mro__ attribute returns a tuple that shows the M

print(D.__mro__)
```

```
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main_
_.A'>, <class 'object'>)
```

In [21]:
```
# Using mro() method: You can also use the mro() method, which returns a list of cl
print(D.mro())
```

```
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main_
_.A'>, <class 'object'>]
```

In [23]:
```
# Using help() function: The help() function provides detailed information about th

help(D)
```

In [24]:
```
# Ques 8

# Create an abstract base class `Shape` with an abstract method `area()`. Then crea
# `Circle` and `Rectangle` that implement the `area()` method.


from abc import ABC, abstractmethod
import math

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

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

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

```
# Subclass for Rectangle
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

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

# Testing the classes
circle = Circle(5)
rectangle = Rectangle(4, 6)

print(f"Circle Area: {circle.area()}")
print(f"Rectangle Area: {rectangle.area()}")
```

```
Circle Area: 78.53981633974483
Rectangle Area: 24
```

In [25]:

```
# Ques 9

# Demonstrate polymorphism by creating a function that can work with different shap
# calculate  and print their areas.

# Here's a demonstration of polymorphism using the Shape abstract base class and th
# Rectangle subclasses. We will create a function that works with different shape c
# and print their areas.

from abc import ABC, abstractmethod
import math

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

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

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

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

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

# Polymorphic function to calculate area
def print_area(shape):
    print(f"The area of the shape is: {shape.area()}")

# Creating objects
circle = Circle(5)
rectangle = Rectangle(4, 6)

# Demonstrating polymorphism
```

```
print_area(circle)     # Works with Circle
print_area(rectangle)  # Works with Rectangle
```

```
The area of the shape is: 78.53981633974483
The area of the shape is: 24
```

In [1]:
```
# Ques 10

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

class BankAccount:
    def __init__(self, account_number, initial_balance=0):
        # Private attributes
        self.__account_number = account_number  # Private account number
        self.__balance = initial_balance        # Private balance

    # Method to deposit money
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited ${amount}. New balance is ${self.__balance}.")
        else:
            print("Deposit amount must be positive.")

    # Method to withdraw money
    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew ${amount}. New balance is ${self.__balance}.")
        elif amount > self.__balance:
            print("Insufficient funds.")
        else:
            print("Withdrawal amount must be positive.")

    # Method to check the balance
    def get_balance(self):
        return self.__balance

    # Method to get the account number
    def get_account_number(self):
        return self.__account_number

# Testing the BankAccount class
account = BankAccount("987654321", 500)  # Creating an account with an initial balc

# Checking account information
print(f"Account Number: {account.get_account_number()}")
print(f"Current Balance: ${account.get_balance()}")

# Performing deposit and withdrawal operations
account.deposit(200)  # Depositing money
account.withdraw(100) # Withdrawing money
account.withdraw(1000) # Trying to withdraw more than balance
```

```
Account Number: 987654321
Current Balance: $500
Deposited $200. New balance is $700.
Withdrew $100. New balance is $600.
Insufficient funds.
```

In [ ]:
```
# Ques 11

# Write a class that overrides the `__str__` and `__add__` magic methods. What will
# you to do?
```

```
# In Python, magic methods (also known as dunder methods) are special methods that
# interact with Python's built-in syntax. Two common magic methods are __str__ and

#  1 __str__(self): This method is used to provide a human-readable string represer
#     an object when you call str() on it or use print().

# 2 __add__(self, other): This method is used to define how objects of your class
#      behave when the + operator is used between them.
```

In [2]:
```
# What do these methods allow you to do?

# __str__: It allows you to define a custom string representation of your class. Wh
#     of the class is printed or converted to a string, this method controls what i

# __add__: It allows you to define custom behavior for the + operator. When two ins
#      class are added together using +, this method determines what happens.

# Example:

# Let's create a class called Point, which represents a point in a 2D space with x
#  We'll override both the __str__ and __add__ methods.


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

    # Overriding the __str__ method to provide a readable string representation
    def __str__(self):
        return f"Point({self.x}, {self.y})"

    # Overriding the __add__ method to allow adding two Point objects
    def __add__(self, other):
        if isinstance(other, Point):
            return Point(self.x + other.x, self.y + other.y)
        return NotImplemented

# Creating two Point objects
point1 = Point(2, 3)
point2 = Point(5, 7)

# Printing the Point objects (uses the __str__ method)
print(point1)  # Output: Point(2, 3)
print(point2)  # Output: Point(5, 7)

# Adding two Point objects (uses the __add__ method)
point3 = point1 + point2
print(point3)  # Output: Point(7, 10)
```

```
Point(2, 3)
Point(5, 7)
Point(7, 10)
```

In [3]:
```
# Ques 12

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

# You can create a Python decorator that measures and prints the execution time of
# by using the time module to track when the function starts and finishes. Here's h

import time
```

```
def execution_time_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()  # Record the start time
        result = func(*args, **kwargs)  # Execute the function
        end_time = time.time()  # Record the end time
        execution_time = end_time - start_time  # Calculate execution time
        print(f"Execution time of {func.__name__}: {execution_time:.6f} seconds")
        return result  # Return the result of the function
    return wrapper

# Example usage of the decorator
@execution_time_decorator
def example_function():
    time.sleep(2)  # Simulating a function that takes 2 seconds to execute
    print("Function is done!")

# Call the decorated function
example_function()
```

```
Function is done!
Execution time of example_function: 2.001426 seconds
```

In [ ]:
```
# Ques 13

# Explain the concept of the Diamond Problem in multiple inheritance. How does Pyth

# The Diamond Problem in Multiple Inheritance

# The Diamond Problem is a well-known issue in object-oriented programming that occ
# class inherits from two classes that both inherit from a common base class. The p
# when there is ambiguity in determining which version of a method or attribute sho
# from the base class.
```

In [ ]:
```
#       A
#      / \
#     B   C
#      \ /
#       D

# Class B and class C both inherit from class A.
# Class D inherits from both B and C.

# If both B and C override a method from A, and D does not override it, the questio
# when D calls the method, should it use the version from B or C?
```

In [5]:
```
class A:
    def show(self):
        print("Method from class A")

class B(A):
    def show(self):
        print("Method from class B")

class C(A):
    def show(self):
        print("Method from class C")

class D(B, C):
    pass

d = D()
d.show()  # Which method will be called? B's or C's?
```

```
Method from class B
```

In [ ]:
```
# How Python Resolves the Diamond Problem

# Python uses Method Resolution Order (MRO) to resolve the ambiguity in multiple in
# MRO defines the order in which Python looks for a method in the hierarchy of clas
# follows the C3 Linearization Algorithm to ensure that the inheritance graph is re
# that preserves the parent-child relationships and avoids conflicts.

# Summary:

# Diamond Problem: Ambiguity that arises in multiple inheritance when a class inher
# or more classes that share a common base class, making it unclear which method or
# should be inherited.

# Python's Resolution: Python resolves the diamond problem using the Method Resolut
# which follows the C3 Linearization Algorithm. It defines a clear order for lookin
# attributes in the inheritance hierarchy.

# You can view the MRO of a class by calling mro() or using the __mro__ attribute.

# This ensures that multiple inheritance works predictably in Python without the am
# in languages that don't have such a resolution mechanism.
```

In [ ]:
```
# Ques 14

# Write a class method that keeps track of the number of instances created from a c

# To keep track of the number of instances created from a class, you can use a clas
# that increments each time a new instance is created. A class method can be used t
# variable to retrieve the number of instances.
```

In [6]:
```
class InstanceCounter:
    # Class variable to keep track of the number of instances
    instance_count = 0

    def __init__(self):
        # Increment the class variable each time a new instance is created
        InstanceCounter.instance_count += 1

    @classmethod
    def get_instance_count(cls):
        # Class method to return the current instance count
        return cls.instance_count

# Creating instances of the class
obj1 = InstanceCounter()
obj2 = InstanceCounter()
obj3 = InstanceCounter()

# Using the class method to get the number of instances
print(f"Number of instances created: {InstanceCounter.get_instance_count()}")
```

```
Number of instances created: 3
```

In [ ]:
```
# Key Points:

# Class Variable: Tracks the number of instances created.

# Class Method: Allows access to the class variable.

# The class method can be called without creating an instance
# (e.g., InstanceCounter.get_instance_count()).
```

In [7]:
```python
# Ques 15

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

# You can implement a static method in Python using the @staticmethod decorator. A
# not depend on the instance or class and does not modify the class or instance sta
# In this case, we'll implement a static method to check whether a given year is a


class YearChecker:

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

# Example usage of the static method
print(YearChecker.is_leap_year(2020))  # True, 2020 is a leap year
print(YearChecker.is_leap_year(1900))  # False, 1900 is not a leap year
print(YearChecker.is_leap_year(2000))  # True, 2000 is a leap year
```

```
True
False
True
```

In [ ]:
```python
# Key Points:

# Static Method: It can be called directly on the class without creating an instanc

# Leap Year Logic: Divisible by 4 but not by 100, or divisible by 400.
```

In [ ]:

In [ ]: