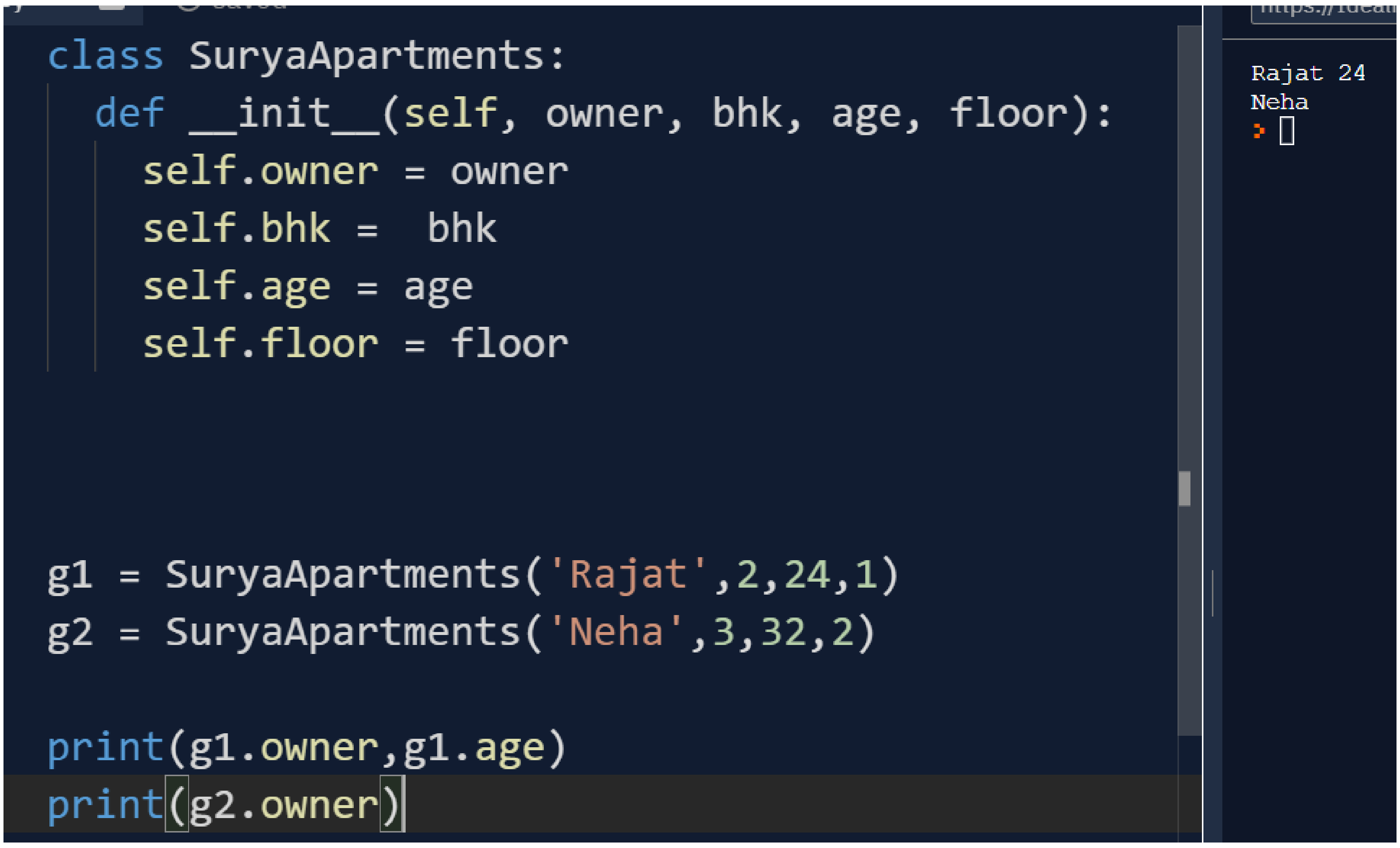
PYTHON NOTES-2

OBJECT ORIENTED PROGRAMMING IN PYTHON

- Class is a blueprint for creating objects.



```python
class SuryaApartments:
  def __init__(self, owner, bhk, age, floor):
    self.owner = owner
    self.bhk =  bhk
    self.age = age
    self.floor = floor



g1 = SuryaApartments('Rajat',2,24,1)
g2 = SuryaApartments('Neha',3,32,2)

print(g1.owner,g1.age)
print(g2.owner)
```

```
Rajat 24
Neha
```

- All classes have a function called ***__init__ ()***, which is always executed when the class is being initiated.

- Use the ***__init__ ()*** function to assign values to object properties, or other operations that are necessary to do when the object is being created.

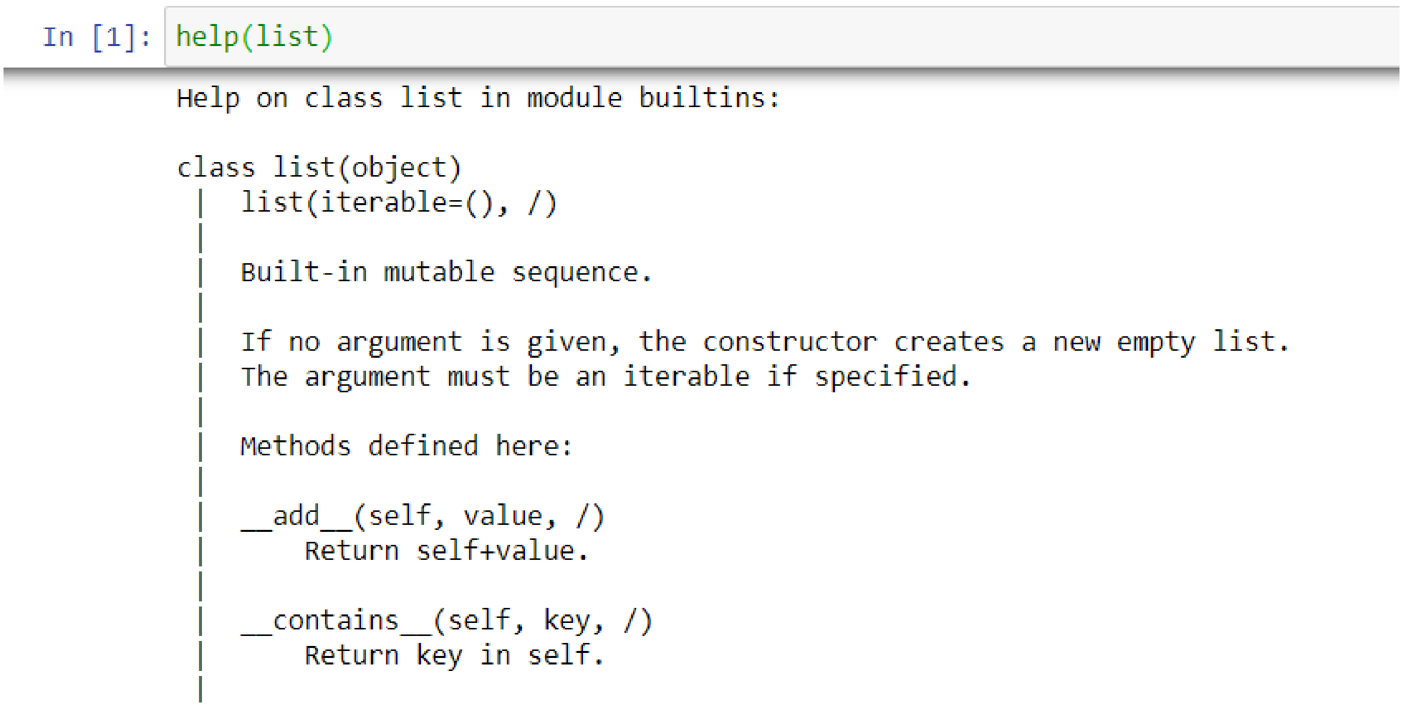
Create a class named Person, use the __init__() function to assign values for name and age:

```python
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

- The ***__init__ ()*** function is called automatically every time the class is being used to create a new object.

ATTRIBUTES AND METHODS

- OOP allow us to write a code which is repeatable, well- organised , memory efficient.

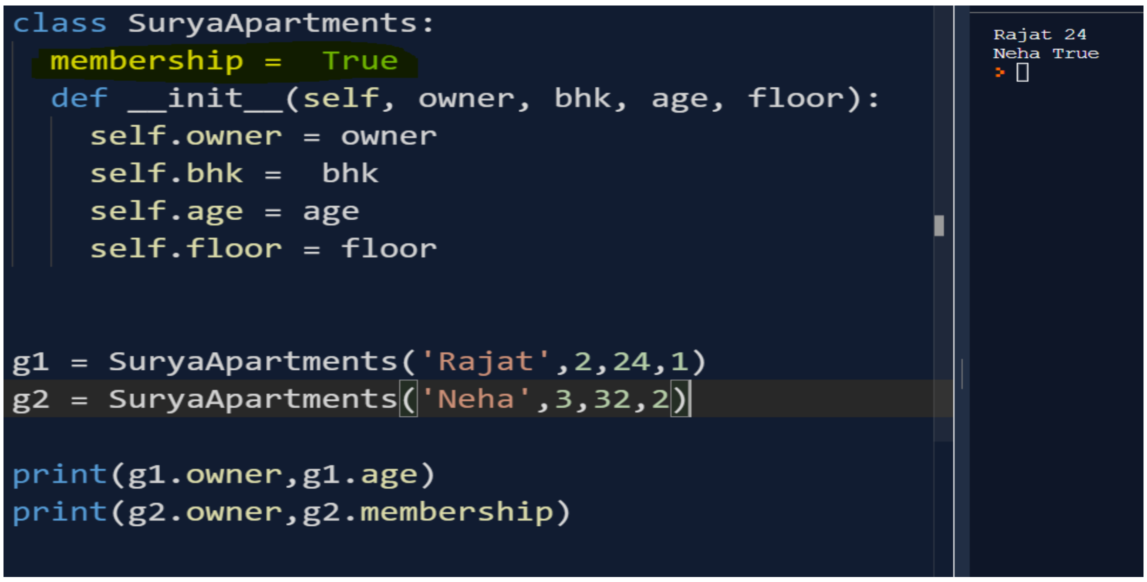


```
In [1]: help(list)

Help on class list in module builtins:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
```

- Help keyword used for getting the blueprint of the class.

CLASS OBJECT ATTRIBUTES / CLASS VARIABLE / CLASS ATTRIBUTES

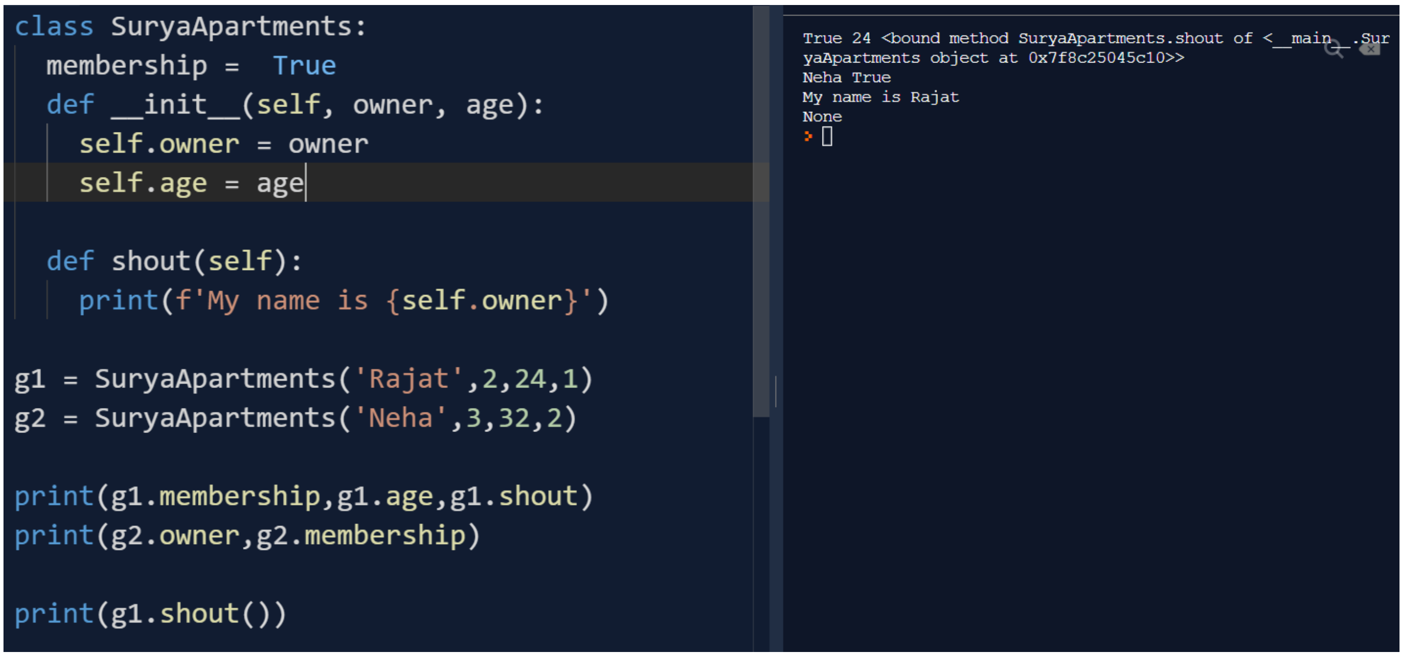
- Class object attributes are static in nature.
- A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.



```python
class SuryaApartments:
  membership =  True
  def __init__(self, owner, bhk, age, floor):
    self.owner = owner
    self.bhk =  bhk
    self.age = age
    self.floor = floor


g1 = SuryaApartments('Rajat',2,24,1)
g2 = SuryaApartments('Neha',3,32,2)

print(g1.owner,g1.age)
print(g2.owner,g2.membership)
```

```
Rajat 24
Neha True
```

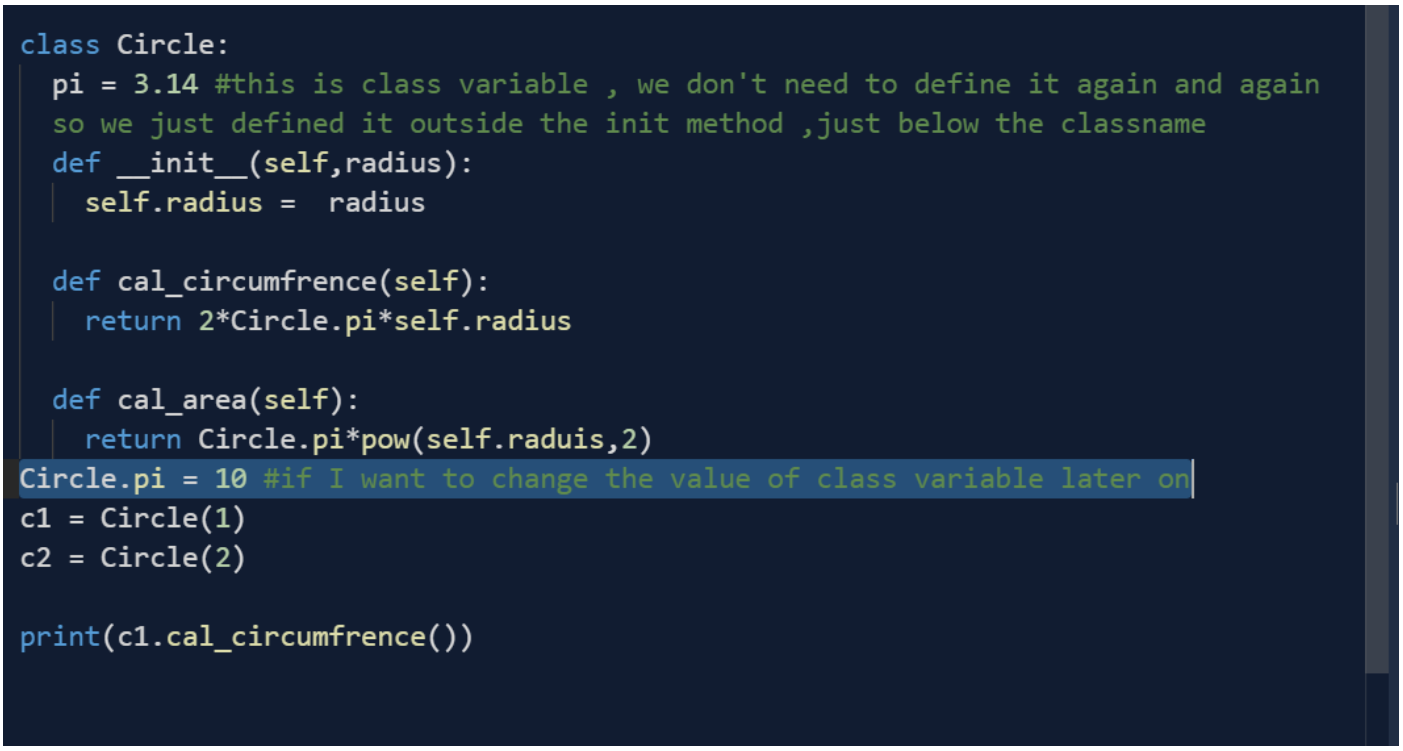
- Membership = True, here is class object attribute or class variable .



```python
class SuryaApartments:
  membership =  True
  def __init__(self, owner, age):
    self.owner = owner
    self.age = age

  def shout(self):
    print(f'My name is {self.owner}')

g1 = SuryaApartments('Rajat',2,24,1)
g2 = SuryaApartments('Neha',3,32,2)

print(g1.membership,g1.age,g1.shout)
print(g2.owner,g2.membership)

print(g1.shout())
```

```
True 24 <bound method SuryaApartments.shout of <__main__.Sur
yaApartments object at 0x7f8c25045c10>>
Neha True
My name is Rajat
None
```

Example:



```python
class Circle:
  pi = 3.14 #this is class variable , we don't need to define it again and again
  so we just defined it outside the init method ,just below the classname
  def __init__(self,radius):
    self.radius =  radius

  def cal_circumfrence(self):
    return 2*Circle.pi*self.radius

  def cal_area(self):
    return Circle.pi*pow(self.raduis,2)
Circle.pi = 10 #if I want to change the value of class variable later on
c1 = Circle(1)
c2 = Circle(2)

print(c1.cal_circumfrence())
```

- For changing the value of class variable on later stages the just give this syntax:

```
className.classvaribale =  new value
```

- How to know how many variables an object has?

```
objectName.__dict__
```

Above command will give you all the variable an object will have in form of dictionary.

https://wwwyoutube.com/watch?v=BLtIZZimuVE&list=PLwgFb6VsUj_lQTpQKDtLXKXElQychT_2j&index=194&t=0s

Important Note:

```
class Exam:
    grace_marks = 0.05
    def __init__(self,subject,gain):
     self.subject = subject
     self.gain = gain

    def total_marks(self):
      return self.gain + self.gain*self.grace_marks # if you need different class varibale value for different objects then we have to self.classvaribale instead of classname.classvaribale.

s1 = Exam('History',55)
s2 = Exam('Maths',90)
s3 = Exam('Hindi',72)

s2.grace_marks = 0.1     #for giving specific value to class variable of particular object

print(s1.subject)
print(s1.total_marks())
print(s2.total_marks())
```

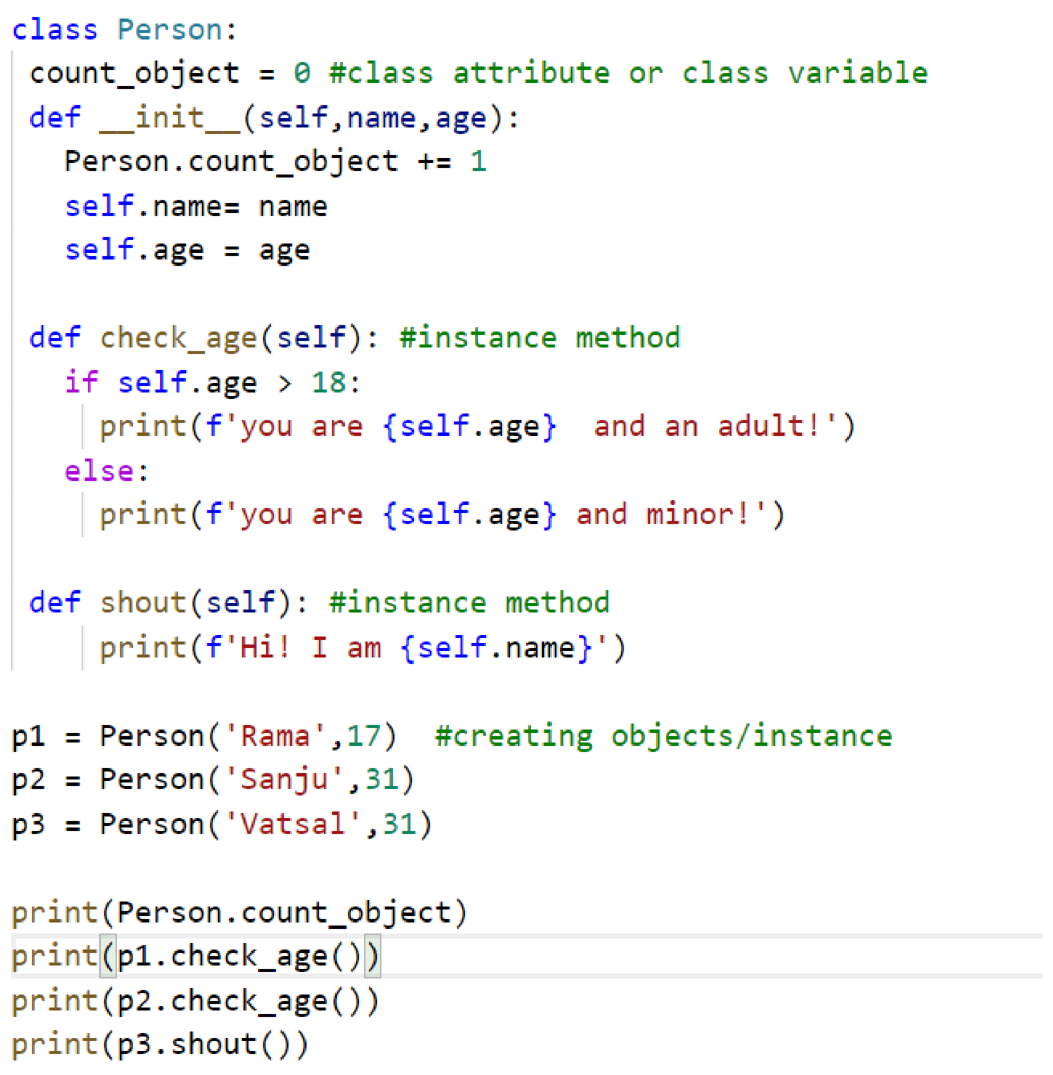
- In above code for all subject grace marks were 0.05 and for one subject it was 0.1, so here we can see that class variable value is not same for all object. So, I have to take self.class_variable name instead of className.class_variable in return statement.

Code Exercise: count the object created.

```
class Person:
 count_object = 0
 def __init__(self,name,age):
   Person.count_object += 1
   self.name= name
   self.age = age

p1 = Person('Ra',23)
p2 = Person('Sa',31)
p3 = Person('Va',31)

print (Person.count_object)
```

```
Output: 3
```

INSTANCE METHODS



```
class Person:
    count_object = 0 #class attribute or class variable
    def __init__(self,name,age):
        Person.count_object += 1
        self.name= name
        self.age = age

    def check_age(self): #instance method
        if self.age > 18:
            print(f'you are {self.age}  and an adult!')
        else:
            print(f'you are {self.age} and minor!')

    def shout(self): #instance method
        print(f'Hi! I am {self.name}')

p1 = Person('Rama',17)  #creating objects/instance
p2 = Person('Sanju',31)
p3 = Person('Vatsal',31)

print(Person.count_object)
print(p1.check_age())
print(p2.check_age())
print(p3.shout())
```

- Instance methods can be used with all the class instance / class objects which we will create inside the class.

- Instance methods are the most common type of methods in Python classes. These are so called because they can access unique data of their instance. If you have two objects each created from a car class, then they each may have different properties. They may have different colors, engine sizes, seats, and so on.

- Instance methods must have **self** as a parameter, but you don’t need to pass this in every time. Self is another Python special term. Inside any instance method, you can use self to access any data or methods that may reside in your class. You won’t be able to access them without going through self.

- Finally, as instance methods are the most common, there’s no decorator needed. Any method you create will automatically be created as an instance method, unless you tell Python otherwise.

CLASS METHODS

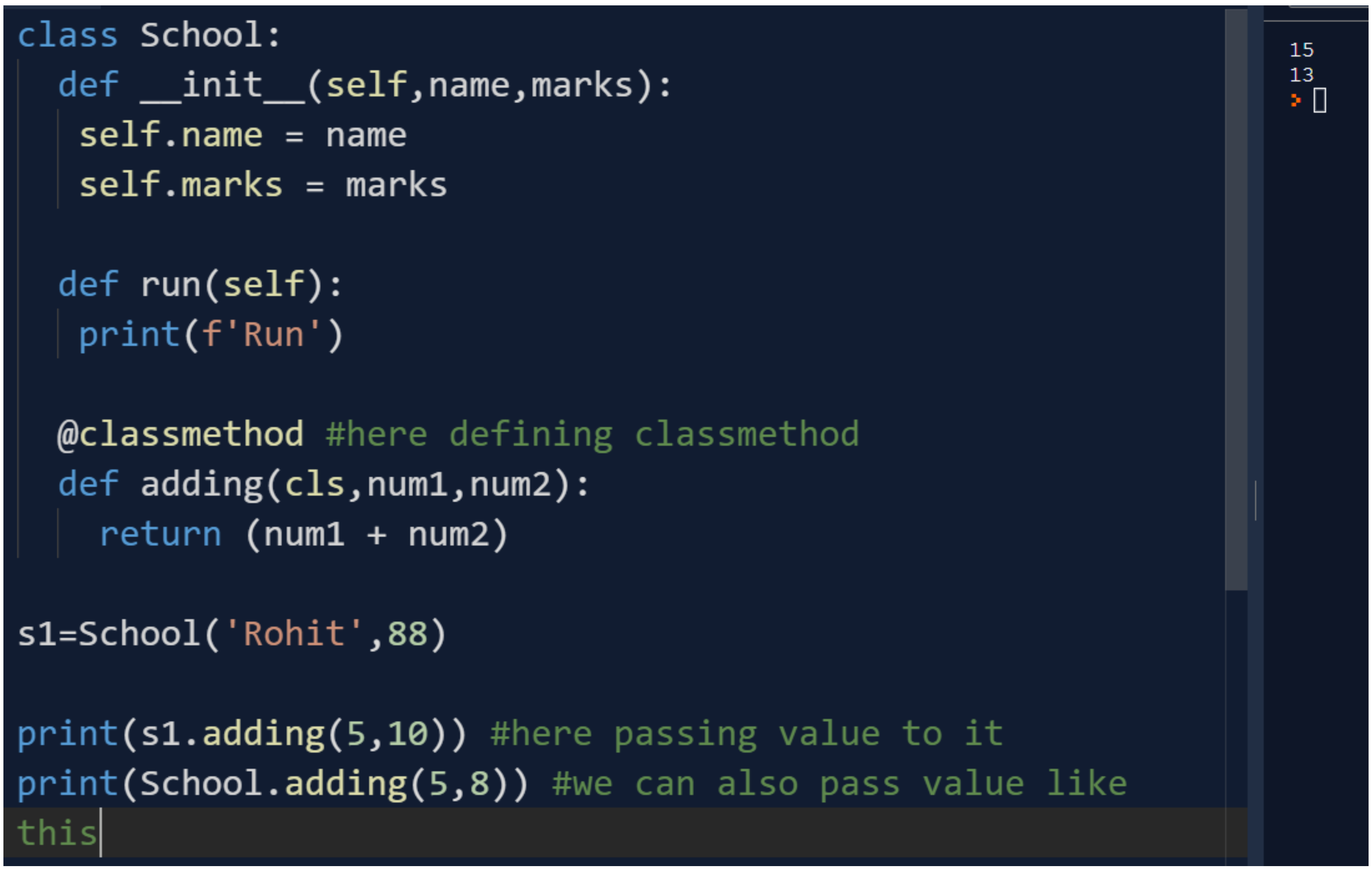
- Class methods know about their class. They can't access specific instance data, but they can call other static methods.
- Class methods don't need **self** as an argument, but they do need a parameter called **cls**. This stands for **class**, and like self, gets automatically passed in by Python.
- Class methods are created using the **@classmethod** decorator

Syntax: **@classmethod**

- Convert a function to be a class method.
- A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:
- Syntax

```
class C:
 @classmethod
    def function_name (cls, arg1, arg2, ...)
```

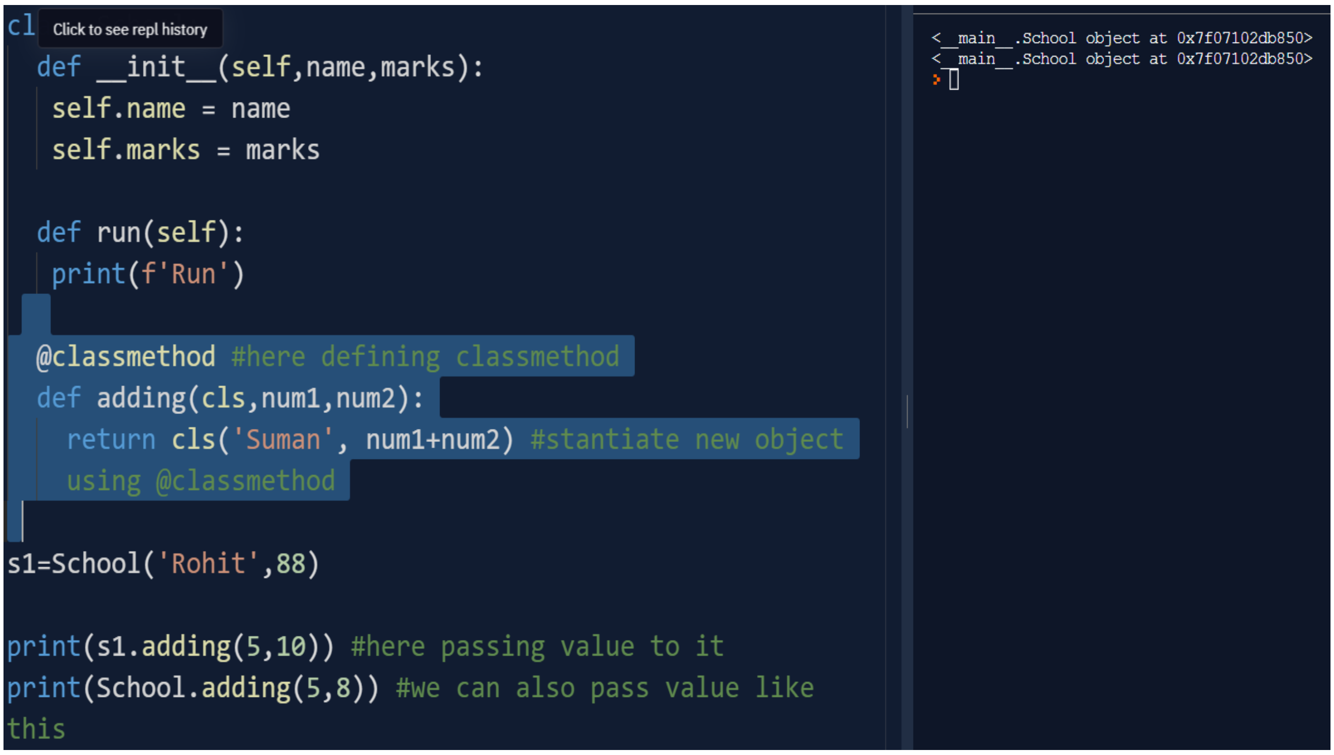
- It can be called either on the class (e.g. C.f()) or on an instance
- (e.g. **C () .f ()**). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.
- Class methods are different than C++ or Java static methods. If you want those, see the static method builtin
- **We can also create new object inside the class using @classmethod:**



```
class School:
  def __init__(self,name,marks):
    self.name = name
    self.marks = marks

  def run(self):
    print(f'Run')

  @classmethod #here defining classmethod
  def adding(cls,num1,num2):
    return (num1 + num2)

s1=School('Rohit',88)

print(s1.adding(5,10)) #here passing value to it
print(School.adding(5,8)) #we can also pass value like this
```

```
15
13
```

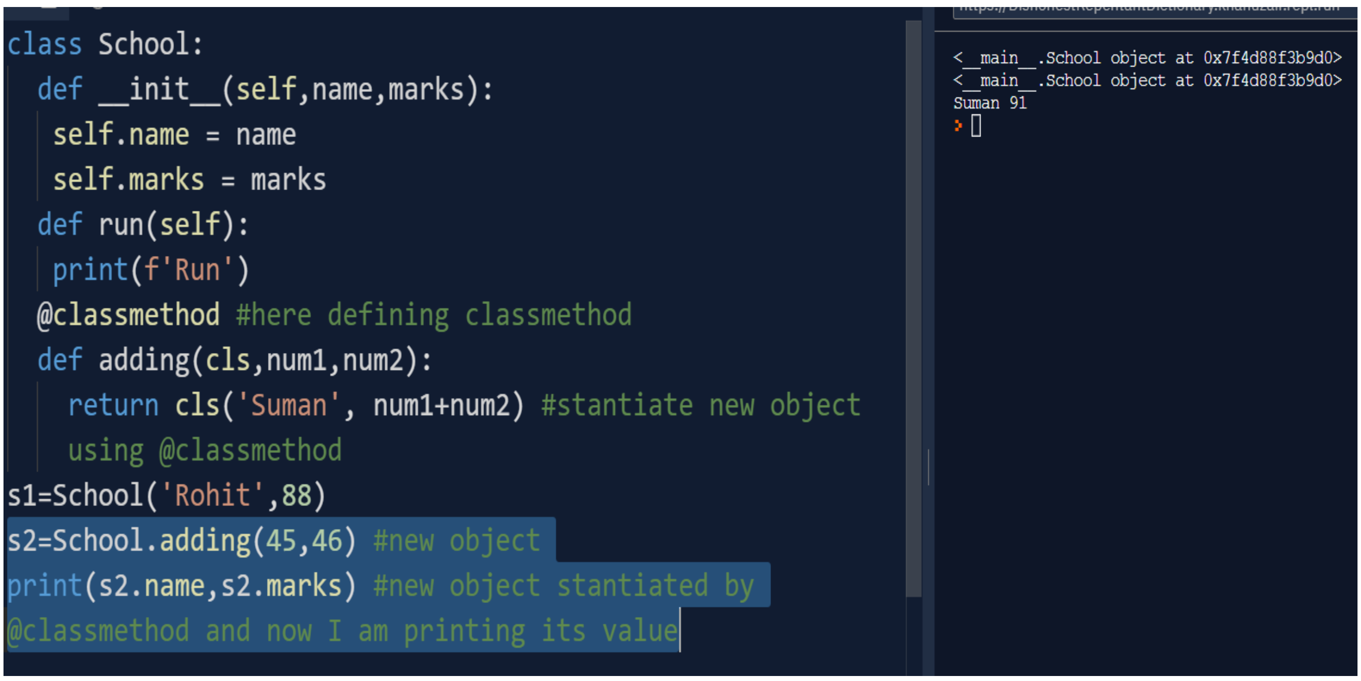
We can also create a new object using @classmethod: Here I have added one new object ‘Suman’ in class School using @classmethod.



```
cl  Click to see repl history
  def __init__(self,name,marks):
    self.name = name
    self.marks = marks

  def run(self):
    print(f'Run')

  @classmethod #here defining classmethod
  def adding(cls,num1,num2):
    return cls('Suman', num1+num2) #stantiate new object
    using @classmethod

s1=School('Rohit',88)

print(s1.adding(5,10)) #here passing value to it
print(School.adding(5,8)) #we can also pass value like
this
```

```
<__main__.School object at 0x7f07102db850>
<__main__.School object at 0x7f07102db850>
> 
```

- Printing value of new object here



```
class School:
  def __init__(self,name,marks):
    self.name = name
    self.marks = marks
  def run(self):
    print(f'Run')
  @classmethod #here defining classmethod
  def adding(cls,num1,num2):
    return cls('Suman', num1+num2) #stantiate new object
    using @classmethod
s1=School('Rohit',88)
s2=School.adding(45,46) #new object
print(s2.name,s2.marks) #new object stantiated by
@classmethod and now I am printing its value
```

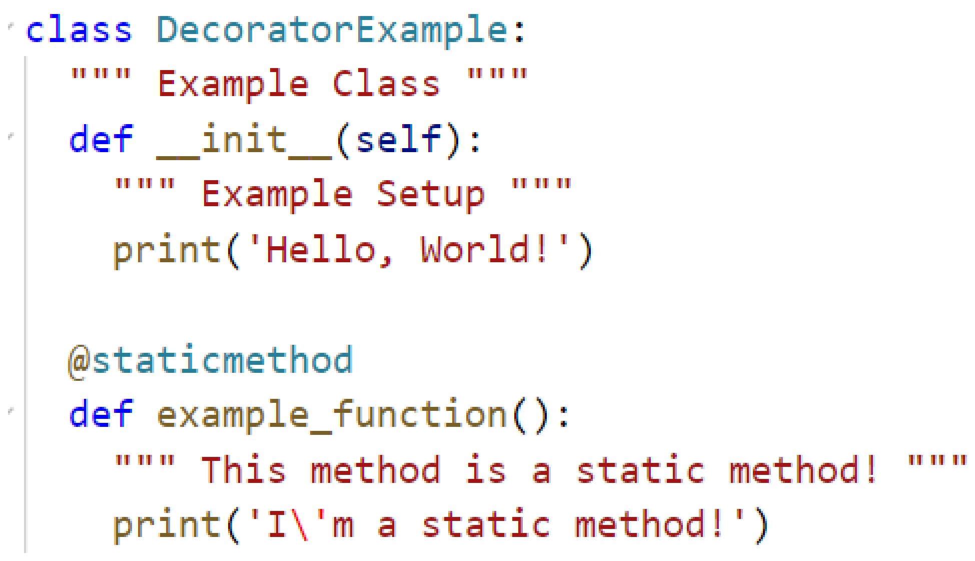
```
<__main__.School object at 0x7f4d88f3b9d0>
<__main__.School object at 0x7f4d88f3b9d0>
Suman 91
> 
```

STATIC METHOD

- It has same work as @classmethod but the only one difference that in @staticmethod, we do not have access to class (cls).

- Static methods are methods that are related to a class in some way, but don’t need to access any class-specific data. You don’t have to use **self**, and you don’t even need to instantiate an instance, you can simply call your method.

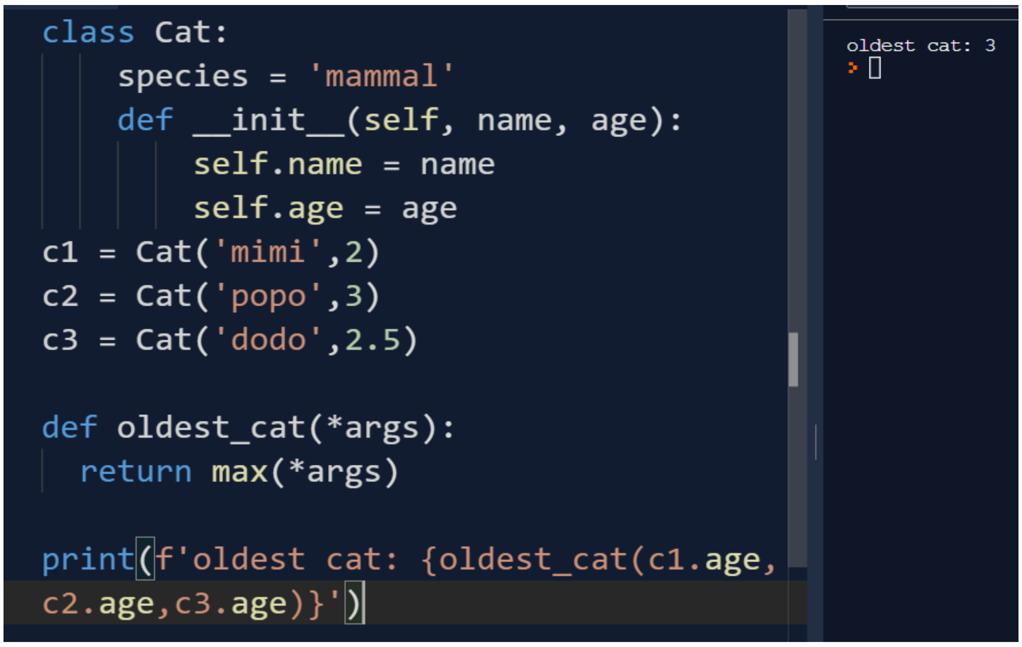
- Static methods are great for utility functions, which perform a task in isolation. They don’t need to (and cannot) access class data. They should be completely self-contained, and only work with data passed in as arguments. You may use a static method to add two numbers together, or print a given string.



```
class DecoratorExample:
  """ Example Class """
  def __init__(self):
    """ Example Setup """
    print('Hello, World!')

  @staticmethod
  def example_function():
    """ This method is a static method! """
    print('I\'m a static method!')
```

CODE EXERCISE

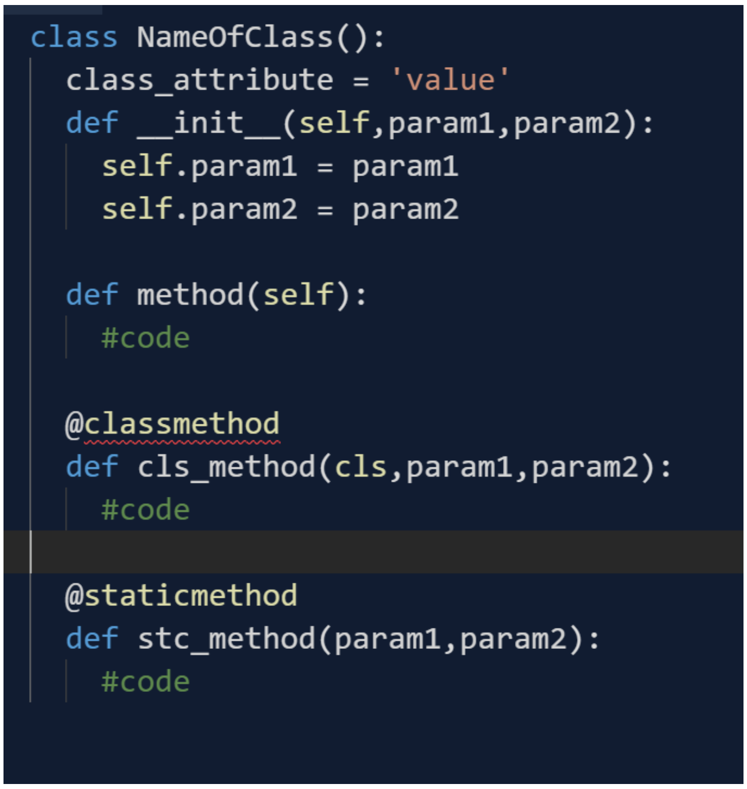
```
#Given the below class:
class Cat:
    species = 'mammal'
    def __init__ (self, name, age):
        self.name = name
        self.age = age
# 1 Instantiate the Cat object with 3 cats
# 2 Create a function that finds the oldest cat
# 3 Print out: "The oldest cat is x years old.". x will be the oldest cat age
by using the function in #2
```



```python
class Cat:
    species = 'mammal'
    def __init__(self, name, age):
        self.name = name
        self.age = age
c1 = Cat('mimi',2)
c2 = Cat('popo',3)
c3 = Cat('dodo',2.5)

def oldest_cat(*args):
  return max(*args)

print(f'oldest cat: {oldest_cat(c1.age,
c2.age,c3.age)}')
```

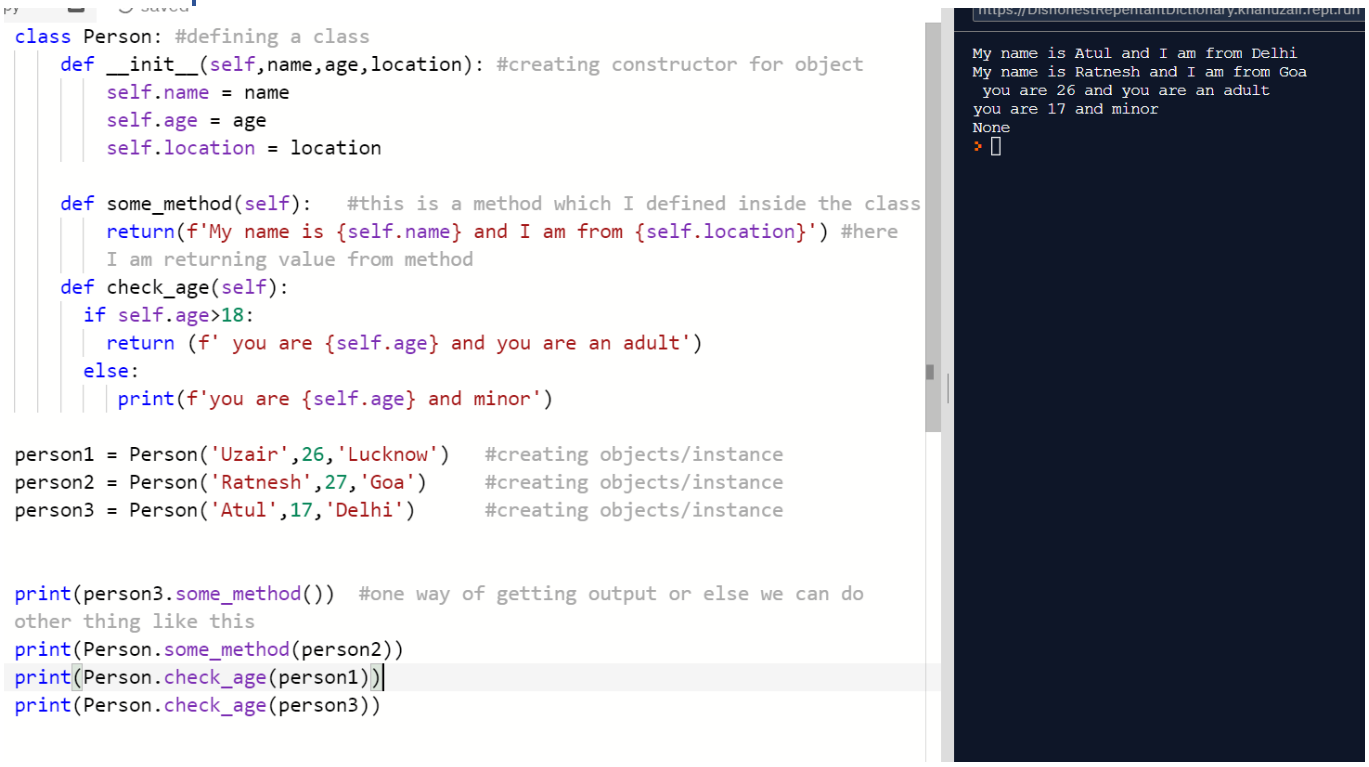
```
oldest cat: 3
```

Summary: General Syntax for class, @classmethod, @staticmethod



```python
class NameOfClass():
    class_attribute = 'value'
    def __init__(self,param1,param2):
        self.param1 = param1
        self.param2 = param2

    def method(self):
        #code

    @classmethod
    def cls_method(cls,param1,param2):
        #code

    @staticmethod
    def stc_method(param1,param2):
        #code
```

An example:



```python
class Person: #defining a class
    def __init__(self,name,age,location): #creating constructor for object
        self.name = name
        self.age = age
        self.location = location

    def some_method(self):   #this is a method which I defined inside the class
        return(f'My name is {self.name} and I am from {self.location}') #here
        I am returning value from method
    def check_age(self):
      if self.age>18:
        return (f' you are {self.age} and you are an adult')
      else:
         print(f'you are {self.age} and minor')

person1 = Person('Uzair',26,'Lucknow')     #creating objects/instance
person2 = Person('Ratnesh',27,'Goa')       #creating objects/instance
person3 = Person('Atul',17,'Delhi')        #creating objects/instance


print(person3.some_method())  #one way of getting output or else we can do
other thing like this
print(Person.some_method(person2))
print(Person.check_age(person1))
print(Person.check_age(person3))
```

```
My name is Atul and I am from Delhi
My name is Ratnesh and I am from Goa
 you are 26 and you are an adult
you are 17 and minor
None
> 
```

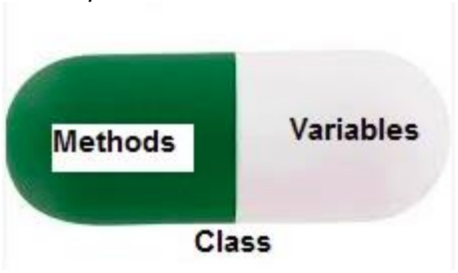
1. **Instance Methods:** The most common method type. Able to access data and properties unique to each instance.
2. **Static Methods:** Cannot access anything else in the class. Totally self-contained code.
3. **Class Methods:** Can access limited methods in the class. Can modify class specific details.

ENCAPSULATION IN PYTHON

- Encapsulation is binding of data and function-that manipulate the data.

- Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those type of variables are known as **private variable**.

- A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.



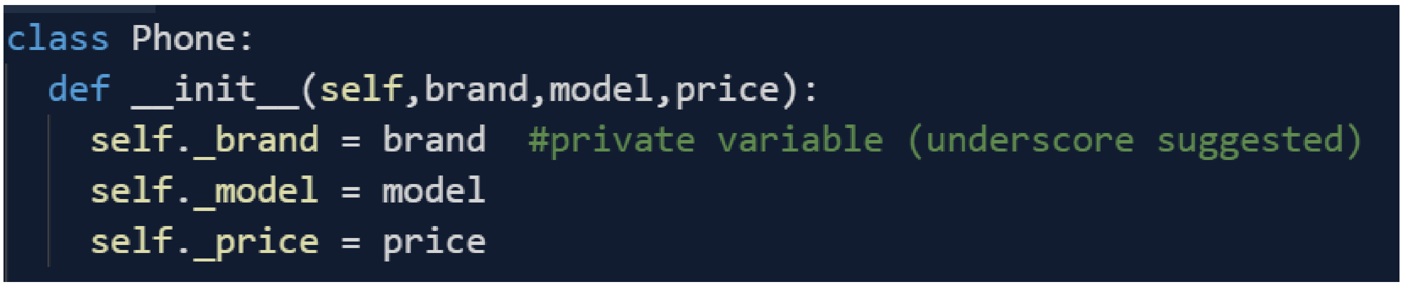
- Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section". As using encapsulation also hides the data. In this example, the data of any of the sections like sales, finance or accounts are hidden from any other section.

ABSTRACTION IN PYTHON

- Abstraction means hiding the complexity and only showing the essential features of the object. So, in a way, Abstraction means hiding the real implementation and we, as a user, knowing only how to use it.

- Abstraction in Python is achieved by using abstract classes and interfaces.

- Underscore in the starting of variable name suggests that this variable is private.



```python
class Phone:
  def __init__(self,brand,model,price):
    self._brand = brand  #private variable (underscore suggested)
    self._model = model
    self._price = price
```

- Double underscore __ suggests Dunder method.

- An abstract class is a class that generally provides incomplete functionality and contains one or more abstract methods. Abstract methods are the methods that generally don’t have any implementation, it is left to the sub classes to provide implementation for the abstract methods.

- An interface should just provide the method names without method bodies. Subclasses should provide implementation for all the methods defined in an interface. Note that in Python there is no support for creating interfaces explicitly, you will have to use abstract class. In Python you can create an interface using abstract class. If you create an abstract class which contains only abstract methods that acts as an interface in Python.

INHERITENCE

- Inheritance allows us to define a class that inherits all the methods and properties from another existing class.
- Parent class is the class being inherited from, also called base class.
- Child class is the class that inherits from another class, also called derived class.

Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

Example

Create a class named `Person`, with `firstname` and `lastname` properties, and a `printname` method:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")
x.printname()
```

Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

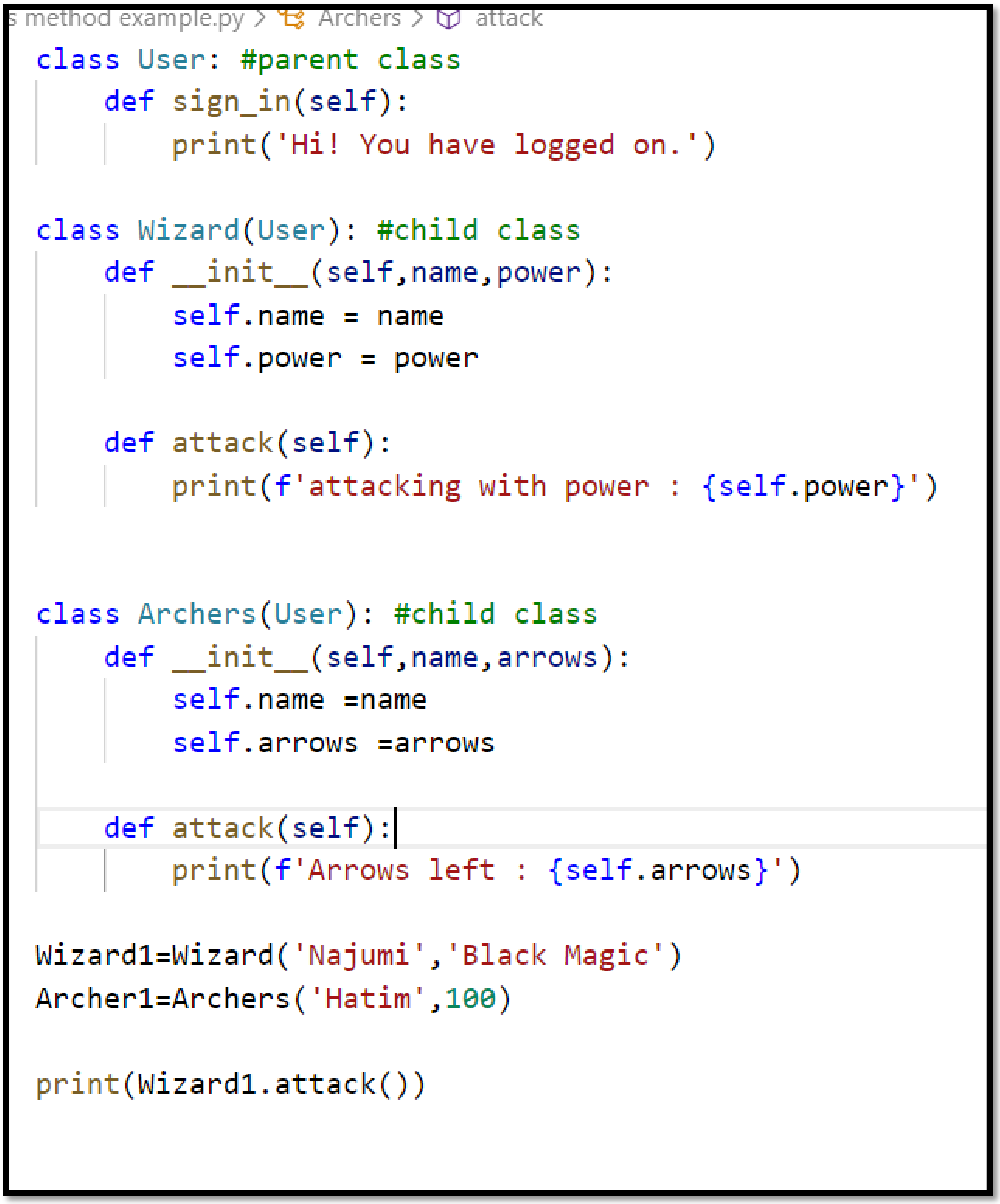
Example

Create a class named `Student`, which will inherit the properties and methods from the `Person` class:

```
class Student(Person):
  pass
```

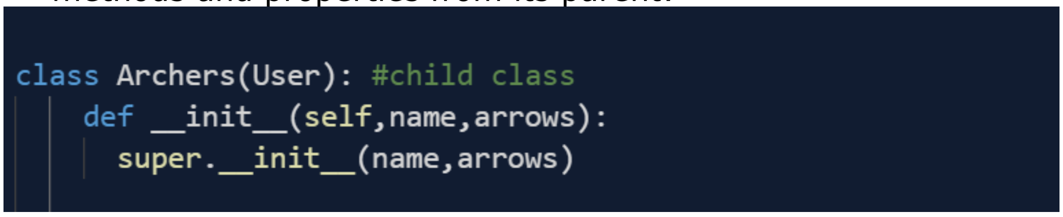
- The `__init__ ()` function is called automatically every time the class is being used to create a new object.
- When you add the `__init__ ()` function, the child class will no longer inherit the parent's `__init__ ()` function.
- The child's `__init__ ()` function **overrides** the inheritance of the parent's `__init__ ()` function.

- To keep the inheritance of the parent's `__init__ ()` function, add a call to the parent's `__init__ ()` function.



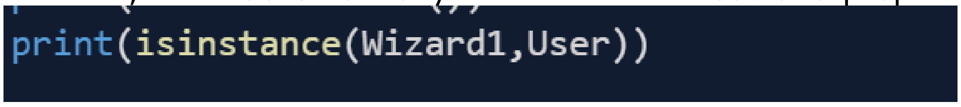
```
s method example.py > Archers > attack
class User: #parent class
    def sign_in(self):
        print('Hi! You have logged on.')

class Wizard(User): #child class
    def __init__(self,name,power):
        self.name = name
        self.power = power

    def attack(self):
        print(f'attacking with power : {self.power}')


class Archers(User): #child class
    def __init__(self,name,arrows):
        self.name =name
        self.arrows =arrows

    def attack(self):
        print(f'Arrows left : {self.arrows}')

Wizard1=Wizard('Najumi','Black Magic')
Archer1=Archers('Hatim',100)

print(Wizard1.attack())
```

- Python also has a **super ()** function that will make the child class inherit all the methods and properties from its parent:



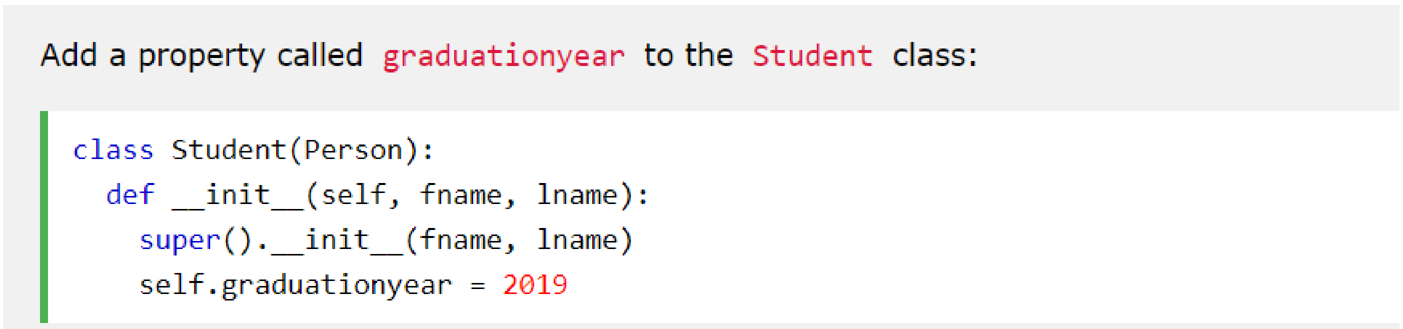
```
class Archers(User): #child class
    def __init__(self,name,arrows):
        super.__init__(name,arrows)
```

- By using the `super ()` function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.



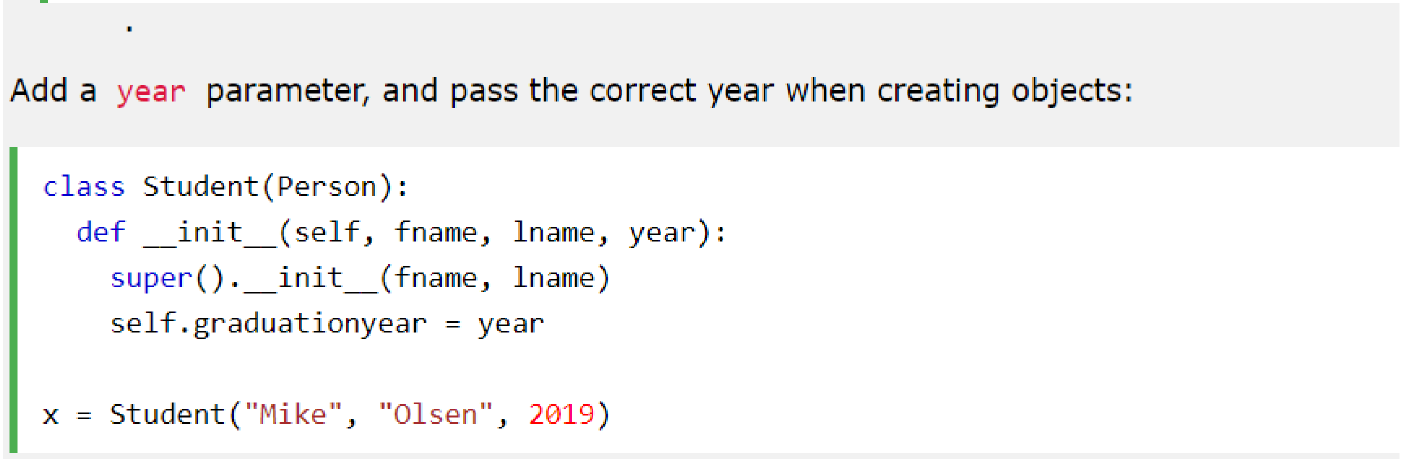
```python
print(isinstance(Wizard1,User))
```

- The `isinstance ()` function returns `True` if the specified object is of the specified type, otherwise `False`.
- isinstance () is used for checking whether an object belongs to particular class or not.



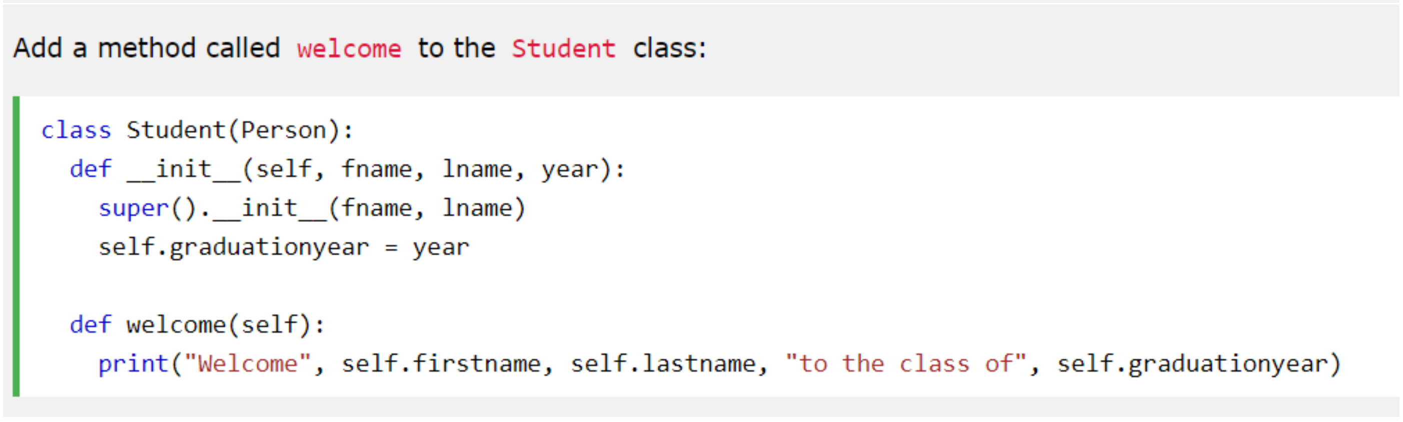
Add a property called `graduationyear` to the `Student` class:

```python
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```



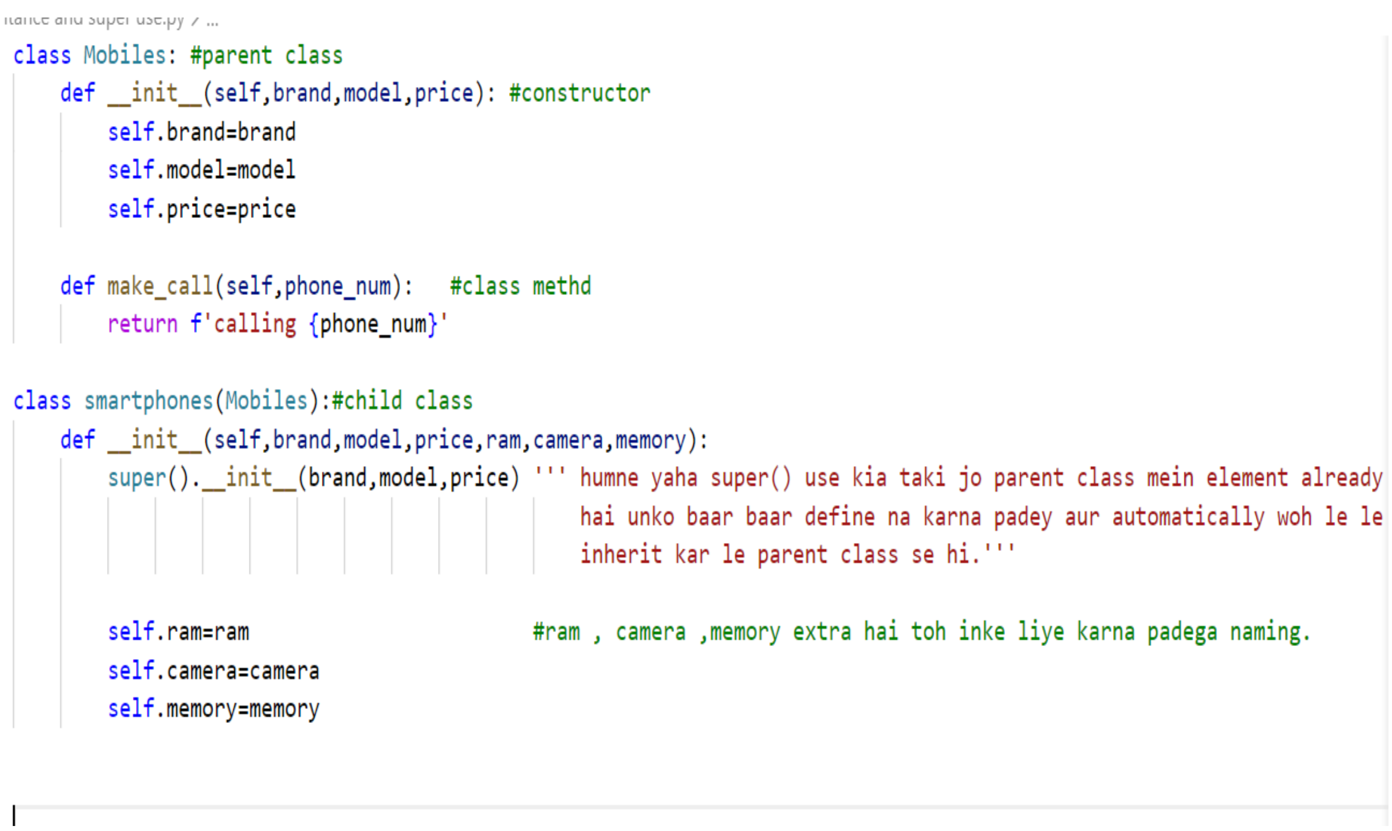
Add a `year` parameter, and pass the correct year when creating objects:

```python
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
```



Add a method called `welcome` to the `Student` class:

```python
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```



itance and super use.py > ...

```python
class Mobiles: #parent class
    def __init__(self,brand,model,price): #constructor
        self.brand=brand
        self.model=model
        self.price=price

    def make_call(self,phone_num):   #class methd
        return f'calling {phone_num}'

class smartphones(Mobiles):#child class
    def __init__(self,brand,model,price,ram,camera,memory):
        super().__init__(brand,model,price) ''' humne yaha super() use kia taki jo parent class mein element already
                                                hai unko baar baar define na karna padey aur automatically woh le le
                                                inherit kar le parent class se hi.'''

        self.ram=ram                           #ram , camera ,memory extra hai toh inke liye karna padega naming.
        self.camera=camera
        self.memory=memory
```

MULTILEVEL INHERITANCE

```
class Mobiles: #parent class
    def __init__(self,brand,model,price): #constructor
        self.brand=brand
        self.model=model
        self.price=price

    def make_call(self,phone_num):   #class methd
        return f'calling {phone_num}'

    def full_name(self):
        return f'{self.brand} {self.model}'

class smartphones(Mobiles):#child class
    def __init__(self,brand,model,price,ram,camera,memory):
        super().__init__(brand,model,price)
        self.ram=ram
        self.camera=camera
        self.memory=memory

class PremiumPhones(smartphones): #child class
    '''this is multilevel inheritance, 1st we inherited smartphones class from
Mobile class and now we inherited
    premium phone class from smartphone class '''
    def __init__(self,brand,model,price,ram,camera,memory,battery,processor):
        super().__init__(brand,model,price,ram,camera,memory)
        self.battery= battery
        self.processor= processor

Premium1=PremiumPhones('Apple','X10',65000,'6 GB','48P','128 GB','5000mAh','Mac
')
print(Premium1.__dict__)
print(Premium1.full_name())
```

OUTPUT

```
{'brand': 'Apple', 'model': 'X10', 'price': 65000, 'ram': '6 GB', 'camera': '48P',
'memory': '128 GB', 'battery': '5000mAh', 'processor': 'Mac'}

Apple X10
```

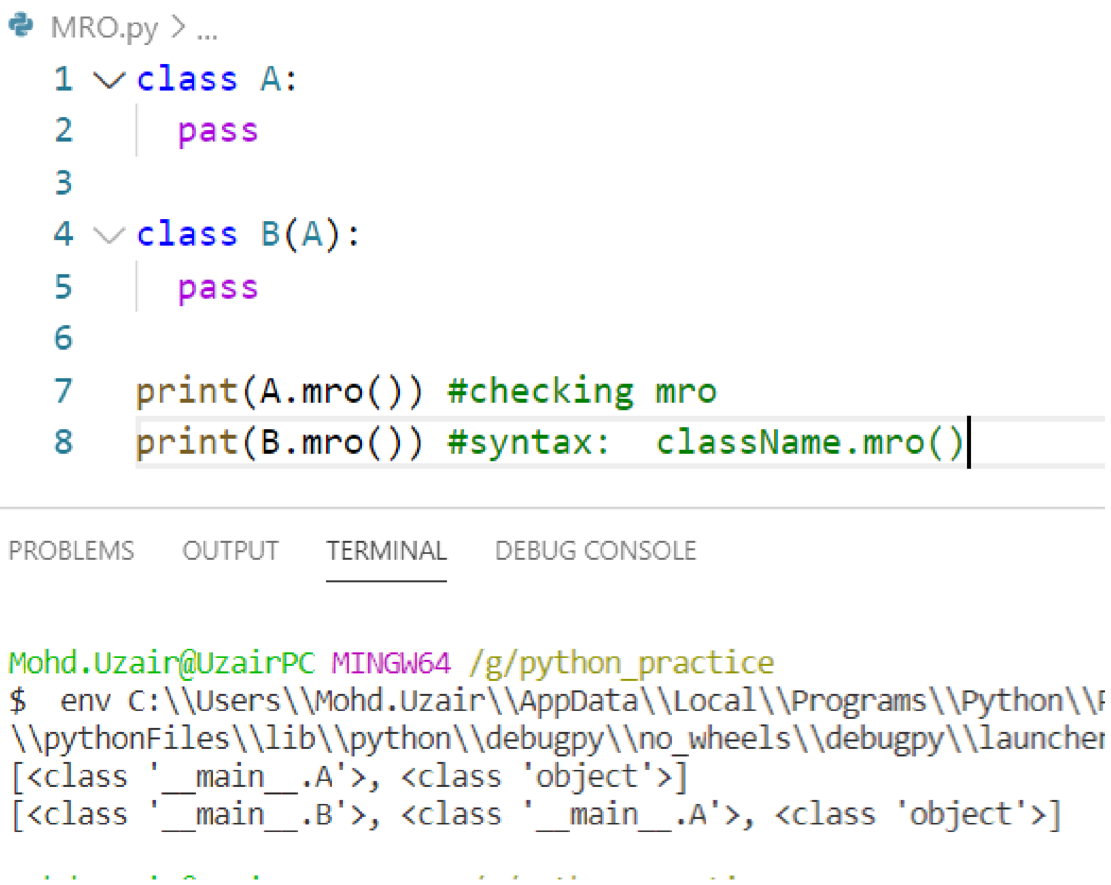
METHOD OVERRIDING

- Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to **override** the method in the super-class/parent class.

```
- class Mobiles: #parent class
-     def __init__(self,brand,model,price): #constructor
-         self.brand=brand
-         self.model=model
-         self.price=price
-
-     def make_call(self,phone_num):   #class methd
-         return f'calling {phone_num}'
-
-     def full_name(self):
-         return f'{self.brand} {self.model}'
-
- class smartphones(Mobiles):#child class
-     def __init__(self,brand,model,price,ram,camera,memory):
-         super().__init__(brand,model,price)
-         self.ram=ram
-         self.camera=camera
-         self.memory=memory
-
-     def full_name(self):
-         print(f'Get {self.brand} {self.model} at discounted price of {sel
  f.price}')
-
-     '''ABOVE METHOD WILL BE CALLED FOR SMARTPHONE CLASS INSTEAD OF PARENT
   CLASS METHOD. THIS IS METHOD OVERRIDING'''
-
- class PremiumPhones(smartphones): #child class
-     def __init__(self,brand,model,price,ram,camera,memory,battery,process
  or):
-         super().__init__(brand,model,price,ram,camera,memory)
-         self.battery= battery
-         self.processor= processor
-
- Premium1=PremiumPhones('Apple','X10',65000,'6 GB','48P','128 GB','5000mAh
  ','Mac')
- print(Premium1.full_name())
```

METHOD RESOLUTION ORDER (MRO)

- In python, method resolution order defines the order in which the base classes are searched when executing a method. First, the method or attribute is searched within a class and then it follows the order we specified while inheriting. MRO also called Linearization of a class.



```python
class A:
    pass

class B(A):
    pass

print(A.mro()) #checking mro
print(B.mro()) #syntax:  className.mro()
```

```
Mohd.Uzair@UzairPC MINGW64 /g/python_practice
$  env C:\\Users\\Mohd.Uzair\\AppData\\Local\\Programs\\Python\\
\\pythonFiles\\lib\\python\\debugpy\\no_wheels\\debugpy\\launcher
[<class '__main__.A'>, <class 'object'>]
[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```

More details at : http://www.srikanthtechnologies.com/blog/python/mro.aspx

POLYMORPHISM

- It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios.
- We know that the `+` operator is used extensively in Python programs. But it does not have a single usage.
- For integer data types, `+` operator is used to perform arithmetic addition operation.

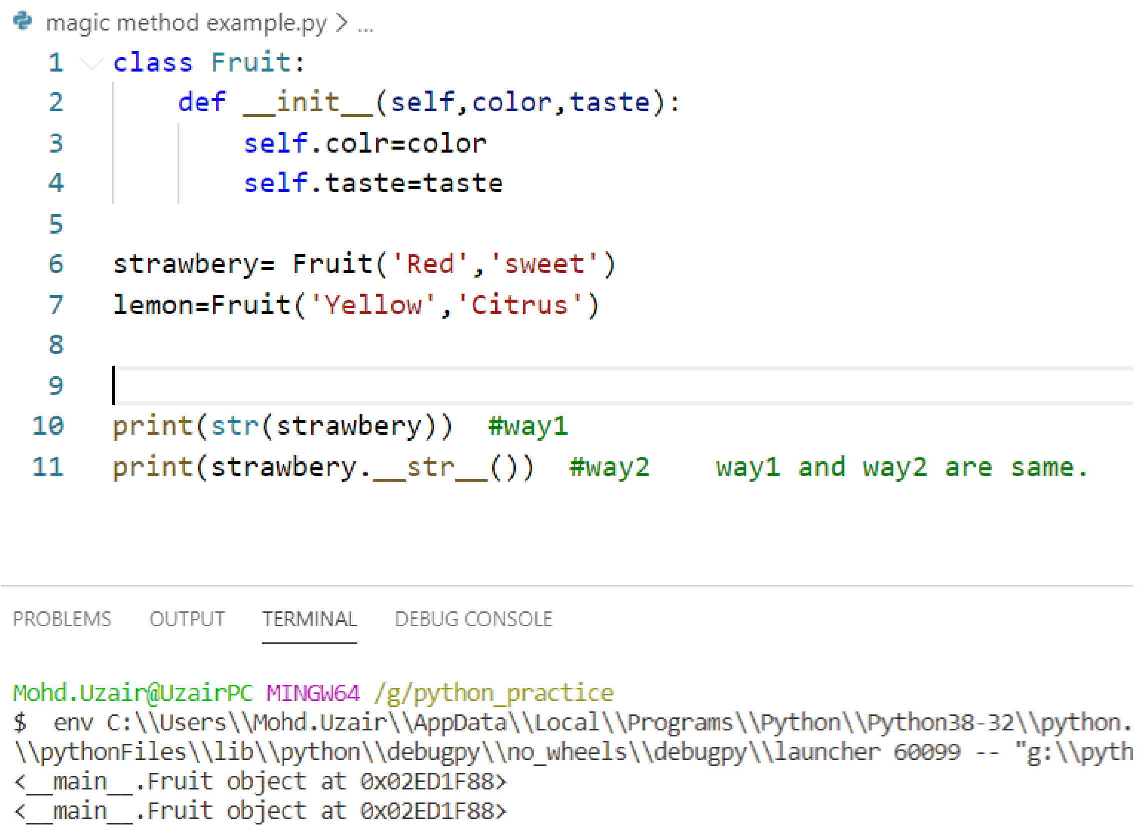
```python
 num1 = 1
num2 = 2
print(num1+num2)
```

- Similarly, for string data types, `+` operator is used to perform concatenation.

```python
str1 = "Python"
str2 = "Programming"
print (str1+" "+str2)
```

DUNDER METHOD/MAGIC METHOD

- Dunder or magic methods in Python are the methods having two prefix and suffix underscores in the method name. Dunder here means "Double Under (Underscores)". These are commonly used for operator overloading.
- For example, __inti__, __add__, __repr__, __len__, __str__

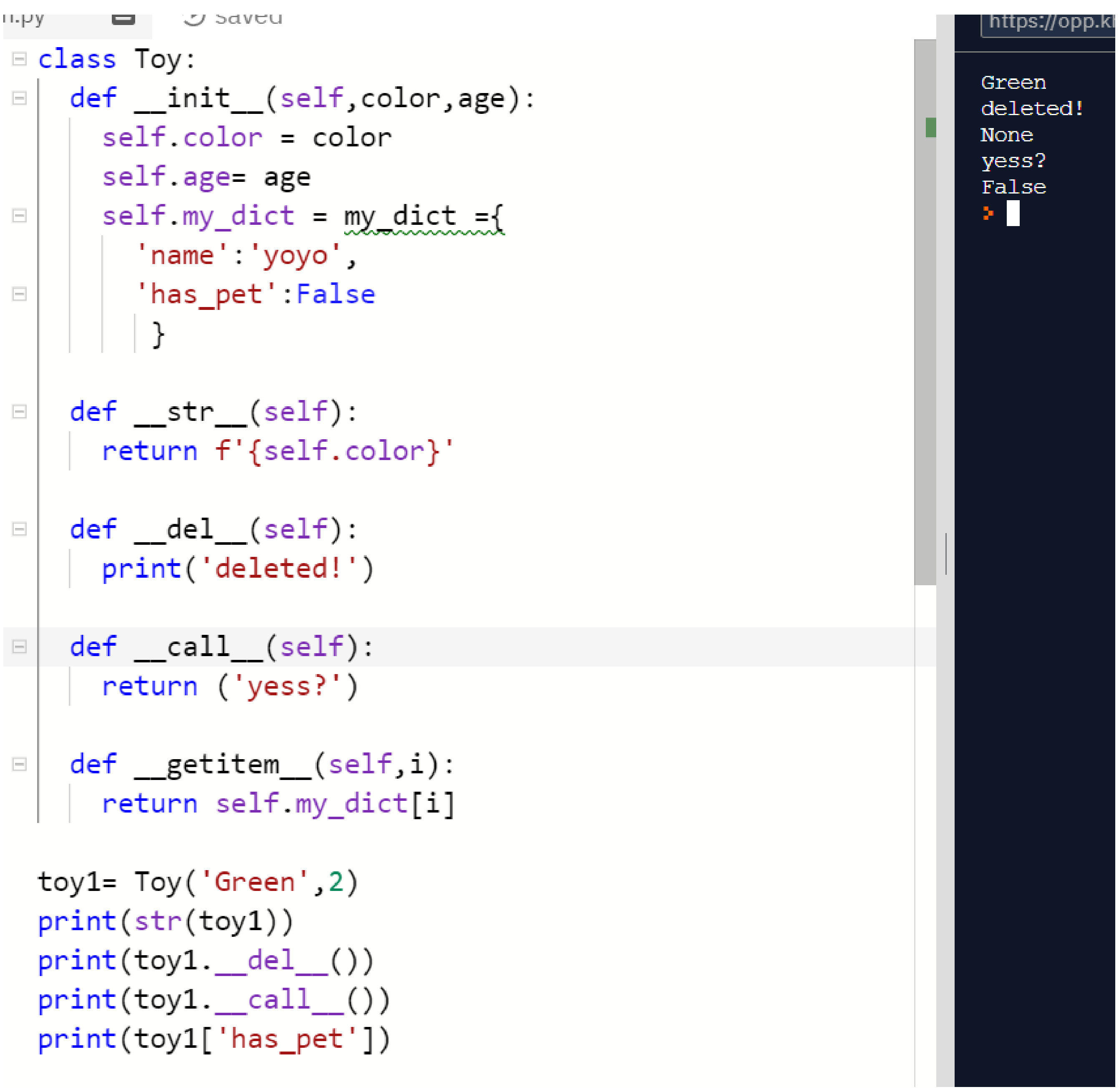


magic method example.py > ...

```python
class Fruit:
    def __init__(self,color,taste):
        self.colr=color
        self.taste=taste

strawbery= Fruit('Red','sweet')
lemon=Fruit('Yellow','Citrus')


print(str(strawbery))  #way1
print(strawbery.__str__())  #way2    way1 and way2 are same.
```

```
Mohd.Uzair@UzairPC MINGW64 /g/python_practice
$  env C:\\Users\\Mohd.Uzair\\AppData\\Local\\Programs\\Python\\Python38-32\\python.
\\pythonFiles\\lib\\python\\debugpy\\no_wheels\\debugpy\\launcher 60099 -- "g:\\pyth
<__main__.Fruit object at 0x02ED1F88>
<__main__.Fruit object at 0x02ED1F88>
```

- We can modify Dunder method to work out our code as we want.



n.py saved

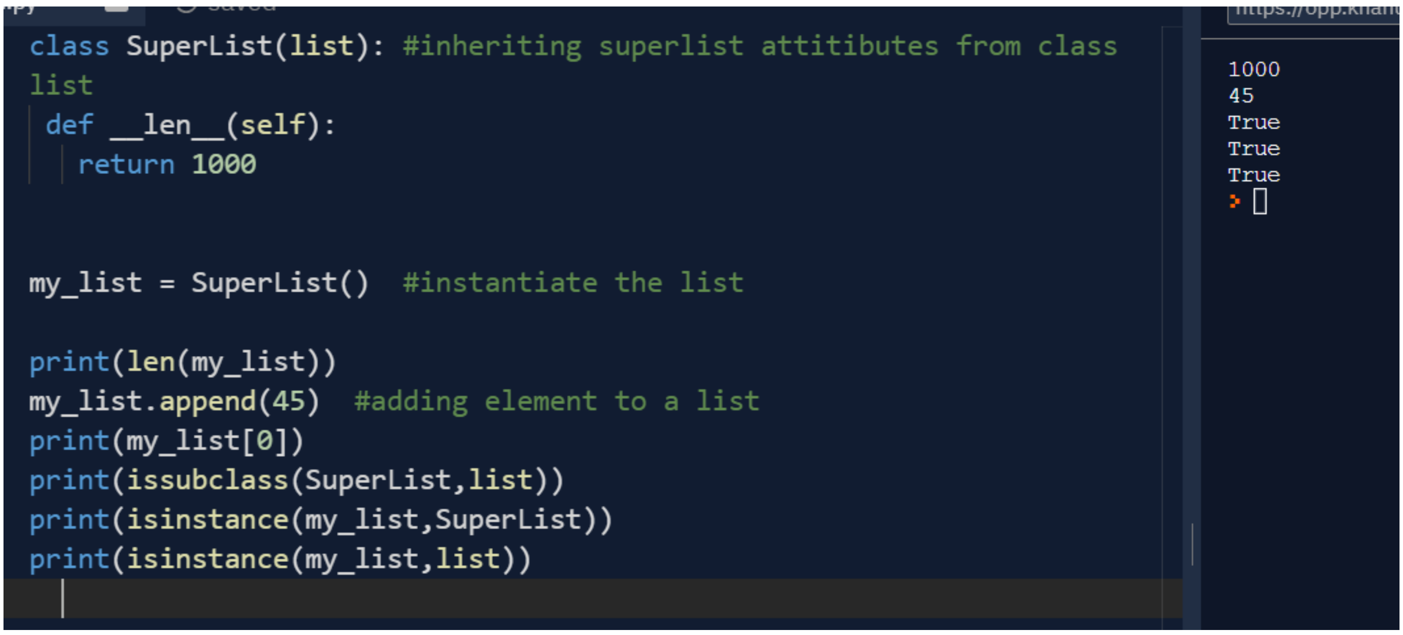
```python
class Toy:
  def __init__(self,color,age):
    self.color = color
    self.age= age
    self.my_dict = my_dict ={
      'name':'yoyo',
      'has_pet':False
      }

  def __str__(self):
    return f'{self.color}'

  def __del__(self):
    print('deleted!')

  def __call__(self):
    return ('yess?')

  def __getitem__(self,i):
    return self.my_dict[i]

toy1= Toy('Green',2)
print(str(toy1))
print(toy1.__del__())
print(toy1.__call__())
print(toy1['has_pet'])
```

https://opp.k

```
Green
deleted!
None
yess?
False
> 
```

Code Exercise:

```
Create a class named SuperList.
Using Dunder method __len__ modify it.
Create an object of class SuperList.
Add element to object of list created.
```



```python
class SuperList(list): #inheriting superlist attitibutes from class
list
  def __len__(self):
    return 1000


my_list = SuperList()  #instantiate the list

print(len(my_list))
my_list.append(45)  #adding element to a list
print(my_list[0])
print(issubclass(SuperList,list))
print(isinstance(my_list,SuperList))
print(isinstance(my_list,list))
```

```
1000
45
True
True
True
```