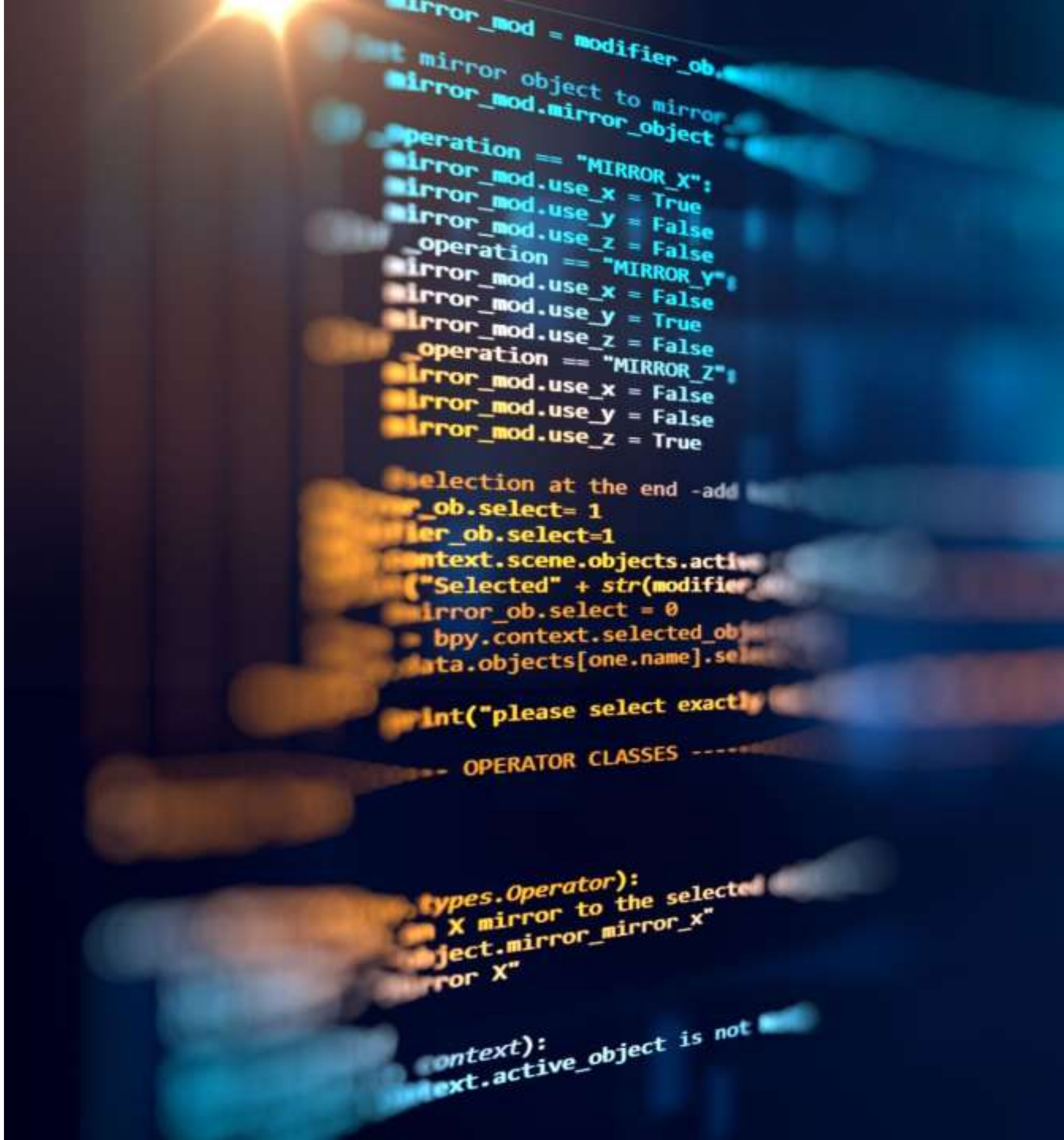
Python as Object-Oriented Programming

Unit 2

Overview

- Introduction to Object-Oriented Programming
- Classes and Objects
- Encapsulation
- Inheritance
- Polymorphism



Procedure vs Object oriented

Procedure vs Object Oriented

Feature	Procedural Programming	Object-Oriented Programming
Data Handling	Data is stored separately (dictionary).	Data is inside objects (encapsulation).
Function Calls	Functions operate on external data.	Methods operate on object properties.
Scalability	Harder to scale (managing multiple entities is complex).	Easily scalable (multiple objects can be created).
Code Reusability	Code duplication across functions.	Code is reusable with objects.

Comparison program

Let's discuss it latter

🍳 Procedure-Oriented Programming (POP) - A Step-by-Step Kitchen

In a **procedure-oriented** restaurant, everything is done through a **series of steps (functions)**.

The data (ingredients) is passed from one function (chef) to another.

Each step follows a **specific sequence**, and there is no direct connection between the data and the functions.

How it works in a restaurant:

The waiter **takes the order** from the customer.

The order is **written on a slip** and sent to the kitchen.

The chef **reads the slip** and starts cooking.

Once the dish is ready, the chef **hands it over** to another waiter.

The waiter **delivers the food** to the customer.

Problems in this system:

01

Each step is **dependent on the previous one**.

02

If there is an error (e.g., the slip gets lost), the whole process **fails**.

03

The chefs, waiters, and ingredients are **not grouped together**; they work separately.

04

If a new dish is added to the menu, **every step needs to be modified**.

Example in Procedural Python

```
def take_order():
    return "Burger"

def cook_food(order):
    return f"Cooking {order}"

def serve_food(food):
    return f"Serving {food}"
```

Example in Procedural Python

```
# Process flow
order = take_order()
food = cook_food(order)
print(serve_food(food))  # Output: Serving Cooking Burger
```

🍔 Object-Oriented Programming (OOP) - A Well-Organized Restaurant

In an **OOP-based restaurant**, everything is **grouped into objects**.

Each object (e.g., **Waiter, Chef, Customer**) has its own **attributes and methods** (actions it can perform).

Instead of passing data from one function to another, each object **manages its own data**.

How it works in a restaurant:

Each waiter is an **object** that takes orders and serves customers.

Each chef is an **object** responsible for cooking.

The **order** is stored **inside the object**, so it doesn’t get lost.

If a new dish is added, only the **Chef class** needs to be modified.

Example in OOP Python

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


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

```
def place_order(self):
        print(f"{self.name} orders a
{self.order}")


class Chef:
    def prepare_food(self, order):
        print(f"Chef is cooking
{order}")
        return f"{order} is ready!"


class Waiter:
    def serve_food(self, customer,
food):
        print(f"Serving {food} to
{customer.name}")
```

Example in OOP Python: creating objects

```python
# Creating objects
restaurant = Restaurant("OOP Diner")
customer1 = Customer("Alice", "Pasta")
chef = Chef()
waiter = Waiter()

# Process flow
customer1.place_order()
food = chef.prepare_food(customer1.order)
waiter.serve_food(customer1, food)
```

Output:
 # Alice orders a Pasta
 # Chef is cooking Pasta
 # Serving Pasta is ready! to Alice

Introduction to Object-Oriented Programming

Understanding __init__ in Python OOP

📌 **What is __init__ in Python?**

- __init__ is a **constructor method** in Python.
- It is **automatically** called when an object of a class is created.
- It **initializes** the object's attributes with values.

```
class ClassName:
    def __init__(self, parameter1, parameter2):
        self.attribute1 = parameter1
        self.attribute2 = parameter2
```

📌 What is self?

Self:

- In Python, self is a **convention** used as the **first parameter** in instance methods of a class.
- It represents the **current instance** of the class and allows access to the **attributes and methods** of that instance.
- self is **not a keyword** in Python; it is just a naming convention.
- However, using self as the first parameter is strongly recommended.

🎯 Why is self Important?

- It helps to **differentiate instance attributes from local variables**.
- It allows each object (instance) to have **its own copy of attributes**.
- It provides a way to access **methods and attributes within the class**.

Example

```
class Student:

    def __init__(self, name, age):

        self.name = name  # Assign the value of name to the
object

        self.age = age    # Assign the value of age to the
object

    def display_info(self):

        print(f"Student Name: {self.name}, Age: {self.age}")
```

Example

```
# Creating an object of the Student class
student1 = Student("Alice", 20)


# Calling the method to display details
student1.display_info()
```

Explaination

The __init__ method is called automatically when we create student1.

"Alice" is assigned to self.name, and 20 is assigned to self.age.

The display_info() method prints the student's details.

🚀 Example: Without self (Incorrect Code)

```
class Car:

    def __init__(brand, model):   # ❌ Incorrect: 'brand'
                                     should be 'self'

        brand.model = model   # ❌ Incorrect: 'brand' does not
                                 refer to the instance


    def show_model(brand):   # ❌ Incorrect: should use 'self'

        print(f"Car model: {brand.model}")


car1 = Car("Tesla")   # ❌ This will raise an error

car1.show_model()
```

📌 Example: self with Multiple Objects

```
class Animal:

    def __init__(self, species):

        self.species = species  # Each instance
gets its own species


    def speak(self):

        print(f"I am a {self.species}!")
```

🔍 **Explanation:**

•self.species = species allows each object to store its **own species name**.

•Even though both dog and cat use the same class, their **instance attributes** are different.

📌 Example: self with Multiple Objects

```
# Creating different objects
dog = Animal("Dog")
cat = Animal("Cat")

dog.speak()  # Output: I am a Dog!
cat.speak()  # Output: I am a Cat!
```

📌 Example: self in Class Methods vs Static Methods

```
class MathOperations:
    def instance_method(self, x, y):  # Uses self (instance method)
        return x + y

    @staticmethod
    def static_method(x, y):  # No self (static method)
        return x * y
```

🔍 **Key Difference:**
•instance_method() uses self to refer to the instance.
•static_method() does **not** use self because it doesn't need instance data.

📌 Example: self in Class Methods vs Static Methods

```
math_obj = MathOperations()
print(math_obj.instance_method(2, 3))  # Output: 5
print(MathOperations.static_method(2, 3))  # Output: 6
```

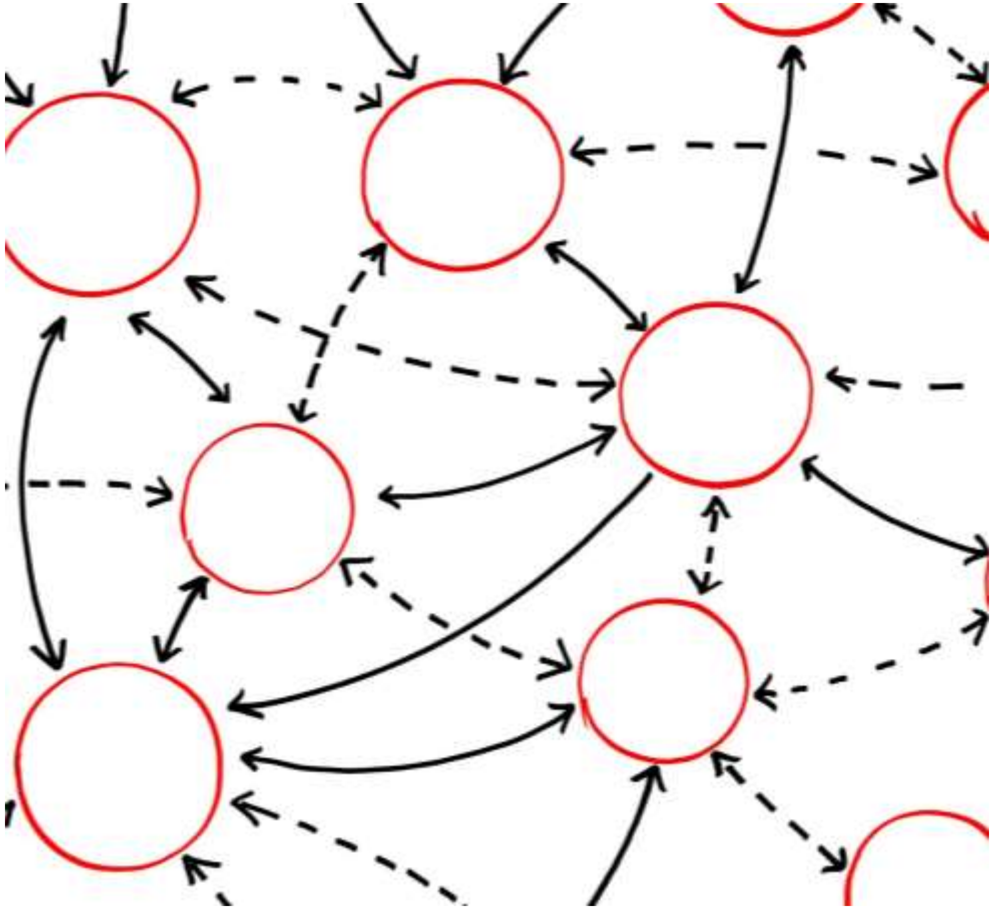
🔍 **Key Difference:**

- instance_method() uses self to refer to the instance.
- static_method() does **not** use self because it doesn't need instance data.

self in Python OOP

Concept	With self	Without self
Access instance variables	✅ Yes	❌ No
Modify instance attributes	✅ Yes	❌ No
Call instance methods	✅ Yes	❌ No
Used in static methods	❌ No	✅ Yes

Definition and principles of OOP



Encapsulation

Encapsulation is the principle of bundling data and methods that operate on that data within a single unit or class.

Inheritance

Inheritance allows a new class to inherit properties and methods from an existing class, promoting code reusability.

Polymorphism

Polymorphism enables objects to be treated as instances of their parent class, allowing for flexible method implementation.

Abstraction

Abstraction involves hiding complex implementation details and showing only essential features of an object.

Advantages of OOP

Improved Code Organization

OOP facilitates better organization of code through encapsulation, making it easier to understand and manage.

Code Reusability

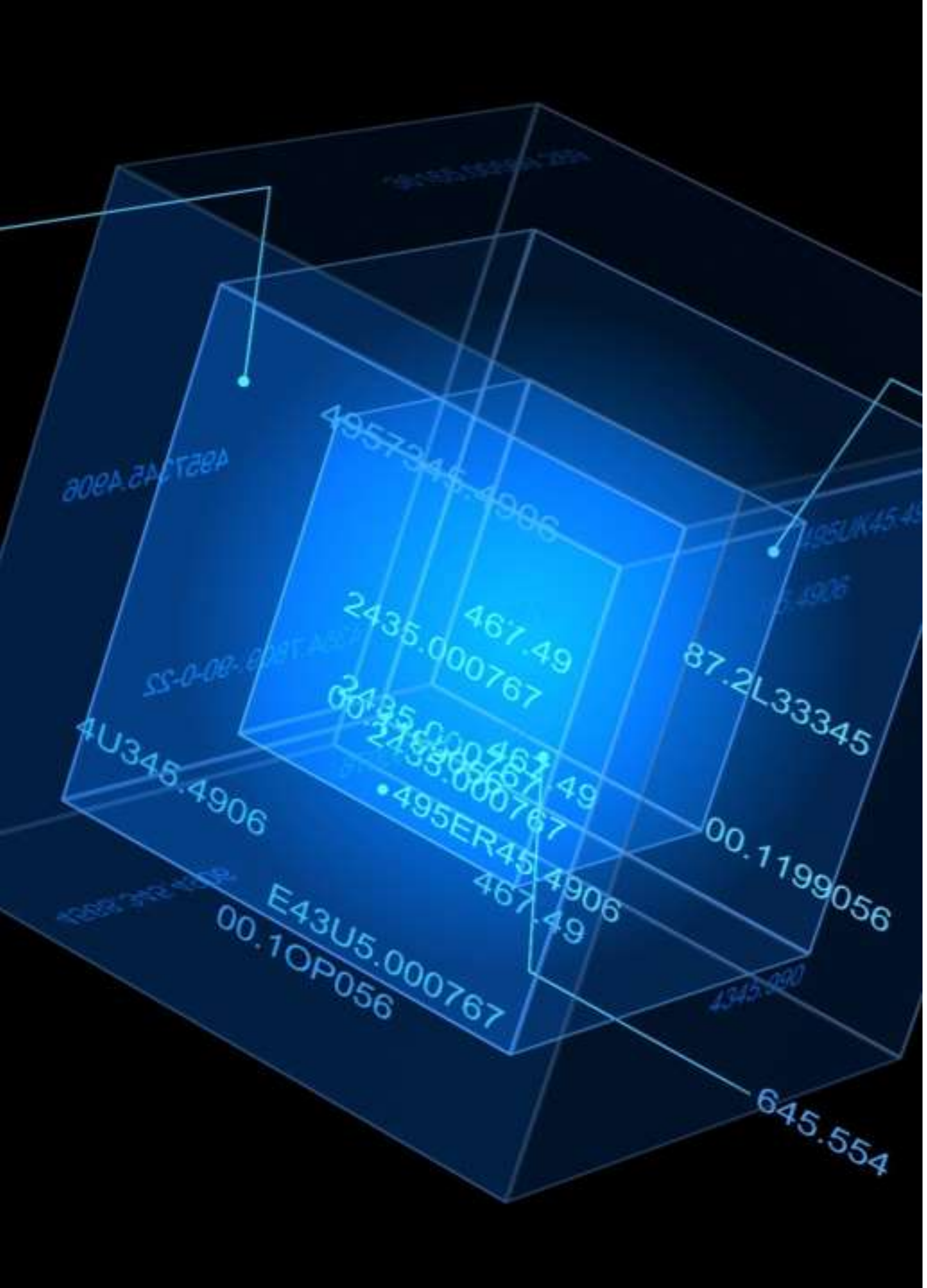
OOP promotes reusability of code components through inheritance and polymorphism, saving time and effort in development.

Easier Maintenance

With OOP, maintaining and updating code becomes simpler due to its modular nature, reducing the likelihood of bugs.

Enhanced Collaboration

OOP allows for better collaboration among developers, especially in large-scale projects, leading to improved teamwork.



Comparison with other programming paradigms

Object-Oriented Programming

OOP emphasizes the use of objects to encapsulate data and methods, promoting modular and reusable code.

Procedural Programming

Procedural programming focuses on a sequence of steps and procedures to perform tasks, which can lead to more complex code management.

Managing Complexity

OOP provides techniques to better manage complexity in software development through encapsulation and abstraction.

Classes and Objects

Understanding classes and objects

- **Creating a Class :**In Python, classes are defined using the 'class' keyword, which is fundamental to object-oriented programming.

- **Attributes and Methods:** Classes can encapsulate attributes (data) and methods (functions), which define the behaviours of the objects created from them.

- **Instantiating Objects:** Objects are created from classes, allowing you to access their properties and methods, leading to modular programming.

Creating and initializing objects

```
class Person:
    def __init__(self,name,age):
        self.name = name
        self.age = age
    def getName(self):
        print(”My name
is:",self.name)
    def getAge(self):
        print(”Age:",self.age)


p = Person("vibhooti", 22)
p.getName()
p.getAge()
```

Object Creation

Objects are instantiated by invoking the class name, which acts like a function in programming.

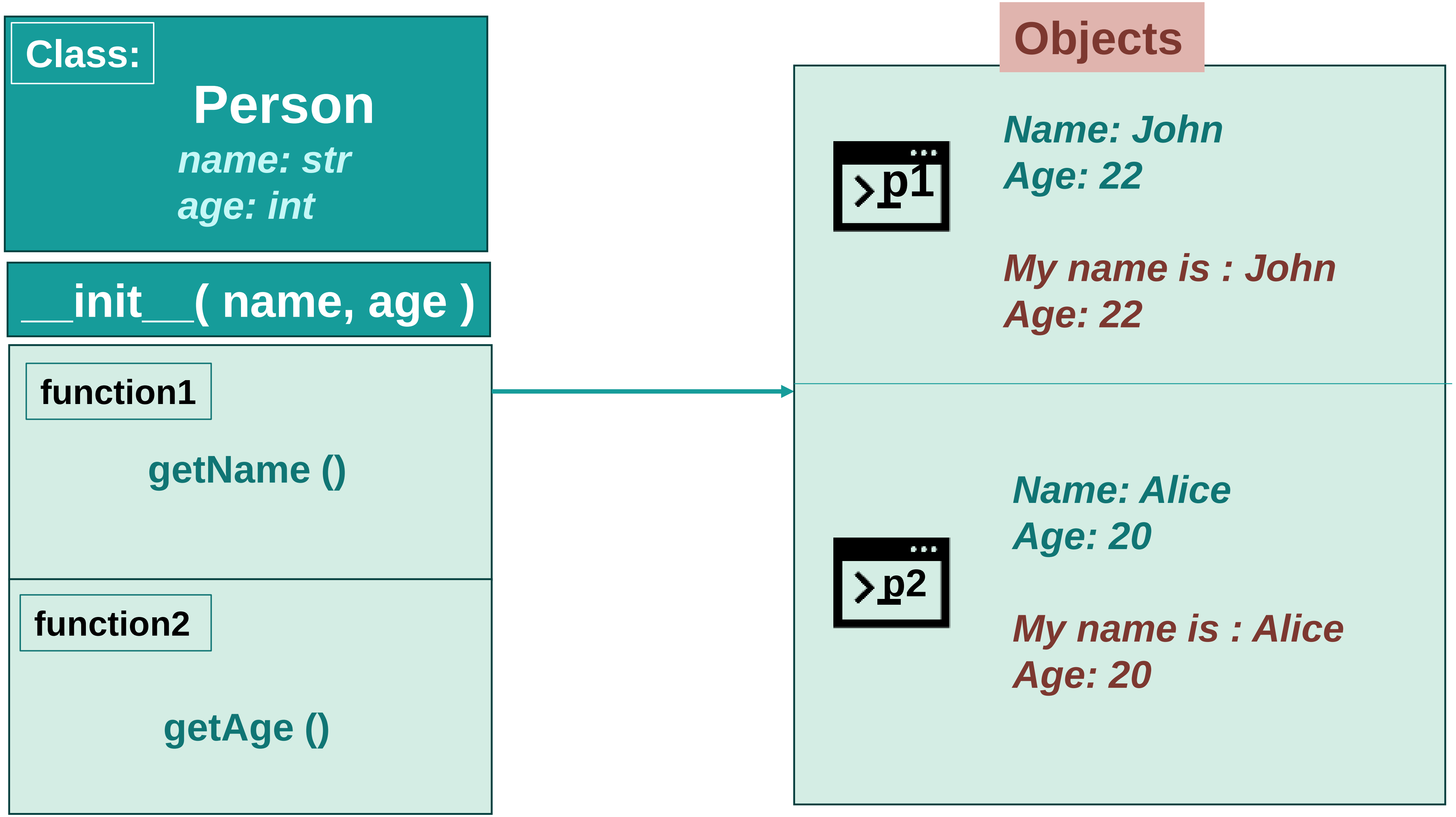
Constructor Method

The constructor method (__init__) initializes the new object's attributes using the provided arguments.

Initializing Attributes

Attributes of an object are set during initialization, allowing for customized object properties.

Class and Object Diagram



Object methods and attributes

Definition of Methods

Methods are functions defined within a class that can perform operations on an object's attributes.

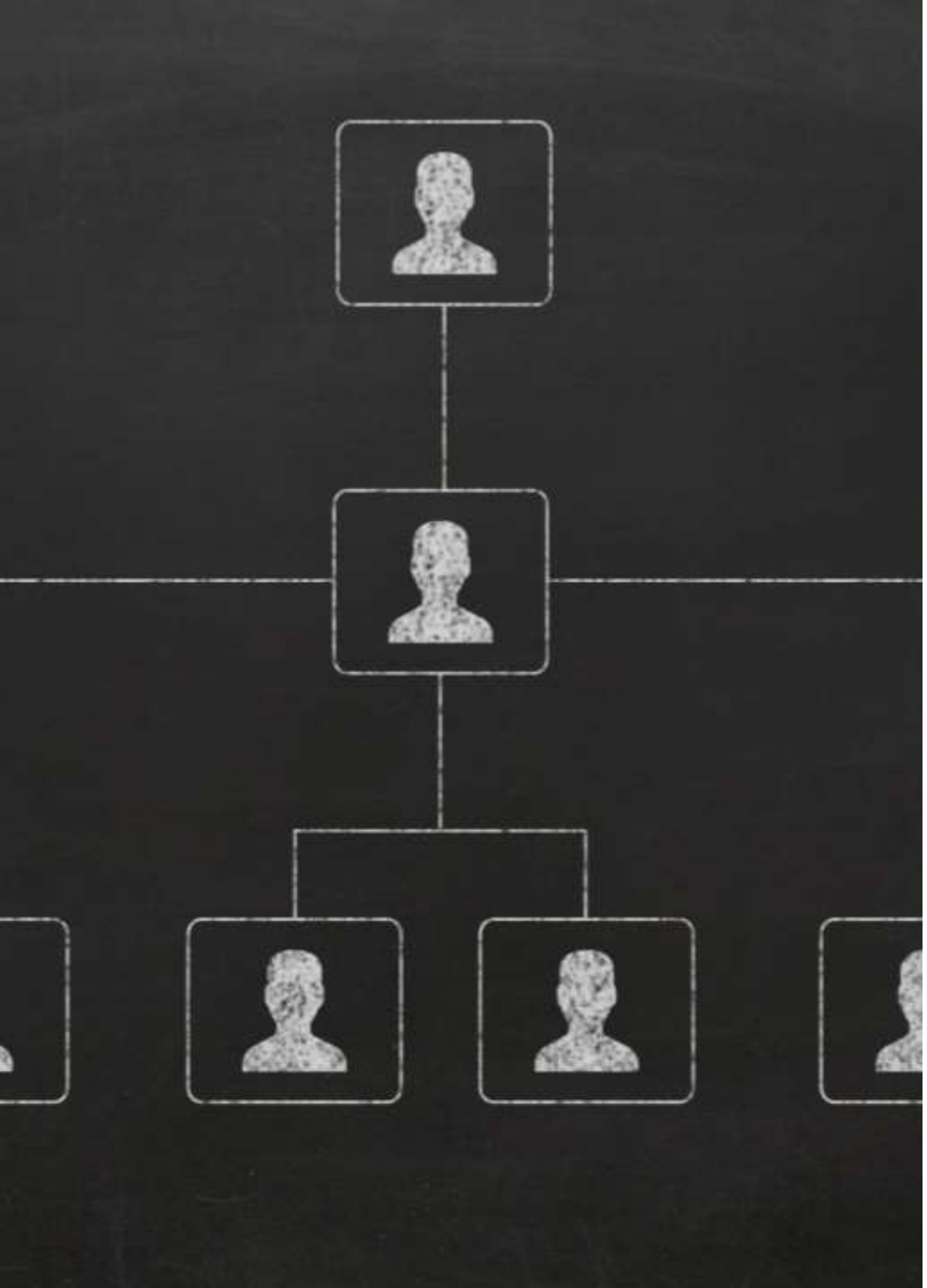
Understanding Attributes

Attributes are variables that represent the state or characteristics of an object, holding its data.

Importance in OOP

Understanding methods and attributes is fundamental in object-oriented programming to effectively manage data and behavior.

Inheritance



Understanding Inheritance

Definition of Inheritance

Inheritance in Python allows a new class to inherit properties and methods from an existing class, enhancing code reusability.

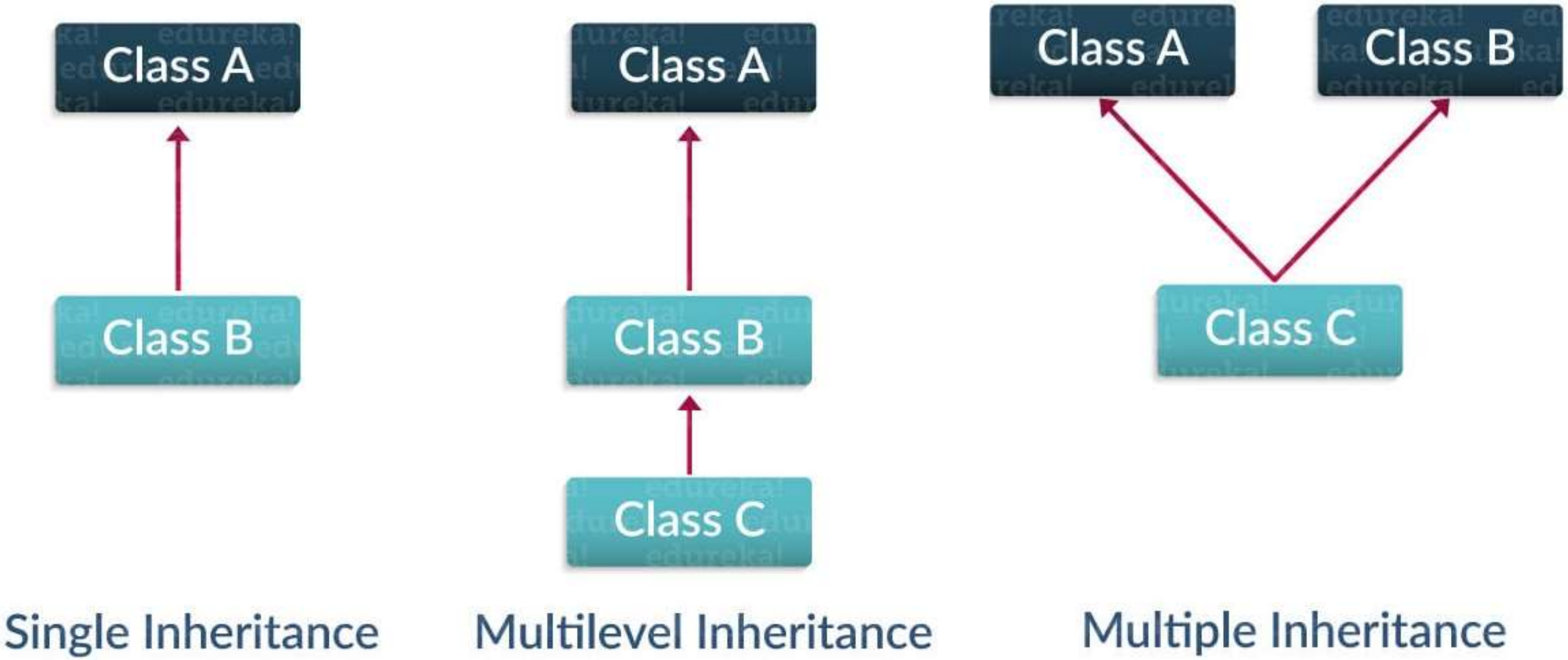
Creating Derived Classes

To implement inheritance, define a new class that derives from the parent class, gaining access to its features.

Benefits of Inheritance

Inheritance promotes code reusability, making it easier to manage and extend functionalities in programming.

Types of Inheritance



Types of Inheritance

Single Inheritance

Single inheritance allows a class to inherit from one parent class, promoting simplicity in class relationships.

Multiple Inheritance

Multiple inheritance enables a class to inherit from multiple parent classes, allowing for greater flexibility in design.

Multilevel Inheritance

Multilevel inheritance involves a class inheriting from another derived class, creating a chain of inheritance.

Inheritance example

```
# OOP - Class Inheritance
class Car:
  def __init__(self):
    self.wheels = 4
    self.seats = 5
  def drive(self):
    print("Driving a car...")
myCar = Car()
myCar.drive()
class SportsCar(Car):
  def __init__(self):
    super().__init__()
    self.engine_power = '400 HP'
    self.seats = 2
```

```
  def drive(self):
    print("Driving a sports
car...")
mySportsCar = SportsCar()
mySportsCar.drive()


class SportsCar(Car):
  def __init__(self):
    super().__init__()
    self.engine_power = '400 HP'
    self.seats = 2


mySportsCar = SportsCar()
mySportsCar.drive()
```

❖ What is Multiple Inheritance?

- **Multiple Inheritance** is a feature in object-oriented programming where a class can **inherit attributes and methods from more than one parent class**.
- Python supports multiple inheritance directly.
- This means a class can **combine functionalities of multiple base classes**, allowing for more flexible and reusable code.
- Syntax:

```
class Base1:
    # code for Base1

class Base2:
    # code for Base2

class Derived(Base1, Base2):
    # code for Derived that inherits from Base1 and Base2
```

Multiple inheritance example

```
# Multiple Inheritance Example
class Engine:
    def __init__(self, engine_type):
        self.engine_type = engine_type

    def start(self):
        print(f"{self.engine_type} engine started")

class Wheels:
    def __init__(self, wheel_count):
        self.wheel_count = wheel_count

    def rotate(self):
        print(f"{self.wheel_count} wheels rotating")
```

Multiple inheritance

```
class Car(Engine, Wheels):  # Car inherits from both Engine
and Wheels
    def __init__(self, engine_type, wheel_count, brand):
        Engine.__init__(self, engine_type)
        Wheels.__init__(self, wheel_count)
        self.brand = brand

    def drive(self):
        print(f"{self.brand} car is driving")

# Create an object of Car class
myCar = Car("Petrol", 4, "Toyota")
myCar.start()     # Inherited from Engine
myCar.rotate()    # Inherited from Wheels
myCar.drive()     # Method from Car class
```

Multilevel Inheritance

- **Multilevel Inheritance** is a type of inheritance where a class is derived from a class, which is **already derived from another class**.
- Each level of inheritance **inherits properties and methods** from the class above it.

```
class BaseClass:
    # Base class code

class DerivedClass1(BaseClass):
    # Inherits from BaseClass

class DerivedClass2(DerivedClass1):
    # Inherits from DerivedClass1 (and indirectly from
BaseClass)
```

Multilevel Inheritance

```
# Multilevel Inheritance Example
class Animal:
    def __init__(self, species):
        self.species = species

    def breathe(self):
        print(f"{self.species} is breathing")

class Mammal(Animal):  # Mammal inherits from Animal
    def __init__(self, species, is_warm_blooded):
        super().__init__(species)  # Call Animal's __init__
        self.is_warm_blooded = is_warm_blooded

    def feed_milk(self):
        print(f"{self.species} feeds milk (Warm-blooded: 
{self.is_warm_blooded})")
```

Multilevel Inheritance

```
class Dog(Mammal):  # Dog inherits from Mammal
    def __init__(self, breed, species="Dog", is_warm_blooded=True):
        super().__init__(species, is_warm_blooded)  # Call Mammal's
__init__
        self.breed = breed

    def bark(self):
        print(f"{self.breed} dog barks")

# Create an object of Dog class
myDog = Dog("Labrador")
myDog.breathe()    # From Animal
myDog.feed_milk()  # From Mammal
myDog.bark()       # From Dog
```