OOPS -1

Assignment Solution

PW SKILLS

1. **What is Object-Oriented Programming (OOP), and how does it differ from procedural programming?**

Answer

Object-Oriented Programming (OOP) is a programming paradigm that focuses on creating objects, which are instances of classes. OOP is based on the following core principles:

- Encapsulation: Binding data (attributes) and the methods that operate on that data within a single unit (the class).
- Abstraction: Focusing on essential features and hiding unnecessary details.
- Inheritance: Creating new classes based on existing ones, inheriting their properties and behaviors.
- Polymorphism: The ability of objects to take on many forms, allowing objects of different classes to be treated as objects of a common superclass.

In contrast, procedural programming is a programming paradigm that focuses on breaking down a program into a series of procedures or functions. Procedural programming is more concerned with describing the steps that the program should take to achieve a particular result, rather than modeling real-world objects and their interactions.

2. **Define the terms Class, Attribute, and Object in the context of OOP in Python.**

Answer

Class: A class is a blueprint or template for creating objects. It defines the data (attributes) and the methods (functions) that the objects of that class will have.

Attribute: An attribute is a variable that is associated with a class or an object. Attributes can be used to store data or define the characteristics of an object.

Object: An object is an instance of a class. It is a real-world entity that has its own state (attributes) and behavior (methods).

3. Explain the role of instances in Python classes. How are instances created, and what purpose do they serve?

Answer

Instances: Instances are the individual objects created from a class. Each instance has its own set of attributes and can perform the methods defined in the class. Instances are created using the class name followed by parentheses, like this: `my_object = ClassName()`.

Instances serve the purpose of allowing you to create and work with specific objects based on the class definition.

4. Describe the process of defining a class in Python, including the syntax and components involved.

Answer:

Defining a class in Python:

```
class ClassName:
    # class attributes
    class_attr1 = value
    class_attr2 = value

    def __init__(self, param1, param2):
        # instance attributes
        self.instance_attr1 = param1
        self.instance_attr2 = param2

    # methods
    def method1(self):
        # method implementation
        pass

    def method2(self, arg):
        # method implementation
        pass
```

``

The key components are:

- `class ClassName:` defines the class name.
- `__init__(self, ...)` is the constructor method, used to initialize the object's attributes.
- Instance attributes are defined using `self.attribute_name = value`.
- Methods are defined using the `def method_name(self, ...)` syntax.

5. How are functions defined within a class in Python? What is the significance of the 'self' parameter?

Answer.

Defining functions within a class:

Functions defined within a class are called methods. The first parameter of a method is conventionally named `self`, and it represents the instance of the class that the method is being called on.

The `self` parameter allows the method to access and manipulate the instance's attributes.

6.Explain how class functions can be called both from outside and inside the class in Python, providing examples for each scenario.

Answer:

Calling class functions:

- From outside the class: `object_instance.method_name(arguments)`
- From inside the class: `self.method_name(arguments)`

7. **Differentiate between class attributes and instance attributes in Python, and provide examples of each.**

Answer

Class attributes vs. Instance attributes:

- Class attributes: Attributes that are shared among all instances of a class. They are defined at the class level.
- Instance attributes: Attributes that are specific to each instance of a class. They are defined within the `__init__` method or other instance methods.

8. **Differentiate between class method ,instance method and static method in Python, and provide examples of each.**

Answer:

Class method, Instance method, and Static method:

- Instance method: A method that takes the `self` parameter and can access and modify instance attributes.
- Class method: A method that takes the `cls` parameter and can access and modify class attributes. Defined using the `@classmethod` decorator.
- Static method: A method that does not take the `self` or `cls` parameter and does not have access to instance or class attributes. Defined using the `@staticmethod` decorator.

9. **What is a constructor in Python classes? How is it defined, and what is its purpose?**

Answer

Constructors in Python classes: The `__init__` method is the constructor in Python classes. It is called automatically when an object is created from the class. The constructor is used to initialize the object's attributes with the provided arguments.

10. **What are access specifiers in Python ,explain with examples.**

Answer

Access Specifiers in Python:

- In Python, there are no explicit access specifiers like `public`, `private`, and `protected` as in some other object-oriented programming languages. However, Python has a naming convention to indicate the intended access level of a class member:
- **Public:** Variables and methods that are accessible from anywhere, both inside and outside the class. They are named without any special prefixes.

```
class MyClass:
      def __init__(self, value):
          self.public_attr = value
      def public_method(self):
          print("This is a public method.")
```

- **Protected**: Variables and methods that are intended to be accessed only by the class and its subclasses. They are named with a single leading underscore (`_`).

```
lass MyClass:
       def __init__(self, value):
           self._protected_attr = value
       def _protected_method(self):
           print("This is a protected method.")
```

- **Private**: Variables and methods that are intended to be accessed only within the class. They are named with two leading underscores (`__`).

```
  class MyClass:
       def __init__(self, value):
           self.__private_attr = value
       def __private_method(self):
           print("This is a private method.")
```

While Python does not enforce access control, the naming convention is widely used to indicate the intended access level, and it is generally considered good practice to follow it.

11. What is encapsulation in python , explain it with an example.How does it promote data hiding and abstraction, and why is it important in software design.

Answer.

Encapsulation in Python:

Encapsulation is a fundamental concept in object-oriented programming that involves bundling data (attributes) and the methods that operate on that data within a single unit (the class). It promotes data hiding and abstraction.

Data Hiding: Encapsulation allows you to hide the internal implementation details of an object from the outside world. By using private attributes and providing controlled access through methods (getters and setters), you can ensure that the data is accessed and modified in a consistent and secure way.

Abstraction: Encapsulation helps achieve abstraction by exposing only the essential features of an object and hiding the complex implementation details. This allows you to work with objects at a higher level of abstraction, without needing to know or care about the internal workings of the object.

Example:

```
    class BankAccount:
        def __init__(self, owner, balance):
            self.__owner = owner  # Private attribute
            self.__balance = balance  # Private attribute

        def deposit(self, amount):
            self.__balance += amount

        def withdraw(self, amount):
            if self.__balance >= amount:
                self.__balance -= amount
            else:
                print("Insufficient funds.")

        def get_balance(self):
            return self.__balance

        def get_owner(self):
            return self.__owner
    ```
```

In this example, the `BankAccount` class encapsulates the account owner and balance as private attributes, and provides controlled access to these attributes through public methods (`deposit()`, `withdraw()`, `get_balance()`, and `get_owner()`). This promotes data hiding and abstraction, ensuring that the internal implementation details of the `BankAccount` class are hidden from the outside world.

**12. Explain the Getter and Setter method in Python with examples, and explain its importance.**

**Answer**

**Getters and Setters in Python:**

Getters and setters are methods used to access and modify the values of private or protected attributes in a class.

**Getters:**

Getters are methods that provide read-only access to the private or protected attributes of a class. They allow you to retrieve the value of an attribute without exposing the internal implementation details.

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

        def get_name(self):
            return self.__name

        def get_age(self):
            return self.__age
    ```
```

In this example, the `get_name()` and `get_age()` methods are getters that allow access to the private `__name` and `__age` attributes, respectively.

Setters:
Setters are methods that allow you to modify the values of private or protected attributes. They provide a controlled way to update the internal state of an object.

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

         def get_name(self):
             return self.__name

         def get_age(self):
             return self.__age

         def set_age(self, new_age):
             if new_age < 0:
                 print("Age cannot be negative.")
             else:
                 self.__age = new_age
     ```
```

In this example, the `set_age()` method is a setter that allows you to update the `__age` attribute, with a check to ensure that the new age is non-negative.

**Getters and setters are important because they:**

- Provide a way to control access to object attributes.
- Allow for data validation and normalization.
- Enable the implementation of computed properties.
- Facilitate future modifications to the class without affecting existing code that uses the class.

**13. Discuss the concept of inheritance and explain all types of inheritance with suitable examples in Python. How are subclasses created, and what advantages does inheritance offer?**

**Answer:**
**Inheritance in Python:**
Inheritance is a fundamental concept in object-oriented programming that allows you to create new classes (derived or child classes) based on existing classes (base or parent classes). The new classes inherit the attributes and methods of the existing classes, and can also add or modify them.

**Types of Inheritance in Python:**

1. **Single Inheritance: A derived class inherits from a single base class.**
```

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

    def speak(self):
        print("The animal makes a sound.")

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

    def speak(self):
        print("The dog barks.")
```

2. Multiple Inheritance: A derived class inherits from multiple base classes.

```
class Flyer:
    def fly(self):
        print("The animal is flying.")

class Swimmer:
    def swim(self):
        print("The animal is swimming.")

class Duck(Flyer, Swimmer):
    def __init__(self, name):
        self.name = name
```

3. Multilevel Inheritance: A derived class inherits from another derived class.

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

class Parent(GrandParent):
    def introduce(self):
        print(f"Hello, my name is {self.name}.")

class Child(Parent):
    def baby_talk(self):
        print("Goo goo ga ga!")
```

4. Hierarchical Inheritance: Multiple derived classes inherit from a single base class.

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

class Dog(Animal):
    def speak(self):
        print("The dog barks.")

class Cat(Animal):
    def speak(self):
        print("The cat meows.")
```

5. Hybrid inheritance is a combination of multiple inheritance and single inheritance. It occurs when a class inherits from more than one base class, and at least one of the base classes is itself a derived class.

Example:

```
class GrandParent:
    def grandparent_method(self):
        print("This is the GrandParent class.")

class Parent1(GrandParent):
    def parent1_method(self):
        print("This is the Parent1 class.")
class Parent2(GrandParent):
    def parent2_method(self):
        print("This is the Parent2 class.")

class Child(Parent1, Parent2):
    def child_method(self):
        print("This is the Child class.")

# Create an instance of the Child class
child_obj = Child()

# Call methods from different levels of the inheritance hierarchy
child_obj.child_method()
child_obj.parent1_method()
child_obj.parent2_method()
child_obj.grandparent_method()
In this example, the Child class inherits from both Parent1 and
Parent2, which are themselves derived from the GrandParent class.
This creates a hybrid inheritance structure.
```

When you create an instance of the Child class and call its methods, the appropriate methods are invoked based on the MRO (Method Resolution Order) of the Child class.

The output of the above code would be:

This is the Child class.
This is the Parent1 class.
This is the Parent2 class.
This is the GrandParent class.

Advantages of Inheritance:

- **Code Reuse:** Inherited classes can reuse the code from the base class, reducing code duplication.
- **Polymorphism:** Subclasses can override or extend the behavior of the base class.
- **Flexibility:** Inheritance allows for the creation of a hierarchy of related classes, making the code more flexible and maintainable.

14. Explain the concept of method overriding in Python classes. How does it allow subclasses to modify the behavior of superclass methods?

Answer:
Method Overriding in Python:

Method overriding is a feature in object-oriented programming that allows a subclass to provide its own implementation of a method that is already defined in its superclass. This allows subclasses to modify the behavior of superclass methods to suit their specific needs.

Example:

```
class Animal:
      def make_sound(self):
          print("The animal makes a sound.")

  class Dog(Animal):
      def make_sound(self):
          print("The dog barks.")

  dog = Dog()
  dog.make_sound()  # Output: The dog barks.

  animal = Animal()
  animal.make_sound()  # Output: The animal makes a sound.
  ```
```

In this example, the `Dog` class overrides the `make_sound()` method inherited from the `Animal` class. When the `make_sound()` method is called on an instance of the `Dog` class, it executes the implementation in the `Dog` class, rather than the implementation in the `Animal` class.

Method overriding allows subclasses to provide their own specialized implementation of a method, while still maintaining the same method signature as the superclass. This is a key aspect of polymorphism, as it allows objects of different classes to be treated as objects of a common superclass.

15. Discuss the significance of the 'super()' function in Python inheritance. How is it used, and what role does it play in accessing superclass methods and attributes?

**Answer:**

**'super()' Function in Python Inheritance:**

The `super()` function is used to call a method in a superclass from a subclass. It provides a way to access and invoke methods defined in the base class(es) of a class.

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

        def speak(self):
            print("The animal makes a sound.")

    class Dog(Animal):
        def __init__(self, name, breed):
            super().__init__(name)
            self.breed = breed

        def speak(self):
            super().speak()
            print("The dog barks.")
    ```
```

In this example:

- In the `__init__()` method of the `Dog` class, `super().__init__(name)` calls the `__init__()` method of the `Animal` class, allowing the `Dog` class to inherit the `name` attribute.
- In the `speak()` method of the `Dog` class, `super().speak()` calls the `speak()` method of the `Animal` class, and then the `Dog` class adds its own implementation.

The `super()` function plays a crucial role in accessing and invoking methods and attributes from the superclass(es), especially when working with inheritance hierarchies. It helps maintain the linkage between the subclass and its superclass(es), making the code more modular, maintainable, and flexible.

16. Explain MRO (Method Resolution Order) in detail with example.Explain its importance.

Answer

Method Resolution Order (MRO) in Python:

Method Resolution Order (MRO) is the order in which Python searches for a method in the inheritance hierarchy. When a method is called on an object, Python follows the MRO to find the appropriate implementation of the method.

You can view the MRO of a class using the `__mro__` attribute or the `mro()` method of the `type` class:

```
class A:
        pass

    class B(A):
        pass

class C(A):
        pass

    class D(B, C):
        pass

    print(D.__mro__)
    # Output: (<class '__main__.D'>, <class '__main__.B'>, <class
'__main__.C'>, <class '__main__.A'>, <class 'object'>)
```

The MRO is important because it determines which method implementation will be used when a method is called on an object. Python follows a specific algorithm to determine the MRO, known as the C3 Linearization algorithm.

Understanding the MRO is crucial when working with multiple inheritance, as it helps you predict the behavior of your code and avoid potential conflicts or unexpected results.

The MRO is also useful when working with the `super()` function, as it ensures that the correct superclass method is called, even in complex inheritance hierarchies.

In summary, the MRO is a fundamental concept in Python's object-oriented programming model, and understanding it is essential for writing robust and maintainable code, especially when working with inheritance and multiple inheritance.

Programming Assignment:

Please refer to this Colab Notebook: **Week-05.ipynb**