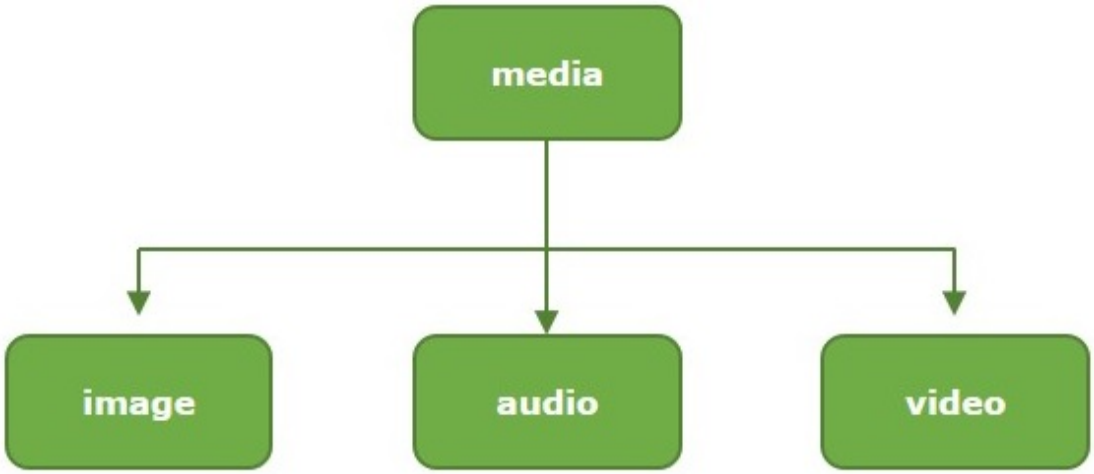
Python - Inheritance

What is Inheritance in Python?

Inheritance is one of the most important features of object-oriented programming languages like Python. It is used to inherit the properties and behaviours of one class to another. The class that inherits another class is called a **child class** and the class that gets inherited is called a **base class or parent class**.

If you have to design a new class whose most of the attributes are already well defined in an existing class, then why redefine them? Inheritance allows capabilities of existing class to be reused and if required extended to design a new class.

Inheritance comes into picture when a new class possesses 'IS A' relationship with an existing class. For example, Car IS a vehicle, Bus IS a vehicle, Bike IS also a vehicle. Here, Vehicle is the parent class, whereas car, bus and bike are the child classes.



Creating a Parent Class

The class whose attributes and methods are inherited is called as parent class. It is defined just like other classes i.e. using the class keyword.

Syntax

The syntax for creating a parent class is shown below –

```
class ParentClassName:
   {class body}
```

Creating a Child Class

Classes that inherit from base classes are declared similarly to their parent class, however, we need to provide the name of parent classes within the parentheses.

Syntax

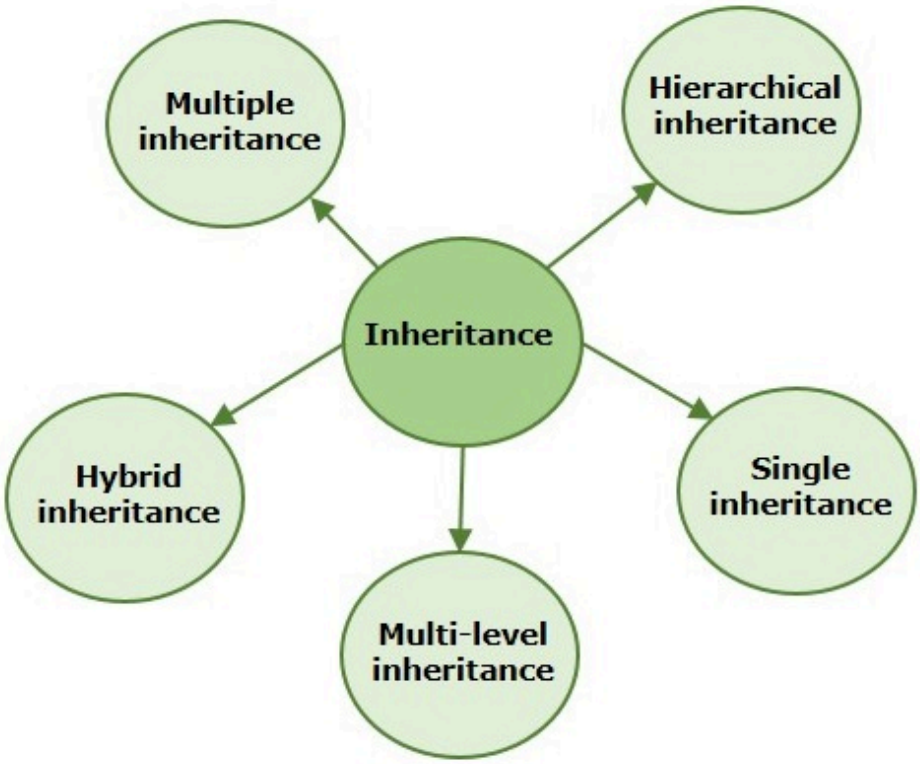
Following is the syntax of child class –

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   {sub class body}
```

Types of Inheritance

In Python, inheritance can be divided in five different categories –

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance



Python - Single Inheritance

This is the simplest form of inheritance where a child class inherits attributes and methods from only one parent class.

Example

The below example shows single inheritance concept in Python –

</> Open Compiler

```
# parent class
class Parent:
   def parentMethod(self):
      print ("Calling parent method")

# child class
class Child(Parent):
   def childMethod(self):
      print ("Calling child method")

# instance of child
c = Child()
# calling method of child class
```

```
c.childMethod()
# calling method of parent class
c.parentMethod()
```

On running the above code, it will print the following result –

```
Calling child method
Calling parent method
```

Python - Multiple Inheritance

Multiple inheritance in Python allows you to construct a class based on more than one parent classes. The Child class thus inherits the attributes and method from all parents. The child can override methods inherited from any parent.

Syntax

```
class parent1:
   #statements

class parent2:
   #statements

class child(parent1, parent2):
   #statements
```

Example

Python's standard library has a built-in divmod() function that returns a two-item tuple. First number is the division of two arguments, the second is the mod value of the two operands.

This example tries to emulate the divmod() function. We define two classes division and modulus, and then have a div_mod class that inherits them.

```
class division:
   def __init__(self, a,b):
      self.n=a
      self.d=b
   def divide(self):
      return self.n/self.d
```

```
class modulus:
   def __init__(self, a,b):
      self.n=a
      self.d=b
   def mod_divide(self):
      return self.n%self.d

class div_mod(division,modulus):
   def __init__(self, a,b):
      self.n=a
      self.d=b
   def div_and_mod(self):
      divval=division.divide(self)
      modval=modulus.mod_divide(self)
      return (divval, modval)
```

The child class has a new method div_and_mod() which internally calls the divide() and mod_divide() methods from its inherited classes to return the division and mod values.

```
x=div_mod(10,3)
print ("division:",x.divide())
print ("mod_division:",x.mod_divide())
print ("divmod:",x.div_and_mod())
```

Output

```
division: 3.3333333333333335
mod_division: 1
divmod: (3.3333333333333335, 1)
```

Method Resolution Order (MRO)

The term **method resolution order** is related to multiple inheritance in Python. In Python, inheritance may be spread over more than one levels. Let us say A is the parent of B, and B the parent for C. The class C can override the inherited method or its object may invoke it as defined in its parent. So, how does Python find the appropriate method to call.

Each Python has a mro() method that returns the hierarchical order that Python uses to resolve the method to be called. The resolution order is from bottom of inheritance order to top.

In our previous example, the div_mod class inherits division and modulus classes. So, the mro method returns the order as follows –

```
[<class '__main__.div_mod'>, <class '__main__.division'>, <class
'__main__.modulus'>, <class 'object'>]
```

Python - Multilevel Inheritance

In multilevel inheritance, a class is derived from another derived class. There exists multiple layers of inheritance. We can imagine it as a grandparent-parent-child relationship.

Example

In the following example, we are illustrating the working of multilevel inheritance.

</> Open Compiler

```
# parent class
class Universe:
   def universeMethod(self):
      print ("I am in the Universe")

# child class
class Earth(Universe):
   def earthMethod(self):
      print ("I am on Earth")

# another child class
class India(Earth):
   def indianMethod(self):
      print ("I am in India")

# creating instance
person = India()
# method calls
person.universeMethod()
person.earthMethod()
person.indianMethod()
```

When we execute the above code, it will produce the following result –

```
I am in the Universe
I am on Earth
I am in India
```

Python - Hierarchical Inheritance

This type of inheritance contains multiple derived classes that are inherited from a single base class. This is similar to the hierarchy within an organization.

Example

The following example illustrates hierarchical inheritance. Here, we have defined two child classes of **Manager class**.

</> Open Compiler

```
# parent class
class Manager:
   def managerMethod(self):
      print ("I am the Manager")

# child class
class Employee1(Manager):
   def employee1Method(self):
      print ("I am Employee one")

# second child class
class Employee2(Manager):
   def employee2Method(self):
      print ("I am Employee two")

# creating instances
emp1 = Employee1()
emp2 = Employee2()
# method calls
emp1.managerMethod()
emp1.employee1Method()
emp2.managerMethod()
emp2.employee2Method()
```

On executing the above program, you will get the following output –

```
I am the Manager
I am Employee one
I am the Manager
I am Employee two
```

Python - Hybrid Inheritance

Combination of two or more types of inheritance is called as Hybrid Inheritance. For instance, it could be a mix of single and multiple inheritance.

Example

In this example, we have combined single and multiple inheritance to form a hybrid inheritance of classes.

</> Open Compiler

```
# parent class
class CEO:
   def ceoMethod(self):
      print ("I am the CEO")

class Manager(CEO):
   def managerMethod(self):
      print ("I am the Manager")

class Employee1(Manager):
   def employee1Method(self):
      print ("I am Employee one")

class Employee2(Manager, CEO):
   def employee2Method(self):
      print ("I am Employee two")

# creating instances
emp = Employee2()
# method calls
emp.managerMethod()
emp.ceoMethod()
emp.employee2Method()
```

On running the above program, it will give the below result –

```
I am the Manager
I am the CEO
I am Employee two
```

The super() function

In Python, **super()** function allows you to access methods and attributes of the parent class from within a child class.

Example

In the following example, we create a parent class and access its constructor from a subclass using the super() function.

</> Open Compiler

```
# parent class
class ParentDemo:
   def __init__(self, msg):
      self.message = msg

   def showMessage(self):
      print(self.message)

# child class
class ChildDemo(ParentDemo):
   def __init__(self, msg):
      # use of super function
      super().__init__(msg)

# creating instance
obj = ChildDemo("Welcome to Tutorialspoint!!")
obj.showMessage()
```

On executing, the above program will give the following result –

```
Welcome to Tutorialspoint!!
```