




PYTHON



OBJECT ORIENTED PROGRAMMING



INTRODUCTION TO OOP



Like other general purpose languages, python is also an object-oriented language since its beginning. Python is an object-oriented programming language. It allows us to develop applications using an Object Oriented approach. In Python, we can easily create and use classes and objects.



Major principles of object-oriented programming system are given below.

1. Object
2. Class
3. Method
4. Inheritance
5. Polymorphism
6. Data Abstraction
7. Encapsulation

OBJECT

The object is an entity that has state and behavior. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute `__doc__`, which returns the doc string defined in the function source code.

CLASS

The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. For example: if you have an employee class then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.

METHOD

The method is a function that is associated with an object. In Python, a method is not unique to class instances. Any object type can have methods.

INHERITANCE

Inheritance is the most important aspect of object-oriented programming which simulates the real world concept of inheritance. It specifies that the child object acquires all the properties and behaviors of the parent object.

By using inheritance, we can create a class which uses all the properties and behavior of another class. The new class is known as a derived class or child class, and the one whose properties are acquired is known as a base class or parent class.

It provides re-usability of the code.

POLYMORPHISM

Polymorphism contains two words "poly" and "morphs". Poly means many and Morphs means form, shape. By polymorphism, we understand that one task can be performed in different ways. For example You have a class animal, and all animals speak. But they speak differently. Here, the "speak" behavior is polymorphic in the sense and depends on the animal. So, the abstract "animal" concept does not actually "speak", but specific animals (like dogs and cats) have a concrete implementation of the action "speak".

ENCAPSULATION

Encapsulation is also an important aspect of object-oriented programming. It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

DATA ABSTRACTION

Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonym because data abstraction is achieved through encapsulation.


Abstraction is used to hide internal details and show only functionalities. Abstracting something means to give names to things so that the name captures the core of what a function or a whole program does.

OBJECT-ORIENTED VS PROCEDURE-ORIENTED PROGRAMMING LANGUAGES

Object-oriented Programming	Procedural Programming
Object-oriented programming is the problem-solving approach and used where computation is done by using objects.	Procedural programming uses a list of instructions to do computation step by step.
It makes the development and maintenance easier.	In procedural programming, It is not easy to maintain the codes when the project becomes lengthy.
It simulates the real world entity. So real-world problems can be easily solved through oops.	It doesn't simulate the real world. It works on step by step instructions divided into small parts called functions.
It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere.	Procedural language doesn't provide any proper way for data binding, so it is less secure.
Example of object-oriented programming languages is C++, Java, .Net, Python, C#, etc.	Example of procedural languages are: C, Fortran, Pascal, VB etc.



PYTHON CLASS AND OBJECTS



As we have already discussed, a class is a virtual entity and can be seen as a blueprint of an object. The class came into existence when it is instantiated. Let's understand it by an example.

Suppose a class is a prototype of a building. A building contains all the details about the floor, doors, windows, etc. we can make as many buildings as we want, based on these details. Hence, the building can be seen as a class, and we can create as many objects of this class.

On the other hand, the object is the instance of a class. The process of creating an object can be called as instantiation.

In this section of the tutorial, we will discuss creating classes and objects in python. We will also talk about how an attribute is accessed by using the class object.


CREATING CLASSES IN PYTHON

In python, a class can be created by using the keyword `class` followed by the class name. The syntax to create a class is given below.

Syntax

```
class ClassName:
```

```
    #statements
```

In python, we must notice that each class is associated with a documentation string which can be accessed by using **<class-name>.__doc__**. A class contains a statement suite including fields, constructor, function, etc. definition.

Consider the following example to create a class Employee which contains two fields as Employee id, and name.

The class also contains a function display() which is used to display the information of the Employee.




```
class Employee:
```

```
    id = 10;
```

```
    name = "Demo Text"
```

```
    def display (self):
```

```
        print(self.id,self.name)
```



Here, the self is used as a reference variable which refers to the current class object. It is always the first argument in the function definition. However, using self is optional in the function call.

CREATING AN INSTANCE OF THE CLASS

A class needs to be instantiated if we want to use the class attributes in another class or method. A class can be instantiated by calling the class using the class name.

The syntax to create the instance of the class is given below.

```
<object-name> = <class-name>(<arguments>)
```

The following example creates the instance of the class Employee defined in the above example.



```
class Employee:
```

```
    id = 10;
```

```
    name = "John"
```

```
    def display (self):
```

```
        print("ID: %d \nName: %s"%(self.id,self.name))
```

```
emp = Employee()
```

```
emp.display()
```



PYTHON CONSTRUCTOR



A constructor is a special type of method (function) which is used to initialize the instance members of the class.

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

Constructor definition is executed when we create the object of this class. Constructors also verify that there are enough resources for the object to perform any start-up task.

CREATING THE CONSTRUCTOR IN PYTHON

In python, the method `__init__` simulates the constructor of the class. This method is called when the class is instantiated. We can pass any number of arguments at the time of creating the class object, depending upon `__init__` definition. It is mostly used to initialize the class attributes. Every class must have a constructor, even if it simply relies on the default constructor.

Consider the following example to initialize the Employee class attributes.


```
class Employee:
```

```
    def __init__(self,name,id):
```

```
        self.id = id;
```

```
        self.name = name;
```

```
    def display (self):
```

```
        print("ID: %d \nName: %s"%(self.id,self.name))
```

```
emp1 = Employee("John",101)
```

```
emp2 = Employee("David",102)
```

```
emp1.display();
```

```
emp2.display();
```

COUNTING THE NUMBER OF OBJECTS OF A CLASS

```
class Student:
    count = 0

    def __init__(self):
        Student.count = Student.count + 1

s1=Student()
s2=Student()
s3=Student()

print("The number of students:",Student.count)
```

PYTHON NON-PARAMETERIZED CONSTRUCTOR EXAMPLE

```
class Student:
```

```
    # Constructor - non parameterized
```

```
    def __init__(self):
```

```
        print("This is non parametrized constructor")
```

```
    def show(self,name):
```

```
        print("Hello",name)
```

```
student = Student()
```

```
student.show("John")
```

PYTHON PARAMETERIZED CONSTRUCTOR EXAMPLE

```
class Student:
    # Constructor - parameterized
    def __init__(self, name):
        print("This is parametrized constructor")
        self.name = name
    def show(self):
        print("Hello",self.name)
student = Student("John")
student.show()
```

PYTHON IN-BUILT CLASS FUNCTIONS

Function	Description
<code>getattr(obj,name,default)</code>	It is used to access the attribute of the object.
<code>setattr(obj, name,value)</code>	It is used to set a particular value to the specific attribute of an object.
<code>delattr(obj, name)</code>	It is used to delete a specific attribute.
<code>hasattr(obj, name)</code>	It returns true if the object contains some specific attribute.

class Student:

def __init__(self,name,id,age):

 self.name = name;

 self.id = id;

 self.age = age

s = Student("John",101,22)

print(getattr(s,'name'))

setattr(s,"age",23)

print(getattr(s,'age'))

print(hasattr(s,'id'))

delattr(s,'age')

print(s.age)

BUILT-IN CLASS ATTRIBUTES

Along with the other attributes, a python class also contains some built-in class attributes which provide information about the class.

The built-in class attributes are given in the below table.

Attribute	Description
<code>__dict__</code>	It provides the dictionary containing the information about the class namespace.
<code>__doc__</code>	It contains a string which has the class documentation
<code>__name__</code>	It is used to access the class name.
<code>__module__</code>	It is used to access the module in which, this class is defined.
<code>__bases__</code>	It contains a tuple including all base classes.


```
class Student:
```

```
    def __init__(self,name,id,age):
```

```
        self.name = name;
```

```
        self.id = id;
```

```
        self.age = age
```

```
    def display_details(self):
```

```
        print("Name:%s, ID:%d, age:%d"%(self.name,self.id))
```

```
s = Student("John",101,22)
```


```
print(s.__doc__)
```

```
print(s.__dict__)
```

```
print(s.__module__)
```



PYTHON INHERITANCE

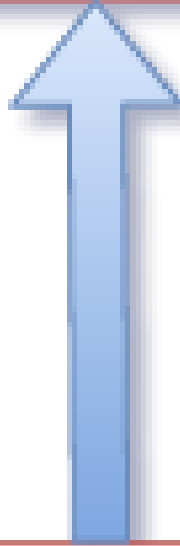


Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial, we will discuss inheritance in detail.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.

Base Class



Derived Class



```
class derived-class(base class):  
    <class-suite>
```



A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

```
class derive-class(<base class 1>, <base class 2>, ..... <base class n>):  
    <class - suite>
```



```
class Animal:
```

```
    def speak(self):
```

```
        print("Animal Speaking")
```

```
#child class Dog inherits the base class Animal
```

```
class Dog(Animal):
```

```
    def bark(self):
```

```
        print("dog barking")
```

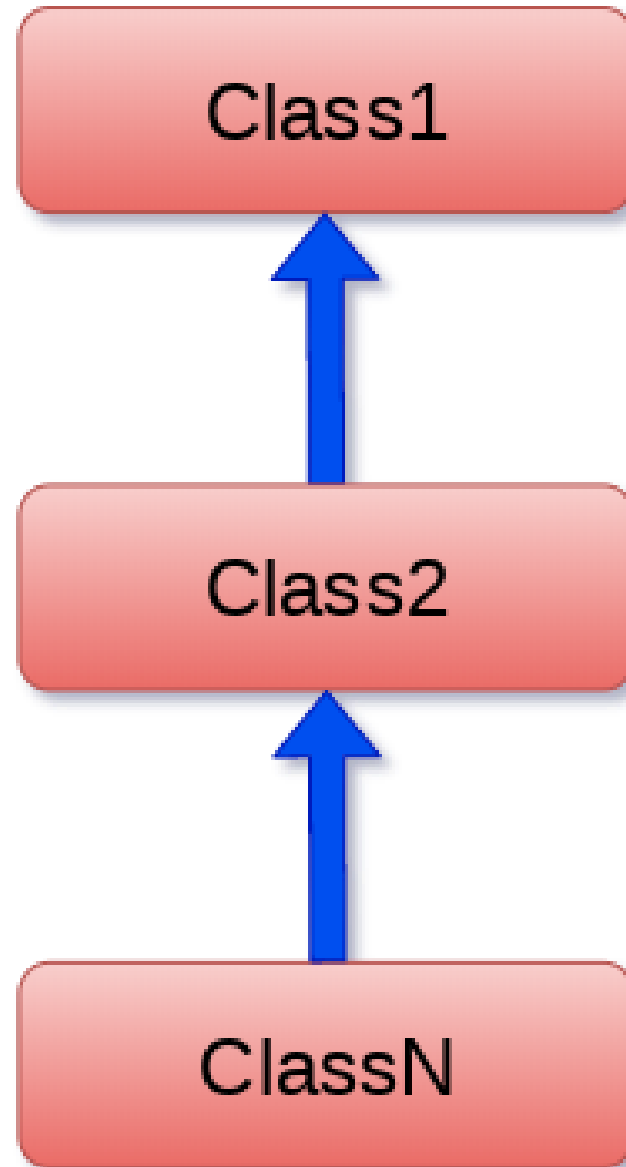
```
d = Dog()
```

```
d.bark()
```

```
d.speak()
```

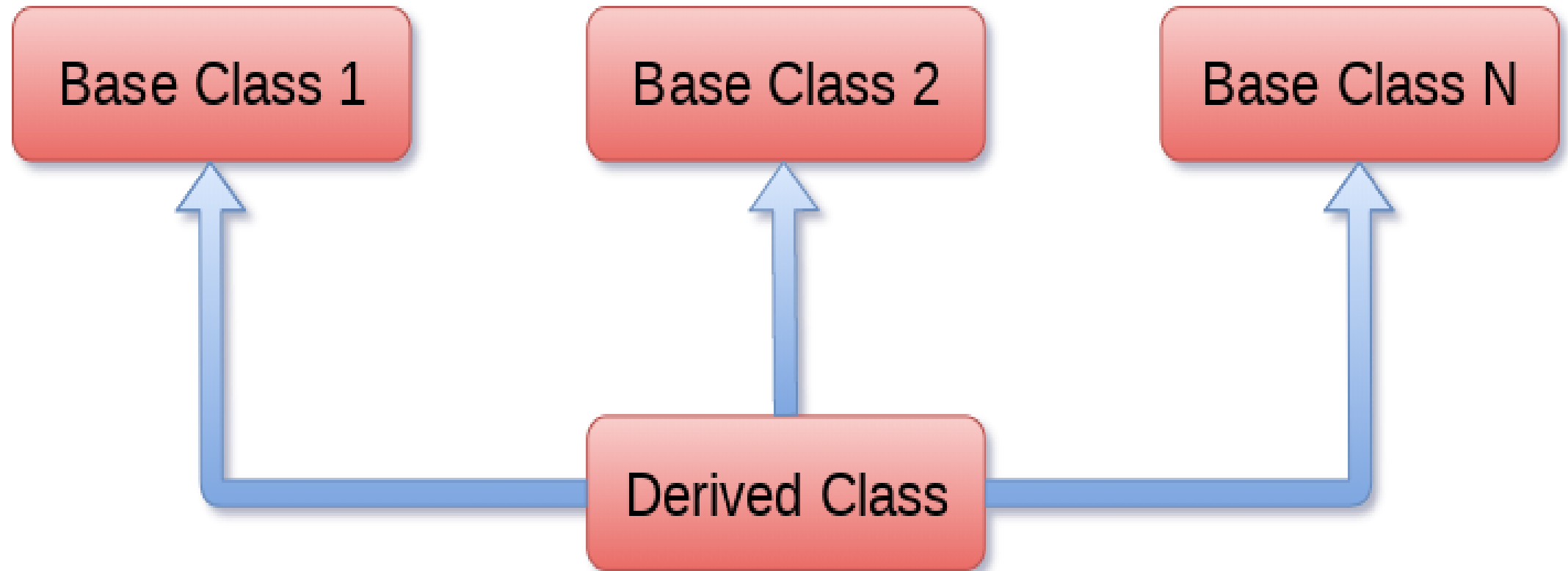
PYTHON MULTI-LEVEL INHERITANCE

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is achieved when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is achieved in python.



PYTHON MULTIPLE INHERITANCE

Python provides us the flexibility to inherit multiple base classes in the child class.





```
class Base1:
```

```
    <class-suite>
```

```
class Base2:
```

```
    <class-suite>
```

```
class BaseN:
```

```
    <class-suite>
```

```
class Derived(Base1, Base2, ..... BaseN):
```

```
    <class-suite>
```

THE ISSUBCLASS(SUB,SUP) METHOD

The `issubclass(sub, sup)` method is used to check the relationships between the specified classes. It returns `true` if the first class is the subclass of the second class, and `false` otherwise.

Consider the following example.

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(issubclass(Derived,Calculation2))
print(issubclass(Calculation1,Calculation2))
```

THE ISINSTANCE (OBJ, CLASS) METHOD

The `isinstance()` method is used to check the relationship between the objects and classes. It returns `true` if the first parameter, i.e., `obj` is the instance of the second parameter, i.e., `class`.

Consider the following example.

```
class Calculation1:
```

```
    def Summation(self,a,b):
```

```
        return a+b;
```

```
class Calculation2:
```

```
    def Multiplication(self,a,b):
```

```
        return a*b;
```

```
class Derived(Calculation1,Calculation2):
```

```
    def Divide(self,a,b):
```


```
        return a/b;
```

```
d = Derived()
```

```
print(isinstance(d,Derived))
```





METHOD OVERRIDING



We can provide some specific implementation of the parent class method in our child class. When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding. We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

Consider the following example to perform method overriding in python.




```
class Animal:
    def speak(self):
        print("speaking")

class Dog(Animal):
    def speak(self):
        print("Barking")

d = Dog()
d.speak()
```



DATA ABSTRACTION IN PYTHON



Abstraction is an important aspect of object-oriented programming. In python, we can also perform data hiding by adding the double underscore (____) as a prefix to the attribute which is to be hidden. After this, the attribute will not be visible outside of the class through the object.

Consider the following example.

class Employee:

 __count = 0;

def __init__(self):

 Employee.__count = Employee.__count+1

def display(self):

print("The number of employees",Employee.__count)

emp = Employee()

emp2 = Employee()

try:

print(emp.__count)

finally:

 emp.display()



CONTINUE IN NEXT UNIT