UNIT III

Classes and Objects – Constructors and Destructors – Instance and Class Variables –

Instance, Class, and Static Methods – Data Hiding – Inheritance: Single, Multiple, Multilevel,

Hierarchical – Method Overriding – Polymorphism and Operator Overloading – Abstract Classes

and Interfaces (ABC module) – Composition vs Inheritance – Dunder (Magic) Methods.

Python Classes and Objects

- Python's object-oriented programming (OOP) allows you to model real-world entities in code, making programs more organized, reusable and easier to maintain. By grouping related data and behavior into a single unit, classes and objects help you write cleaner, more logical code for everything from small scripts to large applications.

Class
A class in Python is a user-defined template for creating objects. It bundles data and functions together, making it easier to manage and use them. When we create a new class, we define a new type of object. We can then create multiple instances of this object type.

Classes and Objects (Here Dog is the Base Class and Bobby is Object)

Creating Class

Classes are created using **class keyword**. Attributes are variables defined inside class and represent properties of the class. Attributes can be accessed using dot **.** **operator** (e.g., MyClass.my_attribute).

```
# define a class
class Dog:
    sound = "bark"  # class attribute
```

Object
An **object** is a specific instance of a class. It holds its own set of data (instance variables) and can invoke methods defined by its class. Multiple objects can be created from same class, each with its own unique attributes.
Let's create an object from **Dog class**

```
class Dog:
    sound = "bark"

dog1 = Dog() # Creating object from class
print(dog1.sound) # Accessing the class
```

Output

bark

Explanation: sound attribute is a class attribute. It is shared across all instances of Dog class, so can be directly accessed through instance **dog1**.

Initiate Object with __init__()

The __init__() method acts as a constructor in Python and is automatically executed when an object is created. It is used to initialize the attributes of the object with the values provided at the time of object Creation

```
class Dog:
    species = "Canine"  # Class attribute

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

# Creating an object of the Dog class
dog1 = Dog("Buddy", 3)

print(dog1.name)
print(dog1.species)
```

Output

```
Buddy

Canine
```

Explanation:

- self refers to the current object and is used to store data inside it.
- Dog("Buddy", 3) creates an object and passes values to __init__().
- self.name and self.age store the object's name and age.
- dog1.name accesses the object's instance attribute and dog1.species accesses the shared class attribute.

__str__() Method

__str__ method in Python allows us to define a custom string representation of an object. By default, when we print an object or convert it to a string using str(), Python uses the default implementation, which returns a string like <__main__.ClassName object at 0x00000123>.

Let's look at an example of using **__str__()** method to provide a readable string output for an object:

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

```
    def __str__(self):
        return f"{self.name} is {self.age} years old."
dog1 = Dog("Buddy", 3)
dog2 = Dog("Charlie", 5)

print(dog1)
print(dog2)
```

Explanation:

- **__str__ Implementation:** Defined as a method in Dog class. Uses self parameter to access instance's attributes (name and age).
- **Readable Output:** When print(dog1) is called, Python automatically uses __str__ method to get a string representation of object. Without __str__, calling print(dog1) would produce something like <__main__.Dog object at 0x00000123>.

Class and Instance Variables in Python

In Python, variables defined in a class can be either class variables or instance variables and understanding distinction between them is crucial for object-oriented programming.

Class Variables

These are variables that are shared across all instances of a class. It is defined at class level, outside any methods. All objects of class share same value for a class variable unless explicitly overridden in an object.

Instance Variables

Variables that are unique to each instance (object) of a class. These are defined within **__init__()** method or other instance methods. Each object maintains its own copy of instance variables, independent of other objects.
Below code shows how class variables are shared across all objects, while instance variables are unique to each object**.**

```
class Dog:
    # Class variable
    species = "Canine"

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

# Create objects
dog1 = Dog("Buddy", 3)
dog2 = Dog("Charlie", 5)
```

```
# Access class and instance variables
print(dog1.species)  # (Class variable)
print(dog1.name)     # (Instance variable)
print(dog2.name)     # (Instance variable)

# Modify instance variables
dog1.name = "Max"
print(dog1.name)     # (Updated instance variable)

# Modify class variable
Dog.species = "Feline"
print(dog1.species)  # (Updated class variable)
print(dog2.species)
```

Explanation:

- **Class Variable (species):** Shared by all instances of class. Changing Dog.species affects all objects, as it's a property of class itself.
- **Instance Variables (name, age):** Defined in __init__() method. Unique to each instance (e.g., dog1.name and dog2.name are different).
- **Accessing Variables:** Class variables can be accessed via class name (Dog.species) or an object (dog1.species). Instance variables are accessed via object (dog1.name).
- **Updating Variables:** Changing Dog.species affects all instances. Changing dog1.name only affects dog1 and does not impact dog2.

Constructors in Python

- In Python, a constructor is a special method that is called automatically when an object is created from a class. Its main role is to initialize the object by setting up its attributes or state.

  The method **__new__** is the **constructor** that creates a new instance of the class while **__init__** is the initializer that sets up the instance's attributes after creation. These methods work together to manage object creation and initialization.

  __new__ Method

  This method is responsible for **creating a new instance of a class**. It allocates memory and **returns the new object**. It is called before **__init__**.

```
class ClassName:
    def __new__(cls, parameters):
        instance = super(ClassName, cls).__new__(cls)
        return instance
```

  To learn more, please refer to "__new__" method

  __init__ Method

This method initializes the newly created instance and is commonly used as a constructor in Python. It is called immediately after the object is created by **__new__** method and is responsible for initializing attributes of the instance.

Syntax:

```
class ClassName:
    def __init__(self, parameters):
        self.attribute = value
```

Note: It is called after __new__ and does not return anything (it returns **None** by default). To learn more, please refer to "__init__" method

Differences Between __init__ and __new__

__new__ method:

- Responsible for creating a new instance of the class.
- Rarely overridden but useful for customizing object creation and especially in singleton or immutable objects.

__init__ method:

- Called immediately after __new__.
- Used to initialize the created object.

Types of Constructors

Constructors can be of two types.

1. Default Constructor

A **default constructor** does not take any parameters other than **self**. It initializes the object with default attribute values.

```
class Car:
    def __init__(self):

        #Initialize the Car with default attributes
        self.make = "Toyota"
        self.model = "Corolla"
        self.year = 2020

# Creating an instance using the default constructor
car = Car()
print(car.make)
print(car.model)
print(car.year)
```

Output

```
Toyota
Corolla
2020
```

2. Parameterized Constructor

A **parameterized constructor** accepts arguments to initialize the object's attributes with specific values.

```
class Car:
    def __init__(self, make, model, year):

        #Initialize the Car with specific attributes.
        self.make = make
        self.model = model
        self.year = year

# Creating an instance using the parameterized constructor
car = Car("Honda", "Civic", 2022)
print(car.make)
print(car.model)
print(car.year)
```

Output

```
Honda
Civic
2022
```

Destructors in Python

- Constructors in Python

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much as in C++ because Python has a garbage collector that handles memory management automatically.

The **__del__()** method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

Syntax of destructor declaration :

```
def __del__(self):
  # body of destructor
```

Note : A reference to objects is also deleted when the object goes out of reference or when the program ends.

Example 1 : Here is the simple example of destructor. By using del keyword we deleted the all references of object 'obj', therefore destructor invoked automatically.

```
# Python program to illustrate destructor
class Employee:

    # Initializing
    def __init__(self):
        print('Employee created.')

    # Deleting (Calling destructor)
    def __del__(self):
        print('Destructor called, Employee deleted.')

obj = Employee()
del obj
```

Output

```
Employee created.

Destructor called, Employee deleted.
```

Note : The destructor was called **after the program ended** or when all the references to object are deleted i.e when the reference count becomes zero, not when object went out of scope.

Example 2: This example gives the explanation of above-mentioned note. Here, notice that the destructor is called after the 'Program End...' printed.

```
# Python program to illustrate destructor

class Employee:

    # Initializing
    def __init__(self):
        print('Employee created')

    # Calling destructor
    def __del__(self):
        print("Destructor called")

def Create_obj():
    print('Making Object...')
    obj = Employee()
    print('function end...')
    return obj

print('Calling Create_obj() function...')
obj = Create_obj()
```

```
print('Program End...')
```

Output

```
Calling Create_obj() function...

Making Object...

Employee created

function end...

Program End...

Destructor called
```

Example 3: Now, consider the following example :

```
# Python program to illustrate destructor

class A:
    def __init__(self, bb):
        self.b = bb

class B:
    def __init__(self):
        self.a = A(self)
    def __del__(self):
        print("die")

def fun():
    b = B()

fun()
```

Output

```
die
```

In this example when the function fun() is called, it creates an instance of class B which passes itself to class A, which then sets a reference to class B and resulting in a **circular reference**.

Generally, Python's garbage collector which is used to detect these types of cyclic references would remove it but in this example the use of custom destructor marks this item as "uncollectable".

Simply, it doesn't know the order in which to destroy the objects, so it leaves them.

Therefore, **if your instances are involved in circular references they will live in memory for as long as the application run.**

NOTE : The problem mentioned in example 3 is resolved in newer versions of python, but it still exists in version < 3.4 .

Example: Destruction in recursion

In Python, you can define a destructor for a class using the __del__() method. This method is called automatically when the object is about to be destroyed by the garbage collector. Here's an example of how to use a destructor in a recursive function:

```
class RecursiveFunction:
    def __init__(self, n):
        self.n = n
        print("Recursive function initialized with n =", n)

    def run(self, n=None):
        if n is None:
            n = self.n
        if n <= 0:
            return
        print("Running recursive function with n =", n)
        self.run(n-1)

    def __del__(self):
        print("Recursive function object destroyed")

# Create an object of the class
obj = RecursiveFunction(5)

# Call the recursive function
obj.run()

# Destroy the object
del obj
```

Output

```
('Recursive function initialized with n =', 5)
('Running recursive function with n =', 5)
('Running recursive function with n =', 4)
('Running recursive function with n =', 3)
('Running recursive function with n =', 2)
('Running recursive function with n =', 1)
Recursive function object destroyed
```

In this example, we define a class RecursiveFunction with an __init__() method that takes in a parameter n. This parameter is stored as an attribute of the object.

We also define a run() method that takes in an optional parameter n. If n is not provided, it defaults to the value of self.n. The run() method runs a recursive function that prints a message to the console and calls itself with n-1.

We define a destructor using the __del__() method, which simply prints a message to the console indicating that the object has been destroyed.

We create an object of the class RecursiveFunction with n set to 5, and call the run() method. This runs the recursive function, printing a message to the console for each call.

Finally, we destroy the object using the del statement. This triggers the destructor, which prints a message to the console indicating that the object has been destroyed.

Note that in this example, the recursive function will continue running until n reaches 0. When n is 0, the function will return and the object will be destroyed by the garbage collector. The destructor will then be called automatically.

Advantages of using destructors in Python:

- **Automatic cleanup**: Destructors provide automatic cleanup of resources used by an object when it is no longer needed. This can be especially useful in cases where resources are limited, or where failure to clean up can lead to memory leaks or other issues.
- **Consistent behavior:** Destructors ensure that an object is properly cleaned up, regardless of how it is used or when it is destroyed. This helps to ensure consistent behavior and can help to prevent bugs and other issues.
- **Easy to use:** Destructors are easy to implement in Python, and can be defined using the __del__() method.
- **Supports object-oriented programming:** Destructors are an important feature of object-oriented programming, and can be used to enforce encapsulation and other principles of object-oriented design.
- **Helps with debugging:** Destructors can be useful for debugging, as they can be used to trace the lifecycle of an object and determine when it is being destroyed.

Difference between Instance Variable and Class Variable

Last Updated : 23 Jul, 2025

- **Instance Variable:** It is basically a class variable without a static modifier and is usually shared by all class instances. Across different objects, these variables can have different values. They are tied to a particular object instance of the class, therefore, the contents of an instance variable are totally independent of one object instance to others.
Example:

```
class Taxes
{
  int count;
  /*...*/
}
```

Class Variable: It is basically a static variable that can be declared anywhere at class level with static. Across different objects, these variables can have only one value. These variables are not tied to any particular object of the class, therefore, can share across all objects of the class.

Example:

```
class Taxes
{
  static int count;
  /*...*/
}
```

Tabular difference between Instance and Class variable:

Instance Variable	Class Variable
It is a variable whose value is instance-specific and now shared among instances.	It is a variable that defines a specific attribute or property for a class.
These variables cannot be shared between classes. Instead, they only belong to one specific class.	These variables can be shared between class and its subclasses.
It usually reserves memory for data that the class needs.	It usually maintains a single shared value for all instances of class even if no instance object of the class exists.
It is generally created when an instance of the class is created.	It is generally created when the program begins to execute.
It normally retains values as long as the object exists.	It normally retains values until the program terminates.
It has many copies so every object has its own personal copy of the instance	It has only one copy of the class variable so it is shared among different objects of the class.

Instance Variable	Class Variable
variable.	
It can be accessed directly by calling variable names inside the class.	It can be accessed by calling with the class name.
These variables are declared without using the static keyword.	These variables are declared using the keyword static.
Changes that are made to these variables through one object will not reflect in another object.	Changes that are made to these variables through one object will reflect in another object.

Class Method vs Static Method vs Instance Method in Python

Last Updated : 23 Jul, 2025

- Three important types of methods in Python are class methods, static methods, and instance methods. Each serves a distinct purpose and contributes to the overall flexibility and functionality of object-oriented programming in Python. In this article, we will see the difference between class method, static method, and instance method with the help of examples in Python.

Class Method in Python

Class methods are associated with the class rather than instances. They are defined using the @classmethod decorator and take the class itself as the first parameter, usually named cls. Class methods are useful for tasks that involve the class rather than the instance, such as creating class-specific behaviors or modifying class-level attributes.

Syntax Python Class Method

```
class C(object):
    @classmethod
    def fun(cls, arg1, arg2, ...):
        ....
```

fun: function that needs to be converted into a class method
returns: a class method for function.

In this example, the MyClass defines a class variable class_variable, and the class_method is a class method that increments this variable. When calling the method with different values, it updates and returns the modified class variable. Instances obj1 and obj2 have their own instance_variable.

Python3

```
class MyClass:
```

```
    class_variable = 0

    def __init__(self, value):
        self.instance_variable = value

    @classmethod
    def class_method(cls, x):
        cls.class_variable += x
        return cls.class_variable

# Creating instances of the class
obj1 = MyClass(5)
obj2 = MyClass(10)

# Calling the class method
print(MyClass.class_method(3))  # Output: 3
print(MyClass.class_method(7))  # Output: 10
```

Output

```
3

10
```

Static Method in Python

Static methods, as the name suggests, are not bound to either the class or its instances. They are defined using the @staticmethod decorator and do not take a reference to the instance or the class as their first parameter. Static methods are essentially regular functions within the class namespace and are useful for tasks that do not depend on instance-specific or class-specific data.

Syntax Python Static Method

```
class C(object):
    @staticmethod
    def fun(arg1, arg2, ...):
        ...
```

returns: a static method for function fun.

In this example, the MathOperations class features static methods add and subtract for performing basic arithmetic operations. These methods can be directly invoked on the class without creating an instance, providing a convenient and state-independent way to perform mathematical operations.

Python3

```
class MathOperations:
    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
```

```
    def subtract(x, y):
        return x - y

# Using static methods without creating an instance
print(MathOperations.add(5, 3))       # Output: 8
print(MathOperations.subtract(10, 4))  # Output: 6
```

Output

```
8

6
```

Instance Method in Python

Instance methods are the most common type of methods in Python classes. They are associated with instances of a class and operate on the instance's data. When defining an instance method, the method's first parameter is typically named self, which refers to the instance calling the method. This allows the method to access and manipulate the instance's attributes.

Syntax Instance Method

```
class MyClass:
    def instance_method(self, arg1, arg2, ...):
        # Instance method logic here
        pass
```

In this example, the Person class defines an instance method introduce which returns a formatted introduction based on the instance's name and age attributes. The instance person1 is created with the name "Kishan" and age 20, and invoking the introduce method prints a personalized introduction for that instance. Note that there's a small typo in the comment mentioning the age; it should be 20 instead of 30.

Python3

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

    def introduce(self):
        return f"Hi, I'm {self.name} and I'm {self.age} years old."


# Creating an instance of the class
person1 = Person("Kishan", 20)

# Calling the instance method
print(person1.introduce())  # Output: Hi, I'm Kishan and I'm 30 years old.
```

Output

```
Hi, I'm Kishan and I'm 20 years old.
```

Python Class Method Vs. Static Method Vs. Instance Method

Aspect	Class Method	Static Method	Instance Method
Definition	A method bound to the class itself, defined with "@classmethod"	A method that does not receive an implicit first argument ('self' or 'cls'), defined with "@staticmethod"	A method defined within a class, taking `self` as the first parameter, representing the instance.
Access	Can access class attributes and modify them.	Cannot access or modify class or instance attributes.	Can access and modify instance attributes.
First Argument	Receives 'cls' as the first argument representing the class.	No implicit first argument is passed to the method.	Receives `self` as the first argument representing the instance.
Usage	Often used for operations that modify or interact with class-level data.	Typically used for utility functions that don't depend on instance or class state.	Commonly used for operations specific to individual instances.
Inheritance	Can be overridden in subclasses, with 'cls' referring to the subclass.	Can be called directly via the class or subclass, but not overridden.	Can be overridden in subclasses, with 'self' referring to the subclass instance.
Access Modifier	Typically used when the method needs access to class-level data.	Suitable when the method doesn't rely on instance or class attributes.	Preferred when the method operates on instance-specific attributes.
Example Use	Calculating statistics across all instances of a class.	Formatting dates, performing mathematical operations.	Returning information specific to an instance, like its attributes.

Aspect	Class Method	Static Method	Instance Method
Usefulness	Useful when dealing with class-level functionality and shared attributes.	Handy for standalone functions related to the class but not dependent on its state.	Essential for modifying and accessing instance attributes and behaviors.

Data Hiding in Python

- In this article, we will discuss data hiding in Python, starting from data hiding in general to data hiding in Python, along with the advantages and disadvantages of using data hiding in python.

What is Data Hiding?
Data hiding is a concept which underlines the hiding of data or information from the user. It is one of the key aspects of Object-Oriented programming strategies. It includes object details such as data members, internal work. Data hiding excludes full data entry to class members and defends object integrity by preventing unintended changes. Data hiding also minimizes system complexity for increase robustness by limiting interdependencies between software requirements. Data hiding is also known as information hiding. In class, if we declare the data members as private so that no other class can access the data members, then it is a process of hiding data.

Data Hiding in Python:
The Python document introduces Data Hiding as isolating the user from a part of program implementation. Some objects in the module are kept internal, unseen, and unreachable to the user. Modules in the program are easy enough to understand how to use the application, but the client cannot know how the application functions. Thus, data hiding imparts security, along with discarding dependency. Data hiding in Python is the technique to defend access to specific users in the application. Python is applied in every technical area and has a user-friendly syntax and vast libraries. Data hiding in Python is performed using the __ double underscore before done prefix. This makes the class members non-public and isolated from the other classes.

Example:

```
class Solution:
  __privateCounter = 0

  def sum(self):
    self.__privateCounter += 1
    print(self.__privateCounter)
```

```
count = Solution()
count.sum()
count.sum()

# Here it will show error because it unable
# to access private member
print(count.__privateCount)
```

Output:

```
Traceback (most recent call last):
  File "/home/db01b918da68a3747044d44675be8872.py", line 11, in <module>
    print(count.__privateCount)
AttributeError: 'Solution' object has no attribute '__privateCount'
```

To rectify the error, we can access the private member through the class name :

```
class Solution:
    __privateCounter = 0

    def sum(self):
        self.__privateCounter += 1
        print(self.__privateCounter)


count = Solution()
count.sum()
count.sum()

# Here we have accessed the private data
# member through class name.
print(count._Solution__privateCounter)
```

Output:

```
1
2
2
```

Advantages of Data Hiding:

1. It helps to prevent damage or misuse of volatile data by hiding it from the public.
2. The class objects are disconnected from the irrelevant data.
3. It isolates objects as the basic concept of OOP.
4. It increases the security against hackers that are unable to access important data.

Disadvantages of Data Hiding:

1. It enables programmers to write lengthy code to hide important data from common clients.

2. The linkage between the visible and invisible data makes the objects work faster, but data hiding prevents this linkage.

Types of Inheritance in Python

- Inheritance is a key concept in object-oriented programming that allows one class (child/derived) to inherit the properties and methods of another class (parent/base). This promotes code reusability and improves maintainability. Here we a going to see the types of **inheritance in Python.**

Types of inheritance

Types of Inheritance in Python

Types of Inheritance depend upon the number of child and parent classes involved. There are four types of inheritance in Python:

Single Inheritance

Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.

Single Inheritance

Example:

```
# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")

# Derived class
class Child(Parent):
    def func2(self):
        print("This function is in child class.")

# Driver code
obj = Child()
obj.func1()
obj.func2()
```

Output

```
This function is in parent class.
This function is in child class.
```

Explanation:

- The Child class inherits the method func1() from Parent.
- It also has its own method func2().
- This shows how one class can extend another using single inheritance.

Multiple Inheritance

When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class.

Multiple Inheritance

```
# Base class 1
class Mother:
    mothername = ""

    def mother(self):
        print(self.mothername)

# Base class 2
class Father:
    fathername = ""

    def father(self):
        print(self.fathername)

# Derived class
class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)

# Driver code
s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```

Output

```
Father : RAM

Mother : SITA
```

Explanation:

- Son inherits from both Mother and Father.
- It can access both mothername and fathername.
- This demonstrates how a class can combine functionalities from multiple sources.

Multilevel Inheritance

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and a grandfather.

Multilevel Inheritance

Example:

```
# Base class
class Grandfather:
    def __init__(self, grandfathername):
        self.grandfathername = grandfathername

# Intermediate class
class Father(Grandfather):
    def __init__(self, fathername, grandfathername):
        self.fathername = fathername
        # Call the constructor of Grandfather
        Grandfather.__init__(self, grandfathername)

# Derived class
class Son(Father):
    def __init__(self, sonname, fathername, grandfathername):
        self.sonname = sonname
        # Call the constructor of Father
        Father.__init__(self, fathername, grandfathername)

    def print_name(self):
        print('Grandfather name :', self.grandfathername)
        print('Father name :', self.fathername)
        print('Son name :', self.sonname)

# Driver code
s1 = Son('Prince', 'Rampal', 'Lal mani')
print(s1.grandfathername)
s1.print_name()
```

Output

```
Lal mani

Grandfather name : Lal mani

Father name : Rampal

Son name : Prince
```

Explanation:

- Son inherits from Father, and Father inherits from Grandfather.
- Each constructor passes values up the inheritance chain using explicit constructor calls.
- All ancestor class attributes are accessible from the bottom-most class (Son).

Hierarchical Inheritance

When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.

Hierarchical Inheritance

Example:

```
# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")

# Derived class 1
class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")

# Derived class 2
class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")

# Driver code
object1 = Child1()
object2 = Child2()

object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

Output

```
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

Explanation:

- Both Child1 and Child2 inherit from the same Parent class.
- Each child can access the func1() method of Parent, but also has its own specific method.
- This pattern is useful when multiple classes need the same base functionality but also have unique behaviors.

Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance. It uses a mix like single, multiple, or multilevel inheritance within the same program. Python's method resolution order (MRO) handles such cases. Hybrid Inheritance

Example:

```
# Base class
class School:
    def func1(self):
        print("This function is in school.")

# Derived class 1 (Single Inheritance)
class Student1(School):
    def func2(self):
        print("This function is in student 1.")

# Derived class 2 (Another Single Inheritance)
class Student2(School):
    def func3(self):
        print("This function is in student 2.")

# Derived class 3 (Multiple Inheritance)
class Student3(Student1, School):
    def func4(self):
        print("This function is in student 3.")

# Driver code
obj = Student3()
obj.func1()
obj.func2()
```

Output

```
This function is in school.

This function is in student 1.
```

Explanation:

- School is the base class inherited by Student1, Student2, and Student3.
- Student1 and Student2 each inherit School (single inheritance).
- Student3 inherits both from Student1 and School (multiple inheritance).
- Python handles method resolution using the MRO, so func1() from School is resolved without ambiguity.

Polymorphism in Python

Last Updated : 22 Jan, 2026

- Polymorphism means "many forms". It refers to the ability of an entity (like a function or object) to perform different actions based on the context.

Technically, in Python, polymorphism allows same method, function or operator to behave differently depending on object it is working with. This makes code more flexible and reusable.

Real Life Example: In a backend payment system, multiple payment options are available such as Credit Card, UPI, NetBanking and Wallet. All payment types use a common method named processPayment() but different implementations:

- Credit Card Payment: validates card, talks to bank API
- UPI Payment: redirects to UPI gateway
- Wallet Payment: checks wallet balance
- NetBanking Payment: redirects to bank login

The method name remains the same, but the action changes based on the payment type.

Types of Polymorphism

Polymorphism in Python refers to ability of the same method or operation to behave differently based on object or context. It mainly includes compile-time and runtime polymorphism.

Polymorphism

Let's discuss each type in detail.

1. Compile-time Polymorphism

Compile-time polymorphism means deciding which method or operation to run during compilation, usually through method or operator overloading.

Languages like Java or C++ support this. But Python doesn't because it's dynamically typed it resolves method calls at runtime, not during compilation. So, true method overloading isn't supported in Python, though similar behavior can be achieved using default or variable arguments.

Example: This code demonstrates method overloading in Python using default and variable-length arguments. The multiply() method works with different numbers of inputs, mimicking compile-time polymorphism.

```
class Calculator:
    def multiply(self, a=1, b=1, *args):
        result = a * b
        for num in args:
            result *= num
        return result

# Create object
calc = Calculator()

# Using default arguments
print(calc.multiply())
print(calc.multiply(4))
```

```
# Using multiple arguments
print(calc.multiply(2, 3))
print(calc.multiply(2, 3, 4))
```

Output

```
1

4

6

24
```

2. Runtime Polymorphism (Overriding)

Runtime polymorphism means that the behavior of a method is decided while program is running, based on the object calling it.

In Python, this happens through **Method Overriding** a child class provides its own version of a method already defined in the parent class. Since Python is dynamic, it supports this, allowing same method call to behave differently for different object types.
Example: This code shows runtime polymorphism using method overriding. The sound() method is defined in base class Animal and overridden in Dog and Cat. At runtime, correct method is called based on object's class.

```
class Animal:
    def sound(self):
        return "Some generic sound"

class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

# Polymorphic behavior
animals = [Dog(), Cat(), Animal()]
for animal in animals:
    print(animal.sound())
```

Output

```
Bark

Meow

Some generic sound
```

Explanation: Here, sound method behaves differently depending on whether object is a Dog, Cat or Animal and this decision happens at runtime. This dynamic nature makes Python particularly powerful for runtime polymorphism.

Polymorphism in Built-in Functions

Python's built-in functions like **len()** and **max()** are polymorphic they work with different data types and return results based on type of object passed. This showcases it's dynamic nature, where same function name adapts its behavior depending on input.

Example: This code demonstrates polymorphism in Python's built-in functions handling strings, lists, numbers and characters differently while using same function name.

```
print(len("Hello"))  # String length
print(len([1, 2, 3]))  # List length

print(max(1, 3, 2))  # Maximum of integers
print(max("a", "z", "m"))  # Maximum in strings
```

Output

```
5
3
3
z
```

Polymorphism in Functions

In Python, polymorphism lets functions accept different object types as long as they support needed behavior. Using duck typing, Python focuses on whether an object has right method not its type allowing flexible and reusable code.

Example: This code demonstrates polymorphism using duck typing as perform_task() function works with different object types (Pen and Eraser), as long as they have a **.use()** method showing flexible and reusable function design.

```
class Pen:
    def use(self):
        return "Writing"

class Eraser:
    def use(self):
        return "Erasing"

def perform_task(tool):
    print(tool.use())

perform_task(Pen())
perform_task(Eraser())
```

Output

```
Writing
```

Erasing

Polymorphism in Operators

In Python, same operator (+) can perform different tasks depending on operand types. This is known as operator overloading. This flexibility is a key aspect of polymorphism in Python.

Example: This code shows operator polymorphism as + operator behaves differently based on data types adding integers, concatenating strings and merging lists all using same operator.

```
print(5 + 10)  # Integer addition
print("Hello " + "World!")  # String concatenation
print([1, 2] + [3, 4])  # List concatenation
```

Output

```
15
Hello World!
[1, 2, 3, 4]
```

Difference between abstract class and interface in Python

Last Updated : 23 Jul, 2025

- In this article, we are going to see the difference between abstract classes and interface in Python, Below are the points that are discussed in this article:
  - What is an abstract class in Python?
  - What is an interface in Python?
  - Difference between abstract class and interface in Python

What is an Abstract class in Python?

A blueprint for other classes might be thought of as an abstract class. You may use it to define a collection of methods that are required for all subclasses derived from the abstract class. An abstract class is one that includes one or more abstract methods. A method that has a declaration but no implementation is said to be abstract. We use an abstract class for creating huge functional units. An abstract class is used to offer a standard interface for various implementations of a component.

Example:

Python does not come with any abstract classes by default. Python has a module called ABC that serves as the foundation for building Abstract Base Classes (ABC). ABC works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base. When a method is decorated with the keyword @abstractmethod, it becomes abstract.

```
from abc import ABC, abstractmethod
class Animal(ABC):
```

```
    def move(self):
        pass

class Human(Animal):

    def move(self):
        print("I can walk and run")

class Snake(Animal):

    def move(self):
        print("I can crawl")

class Dog(Animal):

    def move(self):
        print("I can bark")

class Lion(Animal):

    def move(self):
        print("I can roar")

# Driver code
R = Human()
R.move()

K = Snake()
K.move()

R = Dog()
R.move()

K = Lion()
K.move()
```

Output:

```
I can walk and run

I can crawl

I can bark

I can roar
```

What is an Interface in Python?

The interface in object-oriented languages like Python is a set of method signatures that the implementing class is expected to provide. Writing ordered code and achieving abstraction are both possible through interface implementation.

Example:
Python "object interfaces" are implemented in the module zope.interface. It is maintained by the Zope Toolkit project. Two objects, "Interface" and "Attribute," are directly exported by the package. Several helper methods are also exported by it. Compared to Python's built-in abc module, it strives to give stronger semantics and better error messages.

Declaring interface: In Python, an interface is defined using Python class statements and is a subclass of interface.Interface which is the parent interface for all interfaces.

```
import zope.interface


class MyInterface(zope.interface.Interface):
    x = zope.interface.Attribute("foo")
    def method1(self, x):
        pass
    def method2(self):
        pass

print(type(MyInterface))
print(MyInterface.__module__)
print(MyInterface.__name__)

# get attribute
x = MyInterface['x']
print(x)
print(type(x))
```

Output:

```
An abstract
```

Implementing interface: The interface acts as a blueprint for designing classes, so interfaces are implemented using the implementer decorator on the class. If a class implements an interface, then the instances of the class provide the interface. Objects can provide interfaces directly, in addition to what their classes implement.

Syntax: *@zope.interface.implementer(*interfaces)*
class Class_name:
 # methods

```
import zope.interface


class MyInterface(zope.interface.Interface):
    x = zope.interface.Attribute("foo")
    def method1(self, x):
```

```
    pass
  def method2(self):
    pass

@zope.interface.implementer(MyInterface)
class MyClass:
  def method1(self, x):
    return x**2
  def method2(self):
    return "foo"

obj1 = MyClass()

print(obj1.method1(5))
print(obj1.method2())
```

Output:

We declared that MyClass implements MyInterface. This means that instances of MyClass provide MyInterface.

```
25
foo
```

Difference between abstract class and interface in Python?

	Abstract Class	*Interface*
1.	An abstract features developer's class can consist of abstract as well as concrete methods	All methods of an interface are abstract
2.	It is used when there are some common feature shared by all objects	It is used when all the feature need to be implemented differently for different objects
3.	Its developer responsibility to create a child class for the features of an abstract class	Any 3rd person will responsible for creating a child class

4.	It is comparatively fast	It is comparatively slow

Inheritance and Composition in Python

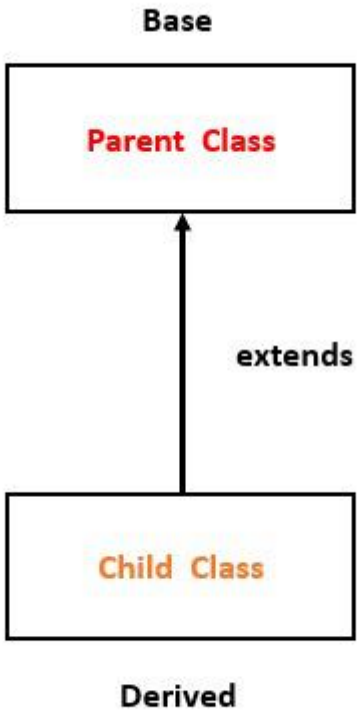
Last Updated : 15 Jul, 2025

- **Prerequisite - Classes and Objects in Python**

This article will compare and highlight the features of **is-a relation** and **has-a relation** in Python.

What is Inheritance (Is-A Relation)?

It is a concept of Object-Oriented Programming. Inheritance is a mechanism that allows us to inherit all the properties from another class. The class from which the properties and functionalities are utilized is called the **parent class (**also called as **Base Class)**. The class which uses the properties from another class is called as **Child Class (**also known as **Derived class)**. Inheritance is also called an **Is-A Relation**.



Inheritance - diagrammatic representation

In the figure above, classes are represented as boxes. The inheritance relationship is represented by an arrow pointing from **Derived Class(Child Class)** to **Base Class(Parent Class)**. The extends keyword denotes that the **Child Class** is inherited or derived from **Parent Class**.

Syntax :

```
# Parent class
```

```
class Parent :
        # Constructor
        # Variables of Parent class


        # Methods
        ...


        ...



# Child class inheriting Parent class
class Child(Parent) :
        # constructor of child class
        # variables of child class
        # methods of child class


        ...


        ...
```

Example :

```
# parent class
class Parent:

   # parent class method
   def m1(self):
      print('Parent Class Method called...')

# child class inheriting parent class
class Child(Parent):

   # child class constructor
   def __init__(self):
      print('Child Class object created...')

   # child class method
   def m2(self):
      print('Child Class Method called...')
```

```
# creating object of child class
obj = Child()

# calling parent class m1() method
obj.m1()

# calling child class m2() method
obj.m2()
```

Output

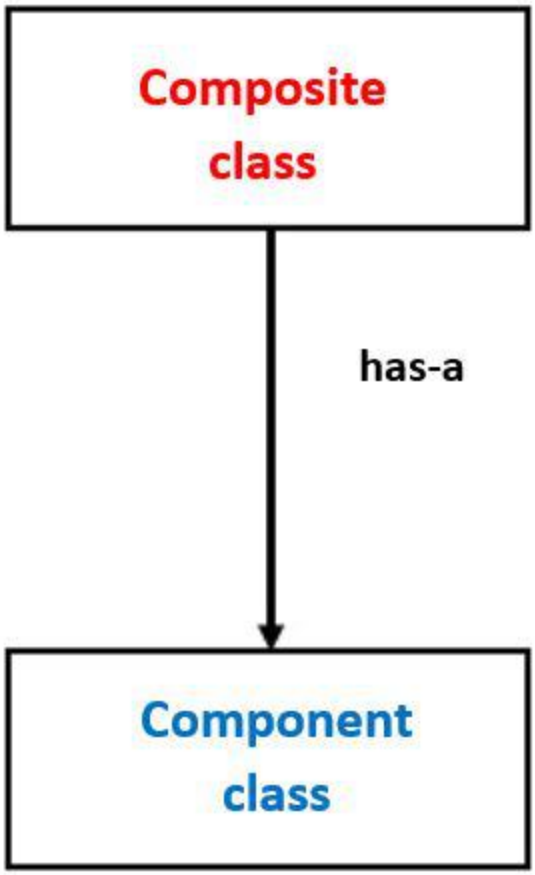
```
Child Class object created...

Parent Class Method called...

Child Class Method called...
```

What is Composition (Has-A Relation)?

It is one of the fundamental concepts of Object-Oriented Programming. In this concept, we will describe a class that references to one or more objects of other classes as an Instance variable. Here, by using the class name or by creating the object we can access the members of one class inside another class. It enables creating complex types by combining objects of different classes. It means that a class Composite can contain an object of another class Component. This type of relationship is known as **Has-A Relation**.



composition - diagrammatic representation

In the above figure Classes are represented as boxes with the class name **Composite** and **Component** representing Has-A relation between both of them.

```
class A :


    # variables of class A

    # methods of class A

    ...

    ...


class B :

    # by using "obj" we can access member's of class A.
```

```
    obj = A()


    # variables of class B
    # methods of class B


    ...
    ...
```

Example :

```
class Component:

   # composite class constructor
    def __init__(self):
        print('Component class object created...')

    # composite class instance method
    def m1(self):
        print('Component class m1() method executed...')


class Composite:

    # composite class constructor
    def __init__(self):

        # creating object of component class
        self.obj1 = Component()

        print('Composite class object also created...')

     # composite class instance method
    def m2(self):

        print('Composite class m2() method executed...')

        # calling m1() method of component class
        self.obj1.m1()


# creating object of composite class
obj2 = Composite()

# calling m2() method of composite class
```

obj2.m2()

Output

```
Component class object created...
Composite class object also created...
Composite class m2() method executed...
Component class m1() method executed...
```

Explanation:

- In the above example, we created two classes **Composite** and **Component** to show the **Has-A Relation** among them.
- In the **Component class**, we have one constructor and an instance method **m1()**.
- Similarly, in **Composite class**, we have one constructor in which we created an object of **Component Class.** Whenever we create an object of **Composite Class**, the object of the **Component class** is automatically created.
- Now in **m2()** method of **Composite class** we are calling **m1()** method of **Component Class** using instance variable **obj1** in which reference of **Component Class** is stored.
- Now, whenever we call **m2()** method of **Composite Class,** automatically **m1()** method of **Component Class** will be called.

Composition vs Inheritance

It's big confusing among most of the people that both the concepts are pointing to **Code Reusability** then **what is** the **difference b/w Inheritance and Composition and when to use Inheritance and when to use Composition?**

Inheritance is used where a class wants to derive the nature of parent class and then modify or extend the functionality of it. **Inheritance** will extend the functionality with extra features allows **overriding of methods**, but in the case of **Composition**, we can only use that class we can not modify or extend the functionality of it. It will not provide extra features. Thus, when one needs to use the class as it without any modification, the composition is recommended and when one needs to change the behavior of the method in another class, then inheritance is recommended.

Comment

R

ronil

Dunder or magic methods in Python

Last Updated : 12 Jan, 2026

- Python magic (dunder) methods are special methods with double underscores __ that enable operator overloading and custom object behavior.

The below code displays the magic methods inherited by int class.
print(dir(int))

Output

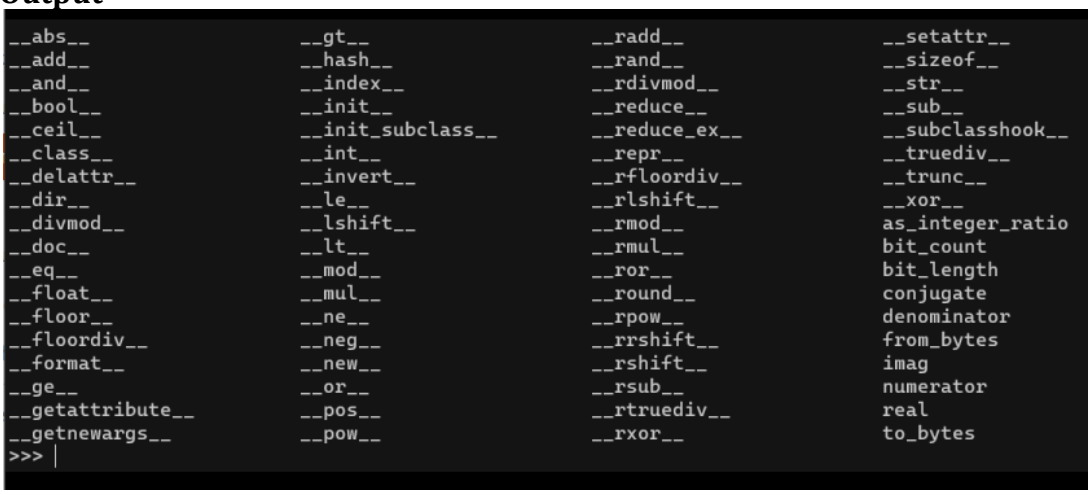
```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__',
'__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '_...
```

Explanation: dir(int) lists all attributes and methods of the int class, including magic (dunder) methods like __add__, __str__, etc.
You can list magic (dunder) methods for any Python object via the terminal by following these steps:

- Open the terminal or command prompt and type python3 to enter the Python console.
- Use the dir() function to list all methods, e.g.

>>> dir(int)

Output



Python Magic Methods
Below are the lists of Python magic methods and their uses.

1. Initialization and Construction

- __new__**:** To get called in an object's instantiation.
- __init__: To get called by the __new__ method.
- __del__**:** It is the destructor.

2. Numeric magic methods

- __trunc__(self): Implements behavior for math.trunc()
- __round__(self,n): Implements behavior for the built-in round()
- __abs__(self): Implements behavior for the built-in abs()
- __neg__(self): Implements behavior for negation
- __pos__(self)**:** Implements behavior for unary positive

3. Arithmetic operators

- __add__(self, other): Implements behavior for the + operator (addition).
- __sub__(self, other): Implements behavior for the - operator (subtraction).
- __mul__(self, other): Implements behavior for the * operator (multiplication).
- __truediv__(self, other): Implements behavior for the / operator (true division).
- __mod__(self, other): Implements behavior for the % operator (modulus).

4. String Magic Methods

- __str__(self)**:** Defines behavior for when str() is called on an instance of your class.
- __repr__(self): **T**o get called by built-int repr() method to return a machine readable representation of a type.
- __hash__(self)**:** It has to return an integer, and its result is used for quick key comparison in dictionaries.
- __bool__(self): Returns True or False to determine the truth value of an object, used in conditions and bool() calls.
- __dir__(self): This method to return a list of attributes of a class.

5. Comparison magic methods

- __eq__(self, other): Defines behavior for the equality operator, ==.
- __ne__(self, other): Defines behavior for the inequality operator, !=.
- __lt__(self, other): Defines behavior for the less-than operator, <.
- __gt__(self, other): Defines behavior for the greater-than operator, >.
- __ge__(self, other): Defines behavior for the greater-than-or-equal-to operator, >=.

Examples of Magic Methods

1. __init__ Method

__init__ method is automatically called when a new instance of a class is created. It is used to initialize the object's attributes, similar to constructors in languages like C++, Java, C#, or PHP.

```
class String:

    def __init__(self, string):
        self.string = string

if __name__ == '__main__':

    s1 = String('Hello')
    print(s1)
```

Output

```
<__main__.String object at 0x791ce72aa900>
```

Explanation:

- **__init__(self, string):** Constructor that initializes the object with a string.
- **s1= String('Hello'):** Creates an instance of String with 'Hello'.
- **print(s1):** Tries to print the object.

2. __repr__ method

__repr__method defines the official string representation of an object, primarily used for debugging. By default, it shows the type and memory address of the object.

```
class String:
    def __init__(self, string):
        self.string = string

    def __repr__(self):
        return 'Object: {}'.format(self.string)

if __name__ == '__main__':
    str1 = String('Hello')
    print(str1)
```

Output

```
Object: Hello
```

Explanation:

- **__repr__(self):** Returns the official string representation of the object; used when printing or debugging.
- **str1 = String('Hello'):** Creates an instance of String with 'Hello'.

3. __add__ method

__add__ method method defines how objects of a class are added together using the '+' operator. It allows operator overloading.

Adding __add__ method to String class:

```
class String:

    def __init__(self, string):
        self.string = string

    def __repr__(self):
        return 'Object: {}'.format(self.string)
```

```
    def __add__(self, other):
        return self.string + other

if __name__ == '__main__':

    string1 = String('Hello')
    print(string1 +' Geeks')
```

Output

```
Hello Geeks
```

Explanation:

- **__init__(self, string):** Initializes the object with a string.
- **__repr__(self):** Provides a readable string representation of the object for debugging.
- **__add__(self, other):** Overloads the + operator to allow adding a string to the object's string attribute.
- string1 + ' Geeks' – Calls __add__, returning 'Hello Geeks'.