PYTHON

From Simple to Complex

With

Examples

AYESHA NOREEN

Bachelor's in Software Engineering,

Master's in Computer Science

from COMSATS University, Islamabad

NOTE!!!

In these notes Screenshots of practice examples and coding are added. The code files are also available in code folder that contain .ipynb files that are created on Jupyter notebook.

Chapter18
OOP in Python

OOP is abbreviation of Object Oriented Programming. In OOP we use classes and objects.

Class:

Class is a blueprint in which we decide object contain which type of properties, functionalities etc. It is useful for designing real life programs. It is convension to use 1st capital letter of class name we can use small letter but using of capital letter is a good convention.Use of capital letter also make difference between pre-defined and user-defined classes.

Four main pillars of OOP

Main pillars of OOP are: (A PIE)

Abstraction

Polymorphism

Inheritance

Encapsulation

We will discuss each concept in next slides.

• __init__() method

__init__() method called every time when object of a class is made and execute

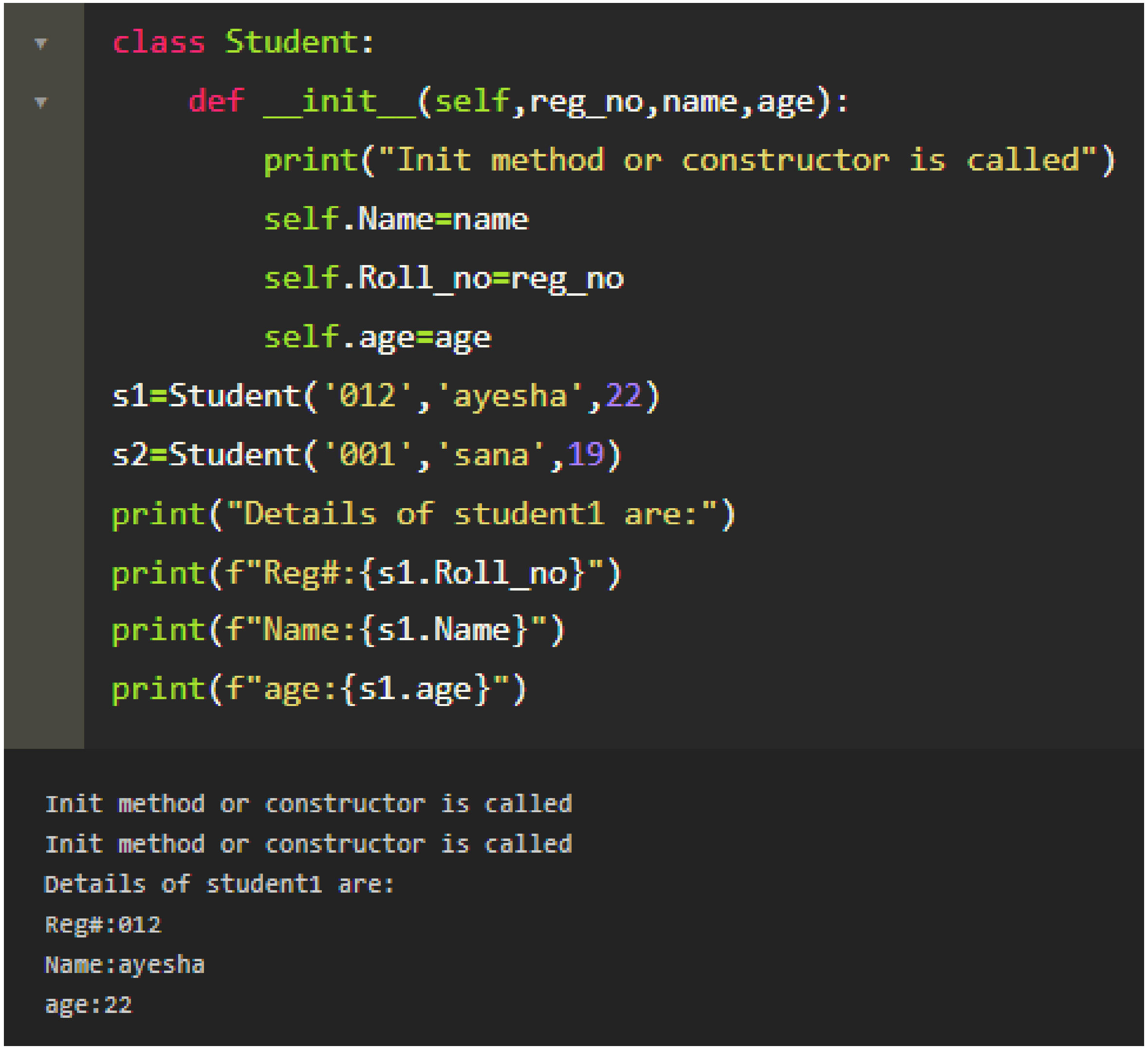
• Self keyword

Self keyword is used as first parameter of init method when an object of a class is formed a copy of that object is send to self than we can access all attributes by using this self keyword. we can write any name instead of self but it is good convention to use self.

• Attributes

In init method all other parameter except self are attributes.

Example

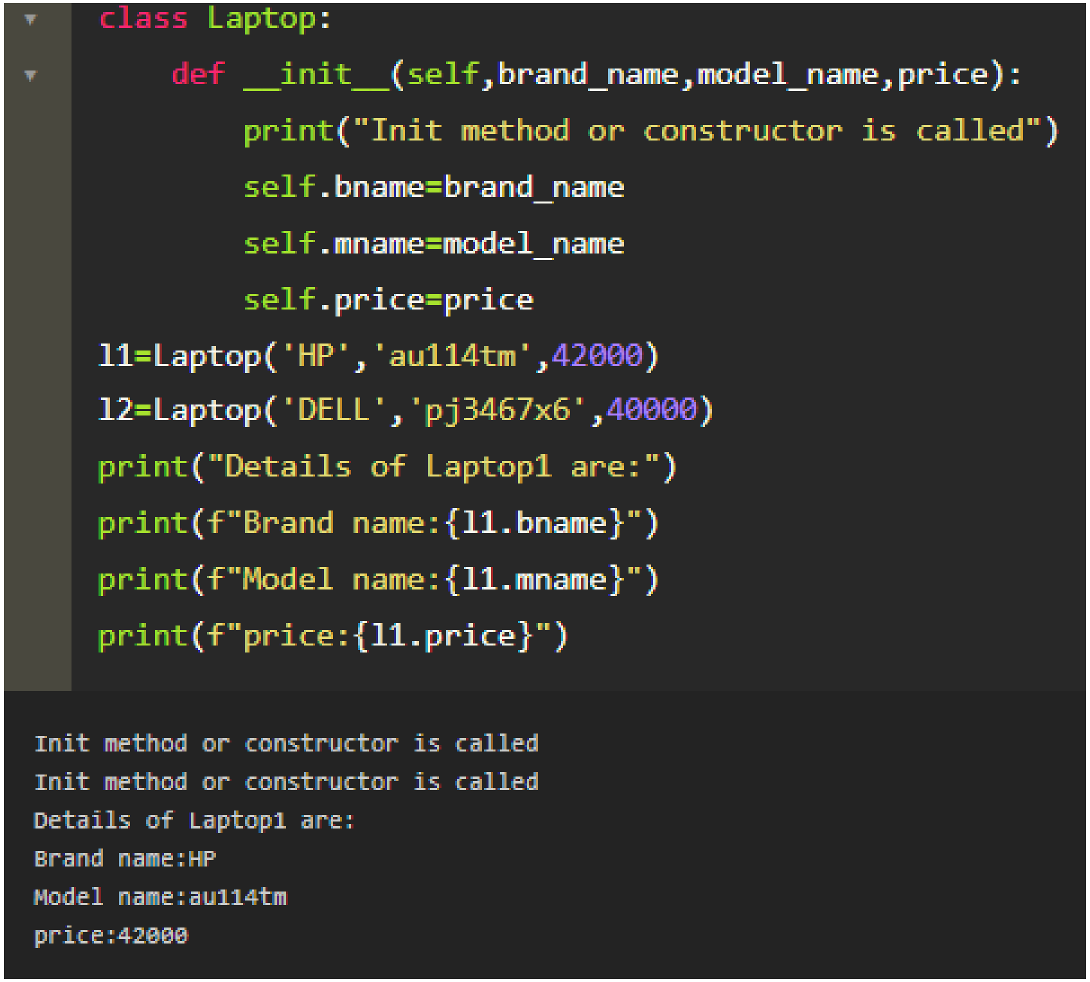


```python
class Student:
    def __init__(self,reg_no,name,age):
        print("Init method or constructor is called")
        self.Name=name
        self.Roll_no=reg_no
        self.age=age
s1=Student('012','ayesha',22)
s2=Student('001','sana',19)
print("Details of student1 are:")
print(f"Reg#:{s1.Roll_no}")
print(f"Name:{s1.Name}")
print(f"age:{s1.age}")
```

```
Init method or constructor is called
Init method or constructor is called
Details of student1 are:
Reg#:012
Name:ayesha
age:22
```

TODO Task

Define a class laptop which shows the brand name, model name and price of different laptops.

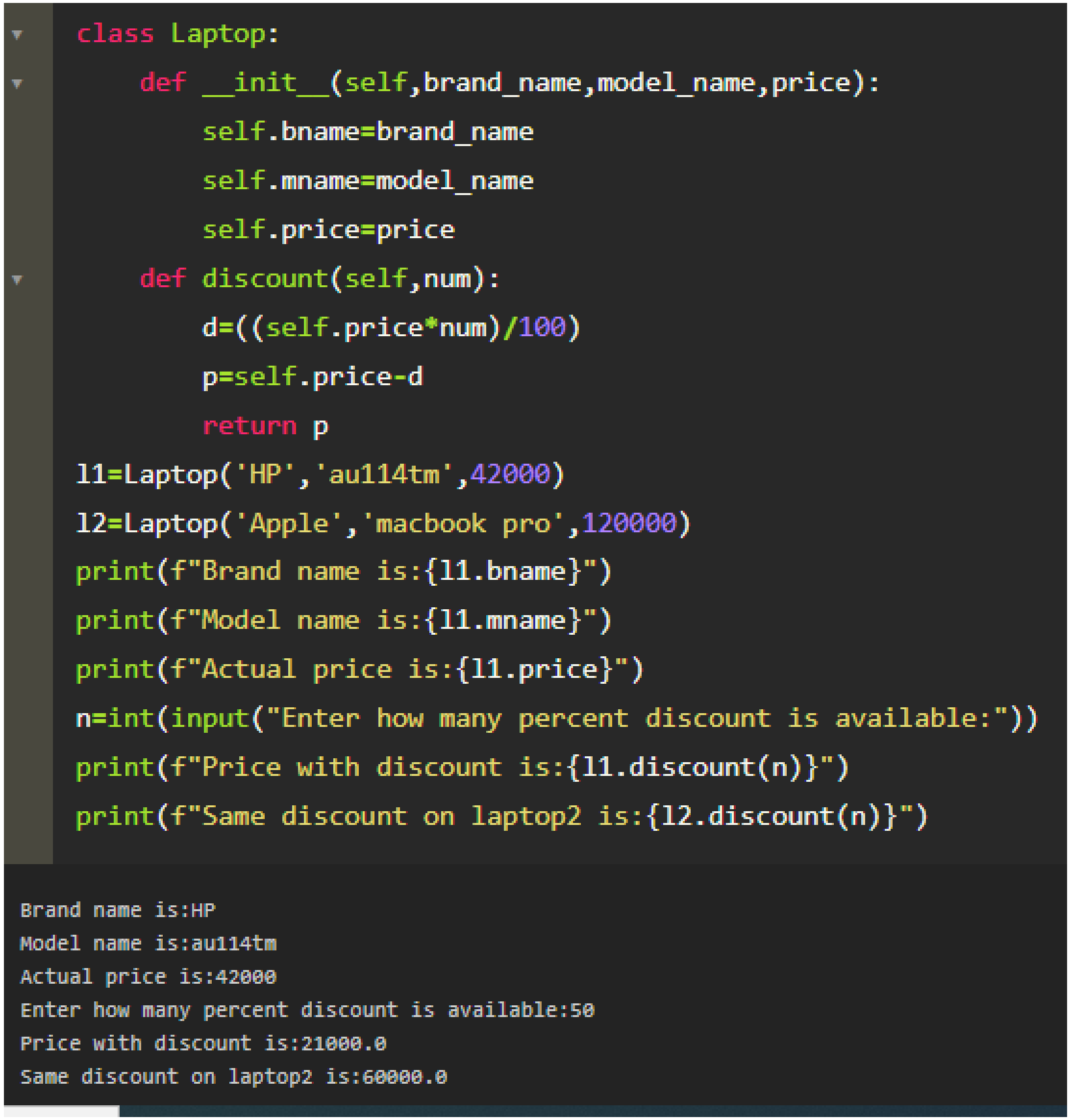


```
class Laptop:
    def __init__(self,brand_name,model_name,price):
        print("Init method or constructor is called")
        self.bname=brand_name
        self.mname=model_name
        self.price=price
l1=Laptop('HP','au114tm',42000)
l2=Laptop('DELL','pj3467x6',40000)
print("Details of Laptop1 are:")
print(f"Brand name:{l1.bname}")
print(f"Model name:{l1.mname}")
print(f"price:{l1.price}")
```

```
Init method or constructor is called
Init method or constructor is called
Details of Laptop1 are:
Brand name:HP
Model name:au114tm
price:42000
```

TODO Task

Make a discount method which calculate discount on price according to given number.

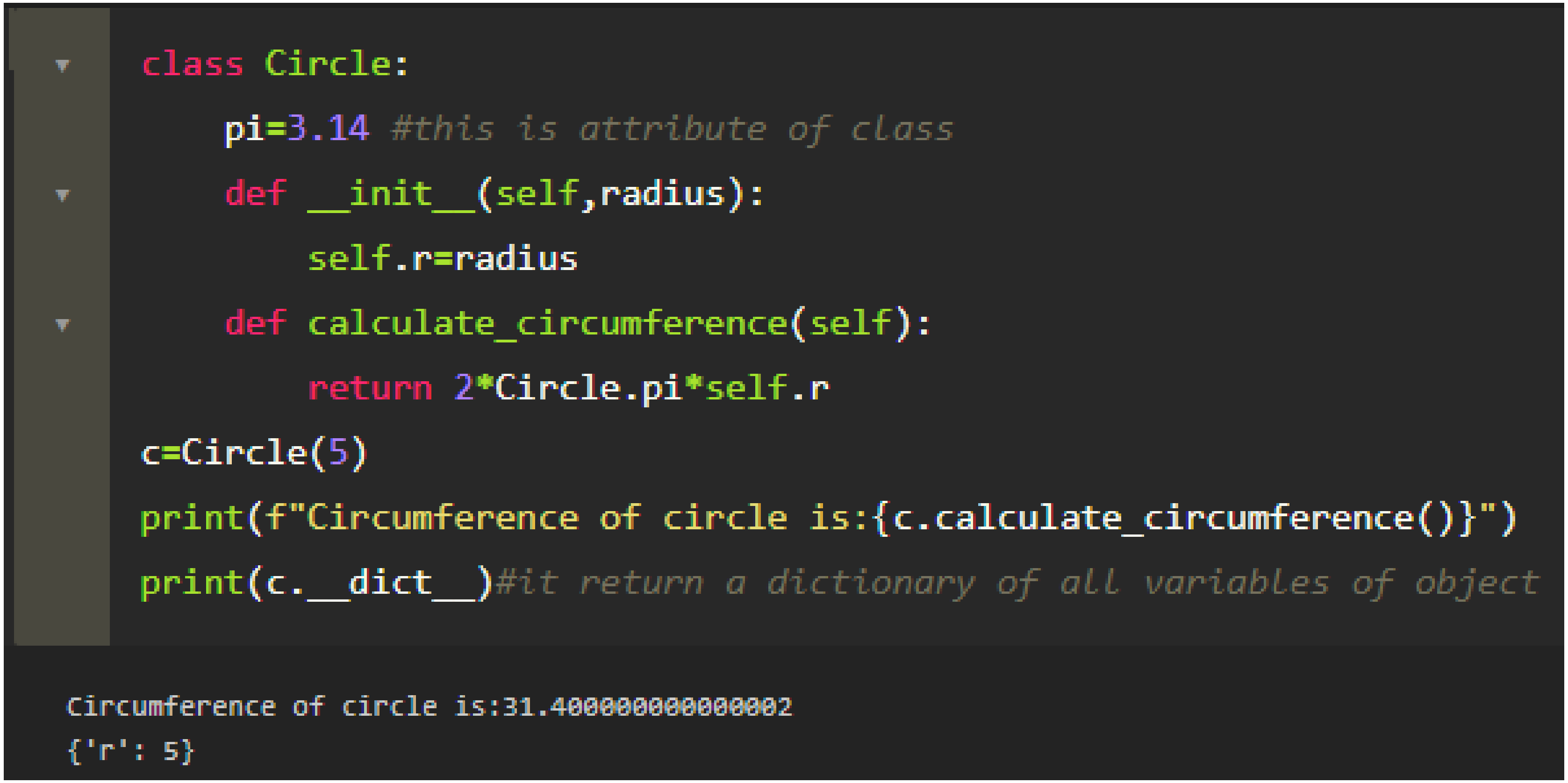


```python
class Laptop:
    def __init__(self,brand_name,model_name,price):
        self.bname=brand_name
        self.mname=model_name
        self.price=price
    def discount(self,num):
        d=((self.price*num)/100)
        p=self.price-d
        return p
l1=Laptop('HP','au114tm',42000)
l2=Laptop('Apple','macbook pro',120000)
print(f"Brand name is:{l1.bname}")
print(f"Model name is:{l1.mname}")
print(f"Actual price is:{l1.price}")
n=int(input("Enter how many percent discount is available:"))
print(f"Price with discount is:{l1.discount(n)}")
print(f"Same discount on laptop2 is:{l2.discount(n)}")
```

```
Brand name is:HP
Model name is:au114tm
Actual price is:42000
Enter how many percent discount is available:50
Price with discount is:21000.0
Same discount on laptop2 is:60000.0
```

• Attributes of a class

Attributes of a class are declare in class same as simple variable declaration but simple variables are accessed directly and the attributes of class are accessed by class name.

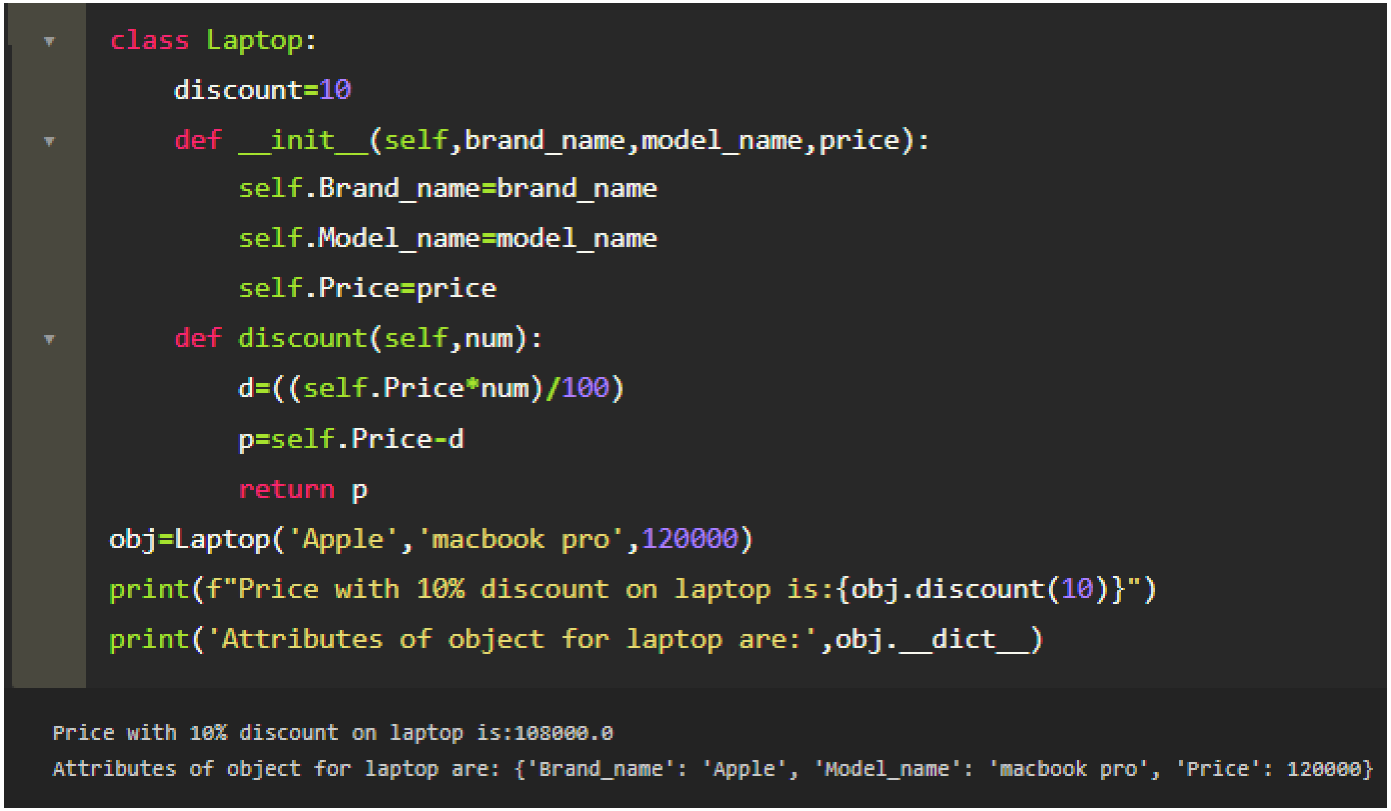


```
class Circle:
    pi=3.14 #this is attribute of class
    def __init__(self,radius):
        self.r=radius
    def calculate_circumference(self):
        return 2*Circle.pi*self.r
c=Circle(5)
print(f"Circumference of circle is:{c.calculate_circumference()}")
print(c.__dict__)#it return a dictionary of all variables of object
```

```
Circumference of circle is:31.400000000000002
{'r': 5}
```

- # Object.__dict__method

It is used to print a dictionary having all attributes of an object as keys and values of all attributes.

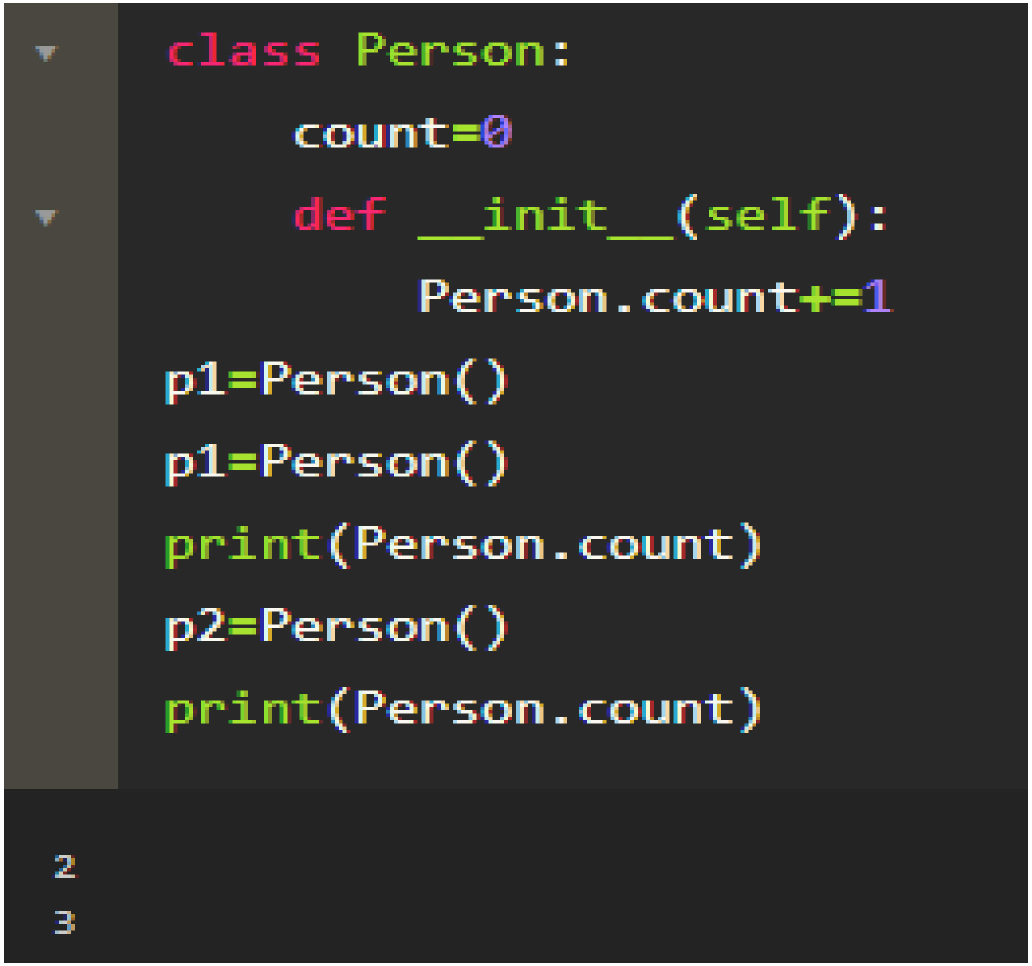


```
class Laptop:
    discount=10
    def __init__(self,brand_name,model_name,price):
        self.Brand_name=brand_name
        self.Model_name=model_name
        self.Price=price
    def discount(self,num):
        d=((self.Price*num)/100)
        p=self.Price-d
        return p
obj=Laptop('Apple','macbook pro',120000)
print(f"Price with 10% discount on laptop is:{obj.discount(10)}")
print('Attributes of object for laptop are:',obj.__dict__)
```

```
Price with 10% discount on laptop is:108000.0
Attributes of object for laptop are: {'Brand_name': 'Apple', 'Model_name': 'macbook pro', 'Price': 120000}
```

- ## TODO Task

Define a class which calculate how many objects a class contain.

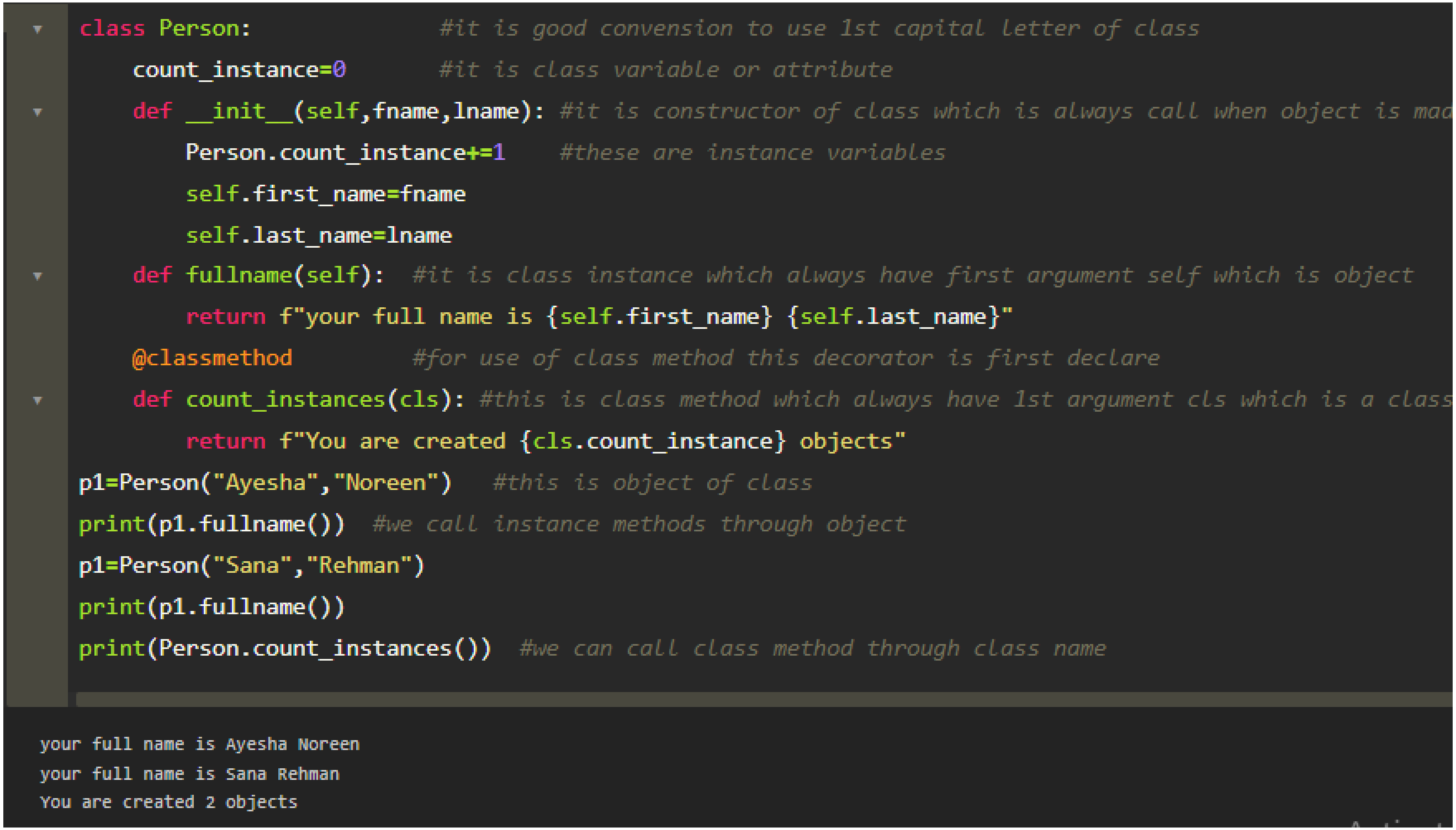


```
class Person:
    count=0
    def __init__(self):
        Person.count+=1
p1=Person()
p1=Person()
print(Person.count)
p2=Person()
print(Person.count)
```

```
2
3
```

• @Class method

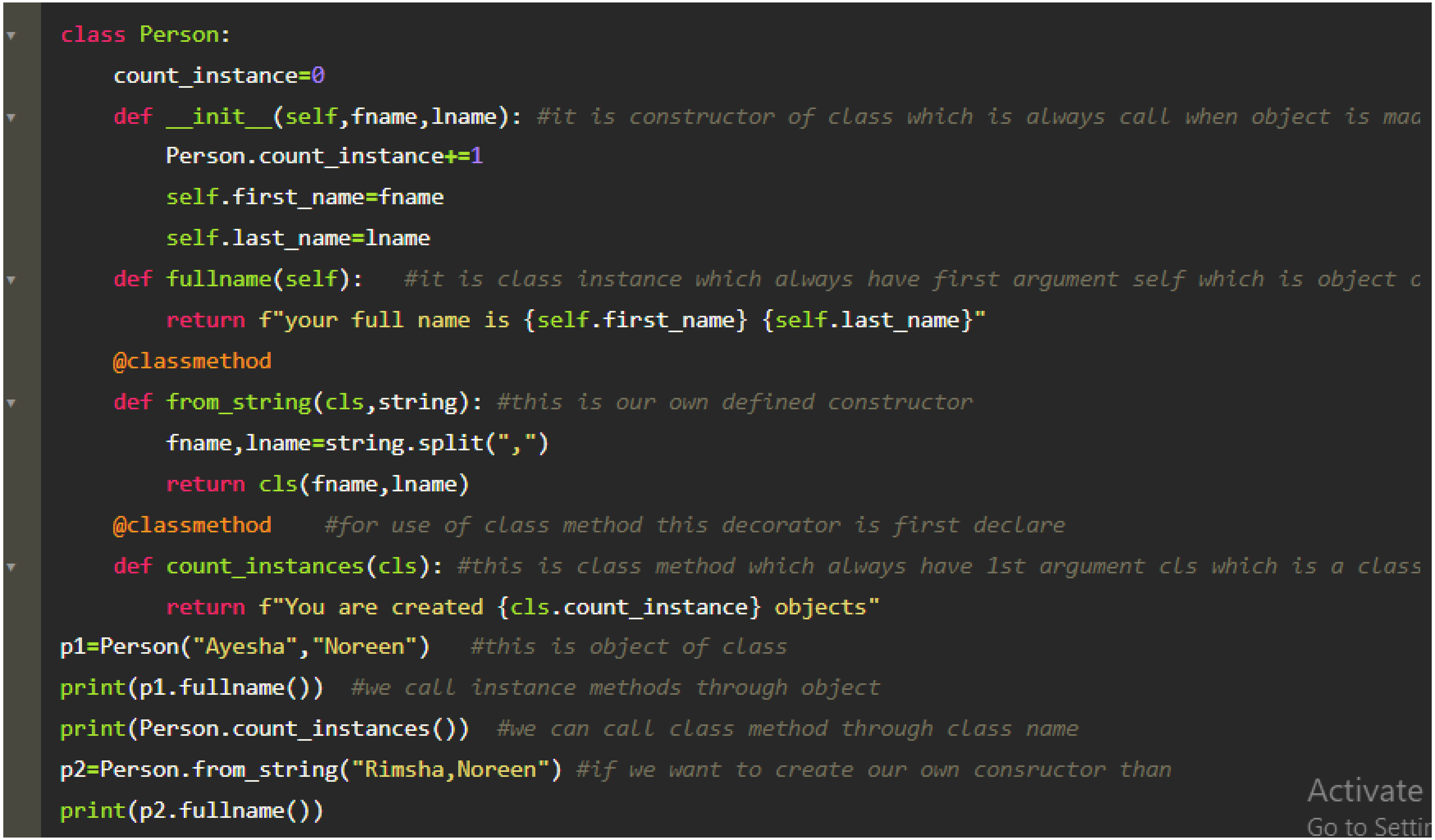
It always take first parameter as cls than other parameters.



```
class Person:              #it is good convension to use 1st capital letter of class
    count_instance=0       #it is class variable or attribute
    def __init__(self,fname,lname): #it is constructor of class which is always call when object is mad
        Person.count_instance+=1    #these are instance variables
        self.first_name=fname
        self.last_name=lname
    def fullname(self):  #it is class instance which always have first argument self which is object
        return f"your full name is {self.first_name} {self.last_name}"
    @classmethod         #for use of class method this decorator is first declare
    def count_instances(cls): #this is class method which always have 1st argument cls which is a class
        return f"You are created {cls.count_instance} objects"
p1=Person("Ayesha","Noreen")   #this is object of class
print(p1.fullname())  #we call instance methods through object
p1=Person("Sana","Rehman")
print(p1.fullname())
print(Person.count_instances())  #we can call class method through class name
```

```
your full name is Ayesha Noreen
your full name is Sana Rehman
You are created 2 objects
```

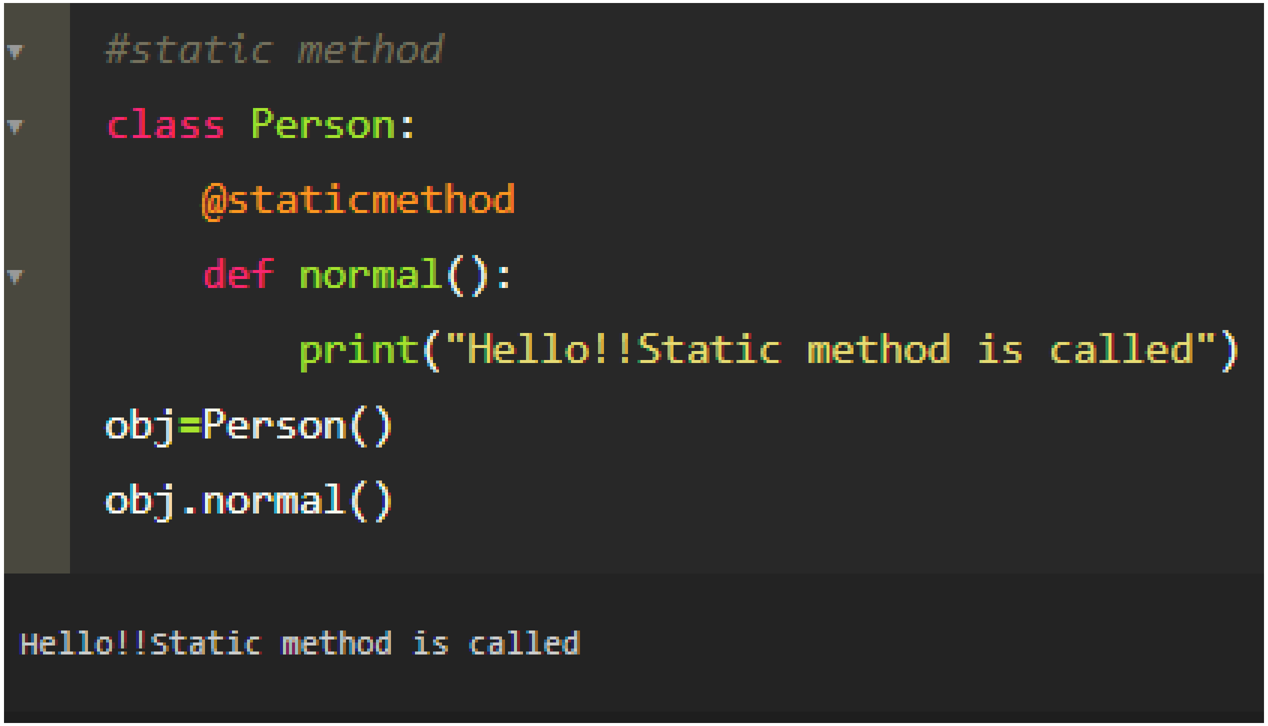
- # @Class method as a Constructor



```
class Person:
    count_instance=0
    def __init__(self,fname,lname): #it is constructor of class which is always call when object is mac
        Person.count_instance+=1
        self.first_name=fname
        self.last_name=lname
    def fullname(self):   #it is class instance which always have first argument self which is object c
        return f"your full name is {self.first_name} {self.last_name}"
    @classmethod
    def from_string(cls,string): #this is our own defined constructor
        fname,lname=string.split(",")
        return cls(fname,lname)
    @classmethod    #for use of class method this decorator is first declare
    def count_instances(cls): #this is class method which always have 1st argument cls which is a class
        return f"You are created {cls.count_instance} objects"
p1=Person("Ayesha","Noreen")    #this is object of class
print(p1.fullname())  #we call instance methods through object
print(Person.count_instances())  #we can call class method through class name
p2=Person.from_string("Rimsha,Noreen") #if we want to create our own consructor than
print(p2.fullname())
```

• Static method

Static method is used when we have no concern with class and object of class. If we want to pass no argument in our function than we use static function because class method always take cls as 1st parameter and class instances also take 1st argument self that is way we make static method. e.g.

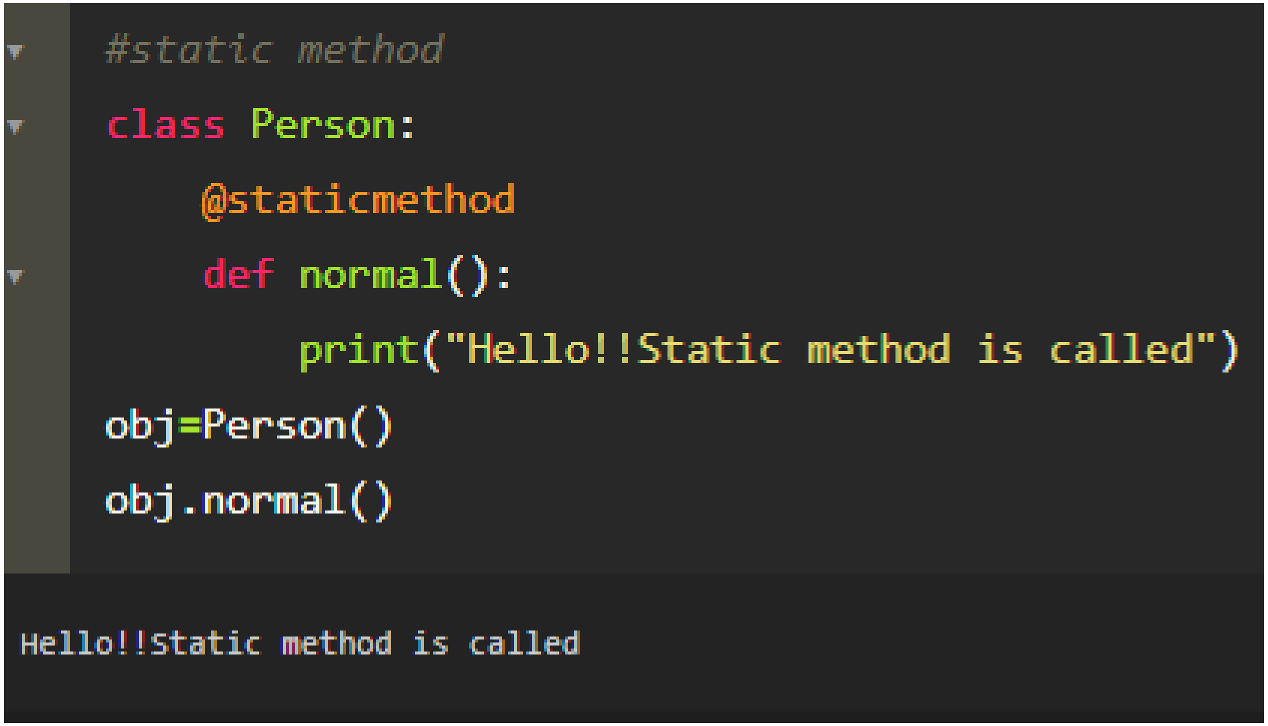


```
#static method
class Person:
    @staticmethod
    def normal():
        print("Hello!!Static method is called")
obj=Person()
obj.normal()
```

```
Hello!!Static method is called
```

- ## Static method

Static method is used when we have no concern with class and object of class. If we want to pass no argument in our function than we use static function because class method always take cls as 1st parameter and class instances also take 1st argument self that is way we make static method. e.g.



```
#static method
class Person:
    @staticmethod
    def normal():
        print("Hello!!Static method is called")
obj=Person()
obj.normal()
```

```
Hello!!Static method is called
```

- **Class variables /attributes vs instance variables /attributes**

Class variables are those variables that we declare in our class these are also called class attributes. class variables or attributes are access through class name while instance variables are those variables that can be access through object and these are declare in init method or instance methods.

- **Class method vs instance method**

Class methods are declare after class decorator which is @classmethod Class method always have 1st argument cls which is class and in this method variables are access through cls. While instance method always have 1st parameter self which is object of class and in this method variables are access by self. These methods are access through object name dot instance method name.

- **Abstraction**

Hide complexity from user is abstraction.

- **Encapsulation**

Is place of useful data at particular place in class. Abstraction is not done until encapsulation is done.

- **Private data In python**

In python nothing is private all is public but a naming convension that we can use for private data is place under score before variable name as _name than python developers consider it as a private variable.

- **Dunder method(Doubleunderscore)/magic method**

Dunder method is a method in which we use double underscore before and after method name as__init__ method or __name__ method etc.

- **Name mangling**

In name mangling when we use _vairablename than print correctly but when we use __with name of variable and print it give error because due to name mangling name is change from __variable name to _classname__variable name when we print _class name__variable name than there is no error

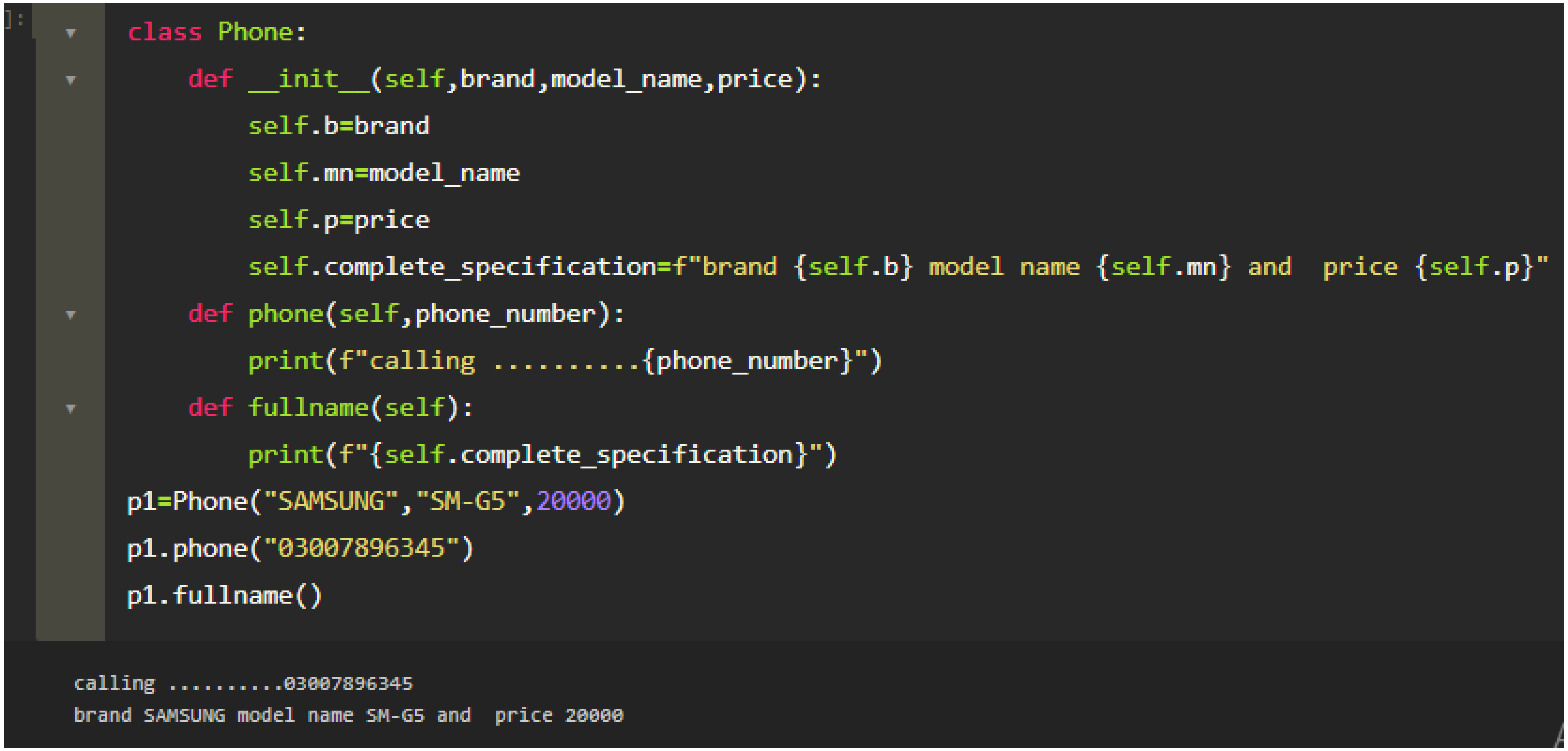
e.g.

__name is variable name and class name is person

Than through name mangling it is converted into _person__name

- # Example

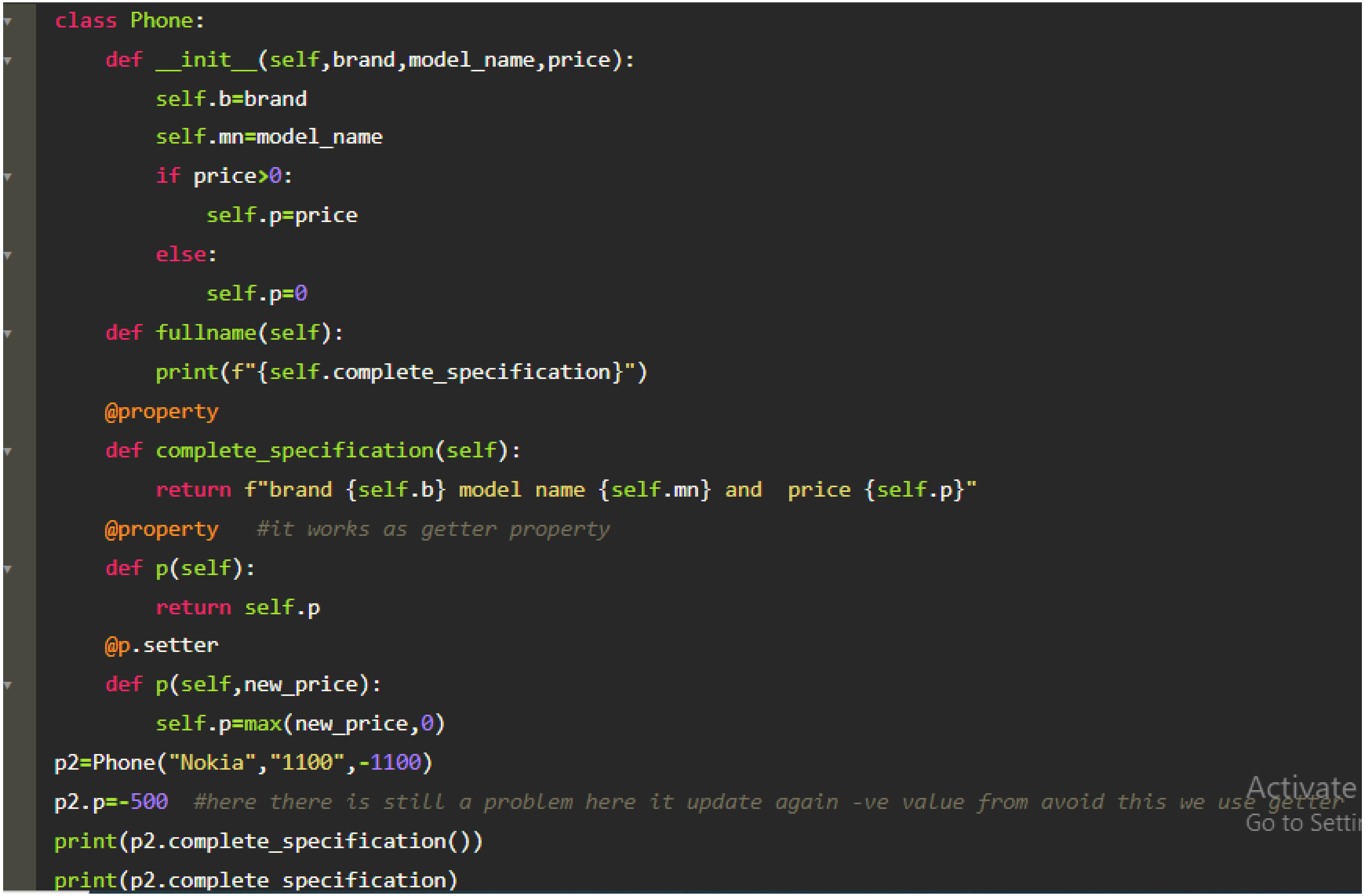
here we create a class phone having instance variables brand, model_name, price and instances init phone and fullname As,



```
class Phone:
    def __init__(self,brand,model_name,price):
        self.b=brand
        self.mn=model_name
        self.p=price
        self.complete_specification=f"brand {self.b} model name {self.mn} and  price {self.p}"
    def phone(self,phone_number):
        print(f"calling ..........{phone_number}")
    def fullname(self):
        print(f"{self.complete_specification}")
p1=Phone("SAMSUNG","SM-G5",20000)
p1.phone("03007896345")
p1.fullname()
```

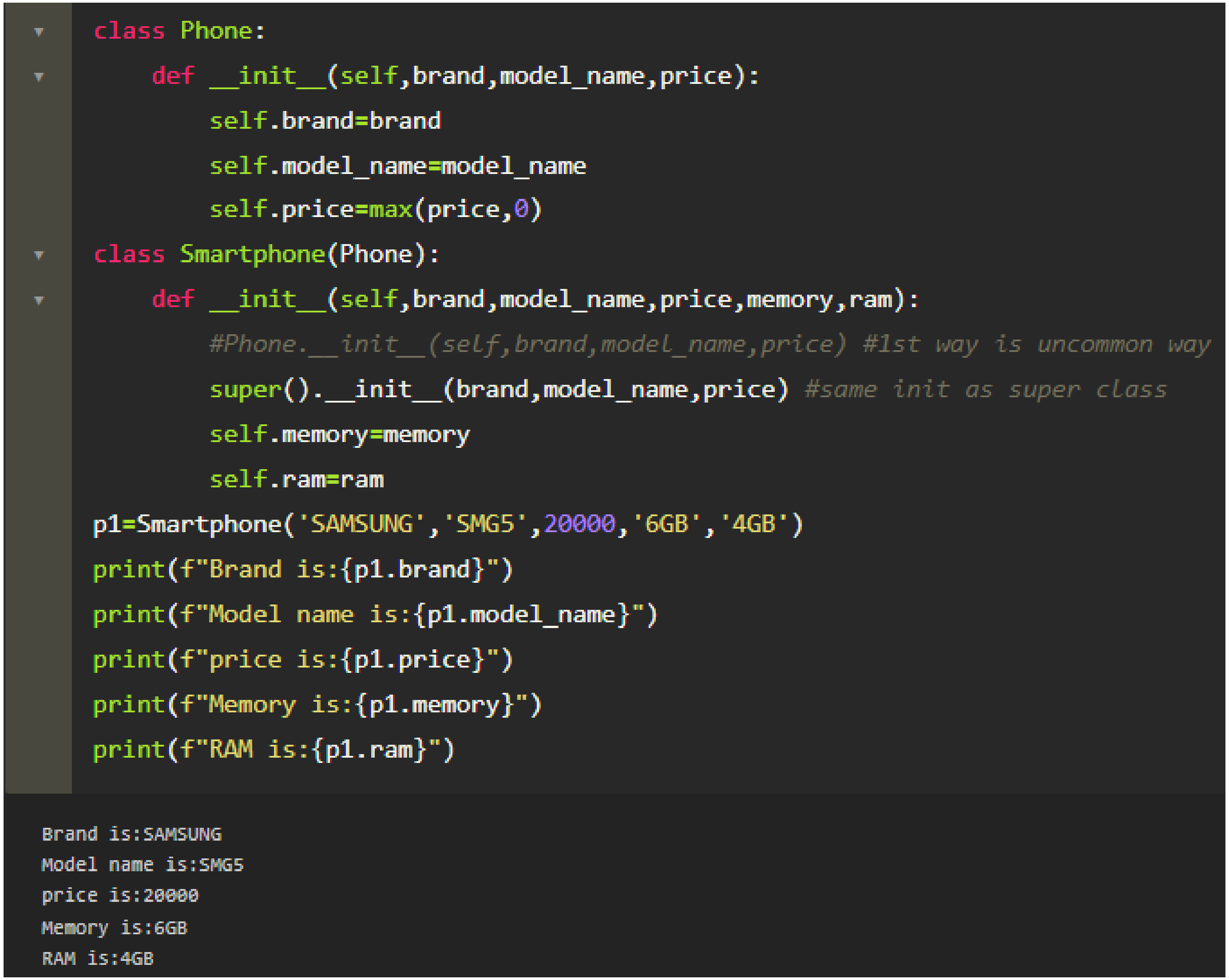
```
calling ..........03007896345
brand SAMSUNG model name SM-G5 and  price 20000
```

• Getter and setter in Python



```python
class Phone:
    def __init__(self,brand,model_name,price):
        self.b=brand
        self.mn=model_name
        if price>0:
            self.p=price
        else:
            self.p=0
    def fullname(self):
        print(f"{self.complete_specification}")
    @property
    def complete_specification(self):
        return f"brand {self.b} model name {self.mn} and  price {self.p}"
    @property   #it works as getter property
    def p(self):
        return self.p
    @p.setter
    def p(self,new_price):
        self.p=max(new_price,0)
p2=Phone("Nokia","1100",-1100)
p2.p=-500  #here there is still a problem here it update again -ve value from avoid this we use getter
print(p2.complete_specification())
print(p2.complete_specification)
```

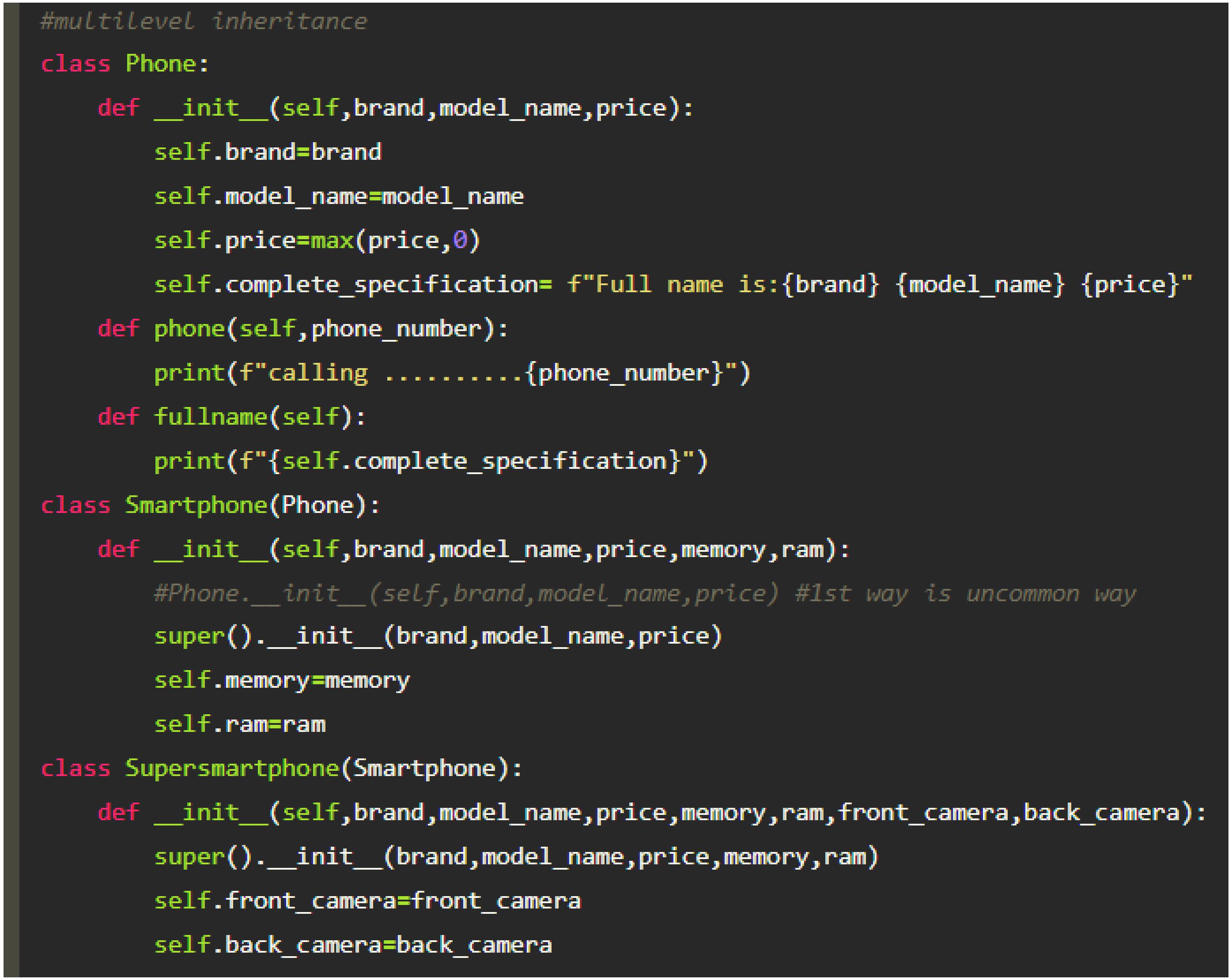
- # Inheritance



```python
class Phone:
    def __init__(self,brand,model_name,price):
        self.brand=brand
        self.model_name=model_name
        self.price=max(price,0)
class Smartphone(Phone):
    def __init__(self,brand,model_name,price,memory,ram):
        #Phone.__init__(self,brand,model_name,price) #1st way is uncommon way
        super().__init__(brand,model_name,price) #same init as super class
        self.memory=memory
        self.ram=ram
p1=Smartphone('SAMSUNG','SMG5',20000,'6GB','4GB')
print(f"Brand is:{p1.brand}")
print(f"Model name is:{p1.model_name}")
print(f"price is:{p1.price}")
print(f"Memory is:{p1.memory}")
print(f"RAM is:{p1.ram}")
```

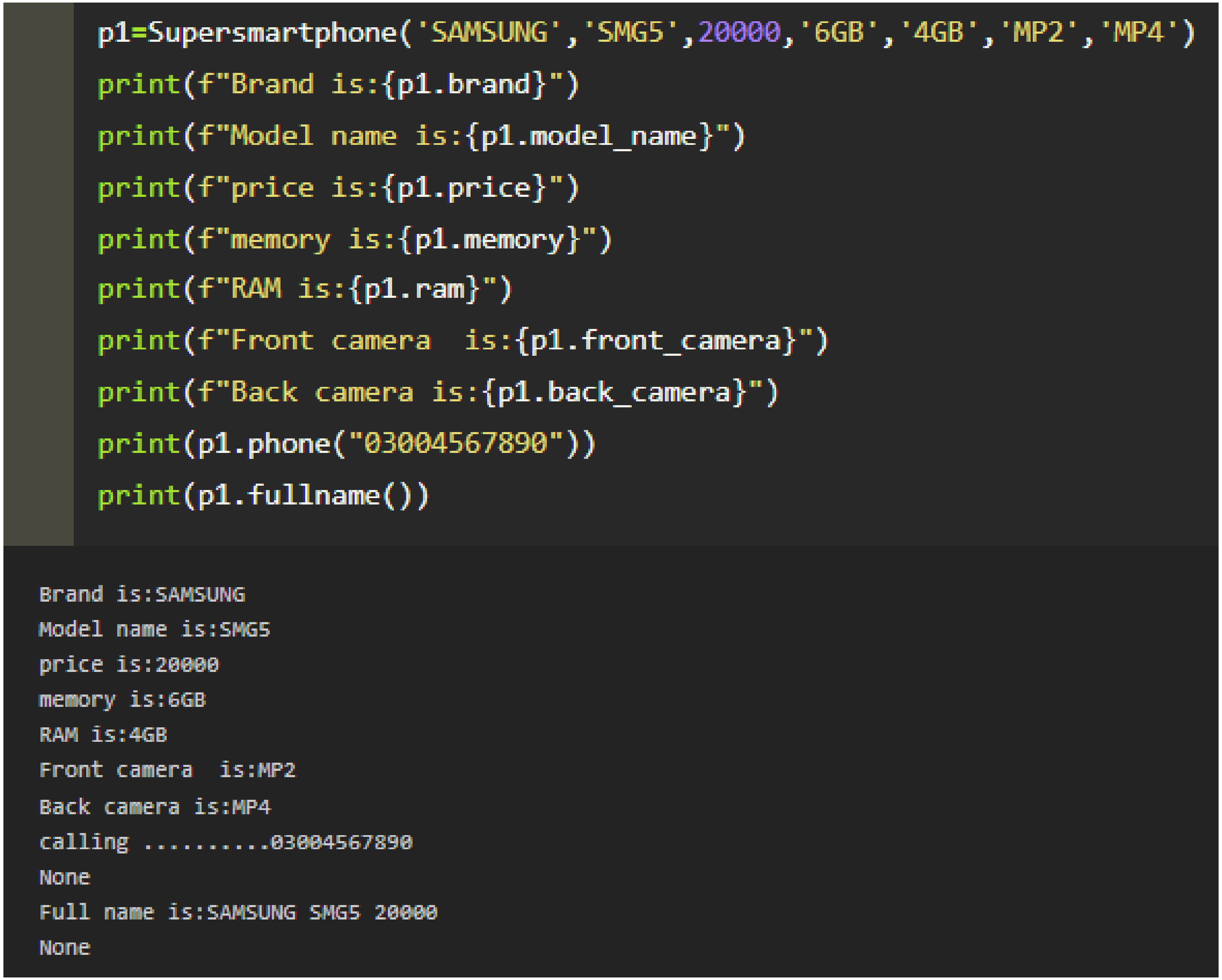
```
Brand is:SAMSUNG
Model name is:SMG5
price is:20000
Memory is:6GB
RAM is:4GB
```

- # Multilevel inheritance



```python
#multilevel inheritance
class Phone:
    def __init__(self,brand,model_name,price):
        self.brand=brand
        self.model_name=model_name
        self.price=max(price,0)
        self.complete_specification= f"Full name is:{brand} {model_name} {price}"
    def phone(self,phone_number):
        print(f"calling ..........{phone_number}")
    def fullname(self):
        print(f"{self.complete_specification}")
class Smartphone(Phone):
    def __init__(self,brand,model_name,price,memory,ram):
        #Phone.__init__(self,brand,model_name,price) #1st way is uncommon way
        super().__init__(brand,model_name,price)
        self.memory=memory
        self.ram=ram
class Supersmartphone(Smartphone):
    def __init__(self,brand,model_name,price,memory,ram,front_camera,back_camera):
        super().__init__(brand,model_name,price,memory,ram)
        self.front_camera=front_camera
        self.back_camera=back_camera
```

- # Multilevel inheritance



```python
p1=Supersmartphone('SAMSUNG','SMG5',20000,'6GB','4GB','MP2','MP4')
print(f"Brand is:{p1.brand}")
print(f"Model name is:{p1.model_name}")
print(f"price is:{p1.price}")
print(f"memory is:{p1.memory}")
print(f"RAM is:{p1.ram}")
print(f"Front camera  is:{p1.front_camera}")
print(f"Back camera is:{p1.back_camera}")
print(p1.phone("03004567890"))
print(p1.fullname())
```

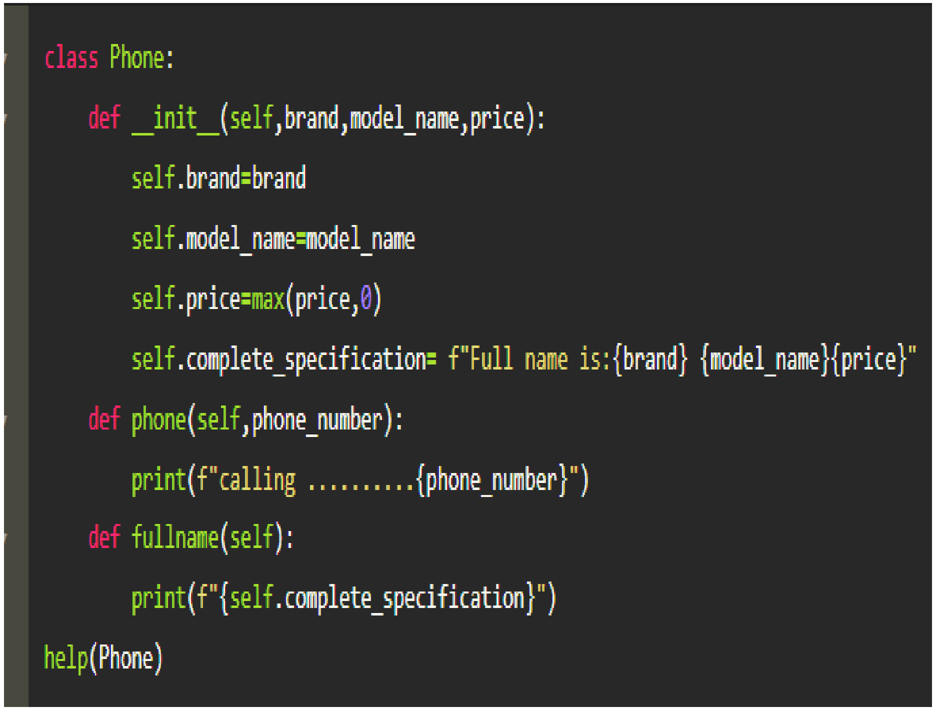
```
Brand is:SAMSUNG
Model name is:SMG5
price is:20000
memory is:6GB
RAM is:4GB
Front camera  is:MP2
Back camera is:MP4
calling ..........03004567890
None
Full name is:SAMSUNG SMG5 20000
None
```

• Method resolution order

Method resolution order is present in every class which guides the object about order of execution of methods of that class. we can check method resolution order of any class by placing class name in help method. we can also use mro() function as, classname.mro() OR   classname.__mro__

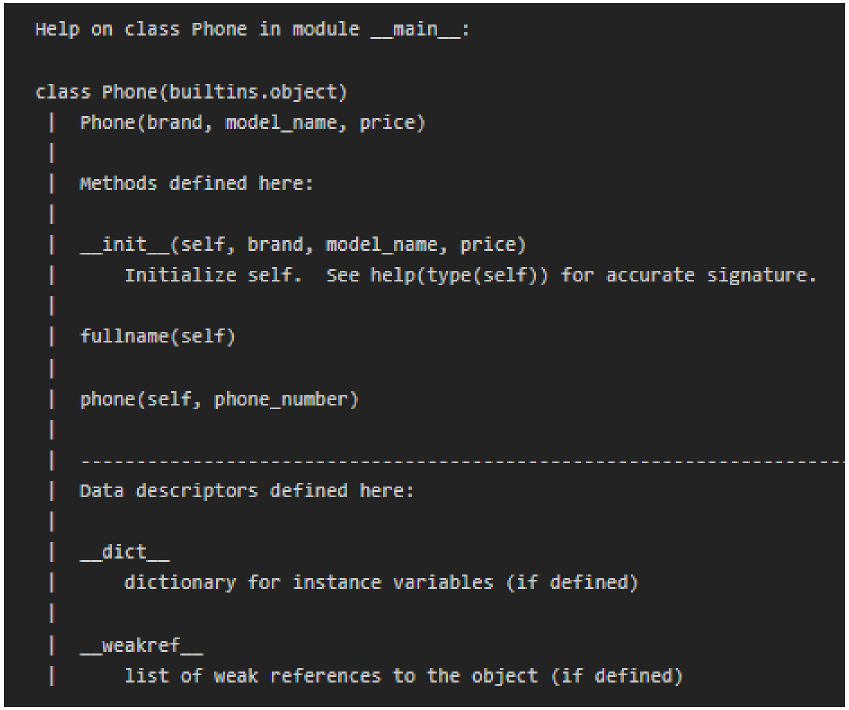
But both above method prints a list which tells as which is call 1st and which is next and use of help method give meaningfull information.

Input



```
class Phone:
    def __init__(self,brand,model_name,price):
        self.brand=brand
        self.model_name=model_name
        self.price=max(price,0)
        self.complete_specification= f"Full name is:{brand} {model_name}{price}"
    def phone(self,phone_number):
        print(f"calling .........{phone_number}")
    def fullname(self):
        print(f"{self.complete_specification}")
help(Phone)
```

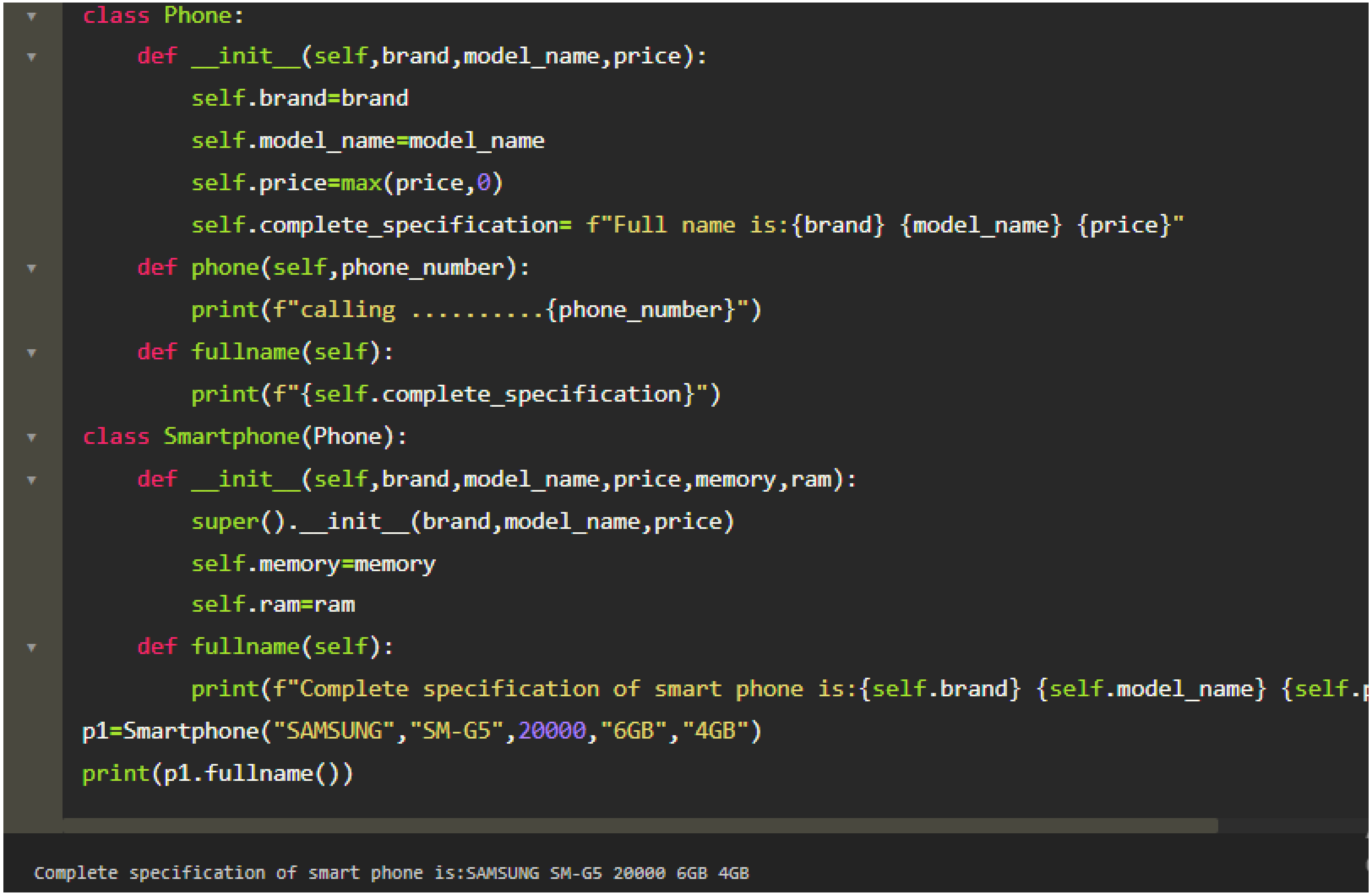
Output



```
Help on class Phone in module __main__:

class Phone(builtins.object)
 |  Phone(brand, model_name, price)
 |
 |  Methods defined here:
 |
 |  __init__(self, brand, model_name, price)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  fullname(self)
 |
 |  phone(self, phone_number)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

- # Method overriding

Method overriding mean if there is a method in parent class and if same name method with same signature (mean parameters are also same) is also present in child class and if we want to call parent class method than it is not called instead child class method is called this is called function overriding this is due to method resolution order.

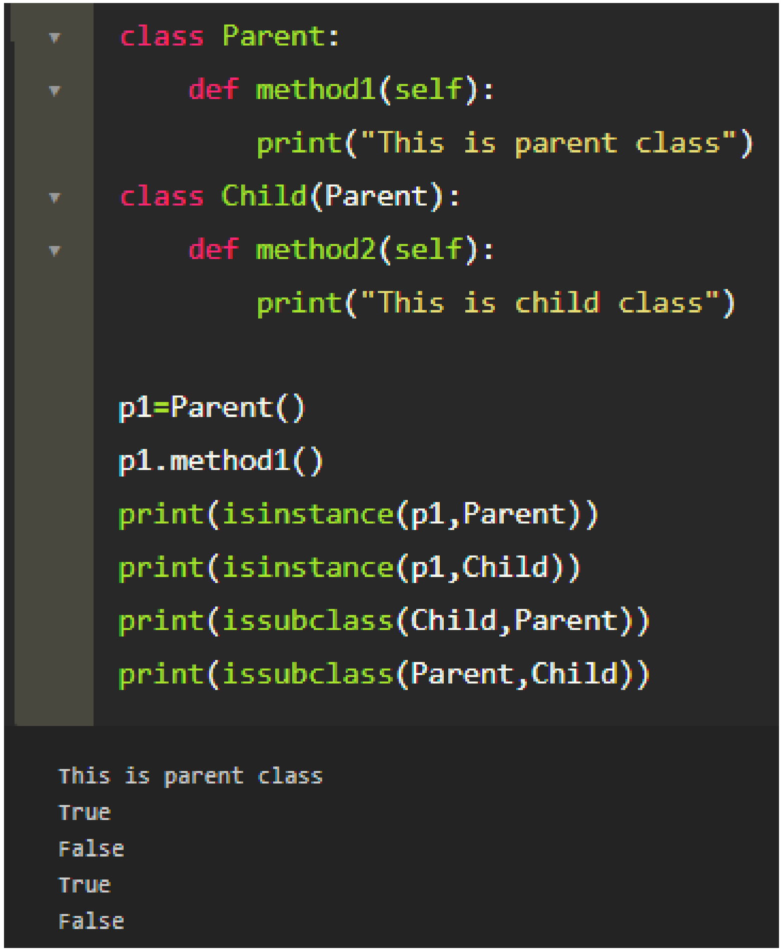


```
class Phone:
    def __init__(self,brand,model_name,price):
        self.brand=brand
        self.model_name=model_name
        self.price=max(price,0)
        self.complete_specification= f"Full name is:{brand} {model_name} {price}"
    def phone(self,phone_number):
        print(f"calling ..........{phone_number}")
    def fullname(self):
        print(f"{self.complete_specification}")
class Smartphone(Phone):
    def __init__(self,brand,model_name,price,memory,ram):
        super().__init__(brand,model_name,price)
        self.memory=memory
        self.ram=ram
    def fullname(self):
        print(f"Complete specification of smart phone is:{self.brand} {self.model_name} {self.p
p1=Smartphone("SAMSUNG","SM-G5",20000,"6GB","4GB")
print(p1.fullname())
```

```
Complete specification of smart phone is:SAMSUNG SM-G5 20000 6GB 4GB
```

isinstance() and issubclass() method

Isinstance() method checks is an instance is instance of a class or not if yes return true otherwise false and issubclass() checks is a class is subclass of other class or not if yes return true otherwise return false.



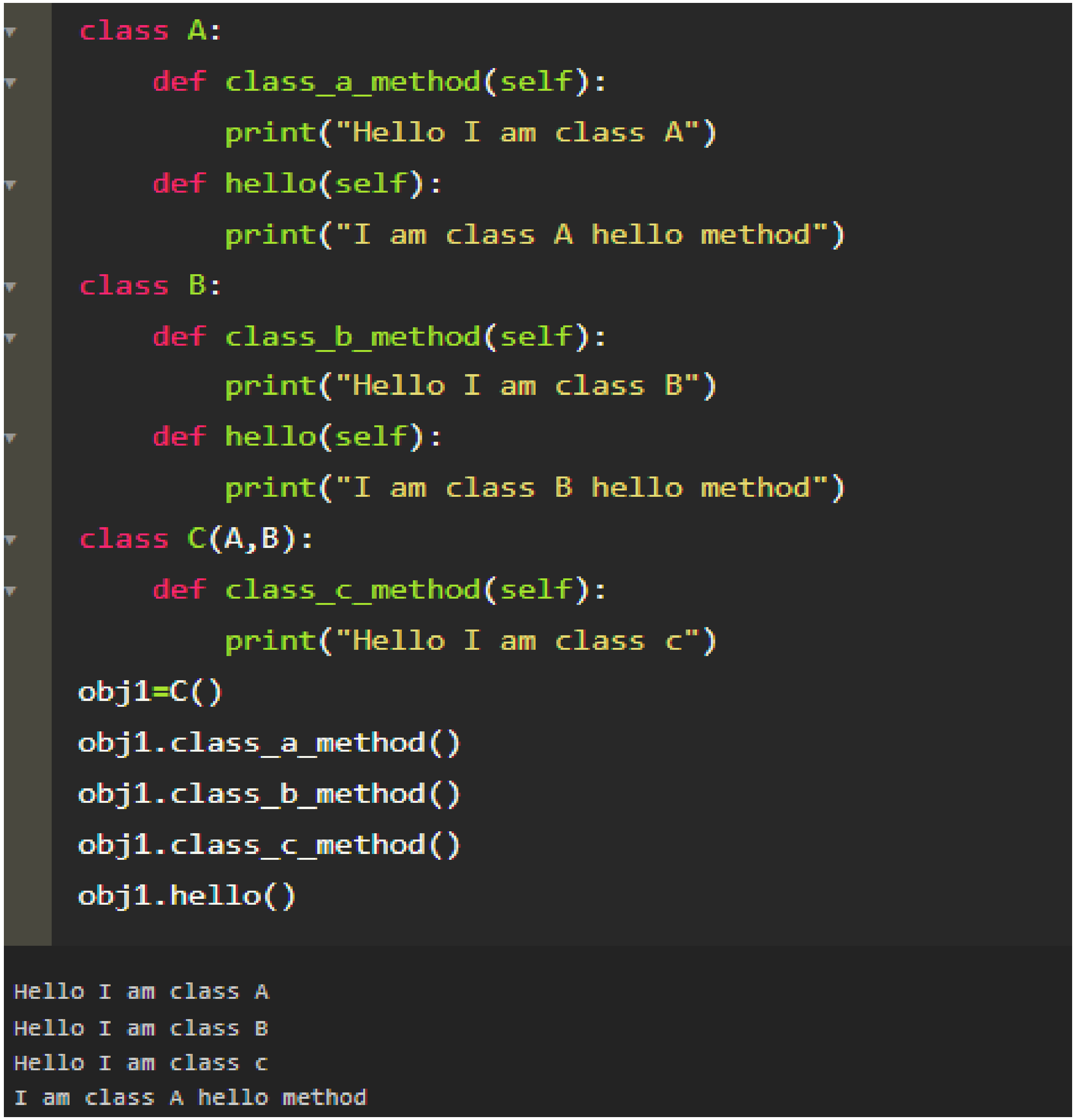
```
class Parent:
    def method1(self):
        print("This is parent class")
class Child(Parent):
    def method2(self):
        print("This is child class")

p1=Parent()
p1.method1()
print(isinstance(p1,Parent))
print(isinstance(p1,Child))
print(issubclass(Child,Parent))
print(issubclass(Parent,Child))
```

```
This is parent class
True
False
True
False
```

- ## Multiple inheritance

In multiple inheritance child class is inherited from many parent classes. In example in next slide we create a class A and a class B and a class C class. Class C inherits both class A and class B this is multiple inheritance. Class C contain functionalities of both class A and class B we can access methods of all classes from object of class C but we create hello method in class A as well as in class B when we call this hello method than class A hello is call why?? The reason is that we inherit C from A and B as C(A,B) when we call mro() of class C than it print order which is first class C than class A than class B executes so class A hello() is call first how we can call class B hello() for this when we inherit class C from A and B than inherit as C(B,A) now mro() method show order as C,B than A so class B hello()is call first.

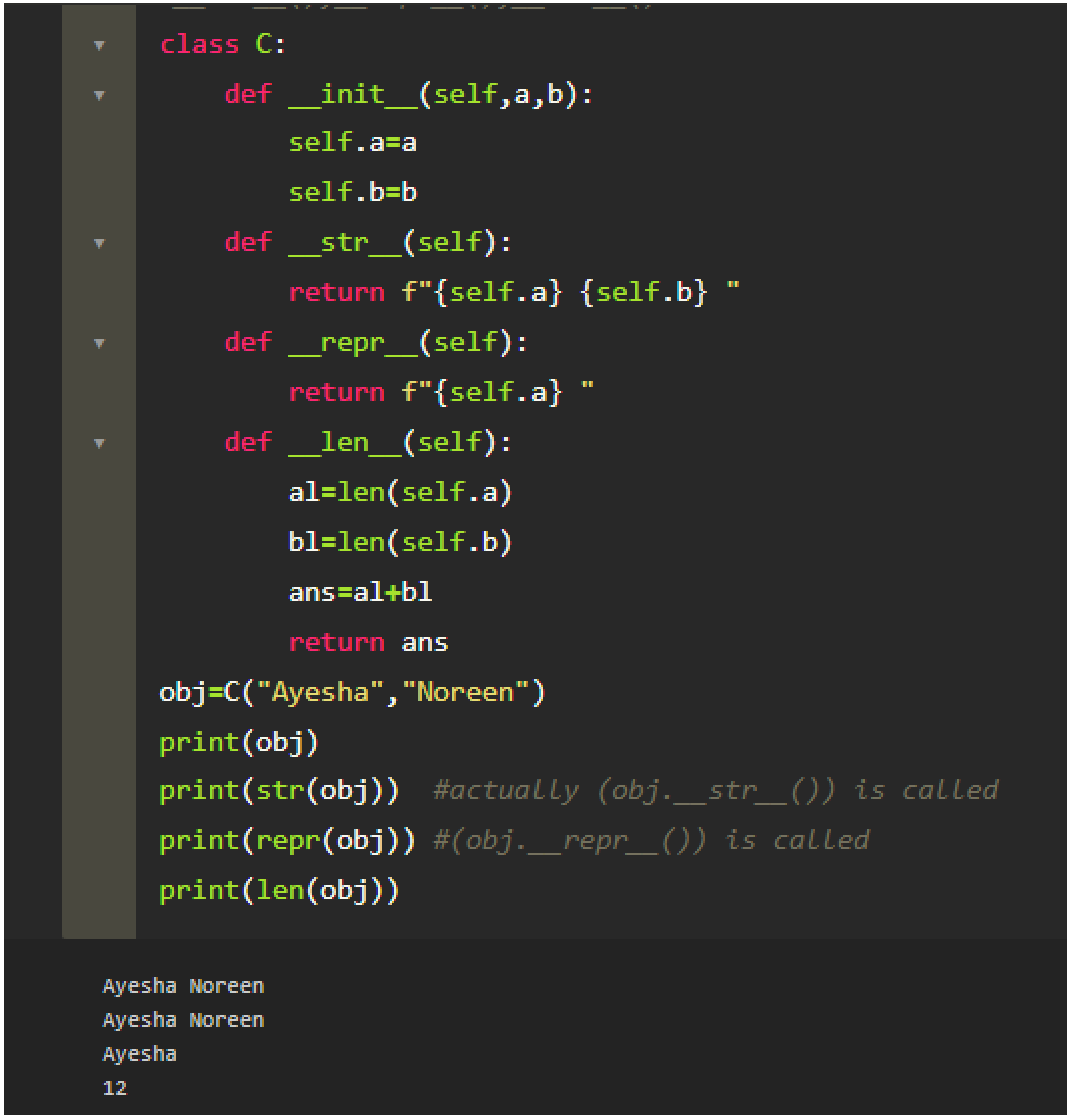
Example



```
class A:
    def class_a_method(self):
        print("Hello I am class A")
    def hello(self):
        print("I am class A hello method")
class B:
    def class_b_method(self):
        print("Hello I am class B")
    def hello(self):
        print("I am class B hello method")
class C(A,B):
    def class_c_method(self):
        print("Hello I am class c")
obj1=C()
obj1.class_a_method()
obj1.class_b_method()
obj1.class_c_method()
obj1.hello()
```

```
Hello I am class A
Hello I am class B
Hello I am class c
I am class A hello method
```

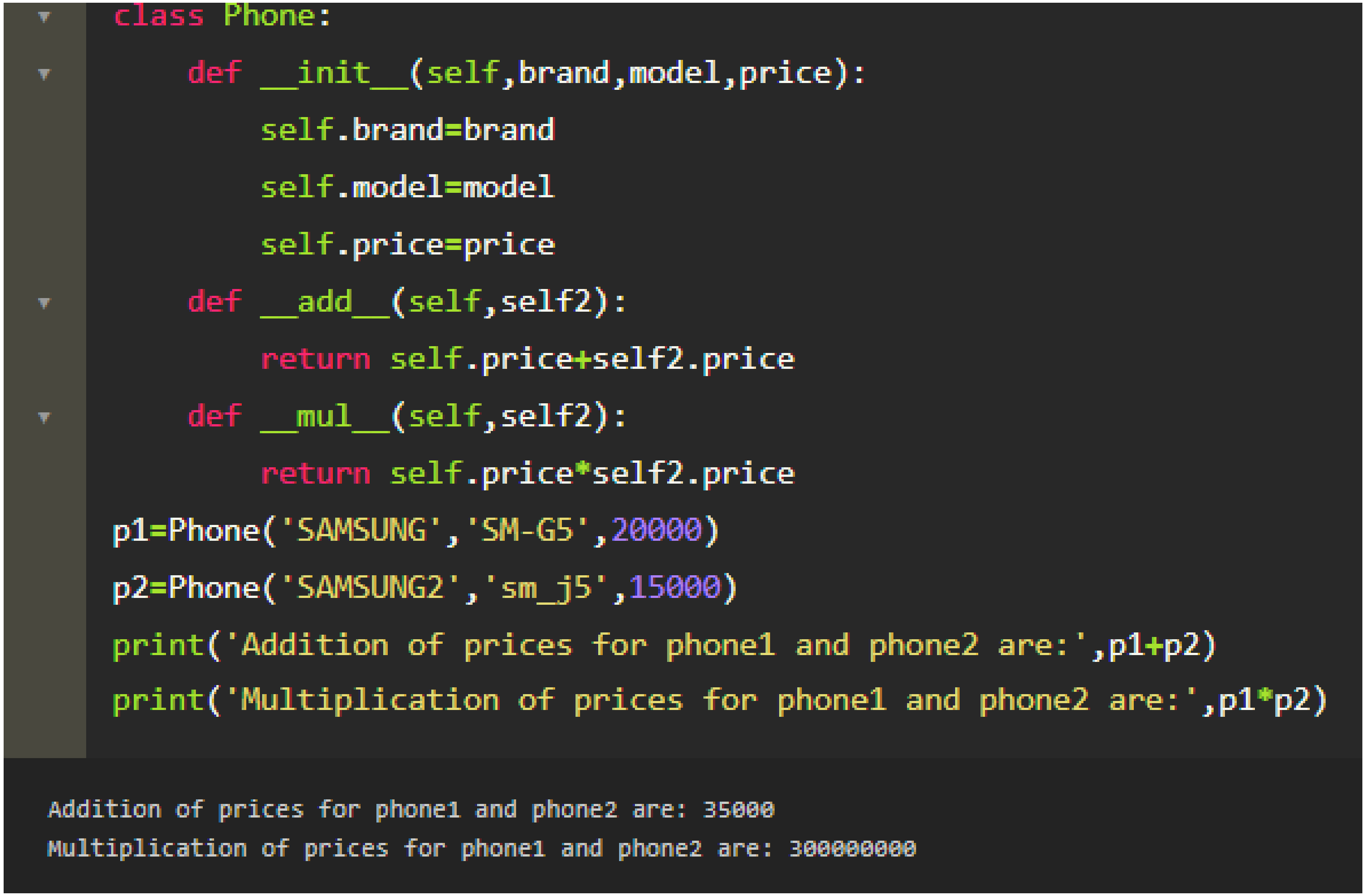
• Special magic methods __str__(),__repr__(),__len__()



```
class C:
    def __init__(self,a,b):
        self.a=a
        self.b=b
    def __str__(self):
        return f"{self.a} {self.b} "
    def __repr__(self):
        return f"{self.a} "
    def __len__(self):
        al=len(self.a)
        bl=len(self.b)
        ans=al+bl
        return ans
obj=C("Ayesha","Noreen")
print(obj)
print(str(obj))  #actually (obj.__str__()) is called
print(repr(obj)) #(obj.__repr__()) is called
print(len(obj))
```

```
Ayesha Noreen
Ayesha Noreen
Ayesha
12
```

- # Special magic methods __add__(),__mul__()



```
class Phone:
    def __init__(self,brand,model,price):
        self.brand=brand
        self.model=model
        self.price=price
    def __add__(self,self2):
        return self.price+self2.price
    def __mul__(self,self2):
        return self.price*self2.price
p1=Phone('SAMSUNG','SM-G5',20000)
p2=Phone('SAMSUNG2','sm_j5',15000)
print('Addition of prices for phone1 and phone2 are:',p1+p2)
print('Multiplication of prices for phone1 and phone2 are:',p1*p2)
```

```
Addition of prices for phone1 and phone2 are: 35000
Multiplication of prices for phone1 and phone2 are: 300000000
```

- **Polymorphism**

Poly mean many and morphism mean shapes. In polymorphism an object behaves differently in different situations. Polymorphism occurs in inheritance. In polymorphism there are more than one form of operator or methods. It use the concept of both overloading and overriding.