Unclassified Object-oriented programming (OOP) is a programming paradigm that organizes code around objects, which are instances of classes. OOP allows you to structure your code in a way that models real-world objects and their interactions. Here's a brief introduction to key concepts in OOP:

1. Classes and Objects:
    - A class is a blueprint or template that defines the attributes (data) and methods (functions) that an object of that class will have.
    - An object is an instance of a class. It represents a specific entity or concept based on the class definition.
    - Example:

```
# Class definition
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        print("Woof!")

# Creating objects (instances) of the class
my_dog = Dog("Buddy")
your_dog = Dog("Max")

# Accessing attributes and calling methods of objects
print(my_dog.name)    # Output: "Buddy"
my_dog.bark()         # Output: "Woof!"
```

2. Attributes and Methods:
    - Attributes are variables that store data specific to each object. They represent the state or characteristics of an object.
    - Methods are functions defined within a class that define the behavior or actions that objects of that class can perform.
    - Attributes and methods are accessed using dot notation on the object.
    - Example:

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

    def calculate_area(self):
        return 3.14 * self.radius**2

my_circle = Circle(5)
print(my_circle.radius)              # Output: 5
print(my_circle.calculate_area())    # Output: 78.5
```

3. Encapsulation:

   - Encapsulation is the principle of bundling data and methods together within a class, hiding the internal details and providing a public interface for interaction.
   - It allows for data abstraction and protection, as the internal implementation is hidden from external access.
   - Access modifiers like public, private, and protected can be used to control the visibility and accessibility of attributes and methods.
   - Example:

```
class BankAccount:
    def __init__(self, account_number):
        self.__account_number = account_number    # Private attribute

    def get_account_number(self):
        return self.__account_number    # Public method to access the
private attribute

my_account = BankAccount("1234567890")
print(my_account.get_account_number())    # Output: "1234567890"
# print(my_account.__account_number)     # Error: Attribute is private
```

4. Inheritance:

   - Inheritance allows you to create a new class (derived or child class) based on an existing class (base or parent class).
   - The derived class inherits the attributes and methods of the base class and can also have its own additional attributes and methods.
   - Inheritance supports code reuse and the creation of hierarchical relationships between classes.
   - Example:

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

    def make_sound(self):
        pass    # Abstract method

class Dog(Animal):
    def make_sound(self):
        print("Woof!")

my_dog = Dog("Buddy")
print(my_dog.name)        # Output: "Buddy"
my_dog.make_sound()       # Output: "Woof!"
```

Advanced

5. Polymorphism:

   - Polymorphism refers to the ability of objects of different Polymorphism is a key concept in object-oriented programming that allows objects of different classes to be treated as objects of a common base class. It enables flexibility and extensibility in code by allowing different objects to respond to the same method call in different ways. Here's an elaboration on polymorphism:

6. Method Overriding:

   - Method overriding is a form of polymorphism where a derived class provides its own implementation of a method that is already defined in its base class.
   - The overridden method in the derived class is called instead of the base class method when the method is invoked on an object of the derived class.
   - Example:

```
class Animal:
    def make_sound(self):
        print("Generic animal sound")

class Dog(Animal):
    def make_sound(self):
        print("Woof!")

class Cat(Animal):
    def make_sound(self):
        print("Meow!")

my_dog = Dog()
my_dog.make_sound()    # Output: "Woof!"

my_cat = Cat()
my_cat.make_sound()    # Output: "Meow!"
```

7. Polymorphic Function Calls:

   - Polymorphism allows functions or methods to accept objects of different classes that implement a common interface or have a common base class.
   - By treating these objects as instances of the base class, the same function or method can be applied to different objects, resulting in different behaviors based on the specific object type.
   - Example:

```
class Shape:
    def calculate_area(self):
        pass    # Abstract method

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

```
    def calculate_area(self):
        return 3.14 * self.radius**2

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

    def calculate_area(self):
        return self.length * self.width

def print_area(shape):
    print("Area:", shape.calculate_area())

my_circle = Circle(5)
print_area(my_circle)    # Output: "Area: 78.5"

my_rectangle = Rectangle(4, 6)
print_area(my_rectangle)    # Output: "Area: 24"
```

8. Polymorphic Collections:

- Polymorphism allows objects of different classes to be stored in the same collection, such as a list or dictionary, by treating them as instances of a common base class.
- This enables you to iterate over the collection and perform common operations on the objects, regardless of their specific types.
- Example:

```
class Animal:
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        print("Woof!")

class Cat(Animal):
    def make_sound(self):
        print("Meow!")

animals = [Dog(), Cat(), Dog()]

for animal in animals:
    animal.make_sound()
# Output:
# "Woof!"
# "Meow!"
# "Woof!"
```

Polymorphism enhances code flexibility and extensibility by allowing different objects to exhibit different behaviors while adhering to a common interface or base class. It simplifies code design, promotes code reuse, and enables more dynamic and flexible programming. Unclassified