Python OOP Concepts

Object Oriented Programming is a fundamental concept in Python, empowering developers to build modular, maintainable and scalable applications.

OOP is a way of organizing code that uses objects and classes to represent real-world entities and their behavior. In OOP, object has attributes thing that has specific data and can perform certain actions using methods.

Key Features of OOP in Python:

- Organizes code into classes and objects
- Supports encapsulation to group data and methods together
- Enables inheritance for reusability and hierarchy
- Allows polymorphism for flexible method implementation
- Improves modularity, scalability, and maintainability

Characteristics of OOP (Object Oriented Programming)

Python supports the core principles of object-oriented programming, which are the building blocks for designing robust and reusable software. The diagram below demonstrates these core principles:

Python OOPs Concepts

1. Class

A class is a collection of objects. Classes are blueprints for creating objects. A class defines a set of attributes and methods that the created objects (instances) can have.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Example: Myclass.Myattribute

Creating a Class

Here, class keyword indicates that we are creating a class followed by name of the class (Dog in this case).

```
class Dog:

    species = "Canine"  # Class attribute


    def __init__(self, name, age):

        self.name = name  # Instance attribute

        self.age = age  # Instance attribute
```

Explanation:

- **class Dog:** Defines a class named Dog.
- **species:** A class attribute shared by all instances of the class.
- **__init__ method:** Initializes the name and age attributes when a new object is created.

2. Objects

An Object is an instance of a Class. It represents a specific implementation of the class and holds its own data.

An object consists of:

- **State:** It is represented by the attributes and reflects the properties of an object.

- **Behavior:** It is represented by the methods of an object and reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Creating Object

Creating an object in Python involves instantiating a class to create a new instance of that class. This process is also referred to as object instantiation.

```
class Dog:
    species = "Canine"  # Class attribute

    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age  # Instance attribute

# Creating an object of the Dog class
dog1 = Dog("Buddy", 3)

print(dog1.name)
print(dog1.species)
```

Output

```
Buddy
Canine
```

Explanation:

- **dog1 = Dog("Buddy", 3):** Creates an object of the Dog class with name as "Buddy" and age as 3.
- **dog1.name:** Accesses the instance attribute name of the dog1 object.

- **dog1.species:** Accesses the class attribute species of the dog1 object.

Self Parameter

Self parameter is a reference to the current instance of the class. It allows us to access the attributes and methods of the object.

Example: In this example, we create a Dog class with both class and instance attributes, then demonstrate how to access them using the self parameter.

```
class Dog:
    species = "Canine"  # Class attribute

    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age  # Instance attribute

dog1 = Dog("Buddy", 3)  # Create an instance of Dog
dog2 = Dog("Charlie", 5)  # Create another instance of Dog

print(dog1.name, dog1.age, dog1.species)  # Access instance and class attributes
print(dog2.name, dog2.age, dog2.species)  # Access instance and class attributes
print(Dog.species)  # Access class attribute directly
```

Output

```
Buddy 3 Canine
Charlie 5 Canine
Canine
```

Explanation:

- **self.name:** Refers to the name attribute of the object (dog1) calling the method.
- **dog1.bark():** Calls the bark method on dog1.

__init__ Method

__init__ method is the constructor in Python, automatically called when a new object is created. It initializes the attributes of the class.

Example: In this example, we create a Dog class and use __init__ method to set the name and age of each dog when creating an object.

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


dog1 = Dog("Buddy", 3)
print(dog1.name)
```

Output

Buddy

Explanation:

- **__init__:** Special method used for initialization.
- **self.name and self.age:** Instance attributes initialized in the constructor.

Class and Instance Variables

In Python, variables defined in a class can be either class variables or instance variables, and understanding the distinction between them is crucial for object-oriented programming.

Class Variables

These are the variables that are shared across all instances of a class. It is defined at the class level, outside any methods. All objects of the class share the same value for a class variable unless explicitly overridden in an object.

Instance Variables

Variables that are unique to each instance (object) of a class. These are defined within the __init__ method or other instance methods. Each object maintains its own copy of instance variables, independent of other objects.

Example: In this example, we create a Dog class to show difference between class variables and instance variables. We also demonstrate how modifying them affects objects differently.

```
class Dog:

    # Class variable

    species = "Canine"


    def __init__(self, name, age):

        # Instance variables

        self.name = name

        self.age = age


# Create objects

dog1 = Dog("Buddy", 3)

dog2 = Dog("Charlie", 5)


# Access class and instance variables

print(dog1.species)  # (Class variable)

print(dog1.name)     # (Instance variable)

print(dog2.name)     # (Instance variable)


# Modify instance variables

dog1.name = "Max"

print(dog1.name)     # (Updated instance variable)
```

```
# Modify class variable
Dog.species = "Feline"
print(dog1.species)  # (Updated class variable)
print(dog2.species)
```

Output

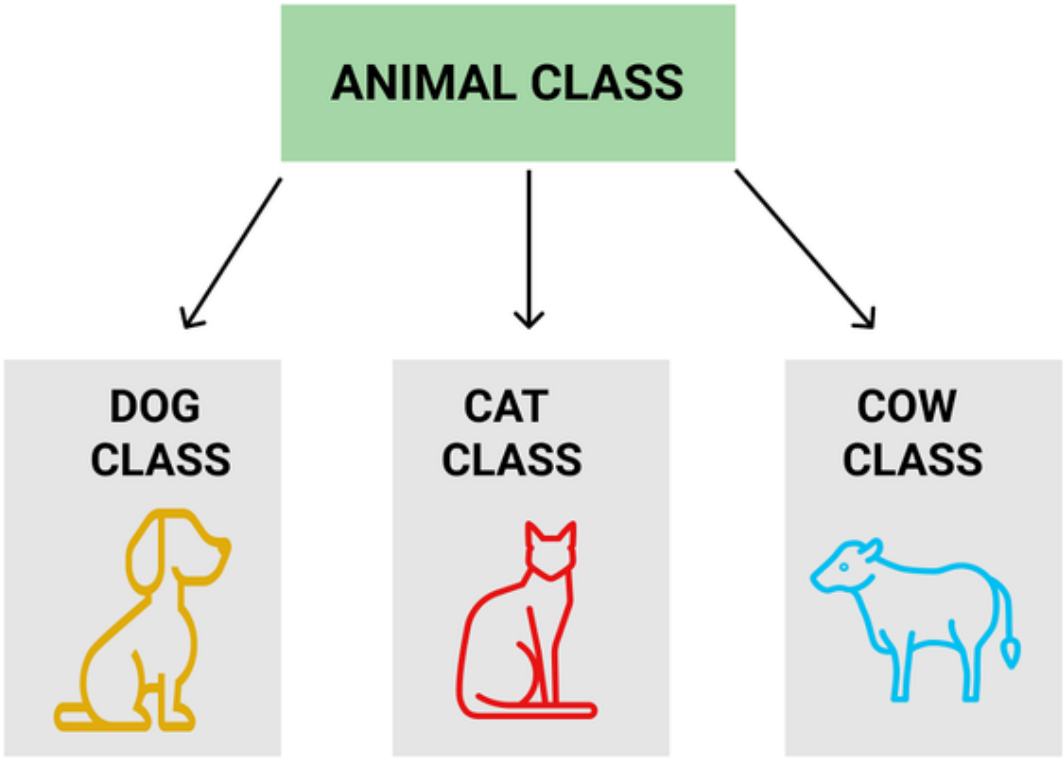
```
Canine
Buddy
Charlie
Max
Feline
Feline
```

Explanation:

- **Class Variable (species):** Shared by all instances of the class. Changing Dog.species affects all objects, as it's a property of the class itself.
- **Instance Variables (name, age):** Defined in the __init__ method. Unique to each instance (e.g., dog1.name and dog2.name are different).
- **Accessing Variables:** Class variables can be accessed via the class name (Dog.species) or an object (dog1.species). Instance variables are accessed via the object (dog1.name).
- **Updating Variables:** Changing Dog.species affects all instances. Changing dog1.name only affects dog1 and does not impact dog2.

3. Inheritance

Inheritance allows a class (child class) to acquire properties and methods of another class (parent class). It supports hierarchical classification and promotes code reuse.



Inheritance

Types of Inheritance:

1. **Single Inheritance:** A child class inherits from a single parent class.
2. **Multiple Inheritance:** A child class inherits from more than one parent class.
3. **Multilevel Inheritance:** A child class inherits from a parent class, which in turn inherits from another class.
4. **Hierarchical Inheritance:** Multiple child classes inherit from a single parent class.
5. **Hybrid Inheritance:** A combination of two or more types of inheritance.

Example: In this example, we create a Dog class and demonstrate single, multilevel and multiple inheritance. We show how child classes can use or extend parent class methods.

```
# Single Inheritance

class Dog:

    def __init__(self, name):

        self.name = name
```

```
    def display_name(self):
        print(f"Dog's Name: {self.name}")

class Labrador(Dog):  # Single Inheritance
    def sound(self):
        print("Labrador woofs")

# Multilevel Inheritance
class GuideDog(Labrador):  # Multilevel Inheritance
    def guide(self):
        print(f"{self.name}Guides the way!")

# Multiple Inheritance
class Friendly:
    def greet(self):
        print("Friendly!")

class GoldenRetriever(Dog, Friendly):  # Multiple Inheritance
    def sound(self):
        print("Golden Retriever Barks")

# Example Usage
lab = Labrador("Buddy")
lab.display_name()
lab.sound()
```

```
guide_dog = GuideDog("Max")

guide_dog.display_name()

guide_dog.guide()


retriever = GoldenRetriever("Charlie")

retriever.display_name()

retriever.greet()

retriever.sound()
```

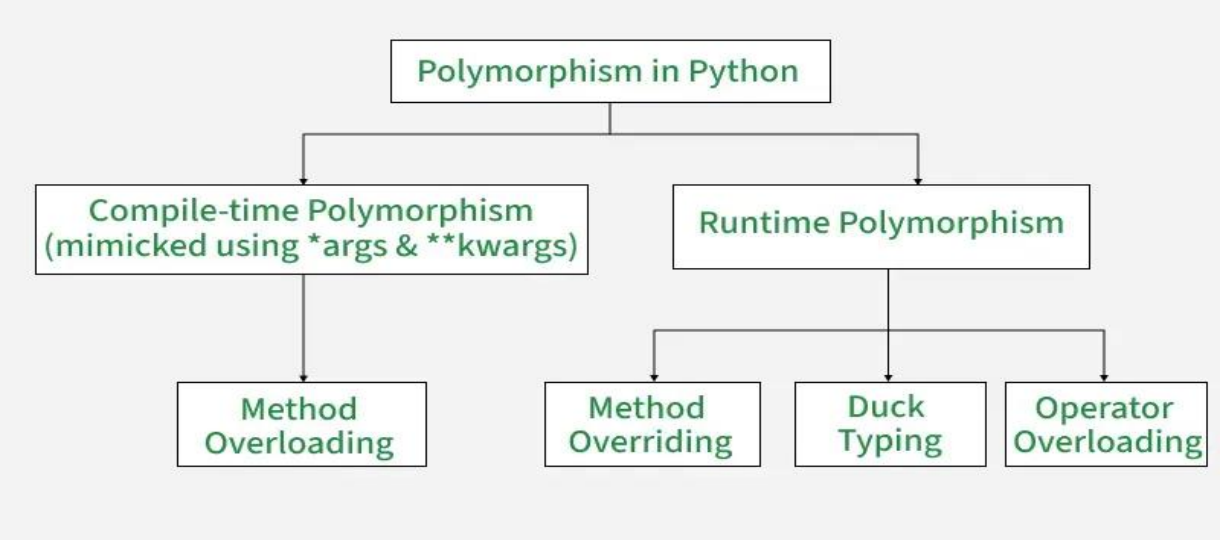
Explanation:

- **Single Inheritance:** Labrador inherits Dog's attributes and methods.
- **Multilevel Inheritance:** GuideDog extends Labrador, inheriting both Dog and Labrador functionalities.
- **Multiple Inheritance:** GoldenRetriever inherits from both Dog and Friendly.

4. Polymorphism

Polymorphism in Python means "same operation, different behavior." It allows functions or methods with the same name to work differently depending on the type of object they are acting upon.

Types of Polymorphism



Polymorphism in Python

1. Compile-Time Polymorphism:

This type of polymorphism is determined during the compilation of the program. It allows methods or operators with the same name to behave differently based on their input parameters or usage. In languages like Java or C++, compile-time polymorphism is achieved through method overloading but it's not directly supported in Python.

In Python:

- True compile-time polymorphism is not supported.
- Instead, Python mimics it using default arguments or *args/**kwargs.
- Operator overloading can also be seen as part of polymorphism, though it is implemented at runtime in Python.

Example:

```
class Calculator:
    def add(self, *args):
        return sum(args)


calc = Calculator()
print(calc.add(5, 10))       # Two arguments
print(calc.add(5, 10, 15))   # Three arguments
print(calc.add(1, 2, 3, 4))  # Any number of arguments
```

Output

15

30

10

2. Run-Time Polymorphism

Run-Time Polymorphism is determined during the execution of the program. It covers multiple forms in Python:

- **Method Overriding:** A subclass redefines a method from its parent class.
- **Duck Typing:** If an object implements the required method, it works regardless of its type.
- **Operator Overloading:** Special methods (__add__, __sub__, etc.) redefine how operators behave for user-defined objects.

Example: In this example, we show run-time polymorphism using method overriding with dog classes and compile-time polymorphism by mimicking method overloading in a calculator class.

```
# Method Overriding

# We start with a base class and then a subclass that "overrides" the speak method.

class Animal:

    def speak(self):

        return "I am an animal."


class Dog(Animal):

    def speak(self):

        return "Woof!"


print(Dog().speak())


# 2 Duck Typing

class Cat:

    def speak(self):

        return "Meow!"


def make_animal_speak(animal):
```

```
    # This function works for both Dog and Cat because they both have a 'speak' method.
    return animal.speak()

print(make_animal_speak(Cat()))
print(make_animal_speak(Dog()))

# 3 Operator Overloading
# We create a simple class that customizes the '+' operator.
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        # This special method defines the behavior of the '+' operator.
        return Vector(self.x + other.x, self.y + other.y)

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(4, 5)
v3 = v1 + v2

print(v3)
```

Explanation:

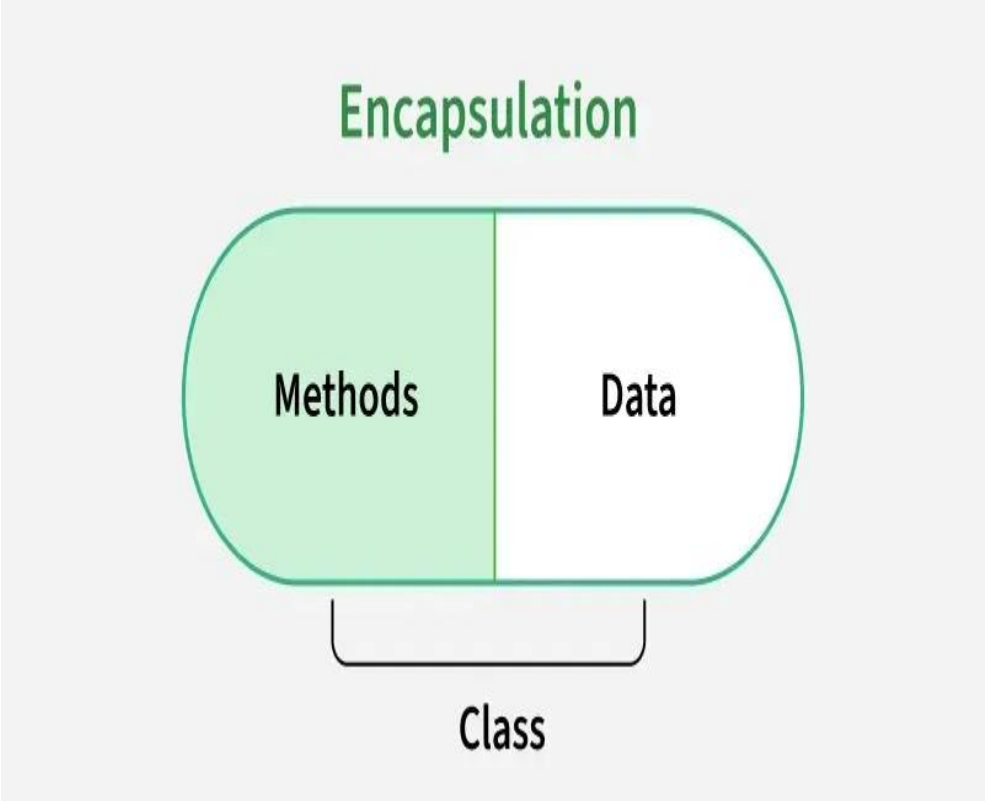
- **Method Overriding:** Dog class overrides the speak method from the parent Animal class. This allows the program to call the speak method on a Dog object and get a specialized "Woof!" response, which is determined at runtime.
- **Duck Typing:** "make_animal_speak" function can accept both a Dog and a Cat object. It doesn't care about their class hierarchy; it only checks if they have a speak method, showcasing Python's flexible typing.
- **Operator Overloading:** "__add__" method within the Vector class is a special method that defines how the + operator behaves for Vector objects. This allows the user to add two vectors using the familiar + symbol, which is a form of syntactic sugar.

5. Encapsulation

Encapsulation is the bundling of data (attributes) and methods (functions) within a class, restricting access to some components to control interactions.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

Encapsulation



Types of Encapsulation:

1. **Public Members**: Accessible from anywhere.
2. **Protected Members**: Accessible within the class and its subclasses.
3. **Private Members**: Accessible only within the class.

Example: In this example, we create a Dog class with public, protected and private attributes. We also show how to access and modify private members using getter and setter methods.

```
class Dog:
    def __init__(self, name, breed, age):
        self.name = name  # Public attribute
        self._breed = breed  # Protected attribute
        self.__age = age  # Private attribute

    # Public method
    def get_info(self):
        return f"Name: {self.name}, Breed: {self._breed}, Age: {self.__age}"

    # Getter and Setter for private attribute
    def get_age(self):
        return self.__age

    def set_age(self, age):
        if age > 0:
            self.__age = age
        else:
            print("Invalid age!")
```

```
# Example Usage
dog = Dog("Buddy", "Labrador", 3)

# Accessing public member
print(dog.name)  # Accessible

# Accessing protected member
print(dog._breed)  # Accessible but discouraged outside the class

# Accessing private member using getter
print(dog.get_age())

# Modifying private member using setter
dog.set_age(5)
print(dog.get_info())
```

Explanation:

- **Public Members:** Easily accessible, such as name.
- **Protected Members**: Used with a single _, such as _breed. Access is discouraged but allowed in subclasses.
- **Private Members:** Used with __, such as __age. Access requires getter and setter methods.

6. Data Abstraction

Abstraction hides the internal implementation details while exposing only the necessary functionality. It helps focus on "what to do" rather than "how to do it."

Types of Abstraction:

- **Partial Abstraction:** Abstract class contains both abstract and concrete methods.
- **Full Abstraction:** Abstract class contains only abstract methods (like interfaces).

Example: In this example, we create an abstract Dog class with an abstract method (sound) and a concrete method. Subclasses implement the abstract method while inheriting the concrete method.

```
from abc import ABC, abstractmethod


class Dog(ABC):  # Abstract Class
    def __init__(self, name):
        self.name = name


    @abstractmethod
    def sound(self):  # Abstract Method
        pass


    def display_name(self):  # Concrete Method
        print(f"Dog's Name: {self.name}")


class Labrador(Dog):  # Partial Abstraction
    def sound(self):
        print("Labrador Woof!")


class Beagle(Dog):  # Partial Abstraction
    def sound(self):
        print("Beagle Bark!")


# Example Usage
dogs = [Labrador("Buddy"), Beagle("Charlie")]
for dog in dogs:
```

```
dog.display_name()  # Calls concrete method
dog.sound()  # Calls implemented abstract method
```

Explanation:

- **Partial Abstraction**: The Dog class has both abstract (sound) and concrete (display_name) methods.
- **Why Use It**: Abstraction ensures consistency in derived classes by enforcing the implementation of abstract methods.