

# Programming Refresher



# OOPs Concepts with Python



# Learning Objectives

By the end of this lesson, you will be able to:

- 👁 Explain the concept of object-oriented programming (OOP) and its characteristics
- 👁 Identify objects and classes
- 👁 Describe methods, attributes, and access modifiers
- 👁 Define abstraction, encapsulation, inheritance, and polymorphism with real-life examples



## Business Scenario

ABC is a banking firm that is currently developing a banking management system application. This application should include customer information accessible to bank employees. Employees should be able to access and edit this information in response to customer requests. However, the current application lacks security, as workers at all levels can access a large amount of data. Additionally, the bank wants to customize the application for specific branches.

The firm has decided to update the application, ensuring that only the necessary customer details are available to employees. Critical information will only be accessible to senior officials. To accomplish this, the organization will apply object-oriented programming (OOP) concepts, including encapsulation and abstraction. Methods, attributes, and access modifiers will also be utilized in the update.

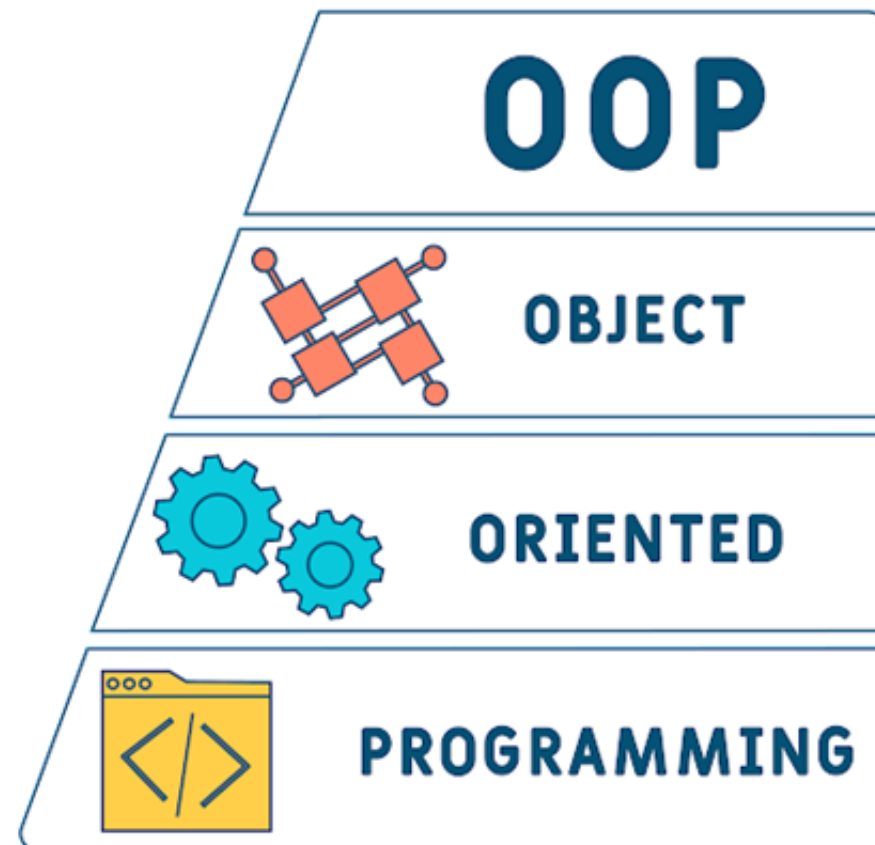




# **Object-Oriented Programming Language**

# What Is OOPs?

OOPs refer to languages that use objects in programming. It aims to implement real-world entities, such as inheritance, information hiding, and polymorphism in programming.



# OOP: Concepts

The four concepts of object-oriented programming are:

Encapsulation

Inheritance

Polymorphism

Abstraction



# **Objects and Classes**





## Discussion

# Python

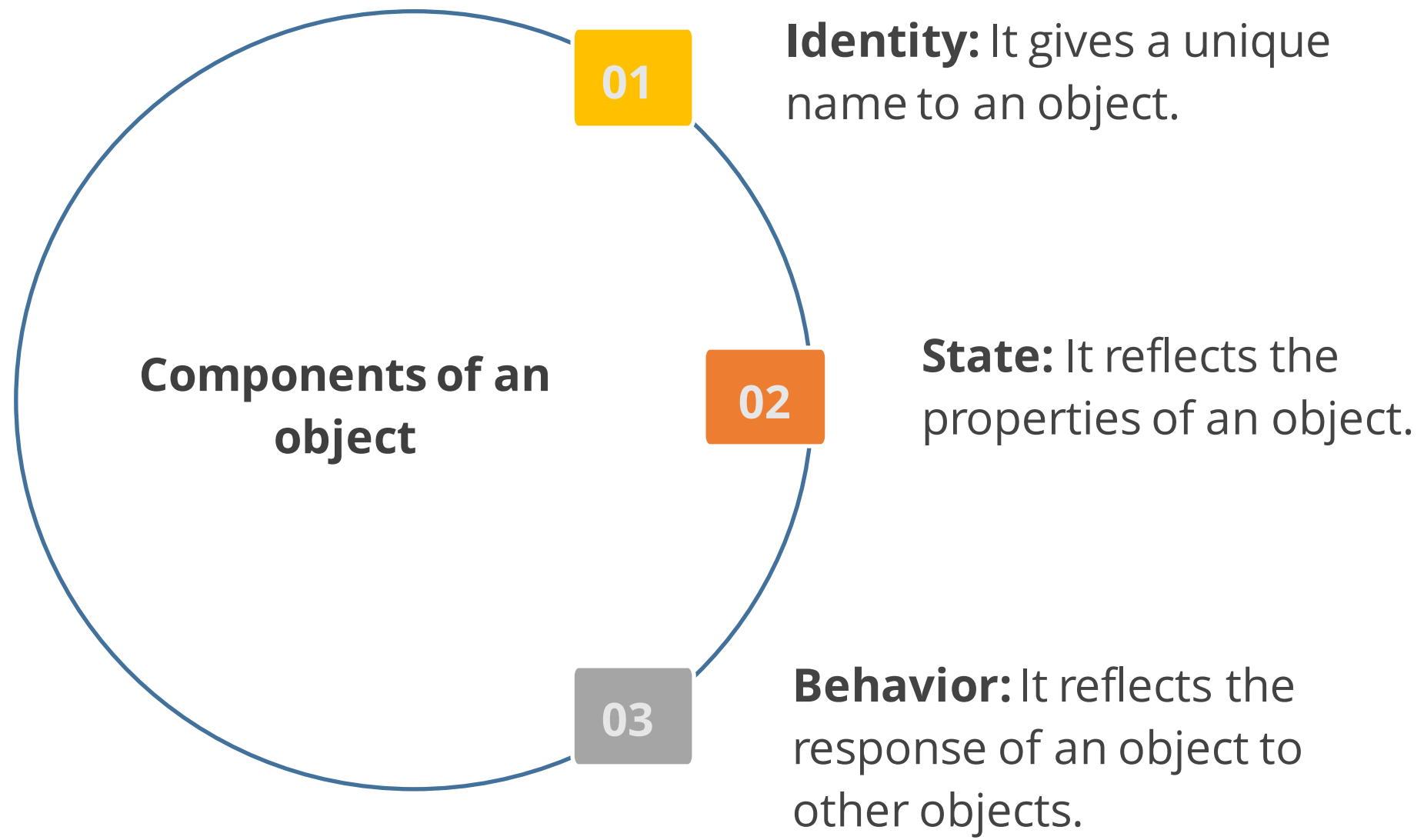
Are objects and classes the same?

- What are objects and their components?
- What are classes?



# Objects

An object represents an entity in the real world that can be distinctly identified. An object consists of the following components:



# Objects: Example

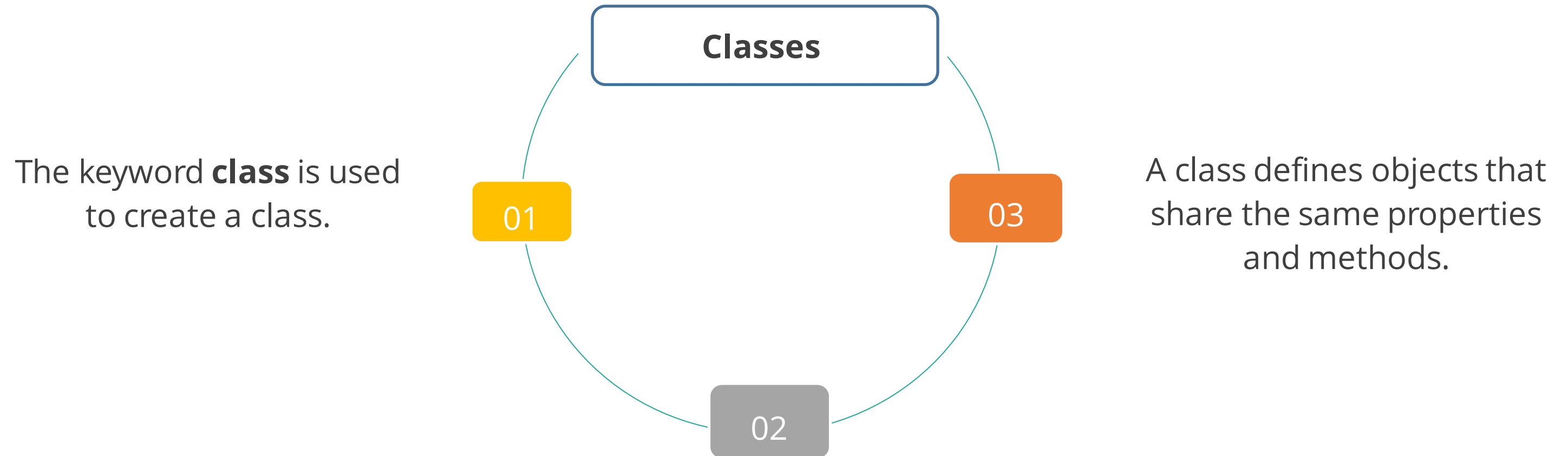
An example of an object is given below:

## Object: Dog

Identity	State or Attribute	Behavior
Name of the dog	Breed	Bark
	Age	Sleep
	Color	Eat

# Classes

A class is a blueprint for an object.



A class is like an object constructor for creating objects.

# Classes: Example

An example of a class is given below:

## Example

```
class Dog:  
    pass
```

Here, the **class** keyword is used to define an empty **class Dog**.

An instance is a specific object created from a particular class.

# Python



Are objects and classes the same?

- What are objects and their components?

Answer: An object represents an entity in the real world that can be distinctly identified. An object consists of its identity, state, and behavior.

- What are classes?

Answer: A class is a blueprint for an object. A class defines objects that share the same properties and methods.



## Methods and Attributes



# Methods

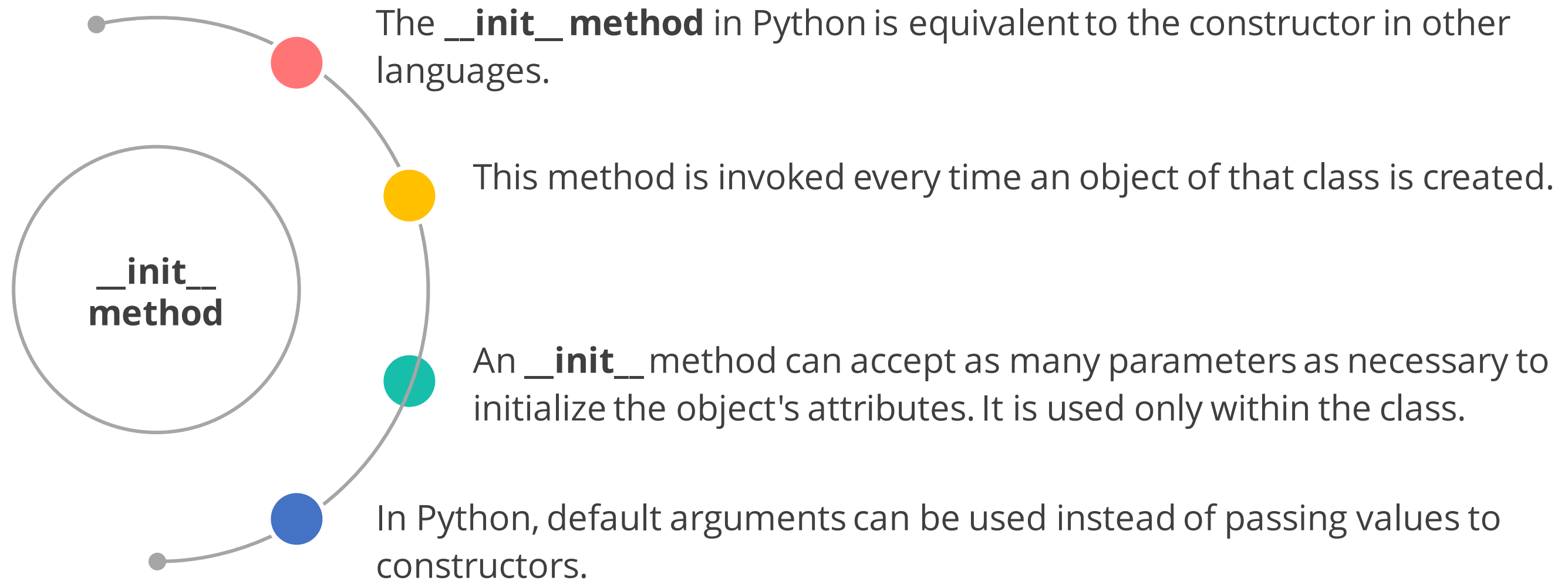
Methods are functions defined inside a class. They are invoked by objects to perform actions on other objects.

**`__init__`** is a method that is automatically called when memory is allocated to a new object.

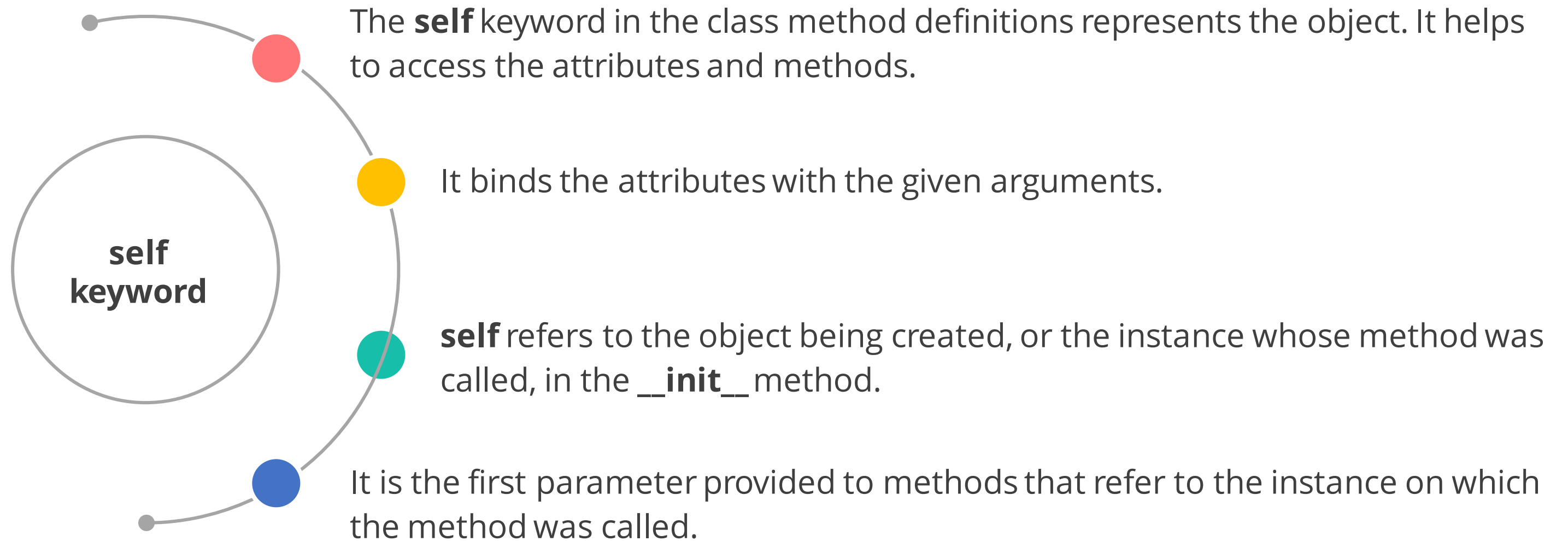
```
# A sample class with init method
class Person:
    # init method or constructor
    def __init__(self, name):
        self.name = name
    )
```

- In the init method, **self** refers to the newly created object.
- In other class methods, it refers to the instance whose method was called.

# The `__init__` method



# Self



# Instantiating Objects

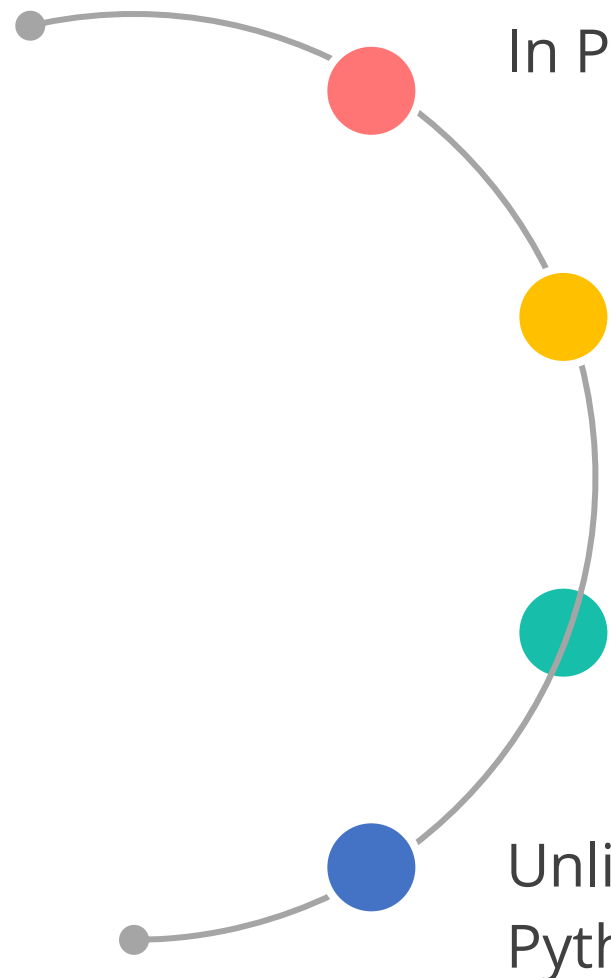
It refers to the creation of objects or instances of a given class. To instantiate a class, the user needs to call the class as if it is a function, passing the arguments as defined in the **\_\_init\_\_** function of the class.

Example: Create an object for student class

```
st1 = student('Alvin Joseph', 21)
```

- Here '**st1**' is an object of a **class student**.
- The values passed in a class are the arguments specified in the **\_\_init\_\_** function of the class.

# Deleting Instances

- 
- In Python, it is not necessary to explicitly delete an object after use.
  - It automatically recognizes and releases the memory when all references to a particular block of memory are no longer needed.
  - Python has automatic garbage collection.
  - Unlike other programming languages (e.g., C++), destructors are not needed in Python classes.

# Attributes

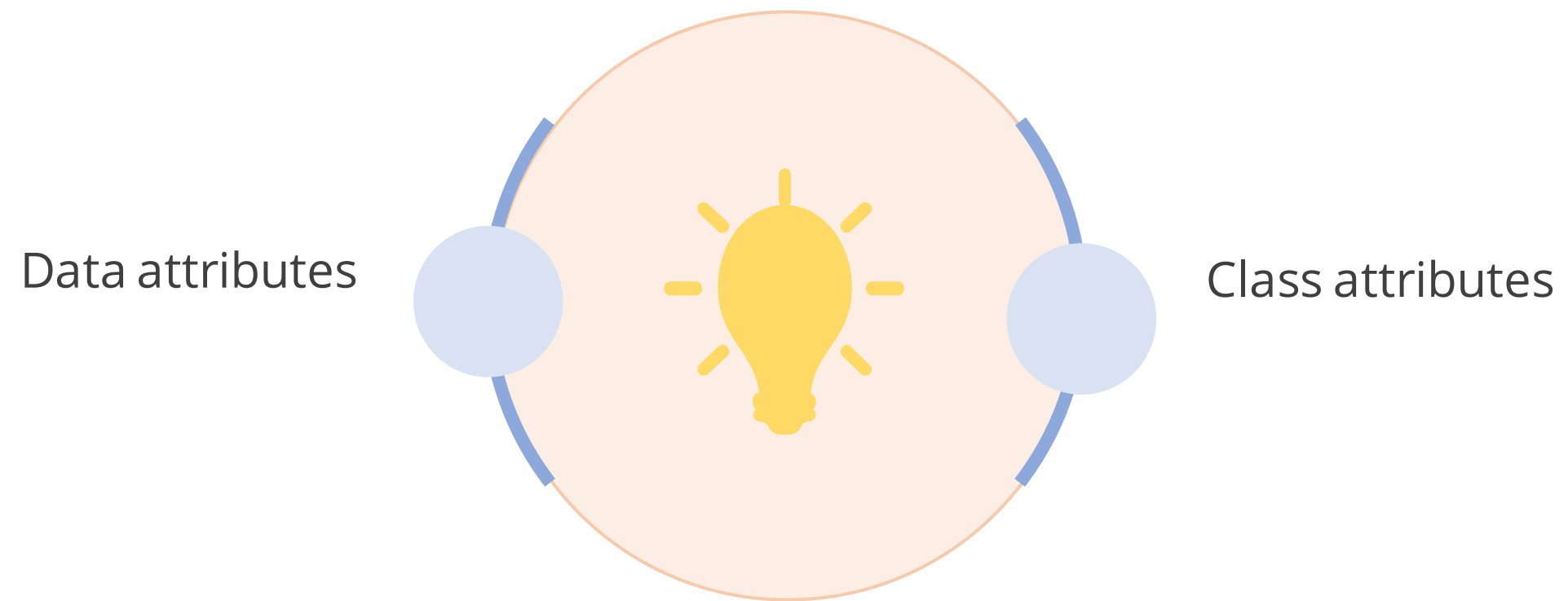
The non-method data stored by the objects are called attributes.

## Object: Dog

Identity	Attribute
Name of the dog	Breed
	Age
	Color

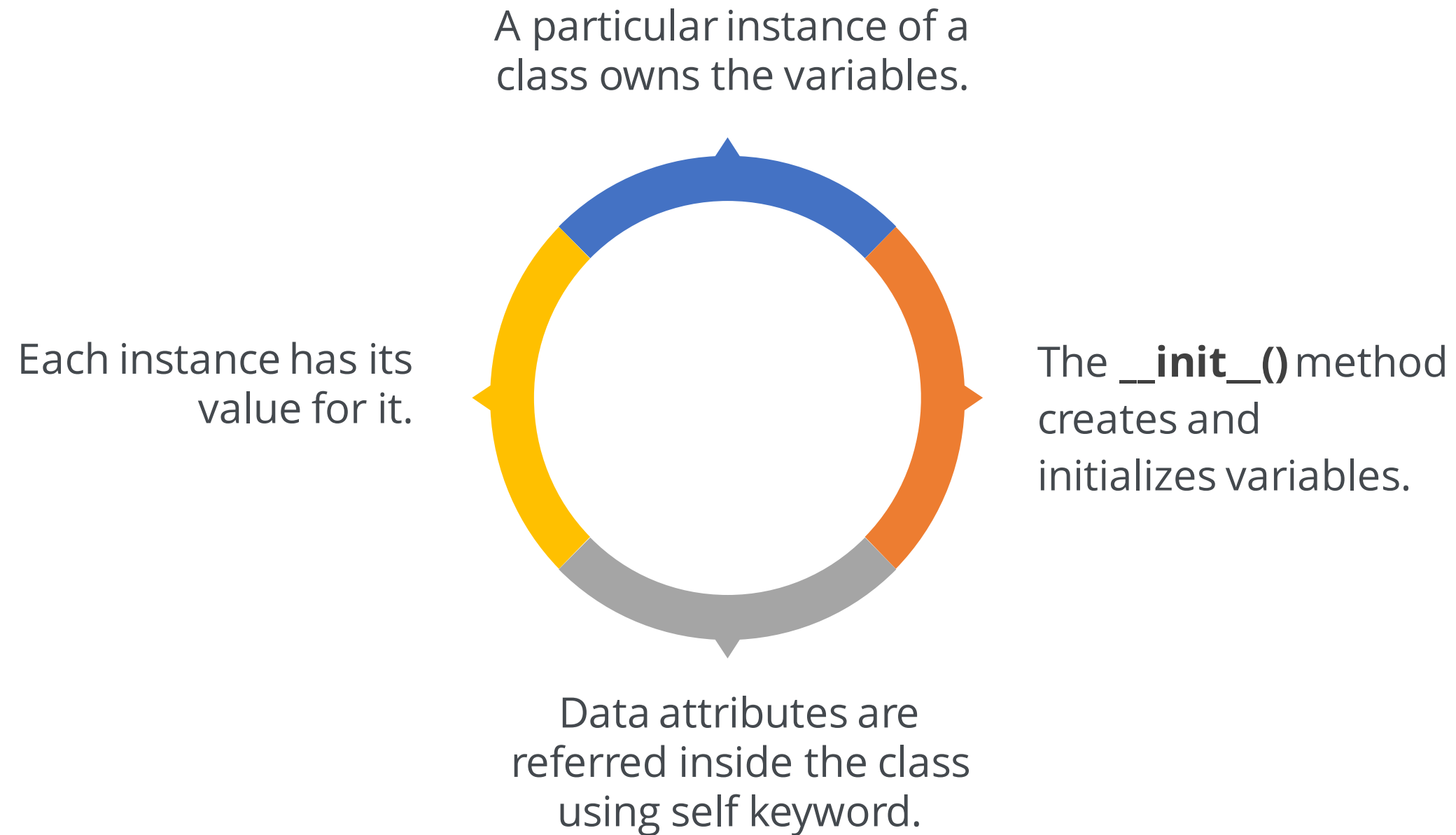
# Types of Attributes

A Python object consists of two types of attributes:



# Data Attributes

The following are some characteristics of data attributes:





# Class Attributes

They are variables that are defined inside a class but outside of any method. Class attributes have the following characteristics:

They are shared by all instances of the class.

They can be accessed from outside the class, which can lead to unexpected behavior.



They can be accessed using the class name or an instance of the class.

They can be used to store constants, default values, or any other data that needs to be shared by all instances of the class.

# Assisted Practice: Create a Class with Attributes And Methods



**Duration: 5 mins**

**Problem Scenario:** Write a program to demonstrate objects and classes using methods and attributes

**Objective:** In this demonstration, you will learn how to create a class and define methods and attributes for it.

## Tasks to Perform:

1. Create a class
2. Declare the desired attributes
3. Create a method that displays the information
4. Initiate the objects
5. Access class attributes and methods through objects

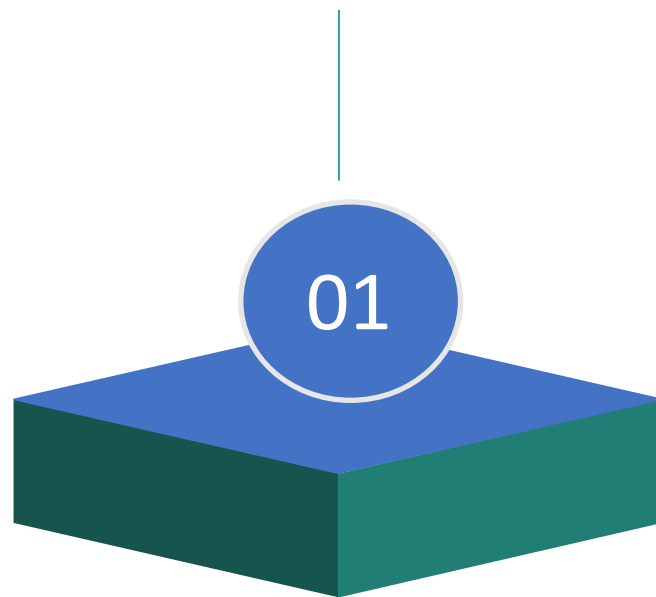


## **Access Modifiers**

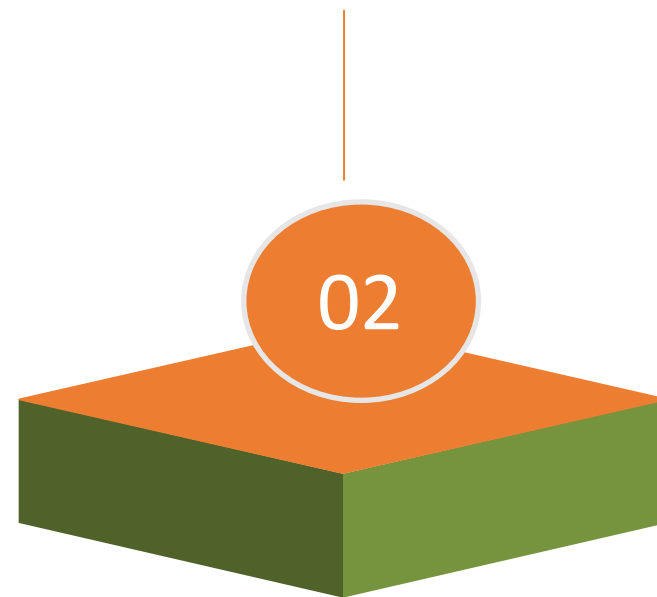
# Access Modifiers

A class in Python has three types of access modifiers. They are special keywords that allow for changing the behavior or properties of class attributes and methods.

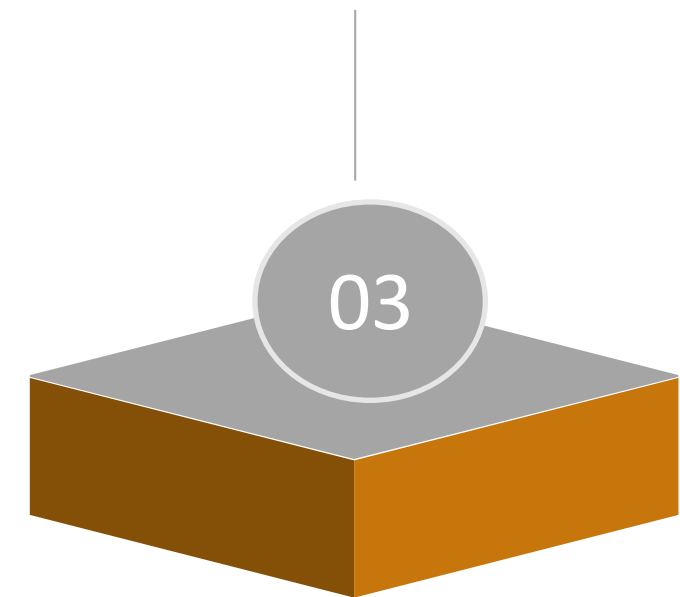
Public access  
modifiers



Protected access  
modifiers



Private access  
modifiers



# Access Modifiers: Public Access Modifier

Public access modifiers have two characteristics:

01

Data members of a class that are declared public can be accessed from any part of the program.

02

All data members and member functions of a class are public by default.

# Public Access Modifier: Example

The following example explains the public access modifier:

## Example

```
class Dog:

    # constructor
    def __init__(self, name, age):

        # public access modifiers
        self.dogName = name
        self.dogage = age
```

# Access Modifiers: Protected Access Modifier

Protected access modifiers have two characteristics:

01

Members of a class that are declared protected are only accessible to a class derived from it.

02

Data members of a class are declared protected by adding a single underscore symbol () before the data member of that class.

# Protected Access Modifier: Example

The following example explains the protected access modifier:

## Example

```
class Dog:  
    # protected access modifiers  
    _name = None  
    _age = None  
    _breed = None
```



# Access Modifiers: Private Access Modifier

A private access modifier is the most secure access modifier.

01

Private members of a class can be accessed within the class only.

02

Data members of a class are declared private by adding a double underscore symbol ( `__` ) before the data member name.

# Private Access Modifier: Example

The following example explains the private access modifier:

Example

```
class Dog:
    # private access modifiers
    __name = None
    __age = None
    __breed = None
```

# Assisted Practice: Access Modifiers



**Duration: 10 mins**

**Objective: To demonstrate public protected and private access modifiers.**

## **Tasks to perform:**

1. Create a parent class with public, private and protected members
2. Create a child class and invoke the public, private and protected members of the parent class
3. Create an object of the child class and call the method to display the data



# Encapsulation



## Discussion

# Encapsulation and Polymorphism



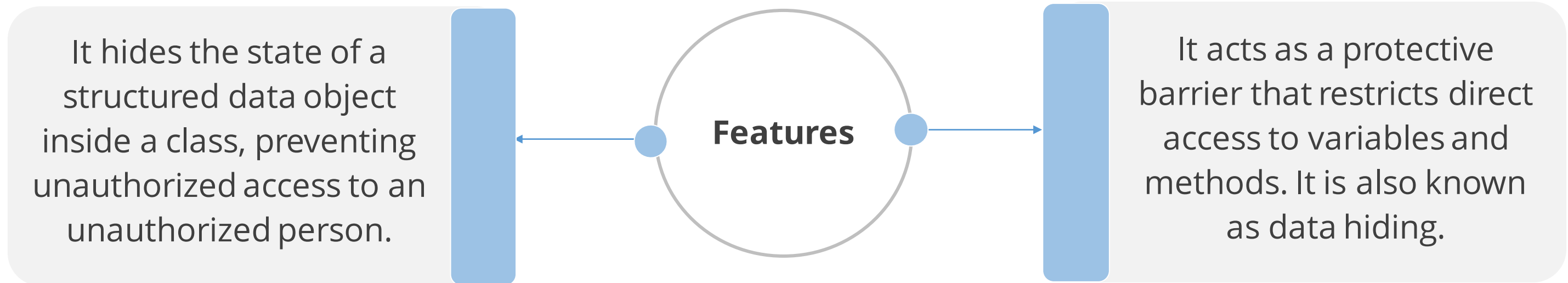
You are working as a data scientist in an application development project where one phase of development is already completed. Once you analyze the project code, you determine that the code could have been written better. Some modules could have been displayed in more than one form instead of coding them separately.

To understand this better, it is important to know about OOPs in detail.

- What is encapsulation?
- What is polymorphism?

# Encapsulation

Encapsulation is the process of binding data members and member functions into a single unit.



# Encapsulation: Example

At a medical store, only the chemist has access to the medicines based on the prescription. This reduces the risk of taking any medicine that is not intended for a patient.

## Example

```
class Encapsulation:  
    def __init__(self, a, b,c):  
        self.public = a  
        self._protected = b  
        self.__private = c
```





# Inheritance

# Inheritance

Inheritance is the process of forming a new class from an existing class or a base class.

Example: A family has three members, father, mother, and son.

Father (Base class)
Tall
Dark

Mother (Base class)
Short
Fair

Also known  
as super  
class

Son (Derived class)
Tall
Fair

Also known  
as sub class

The son is tall and fair. This indicates that he has inherited the features of his father and mother, respectively.

# Types of Inheritance

There are four types of inheritance:

## Single level inheritance:

A class can inherit from only one class.

## Multiple inheritance:

A class can inherit from more than one class.

## Multilevel inheritance:

A derived class is created from another derived class.

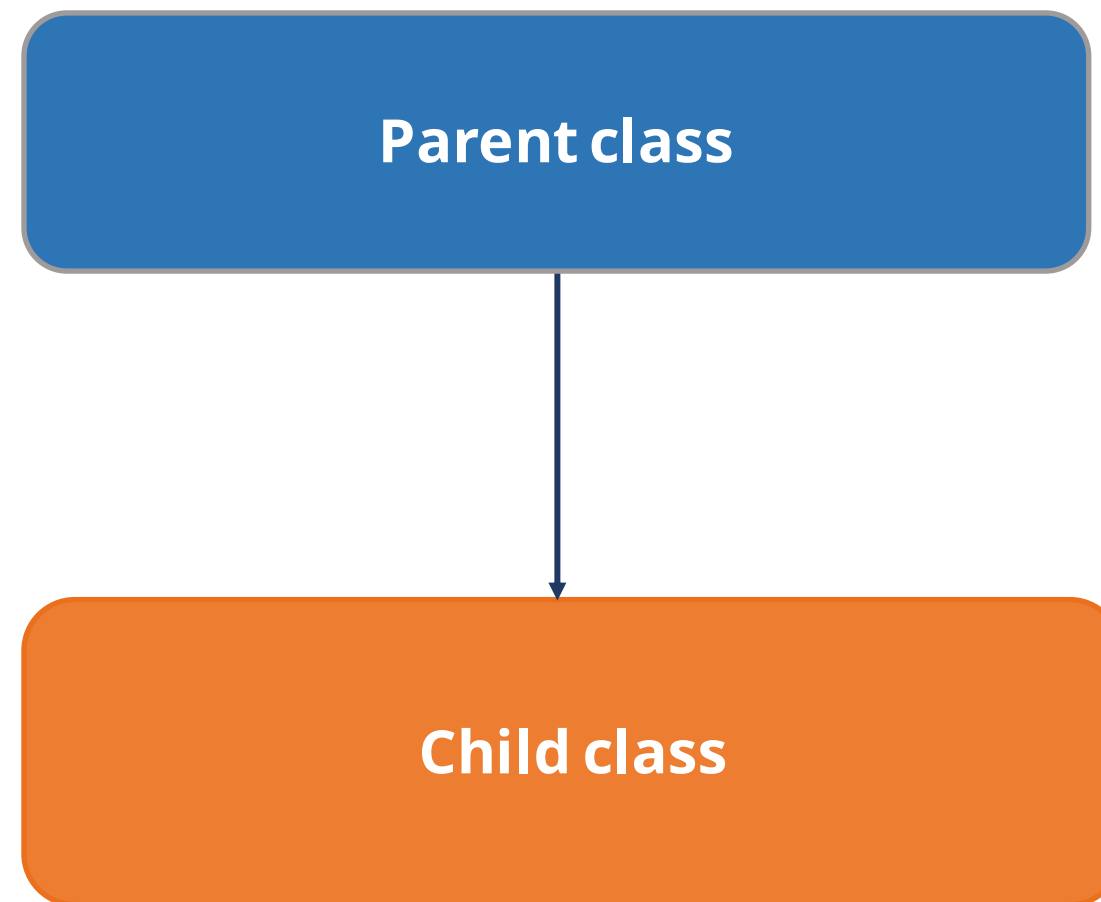
## Hierarchical inheritance:

A base class can have multiple subclasses inherited from it.



# Inheritance: Single Level Inheritance

A class that is derived from one parent class is called single-level inheritance.



# Single Level Inheritance: Example

The following is an example of single level inheritance:

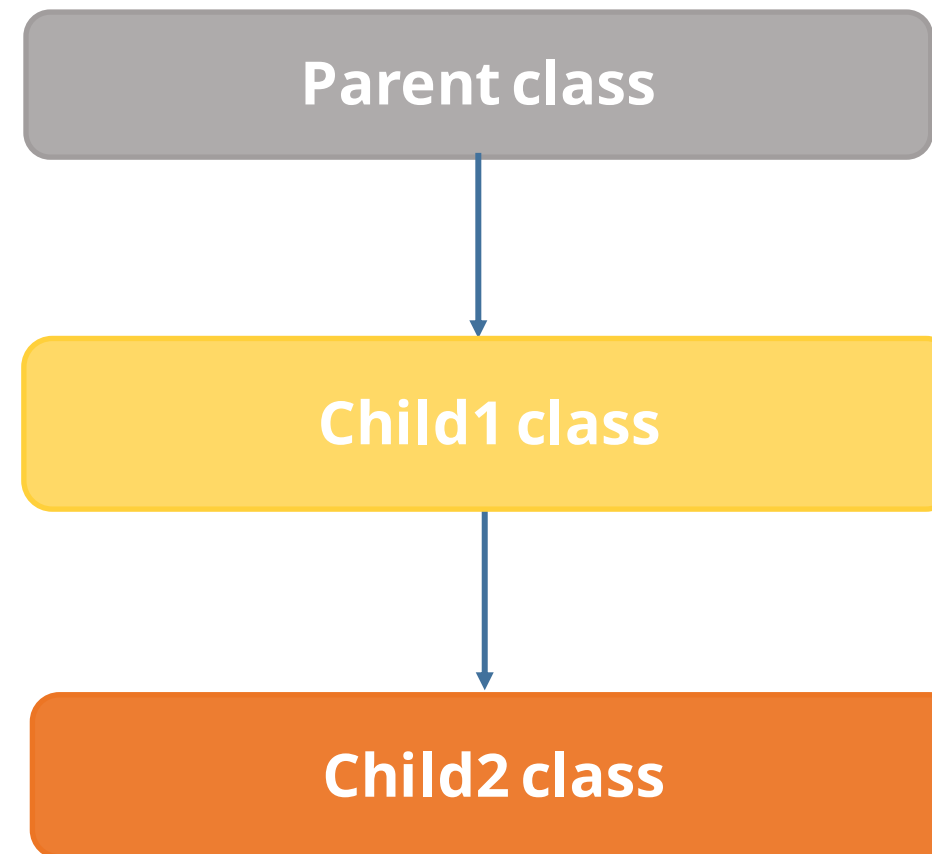
## Example

```
class Parent_class:  
    def parent(self):  
        print("Hey I am the parent class")  
  
class Child_class(Parent_class):  
    def child(self):  
        print("Hey I am the child class derived from the parent")  
  
obj = Child_class()  
obj.parent()  
obj.child()
```

```
Hey I am the parent class  
Hey I am the child class derived from the parent
```

# Inheritance: Multilevel Inheritance

In multilevel inheritance, the features of the parent class and the child class are further inherited into the new child class.



# Multilevel Inheritance: Example

An example of multilevel inheritance is shown below:

## Example

```
class Parent_class:
    def parent(self):
        print("Hey I am the parent class")

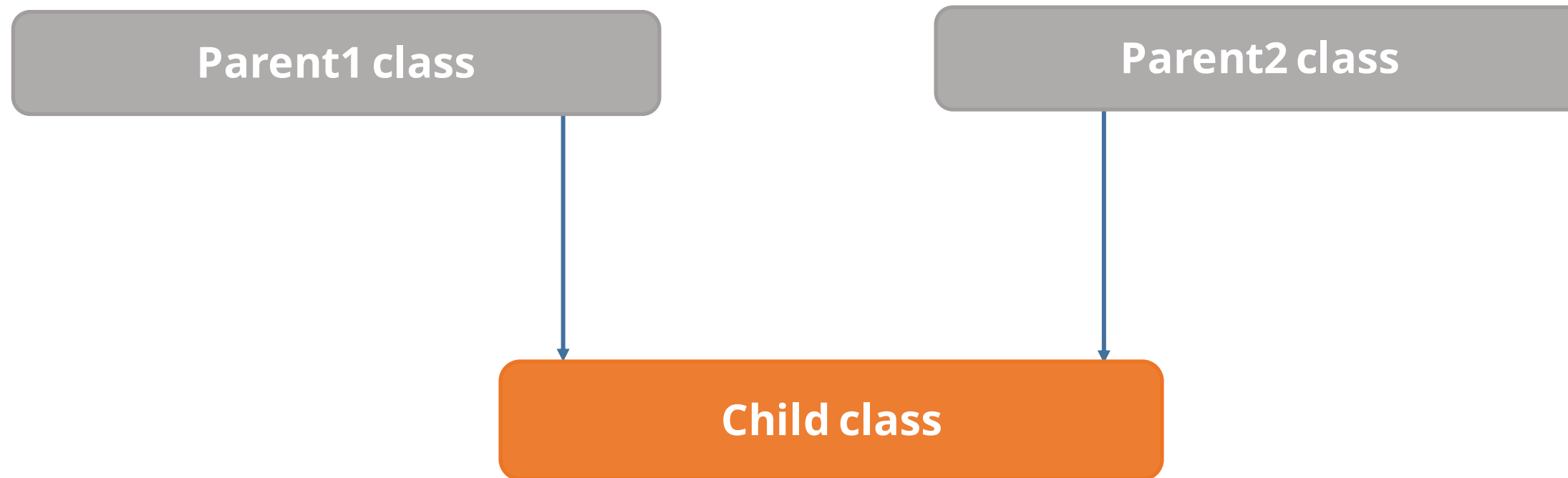
class Child1_class(Parent_class):
    def child1(self):
        print("Hey I am the child1 class derived from the parent")

class Child2_class(Child1_class):
    def child2(self):
        print("Hey I am the child2 class derived from the child1")
obj = Child2_class()
obj.parent()
obj.child1()
obj.child2()
```

```
Hey I am the parent class
Hey I am the child1 class derived from the parent
Hey I am the child2 class derived from the child1
```

# Inheritance: Multiple Inheritance

A class that is derived from more than one parent class is called multiple inheritances.





# Multiple Inheritance: Example

An example of multiple inheritance is given below:

## Example

```
class Father:
    fathername = ""
    def fatherName(self):
        print("Hey I am the father, and my name is : " ,self.fathername)

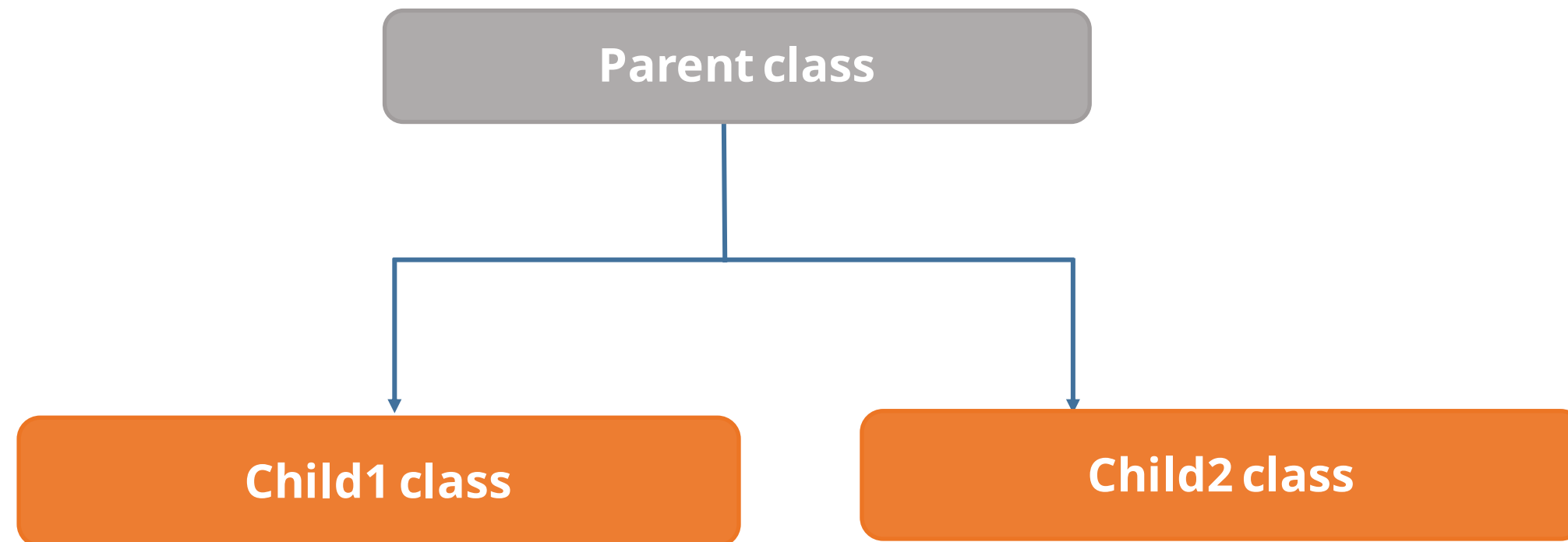
class Mother:
    mothername = ""
    def mother(self):
        print("Hey I am the mother, and my name is : ",self.mothername)

class Child(Mother, Father):
    def parents(self):
        print("My Father's name is :", self.fathername)
        print("My Mother's name is :", self.mothername)
obj = Child()
obj.fathername = "Ryan"
obj.mothername = "Emily"
obj.parents()
```

```
My Father's name is : Ryan
My Mother's name is : Emily
```

# Inheritance: Hierarchical Inheritance

Hierarchical inheritance is the process of creating multiple derived classes from a single base class.



# Hierarchical Inheritance: Example

The following example explains hierarchical inheritance:

## Example

```
: class Parent:
    def Parent_func1(self):
        print("Hello I am the parent.")

    class Child1(Parent):
        def Child_func2(self):
            print("Hello I am child 1.")

    class Child2(Parent):
        def Child_func3(self):
            print("Hello I am child 2.")

object1 = Child1()
object2 = Child2()
object1.Parent_func1()
object1.Child_func2()
object2.Parent_func1()
object2.Child_func3()
```

```
Hello I am the parent.
Hello I am child 1.
Hello I am the parent.
Hello I am child 2.
```

# Assisted Practice: Inheritance



**Duration: 5 mins**

**Problem Scenario:** Write a program to demonstrate inheritance using classes, objects, and methods

**Objective:** In this demonstration, we will learn how to work with inheritance.

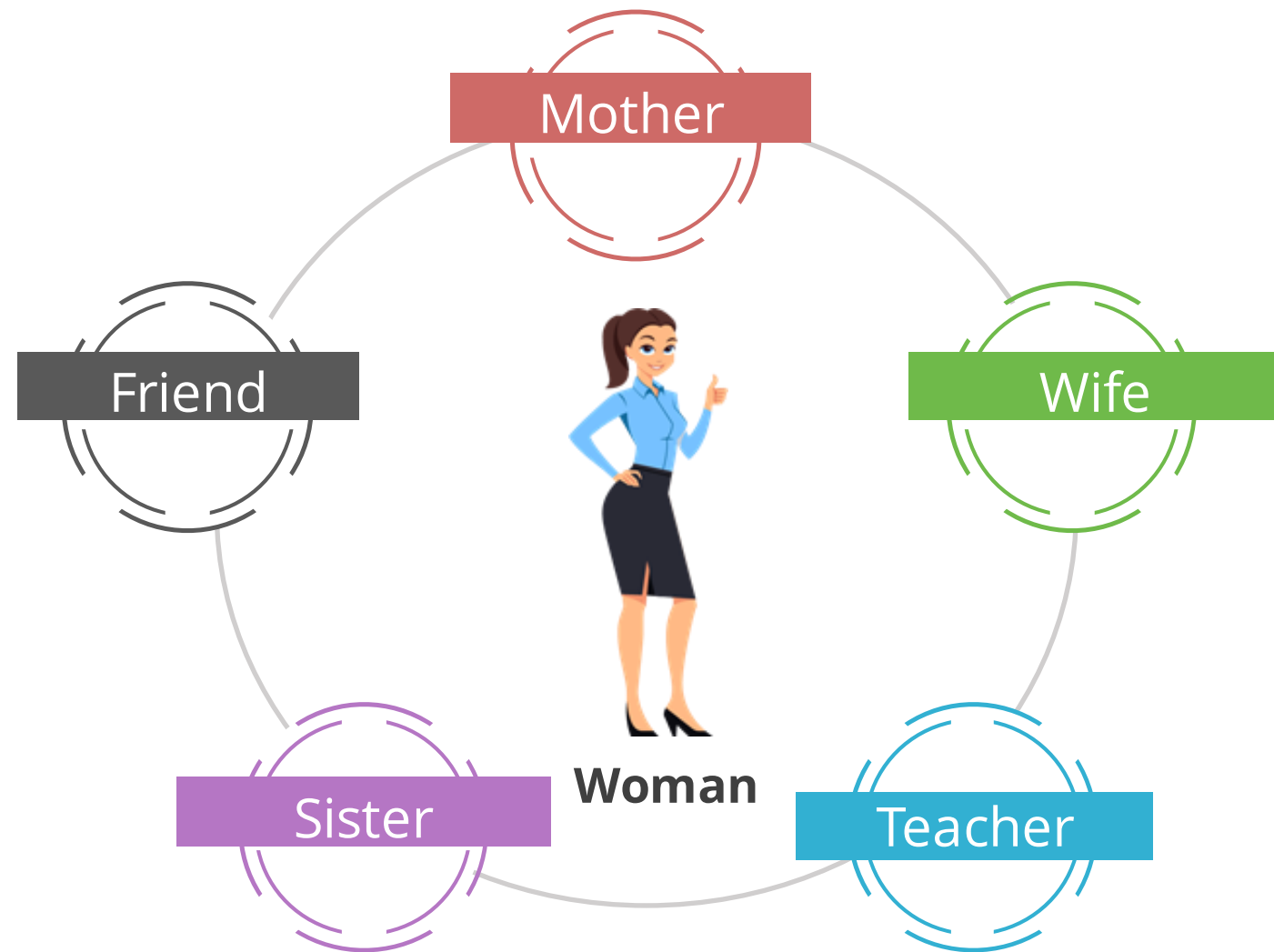
**Tasks to perform:**

1. Create 2 base classes
2. Create a derived class that derives the attributes of the parent class
3. Create the objects of the derived class and retrieve the attributes of the parent class



# Polymorphism

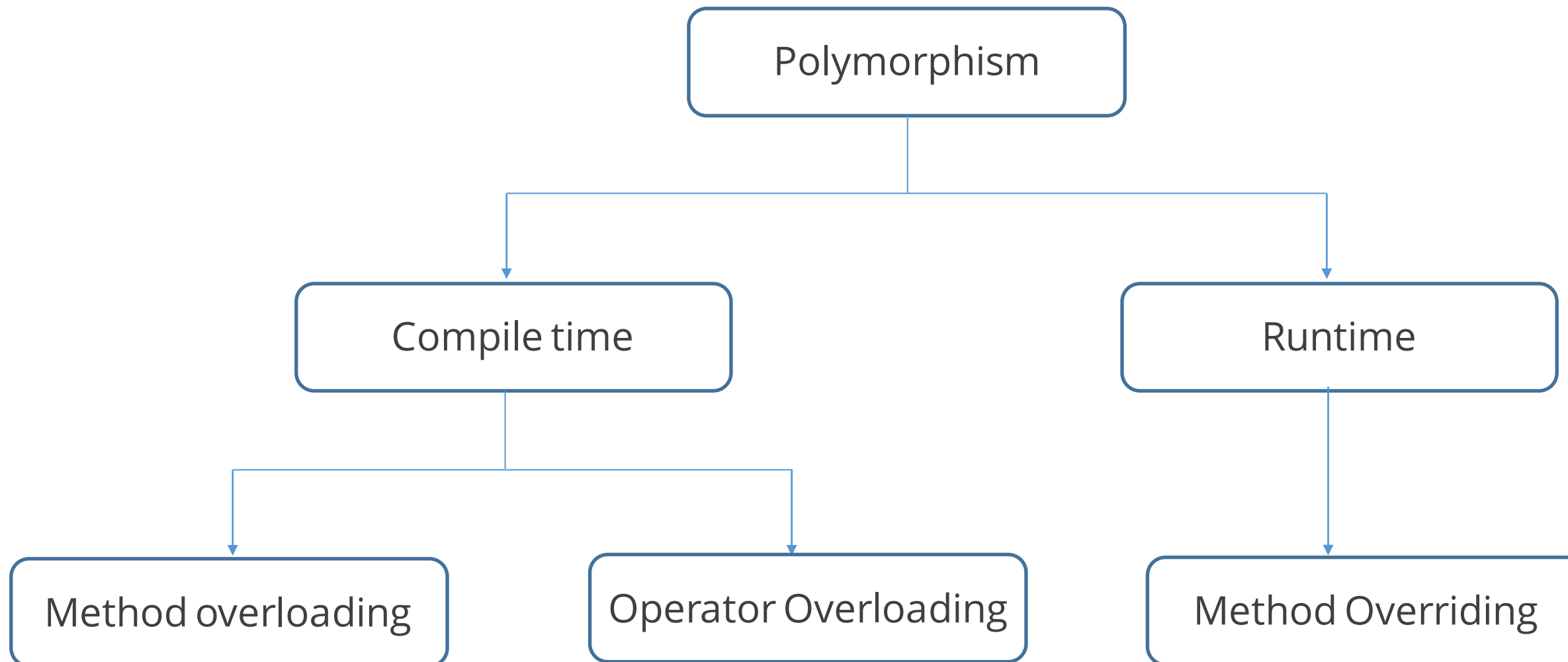
# Polymorphism



- Polymorphism is a Greek word that means many shapes.
- The ability of a message to be displayed in more than one form is known as polymorphism.
- Example: A woman can be a mother, a wife, a teacher, a sister, and a friend at the same time.

# Types of Polymorphism

The types of polymorphism are mentioned below:



# Assisted Practice: Polymorphism



**Duration: 10 mins**

**Problem Scenario:** Write a program to demonstrate polymorphism using classes, objects, and methods

**Objective:** In this demonstration, we will learn how to perform polymorphism.

**Tasks to perform:**

1. Create two classes that contain the same method names
2. Create the objects of the base class and call the methods





# Abstraction

# Abstraction

It allows the representation of complex systems or ideas in a simplified manner.



Example: When one presses a key on the keyboard, the relevant character appears on the screen. One doesn't have to know how exactly this works. This is called abstraction.

# Abstraction

In Python, abstraction works by incorporating abstract classes and methods.

Abstract Class is a class specified in the codes containing abstract methods.



Abstract methods do not have implementation in the abstract class. All implementation is done inside the sub classes

# Abstraction

In Python, abstraction works by incorporating abstract classes and methods.

An abstract class can only be inherited.



Only an object of the derived class can be used to access the features of the abstract class.

# Encapsulation and Polymorphism

You are working as a data scientist in an application development project where one phase of development is already completed. Once you analyze the project code, you determine that the code could have been written better. Some modules could have been displayed in more than one form instead of coding them separately.

To understand this better, it is important to know about OOPs in detail.

- What is encapsulation?

Answer: Encapsulation is the process of binding data members and member functions into a single unit.

- What is polymorphism?

Answer: The ability of a message to be displayed in more than one form is known as polymorphism.



# Assisted Practice: Abstraction



**Duration: 5 mins**

**Problem Scenario:** Write a program to demonstrate abstraction in Python

**Objective:** In this demonstration, we will learn how to implement abstraction.

**Tasks to perform:**

1. Import the necessary packages for creating an abstract class
2. Create a base class containing abstract methods and derived classes containing non-abstract methods
3. Implement the methods of abstract class using objects

# Key Takeaways

- Object-oriented programming aims to implement real-world entities such as inheritance, hiding, and polymorphism in programming.
- An object is an instance of a class.
- A class is a blueprint for an object. A class is a definition of objects with the same properties and methods.
- A class in Python has three types of access modifiers: public, protected, and private.





## Knowledge Check



## Knowledge Check

1

An object is an instance of a(n) \_\_\_\_\_.

- A. Method
- B. Attribute
- C. Class
- D. Function



Knowledge  
Check  
1

An object is an instance of a(n) \_\_\_\_\_.

- A. Method
- B. Attribute
- C. Class
- D. Function

---

The correct answer is **C**

---

**An object is an instance of a class.**



## Knowledge Check 2

Which of the following is NOT an OOPs concept?

- A. Inheritance
- B. Compilation
- C. Abstraction
- D. Encapsulation



## Knowledge Check 2

Which of the following is NOT an OOPs concept?

- A. Inheritance
- B. Compilation
- C. Abstraction
- D. Encapsulation

---

The correct answer are **B**

---

**There are four OOPS concepts: Inheritance, Encapsulation, Polymorphism, and Abstraction.**



**Knowledge  
Check  
3**

**Which of the following is a type of polymorphism? (Select all that apply)**

- A. Compile time polymorphism
- B. Runtime polymorphism
- C. Multiple polymorphism
- D. Multilevel polymorphism



Knowledge  
Check  
3

Which of the following is a type of polymorphism? (Select all that apply)

- A. Compile time polymorphism
- B. Runtime polymorphism
- C. Multiple polymorphism
- D. Multilevel polymorphism



---

The correct answer is **A and B**

---

**The types of polymorphism are compile time polymorphism and runtime polymorphism.**