OOPS ASSIGNMENT

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

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

1. Encapsulation: This concept binds data and methods that manipulate that data into a single unit, called a class or object. Encapsulation helps protect the data from external interference and misuse.
2. Abstraction: Abstraction involves showing only the necessary information to the outside world while hiding the implementation details. This concept helps reduce complexity and improves modularity.
3. Inheritance: Inheritance allows one class to inherit the properties and behavior of another class. The child class inherits all the fields and methods of the parent class and can also add new fields and methods or override the ones inherited from the parent class.
4. Polymorphism: Polymorphism is the ability of an object to take on multiple forms. This can be achieved through method overloading (multiple methods with the same name but different parameters) or method overriding (a child class providing a different implementation of a method already defined in its parent class).
5. Composition: Composition is a concept that allows objects to contain other objects or collections of objects. This enables the creation of complex objects from simpler ones, promoting reusability and modularity.

These five key concepts form the foundation of Object-Oriented Programming and enable developers to create robust, scalable, and maintainable software systems.

Question-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:
    """
    A class representing a Car with attributes for make, model, and year.
    """

    def __init__(self, make, model, year):
        """
        Initializes a Car object with the given make, model, and year.

        Args:
            make (str): The car's make.
            model (str): The car's model.
            year (int): The car's year.
        """
        self.make = make
        self.model = model
        self.year = year

    def display_info(self):
        """
        Displays the car's information.
        """
        print(f"Make: {self.make}")
        print(f"Model: {self.model}")
        print(f"Year: {self.year}")

# Example usage:
my_car = Car("Toyota", "Camry", 2020)
my_car.display_info()
```

```
Make: Toyota
Model: Camry
Year: 2020
```

This code defines a Car class with the following attributes:

- make: The car's make.
- model: The car's model.
- year: The car's year.

The display_info method prints out the car's information. The example usage demonstrates how to create a Car object and display its information.

.

Question-3 Explain the difference between instance methods and class methods. Provide an example of each.

Answer- In object-oriented programming (OOP), methods are functions that belong to a class or an instance of a class. There are two types of methods: instance methods and class methods.

Instance Methods:

Instance methods are methods that belong to an instance of a class. They have access to the instance's attributes and can modify them. Instance methods are used to perform actions that are specific to an individual instance of a class.

Example:

```
class Car:
    def __init__(self, color):
        self.color = color

    def honk(self):
        print(f"The {self.color} car honks!")

my_car = Car("red")
my_car.honk()
```

```
The red car honks!
```

In this example, honk is an instance method that belongs to the my_car instance of the Car class. It has access to the color attribute of the instance and uses it to print a message.

Class Methods:

Class methods are methods that belong to a class itself, rather than to an instance of the class. They have access to the class's attributes and can modify them. Class methods are used to perform actions that are related to the class as a whole, rather than to individual instances.

Example:

```
class Car:
    num_cars = 0

    def __init__(self, color):
        self.color = color
        Car.num_cars += 1

    @classmethod
    def get_num_cars(cls):
        return cls.num_cars

my_car1 = Car("red")
my_car2 = Car("blue")

print(Car.get_num_cars())
```

In this example, get_num_cars is a class method that belongs to the Car class. It has access to the num_cars attribute of the class and returns its value. The @classmethod decorator is used to indicate that this method is a class method.

Note that class methods can also be used as alternative constructors, which can be useful in certain situations.

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

Answer- Python does not support method overloading in the classical sense, unlike languages such as Java or C++. In Python, method overloading is achieved through optional arguments and default values.

Here's an example:

```
class Calculator:
    def calculate(self, *args):
        if len(args) == 1:
            return args[0] ** 2
        elif len(args) == 2:
            return args[0] + args[1]
        else:
            raise ValueError("Invalid number of arguments")

calculator = Calculator()
print(calculator.calculate(5))
print(calculator.calculate(5, 10))
```

```
25
15
```

In this example, the calculate method can take either one or two arguments. The *args syntax allows the method to accept a variable number of arguments. The method then uses the len function to determine the number of arguments passed and performs the corresponding calculation.

Another way to achieve method overloading in Python is by using the **kwargs syntax to accept keyword arguments. Here's an example:

```
class Calculator:
    def calculate(self, **kwargs):
        if 'square' in kwargs:
            return kwargs['square'] ** 2
        elif 'add' in kwargs and 'num1' in kwargs and 'num2' in kwargs:
            return kwargs['num1'] + kwargs['num2']
        else:
            raise ValueError("Invalid keyword arguments")

calculator = Calculator()
print(calculator.calculate(square=5))
print(calculator.calculate(add=True, num1=5, num2=10))
```

```
25
15
```

In this example, the calculate method accepts keyword arguments using the **kwargs syntax. The method then checks for specific keyword arguments and performs the corresponding calculation.

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

Answer- Python has three types of access modifiers:

1. Public Access Modifier: In Python, all variables and methods are public by default. Public members can be accessed from anywhere in the program. They are denoted by no prefix or underscore.
2. Protected Access Modifier: Protected members are intended to be used within the class itself or by its subclasses. They are denoted by a single underscore prefix (e.g., _variable or _method()). While Python does not enforce access control for protected members, it serves as a convention to indicate that the member should not be accessed directly from outside the class.
3. Private Access Modifier: Private members are intended to be used within the class itself only. They are denoted by a double underscore prefix (e.g., **variable or __method()). Python internally changes the name of the private member to include the class name, making it more difficult to access from outside the class. However, this is not a strict access control mechanism, and private members can still be accessed using their mangled name (e.g., _classname**variable).

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

Answer- Here are the five types of inheritance in Python:

1. Single Inheritance In single inheritance, a child class inherits from only one parent class.
2. Multiple Inheritance In multiple inheritance, a child class can inherit from multiple parent classes.
3. Multilevel Inheritance In multilevel inheritance, a child class inherits from a parent class, which in turn inherits from another parent class.
4. Hierarchical Inheritance In hierarchical inheritance, multiple child classes inherit from the same parent class.
5. Hybrid Inheritance Hybrid inheritance is a combination of multiple inheritance and multilevel inheritance.

Here's an example of multiple inheritance:

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

    def eat(self):
        print(f"{self.name} is eating.")

class Mammal:
    def __init__(self, hair_color):
        self.hair_color = hair_color

    def walk(self):
        print("The mammal is walking.")

class Dog(Animal, Mammal):
    def __init__(self, name, hair_color):
        Animal.__init__(self, name)
        Mammal.__init__(self, hair_color)

    def bark(self):
        print("The dog is barking.")

my_dog = Dog("Rex", "Brown")
```

```
my_dog.eat()
my_dog.walk()
my_dog.bark()
```

```
Rex is eating.
The mammal is walking.
The dog is barking.
```

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

Answer- In Python, the Method Resolution Order (MRO) is the order in which Python searches for methods in a class inheritance hierarchy. When a method is called on an object, Python searches for the method in the object's class, then in its parent classes, and so on.

Python uses a technique called C3 Linearization to resolve the MRO. This algorithm ensures that the MRO is consistent and predictable, even in the presence of multiple inheritance.

To retrieve the MRO programmatically, you can use the mro() method, which is available on all classes. Here's an example:

print(C.**mro**) # Output: (**main**.C, **main**.B, **main**.A, object)

```
class A:
    pass

class B(A):
    pass

class C(B):
    pass

print(C.mro())
```

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

In this example, the mro() method returns a list of classes in the MRO of class C. The list includes class C itself, its parent class B, its grandparent class A, and the ultimate parent class object.

Alternatively, you can use the **mro** attribute to access the MRO:

```
print(C.__mro__)  # Output: (__main__.C, __main__.B, __main__.A, object)
```

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

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

Answer-

```
from abc import ABC, abstractmethod
import math
# Abstract base class "Shape"
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

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

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

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

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

# Example usage
circle = Circle(5)
print(f"Circle area: {circle.area():.2f}")

rectangle = Rectangle(4, 6)
print(f"Rectangle area: {rectangle.area()}")
```

```
Circle area: 78.54
Rectangle area: 24
```

In this example:

- We define an abstract base class Shape with an abstract method area().
- We create two subclasses Circle and Rectangle that inherit from Shape.
- Each subclass implements the area() method according to its geometric formula.
- We demonstrate the usage of these classes by creating instances of Circle and Rectangle and calculating their areas.

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

Answer-

Here's an example demonstration of polymorphism in Python:

```
from abc import ABC, abstractmethod
import math

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

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

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

# Subclass "Rectangle"
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height

# Subclass "Triangle"
class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height

    def area(self):
        return 0.5 * self.base * self.height

# Polymorphic function to calculate and print shape areas
def print_shape_areas(shapes):
    for shape in shapes:
        print(f"Area of {type(shape).__name__}: {shape.area():.2f}")

# Example usage
circle = Circle(5)
rectangle = Rectangle(4, 6)
triangle = Triangle(3, 7)

shapes = [circle, rectangle, triangle]
print_shape_areas(shapes)
```

```
Area of Circle: 78.54
Area of Rectangle: 24.00
Area of Triangle: 10.50
```

In this example:

- We define an abstract base class Shape with an abstract method area().
- We create three subclasses Circle, Rectangle, and Triangle that inherit from Shape and implement the area() method.
- We define a polymorphic function print_shape_areas() that takes a list of shapes as input and calculates and prints their areas using the area() method.
- We demonstrate the usage of this function by creating instances of different shape classes and passing them to the print_shape_areas() function.

This example showcases polymorphism, as the print_shape_areas() function works with different shape objects without knowing their specific class types at compile time.

Question-10 Implement encapsulation in a "BankAccount' class with private attributes for balance and account_number', include methods for deposit, withdrawal, and balance inquiry.

Answer- Here's an example implementation of encapsulation in a BankAccount class:

```
class BankAccount:
    def __init__(self, account_number, initial_balance):
        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("Invalid deposit amount.")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew ${amount:.2f}. New balance: ${self.__balance:.2f}")
        elif amount <= 0:
            print("Invalid withdrawal amount.")
        else:
            print("Insufficient funds.")

    def get_balance(self):
        return self.__balance

    def get_account_number(self):
        return self.__account_number

# Example usage
account = BankAccount("1234567890", 1000.0)
print(f"Account Number: {account.get_account_number()}")
print(f"Initial Balance: ${account.get_balance():.2f}")

account.deposit(500.0)
account.withdraw(200.0)
account.withdraw(2000.0)  # Insufficient funds

print(f"Final Balance: ${account.get_balance():.2f}")
```

```
Account Number: 1234567890
Initial Balance: $1000.00
Deposited $500.00. New balance: $1500.00
Withdrew $200.00. New balance: $1300.00
Insufficient funds.
Final Balance: $1300.00
```

In this implementation:

- The BankAccount class encapsulates the account_number and balance attributes by making them private (__account_number and __balance).
- The class provides public methods for deposit (deposit()), withdrawal (withdraw()), and balance inquiry (get_balance() and get_account_number()).
- The deposit() and withdraw() methods modify the private balance attribute while ensuring that the operations are valid (e.g., deposit amount is positive, withdrawal amount does not exceed the balance).
- The get_balance() and get_account_number() methods provide controlled access to the private attributes.

Question-11 Write a class that overrides the strand_add_magic methods. What will these methods allow you to do?

Answer-Here's an example implementation of a class that overrides the **add** and **radd** magic methods:

```
class MagicString:
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        if isinstance(other, str):
            return MagicString(self.value + other)
        else:
            raise TypeError("Unsupported operand type for +")

    def __radd__(self, other):
        if isinstance(other, str):
            return MagicString(other + self.value)
        else:
            raise TypeError("Unsupported operand type for +")

    def __str__(self):
        return self.value

# Example usage
magic_str = MagicString("Hello, ")
print(magic_str + "World!")
print("Magic " + magic_str)
```

```
Hello, World!
Magic Hello,
```

In this example:

- The MagicString class overrides the **add** method to support concatenation with string objects using the + operator.
- The **radd** method is overridden to support reversed concatenation (i.e., when the MagicString object is on the right side of the + operator).
- The **str** method is overridden to provide a string representation of the MagicString object.

These magic methods allow you to perform string concatenation operations with MagicString objects using the + operator, making the class more intuitive and user-friendly.

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

Answer- Here's an example implementation of a decorator that measures and prints the execution time of a function:

In this example:

- The timer_decorator function takes a function func as input and returns a new function wrapper.
- The wrapper function measures the execution time of the original function func by recording the start and end times using time.time().
- The execution time is calculated by subtracting the start time from the end time.
- The execution time is printed to the console along with the name of the function.
- The wraps decorator from the functools module is used to preserve the original function's metadata (e.g., name, docstring) in the wrapped function.
- The example_function is decorated with @timer_decorator to measure its execution time.

```
import time
from functools import wraps

def timer_decorator(func):
    @wraps(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__}' executed in {execution_time:.4f} seconds.")
        return result
    return wrapper

# Example usage
@timer_decorator
def example_function():
    time.sleep(1)  # Simulate some time-consuming operation
    print("Example function executed.")

example_function()
```

```
Example function executed.
Function 'example_function' executed in 1.0021 seconds.
```

In this example:

- The timer_decorator function takes a function func as input and returns a new function wrapper.
- The wrapper function measures the execution time of the original function func by recording the start and end times using time.time().
- The execution time is calculated by subtracting the start time from the end time.
- The execution time is printed to the console along with the name of the function.
- The wraps decorator from the functools module is used to preserve the original function's metadata (e.g., name, docstring) in the wrapped function.
- The example_function is decorated with @timer_decorator to measure its execution time.

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

Answer- The Diamond Problem is a well-known issue in multiple inheritance, where a class inherits conflicting attributes or methods from its parent classes. This problem arises when two classes, B and C, inherit from a common base class A, and then another class D inherits from both B and C.

Here's an illustration of the Diamond Problem:

```
   A
 /   \
```

B C \ / D

Suppose classes B and C override a method m() from class A, and class D inherits from both B and C. When D.m() is called, which implementation of m() should be used: B's or C's?

Python resolves the Diamond Problem using a technique called C3 Linearization, also known as the "Method Resolution Order" (MRO). The MRO is a standard for resolving the order of inheritance in the presence of multiple inheritance.

Here's how Python's MRO resolves the Diamond Problem:

1. List the classes in the inheritance order: D, B, C, A, object.
2. Remove any duplicate classes: D, B, C, A, object.
3. If a class has already been visited, skip it: No changes.
4. The resulting list is the MRO: D, B, C, A, object.

When Python looks for a method or attribute in class D, it searches the classes in the MRO order. In this case, Python would look for m() in the following order:

1. D (class D itself)
2. B (class D's first parent)
3. C (class D's second parent)
4. A (the common base class)
5. object (the ultimate base class)

If Python finds m() in class B, it will use that implementation. If not, it will continue searching in the next classes.

Here's an example code snippet demonstrating Python's MRO:

```
class A:
    def m(self):
        print("A's implementation")

class B(A):
    def m(self):
        print("B's implementation")

class C(A):
    def m(self):
        print("C's implementation")

class D(B, C):
    pass

d = D()
d.m()
```

```
B's implementation
```

In this example, class D inherits from both B and C. When d.m() is called, Python uses the MRO to resolve the method call. Since B is listed before C in the MRO, Python uses B's implementation of m().

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

Answer- Here's an example implementation of a class that keeps track of the number of instances created:

```
class InstanceTracker:
    num_instances = 0

    def __init__(self):
        InstanceTracker.num_instances += 1

    @classmethod
    def get_num_instances(cls):
        return cls.num_instances

# Example usage
print(InstanceTracker.get_num_instances())

instance1 = InstanceTracker()
instance2 = InstanceTracker()
instance3 = InstanceTracker()

print(InstanceTracker.get_num_instances())
```

```
0
3
```

In this example:

- The InstanceTracker class has a class-level attribute num_instances initialized to 0.
- The **init** method increments the num_instances attribute each time a new instance is created.
- The get_num_instances class method returns the current value of num_instances.
- In the example usage, we create three instances of InstanceTracker and print the number of instances before and after creation.

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

Answer- Here's an example implementation of a class with a static method that checks if a given year is a leap year:

```
class LeapYearChecker:
    @staticmethod
    def is_leap_year(year):
        return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)

# Example usage
print(LeapYearChecker.is_leap_year(2020))
print(LeapYearChecker.is_leap_year(2019))
```

```
True
False
```

```
In this example:

- The LeapYearChecker class has a static method is_leap_yea
that takes a year parameter.
- The is_leap_year method checks if the given year is a lea
year using the standard leap year rules:

    - The year must be evenly divisible by 4.
    - If the year is a century year (i.e., it is divisible
    by 100), it must also be divisible by 400.
- In the example usage, we call the is_leap_year method wit
different year values and print the results.
```

In this example:

- The LeapYearChecker class has a static method is_leap_year that takes a year parameter.
- The is_leap_year method checks if the given year is a leap year using the standard leap year rules:
    - The year must be evenly divisible by 4.
    - If the year is a century year (i.e., it is divisible by 100), it must also be divisible by 400.
- In the example usage, we call the is_leap_year method with different year values and print the results.