OOPs Concepts with Python

Learning Objectives

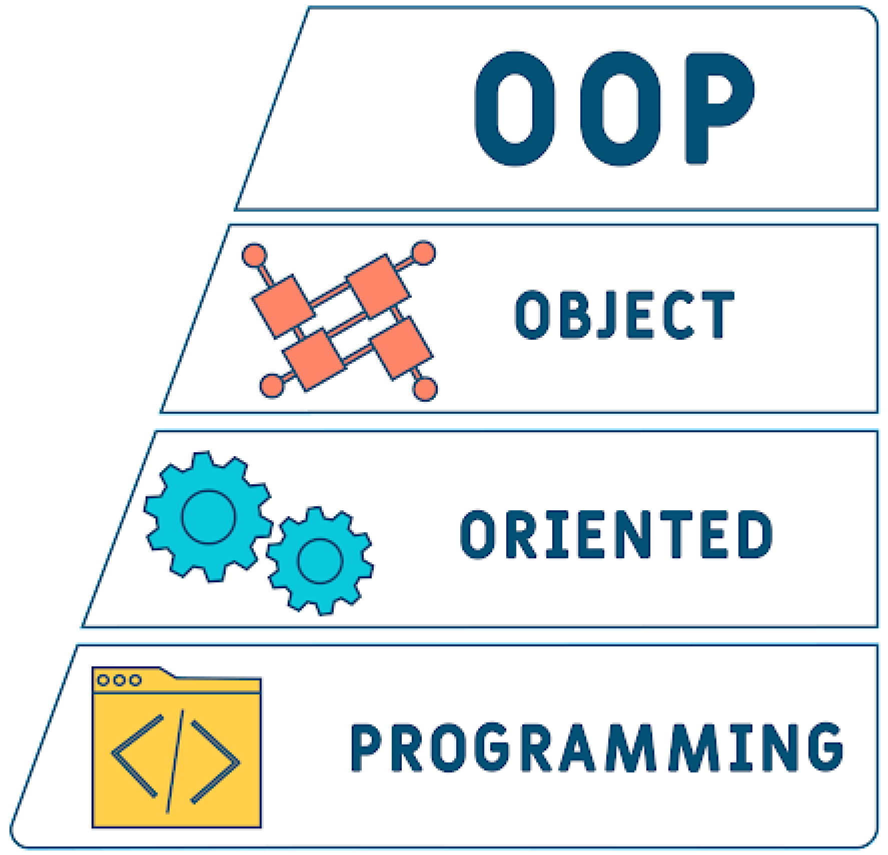
By the end of this lesson, you will be able to:

- Explain OOP and its characteristics
- Identify objects and classes
- Describe methods, attributes, and access modifiers
- Define abstraction, encapsulation, inheritance, and polymorphism with real-life examples

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 Characteristics

- OOP uses a bottom-up approach.
- A program is divided into objects.
- Objects can move freely within member functions.
- OOP is more secure than procedural languages.
- OOP uses access modifiers.

OOP: Concepts

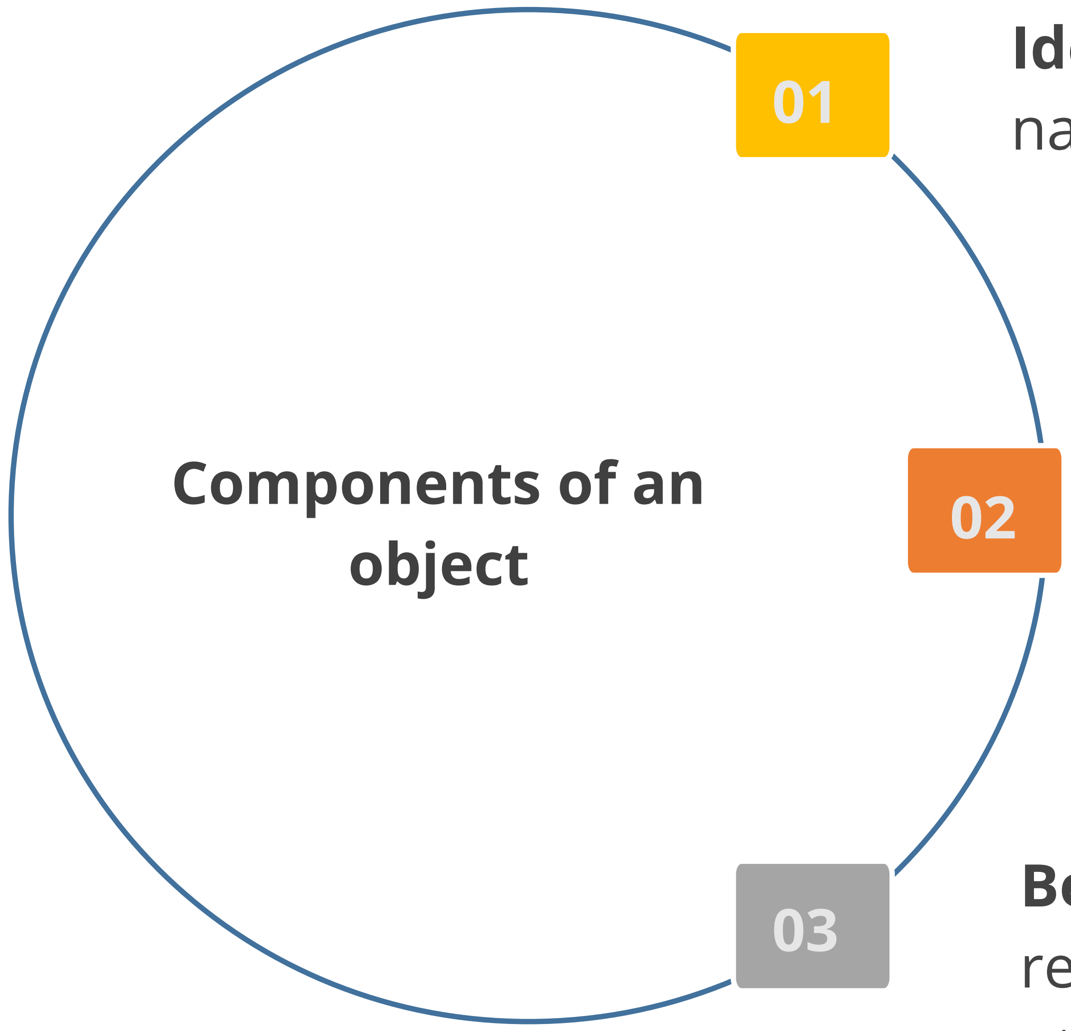
The four concepts of object-oriented programming are:

- Inheritance
- Encapsulation
- Abstraction
- Polymorphism

Objects and Classes

Objects

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



Identity: It gives a unique name to an object.

State: It reflects the properties of an object.

Behavior: It reflects the response of an object to other objects.

Objects: Example

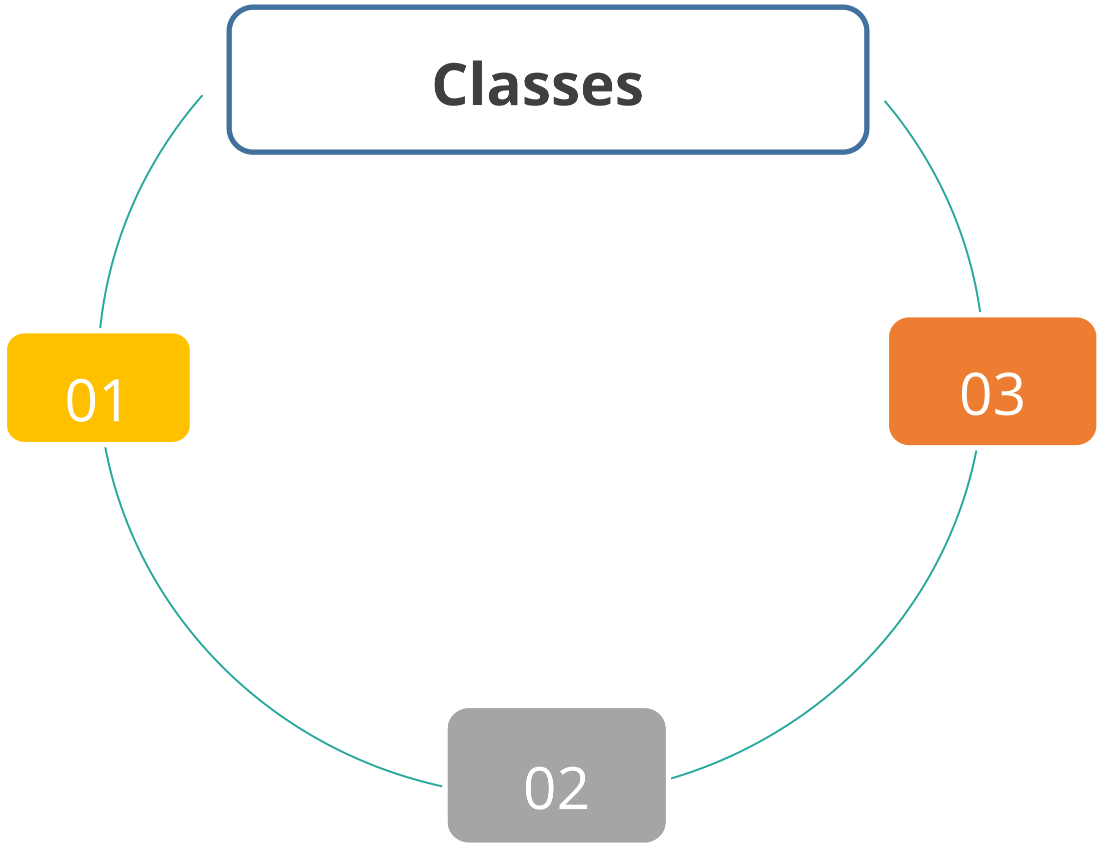
An example of an object is given below:

Object: Dog

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

Classes

A class is a blueprint for an object.



The keyword "**class**" is used to create a class.

The class defines objects that share the same properties and methods.

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.

Classes: Example

The following example illustrates the use of a class with student details.

Example

```
class student :
    """
    A class representing a student.
    """

    def __init__(self, n, a):
        self.name = n
        self.age = a

    def get_age(self):
        return self.age
```

Methods

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

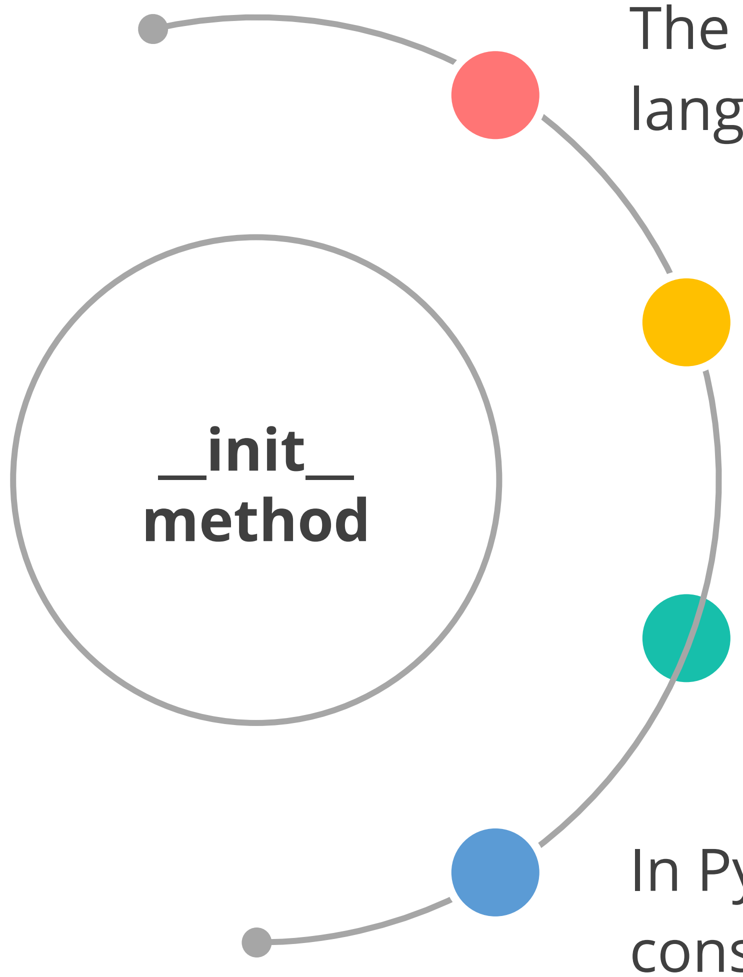
Example

```
# A sample class with init method
class Person:

 # init method or constructor
 def __init__(self, name):
 self.name = name
```

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

- 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



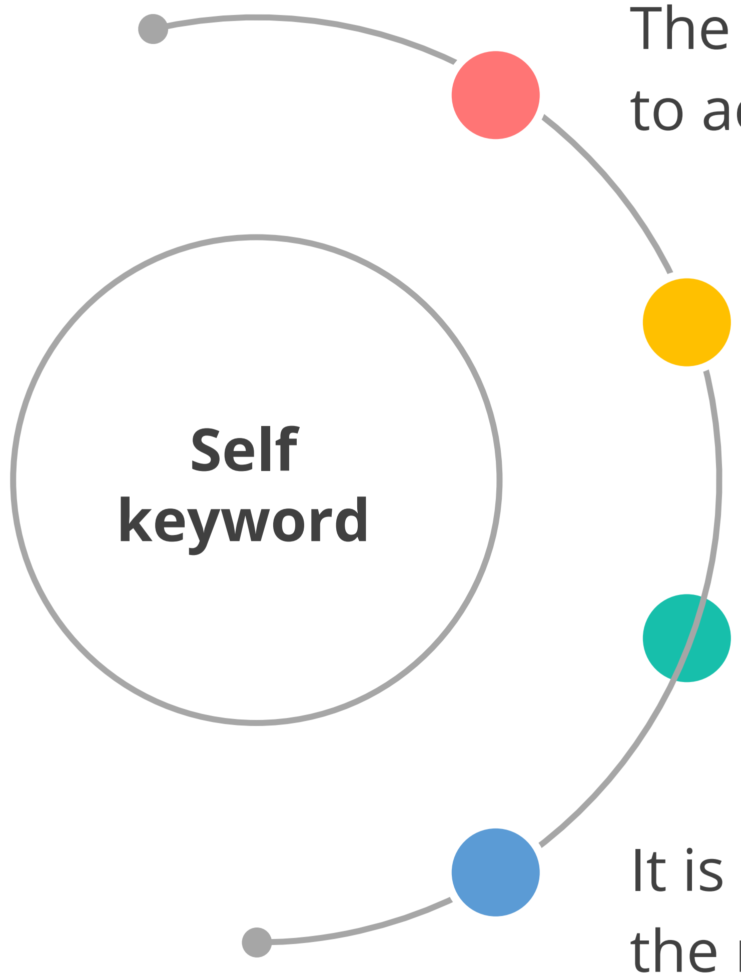
The __init__ method in Python is equivalent to the constructor in other languages.

This method is invoked every time an object of that class is created.

An __init__ method can accept as many parameters as necessary to initialize the object's attributes. It is used only within the class.

In Python, default arguments can be used instead of passing values to constructors.

Self



The self keyword in the class method definitions represents the object. It helps to access the attributes and methods.

It binds the attributes with the given arguments.

Self refers to the object being created, or the instance whose method was called, in the __init__ method.

It is the first parameter provided to methods that refer to the instance on which the method was called.

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.
- The classes do not have any destructor methods.

Attributes

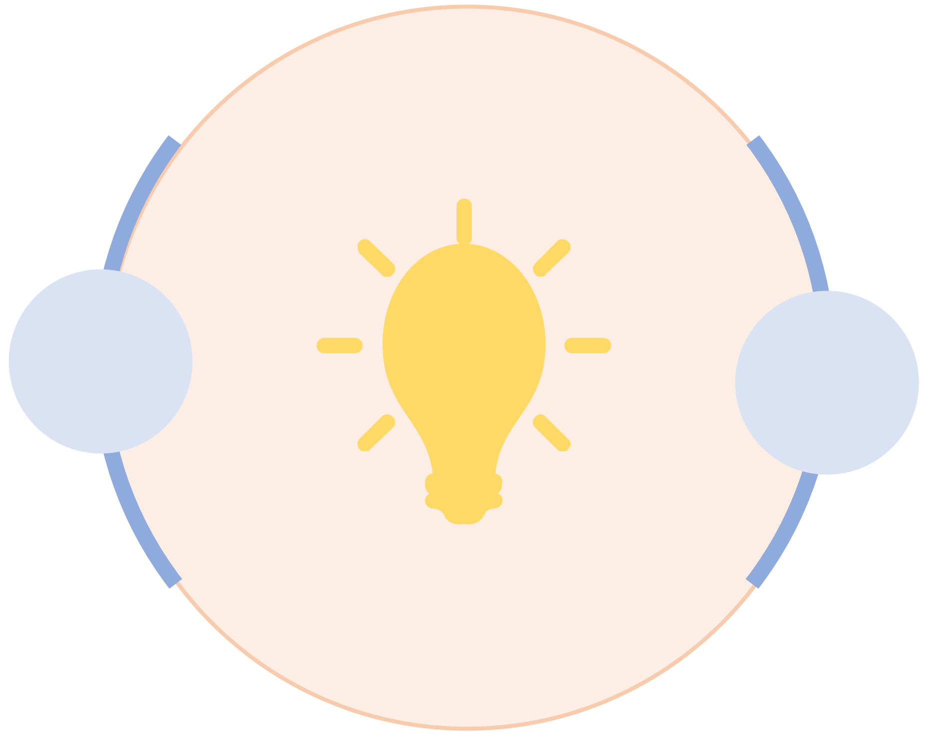
The non-method data stored by the objects are called attributes.

Object: Dog

Identity	Attribute
Name of the dog	Breed
	Age
	Colour

Types of Attributes

A Python object consists of two types of attributes:



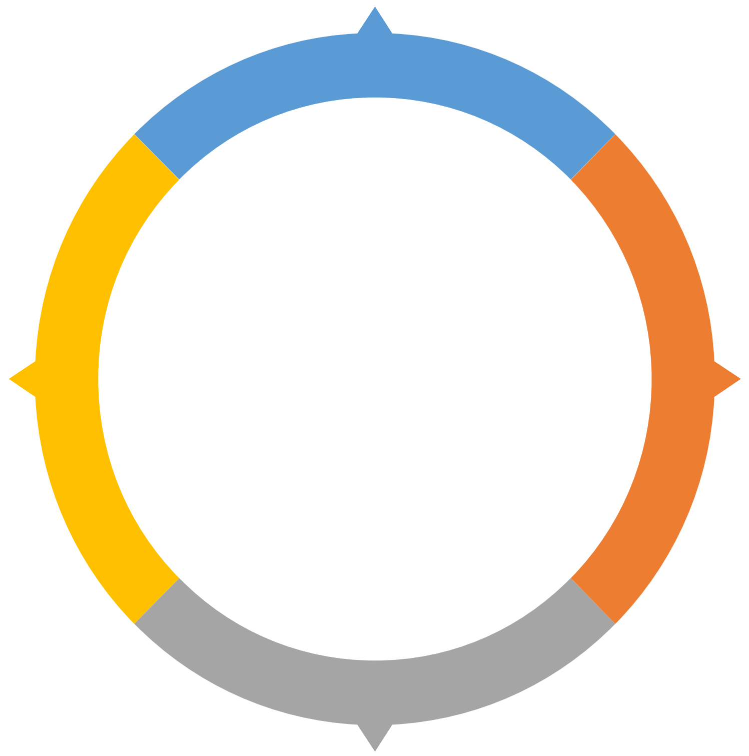
Data attributes

Class attributes

Data Attributes

Following are some characteristics of data attributes:

A particular instance of a class owns the variables.



The __init__() method creates and initializes variables.

Data attributes are referred inside the class using self keyword.

Each instance has its value for it.

Class Attributes

The class is the owner of the class attributes. Following are some characteristics of class attributes:

It is a variable shared by all instances of a class.

These are also known as static variables in some languages.

All instances of a class share the same value for it.

It is used to build class-wise constants and object counters.

Assisted Practice: Create a Class with Attributes And Methods

Duration: 10 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.

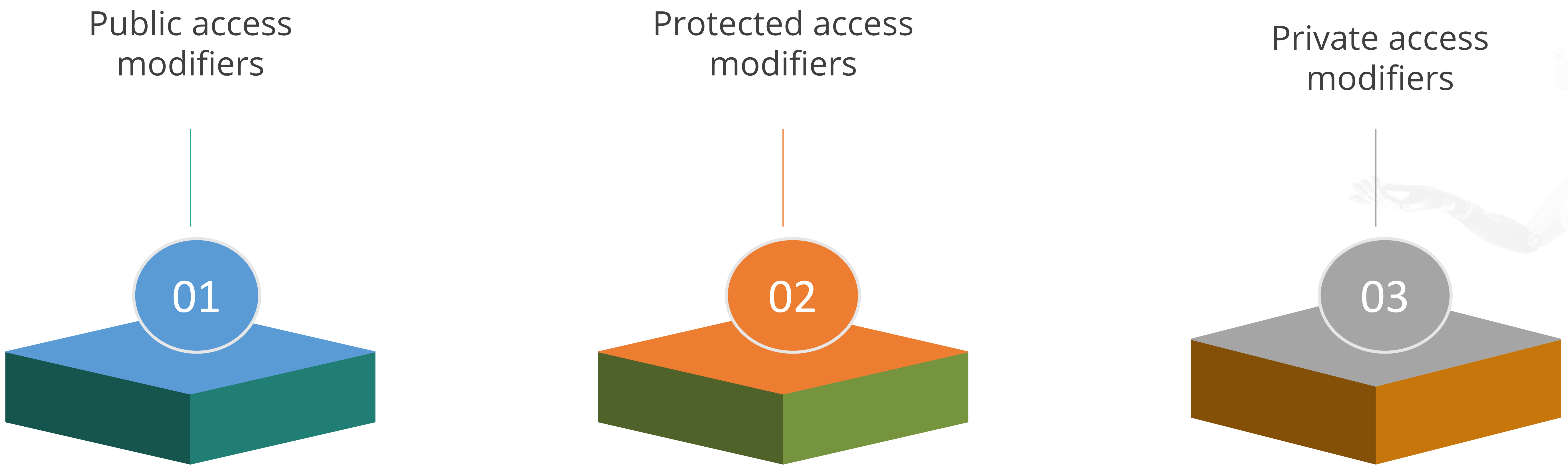
Tasks to Perform:

1. Create a class named person
2. Declare the desired attributes like the age of the person
3. Create a method that displays the age
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.



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.

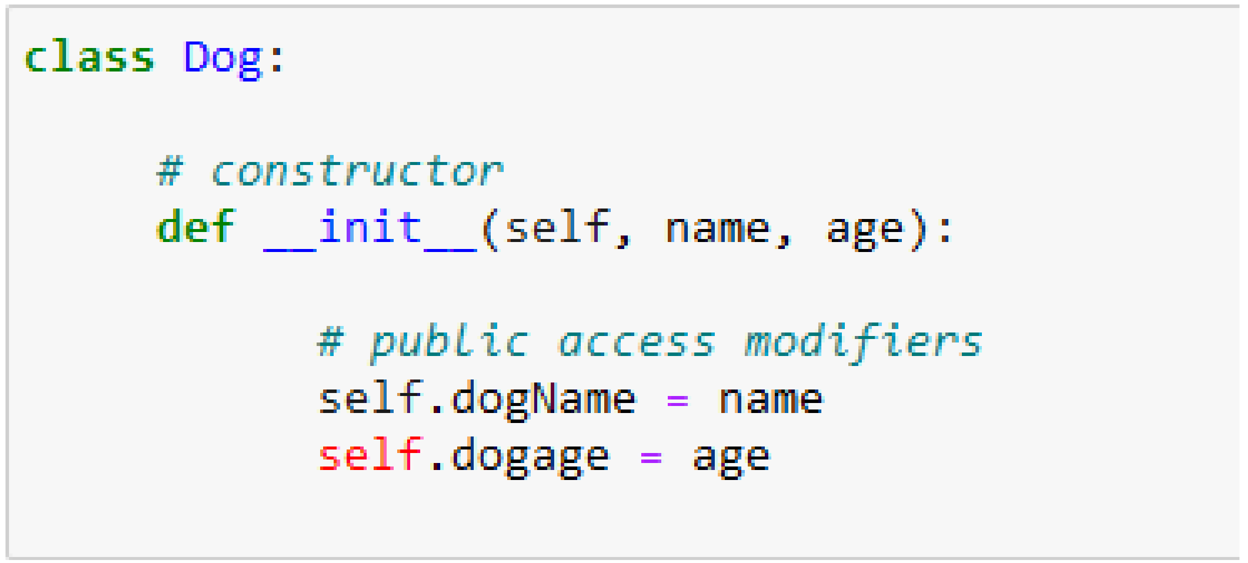
02

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.

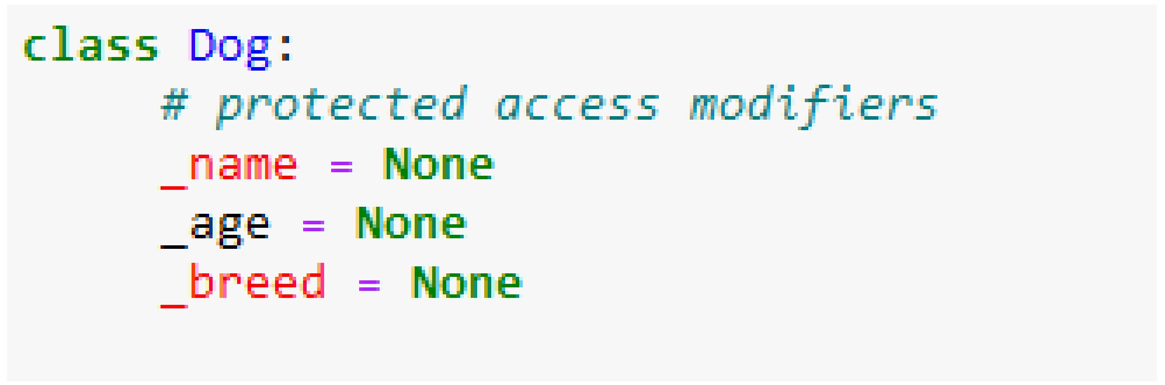
02

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.

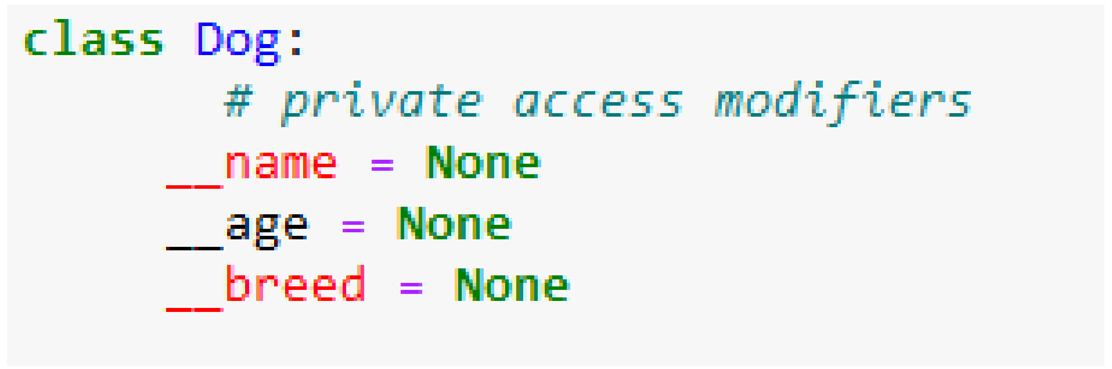
02

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



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

Encapsulation

Encapsulation

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

Features

It hides the state of a structured data object inside a class, preventing unauthorized access to an unauthorized person.

It acts as a protective barrier that restricts direct access to variables and methods. It is also known as data hiding.

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
```

Assisted Practice: Encapsulation

Duration: 10 mins

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

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

Assisted Practice: Encapsulation

Duration: 10 mins

Tasks to perform:

1. Create a parent class with protected members
2. Create a child class that extracts the value of the protected members in the parent class
3. Modify the protected member in the derived class
4. Create the objects of the parent and the child class
5. Print the protected member using the objects

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.

Multilevel inheritance:

A derived class is created from another derived class.

Multiple inheritance:

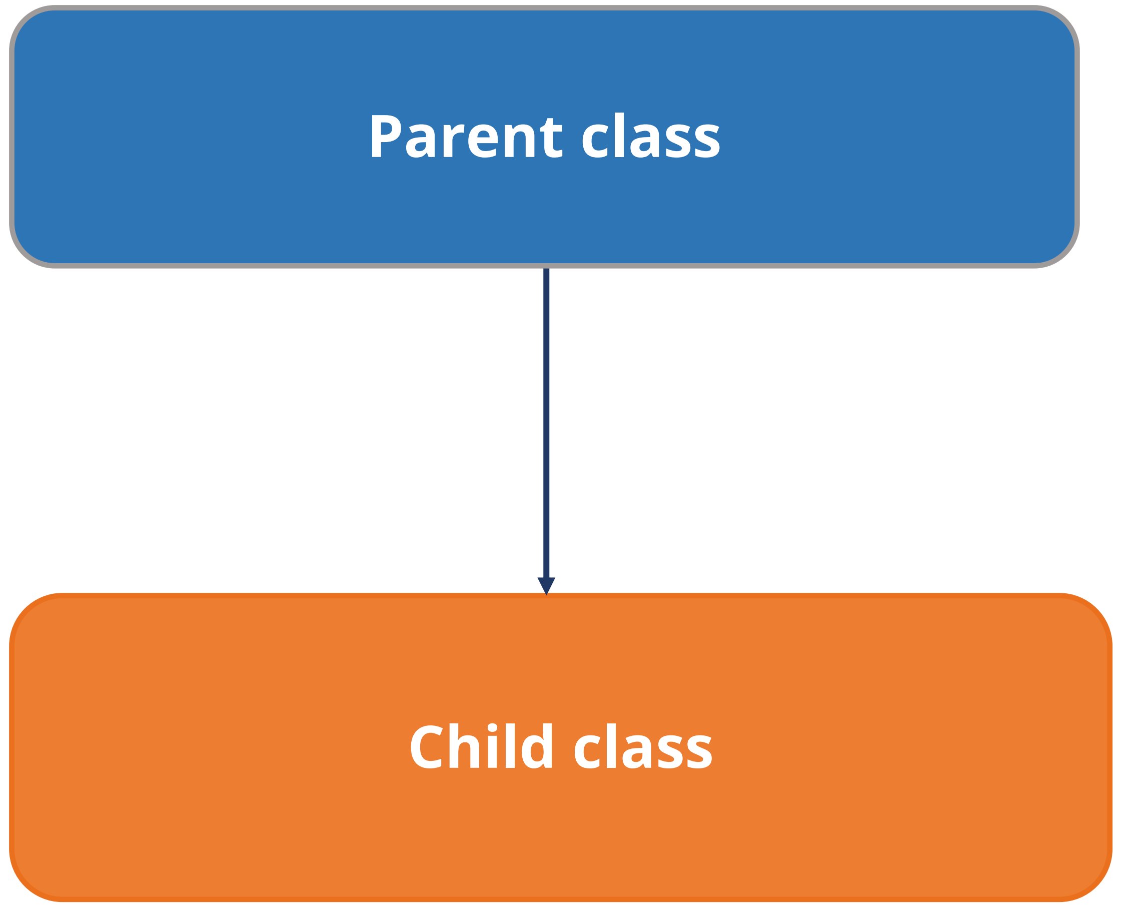
A class can inherit from more than one class.

Hierarchical inheritance:

A base class can have multiple subclasses inherited from it.

Inheritance: Single Level Inheritance

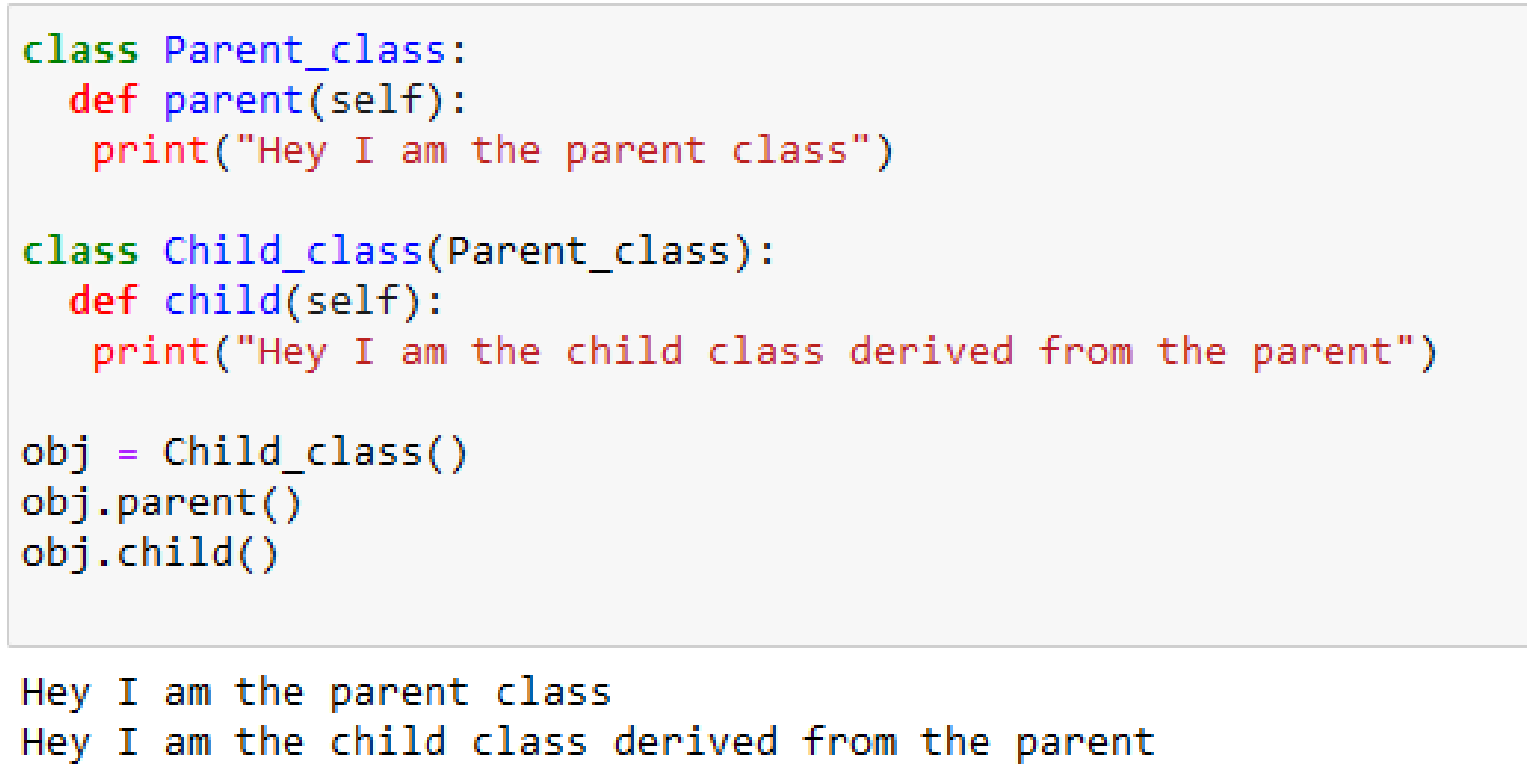
A class that is derived from one parent class is called single level inheritance.



Single Level Inheritance: Example

The following is an example of single level inheritance:

Example

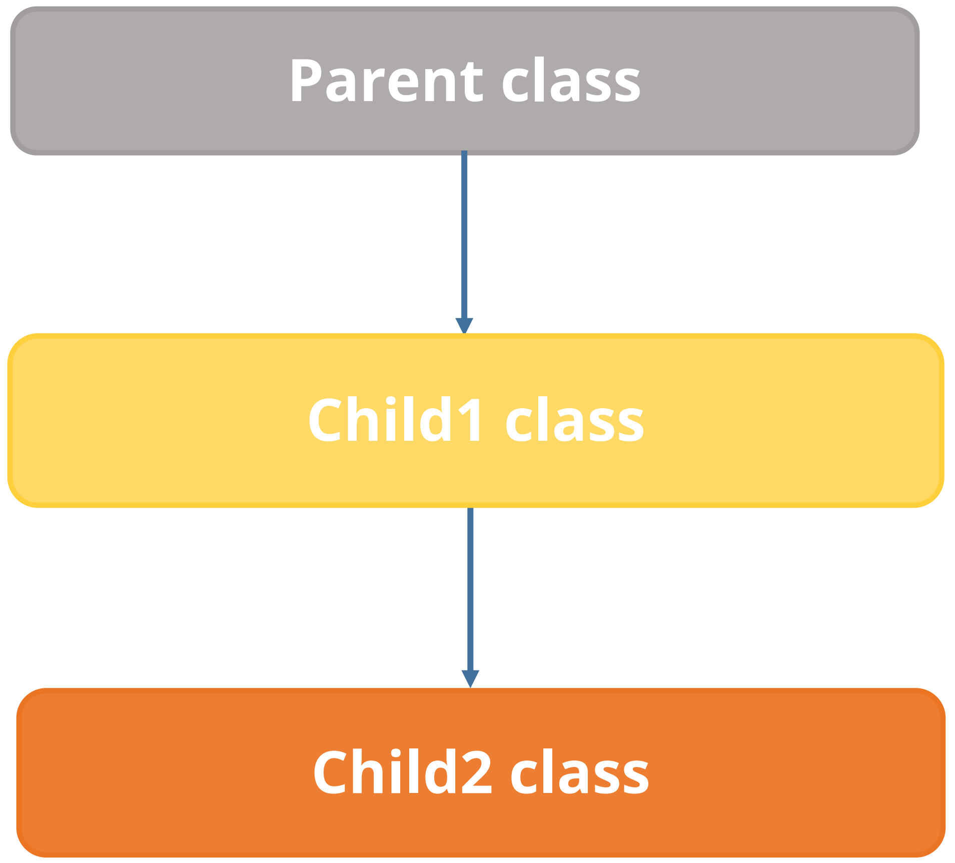


```
class Parent_class:
  def parent(self):
   print("Hey I am the parent class")

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

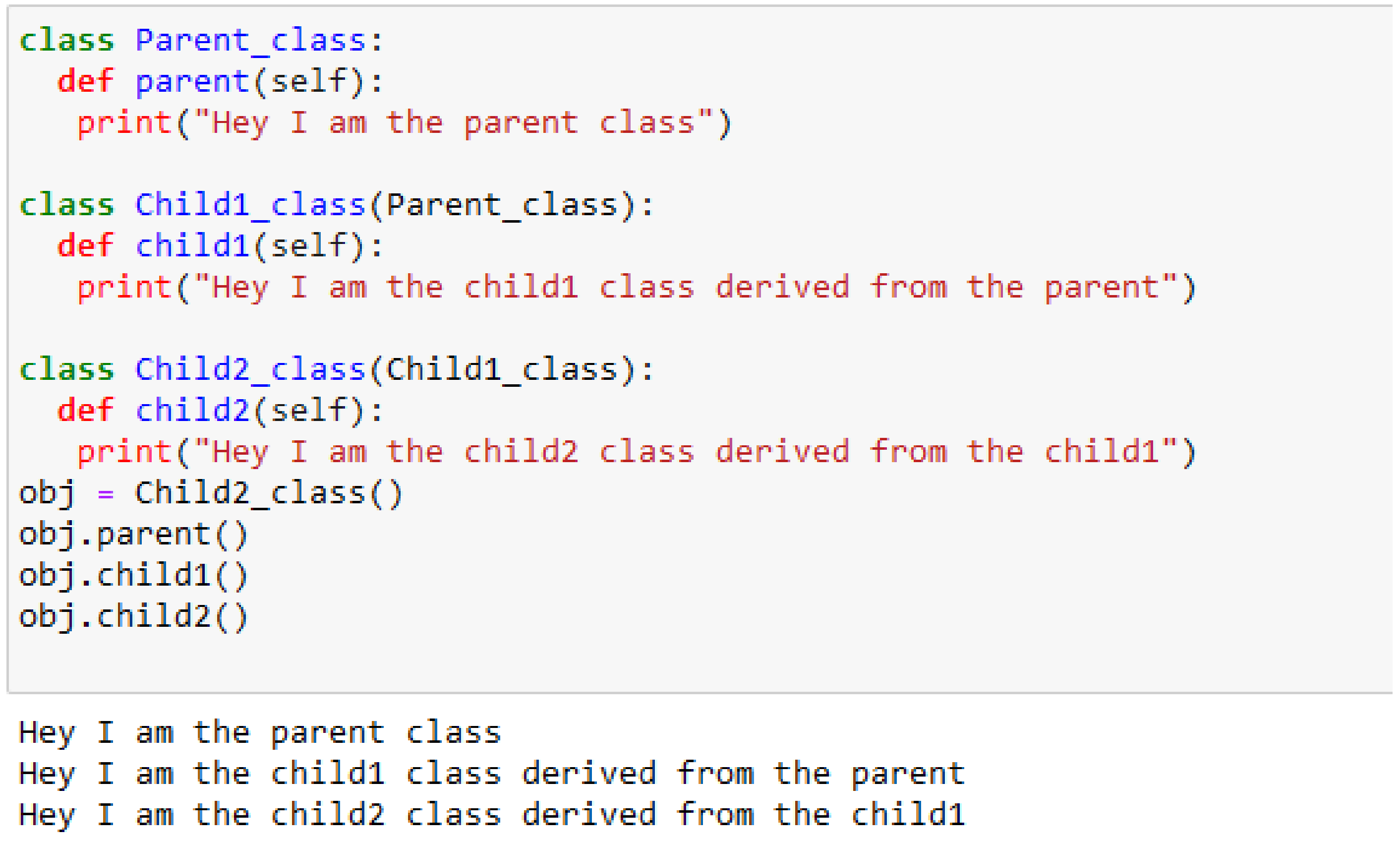
In multilevel inheritance, the features of the parent class and the child class are further inherited into the new child class.



Multilevel Inheritance: Example

An example of multilevel inheritance is shown below:

Example

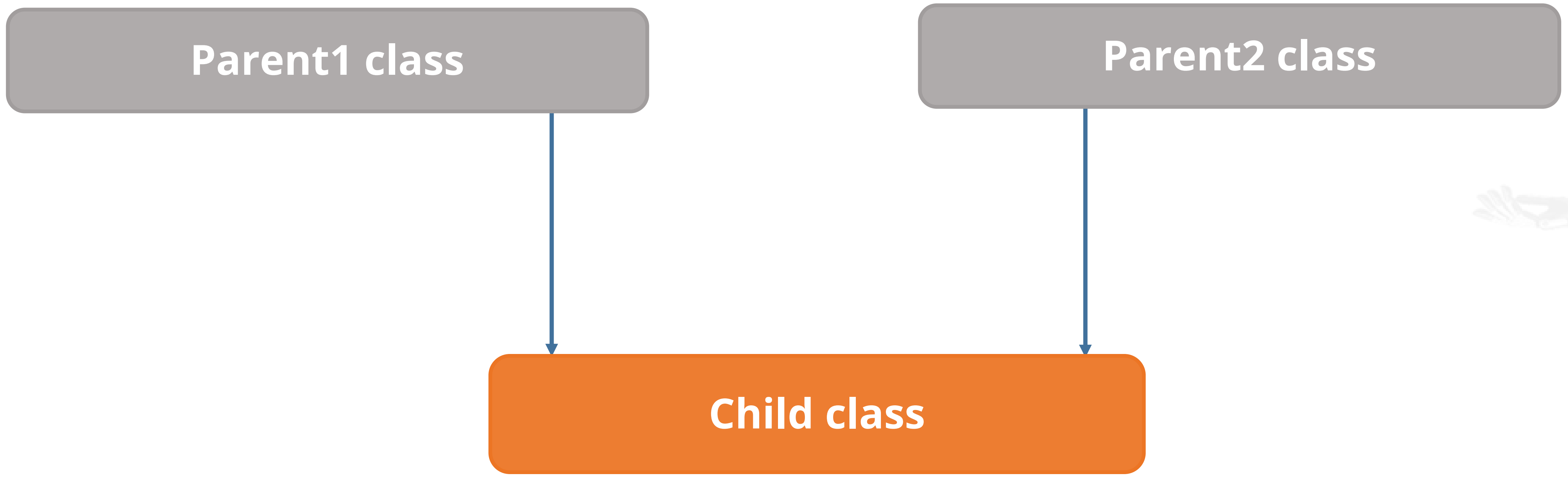


```python
class Parent_class:
  def parent(self):
   print("Hey I am the parent class")

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

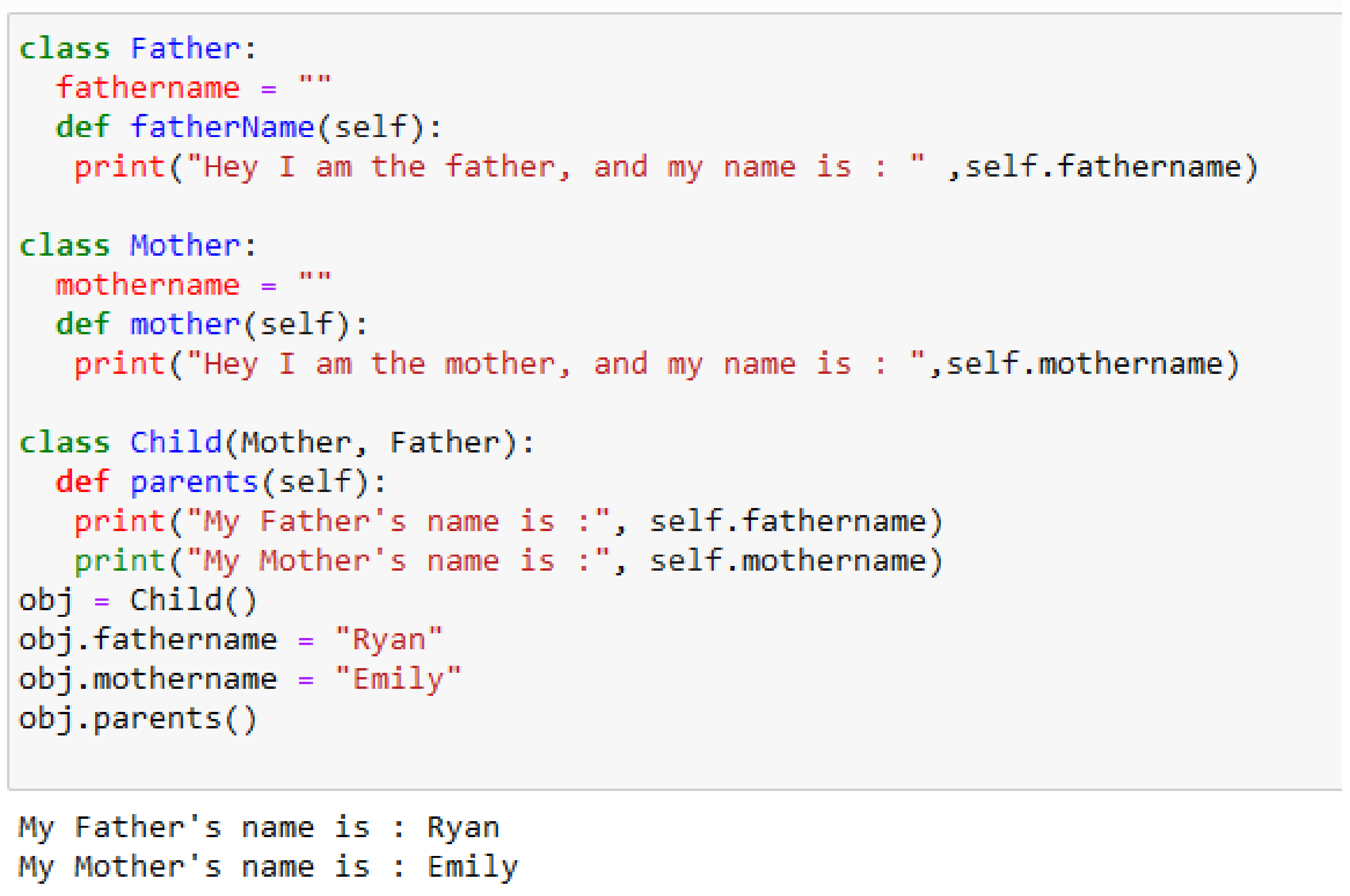
A class that is derived from more than one parent class is called multiple inheritances.



Multiple Inheritance: Example

An example of multilevel inheritance is given below:

Example

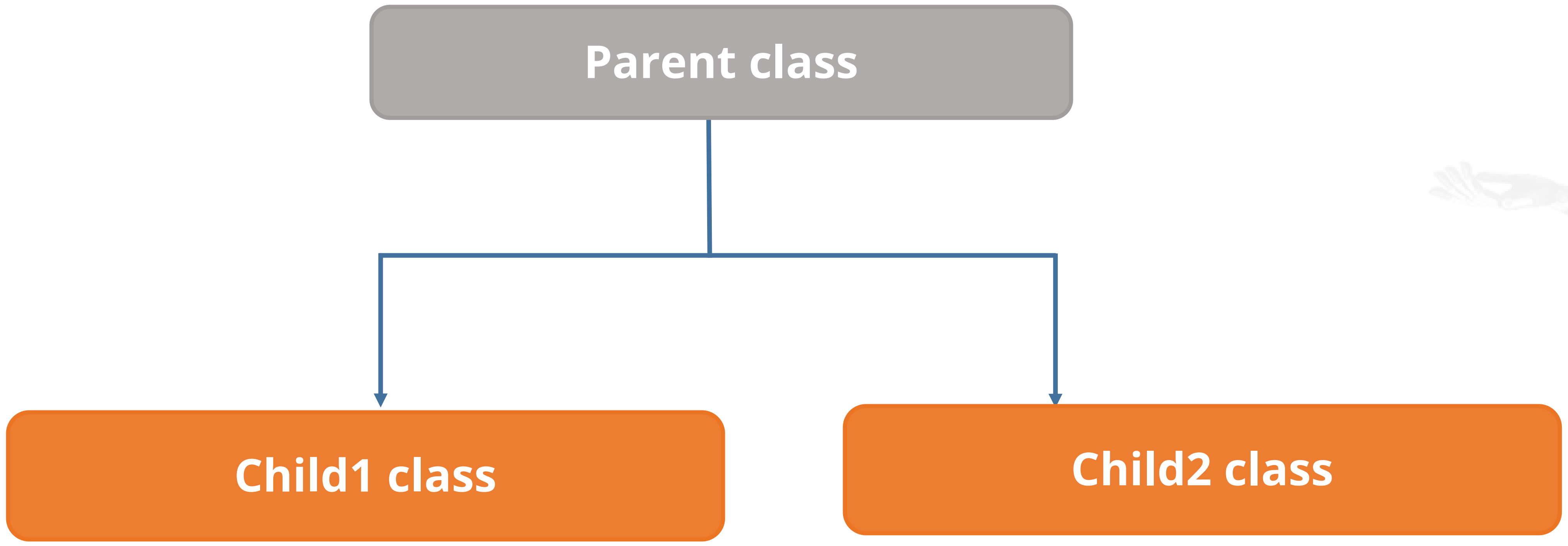


```
class Father:
  fathername = ""
  def fatherName(self):
   print("Hey I am the father, and my name is : " ,self.fathername)

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

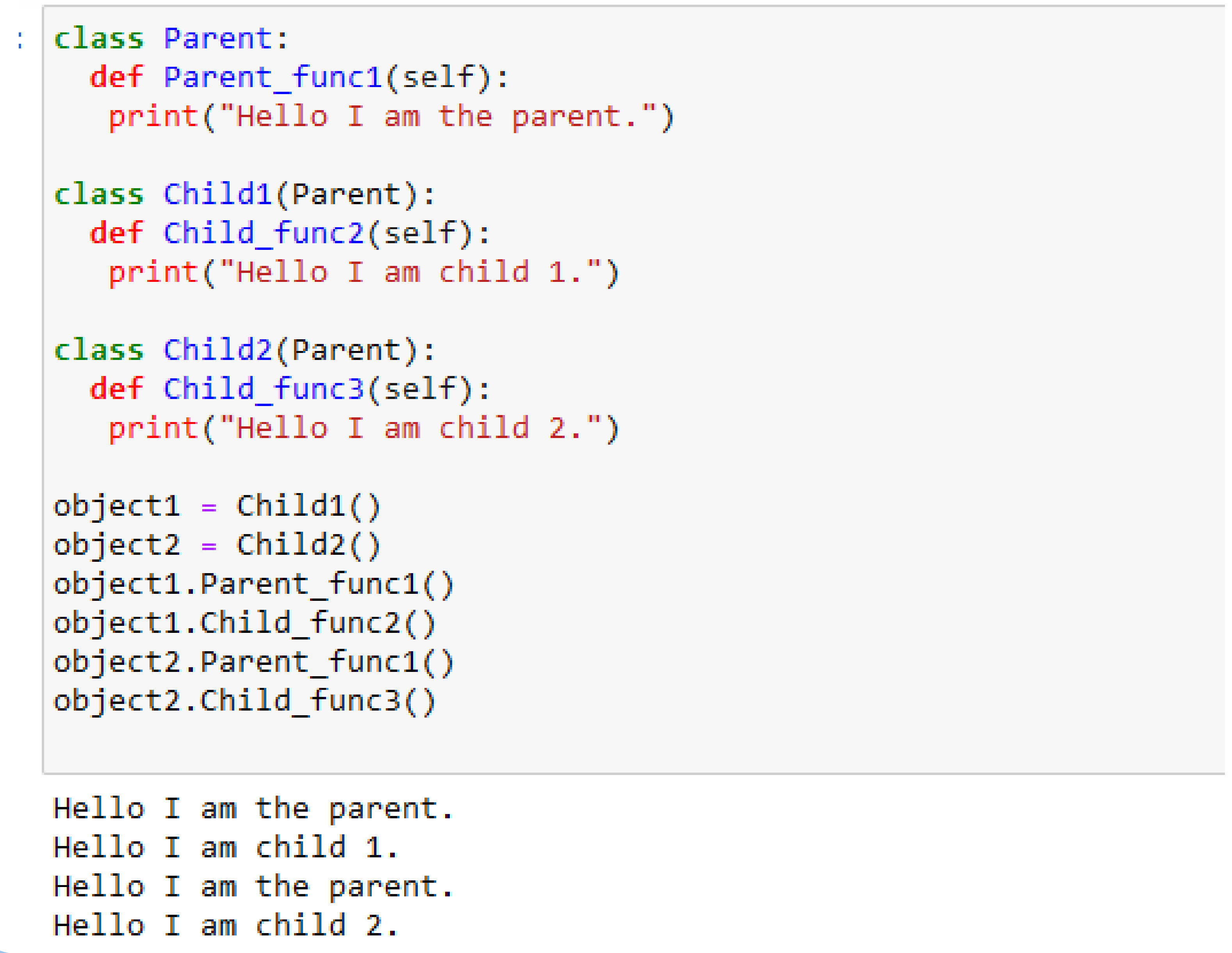
Hierarchical inheritance is the process of creating multiple derived classes from a single base class.



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: 10 mins

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

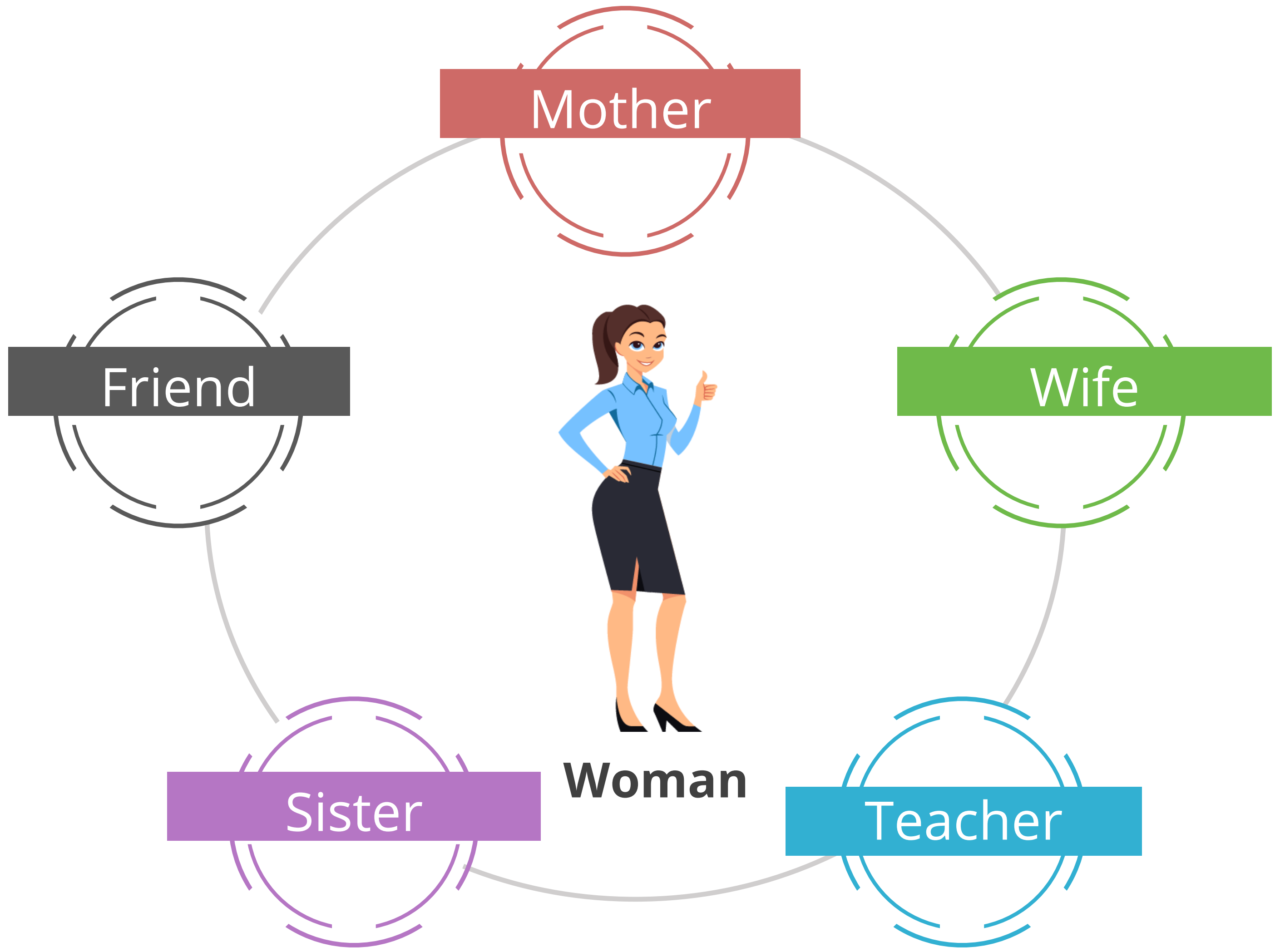
Objective: In this demonstration, we will learn how to work with inheritance.

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

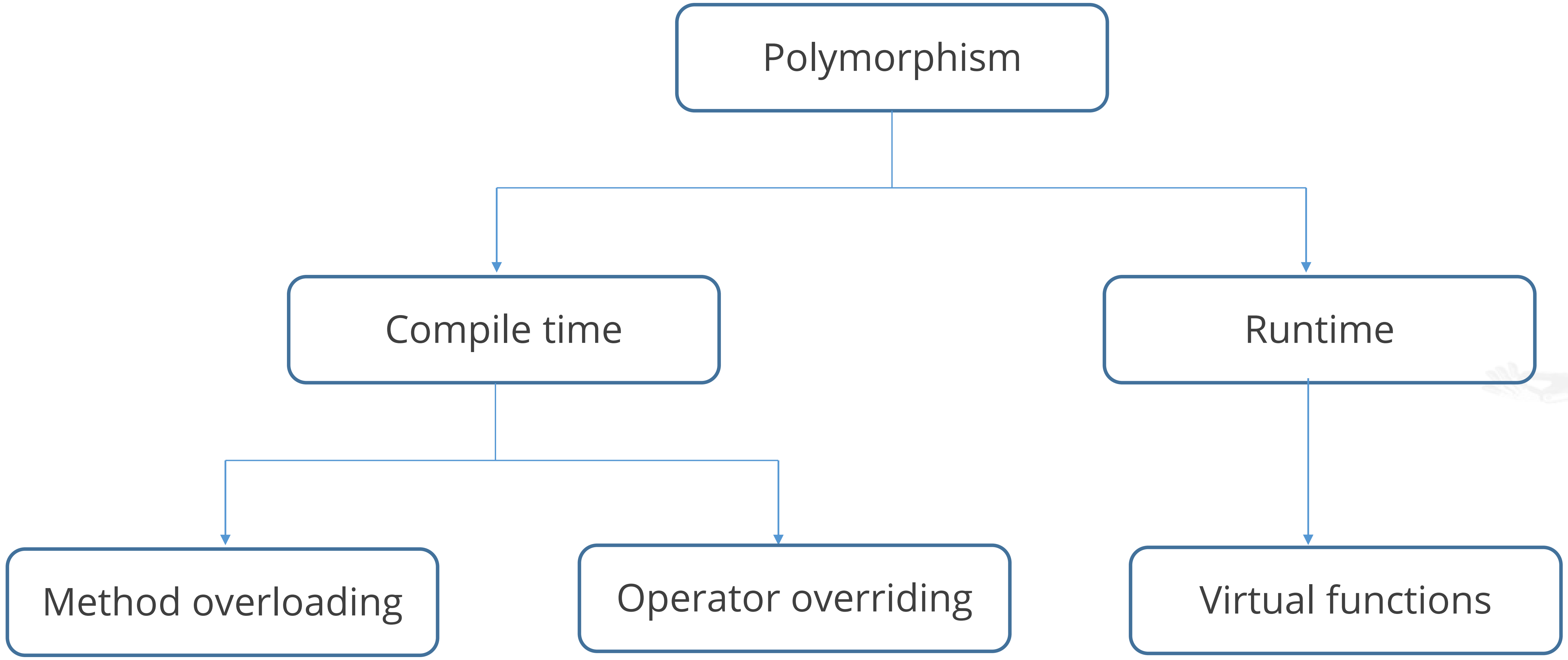
Polymorphism



- 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 daughter, a teacher, and an employee at the same time.

Types of Polymorphism

The types of polymorphism are mentioned below:



Method Overloading

Method overloading is a mechanism where two or more different classes can have the same method name but different sets of parameters.

Example

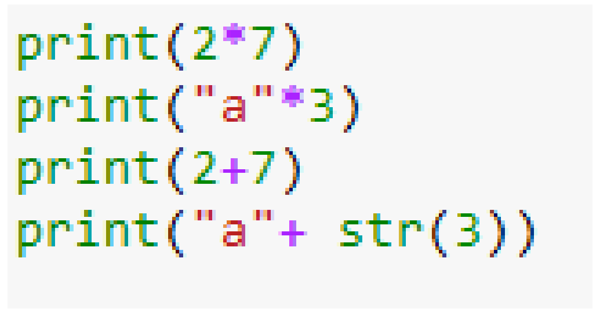
```
class Woman1():
  def Mother(self):
   print("Woman 1 is a mother.")
  def Friend(self):
   print("Woman 1 has 3 friends.")
  def Employee(self):
   print("Woman 1 is a teacher.")

class Woman2():
  def Mother(self):
   print("Woman 2 is a mother.")
  def Friend(self):
   print("Woman 2 has 5 friends.")
  def Employee(self):
   print("Woman 2 is a dancer.")
obj_woman1 = Woman1()
obj_woman2 = Woman2()

for i in (obj_woman1 , obj_woman2):
  i.Mother()
  i.Friend()
  i.Employee()
```

```
Woman 1 is a mother.
Woman 1 has 3 friends.
Woman 1 is a teacher.
Woman 2 is a mother.
Woman 2 has 5 friends.
Woman 2 is a dancer.
```

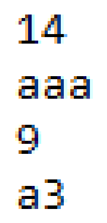
Operator Overloading

Operator overloading is the type of overloading in which an operator can be used in multiple ways.

Example



```
print(2*7)
print("a"*3)
print(2+7)
print("a"+ str(3))
```



```
14
aaa
9
a3
```

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 with the names employee1 and employee2 that contain the same method names
2. Create the objects of the base class and call the methods

Abstraction

Abstraction

Abstraction is the property by which only the essential details are displayed to the user.

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

The following steps can help users access the objects of an abstract class:

An abstract class can only be inherited.

Then an object of the derived class is used to access the features of the abstract class.

Abstract Classes in Python: Example

The following example illustrates the use of an abstract class:

Example

```
# Parent class
from abc import ABCMeta,abstractmethod

class beverage():
  __metaclass__ = ABCMeta

  @abstractmethod
  def ingredients(self):
   pass

  def taste(self):
   pass
```

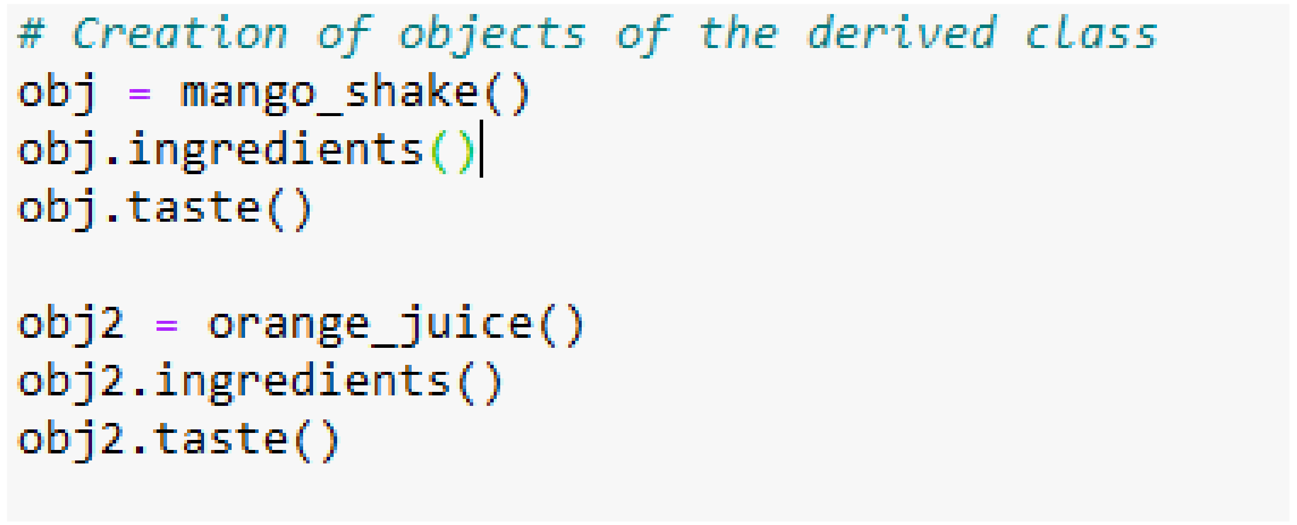
```
# Derived class
class mango_shake(beverage):
  def ingredients(self):
   print("mango" , "milk", "sugar")
  def taste(self):
   print("Yummy!!")
```

```
# Derived class
class orange_juice(beverage):
  def ingredients(self):
   print("orange" , "water", "sugar")
  def taste(self):
   print("Sweet!!")
```

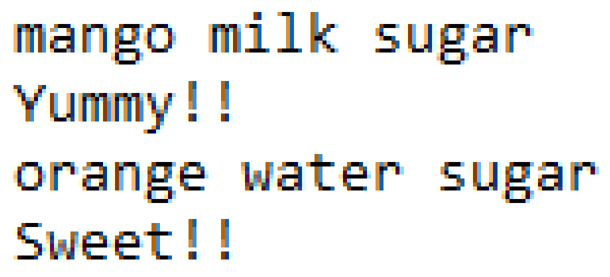
Abstract Classes in Python: Example

The following example illustrates the use of an abstract class:

Example



```
# Creation of objects of the derived class
obj = mango_shake()
obj.ingredients()
obj.taste()

obj2 = orange_juice()
obj2.ingredients()
obj2.taste()
```



```
mango milk sugar
Yummy!!
orange water sugar
Sweet!!
```

Abstract Classes in Python: Example

The following example illustrates the use of an abstract class:

Example

```
#Creation of objects of the abstract class
abstract_obj = beverage()
abstract_obj.ingredients()
abstract_obj.taste()
```

```
#Extracting of abstract class with an object results in the below error

abstract_obj=beverage()
```

```
TypeError: Can't instantiate abstract class beverage with abstract methods ingredients
```

Assisted Practice: Abstraction

Duration: 10 mins

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

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

Tasks to perform:

1. Import the necessary libraries for creating the abstract class
2. Create a base class with the name Food that contains abstract methods
3. Create two derived classes with methods from the base class that contains non-abstract methods
4. Extract the methods of the derived 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.