Oops :

. Introduction to Object-Oriented Programming (OOP)

- **What is OOP?**
    - Definition of OOP
    - Key concepts: Objects, Classes, Methods, Attributes
- **Difference between Procedural Programming and OOP**
    - Procedural Programming: Functions and Procedures
    - OOP: Classes and Objects
- **Advantages of OOP**
    - Code Reusability
    - Modularity
    - Encapsulation
    - Inheritance
    - Polymorphism

2. Classes and Objects

- **Defining a Class**
    - Syntax: `class ClassName:`
    - Example: Basic class definition
- **Creating Objects (Instances)**
    - Instantiation: `obj = ClassName()`
    - Example: Creating an object from a class
- **The `__init__()` Method**
    - Constructor Method: `def __init__(self, parameters):`

    - Purpose and Usage
    - Example: Initializing attributes
- **Attributes**
    - **Instance Attributes**: Specific to an object
        - Example: `self.attribute = value`
    - **Class Attributes**: Shared across all instances
        - Example: `ClassName.class_attribute = value`
- **Methods**
    - **Defining Methods**: `def method_name(self, parameters):`
    - **Calling Methods on Objects**: `obj.method_name()`
    - **The `self` Parameter**
        - Role of `self` in accessing instance attributes and methods
        - Example: Method definitions

3. Encapsulation

- **What is Encapsulation?**
    - Hiding Internal Details
    - Exposing Only Necessary Parts
- **Public and Private Attributes**
    - **Public Attributes**: Accessible from outside the class
        - Example: `self.public_attribute`
    - **Private Attributes**: Not accessible directly (use `_` or `__` prefix)
        - Example: `self._private_attribute` or `self.__private_attribute`
- **Getter and Setter Methods**
    - **Getters**: Access private attributes
        - Example: `@property` decorator

    - **Setters**: Modify private attributes
        - Example: `@attribute_name.setter` decorator
- **Python's `@property` Decorator**
    - Creating properties to manage attribute access
    - Example: Defining properties with getter and setter

4. Inheritance

- **What is Inheritance?**
    - Concept of Reusing Code
    - Derived Classes from Existing Classes
- **Types of Inheritance**
    - **Single Inheritance**: One base class, one derived class
    - **Multiple Inheritance**: One class inherits from multiple classes
    - **Multilevel Inheritance**: A class inherits from another derived class
    - **Hierarchical Inheritance**: Multiple classes inherit from a single base class
- **Creating Derived Classes**
    - Syntax: `class DerivedClass(BaseClass):`
    - Example: Derived class creation
- **Accessing Methods and Attributes**
    - Accessing parent class methods and attributes
    - Example: Using `super()` to call parent class methods
- **Method Overriding**
    - Redefining methods in the derived class
    - Example: Overriding a method from the base class
- **Using `super()`**
    - Calling parent class methods within derived class methods

- Example: `super().method_name()`

5. Polymorphism

- **What is Polymorphism?**
    - Definition and Purpose
- **Method Overriding**
    - Implementing the Same Method Differently in Derived Classes
    - Example: Method overriding in child classes
- **Method Overloading (Python's Limitation)**
    - Python does not support method overloading directly
    - **Workaround**: Using default arguments or variable-length arguments
- **Polymorphism with Functions and Objects**
    - Example: Using polymorphism in function arguments and return types

6. Abstraction

- **What is Abstraction?**
    - Hiding Complex Implementation Details
    - Exposing Only Necessary Functionality
- **Abstract Base Classes (ABC)**
    - Using the `abc` module in Python
    - Example: Defining abstract classes and methods
- **Creating Abstract Methods**
    - Enforcing Implementation in Derived Classes
    - Example: Abstract methods in a base class
- **Importance of Abstraction**
    - Benefits in Large-Scale Systems

7. Operator Overloading

- **What is Operator Overloading?**
    - Redefining Built-in Operators for User-Defined Objects
- **Overloading Comparison Operators**
    - `__eq__()`, `__lt__()`, `__gt__()`, etc.
    - Example: Custom comparison logic
- **Overloading Arithmetic Operators**
    - `__add__()`, `__sub__()`, `__mul__()`, etc.
    - Example: Custom arithmetic operations
- **Best Practices and Use Cases**
    - When and why to use operator overloading

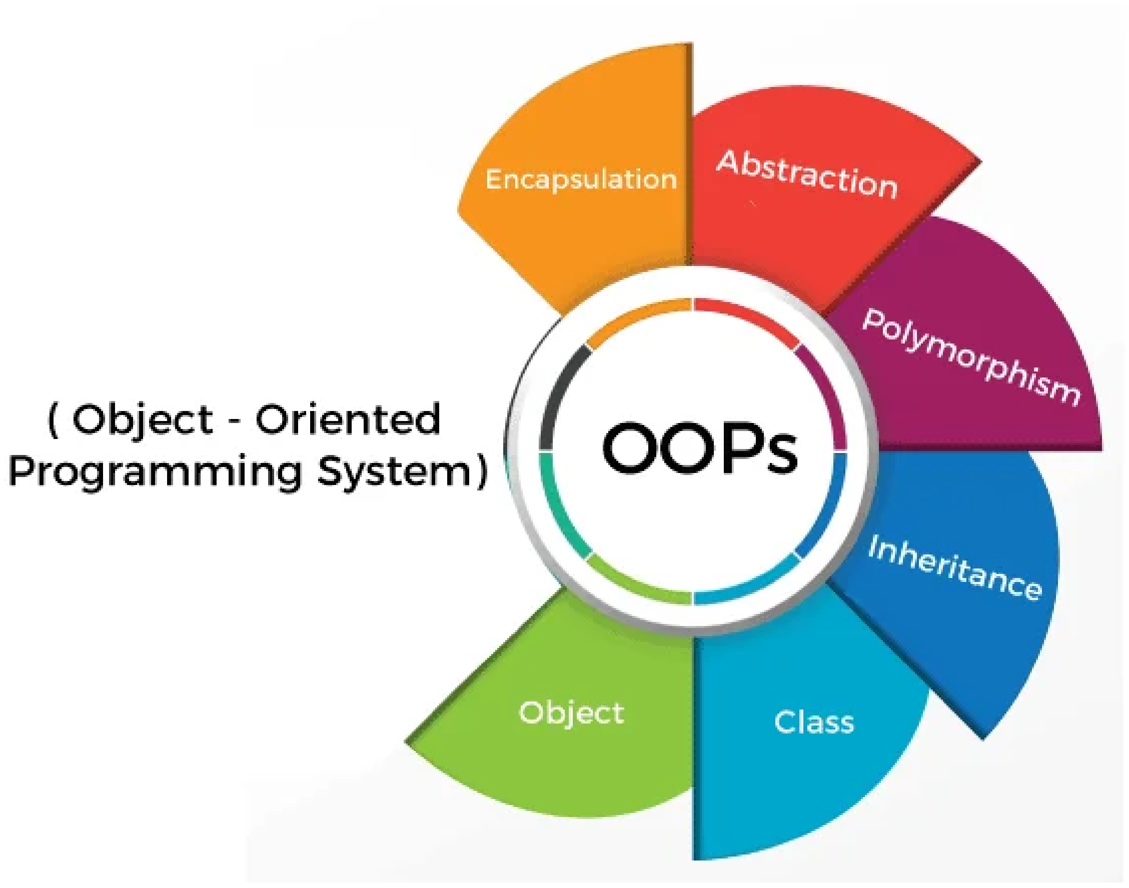
8. Dunder Methods and Magic Methods

- **Common Dunder Methods**
    - `__str__()`: String representation of an object
    - `__repr__()`: Detailed string representation for debugging
    - `__len__()`: Length of an object
    - `__call__()`: Making an object callable
- **Customizing Object Behavior**
    - Enhancing class functionality with magic methods
- **Use Cases for Dunder Methods**
    - Practical examples and scenarios

9. Static Methods, Class Methods, and Instance Methods

- **Instance Methods**
    - Operate on instance data
    - Example: Defining and using instance methods
- **Class Methods**

    - Operate on class data
    - Use `@classmethod` decorator
    - Example: Factory methods or class-level operations
- **Static Methods**
    - Do not operate on instance or class data
    - Use `@staticmethod` decorator
    - Example: Utility functions within a class
- **Property Methods**
    - Manage access to private attributes
    - Use `@property` decorator for getter, setter, and deleter
    - Example: Controlling attribute access and modification



: Why Do We Need OOP in Python?

- **Modularity**: Classes and objects help break down complex problems into smaller, manageable parts.
- **Reusability**: Once a class is written, it can be used multiple times in different parts of the program.
- **Abstraction**: OOP hides unnecessary details from the user, providing a simpler interface.
- **Inheritance**: Inheritance reduces redundancy by allowing new classes to reuse code from existing ones.
- **Polymorphism**: It provides flexibility by allowing different types of objects to be handled using a common interface.
- **Encapsulation**: It protects the data from unintended modifications and makes the code more secure.

Here are some questions based on the importance of OOP concepts in Python to help understand these principles better:

1. Modularity

- **Question**: How does using classes and objects contribute to modularity in a Python program? Can you provide an example where breaking a complex problem into smaller classes improved the design of your solution?
- **Example Answer**: Classes and objects help break down a complex problem into smaller, manageable parts by encapsulating related data and functions within a single unit. For instance, in a project management application, you could have separate classes for `Project`, `Task`, and `User`. Each class manages its own data and methods, making the overall design more modular and easier to manage.

2. Reusability

- **Question**: In what ways does object-oriented programming promote code reusability? Give an example where you reused a class or object in multiple parts of a program.
- **Example Answer**: OOP promotes code reusability by allowing classes to be instantiated multiple times and used in different parts of a program. For example, if you have a `Car` class, you can create multiple car objects (e.g., `car1`, `car2`, `car3`) and use them in various parts of your program, such as in different simulations or for different functionalities, without needing to rewrite the `Car` class code.

3. Abstraction

- **Question**: What is the role of abstraction in OOP, and how does it simplify the interaction with complex systems? Provide an example where abstraction was used to simplify a user interface.
- **Example Answer**: Abstraction in OOP hides the complex implementation details and exposes only the necessary parts to the user. For example, in a `BankAccount` class, the user only interacts with methods like `deposit` and `withdraw`, while the internal calculations and data management are abstracted away. This simplifies the interface and makes the system easier to use.

4. Inheritance

- **Question**: How does inheritance reduce redundancy in code? Can you illustrate with an example where a derived class reused code from a base class?
- **Example Answer**: Inheritance reduces redundancy by allowing a derived class to inherit attributes and methods from a base class. For instance, if you have a base class `Animal` with methods like `eat` and `sleep`, you can create derived classes such as `Dog` and `Cat` that inherit these methods without redefining them. This reduces code duplication and promotes reuse.

5. Polymorphism

- **Question**: How does polymorphism provide flexibility in handling different types of objects? Provide an example where polymorphism allowed you to use a common interface for different object types.

- **Example Answer**: Polymorphism allows different types of objects to be handled through a common interface. For example, if you have a base class `Shape` with a method `draw`, and derived classes like `Circle` and `Rectangle` that implement `draw` differently, you can call `draw` on any `Shape` object regardless of its specific type. This flexibility allows for more generic and adaptable code.

6. Encapsulation

- **Question**: How does encapsulation enhance the security and integrity of data in a program? Provide an example where encapsulation prevented unintended modifications to data.
- **Example Answer**: Encapsulation protects data by restricting direct access to an object's internal state and only allowing modifications through defined methods. For example, in a `Person` class, private attributes like `age` can be accessed and modified only through getter and setter methods. This ensures that the data remains valid and prevents unintended changes from outside the class.

Difference between Object-Oriented and Procedural Oriented Programming

Object-Oriented Programming (OOP)	**Procedural-Oriented Programming (Pop)**
It is a bottom-up approach	It is a top-down approach
Program is divided into objects	Program is divided into functions
Makes use of *Access modifiers* 'public', private', protected'	Doesn't use *Access modifiers*
It is more secure	It is less secure
Object can move freely within member functions	Data can move freely from function to function within programs

It supports inheritance	It does not support inheritance

1. Basic Concept

- **OOP**: Based on the concept of objects and classes. The program is divided into small units called **objects**, which contain both data (attributes) and methods (functions) that operate on the data.
- **Procedural Programming**: Based on the concept of procedure calls. The program is divided into functions or procedures that perform specific tasks. Data is separate from the functions that operate on it.

2. Focus

- **OOP**: Focuses on data abstraction and encapsulation. Objects represent real-world entities, and their internal data is hidden and manipulated only through methods.
- **Procedural Programming**: Focuses on the sequence of actions to be performed (algorithms). Functions are called to operate on global or passed data.

3. Approach

- **OOP**: Follows a **bottom-up approach**, where smaller, reusable objects and classes are built first and then combined into larger systems.
- **Procedural Programming**: Follows a **top-down approach**, where the program starts with a main function and breaks down tasks into subroutines.

4. Structure

- **OOP**: Organizes code into classes and objects. Emphasizes reusability, inheritance, and modularity.
- **Procedural Programming**: Organizes code into procedures (functions) and modules. Emphasizes a clear, linear flow of control.

5. Data Handling

- **OOP**: Encapsulates data within objects. Objects control access to their own data via methods. Data is protected and hidden using access specifiers (private, public).

- **Procedural Programming**: Data is generally passed between procedures or accessed globally. There is less control over data, and it is often shared openly across functions.

6. Key Principles

- **OOP**: Supports core principles like **Encapsulation, Inheritance, Polymorphism**, and **Abstraction**.
- **Procedural Programming**: Does not inherently support these principles but focuses more on structured programming and functional decomposition.

7. Reusability

- **OOP**: Promotes reusability through **inheritance** and **polymorphism**. Classes can be reused across programs, and objects can be extended or modified easily.
- **Procedural Programming**: Reusability is achieved through functions and modules, but extending or modifying the code can be more complex compared to OOP.

8. Example

- **OOP**: Creating a `Car` class with attributes like `speed` and methods like `accelerate()` and `brake()`. You create instances of the `Car` class (objects) to represent different cars.
- **Procedural Programming**: Writing a function `drive_car()` that takes parameters like speed and operates on those parameters directly without the concept of objects or classes.

9. Maintainability

- **OOP**: Easier to maintain because of modularity and the ability to extend classes without modifying existing code.
- **Procedural Programming**: Can become harder to maintain as the program grows, since modifications often require changing multiple functions that share or operate on the same data.

10. Real-World Modeling

- **OOP**: Closer to real-world modeling, as objects represent real-world entities, making it easier to design systems like GUI applications, simulations, or games.
- **Procedural Programming**: Better suited for simpler, sequential tasks where operations need to be performed in a specific order, such as basic calculations or file handling.

Comparison Table

Aspect	OOP	Procedural Programming
Basic Unit	Objects and Classes	Functions and Procedures
Approach	Bottom-Up	Top-Down
Focus	Data and Objects	Functions and Logic
Data Handling	Encapsulated within Objects	Passed between functions or accessed globally
Reusability	Achieved through Inheritance and Polymorphism	Achieved through Function and Module Reuse
Real-World Mapping	Closer to Real-World Entities	Focuses on Task-Oriented Problems
Example	Creating objects from classes	Writing sequential functions to perform tasks
Maintainability	Easier to maintain with larger systems	Harder to maintain as codebase grows

Class

A **class** is a blueprint for creating objects. It defines a set of attributes and methods that the created objects will have.

- **Attributes**: Variables that hold data related to the class.
- **Methods**: Functions defined within a class that can operate on its attributes.

Object

An **object** is an instance of a class. It is a specific realization of the class with actual values for the attributes.

Creating a Class

To create a class in Python, you use the `class` keyword. Here's a basic structure:

```
pythonCopy code
class ClassName:
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2

    def method1(self):
        # Code for method1
        pass

    def method2(self):
        # Code for method2
        pass
```

Creating and Using Objects

Once a class is defined, you can create objects from it by calling the class as if it were a function. You access attributes and methods using the dot notation (`.`).

Examples

Example 1: Basic Class and Object

```
pythonCopy code
# Define the class
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return f"{self.name} says Woof!"
    def get_age(self):
        return f"{self.name} is {self.age} years old."

# Create an object of the class
my_dog = Dog(name="Buddy", age=3)

# Access attributes and methods
print(my_dog.name)         # Output: Buddy
print(my_dog.bark())       # Output: Buddy says Woof!
print(my_dog.get_age())    # Output: Buddy is 3 years old.
```

Example 2: Class with Methods and Attributes

```
pythonCopy code
# Define the class
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

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

    def perimeter(self):
        return 2 * (self.width + self.height)

# Create an object of the class
rect = Rectangle(width=5, height=3)

# Access attributes and methods
print(f"Width: {rect.width}")           # Output: Width: 5
print(f"Height: {rect.height}")         # Output: Height: 3
print(f"Area: {rect.area()}")           # Output: Area: 15
print(f"Perimeter: {rect.perimeter()}")  # Output: Perimeter:
16
```

1. Constructor Overview

In Python, the constructor method is called `__init__()`. It is automatically called when a new instance of a class is created. The purpose of the constructor is to initialize the object's attributes with values.

2. Non-Parameterized Constructor

A **non-parameterized constructor** (also known as a default constructor) does not take any arguments other than `self`. It is used to initialize an object with default values.

Purpose

- To provide default values for attributes when an object is created without specific values.

Example

```
pythonCopy code
class Car:
    def __init__(self):
        self.make = "Unknown"
        self.model = "Unknown"
        self.year = 2000

# Create an object using the non-parameterized constructor
default_car = Car()

# Access attributes
print(default_car.make)  # Output: Unknown
print(default_car.model) # Output: Unknown
print(default_car.year)  # Output: 2000
```

3. Parameterized Constructor

A **parameterized constructor** takes one or more arguments in addition to `self`. It allows for the initialization of an object with specific values provided at the time of object creation.

Purpose

- To initialize an object with specific values supplied during object creation, making the object more flexible and customizable.

Example

```
pythonCopy code
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

# Create an object using the parameterized constructor
my_car = Car(make="Toyota", model="Camry", year=2023)

# Access attributes
print(my_car.make)  # Output: Toyota
print(my_car.model) # Output: Camry
print(my_car.year)  # Output: 2023
```

4. Why We Need Constructors

- **Initialization**: Constructors initialize an object's attributes to ensure that the object starts in a valid state.

- **Flexibility**: Parameterized constructors allow creating objects with different states, making the class more versatile.
- **Encapsulation**: By using constructors, you encapsulate the initialization logic within the class, ensuring consistency and reducing errors.

Comparison

- **Parameterized Constructor**: Allows initialization with specific values provided at the time of object creation. It is useful when you need to set initial values based on input data.
- **Non-Parameterized Constructor**: Initializes attributes with default values. It is useful when you want to provide default settings or when initialization does not depend on input data.

The `self` keyword in Python is crucial for working with object-oriented programming (OOP). It is used within class methods, including constructors, to refer to the instance of the class on which the method is being called. Here's a detailed explanation of its use and purpose:

Purpose of `self` Keyword

1. **Instance Reference**: `self` refers to the instance of the class. It allows access to the attributes and methods of the instance. Without `self`, you wouldn't be able to access or modify instance-specific data.
2. **Attribute Access**: It is used to access or set instance attributes (variables) within methods. This ensures that each instance maintains its own state.
3. **Method Access**: `self` is used to call other methods within the same class. This is important for maintaining the internal consistency of an object.

4. **Initialization**: In constructors (`__init__`), `self` allows the initialization of instance attributes, ensuring that each object has its own set of data.

How `self` Is Used

- **In Constructors**: When defining the `__init__()` method (constructor), `self` allows you to initialize the instance attributes with values passed to the constructor.
- **In Methods**: `self` is used to refer to instance variables and other methods within the class. This way, methods can operate on the instance's data and interact with each other.

Example

Here's a detailed example illustrating the use of `self` in constructors and methods:

```
pythonCopy code
class Person:
    def __init__(self, name, age):
        # `self` is used to initialize instance attributes
        self.name = name
        self.age = age

    def greet(self):
        # `self` is used to access instance attributes within
methods
        return f"Hello, my name is {self.name} and I am {self.
age} years old."

    def have_birthday(self):
        # `self` is used to modify instance attributes
        self.age += 1
        return f"Happy Birthday! Now I am {self.age} years ol
d."

# Create an instance of the Person class
person = Person("Alice", 30)
```

```
# Access method using `self` to get instance data
print(person.greet())  # Output: Hello, my name is Alice and I
am 30 years old.

# Modify instance data using `self`
print(person.have_birthday())  # Output: Happy Birthday! Now I
am 31 years old.
```

Key Points

- **Instance-Specific**: Each object created from the class has its own set of instance attributes. `self` ensures that the methods and attributes accessed belong to the particular object.
- **Self in Methods**: When calling `self` methods, it is used to access other methods or attributes of the class. For example, `self.greet()` or `self.have_birthday()`.
- **Consistency**: Using `self` ensures that attributes and methods belong to the instance, not the class itself. This helps in maintaining the correct state and behavior of each object.

Why We Use `self`

- **Clarity**: It makes it clear that the attributes and methods belong to the instance, distinguishing them from local variables or other class-level elements.
- **Encapsulation**: `self` provides a way to access and modify instance-specific data, keeping the class's data and behavior encapsulated.
- **Flexibility**: It allows methods to operate on instance-specific data and maintain the object's state.

Attributes (Variables in OOP)

Attributes are variables that belong to a class or an object. Attributes are used to store information about the object.

Instance Attributes vs Class Attributes

- **Instance Attributes**: These are unique to each object. They are defined within the `__init__()` method or another method of the class and are accessed using the `self` keyword.
- **Class Attributes**: These are shared across all instances of the class. They are defined outside of any method within the class and are accessed using the class name or the object.

Class Attributes

Class attributes are shared across all instances of a class. They are defined within the class but outside of any instance methods. All instances of the class have access to class attributes.

Accessing Class Attributes

You can access class attributes directly through the class name or through an instance of the class.

Modifying Class Attributes

Modifications to class attributes affect all instances of the class unless overridden by instance attributes.

Example

```
class Dog:
    species = "Canis familiaris"  # Class attribute

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

# Accessing class attribute
print(Dog.species)  # Output: Canis familiaris

# Creating instances
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)

# Accessing class attribute through instances
print(dog1.species)  # Output: Canis familiaris
print(dog2.species)  # Output: Canis familiaris

# Modifying class attribute
Dog.species = "Canis lupus familiaris"

print(dog1.species)  # Output: Canis lupus familiaris
print(dog2.species)  # Output: Canis lupus familiaris
```

Instance Attributes

Instance attributes are unique to each instance of a class. They are defined within the `__init__()` method and are specific to the individual instance.

Accessing Instance Attributes

Instance attributes are accessed using the instance of the class.

Modifying Instance Attributes

You can modify instance attributes directly through the instance.

Adding New Instance Attributes

You can add new attributes to an instance after it has been created.

Example

```
pythonCopy code
class Car:
    def __init__(self, make, model, year):
        self.make = make  # Instance attribute
        self.model = model  # Instance attribute
        self.year = year  # Instance attribute

# Creating an instance
car1 = Car("Toyota", "Corolla", 2020)

# Accessing instance attributes
print(car1.make)   # Output: Toyota
print(car1.model)  # Output: Corolla
print(car1.year)   # Output: 2020

# Modifying instance attributes
car1.year = 2021
print(car1.year)   # Output: 2021

# Adding new instance attributes
car1.color = "Blue"
print(car1.color)  # Output: Blue
```

Summary

- **Class Attributes**:
    - Defined at the class level.
    - Shared among all instances.
    - Can be accessed or modified using the class name or instances.
    - Changes affect all instances unless overridden by instance attributes.
- **Instance Attributes**:
    - Defined within the `__init__()` method.

- Unique to each instance.
- Accessed and modified through the instance.
- New attributes can be added to an instance at runtime.

Instance Methods

Instance methods are the most common methods in a class. They operate on instance data and can access and modify instance attributes. They are defined with at least one parameter, typically named `self`, which refers to the instance on which the method is called.

Definition and Access

Instance methods are defined within a class and accessed through an instance of the class. They can modify the instance's attributes and call other instance methods.

Example

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

    def greet(self):
        return f"Hello, my name is {self.name} and I am {self.
age} years old."

    def have_birthday(self):
        self.age += 1  # Modifying an instance attribute
        return f"Happy Birthday, {self.name}! You are now {sel
f.age} years old."

# Creating an instance
person1 = Person("Alice", 30)
```

```
# Accessing and using instance methods
print(person1.greet())          # Output: Hello, my name is Al
ice and I am 30 years old.
print(person1.have_birthday())  # Output: Happy Birthday, Alic
e! You are now 31 years old.
```

Class Methods

Class methods operate on the class itself, rather than on instances of the class. They are defined using the `@classmethod` decorator and take a `cls` parameter, which refers to the class, not the instance. They are used to access or modify class-level data, such as class attributes.

Definition and Access

Class methods are defined with the `@classmethod` decorator and can be called using the class name or an instance. They are often used for factory methods or alternative constructors.

Example

```
class Car:
    # Class attribute
    number_of_wheels = 4

    def __init__(self, make, model):
        self.make = make
        self.model = model

    @classmethod
    def display_wheels(cls):
        return f"Cars typically have {cls.number_of_wheels} wh
eels."
```

```
    @classmethod
    def set_wheels(cls, wheels):
        cls.number_of_wheels = wheels

# Accessing class methods
print(Car.display_wheels())  # Output: Cars typically have 4 w
heels.

# Modifying class attribute via class method
Car.set_wheels(6)
print(Car.display_wheels())  # Output: Cars typically have 6 w
heels.

# Creating an instance
car1 = Car("Toyota", "Camry")
print(car1.display_wheels())  # Output: Cars typically have 6
wheels.
```

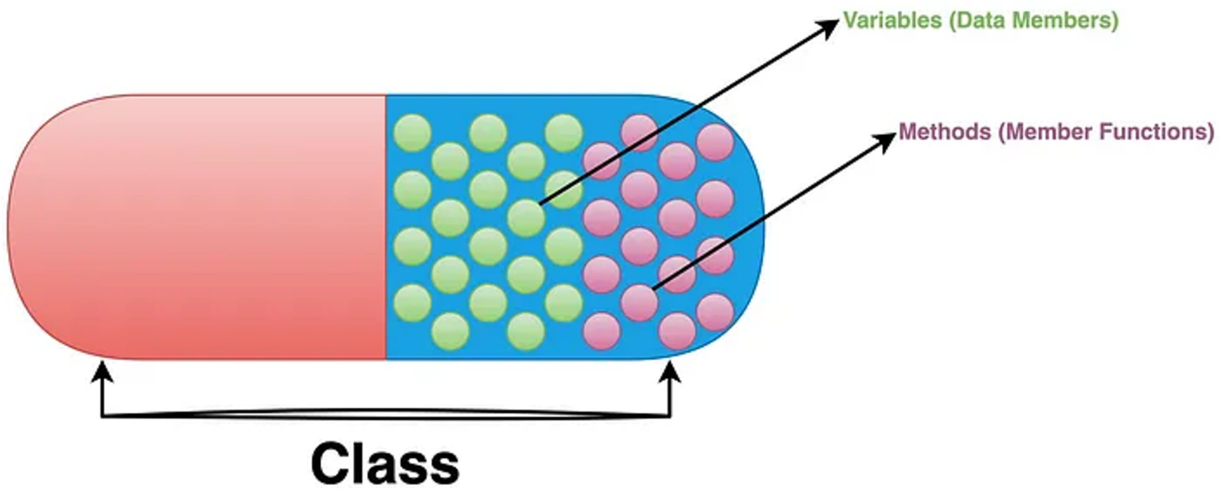
Summary

- **Instance Methods**:
    - Operate on instance data.
    - Defined with `self` as the first parameter.
    - Can access and modify instance attributes.
    - Example usage: Accessing or modifying data specific to a particular instance.
- **Class Methods**:
    - Operate on class data.
    - Defined with `@classmethod` decorator and `cls` as the first parameter.
    - Can access and modify class attributes.

- Example usage: Creating factory methods or modifying data shared across all instances.

Summary Table:

Access Type	Can be Accessed via Object (Instance)	Can be Accessed via Class	Explanation
Instance Method	Yes	No (unless manually passing an instance)	Instance methods require `self`, which is not available via class.
Class Method	Yes	Yes	Class methods require `cls`, and can be accessed via both.
Class Variable	Yes	Yes	Class variables are shared by all instances and the class itself.
Instance Variable	Yes	No	Instance variables belong to a specific instance.
Static Method	Yes	Yes	Static methods can be called without `self` or `cls`.



Encapsulation in Python

Encapsulation is a fundamental concept in Object-Oriented Programming (OOP) that involves bundling the data (attributes) and the methods (functions) that operate on the data into a single unit (class). Encapsulation restricts access to certain details of an object and only exposes the necessary parts to the outside world. This helps to protect the integrity of the data and provides a clear interface for interacting with the object.

What is Encapsulation?

Encapsulation is the mechanism of restricting access to certain components of an object and making some of its data and methods private to control how they are accessed or modified. It hides the internal implementation details and allows the user to interact with the object through a defined interface.

Hiding Internal Details and Exposing Necessary Parts

- **Public Attributes/Methods**: These are accessible from anywhere in the program.
- **Private Attributes/Methods**: These are restricted to the class and are not directly accessible outside the class.

Encapsulation ensures that the object's internal state cannot be directly altered from outside, which prevents unintended interference and misuse.

Importance of Encapsulation

1. **Data Protection**: Encapsulation protects the internal state of an object from unintended or unauthorized changes by restricting access.
2. **Data Hiding**: It allows hiding the internal implementation details of an object, exposing only what is necessary through a well-defined interface.
3. **Modularity**: By encapsulating data and methods, you can compartmentalize code, making it easier to understand, maintain, and debug.
4. **Controlled Access**: You can control the level of access to data by defining getter and setter methods, ensuring that data is accessed and modified safely.

Private and Public Attributes

Public Attributes and Methods

In Python, attributes and methods are **public** by default. This means they can be accessed from anywhere in the program, both inside and outside the class.

Example:

```
pythonCopy code
class Car:
    def __init__(self, model, year):
        self.model = model  # Public attribute
        self.year = year    # Public attribute

    def get_car_info(self):  # Public method
        return f"{self.model}, {self.year}"

car = Car("Toyota", 2022)
print(car.model)  # Accessing public attribute
print(car.get_car_info())  # Calling public method
```

Output:

```
yamlCopy code
Toyota
Toyota, 2022
```

Private Attributes and Methods (Using `_` and `__`)

Python does not have true private variables, but it uses naming conventions to create private-like attributes and methods.

- **Single Underscore (`_`)**: This is a convention used to indicate that an attribute or method is intended for internal use only (i.e., it is private by convention, but not enforced by the language). It is still accessible, but developers are discouraged from using it directly.
- **Double Underscore (`__`)**: This triggers name mangling, which makes it harder (but not impossible) to access the attribute or method from outside the class.

Example with Single Underscore:

```
pythonCopy code
class Laptop:
    def __init__(self, brand, price):
        self._brand = brand  # Private by convention
        self._price = price  # Private by convention

    def get_price(self):
        return f"The price of the {self._brand} laptop is ${se
lf._price}"

laptop = Laptop("Dell", 1200)
print(laptop._price)  # Technically accessible, but discourage
d
```

Example with Double Underscore:

```
pythonCopy code
class Laptop:
    def __init__(self, brand, price):
        self.__brand = brand  # Private (name-mangled)
        self.__price = price  # Private (name-mangled)

    def get_price(self):
        return f"The price of the {self.__brand} laptop is ${s
elf.__price}"

laptop = Laptop("HP", 1500)
# print(laptop.__price)  # This would raise an AttributeError
print(laptop._Laptop__price)  # Accessing via name mangling
```

Output:

```
yamlCopy code
1500
```

In the second example, the double underscore makes the attribute harder to access by outsiders, but it is still possible through name mangling (`_ClassName__attribute`).

Getter and Setter Methods

Encapsulation often uses getter and setter methods to provide controlled access to an object's attributes.

- **Getter Method**: Used to retrieve the value of a private attribute.
- **Setter Method**: Used to set or modify the value of a private attribute while enforcing any necessary validation or constraints.

Example:

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

    # Getter method
    def get_balance(self):
        return self.__balance

    # Setter method
    def set_balance(self, amount):
        if amount >= 0:
            self.__balance = amount
        else:
            print("Invalid amount. Balance cannot be negativ
e.")

account = BankAccount(1000)
print(account.get_balance())  # Output: 1000
account.set_balance(1200)
print(account.get_balance())  # Output: 1200
```

Python's `@property` Decorator for Property Methods

The `@property` decorator provides a Pythonic way to define getter, setter, and deleter methods for attributes. It allows you to access attributes like regular variables while still enforcing encapsulation.

Using `@property` for Getter Methods

Example:

```
pythonCopy code
class Employee:
    def __init__(self, name, salary):
        self.__name = name
        self.__salary = salary

    @property
    def salary(self):
        return self.__salary

employee = Employee("John", 5000)
print(employee.salary)  # Output: 5000
```

Here, `employee.salary` looks like accessing a regular attribute, but it calls the getter method defined by `@property`.

Using `@property.setter` for Setter Methods

Example:

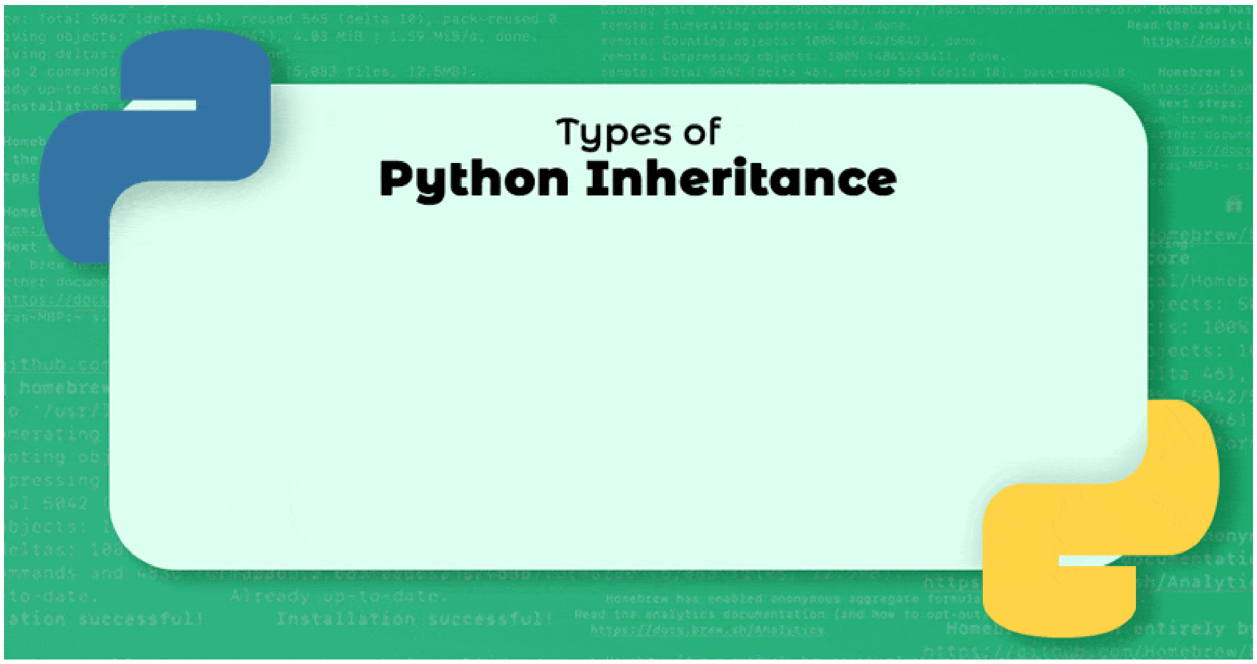
```
pythonCopy code
class Employee:
    def __init__(self, name, salary):
        self.__name = name
        self.__salary = salary

    @property
    def salary(self):
        return self.__salary

    @salary.setter
    def salary(self, amount):
        if amount >= 0:
            self.__salary = amount
        else:
            print("Invalid salary. It cannot be negative.")

employee = Employee("John", 5000)
employee.salary = 6000  # Calls the setter method
```

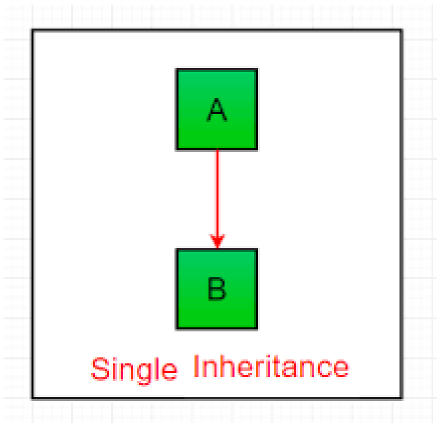
```
print(employee.salary)   # Output: 6000
```

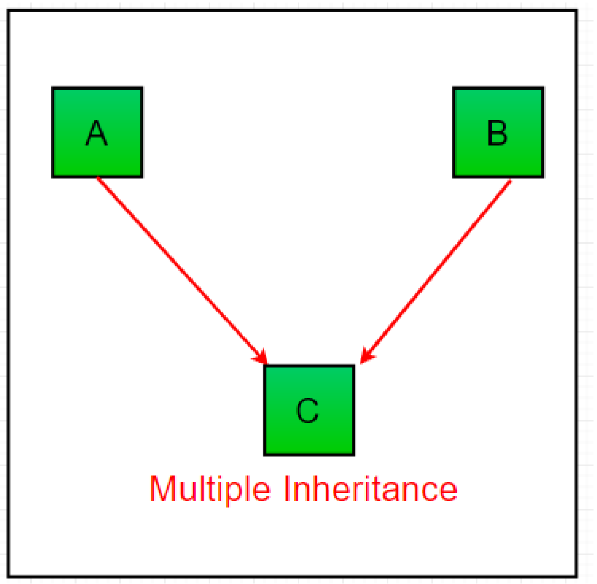
Using `@property.deleter` for Deleter Methods

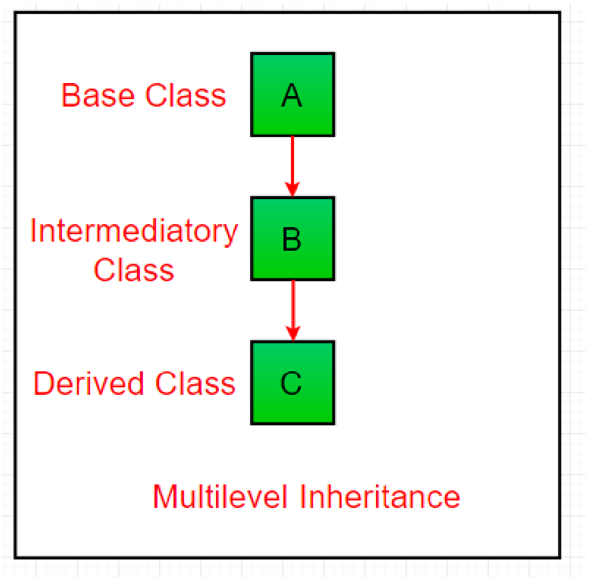
Example:

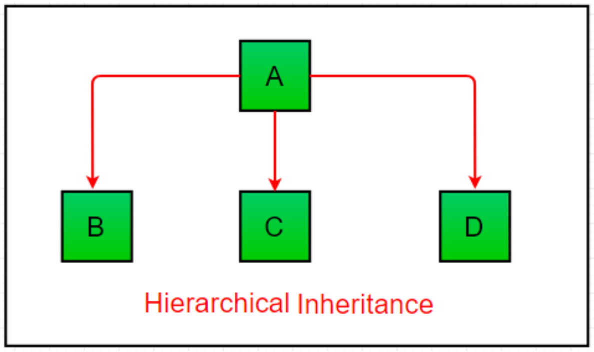
```
pythonCopy code
class Employee:
    def __init__(self, name, salary):
        self.__name = name
        self.__salary = salary

    @property
    def salary(self):
        return self.__salary

    @salary.deleter
    def salary(self):
        del self.__salary

employee = Employee("John", 5000)
del employee.salary  # Deletes the salary attribute
```


Types of
Python Inheritance


A
B
Single Inheritance


A
B
C
Multiple Inheritance





4. Inheritance in Python

Inheritance is one of the core concepts in Object-Oriented Programming (OOP) that allows a new class (derived or child class) to inherit attributes and methods from an existing class (base or parent class). This promotes code reusability and hierarchical relationships between classes.

What is Inheritance?

Inheritance is the mechanism by which one class can inherit the attributes and methods of another class. This allows you to create a hierarchy of classes that share functionality and reduce code duplication.

- **Base Class (Parent Class)**: The class whose properties and methods are inherited.
- **Derived Class (Child Class)**: The class that inherits from another class.

Concept of Reusing Code through Inheritance

Inheritance allows you to reuse code by creating a new class that inherits the behavior of an existing class. Instead of writing all the methods and attributes again, you can create a class that automatically inherits from another class and extends or modifies its functionality.

Types of Inheritance in Python

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 class inherits from a class that is already a derived class.
4. **Hierarchical Inheritance**: Multiple child classes inherit from a single parent class.

1. Single Inheritance

One class inherits from a single base class.

```
pythonCopy code
class Animal:  # Base class
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound"

class Dog(Animal):  # Derived class from Animal (Single Inheri
tance)
    def speak(self):
        return f"{self.name} barks"

# Example usage:
dog = Dog("Buddy")
print(dog.speak())  # Output: Buddy barks
```

2. Multiple Inheritance

A class can inherit from more than one base class.

```
pythonCopy code
class Animal:  # Base class
    def __init__(self, name):
        self.name = name

class Dog:
    def bark(self):
        return f"{self.name} barks"

class Cat:
    def meow(self):
        return f"{self.name} meows"

class CatDog(Animal, Dog, Cat):  # Inherits from both Dog and
Cat (Multiple Inheritance)
    def speak(self):
        return f"{self.name} can both bark and meow"

# Example usage:
cat_dog = CatDog("Tommy")
print(cat_dog.speak())  # Output: Tommy can both bark and meow
```

3. Multilevel Inheritance

A class inherits from another derived class, which in turn inherits from a base class.

```
pythonCopy code
class Animal:  # Base class
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound"
```

```
class Dog(Animal):  # Inherits from Animal (First level)
    def speak(self):
        return f"{self.name} barks"

class Puppy(Dog):  # Inherits from Dog (Second level)
    def speak(self):
        return f"{self.name} barks in a cute way"

# Example usage:
puppy = Puppy("Max")
print(puppy.speak())  # Output: Max barks in a cute way
```

4. Hierarchical Inheritance

Multiple classes inherit from the same base class.

```
pythonCopy code
class Animal:  # Base class
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound"

class Dog(Animal):  # Inherits from Animal (Hierarchical Inher
itance)
    def speak(self):
        return f"{self.name} barks"

class Cat(Animal):  # Another class inherits from Animal
    def speak(self):
        return f"{self.name} meows"

# Example usage:
```

```
dog = Dog("Buddy")
cat = Cat("Whiskers")

print(dog.speak())  # Output: Buddy barks
print(cat.speak())  # Output: Whiskers meows
```

5. Hybrid Inheritance

This is a combination of more than one type of inheritance.

```
pythonCopy code
class Animal:  # Base class
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound"

class Dog(Animal):  # Inherits from Animal (Multilevel & Hiera
rchical)
    def speak(self):
        return f"{self.name} barks"

class Cat(Animal):  # Inherits from Animal (Hierarchical)
    def speak(self):
        return f"{self.name} meows"

class Puppy(Dog):  # Inherits from Dog (Multilevel Inheritanc
e)
    def speak(self):
        return f"{self.name} barks softly"

# Example usage:
puppy = Puppy("Bella")
print(puppy.speak())  # Output: Bella barks softly
```

```
cat = Cat("Tom")
print(cat.speak())  # Output: Tom meows
```

Summary of Each Type:

- **Single Inheritance**: One class inherits from one parent class.
- **Multiple Inheritance**: One class inherits from multiple parent classes.
- **Multilevel Inheritance**: A class inherits from another derived class.
- **Hierarchical Inheritance**: Multiple classes inherit from a single base class.
- **Hybrid Inheritance**: A combination of multiple inheritance types.

Method Overriding

Method overriding allows a derived class to provide a specific implementation of a method that is already defined in its parent class. This is useful when the child class needs to modify the behavior of a parent class method.

Example:

```
pythonCopy code
class Parent:
    def show_message(self):
        return "Message from the parent class"

class Child(Parent):
    def show_message(self):  # Overriding the parent class met
hod
        return "Message from the child class"

child = Child()
print(child.show_message())  # Output: Message from the child
```

```
class
```

In this example, the `show_message()` method in the `Child` class overrides the same method in the `Parent` class.

Using `super()`

The `super()` function allows you to call a method from the parent class within the derived class. This is commonly used in method overriding when you want to extend the behavior of the parent class method rather than completely replacing it.

Example:

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

    def speak(self):
        return f"{self.name} makes a sound"

class Dog(Animal):
    def speak(self):
        parent_message = super().speak()  # Calling the parent
class method
        return f"{parent_message} and barks"

dog = Dog("Buddy")
print(dog.speak())  # Output: Buddy makes a sound and barks
```

Access Parent Class Methods:

- **Reason**: To call a method defined in the parent class from within the child class.

- **Example**: Overriding a method in a child class but still needing to call the parent class's implementation.

2. Extend Parent Class Methods:

- **Reason**: To extend or enhance the functionality of a parent class's method while retaining its core behavior.
- **Example**: Adding additional logging or processing in the child class's method but keeping the parent class's behavior.

3. Ensure Proper Initialization:

- **Reason**: To ensure that the parent class's `__init__` method is called to initialize attributes properly.
- **Example**: When initializing a child class, `super().__init__()` ensures that the parent class's attributes are set up.

4. Support Multiple Inheritance:

- **Reason**: To handle multiple inheritance scenarios where a class inherits from more than one parent class.
- **Example**: Using `super()` in a method to ensure that methods from all parent classes are called in the correct order.

5. Avoid Direct Parent Class Name:

- **Reason**: To avoid hardcoding the parent class name, which makes the code more maintainable and adaptable.
- **Example**: Using `super()` instead of `ParentClass.method(self)` to call the parent class's method.

6. Facilitate Method Resolution Order (MRO):

- **Reason**: To leverage Python's Method Resolution Order in multiple inheritance scenarios to ensure correct method chaining.
- **Example**: In a complex class hierarchy, `super()` helps follow the MRO to call the appropriate method from the parent classes.

Example of Using `super()`:

Example 1: Extending Functionality

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

    def speak(self):
        return f"{self.name} makes a sound"

class Dog(Animal):
    def speak(self):
        # Call parent class method using super()
        parent_sound = super().speak()
        return f"{parent_sound} but also {self.name} barks"
```

Example 2: Ensuring Proper Initialization

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

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Ensure Animal's __init__ is
called
        self.breed = breed
```

Example 3: Handling Multiple Inheritance

```
pythonCopy code
class Parent1:
    def method(self):
        print("Parent1 method")

class Parent2:
    def method(self):
        print("Parent2 method")

class Child(Parent1, Parent2):
    def method(self):
        super().method()  # Calls Parent1's method
        super(Parent1, self).method()  # Calls Parent2's metho
d
```

What is Polymorphism in Python?

Polymorphism is an important concept in object-oriented programming that allows objects of different classes to be treated as objects of a common parent class. In Python, polymorphism allows the same method or function to behave differently based on the object that calls it or the type of data it operates on.

The term **Polymorphism** comes from two Greek words:

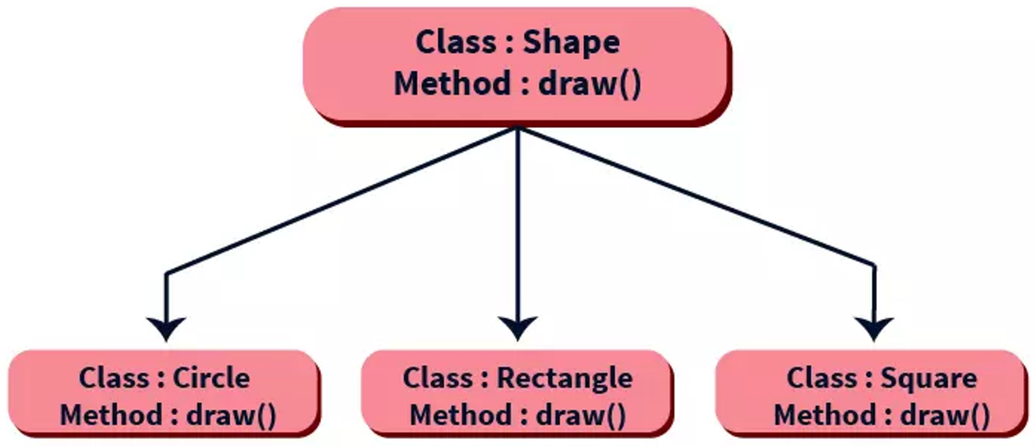
- **Poly** meaning many
- **Morphism** meaning forms

So, polymorphism means "many forms." This means that different classes can implement methods with the same name but behave differently depending on the class's specific context.

Key Points of Polymorphism:

1. **Method Polymorphism**: Allows different objects to call the same method, and the method behaves differently based on the object.
2. **Operator Polymorphism**: Enables operators to behave differently with different types of operands (operator overloading).
3. **Function Polymorphism**: Allows functions to accept different data types and behave differently based on the input.
4. **Polymorphism with Inheritance**: Child classes can override methods from a parent class, implementing their own functionality, and demonstrate polymorphism.

Let's break down polymorphism in Python with examples.



1. Operator Overloading Polymorphism

In Python, operators like `+`, `*`, etc., exhibit polymorphism. The behavior of the operator changes depending on the type of data it is working with.

Example:

```
pythonCopy code
# Using the + operator with different data types
a = 5
b = 10
print(a + b)  # Output: 15 (numerical addition)

str1 = "Hello"
str2 = "World"
print(str1 + " " + str2)  # Output: Hello World (string concat
enation)
```

In this example, the `+` operator behaves differently depending on whether it is adding numbers or concatenating strings. This is an example of operator overloading polymorphism.

2. Function Polymorphism

Functions in Python can accept arguments of various types and still work as expected. This is another form of polymorphism.

Example:

```
pythonCopy code
# Using the len() function on different data types
print(len("Hello"))  # Output: 5 (length of the string)
print(len([1, 2, 3]))  # Output: 3 (length of the list)
print(len({"name": "Alice", "age": 25}))  # Output: 2 (number
of dictionary items)
```

Here, the `len()` function works with different data structures (string, list, dictionary) and gives the correct result depending on the type of data. This is an example of function polymorphism.

3. Polymorphism with Class Methods

You can create your own classes and methods that exhibit polymorphism. Two or more classes can implement the same method name, and the method will behave differently depending on the class from which it is called.

Example:

```
pythonCopy code
class Liquid:
    def __init__(self, name, formula):
        self.name = name
        self.formula = formula

    def info(self):
        print(f"I am Liquid Form. I am {self.name}, and my for
mula is {self.formula}")

    def property(self):
        print("I am a clear and light form.")
```

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

    def info(self):
        print(f"I am Solid Form. I am {self.name}, and my form
ula is {self.formula}")

    def property(self):
        print("I can be transparent or opaque.")

# Polymorphism with class methods
liquid_1 = Liquid("Water", "H2O")
solid_1 = Solid("Ice", "H2O")

for material in (liquid_1, solid_1):
    material.info()
    material.property()
```

Output:

```
cssCopy code
I am Liquid Form. I am Water, and my formula is H2O
I am a clear and light form.
I am Solid Form. I am Ice, and my formula is H2O
I can be transparent or opaque.
```

Here, both `Liquid` and `Solid` classes have the same method names (`info()` and `property()`), but they behave differently based on the class instance used.

4. Polymorphism with Objects and Functions

You can pass objects of different classes to the same function, and the function will call the appropriate method based on the class type of the object.

Example:

```
class PlayStation:
    def type(self):
        print("PlayStation 5")

    def company(self):
        print("Sony")

class XBOX:
    def type(self):
        print("XBOX Series S")

    def company(self):
        print("Microsoft")

def func(obj):
    obj.type()
    obj.company()

# Polymorphism with objects and functions
obj_playstation = PlayStation()
obj_xbox = XBOX()

func(obj_playstation)  # Output: PlayStation 5, Sony
func(obj_xbox)         # Output: XBOX Series S, Microsoft
```

Here, the function `func()` accepts objects from different classes (`PlayStation` and `XBOX`) and calls their respective methods.

5. Polymorphism with Inheritance (Method Overriding)

Polymorphism is commonly seen with inheritance, where child classes can override the methods of the parent class. This is known as method overriding.

Example:

```
pythonCopy code
class F1:
    def run(self):
        print("Win")

class F2(F1):
    def run(self):
        print("2nd place - F2 cars are slower than F1 cars")

class F3(F1):
    def run(self):
        print("3rd place - F3 cars are slower than F2 and F1 c
ars")

# Polymorphism with inheritance
obj_f1 = F1()
obj_f2 = F2()
obj_f3 = F3()

obj_f1.run()  # Output: Win
obj_f2.run()  # Output: 2nd place - F2 cars are slower than F1
cars
obj_f3.run()  # Output: 3rd place - F3 cars are slower than F2
and F1 cars
```

In this example:

- `F2` and `F3` inherit the `run()` method from `F1`, but they override it with their own implementations. When the method is called on an object of these classes, the corresponding overridden method is executed.

1. Abstract Class:

An **abstract class** is a class that cannot be instantiated directly. It is designed to serve as a blueprint for other classes. It may contain both implemented methods (regular methods) and **abstract methods** (methods without implementation). The primary purpose of an abstract class is to enforce a common interface for its subclasses while allowing subclasses to provide their specific implementations for abstract methods.

Key Points:

- **Cannot be instantiated**: You cannot create an object of an abstract class.
- **Serves as a blueprint**: It provides a base structure for other classes (subclasses) to inherit from.
- **Contains abstract methods**: Abstract classes typically include one or more abstract methods that must be implemented by any concrete (non-abstract) subclass.
- **May have implemented methods**: Abstract classes can also have fully implemented methods that can be inherited by subclasses.

Example:

```
pythonCopy code
from abc import ABC, abstractmethod

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

    @abstractmethod
    def area(self):  # Abstract method
        pass  # No implementation
    def describe(self):
        return f"This is a {self.name}"
```

In this example, `Shape` is an abstract class. It cannot be instantiated, but it provides a structure for its subclasses. It has an abstract method `area()` that must be implemented by any subclass.

2. Abstract Method:

An **abstract method** is a method that is declared in an abstract class but does not have any implementation. It simply defines a method signature, and it is expected that any subclass that inherits from the abstract class will provide a concrete implementation of that method.

Key Points:

- **Declared but not implemented**: Abstract methods only declare the method signature; they don't provide the implementation.
- **Must be overridden**: Any subclass of the abstract class **must** implement the abstract method; otherwise, the subclass itself will also be considered abstract.
- **Enforces behavior**: Abstract methods ensure that all subclasses follow a certain behavior by requiring them to implement specific methods.

Example:

```
class Rectangle(Shape):  # Concrete subclass
    def __init__(self, width, height):
        super().__init__("Rectangle")
        self.width = width
        self.height = height

    def area(self):  # Implementing the abstract method
```

```
        return self.width * self.height
```

In this case, `Rectangle` is a concrete subclass of the abstract class `Shape`. It implements the abstract method `area()` from the `Shape` class. Because it provides a concrete implementation of the `area()` method, `Rectangle` is not considered abstract and can be instantiated.

Key Differences:

Aspect	Abstract Class	Abstract Method
Instantiation	Cannot be instantiated directly.	Cannot be called directly (must be implemented).
Purpose	Serves as a blueprint for other classes.	Ensures specific methods are implemented in subclasses.
Definition	Declares a class that is meant to be subclassed.	Declares a method signature without implementation.
Contains	Can contain both implemented and abstract methods.	Declares only method signature without code.
Implementation	The concrete (subclass) class must implement the abstract methods of the abstract class.	Subclasses must provide a concrete implementation for the method.

Summary:

- An **abstract class** provides a base structure and may contain both abstract and concrete methods. It defines a framework that other classes must follow.
- An **abstract method** is a method that doesn't have an implementation and must be implemented in any subclass of the abstract class.

Abstraction in Python

Abstraction is one of the fundamental concepts of Object-Oriented Programming (OOP). It is a way to **hide the complex implementation details** and **show only the essential features** of the object. The goal of abstraction is to reduce complexity and increase efficiency.

In Python, abstraction can be achieved using **abstract classes** and **abstract methods**, which are provided by the `abc` module (short for Abstract Base Classes).

Key Points of Abstraction:

1. **Hide Implementation Details**: Abstraction focuses on hiding the internal workings of a system, exposing only what is necessary.
2. **Define Common Interface**: Abstract classes and methods define a common interface for all subclasses. The subclasses must implement these methods.
3. **Ensure Consistency**: Abstraction helps in enforcing consistency among different classes that share similar behaviors, making sure that all derived classes implement certain behaviors.
4. **Enhance Code Reusability**: By providing a common interface through abstraction, you promote code reusability.

How Abstraction Works in Python

- **Abstract Class**: This is a class that cannot be instantiated and is meant to be subclassed. It often contains one or more abstract methods.
- **Abstract Method**: A method that is declared, but contains no implementation. Subclasses of the abstract class are required to implement the abstract method.

When to Use Abstraction

- **When you want to define a common interface** for a group of related classes.
- **When you need to enforce certain methods** in subclasses, ensuring they all implement a specific behavior.
- **When you want to hide complexity**, only exposing relevant details to the user or other parts of the program.

Why Use Abstraction

- **Improves Code Maintainability**: By focusing on what the object does instead of how it does it, abstraction makes your code more readable and easier to maintain.

- **Reduces Code Duplication**: Abstraction promotes reusability by providing a common interface.
- **Simplifies the User Interface**: The user of the object only needs to know what the object can do, not how it does it, which simplifies interaction with complex systems.

Example of Abstraction in Python

Let's illustrate abstraction by building a system that defines a general concept of shapes but hides the complexity of calculating areas for specific shapes.

1. **Abstract Base Class**: The abstract class `Shape` provides an interface for the `area` method, which must be implemented by any class that inherits from `Shape`.

```
pythonCopy code
from abc import ABC, abstractmethod

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

    @abstractmethod
    def area(self):
        pass

    def describe(self):
        return f"This is a {self.name}."

# Concrete class for a Rectangle
class Rectangle(Shape):
    def __init__(self, width, height):
        super().__init__("Rectangle")
        self.width = width
        self.height = height
    # Implementation of the abstract method
    def area(self):
        return self.width * self.height

# Concrete class for a Circle
class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius

    # Implementation of the abstract method
    def area(self):
        return 3.14 * (self.radius ** 2)

# Usage
rect = Rectangle(10, 5)
circle = Circle(7)

print(rect.describe())  # Output: This is a Rectangle.
print("Rectangle Area:", rect.area())  # Output: Rectangle Are
a: 50

print(circle.describe())  # Output: This is a Circle.
print("Circle Area:", circle.area())  # Output: Circle Area: 1
53.86
```

Detailed Explanation of Example:

1. **Shape (Abstract Class)**:
    - The `Shape` class is an abstract class because it inherits from `ABC`.
    - The `area()` method is an abstract method, meaning it doesn't have an implementation in the `Shape` class and **must** be implemented in any subclass of `Shape`.

- `describe()` is a concrete method in the abstract class, meaning it is not abstract and can be used directly by any subclass.

2. **Rectangle and Circle (Concrete Classes)**:
   - Both `Rectangle` and `Circle` inherit from `Shape`.
   - They provide their own implementations of the `area()` method, which was declared abstract in the `Shape` class. This is mandatory because they are concrete classes and must fulfill the contract of the abstract class by implementing the abstract method.
   - Both classes use the `super()` function to call the constructor of the abstract `Shape` class.

When and Why to Use Abstraction in Code:

- **When to Use Abstraction**:
  - When designing systems where multiple objects share a similar concept but may have different implementations (e.g., different types of shapes or vehicles).
  - When you want to enforce certain methods across all subclasses.
  - When creating frameworks or libraries where users are expected to implement certain functionalities.
- **Why Use Abstraction**:
  - To ensure consistency across multiple classes that represent related entities.
  - To enforce a contract, making sure that all subclasses implement essential methods.
  - To provide a simplified, user-friendly interface for complex systems, focusing on what the system does rather than how it does it.

Benefits of Abstraction:

- **Modularity**: You can separate implementation details from the interface, improving modularity.

- **Ease of Use**: Users of abstract classes do not need to know the internal details of the implementation.
- **Extensibility**: Abstract classes allow you to extend and introduce new implementations without altering existing code, following the Open-Closed Principle (OCP) of SOLID principles in OOP.

Operator Overloading in Python

Operator overloading is a feature in Python that allows developers to redefine the behavior of built-in operators for user-defined objects. By overloading operators, you can make instances of custom classes behave like built-in types, such as integers, strings, or lists. This improves code readability and allows intuitive interaction with objects of your class.

What is Operator Overloading?

- **Definition**: Operator overloading allows you to define how an operator (like `+`, , , `==`, etc.) should behave when applied to objects of a user-defined class. By defining special methods, you can control the behavior of these operators for your custom objects.
- **Example**: You can overload the `+` operator to add two instances of a custom class, just like adding two numbers.

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

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

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

# Creating two points
```

```
p1 = Point(2, 3)
p2 = Point(4, 5)

# Adding two points using the overloaded + operator
result = p1 + p2
print(result)  # Output: Point(6, 8)
```

Here, the `+` operator is overloaded to add two `Point` objects, resulting in a new `Point` object with the sum of their respective `x` and `y` values.

Redefining Built-in Operators for User-defined Objects

Python provides special methods, also known as **magic methods** or **dunder methods**, which are used to define how operators behave for user-defined objects.

Overloading Arithmetic Operators

Arithmetic operators such as `+`, `-`, `*`, `/`, etc., can be overloaded by defining the corresponding special methods in your class.

- `__add__`: **Overloading the `+` operator**
- `__sub__`: **Overloading the operator**
- `__mul__`: **Overloading the operator**
- `__truediv__`: **Overloading the `/` operator**

Example:

```
pythonCopy code
class ComplexNumber:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        return ComplexNumber(self.real + other.real, self.imag
+ other.imag)

    def __sub__(self, other):
        return ComplexNumber(self.real - other.real, self.imag
- other.imag)

    def __repr__(self):
        return f"{self.real} + {self.imag}i"

# Creating two complex numbers
c1 = ComplexNumber(2, 3)
c2 = ComplexNumber(1, 4)

# Overloading the + and - operators
print(c1 + c2)  # Output: 3 + 7i
print(c1 - c2)  # Output: 1 - 1i
```

Overloading Comparison Operators

Comparison operators such as `==`, `<`, `>`, etc., can also be overloaded by defining special methods in your class.

- **`__eq__`: Overloading the `==` operator**
- **`__lt__`: Overloading the `<` operator**
- **`__gt__`: Overloading the `>` operator**

Example:

```
pythonCopy code
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __eq__(self, other):
        return self.age == other.age

    def __lt__(self, other):
        return self.age < other.age

    def __gt__(self, other):
        return self.age > other.age

# Creating two person objects
p1 = Person("Alice", 30)
p2 = Person("Bob", 25)

# Comparing their ages using overloaded comparison operators
print(p1 == p2)  # Output: False
print(p1 < p2)   # Output: False
print(p1 > p2)   # Output: True
```

Here, the comparison operators `==`, `<`, and `>` are overloaded to compare the `age` attribute of two `Person` objects.

Best Practices for Operator Overloading

1. **Be Consistent**: Overloaded operators should behave logically. For example, if you overload `+` for addition, the result should resemble the expected behavior of addition.
2. **Use Readability as a Guide**: Overloading operators should make the code easier to understand, not harder. Avoid overloading operators in ways that could confuse readers of the code.
3. **Avoid Excessive Overloading**: While operator overloading can be powerful, it should be used judiciously. Too much operator overloading can lead to unclear and difficult-to-maintain code.
4. **Maintain Symmetry in Operations**: When overloading operators like `+`, ensure that the operation is symmetric when appropriate (i.e., `a + b` should equal `b + a`).

5. **Use When Appropriate**: Operator overloading is most useful for classes that represent mathematical or data structures (like complex numbers, vectors, matrices, etc.), where the operations make intuitive sense.

Use Cases for Operator Overloading

- **Mathematical Objects**: Overloading arithmetic operators is useful for mathematical objects like vectors, matrices, and complex numbers.
- **Data Structures**: Operator overloading can be useful in custom data structures where operations like addition or comparison are needed.
- **Readable and Intuitive APIs**: When designing libraries, operator overloading can lead to more readable and intuitive APIs. For example, in a game engine, you might overload operators to add vectors, move objects, or compare scores.

Dunder Methods and Magic Methods

Understanding Dunder Methods (Magic Methods) in Python

Dunder methods, short for "double underscore" methods, are special methods in Python that start and end with double underscores (e.g., `__init__`, `__str__`). They are also known as magic methods. These methods allow you to define how objects of your class should behave with respect to built-in operations and functions.

What Are Dunder Methods?

Dunder methods are predefined methods that Python uses to implement operator overloading, custom behavior, and more. They enable you to customize how your objects interact with Python's language features and built-in functions.

For example:

- `__init__`: Initializes a new instance of a class.
- `__str__`: Defines a human-readable string representation of an object.
- `__add__`: Defines the behavior for the `+` operator.

- `__eq__`: Defines the behavior for the `==` operator.

When to Use Dunder Methods

Dunder methods are used when you need to:

- **Customize Object Behavior**: Define how objects of your class should behave with built-in operations (e.g., addition, subtraction).
- **Implement Operator Overloading**: Customize the behavior of operators for your objects (e.g., `+`, ,).
- **Enhance Readability**: Provide meaningful string representations of your objects for easier debugging and display.
- **Implement Iteration**: Make objects iterable in loops or other iterable contexts.
- **Define Context Manager Behavior**: Control what happens when an object is used in a `with` statement.

Where to Use Dunder Methods

You use dunder methods in class definitions to extend or override built-in functionality. They are particularly useful in:

- **Custom Classes**: When creating your own classes that need specific behaviors or interactions with built-in operations.
- **Data Structures**: When implementing custom data structures (e.g., custom containers, iterators).
- **Operator Overloading**: When you want to define how operators should work with instances of your class.
- **Context Managers**: When implementing classes that need to manage resources with the `with` statement.

Most Common Magic Methods (Dunder Methods) in Python

Magic methods (or dunder methods) in Python are special methods that start and end with double underscores (`__`). They allow you to define how your custom objects should behave with Python's built-in operations and functions. Here, we'll explore some of the most commonly used magic methods, including `__str__`, `__repr__`, `__call__`, and others, along with examples and explanations.

1. `__init__`

Purpose: Initializes a new object of the class.

Usage: This method is called when you create a new instance of the class.

Example:

```
pythonCopy code
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Creating an instance of Person
p = Person("Alice", 30)
print(p.name)  # Output: Alice
print(p.age)   # Output: 30
```

2. `__str__`

Purpose: Defines a human-readable string representation of an object.

Usage: This method is used by the `print()` function and `str()`.

Example:

```
pythonCopy code
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old"

p = Person("Alice", 30)
```

```
print(p)  # Output: Alice is {self.age} years old
```

Explanation: `__str__` provides a readable string representation of the object. This is useful for debugging and when displaying the object to end users.

3. `__repr__`

Purpose: Defines an unambiguous string representation of an object that ideally could be used to recreate the object.

Usage: This method is used by the `repr()` function and is also used in interactive mode.

Example:

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

    def __repr__(self):
        return f"Person(name='{self.name}', age={self.age})"

p = Person("Alice", 30)
print(repr(p))  # Output: Person(name='Alice', age=30)
```

Explanation: `__repr__` is meant for developers and provides a detailed representation of the object that can be used to recreate the object. It is useful for debugging.

4. `__call__`

Purpose: Allows an instance of a class to be called as if it were a function.

Usage: This method is invoked when an instance is called.

Example:

```
pythonCopy code
class Adder:
    def __init__(self, initial_value):
        self.initial_value = initial_value

    def __call__(self, value):
        return self.initial_value + value

# Creating an instance of Adder
add_five = Adder(5)
result = add_five(10)  # Calling the instance like a function
print(result)  # Output: 15
```

Explanation: `__call__` allows you to define behavior for when an object is called as a function. This is useful for objects that should act like functions or for implementing function-like objects.

5. `__len__`

Purpose: Defines the behavior of the `len()` function for an object.

Usage: This method is used when `len()` is called on an object.

Example:

```
pythonCopy code
class MyList:
    def __init__(self, *args):
        self.items = list(args)

    def __len__(self):
        return len(self.items)

my_list = MyList(1, 2, 3, 4)
print(len(my_list))  # Output: 4
```

Explanation: `__len__` allows you to define how the length of an object should be calculated, enabling the use of `len()`.

6. `__getitem__` and `__setitem__`

Purpose: Define behavior for accessing and setting items using the bracket notation (`[]`).

Usage: `__getitem__` is used for getting items, and `__setitem__` is used for setting items.

Example:

```
pythonCopy code
class MyDict:
    def __init__(self):
        self.data = {}

    def __getitem__(self, key):
        return self.data.get(key, "Key not found")

    def __setitem__(self, key, value):
        self.data[key] = value

    def __delitem__(self, key):
        del self.data[key]

my_dict = MyDict()
my_dict["name"] = "Alice"
print(my_dict["name"])  # Output: Alice
```

Explanation: `__getitem__` allows you to retrieve items using `obj[key]`, while `__setitem__` lets you set items. This is useful for custom data structures.

7. `__eq__`, `__lt__`, `__gt__`, etc.

Purpose: Define behavior for comparison operators (`==`, `<`, `>`, etc.).

Usage: These methods are invoked when comparison operators are used.

Example:

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

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __lt__(self, other):
        return (self.x ** 2 + self.y ** 2) < (other.x ** 2 + o
ther.y ** 2)

p1 = Point(2, 3)
p2 = Point(2, 3)
p3 = Point(1, 1)

print(p1 == p2)  # Output: True
print(p1 < p3)   # Output: False
```

Explanation: These methods (`__eq__`, `__lt__`, etc.) allow you to define how your objects should be compared using comparison operators.

8. `__iter__` and `__next__`

Purpose: Define behavior for iteration (making an object iterable).

Usage: `__iter__` returns an iterator object, and `__next__` defines how to get the next item.

Example:

```
pythonCopy code
class Countdown:
    def __init__(self, start):
        self.start = start

    def __iter__(self):
        self.current = self.start
        return self

    def __next__(self):
        if self.current <= 0:
            raise StopIteration
        self.current -= 1
        return self.current

countdown = Countdown(5)
for num in countdown:
    print(num)
```

Explanation: `__iter__` initializes the iterator, and `__next__` provides the next value. This allows the object to be used in loops and other iterable contexts.

Summary

Magic methods or dunder methods enable you to customize the behavior of your objects in Python. By implementing these methods, you can define how your objects interact with built-in operations and functions, making your code more intuitive and expressive.

- `__init__`: Object initialization.
- `__str__`: Human-readable string representation.
- `__repr__`: Unambiguous string representation for debugging.
- `__call__`: Making objects callable like functions.
- `__len__`: Defining behavior for `len()`.

- `__getitem__`, `__setitem__`: Accessing and setting items using `[]`.
- `__eq__`, `__lt__`, etc.: Comparison operators.
- `__iter__`, `__next__`: Making objects iterable.

Customizing Object Behavior with Magic Methods

Magic methods allow you to enhance and customize the behavior of objects beyond the basic operations that Python provides. By overloading these methods, you can:

- Control how objects are represented (`__str__`, `__repr__`).
- Define how they are compared (`__eq__`, `__lt__`).
- Manage how they interact with arithmetic and logical operators (`__add__`, `__sub__`).
- Allow them to behave like containers (`__getitem__`, `__setitem__`).
- Enable iteration over your custom objects (`__iter__`, `__next__`).

Use Cases for Dunder Methods

1. **Custom Data Structures**: You can implement your own data structures like stacks, queues, or matrices that behave like built-in data types by overloading methods such as `__getitem__`, `__setitem__`, `__len__`, and more.
2. **Debugging and Logging**: Implementing `__repr__` ensures that objects have meaningful representations during debugging sessions, improving code maintainability.
3. **Mathematical Operations**: If you define a custom class to represent a mathematical concept (e.g., vectors, complex numbers), you can overload arithmetic operators like `__add__`, `__mul__`, etc., to implement appropriate behavior.
4. **Callable Objects**: Overloading `__call__` can be useful when you want to treat objects as functions (e.g., for stateful behavior or dependency injection).
5. **Iterators and Generators**: Custom iterators can be built by implementing `__iter__` and `__next__`, which is useful for implementing custom loops and generators in Python.

Flow of Execution in python:

Example 1: Simple Class and Object

```
pythonCopy code
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def display_info(self):
        return f"Car make: {self.make}, Model: {self.model}"

car1 = Car("Toyota", "Corolla")
print(car1.display_info())
```

Flow Explanation:

- A class `Car` is defined with an `__init__()` method (constructor) that initializes two instance variables: `make` and `model`.
- The method `display_info()` is defined to return a string containing the make and model of the car.
- An object `car1` is created using the `Car` class, passing `"Toyota"` and `"Corolla"` as arguments to the constructor.
- `car1.display_info()` is called, which returns and prints the car's make and model.

Example 2: Class with Inheritance

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

    def sound(self):
        return "This animal makes a sound."

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

    def sound(self):
        return f"{self.name} barks."

dog1 = Dog("Buddy", "Golden Retriever")
print(dog1.sound())
```

Flow Explanation:

- The `Animal` class is defined with a constructor that initializes the `name` of the animal and a method `sound()` that returns a generic message.
- The `Dog` class is defined as a subclass of `Animal`. It calls the parent class's constructor using `super()` to initialize the `name` attribute and defines an additional attribute `breed`.
- The `Dog` class overrides the `sound()` method to provide a specific implementation for dogs.
- An object `dog1` is created using the `Dog` class. The method `dog1.sound()` is called, which invokes the overridden `sound()` method from the `Dog` class and prints `"Buddy barks."`.

Example 3: Advanced Class and Object (Polymorphism)

```
pythonCopy code
class Shape:
    def area(self):
        return "Calculating area."

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

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

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

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

shapes = [Rectangle(5, 10), Circle(7)]
for shape in shapes:
    print(shape.area())
```

Flow Explanation:

- A base class `Shape` is defined with a method `area()` that returns a placeholder message.
- The `Rectangle` class inherits from `Shape` and implements its own version of the `area()` method by calculating the area of a rectangle.
- The `Circle` class also inherits from `Shape` and provides its own implementation of the `area()` method, calculating the area of a circle.

- A list `shapes` is created with objects of both `Rectangle` and `Circle`.
- A loop iterates through the `shapes` list, and for each shape object, the respective `area()` method is called, demonstrating polymorphism. The `Rectangle` and `Circle` objects each use their own implementation of the `area()` method.

Method Overloading in Python

Method overloading refers to the ability of a method to perform different functions based on the input parameters. In many languages (like Java, C++), method overloading allows multiple methods with the same name but different parameter types or numbers of parameters to exist within the same class.

However, Python **does not** natively support method overloading in the traditional sense. In Python, you cannot define multiple methods with the same name but different arguments. Instead, Python handles this by allowing default values for parameters and using `*args` and `**kwargs` to accept a variable number of arguments.

Simulated Method Overloading in Python:

Python achieves a similar effect by using default arguments or handling multiple arguments in a single method using conditionals or `*args`/`**kwargs`.

Example 1: Using Default Arguments:

```
pythonCopy code
class MathOperations:
    def multiply(self, a, b=1, c=1):
        return a * b * c

math_op = MathOperations()

print(math_op.multiply(5))          # Multiplies 5 by default b
and c (5 * 1 * 1 = 5)
print(math_op.multiply(5, 2))       # Multiplies 5 * 2 (default
c is 1, 5 * 2 * 1 = 10)
print(math_op.multiply(5, 2, 3))   # Multiplies 5 * 2 * 3 = 30
```

Here, the `multiply` method performs multiplication based on how many arguments are passed. This allows for flexibility without having to overload the method.

Example 2: Using `*args` for Variable Number of Arguments:

```
pythonCopy code
class MathOperations:
    def add(self, *args):
        return sum(args)

math_op = MathOperations()

print(math_op.add(1, 2))           # Output: 3
print(math_op.add(1, 2, 3))        # Output: 6
print(math_op.add(1, 2, 3, 4))     # Output: 10
```

Here, the method `add()` accepts a variable number of arguments using `*args`.

Operator Overloading in Python

Operator overloading allows you to define or modify the behavior of operators (`+`, `-`, `*`, etc.) for user-defined objects (like classes). This is achieved by overriding special methods (called **dunder methods** or **magic methods**) in a class that correspond to specific operators.

For example, in Python, you can override the `__add__()` method in a class to change the behavior of the `+` operator for instances of that class.

Example of Operator Overloading:

Let's take a simple example of overloading the `+` operator for a custom class.

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

    # Overloading the '+' operator
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    # Overloading the str method for easy printing
    def __str__(self):
        return f"Point({self.x}, {self.y})"

p1 = Point(2, 3)
p2 = Point(4, 5)

p3 = p1 + p2   # This will invoke the __add__ method
print(p3)      # Output: Point(6, 8)
```

Explanation:

- `__add__()`: We have overridden the `__add__()` method to allow the `+` operator to add two `Point` objects. Instead of performing simple addition, the method adds the corresponding `x` and `y` values of the two `Point` objects.
- `__str__()`: This method is overridden so that when we print the `Point` object, it returns a string representation of the point, e.g., `Point(6, 8)`.

Other Operator Overloading Examples:

1. Overloading the `*` Operator:

```
pythonCopy code
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    # Overloading the '*' operator to perform scalar multiplic
ation
    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)

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

v = Vector(2, 3)
v_scaled = v * 3  # This will invoke the __mul__ method
print(v_scaled)   # Output: Vector(6, 9)
```

Here, the `__mul__()` method is used to perform scalar multiplication of a vector.

2. Overloading the `==` Operator:

```
pythonCopy code
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Overloading the '==' operator to compare two Person obje
cts
    def __eq__(self, other):
        return self.name == other.name and self.age == other.a
ge

p1 = Person("Alice", 30)
p2 = Person("Alice", 30)
p3 = Person("Bob", 25)

print(p1 == p2)  # Output: True (because the name and age are
the same)
```

```
print(p1 == p3)  # Output: False
```

https://www.youtube.com/playlist?list=PLH5lMW7dI2qdC6q3JqBu-zXvYU5SxHO5m