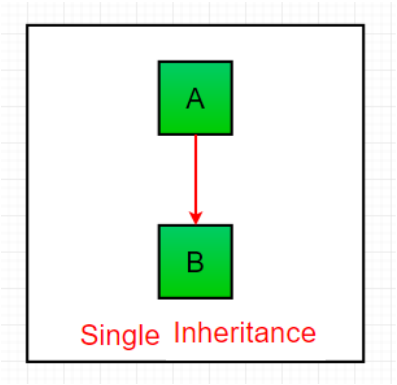
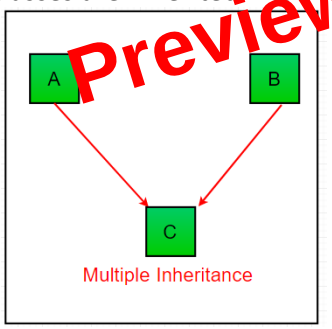


# Python program to demonstrate 	
# single inheritance 	
# Base class 	
class Parent:
def func1(self):
print("This function is in parent class.")
# Derived class 	
class Child(Parent):
def func2(self):
print("This function is in child class.")
 	
# Driver's code 	
object = Child()
object.func1()
object.func2()
	Output:
	
	This function is in parent class.
	This function is in child class.

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



Syntax:

```
class A:
   # variable of class A
   # functions of class A

class B:
   # variable of class A
   # functions of class A

class C(A, B):
   # class C inheriting property of both class A and B
   # add more properties to class C
```

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

Real Life Example of method overriding

```
class Bank:
    def getroi(self):
        return 10;
class SBI(Bank):
    def getroi(self):
        return 7;

class ICICI(Bank):
    def getroi(self):
        return 8;
b1 = Bank()
b2 = SBI()
b3 = ICICI()
print("Bank Rate of interest:",b1.getroi());
print("SBI Rate of interest:",b2.getroi());
print("ICICI Rate of interest:",b3.getroi());
```

Output:

```
Bank Rate of interest: 10
SBI Rate of interest: 7
ICICI Rate of interest: 8
```


==========

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.

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)
```

Output:

```
The number of employees 2
AttributeError: 'Employee' object has no attribute '__count'
```

The output is:

```
  File "C:/Users/User/.spyder-py3/temp.py", line 10, in <module>
    b=Bus()

TypeError: Can't instantiate abstract class Bus with abstract methods getNoOfWheels
```

If we remove the @abstractmethod decorator, then the method becomes a normal method and the child class may or may not give implementation to it.

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

   def getNoOfWheels(Self):
      pass

class Bus(Vehicle):
  pass

b=Bus()
```

The output is: It is not giving any output, but it is not giving any error.

```
In [9]: runfile('C:/Users/User/.spyder-py3/temp.py'

In [10]:
```

==========

Points to be remembered on Abstract Methods and Classes in Python

1.If a class containing one abstract method and if we are extending ABC class then instantiation id not possible;

for Abstract class with an abstract method instantiation (creating an object) is not possible**.**

Consider the below example which is a concrete class that does not contain an abstract method and an abstract class, and hence we can perform instantiation/create an object

```
class Test:
   pass

t=Test()
```

==========

The below example contains an abstract class but does not include an abstract method, and therefore, we can perform instantiation/can create an object.

An abstract class can contain a zero number of abstract methods

```
from abc import *
class Test(ABC):
   pass

t=Test()
```

==========

But, the following example contains an abstract method and does not include an abstract class, and hence we can perform instantiation.

```
from abc import *
```

The Purpose of Interface

An abstract class containing only abstract method acts as requirement specification, anyone can provide an implementation in their way, and hence a single interface contain multiple applications in their form.

The following example demonstrates the purpose of the interface.

```
from abc import *
class CollegeAutomation(ABC):    ##requirement apecification
   @abstractmethod
   def method1(self):pass
   @abstractmethod
   def method2(self):pass
   @abstractmethod
   def method3(self):pass

class AaruSoftImpl(CollegeAutomation):
   def method1(self):
      print("Method1 implementation")

   def method2(self):
      print("Method2 implementation")

   def method3(self):
      print("Method3 implementation")

d=AaruSoftImpl()  ##creating an object and calling method1 method2 and method3
d.method1()
d.method2()
d.method3()
```


The output is:

```
In [1]: runfile('C:/Users/Desktop/untitled0.spyder-py3',
Method1 implementation
Method2 implementation
Method3 implementation
```

The difference between Interface vs Abstract class vs Concrete class

Interface	Abstract class	Concrete class
If we do not know anything about implementation and we want to know requirement specification, then we should go for Interface	If we are talking about implementation but not wholly, then we should go for abstract class	If we are talking about complete implementation and ready to provide service, then we should go for a concrete class

The following example demonstrates the difference between interface, abstract class, and concrete class.

```
from abc import *
class CollegeAutomation(ABC):  ##Interfcae, because it contains only abstract methods
   @abstractmethod
   def method1(self):pass
   @abstractmethod
```