What is the output of the following code snippet?

class Customer:  
    def \_\_init\_\_(self,id1):  
        self.id = id1  
  
c1=Customer(200)  
print(c1.id1)

200

Error

None

**Answer - Error**

**//-------------------------------------------------------------------------------**

What is the output of the following code snippet?

class Customer:  
    def \_\_init\_\_(id,self,age):  
        id.self=self  
        id.age=age  
  
c1=Customer(100,20)  
print(c1.self)

100

20

Error

**Answer -   
//--------------------------------------------------------------------**

* OOP is a style of programming which allows us to club data and behavior together.
* This is more suited for coding real life scenarios.
* Objects are real world entities
* Class is just a classification. It is just a concept.
* Class is a description of attributes and behavior that objects of that classification should possess.
* Attributes are created in a special function called \_\_init\_\_ and behaviors are created using functions called methods.
* Objects can be created using ClassName() or using object literals for some of the built in classes
* Attributes are created using reference\_variable.attribute\_name = value syntax.
* Behavior is created by defining a function inside the class having a special parameter called self

**//-----------------------------------------------------------**

)What is the output of the below code snippet?

class Example:  
    def \_\_init\_\_(self,num):  
        self.num=num  
  
    def set\_num(self,num):  
        self.num=num  
  
    def get\_num(self):  
        return self.num  
obj=Example(10)  
print(obj.get\_num())  
obj.set\_num(15)  
print(obj.get\_num())

a. 10

   10

b. 10

   15

c. Error: constructor cannot accept a value

a

b

c   ANS: option b

2)What is the output of the following code snippet?

class Customer:  
    def \_\_init\_\_(self):  
        cust\_id = 100  
  
c1=Customer()  
print(c1.cust\_id)

100

Error

None

Ans : Error

3)class Customer:  
    def \_\_init\_\_(self,id1):  
        self.id = id1  
  
c1=Customer(200)  
print(c1.id1)

200

Error

None

Ans: error

**//-----------------------------------------------------------------------------**

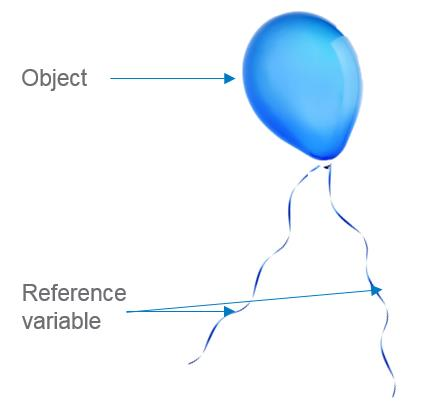
[9:49 AM] Ashutosh (Unverified)

# Multiple References                       Can one balloon have multiple ribbons?



# Multiple References - Try outProblem Statement

Just like a balloon can have multiple ribbons, an object can also have multiple reference variables. Both the references are referring to the same object. When you assign an already created object to a variable, a new object is not created.

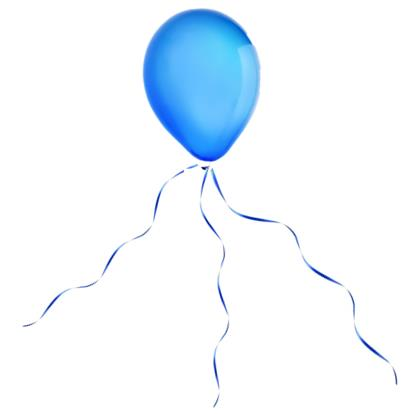


class Mobile:  
    def \_\_init\_\_(self, price, brand):  
        print ("Inside constructor")  
        self.price = price  
        self.brand = brand

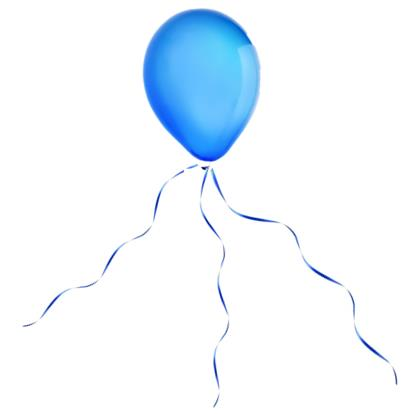
mob1=Mobile(1000, "Apple")  
mob2=mob1  
print ("Id of object referred by mob1 reference variable is :", id(mob1))  
print ("Id of object referred by mob2 reference variable is :", id(mob2))  
#mob1 and mob2 are reference variables to the same object

[9:52 AM] Ashutosh (Unverified)

# Multiple References - Updating Object                                 Let's say a balloon has three ribbons connecting it. If I change the color of the balloon tied to ribbon 1, what will be the color of the balloon tied to ribbons 2 and 3?



# Multiple References - Updating Object - Try out                          Just like the balloon with multiple ribbons, if we change the attribute of an object through one reference variable, it immediately reflects in other reference variable as there is only one balloon ultimately!



class Mobile:  
    def \_\_init\_\_(self, price, brand):  
        self.price = price  
        self.brand = brand

mob1=Mobile(1000, "Apple")  
print("Price of mobile 1 :", mob1.price)

mob2=mob1  
mob2.price=3000

print("Price of mobile 1 :", mob1.price)  
print("Price of mobile 2 :", mob2.price)

//------------------------------------------------------------------------------------------------------

Reference variable - Summary                    Reference variables hold the objects

1. We can create objects without reference variable as well
2. An object can have multiple reference variables
3. Assigning a new reference variable to an existing object does not create a new object

//-------------------------------------------------------------------

Q1 of 3outlined\_flag

Analyze the below code snippet and identify how many objects and reference variables will be there at the end of line 9.

class Table:                 #Line1  
    def \_\_init\_\_(self):      #Line2  
        self.no\_of\_legs=4    #Line3  
        self.glass\_top=None  #Line4  
        self.wooden\_top=None #Line5  
dining\_table=Table()         #Line6  
back\_table=Table()           #Line7  
front\_table=back\_table       #Line8  
back\_table=dining\_table      #Line9

a) 2 Objects, 4 Reference Variables

b) 2 Objects, 3 Reference Variables

c) 4 Objects, 2 Reference Variables

d) 4 Objects, 3 Reference Variables

a

b

c

d

Q2 of 3outlined\_flag

Analyze the below code snippet and identify how many reference variables refer to object created in Line 7 at the end of Line 11?

class Table:                  #Line1  
    def \_\_init\_\_(self):       #Line2  
        self.no\_of\_legs=4     #Line3  
        self.glass\_top=None   #Line4  
        self.wooden\_top=None  #Line5  
dining\_table=Table()          #Line6  
back\_table=Table()            #Line7  
front\_table=back\_table        #Line8  
back\_table=dining\_table       #Line9  
dining\_table=front\_table      #Line10  
front\_table=back\_table        #Line11

0

1

2

Q3 of 3outlined\_flag

Consider the below code snippet:

class Table:                         #Line1  
    def \_\_init\_\_(self):              #Line2  
        self.no\_of\_legs=4            #Line3  
        self.glass\_top=None          #Line4  
        self.wooden\_top=None         #Line5  
    def identify\_rate(self):         #Line6  
        if(self.glass\_top==True):    #Line7  
            rate=20000               #Line8  
        elif(self.wooden\_top==True): #Line9  
            rate=30000               #Line10  
        else:                        #Line11  
            rate=0                   #Line12  
        return rate                  #Line13  
dining\_table=Table()                 #Line14

//-------------------------------------------------------------------------------------------

[9:55 AM] Ashutosh (Unverified)

Q1 of 3outlined\_flag

Analyze the below code snippet and identify how many objects and reference variables will be there at the end of line 9.

class Table:                 #Line1  
    def \_\_init\_\_(self):      #Line2  
        self.no\_of\_legs=4    #Line3  
        self.glass\_top=None  #Line4  
        self.wooden\_top=None #Line5  
dining\_table=Table()         #Line6  
back\_table=Table()           #Line7  
front\_table=back\_table       #Line8  
back\_table=dining\_table      #Line9

a) 2 Objects, 4 Reference Variables

b) 2 Objects, 3 Reference Variables

c) 4 Objects, 2 Reference Variables

d) 4 Objects, 3 Reference Variables

a

b

c

d

Q2 of 3outlined\_flag

Analyze the below code snippet and identify how many reference variables refer to object created in Line 7 at the end of Line 11?

class Table:                  #Line1  
    def \_\_init\_\_(self):       #Line2  
        self.no\_of\_legs=4     #Line3  
        self.glass\_top=None   #Line4  
        self.wooden\_top=None  #Line5  
dining\_table=Table()          #Line6  
back\_table=Table()            #Line7  
front\_table=back\_table        #Line8  
back\_table=dining\_table       #Line9  
dining\_table=front\_table      #Line10  
front\_table=back\_table        #Line11

0

1

2

Q3 of 3outlined\_flag

Consider the below code snippet:

class Table:                         #Line1  
    def \_\_init\_\_(self):              #Line2  
        self.no\_of\_legs=4            #Line3  
        self.glass\_top=None          #Line4  
        self.wooden\_top=None         #Line5  
    def identify\_rate(self):         #Line6  
        if(self.glass\_top==True):    #Line7  
            rate=20000               #Line8  
        elif(self.wooden\_top==True): #Line9  
            rate=30000               #Line10  
        else:                        #Line11  
            rate=0                   #Line12  
        return rate                  #Line13  
dining\_table=Table()                 #Line14

Which among the following statements placed after line 14 will result in an error?

dining\_table.no\_of\_legs=6

glass\_top=True

print (dining\_table.rate)

[9:56 AM] Ashutosh (Unverified)

Problem Statement

A vehicle is identified by its mileage (in kms per litre) and fuel left (in litres) in the vehicle. From the fuel left, 5 litres will always be considered as reserve fuel. At any point of time, the driver of the vehicle may want to know:

* the maximum distance that can be covered without using the reserve fuel
* how many kms he/she has already travelled based on the initial fuel the vehicle had

Identify the class name and attributes so as to represent a vehicle from the information given.

* \_\_init\_\_()
* Vehicle
* Car
* identify\_disctance\_that\_can\_be\_travelled()
* mileage
* fuel\_left
* identify\_distance\_travelled(initial\_fuel)

Write a Python program to implement the class chosen with its attributes and methods based on the requirements given below:

**identify\_distance\_that\_can\_be\_travelled():** Return the distance that can be travelled by the vehicle without using the reserve fuel. If the fuel left is less than or equal to reserve fuel, the method should return 0.

**identify\_distance\_travelled(initial\_fuel):** Return the distance so far travelled by the vehicle based on the initial fuel,fuel left and mileage.

Assume that initial fuel is always greater than fuel left.Represent a vehicle and test your program by initializing the instance variables and invoking the appropriate methods.

//----------------------------------------------------------------------------------------------

# \_str\_\_ Tryout                   or a more readable output when printing an object we can use the inbuilt special \_\_str\_\_ method. This method MUST return a string and this string will be used when the object is printed. This is useful in debugging as we can print the values of the attributes.

Code -

# class Shoe:

    def \_\_init\_\_(self,price,material):

        self.price=price

        self.material=material

    def \_\_str\_\_(self):

        print('str is calling')

        return "Shoe with price: " + str(self.price) + " and material: " + self.material

 #Create an Instance

s1=Shoe(1000,'Canvas')

print(s1)

Output - Shoe with price: 1000 and material: Canvas

//-------------------------------------------------------------------------

# Private Data Access -

# Problem Statement-

We can put a lock on that data by adding a double underscore in front of it, as shown in below code.

Adding a double underscore makes the attribute a private attribute. Private attributes are those which are accessible only inside the class. This method of restricting access to our data is called **encapsulation.**

**Code -**

class Customer:

    def \_\_init\_\_(self, cust\_id, name, age, wallet\_balance):

        self.cust\_id = cust\_id

        self.name = name

        self.age = age

        self.\_\_wallet\_balance = wallet\_balance //When we put a double underscore in front of the attribute name,to make it private

    def update\_balance(self,amount):

        if amount < 1000 and amount > 0:

            self.wallet\_balance += amount

    def show\_balance(self):

            print("The balance is ",self.wallet\_balance)

 c1=Customer(100, "Gopal", 24, 1000)

c1.update\_balance(500)

c1.show\_balance()

print(c1.\_\_wallet\_balance)

Output - The balance is 1500

//--------------------------------------------------------------------------

# Private Data Update - Caution !    Problem Statement

If we try to assign a value to a private variable, we end up creating a new attribute in python. Thus this code does not give an error, but it is logically flawed and does not produce the intended result.  class Customer:

    def \_\_init\_\_(self, cust\_id, name, age, wallet\_balance):

        self.cust\_id = cust\_id

        self.name = name

        self.age = age

        self.\_\_wallet\_balance = wallet\_balance

    def update\_balance(self, amount):

        if amount < 1000 and amount > 0:

            self.\_\_wallet\_balance += amount

    def show\_balance(self):

        print ("The balance is ",self.\_\_wallet\_balance)

c1=Customer(100, "Gopal", 24, 1000)

c1.\_\_wallet\_balance = 10000000000

c1.show\_balance()

//--------------------------------------------------------------------------

# Getters & Setters -

# Problem Statement -

To have a error free way of accessing and updating private variables, we create specific methods for this.

* The methods which are meant to set a value to a private variable are called **setter methods**
* The methods meant to access private variable values are called **getter methods**

# Accessing Private Variables (Getter ) -

# Code -

# class Person:

    def \_\_init\_\_(self,name,age):

        self.\_\_name=name

        self.\_\_age=age

    def get\_name(self):

        return self.\_\_name

    def get\_age(self):

        return self.\_\_age

p1=Person('John',20)

print(p1.get\_name())

//----------------------------------------------------------------------

**Inheritance -   
  
code -**

class A:

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

print('Init:A')

def m1(self):

print("hello m1 in A")

class B(A):

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

super().\_\_init\_\_()

print("Init:B")

def m2(self):

print("hello B in m2")

b1 = B()

b1.m2()

b1.m1()

**Output** -

Init:A

Init:B

hello B in m2

hello m1 in A

//-------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------

Inheritance - Introduction

Types of phones

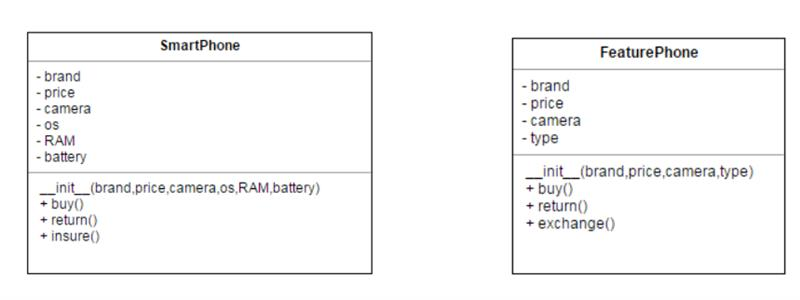
Let us say that the online shopping app wants to sell different types of phones:

* Feature phones and Smartphones



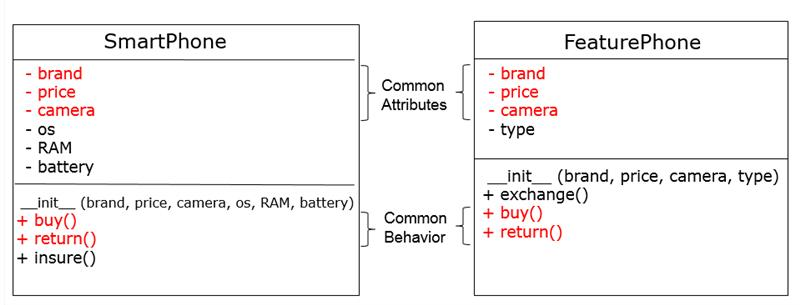
Class diagram

The below are the class diagrams for both the classes:

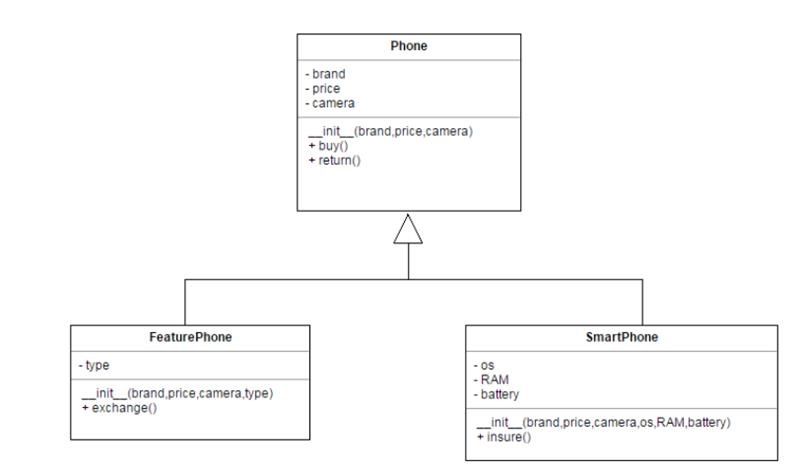


Common attributes & behaviors

We can see that both the class have a lot in common. This is because they both are ultimately phones and each is just a special type of phone.



In our example, FeaturePhone is inheriting the Phone and SmartPhone is inheriting the Phone class (SmartPhone "is-A" phone, FeaturePhone "is-A" phone). So Phone is the parent class and FeaturePhone and SmartPhone are derived classes.



What are the advantages of inheritance?

There are three main advantages of inheritance:

* We can keep common properties in a single place. Thus any changes needs to be made need not be repeated.
* Inheritance encourages code reuse thus saving us time.
* If we want to add a new type of phone later on, we can simply inherit the Phone class instead of writing it from scratch.

Inheritance - Independent Classes - Try out

Problem Statement

Let us look at inheritance in code. For now we will create the Phone class with necessary attributes and methods. We will create FeaturePhone and SmartPhone classes without any attributes or methods now. We will create them later.

class Phone:

   def \_\_init\_\_(self, price, brand, camera):

       self.price = price

       self.brand = brand

       self.camera = camera

   def buy(self):

       print ("Buying a phone")

   def return\_phone(self):

       print ("Returning a phone")

class FeaturePhone:

   pass

class SmartPhone:

   pass

Phone(10000,"Apple","13px").buy()

//--------------------------------------------------------

Inheritance - Connecting Classes - Try out

Problem Statement

To create an inheritance relationship between the classes, mention the name of the parent class in brackets as shown:

**Code -**

class Phone:

   def \_\_init\_\_(self, price, brand, camera):

       self.price = price

       self.brand = brand

       self.camera = camera

   def buy(self):

       print ("Buying a phone")

   def return\_phone(self):

       print ("Returning a phone")

class FeaturePhone(Phone):

   pass

class SmartPhone(Phone):

   pass

featurephone= FeaturePhone(10000,"Apple","13px")

featurephone.buy()

**Output -**

**Buying a phone**

 //-------------------------------------------

What gets inherited?

Inheriting Constructor - Try out 1

Since the SmartPhone class is inheriting the Phone class, the SmartPhone class inherits the constructor of the Phone class.

class Phone:

   def \_\_init\_\_(self, price, brand, camera):

       print ("Inside phone constructor")

       self.price = price

       self.brand = brand

       self.camera = camera

   def buy(self):

       print ("Buying a phone")

   def return\_phone(self):

       print ("Returning a phone")

class FeaturePhone(Phone):

   pass

class SmartPhone(Phone):

   pass

s=SmartPhone(20000, "Apple", 13)

 //---------------------------------------------------------------------

Problem Statement

Consider the below code. Since the SmartPhone class has its own constructor, the Phone class constructor is not inherited. Hence the attributes in the Phone class are also not inherited.

class Phone:

   def \_\_init\_\_(self, price, brand, camera):

       print ("Inside phone constructor")

       self.\_\_price = price

       self.brand = brand

       self.camera = camera

   def buy(self):

       print ("Buying a phone")

   def return\_phone(self):

       print ("Returning a phone")

class FeaturePhone(Phone):

   pass

class SmartPhone(Phone):

   def \_\_init\_\_(self, os, ram):

       self.os = os

       self.ram = ram

       print ("Inside SmartPhone constructor")

   def buy(self):

       print ("Buying a SmartPhone")

s=SmartPhone("Android", 2)

print(s.os)

print(s.brand)

//------------------------------------------------------------------

Problem Statement

A child class cannot directly access the private attributes of the parent class.

class Phone:

   def \_\_init\_\_(self, price, brand, camera):

       print ("Inside phone constructor")

       self.\_\_price = price

       self.brand = brand

       self.camera = camera

   def buy(self):

       print ("Buying a phone")

   def return\_phone(self):

       print ("Returning a phone")

class FeaturePhone(Phone):

   pass

class SmartPhone(Phone):

   def check(self):

       print(self.\_\_price)

s=SmartPhone(20000, "Apple", 13)

s.check()

 //--------------------------------------------------------

Inheriting Methods - Try out

Problem Statement

Apart from attributes, the child class inherits the methods of the parent class as shown:

**Code -**

class Phone:

   def \_\_init\_\_(self, price, brand, camera):

       print ("Inside phone constructor")

       self.\_\_price = price

       self.brand = brand

       self.camera = camera

   def buy(self):

       print ("Buying a phone")

   def return\_phone(self):

       print ("Returning a phone")

class FeaturePhone(Phone):

   pass

class SmartPhone(Phone):

   pass

s=SmartPhone(20000, "Apple", 13)

s.buy()

**Output -**

**Inside phone constructor**

**Inside smartphone constructor**

**Buying a phone**

 //-----------------------------------------------------------------------

Method Overriding - Try out

Problem Statement

Sometimes a child may not want to use what it has inherited from the parent. The same holds true for OOP as well. If the child class does not want to use a method inherited from the parent class then it may create its own method with the same name.

When the child has a method with the same name as that of the parent, it is said to override the parent’s method. This is called as **Method Overriding**. Method overriding is also called as **Polymorphism**.

Q ) Method Overriding -

class Parent:

   def purchaseBike(self):

       print("Parent wants to purchase Hero Bike")

   def marry(self):

       print("Parent decided marry for our child with ABC")

   def property(self):

       print("Car+Gold+Money")

class Child(Parent):

   def purchaseBike(self):

       print("Child wants to purchase R1-5 Bike")

   def marry(self):

       print("Child decided marry with PQR")

c=Child()

c.property()

c.purchaseBike()

**Output** -

**Car+Gold+Money**

**Child wants to purchase R1-5** Bike

//-------------------

 Q ) - Method Overriding -

class Sim:

   def network(self):

       print("default network!!")

class Jio(Sim):

   def network(self):

       print("jio networl")

class BSNL(Sim):

   def network(self):

       print("BSNL network")

class Idea(Sim):

   def network(self):

       print("Idea network")

s1=Jio()

s1.network()

s2=BSNL()

s2.network()

s3=Idea()

s3.network()

**Output -**

**Jio network**

**BSNL network**

**default network!!**

**Idea network**

**default network!!**

 //----------

Q ) - Method Overriding -

class Phone:

   def \_\_init\_\_(self, price, brand, camera):

       print ("Inside phone constructor")

       self.\_\_price = price

       self.brand = brand

       self.camera = camera

   def buy(self):

       print ("Buying a phone")

   def return\_phone(self):

       print ("Returning a phone")

class FeaturePhone(Phone):

   pass

class SmartPhone(Phone):

   def buy(self):

       print ("Buying a smartphone")

s=SmartPhone(20000, "Apple", 13)

s.buy()

**Output -**

**Inside phone constructor**

**Buying a smartphone**

//---------------------------------------------------------------------------------