

# Programming Refresher



# OOP Concepts with Python



# Learning Objectives

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

- Apply the principles of object-oriented programming to structure and organize code more effectively
- Create and implement objects and classes in Python to solve practical problems
- Develop and utilize methods, attributes, and access modifiers within Python classes to manage class functionality and data protection effectively
- Apply abstraction and encapsulation in Python with practical examples to improve code modularity and security



## Business Scenario

ABC, a banking firm, is developing a banking management system application. This application includes customer information accessible to bank employees. Employees can access and edit this information based on customer requests. However, the current application lacks security, allowing workers at all levels to access classified data. Additionally, the bank wants to customize the application for specific branches.

To address these issues, the firm plans to update the application to ensure customer details are available to employees. Critical information will be accessible only to senior officials. The update will apply object-oriented programming (OOP) concepts, including encapsulation and abstraction, along with methods, attributes, and access modifiers.

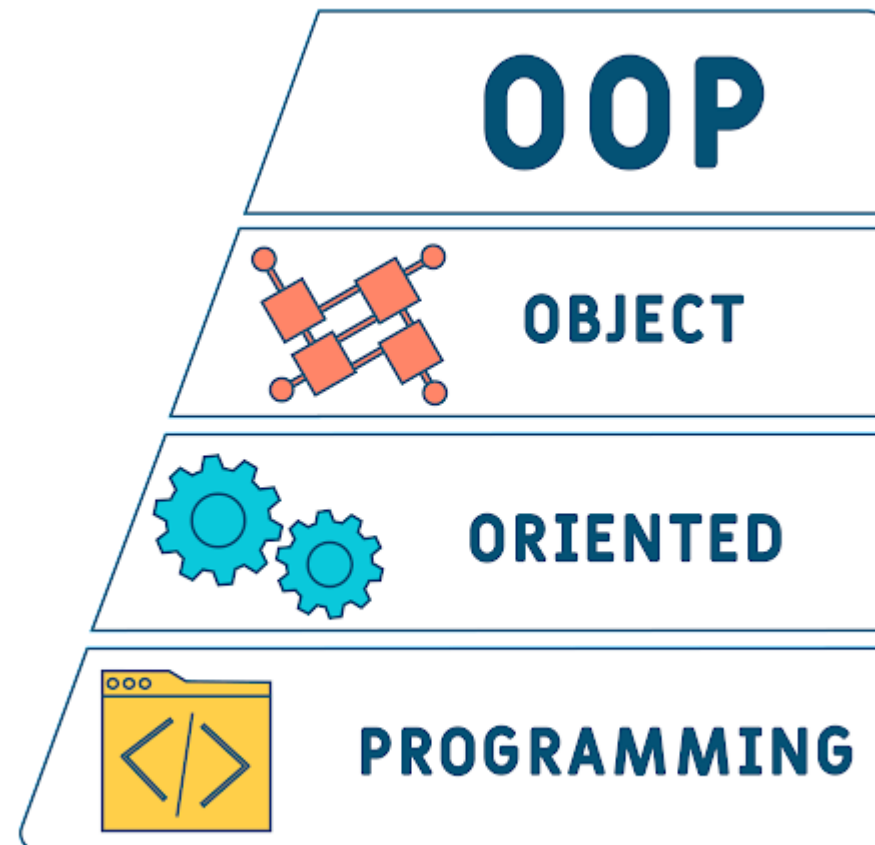




# **Object-Oriented Programming (OOP) Language**

# What Is OOP?

Object-Oriented Programming (OOP) refers to languages that use objects in programming. It aims to implement real-world entities such as inheritance, information hiding, and polymorphism.



# OOP: Concepts

The four concepts of object-oriented programming are:

Encapsulation

Inheritance

Polymorphism

Abstraction

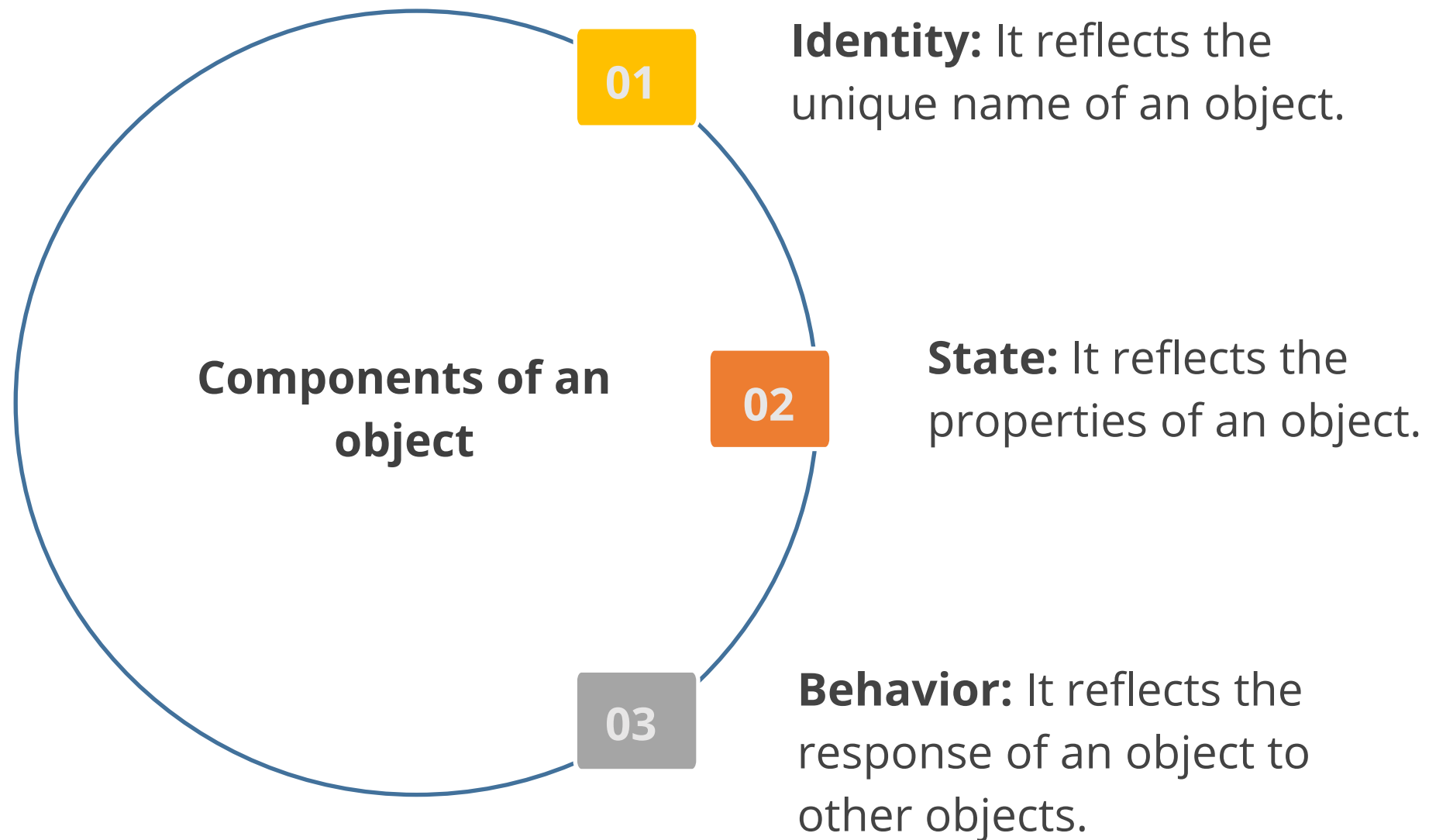


# **Objects and Classes**



# Objects

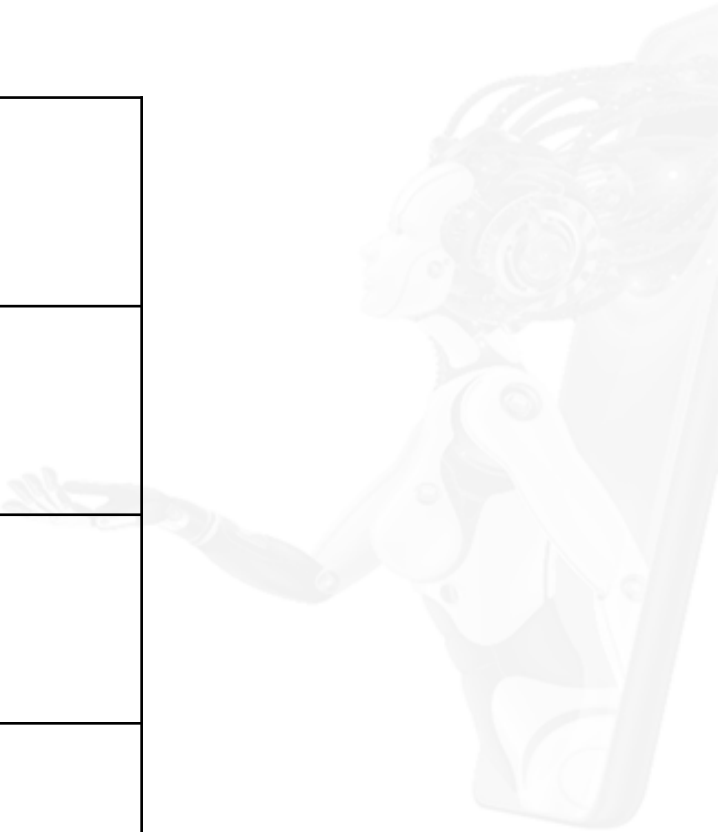
An object represents a real-world entity that can be distinctly identified. It consists of the following components:



# Objects: Example

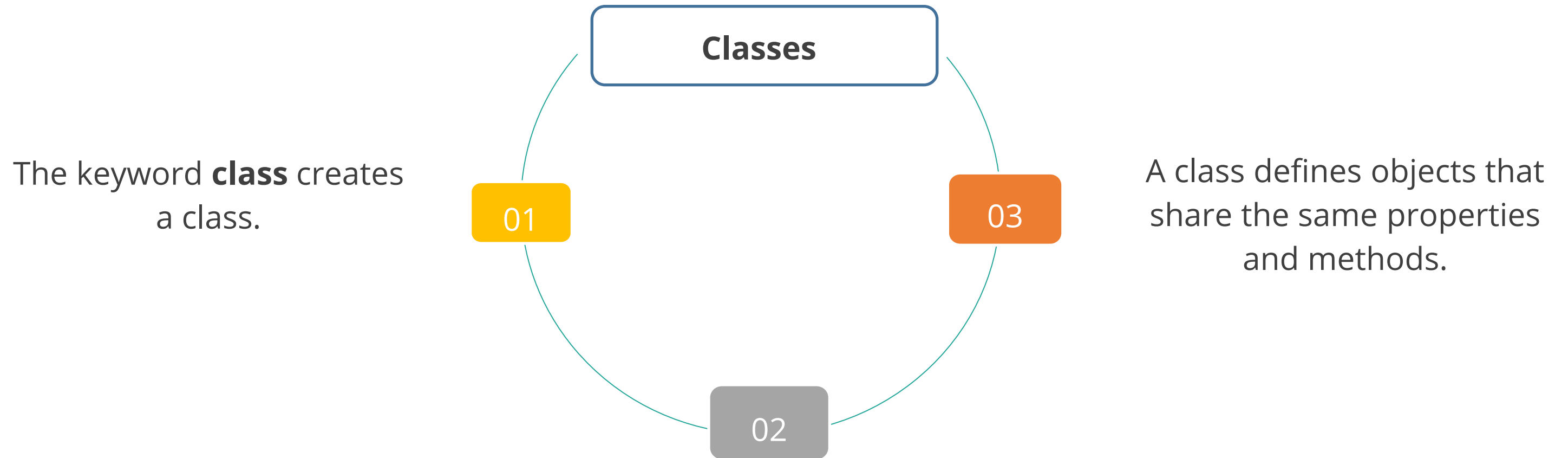
Object: Dog

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

## Example

```
class Dog:  
    pass
```

Here, the **class** keyword defines an empty **class** named **Dog**.

An instance is a specific object created from a class.



## Methods and Attributes

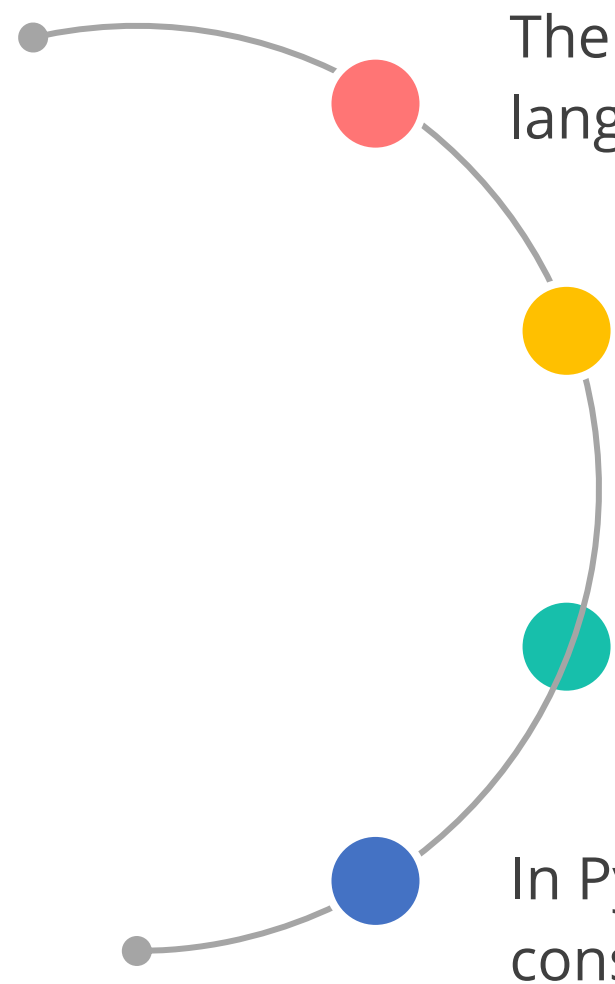
# Methods: Syntax

Methods are functions defined inside a class. Objects invoke these methods to perform actions on other objects.

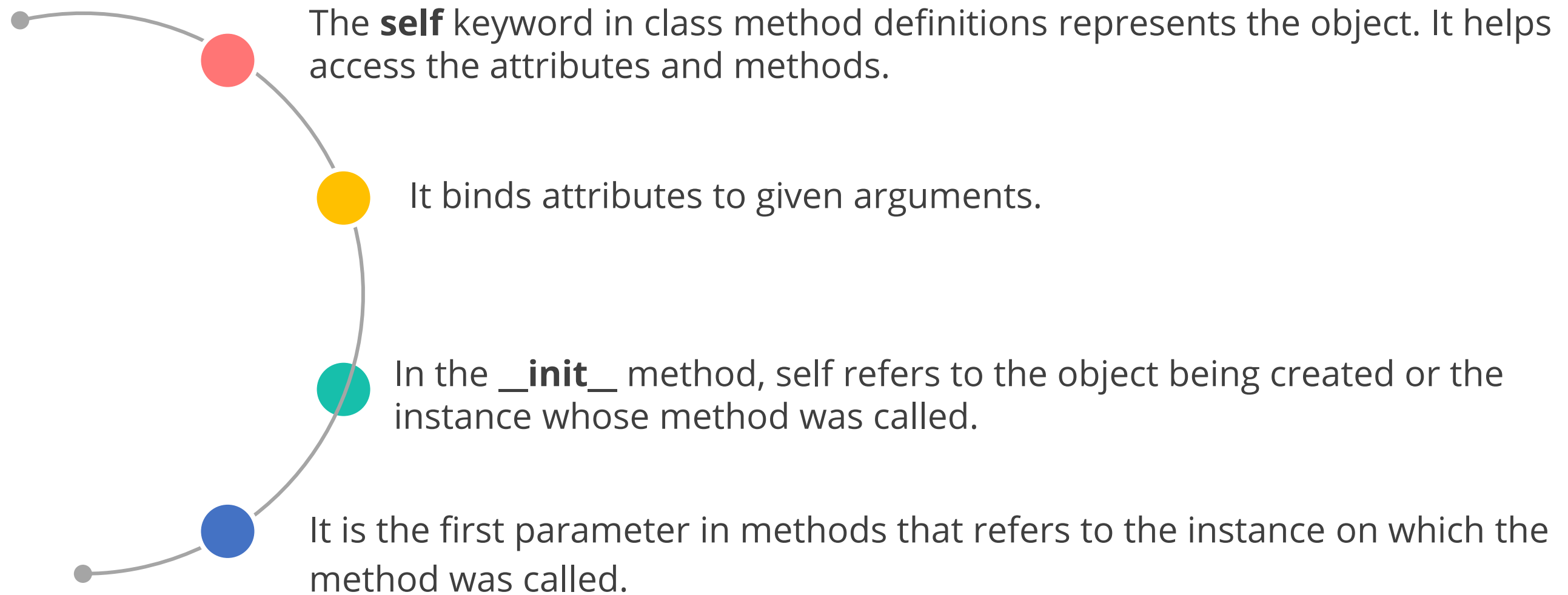
```
# Define a new class with method and its parameters
class ClassName:
    def method_name(self, param1, param2, ...):
        # Method body
        # Operations using self and parameters
        return result
```

- The **ClassName:** class defines a new class named **ClassName**.
- **def method\_name(self, param1, param2, ...):** defines a method named **method\_name** in the class. The first parameter, **self**, refers to the class instance.
- **# Method body** is the indented lines after the method definition where operations are performed.

# Methods: The `__init__` method

- 
- The **`__init__`** method in Python is like constructors in other programming languages.
  - This method runs every time you create an object of the class.
  - The **`__init__`** method can take multiple parameters to initialize the object's attributes. It is used only within the class.
  - In Python, you can use default arguments instead of always passing values to constructors.

# Methods: Self





# Methods: Example

The below example showcases the definition of a class with an **\_\_init\_\_** method that initializes an instance attribute.

**\_\_init\_\_** is a method automatically called when a new object is created.

```
# 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, **self** accesses the attributes and methods of the instance that invoked the method.

# Instantiating Objects

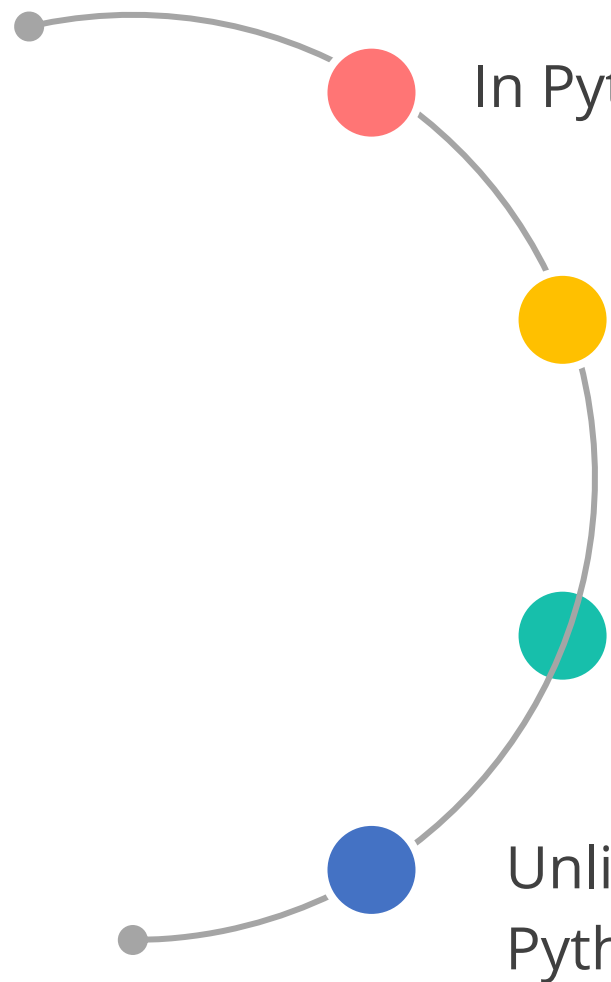
Instantiating objects means creating instances of a class. To instantiate a class, call the class like a function, passing arguments as defined in the `__init__` method.

Example: Create an object for student class

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

- Here **st1** is an object of the **class student**.
- The values passed to the class are the arguments defined in the **`__init__`** method.

# Deleting Instances

- 
- In Python, you do not need to explicitly delete an object after use.
  - Python automatically releases memory when all references to an object are no longer needed.
  - Python uses automatic garbage collection.
  - Unlike other programming languages (e.g., C++), destructors are not needed in Python classes.

# Attributes

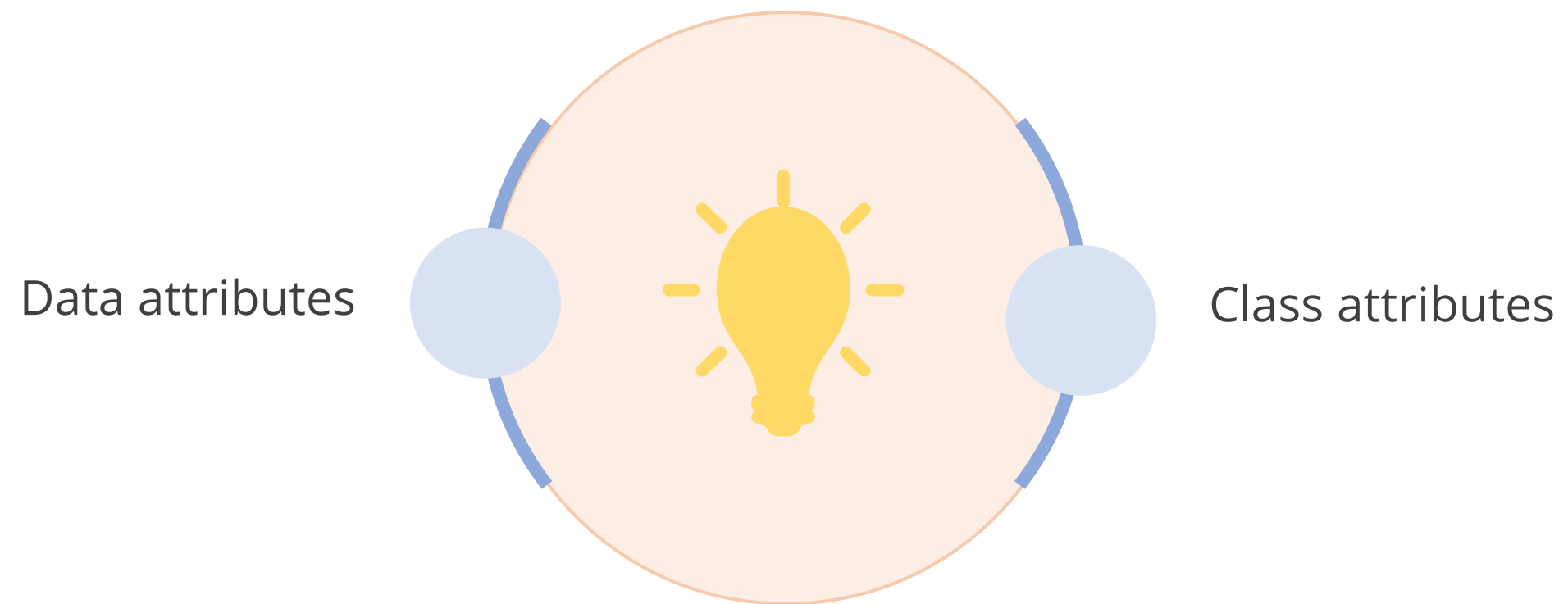
Attributes are non-method data stored by objects.

## Object: Dog

Identity	Attribute
Name of the dog	Breed
	Age
	Color

# Types of Attributes

A Python object has two types of attributes:

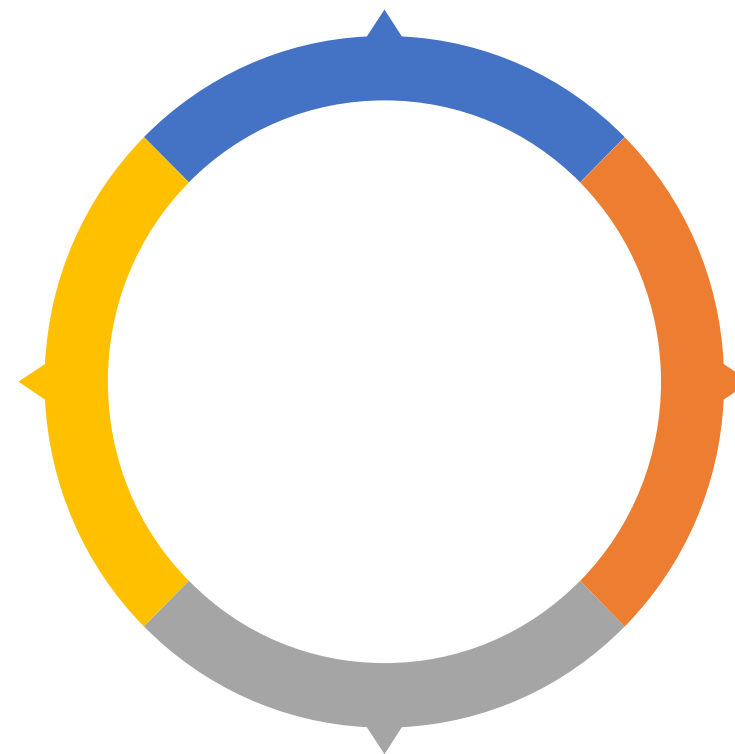


# Data Attributes

Characteristics of data attributes include:

Each instance of a class has its own variables.

Each instance has its value for each variable.



The **`__init__`** method creates and initializes variables.

Data attributes are referred to inside the class using the **`self`** keyword.

# Class Attributes

Class attributes are variables defined inside a class but outside any method. They 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 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.

## 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

**Note:** Please refer to the **Reference Material** section to download the **Jupyter Notebook** files for the mentioned topic.



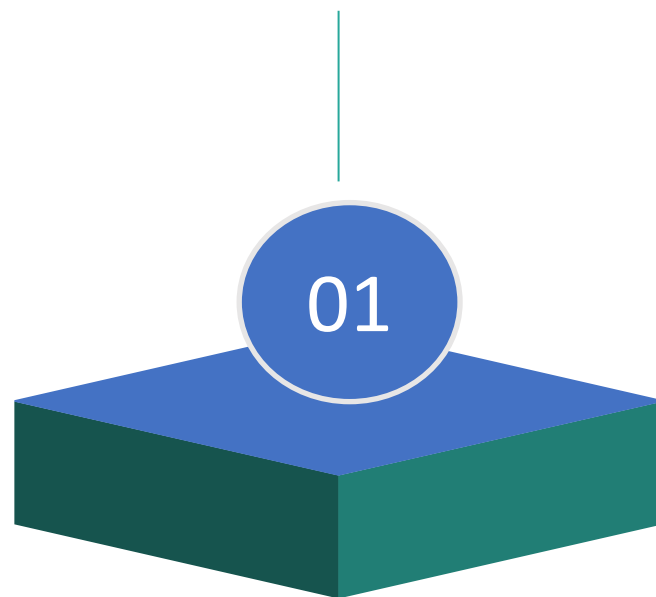


## **Access Modifiers**

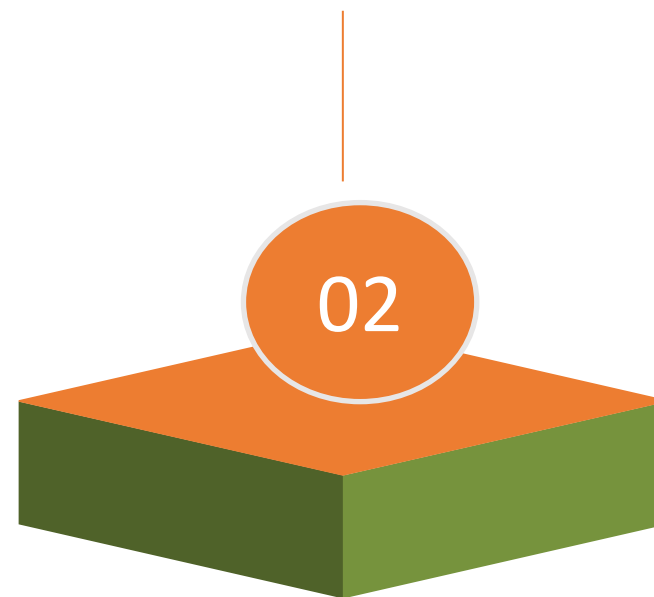
# Access Modifiers

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.

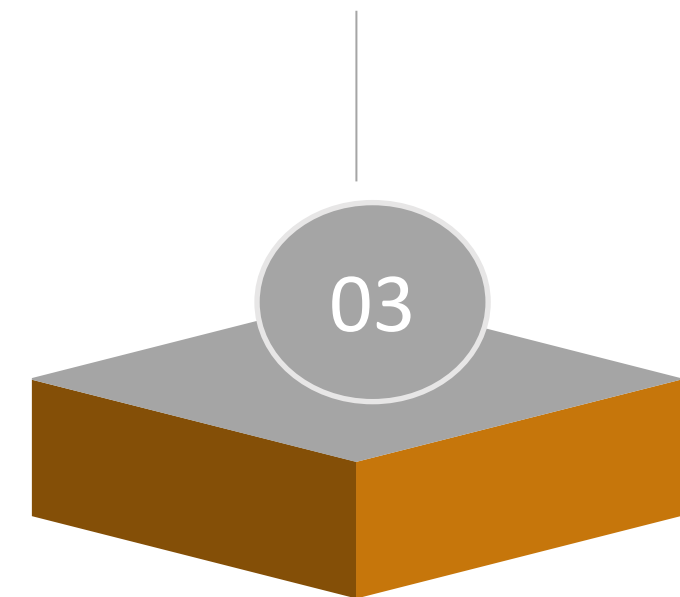
Public access  
modifiers



Protected access  
modifiers



Private access  
modifiers



# Access Modifiers: Public Access Modifier

Public access modifiers have two characteristics:

01

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

02

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

# Public Access Modifier: Example

## 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

## 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 only be accessed within the class.

02

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

# Private Access Modifier: Example

## Example

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



## Assisted Practice: Access Modifiers



**Duration: 10 mins**

**Problem Scenario:** Write a program To demonstrate public, protected, and private access modifiers

**Objective:** In this demonstration, you will learn how to create to demonstrate objects and classes using methods and attributes.

### 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

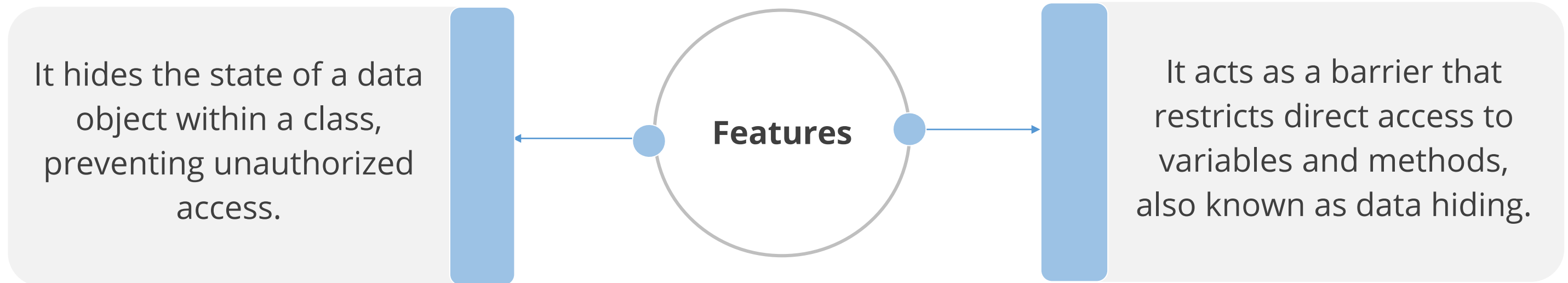
**Note:** Please refer to the **Reference Material** section to download the **Jupyter Notebook** files for the mentioned topic.



# Encapsulation

# Encapsulation

Encapsulation binds data members and functions into a single unit.



# Encapsulation: Example

At a medical store, only the pharmacist has access to the medicines based on the prescription. This reduces the risk of taking medicine 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 base class.

Example: A family has three members: a father, a mother, and a 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, indicating he inherited these features from his parents. Inheritance also refers to superclass and subclass.

# 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.

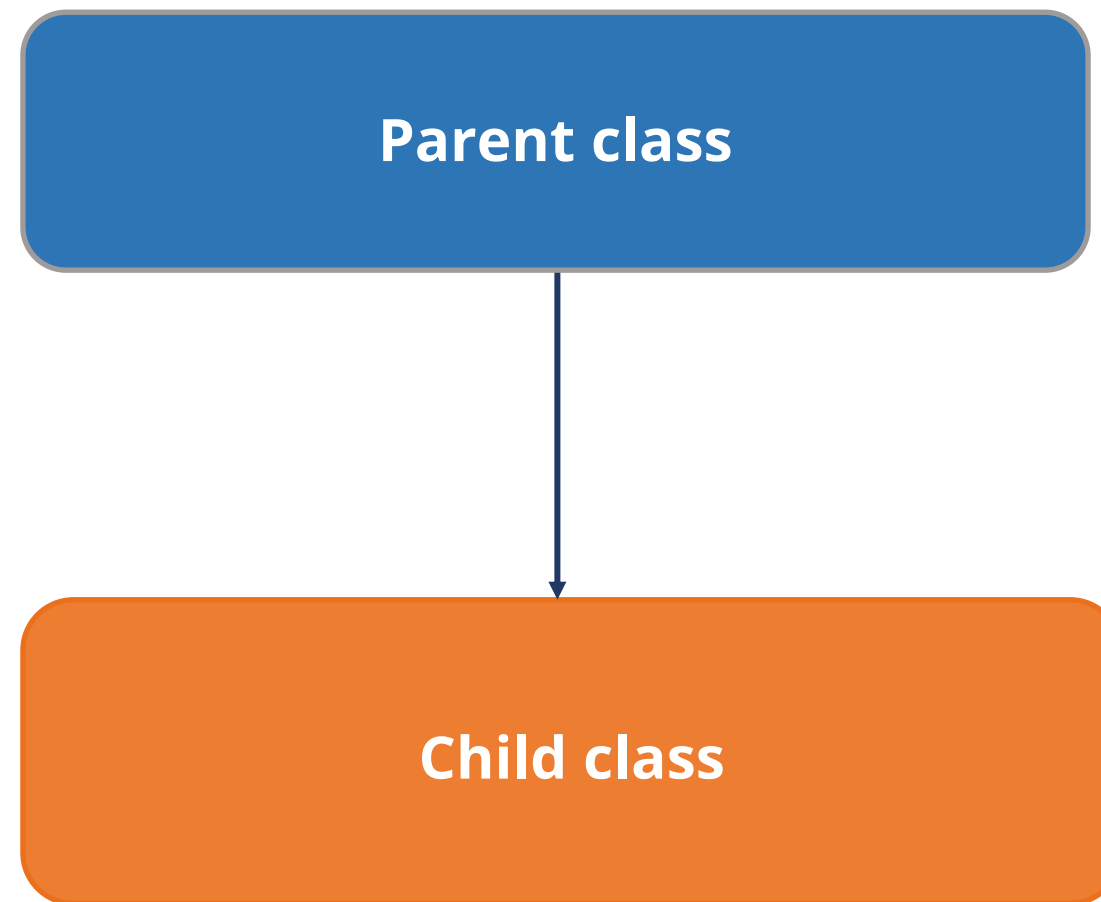
## Hierarchical inheritance

A base class can have multiple subclasses inherited from it.



# Inheritance: Single Level Inheritance

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





# Single Level Inheritance: Example

## Example

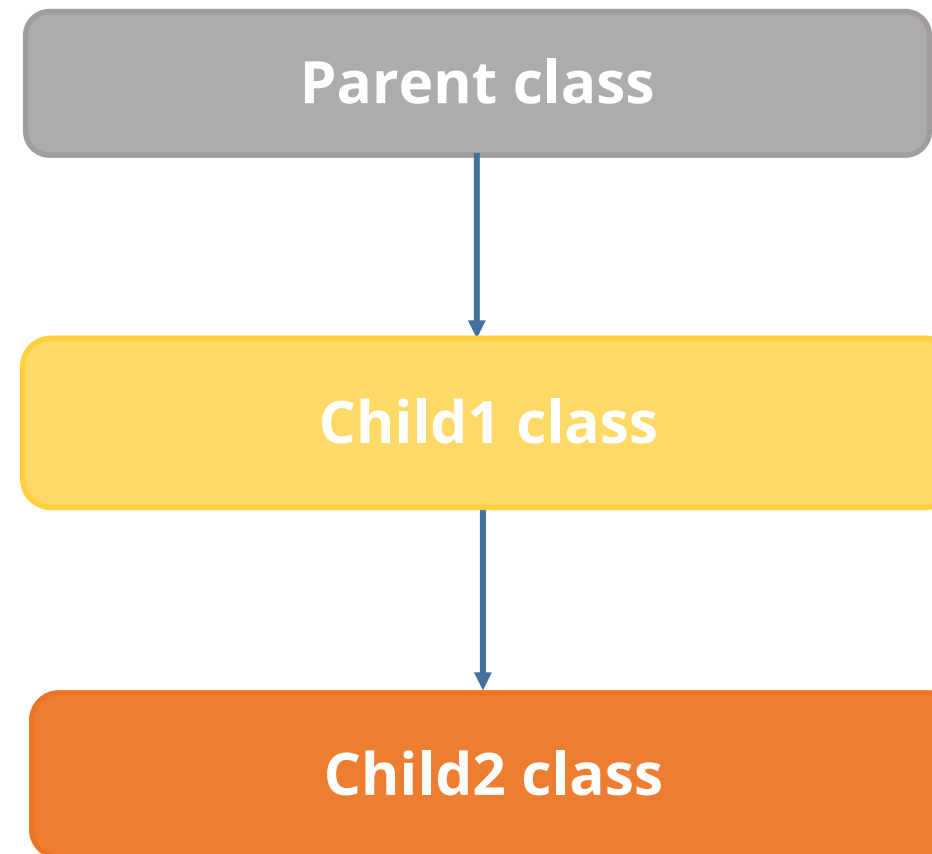
```
class Parent_class:  
    def parent(self):  
        print("Hey I am the parent class")  
  
class Child_class(Parent_class):  
    def child(self):  
        print("Hey I am the child class derived from the parent")  
  
obj = Child_class()  
obj.parent()  
obj.child()
```

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 by the new child class.



# Multilevel Inheritance: Example

## Example

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

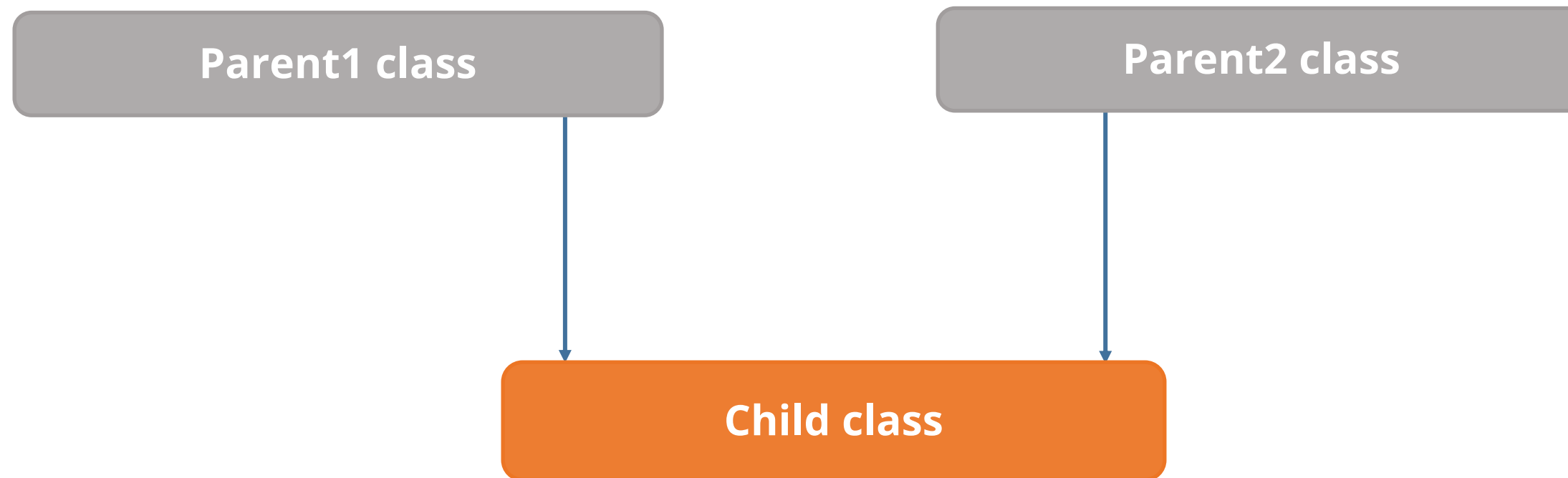
class Child1_class(Parent_class):
    def child1(self):
        print("Hey I am the child1 class derived from the parent")

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 derived from more than one parent class is called multiple inheritance.



# Multiple Inheritance: Example

## Example

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

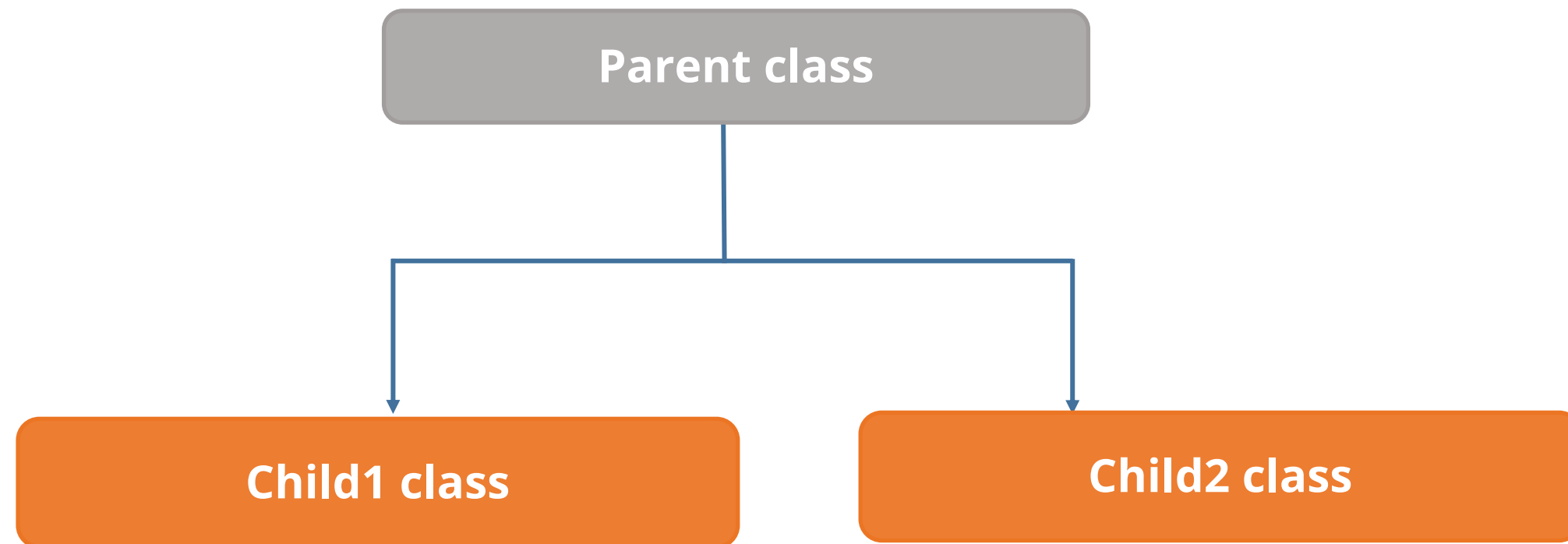
class Mother:
    mothername = ""
    def mother(self):
        print("Hey I am the mother, and my name is : ",self.mothername)

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

## 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

#### *Note*

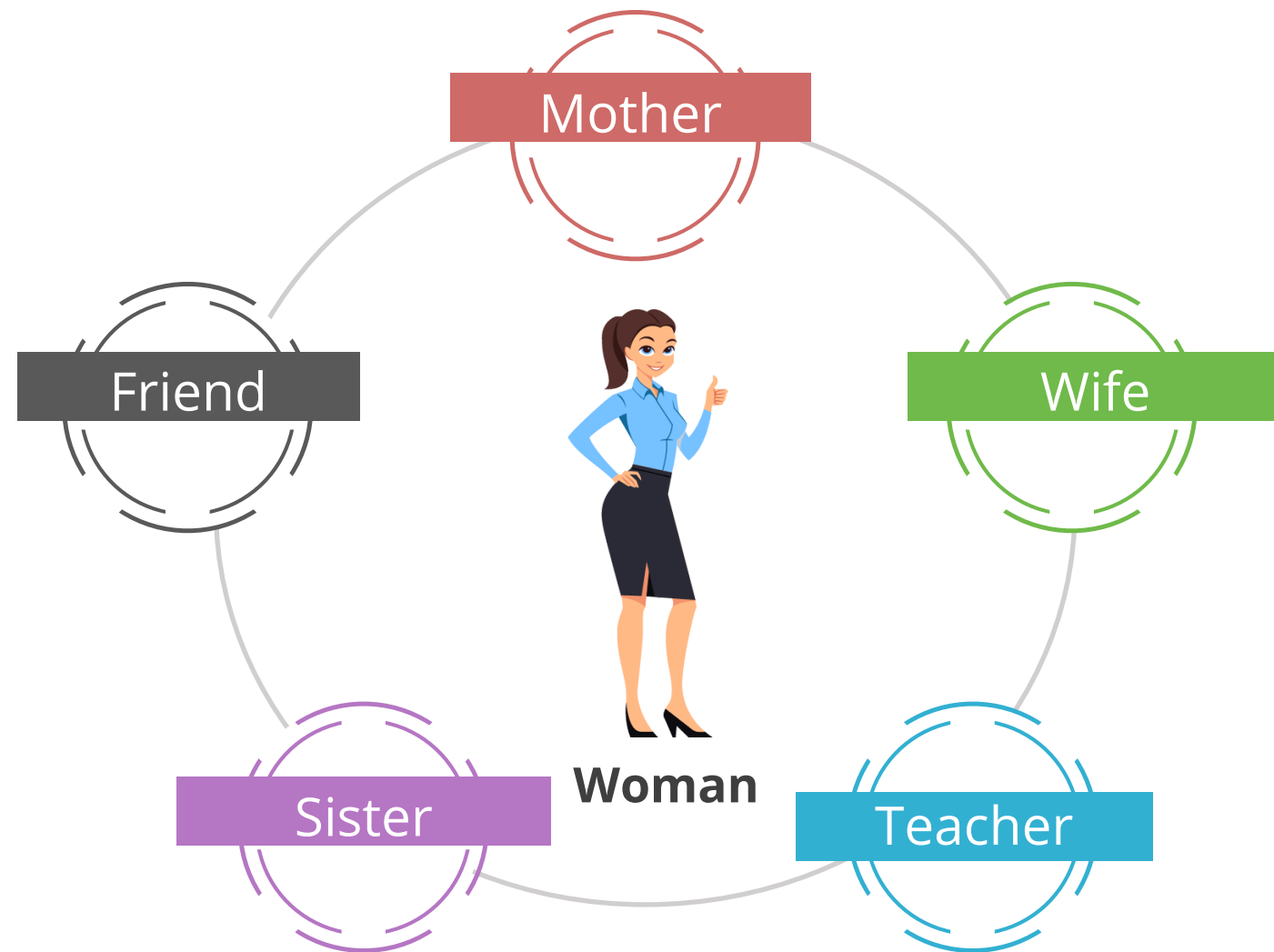
**Note:** Please refer to the **Reference Material** section to download the **Jupyter Notebook** files for the mentioned topic.





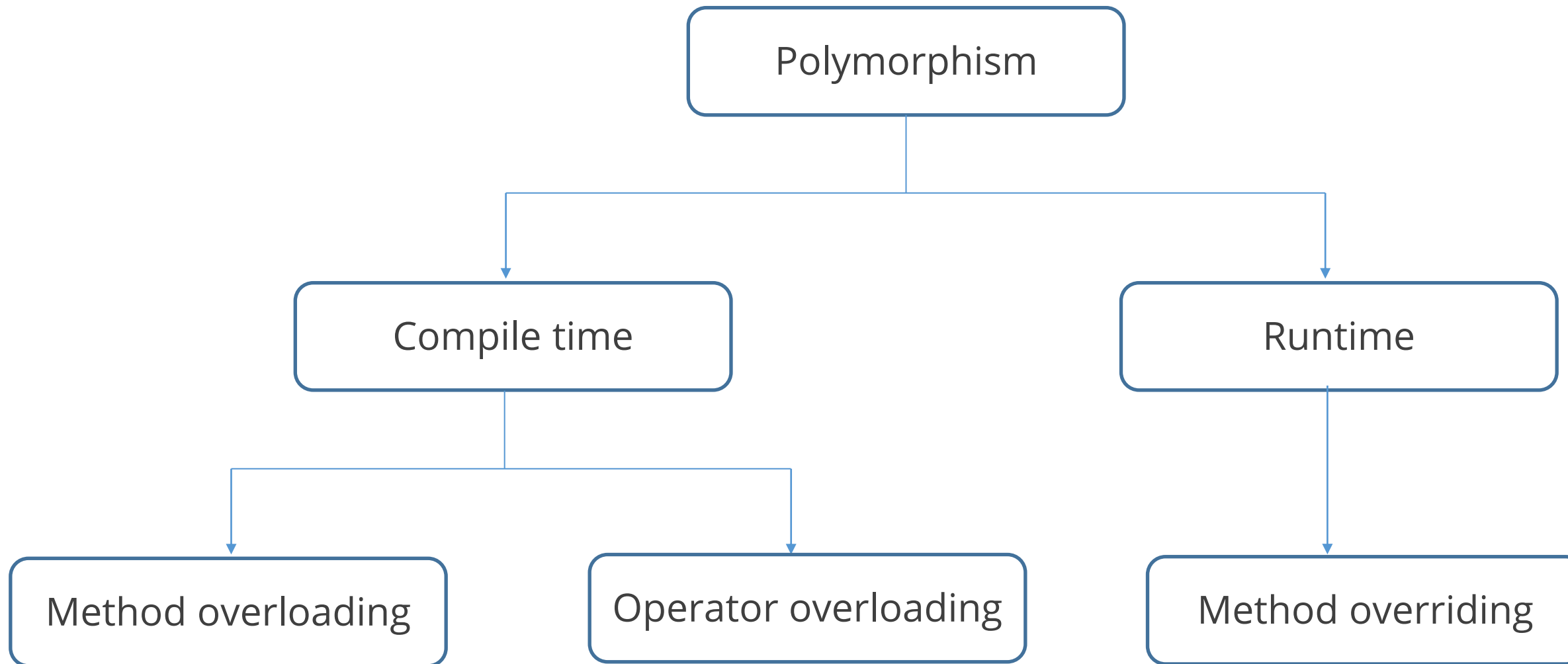
# Polymorphism

# Polymorphism



- Polymorphism comes from a Greek word meaning 'many shapes'.
- Polymorphism is the ability of a message to be displayed in multiple forms.
- Example: A woman can simultaneously be a mother, wife, teacher, sister, and friend.

# Types of Polymorphism



## 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

#### Note

**Note:** Please refer to the Reference Material section to download the **Jupyter Notebook** files for the mentioned topic.



**Abstraction**

# Abstraction

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



For example, when one presses a key on the keyboard, the relevant character appears on the screen. One doesn't have to know how this works. This is called abstraction.

# Abstraction

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

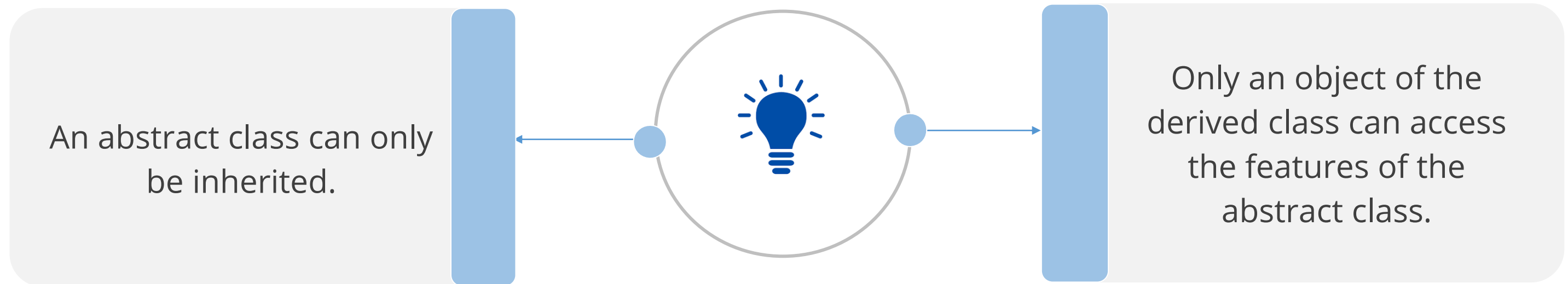
An abstract class is a class specified in the code containing abstract methods.



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

# Abstraction

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





## 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 the abstract class using objects

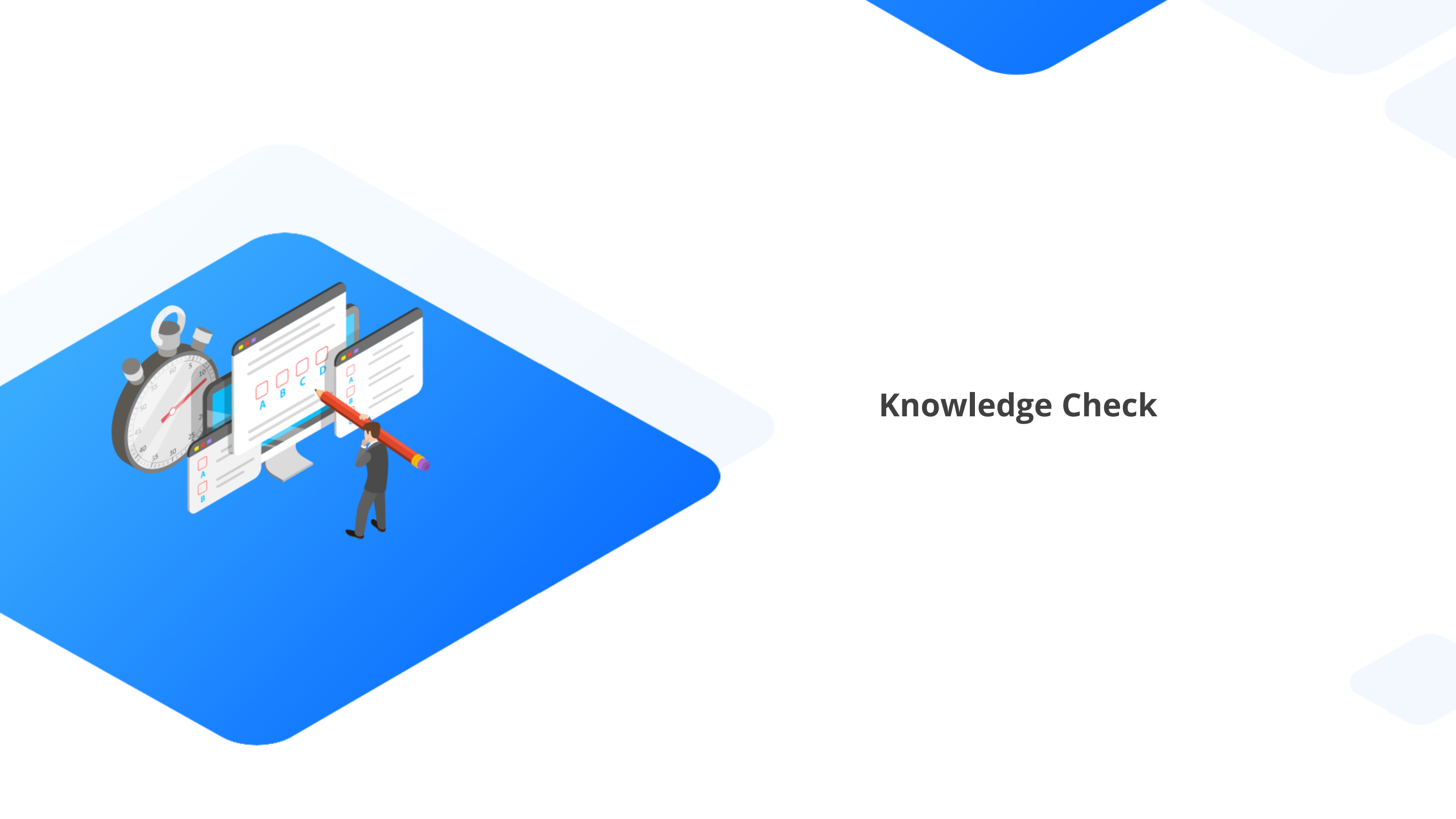
### *Note*

**Note:** Please refer to the **Reference Material** section to download the **Jupyter Notebook** files for the mentioned topic.

# 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 OOP concept?

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



## Knowledge Check 2

Which of the following is NOT an OOP concept?

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



---

The correct answer are **B**

---

**There are four OOP 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 two types of polymorphism are compile-time and runtime polymorphism.**



**Thank You!**