Website: <https://www.programiz.com/python-programming/inheritance>

**Python Inheritance**

In this tutorial, we will learn about Python inheritance and its types with the help of examples.

Like any other OOP languages, Swift also supports the concept of class inheritance.

Inheritance allows us to create a new class from an existing class.

The new class that is created is known as **subclass** (child or derived class) and the existing class from which the child class is derived is known as **superclass** (parent or base class).

## Python Inheritance Syntax

Here's the syntax of the inheritance in Python,

# define a superclass

class super\_class:

# attributes and method definition

# inheritance

class sub\_class(super\_class):

# attributes and method of super\_class

# attributes and method of sub\_class

Here, we are inheriting the sub\_class class from the super\_class class.

## Example 1: Python Inheritance

class Animal:

# attribute and method of the parent class

name = ""

def eat(self):

print("I can eat")

# inherit from Animal

class Dog(Animal):

# new method in subclass

def display(self):

# access name attribute of superclass using self

print("My name is ", self.name)

# create an object of the subclass

labrador = Dog()

# access superclass attribute and method

labrador.name = "Rohu"

labrador.eat()

# call subclass method

labrador.display()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

I can eat

My name is Rohu

In the above example, we have derived a subclass Dog from a superclass Animal. Notice the statements,

labrador.name = "Rohu"

labrador.eat()

Here, we are using labrador (object of Dog) to access name and eat() of the Animal class. This is possible because the subclass inherits all attributes and methods of the superclass.

Also, we have accessed the name attribute inside the method of the Dog class using self.

## is-a relationship

In Python, inheritance is an **is-a** relationship. That is, we use inheritance only if there exists an **is-a** relationship between two classes. For example,

1. **Car** is a **Vehicle**
2. **Apple** is a **Fruit**
3. **Cat** is an **Animal**

Here, **Car** can inherit from **Vehicle**, **Apple** can inherit from **Fruit**, and so on.

### Example 2: Inheritance in Python

Let's take a look at another example of inheritance in Python,

A polygon is a closed figure with **3** or more sides. Say, we have a class called Polygon defined as follows,

class Polygon:

def \_\_init\_\_(self, no\_of\_sides):

self.n = no\_of\_sides

self.sides = [0 for i in range(no\_of\_sides)]

def inputSides(self):

self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

def dispSides(self):

for i in range(self.n):

print("Side",i+1,"is",self.sides[i])

This class has data attributes to store the number of sides n and magnitude of each side as a list called sides.

* The inputSides() method takes in the magnitude of each side
* The dispSides() method displays these side lengths

A triangle is a polygon with **3** sides. So, we can create a class called Triangle which **inherits** from Polygon. This makes all the attributes of Polygon class available to the Triangle class.

We don't need to define them again **(code reusability)**. Triangle can be defined as follows.

class Triangle(Polygon):

def \_\_init\_\_(self):

Polygon.\_\_init\_\_(self,3)

def findArea(self):

a, b, c = self.sides

# calculate the semi-perimeter

s = (a + b + c) / 2

area = (s\*(s-a)\*(s-b)\*(s-c)) \*\* 0.5

print('The area of the triangle is %0.2f' %area)

However, the Triangle class has a new method findArea() to find and print the area of the triangle.

Now let's see the complete working code of the example above including creating an object,

class Polygon:

# Initializing the number of sides

def \_\_init\_\_(self, no\_of\_sides):

self.n = no\_of\_sides

self.sides = [0 for i in range(no\_of\_sides)]

def inputSides(self):

self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

# method to display the length of each side of the polygon

def dispSides(self):

for i in range(self.n):

print("Side",i+1,"is",self.sides[i])

class Triangle(Polygon):

# Initializing the number of sides of the triangle to 3 by

# calling the \_\_init\_\_ method of the Polygon class

def \_\_init\_\_(self):

Polygon.\_\_init\_\_(self,3)

def findArea(self):

a, b, c = self.sides

# calculate the semi-perimeter

s = (a + b + c) / 2

# Using Heron's formula to calculate the area of the triangle

area = (s\*(s-a)\*(s-b)\*(s-c)) \*\* 0.5

print('The area of the triangle is %0.2f' %area)

# Creating an instance of the Triangle class

t = Triangle()

# Prompting the user to enter the sides of the triangle

t.inputSides()

# Displaying the sides of the triangle

t.dispSides()

# Calculating and printing the area of the triangle

t.findArea()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Enter side 1 : 3

Enter side 2 : 5

Enter side 3 : 4

Side 1 is 3.0

Side 2 is 5.0

Side 3 is 4.0

The area of the triangle is 6.00

Here, we can see that even though we did not define methods like inputSides() or dispSides() for class Triangle separately, we were able to use them.

If an attribute is not found in the class itself, the search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

## Method Overriding in Python Inheritance

In the previous example, we see the object of the subclass can access the method of the superclass.

**However, what if the same method is present in both the superclass and subclass?**

In this case, the method in the subclass overrides the method in the superclass. This concept is known as method overriding in Python.

### Example: Method Overriding

class Animal:

# attributes and method of the parent class

name = ""

def eat(self):

print("I can eat")

# inherit from Animal

class Dog(Animal):

# override eat() method

def eat(self):

print("I like to eat bones")

# create an object of the subclass

labrador = Dog()

# call the eat() method on the labrador object

labrador.eat()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

I like to eat bones

In the above example, the same method eat() is present in both the Dog class and the Animal class.

Now, when we call the eat() method using the object of the Dog subclass, the method of the Dog class is called.

This is because the eat() method of the Dog subclass overrides the same method of the Animal superclass.

## The super() Method in Python Inheritance

Previously we saw that the same method in the subclass overrides the method in the superclass.

However, if we need to access the superclass method from the subclass, we use the super() method. For example,

class Animal:

name = ""

def eat(self):

print("I can eat")

# inherit from Animal

class Dog(Animal):

# override eat() method

def eat(self):

# call the eat() method of the superclass using super()

super().eat()

print("I like to eat bones")

# create an object of the subclass

labrador = Dog()

labrador.eat()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

I can eat

I like to eat bones

In the above example, the eat() method of the Dog subclass overrides the same method of the Animal superclass.

Inside the Dog class, we have used

# call method of superclass

super().eat()

to call the eat() method of the Animal superclass from the Dog subclass.

So, when we call the eat() method using the labrador object

# call the eat() method

labrador.eat()

Both the overridden and the superclass version of the eat() method is executed.

## Uses of Inheritance

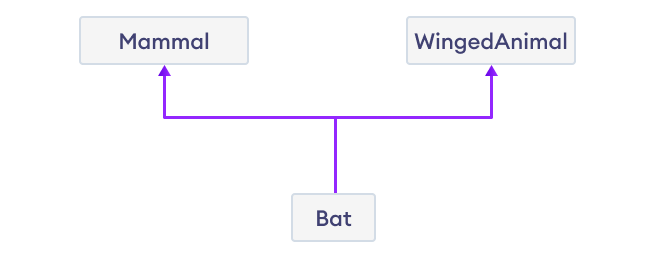
1. Since a child class can inherit all the functionalities of the parent's class, this allows code reusability.
2. Once a functionality is developed, you can simply inherit it. No need to reinvent the wheel. This allows for cleaner code and easier to maintain.
3. Since you can also add your own functionalities in the child class, you can inherit only the useful functionalities and define other required features.

# Python Multiple Inheritance

In this tutorial, we'll learn about multiple inheritance in Python with the help of examples.

A [class](https://www.programiz.com/python-programming/class) can be derived from more than one superclass in Python. This is called multiple [inheritance](https://www.programiz.com/python-programming/inheritance).

For example, A class Bat is derived from superclasses Mammal and WingedAnimal. It makes sense because bat is a mammal as well as a winged animal.

Multiple Inheritance

## Python Multiple Inheritance Syntax

class SuperClass1:

# features of SuperClass1

class SuperClass2:

# features of SuperClass2

class MultiDerived(SuperClass1, SuperClass2):

# features of SuperClass1 + SuperClass2 + MultiDerived class

Here, the MultiDerived class is derived from SuperClass1 and SuperClass2 classes.

## Example: Python Multiple Inheritance

class Mammal:

def mammal\_info(self):

print("Mammals can give direct birth.")

class WingedAnimal:

def winged\_animal\_info(self):

print("Winged animals can flap.")

class Bat(Mammal, WingedAnimal):

pass

# create an object of Bat class

b1 = Bat()

b1.mammal\_info()

b1.winged\_animal\_info()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Mammals can give direct birth.

Winged animals can flap.

In the above example, the Bat class is derived from two super classes: Mammal and WingedAnimal. Notice the statements,

b1 = Bat()

b1.mammal\_info()

b1.winged\_animal\_info()

Here, we are using b1 (object of Bat) to access mammal\_info() and winged\_animal\_info() methods of the Mammal and the WingedAnimal class respectively.

## Python Multilevel Inheritance

In Python, not only can we derive a class from the superclass but you can also derive a class from the derived class. This form of inheritance is known as **multilevel inheritance**.

Here's the syntax of the multilevel inheritance,

class SuperClass:

# Super class code here

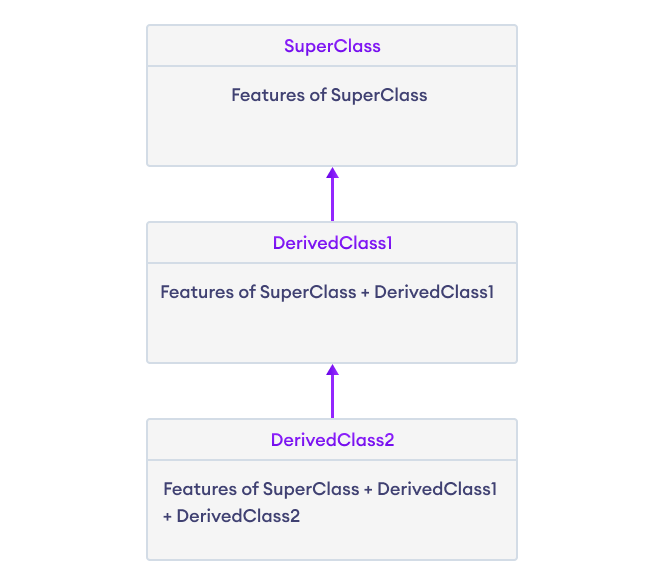
class DerivedClass1(BaseClass):

# Derived class 1 code here

class DerivedClass2(DerivedClass1):

# Derived class 2 code here

Here, the DerivedClass1 class is derived from the SuperClass class, and the DerivedClass2 class is derived from the DerivedClass1 class.

Multilevel Inheritance in Python

## Example: Python Multilevel Inheritance

class SuperClass:

def super\_method(self):

print("Super Class method called")

# define class that derive from SuperClass

class DerivedClass1(SuperClass):

def derived1\_method(self):

print("Derived class 1 method called")

# define class that derive from DerivedClass1

class DerivedClass2(DerivedClass1):

def derived2\_method(self):

print("Derived class 2 method called")

# create an object of DerivedClass2

d2 = DerivedClass2()

d2.super\_method() # Output: "Super Class method called"

d2.derived1\_method() # Output: "Derived class 1 method called"

d2.derived2\_method() # Output: "Derived class 2 method called"

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Super Class method called

Derived class 1 method called

Derived class 2 method called

In the above example, DerivedClass2 is derived from DerivedClass1, which is derived from SuperClass.

It means that DerivedClass2 inherits all the attributes and methods of both DerivedClass1 and SuperClass.

Hence, we are using d2 (object of DerivedClass2) to call methods from SuperClass, DerivedClass1, and DerivedClass2.

## Method Resolution Order (MRO) in Python

If two superclasses have the same method name and the derived class calls that method, Python uses the MRO to search for the right method to call. For example,

class SuperClass1:

def info(self):

print("Super Class 1 method called")

class SuperClass2:

def info(self):

print("Super Class 2 method called")

class Derived(SuperClass1, SuperClass2):

pass

d1 = Derived()

d1.info()

# Output: "Super Class 1 method called"

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, SuperClass1 and SuperClass2 both of these classes define a method info().

So when info() is called using the d1 object of the Derived class, Python uses the **MRO** to determine which method to call.

In this case, the **MRO** specifies that methods should be inherited from the leftmost superclass first, so info() of SuperClass1 is called rather than that of SuperClass2.