PYTHON OBJECT ORIENTED PROGRAMMING

EXERCISES BECOME A PRO DEVELOPER

EDCORNER LEARNING

Python
Object Oriented
Programming Exercises   Become a
Pro Developer

Python Object Oriented Programming Exercises Become a Pro Developer

Edcorner Learning

Table of Contents

Introduction

Python is a general-purpose interpreted, interactive, object- oriented, and a powerful programming language with dynamic semantics. It is an easy language to learn and become expert. Python is one among those rare languages that would claim to be both easy and powerful. Python's elegant syntax and dynamic typing alongside its interpreted nature makes it an ideal language for scripting and robust application development in many areas on giant platforms.

Python helps with the modules and packages, which inspires program modularity and code reuse. The Python interpreter and thus the extensive standard library are all available in source or binary form for free of charge for all critical platforms and can be freely distributed. Learning Python doesn't require any pre- requisites. However, one should have the elemental understanding of programming languages.

This Book consist of Indepth Python OOPS Concepts and 73 python Object Oriented Programming coding exercises to practice different topics.

In each exercise we have given the exercise coding statement you need to complete and verify your answers. We also attached our own input output screen of each exercise and their solutions.

Learners can use their own python compiler in their system or can use any online compilers available.

We have covered all level of exercises in this book to give all the learners a good and efficient Learning method to do hands on python different scenarios.

1 Overview of Python OOPS

Python's OOPS ideas are extremely similar to how we solve problems in the real world by writing programs. The most common method in programming is to solve any problem by creating objects.

The term "object-oriented programming" refers to this method. It is simpler to write and understand code thanks to object-oriented programming, which correlates our instructions with difficulties encountered in the actual world. They represent actual people, businesses, and employees as "software objects" containing "data" and "functions" that can be performed on them.

2 What in Python is OOPS?

Object Oriented Programming System is referred to as OOPS in programming. To create a programme using classes and objects is a paradigm for or methodology for programming. Every entity is treated as an object by OOPS.

Python's object-oriented programming is focused on objects. Any OOPS-based programming that is developed solves our problem but takes the shape of objects. For a particular class, we are free to produce as many objects as we choose.

What then are things? Anything with characteristics and certain behaviours is an object. The terms "variables of the object" and "functions of the object" are frequently used to describe an object's properties and behaviours, respectively. Objects might be conceptual or actual objects.

Let's say that a pen exists in the real world. A pen's characteristics are its colour and type (gel pen or ball pen). Additionally, the pen's behaviour may include the ability to write, draw, etc.

A logical object might be any file on your computer. Files can retain data, be downloaded, shared, and have other behaviours. They have properties like file name, file location, and file size.

3 Some of the Major Advantages of OOPS are:

- Writing readable and reuseable codes helps them reduce the amount of redundant code (using inheritance).
- Because they are so closely related to real-world circumstances, they are simpler to visualise. For instance, the ideas of objects, inheritance, and abstractions have a connection to real-world situations.
- Each object in oops represents a different section of the code and has its own logic and data for inter-object communication. Therefore, there are no difficulties with the code.

4 Difference between procedural and object-oriented programming

Python adheres to four different programming paradigms, did you know that?

Imperative, functional, procedural, and object-oriented programming are some of them.

Procedural Oriented Programming (POP) and Object-Oriented Programming are two of the most significant programming paradigms in Python (OOP). Let's start.

1. What Are They, First?

Let's examine the methodology each paradigm employs:

Suppose you want to prepare Maggie for dinner! Then you follow a series of stages, including —

Warm up some water in a pan.

Include Maggie in it

add masala

Prepare and serve.

Similar to this, POP needs to follow a specific set of steps in order to function. POP is made up of functions. There are different parts of a POP programme called functions, each of which is responsible for a different job. The functions are set up in a particular order, and the programme control happens one step at a time.

OOP: Object-oriented programming. The programme is divided into items. These objects are the entities that blend the traits and workings of things found in the real world.

2. Which Locations Do They Prefer?

POP is only appropriate for little jobs. Because the code becomes more difficult as the program's duration increases and becomes bloated with functions, Debugging gets even more challenging.

OOP works well for bigger issues. Recursion can be used to make the code reusable, which makes it simpler and cleaner.

3. Which Offers Greater Security?

POP offers the functions along with all the data, which makes it less secure. Thus, our data are not secret. If you wish to protect your login passwords or any other secret information, POP is not a recommended solution!

OOP offers security by data hiding, making it more secure. Encapsulation is a unique idea in OOP that grants it the ability to hide data (we will read about this further).

4. Programming approach

POP adheres to programming from the top down. The top-down method of programming focuses on dissecting a complex issue into manageable units of code. The minor portions of the problems are then resolved.

OOPS principles adhere to the Bottom-up method of programming. The Bottom-Up method prioritises resolving the smaller issues at their most fundamental level before incorporating them into a comprehensive and entire solution.

5. Utilization of Access Modifiers:

When applying the ideas of inheritance, access specifiers or access modifiers are used in Python to restrict the access of class variables and class methods outside of the class. The keywords Public, Private, and Protected can be used to do this.

No access modifiers like "public," "private," or "protected" are used in POP. To employ the aforementioned modifiers, POP lacks the concepts of classes and inheritance. Modifiers for access are supported by OOP. They understand inheritance, so they can use terms like "public," "private," or "protected."

Note:

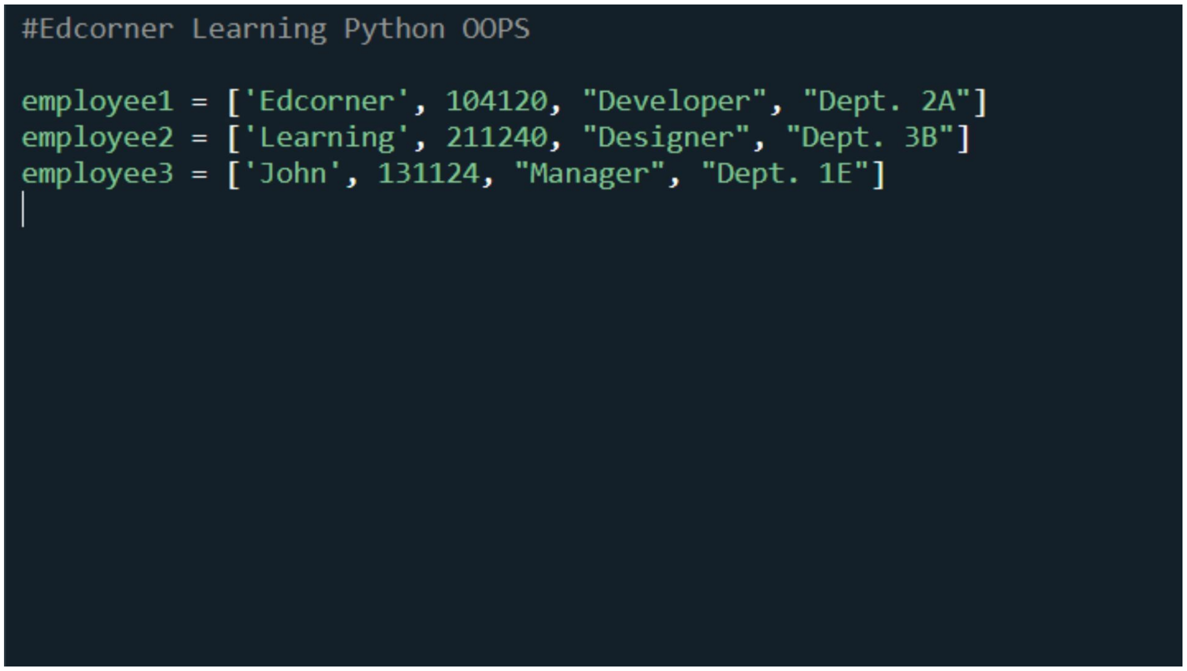
The access modifiers in Python come in quite handy when working with inheritance ideas. Class methods can make use of this idea as well.

5 Python's Class and Objects

Let's say you want to keep track of how many books you own. You can easily do that by utilising a variable. Also possible is to compute the sum of five numbers and save the result in a variable.

Simple values are intended to be stored in a variable using primitive data structures like numbers, characters, and lists. Consider your name, the square root of a number, or the quantity of marbles (say).

But what if you need to keep a record of every employee in your business? For instance, if you try to put all of your employees in a list, you can later become confused about which index of the list corresponds to which individual details (e.g. which is the name field, or the empID or age etc.)



```
#Edcorner Learning Python OOPS

employee1 = ['Edcorner', 104120, "Developer", "Dept. 2A"]
employee2 = ['Learning', 211240, "Designer", "Dept. 3B"]
employee3 = ['John', 131124, "Manager", "Dept. 1E"]
```

Even if you try to keep them in a dictionary, the entire codebase will eventually become too complicated to manage. Therefore, we employ Python Classes in these situations.

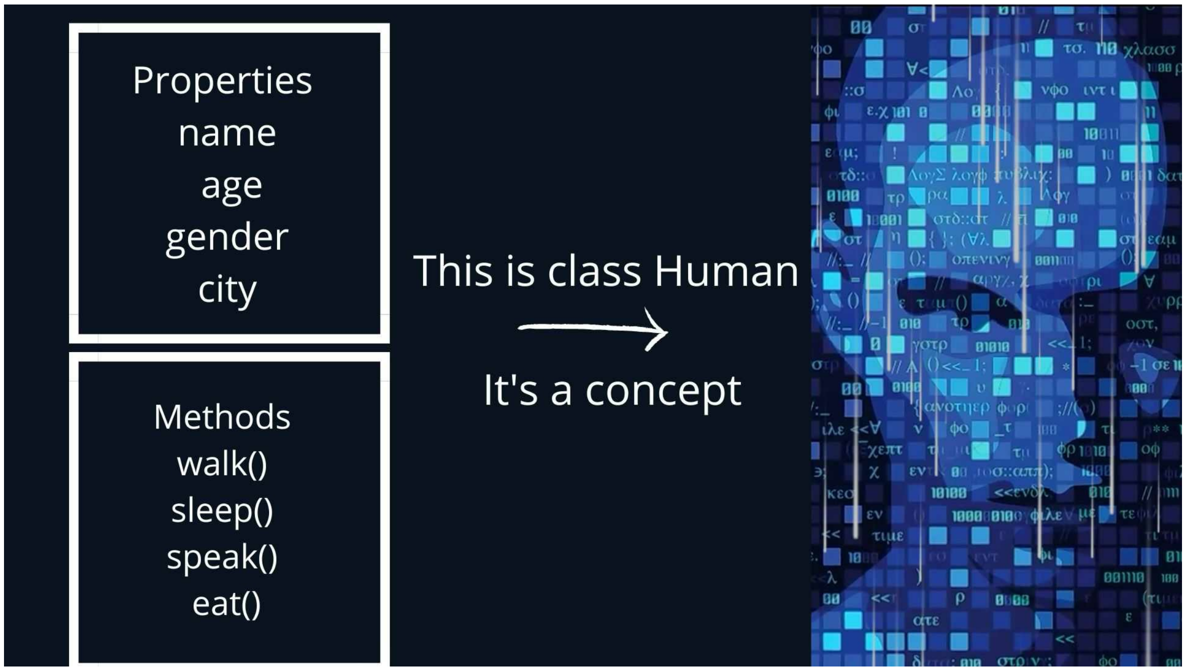
To create user-defined data structures in Python, use a class. Classes set up functions, called "methods," that describe how an object made from a class can behave and what actions it can take. Classes and objects are the main topics covered by Python's OOPS ideas.

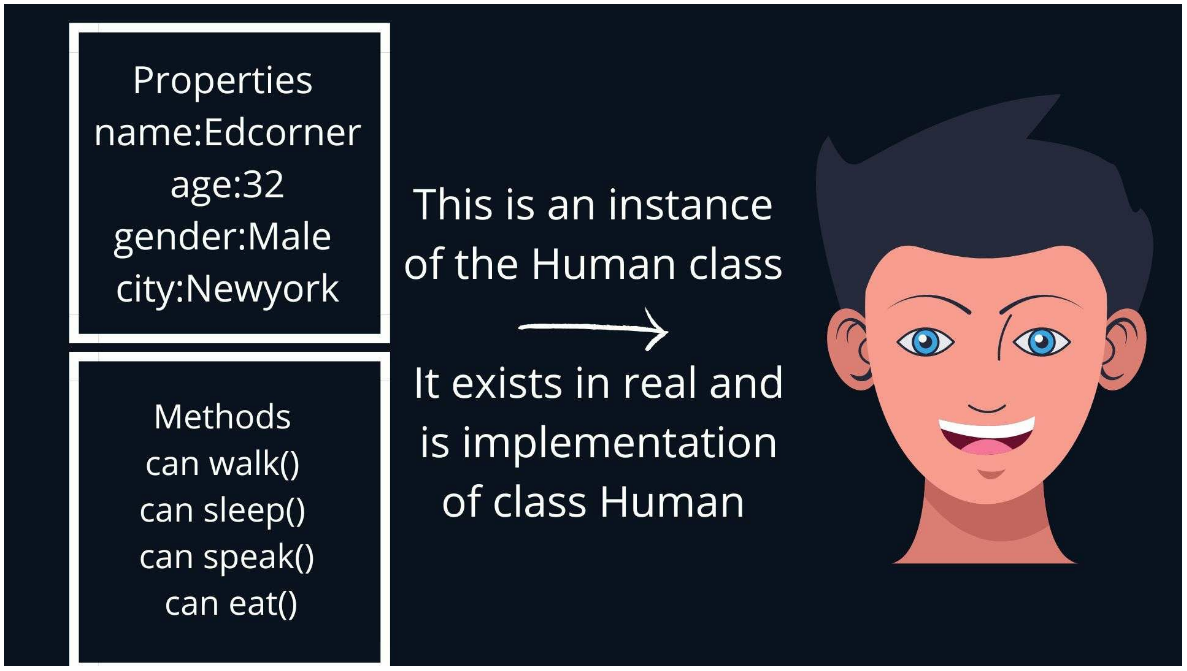
Classes simplify the code by avoiding complicated codebases. It accomplishes this by developing a model or design for how everything ought to be defined. Any object that derives from the class should have the characteristics or capabilities specified by this clause.

Note:

Simple structural definition is all that a class does. It makes no specific reference to anything or anyone. For instance, the class HUMAN has attributes like name, age, gender, and city. It does not identify any particular HUMAN, but it does describe the qualities and capabilities that a HUMAN or an object of the HUMAN class ought to have.

The term "object" refers to a class instance. It is the class's actual implementation and is real.


Properties
name
age
gender
city
Methods
walk()
sleep()
speak()
eat()
This is class Human
It's a concept



A collection of data (variables) and methods (functions) that access the data is referred to as an object. It is the actual class implementation.

Take this example where Human is a class. This class only serves as a template for what Human should be, not an actual implementation. You could argue that the "Human" class only makes logical sense.

However, "Edcorner" is a Human class object (please refer the image given above for understanding). It follows that Edcorner was built using the blueprint for the Human class, which holds the actual data. Unlike "Human," "Edcorner" is a real person (which just exists logically). He is a genuine person who embodies all the characteristics of the class Human, including having a name, being male, being 32 years old, and residing in Newyork. Additionally, Edcorner uses all of the methods included in the Human class; for example, Edcorner can walk, speak, eat, and sleep.

And many humans can be produced utilising the class Human blueprint. For instance, by leveraging objects and the blueprint for the class Human, we could generate many more persons.

Short Tip:

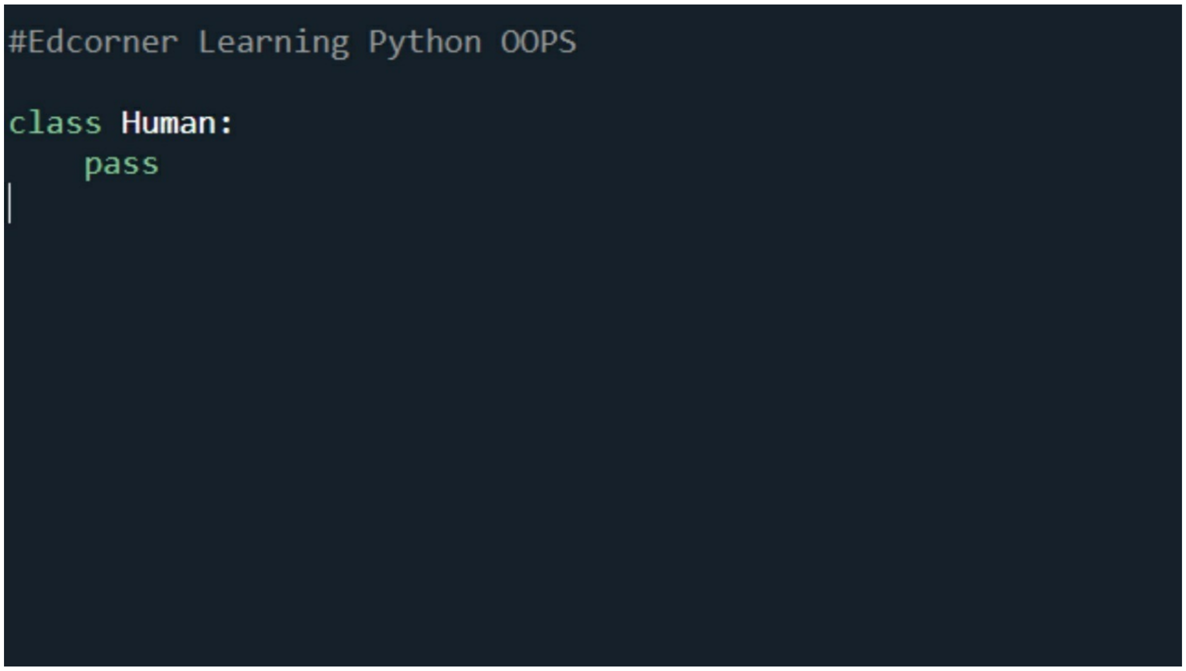
class = draught (suppose an architectural drawing). The Object is a real item that was created using the "blueprint" (suppose a house). An instance is a representation of the item that is virtual but not a true copy.

No memory is allotted to a class when it is defined; just the object's blueprint is produced. Memory allocation only takes place during object or instance creation. The actual data or information is contained in the object or instance.

6 What is a Class Definition in Python?

Python classes are defined using the word class, which is then followed by the class name and a colon.

Syntax:



```
#Edcorner Learning Python OOPS

class Human:
    pass
```

Below the class definition, indented code is regarded as a component of the class body.

'pass' is frequently used as a stand-in for code whose implementation we can forego for the moment. We may execute the Python code without throwing an error by using the "pass" keyword.

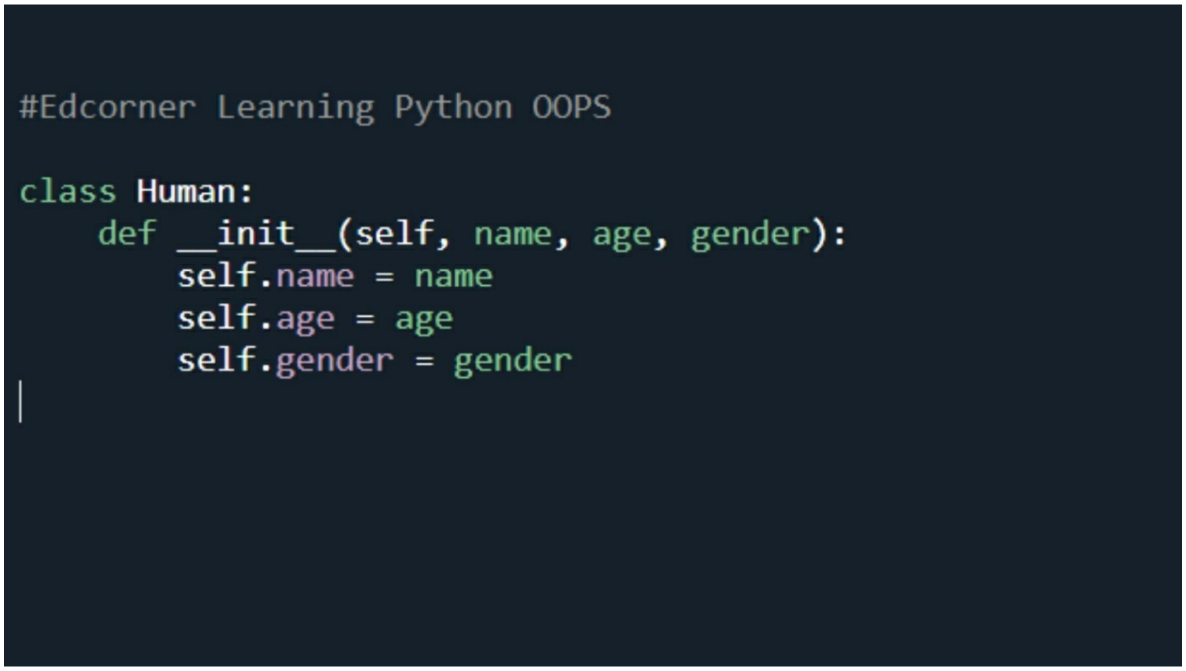
7 An __init__ method: What is it?

In a method named init, the qualities that all Human objects must possess are specified (). When a new Human object is formed, __init__() assigns the values we supply inside the object's properties to set the object's initial state. In other words, each new instance of the class is initialised via __init__(). Any number of parameters can be passed to __init__(), but self is always the first parameter.

The self parameter contains a reference to the active class instance. This means that we can access the data of an object's variables by using the self argument, which corresponds to the address of the current object of a class.

Since this self points to the address of each individual object and returns its corresponding value, even if there are 1000 instances (objects) of a class, we can always access each of their unique data.

Let's look at how the Human class defines __init__():



```
#Edcorner Learning Python OOPS

class Human:
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
```

We use the self variable three times in the body of. init__() for the following purposes:

self.name = 'name' creates the name attribute and sets the name parameter's value as its value.

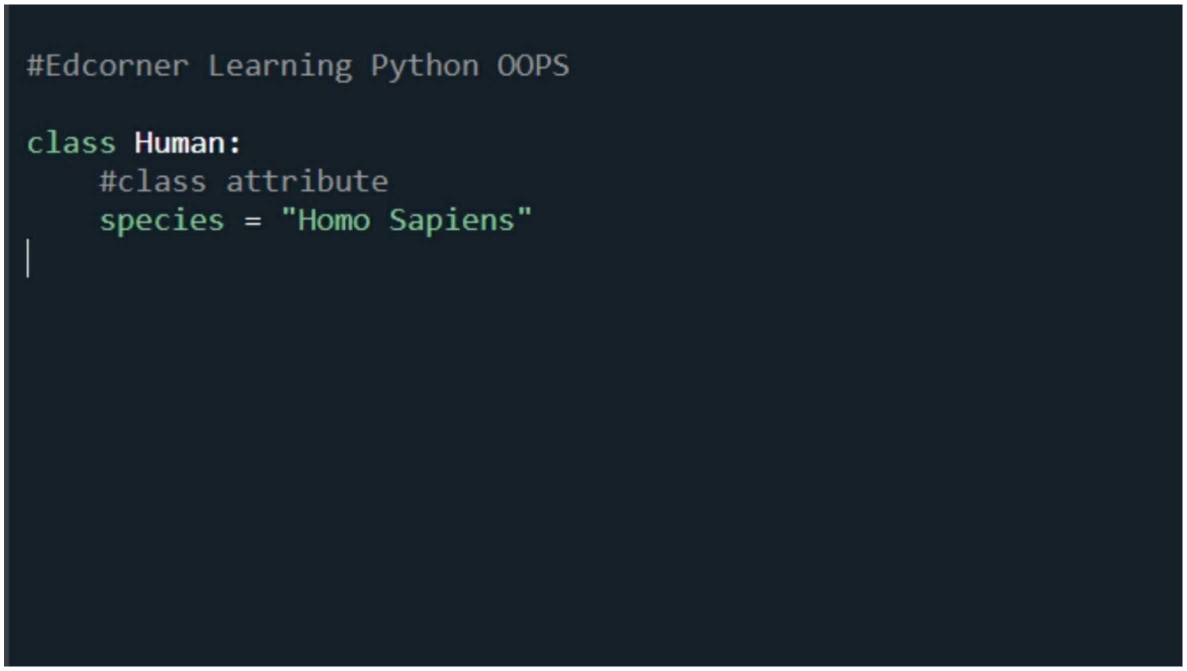
self.age = The age attribute is created and given the value of the age parameter that was supplied.

self.gender = The gender parameter value is produced and assigned to the gender attribute.

In Python, there are two categories of attributes:

Class attribute number 1:

These variables apply to all instances of the class equally. For each new instance that is created, they do not have new values. Just after the class definition, they are defined.

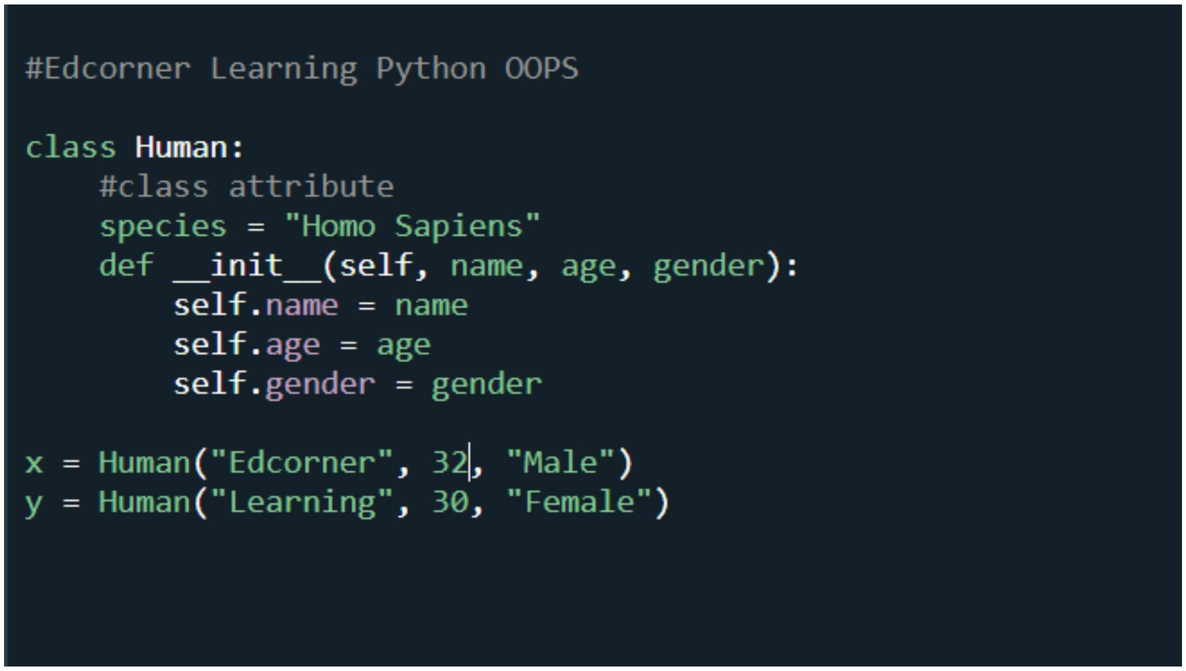


```
#Edcorner Learning Python OOPS

class Human:
    #class attribute
    species = "Homo Sapiens"
```

In this case, whatever object we create will have a fixed value for the species.

2. Instance Information:

The variables that are defined inside of any class function are known as instance attributes. Every instance of the class has a unique value for the instance attribute. These values rely on the value that was supplied when the instance was created.

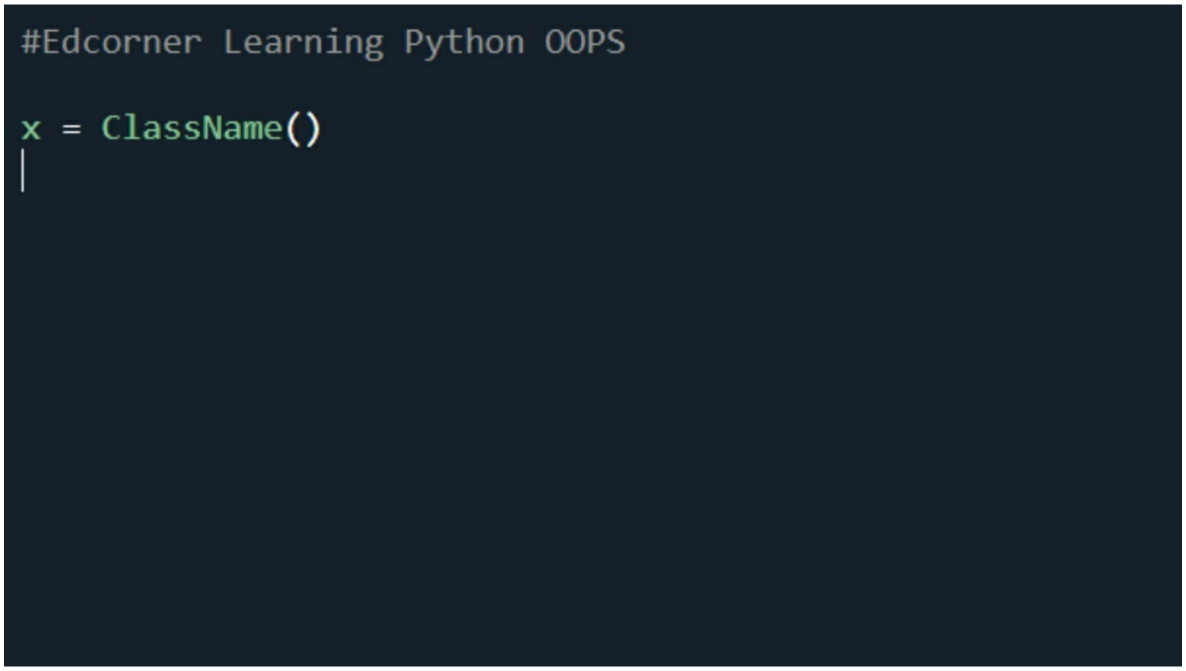


```python
#Edcorner Learning Python OOPS

class Human:
    #class attribute
    species = "Homo Sapiens"
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

x = Human("Edcorner", 32, "Male")
y = Human("Learning", 30, "Female")
```

The instance attributes in this case are name, age, and gender. When a new instance of the class is created, they will have different values.

8 Creating a Class Object

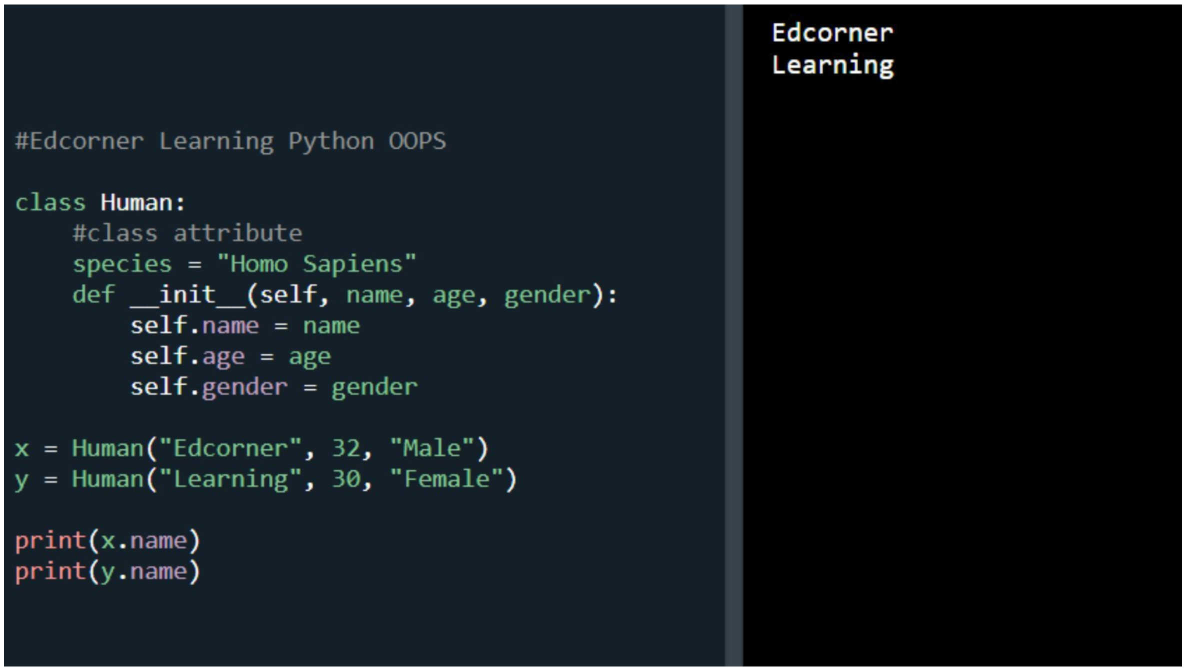
It is referred to as instantiating an object when a new instance of a class is created. The class name and parantheses can be used to create an object. The object of a class can be assigned to any variable.



```
#Edcorner Learning Python OOPS

x = ClassName()
```

Memory is allocated to an object as soon as it is created. Therefore, employing the operator == to compare two instances of the same class will result in false (because both will have different memory assigned).

The values for name, age, and gender must also be supplied when creating objects of the Human class.



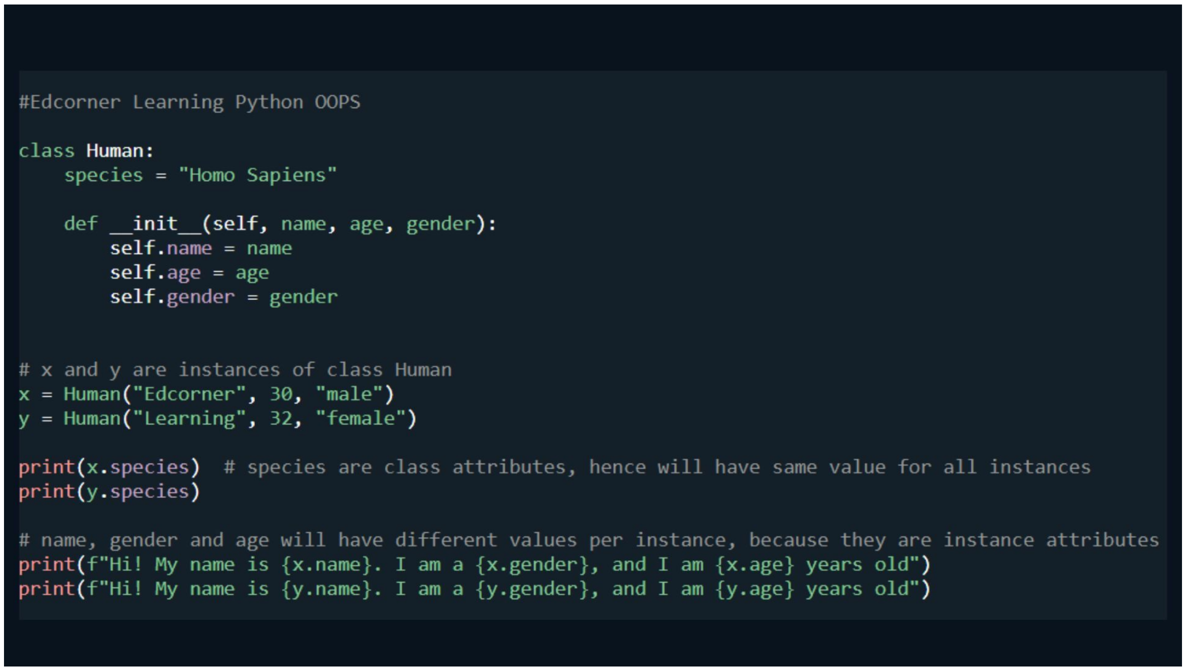
```
#Edcorner Learning Python OOPS

class Human:
    #class attribute
    species = "Homo Sapiens"
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

x = Human("Edcorner", 32, "Male")
y = Human("Learning", 30, "Female")

print(x.name)
print(y.name)
```

```
Edcorner
Learning
```

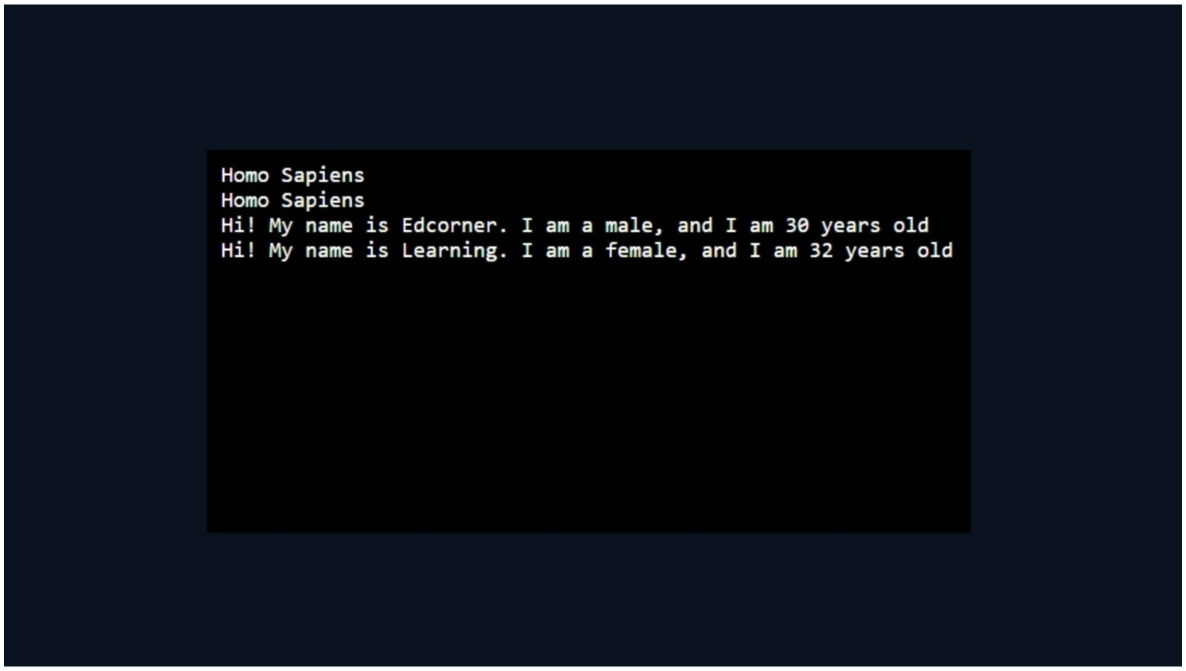
Here, we have created two Human class instances by passing in all the necessary inputs.

A TypeError will be raised if the necessary parameters are not given. TypeError: The three necessary positional arguments for init(), "name," "age," and "gender," are missing.

Now let's look at how to use class objects to retrieve those values. The dot notation allows us to access the instance values.

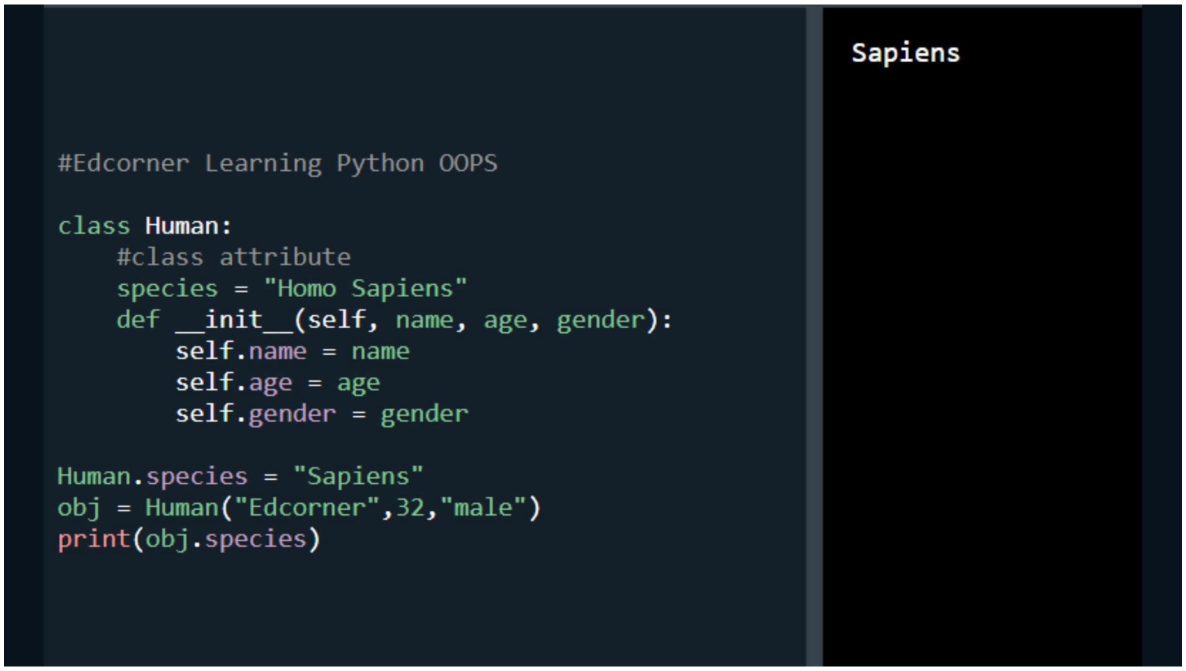


```python
#Edcorner Learning Python OOPS

class Human:
    species = "Homo Sapiens"

    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender


# x and y are instances of class Human
x = Human("Edcorner", 30, "male")
y = Human("Learning", 32, "female")

print(x.species)  # species are class attributes, hence will have same value for all instances
print(y.species)

# name, gender and age will have different values per instance, because they are instance attributes
print(f"Hi! My name is {x.name}. I am a {x.gender}, and I am {x.age} years old")
print(f"Hi! My name is {y.name}. I am a {y.gender}, and I am {y.age} years old")
```



```
Homo Sapiens
Homo Sapiens
Hi! My name is Edcorner. I am a male, and I am 30 years old
Hi! My name is Learning. I am a female, and I am 32 years old
```

As a result, we discover that the dot operator is all we need to access the instance and class attributes.



```python
#Edcorner Learning Python OOPS

class Human:
    #class attribute
    species = "Homo Sapiens"
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

Human.species = "Sapiens"
obj = Human("Edcorner",32,"male")
print(obj.species)
```

```
Sapiens
```

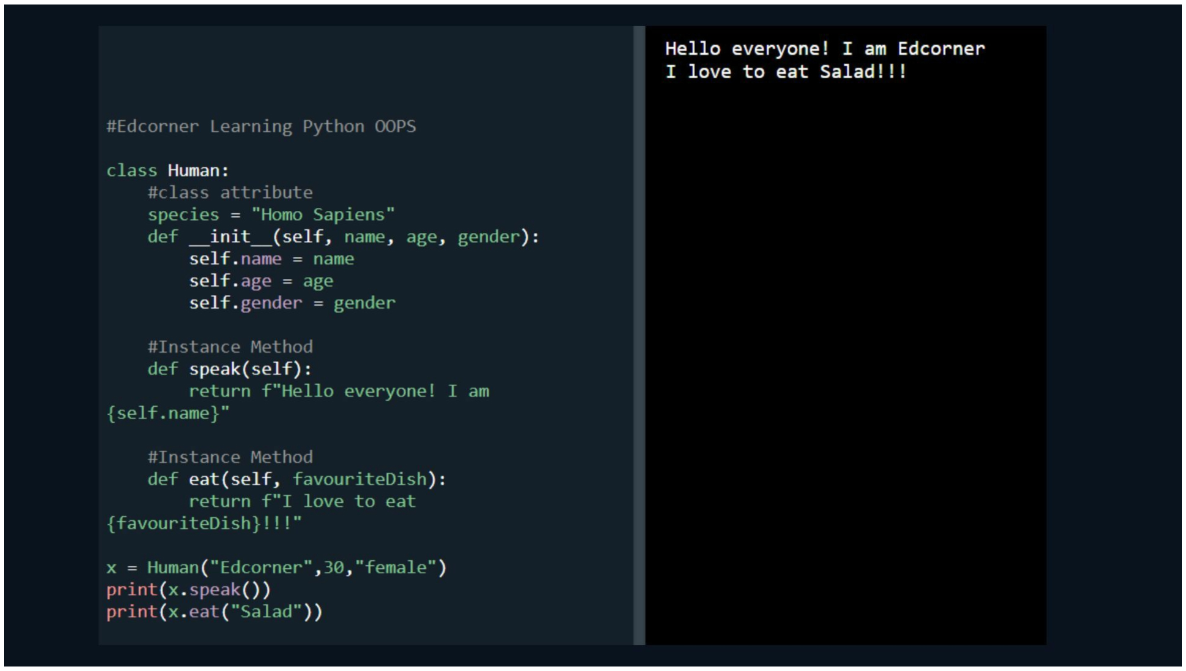
The class attributes in the example above have the identical values of "Homo Sapiens," but the instance attributes have various values depending on the parameter we gave when constructing our object.

However, we can alter the value of a class attribute by setting a new value to classname.classAttribute.

9 What is Instance Methods with Example

A function inside a class that can only be called from instances of that class is termed an instance method. An instance method's first parameter is always self, much as init().

Let's take an example and implement some functions



```
#Edcorner Learning Python OOPS

class Human:
    #class attribute
    species = "Homo Sapiens"
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    #Instance Method
    def speak(self):
        return f"Hello everyone! I am
{self.name}"

    #Instance Method
    def eat(self, favouriteDish):
        return f"I love to eat
{favouriteDish}!!!"

x = Human("Edcorner",30,"female")
print(x.speak())
print(x.eat("Salad"))
```

```
Hello everyone! I am Edcorner
I love to eat Salad!!!
```

Two instance methods exist for the Human class:

speak() produces a string containing the Human's name.

eat() returns a string with the Human's favourite food and accepts the single input "favouriteDish".

Now that we have a solid understanding of what Python classes, objects, and methods are, it is time to turn our attention to the foundational ideas of OOP.

10 The four core ideas of object-oriented programming are:

- Inheritance
- Encapsulation
- Polymorphism
- Abstraction of data.

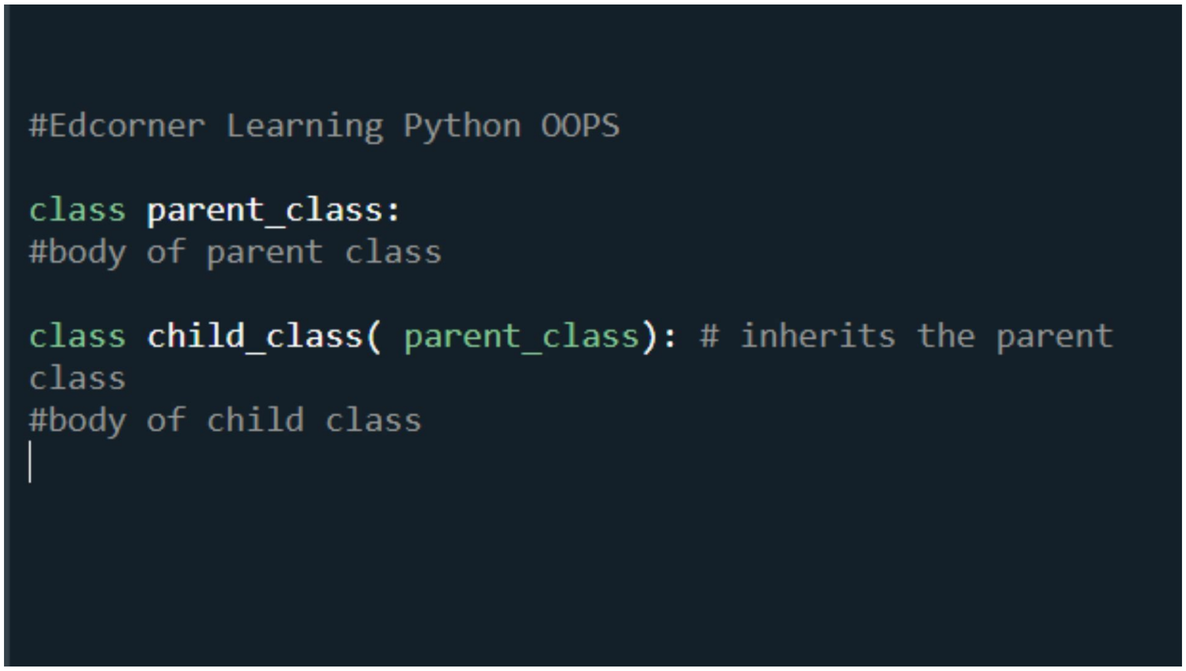
Let's take a closer look at each of the Python OOPS ideas.

11 Inheritance

People frequently tell newborns that they have face features that resemble those of their parents or that they have inherited particular traits from their parents. It's possible that you've also observed that you share a few characteristics with your parents.

The real-world situation is fairly similar to inheritance as well. However, in this case, the "parent classes"' features are passed down to the "child classes." They are referred to as "properties" and "methods" here, along with the qualities they inherit.

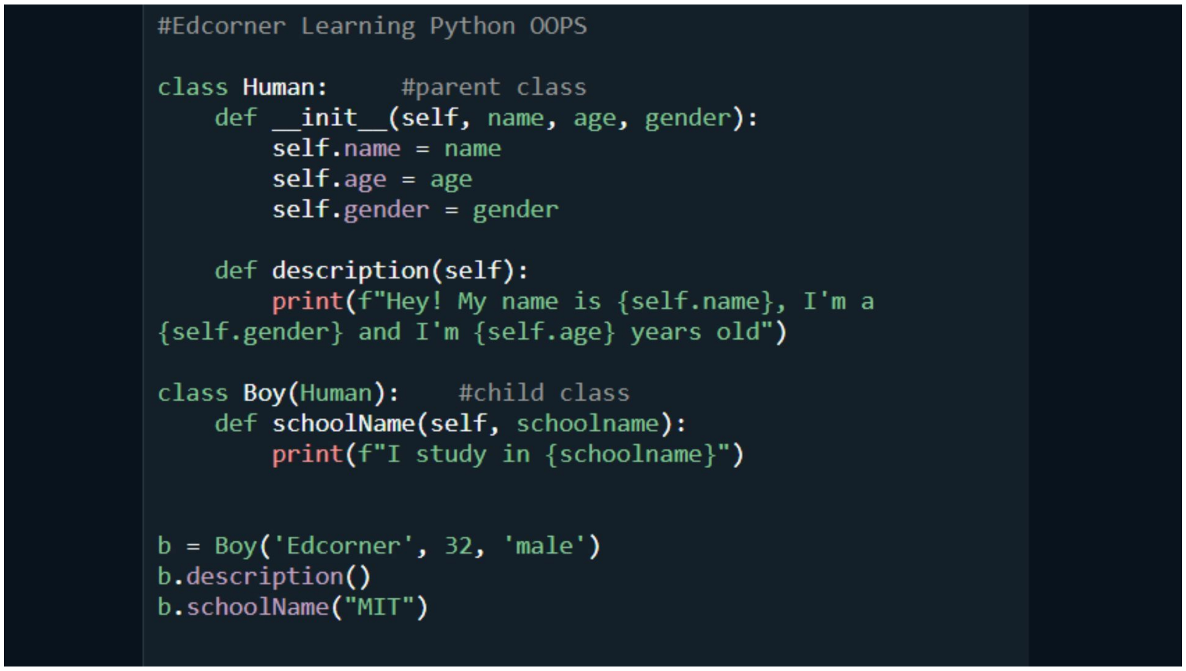
A class can derive its methods and attributes from another class's by using the process known as inheritance. Inheritance is the process of a child class receiving the properties of a parent class.



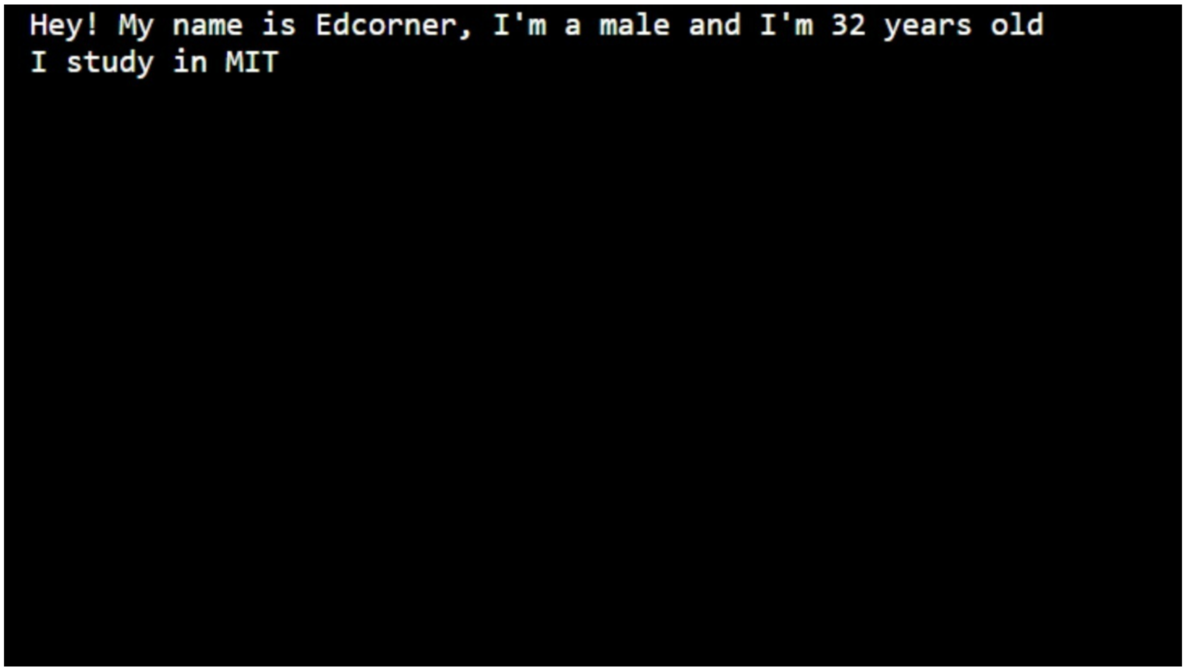
```
#Edcorner Learning Python OOPS

class parent_class:
#body of parent class

class child_class( parent_class): # inherits the parent
class
#body of child class
```

The Parent class is the class from which the properties are inherited, and the Child class is the class from which the properties are inherited.

As we did in our previous examples, we define a typical class. After that, we may declare the child class and provide the name of the parent class it is descended from in parenthesis.



```
#Edcorner Learning Python OOPS

class Human:     #parent class
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    def description(self):
        print(f"Hey! My name is {self.name}, I'm a
{self.gender} and I'm {self.age} years old")

class Boy(Human):    #child class
    def schoolName(self, schoolname):
        print(f"I study in {schoolname}")


b = Boy('Edcorner', 32, 'male')
b.description()
b.schoolName("MIT")
```

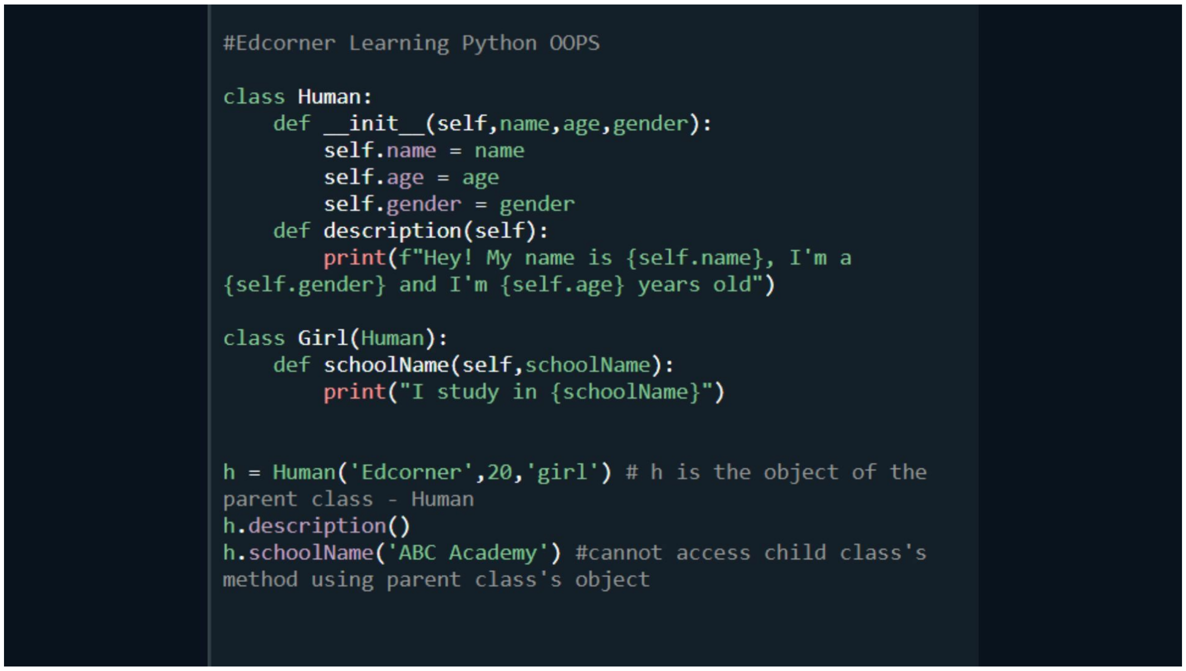


```
Hey! My name is Edcorner, I'm a male and I'm 32 years old
I study in MIT
```

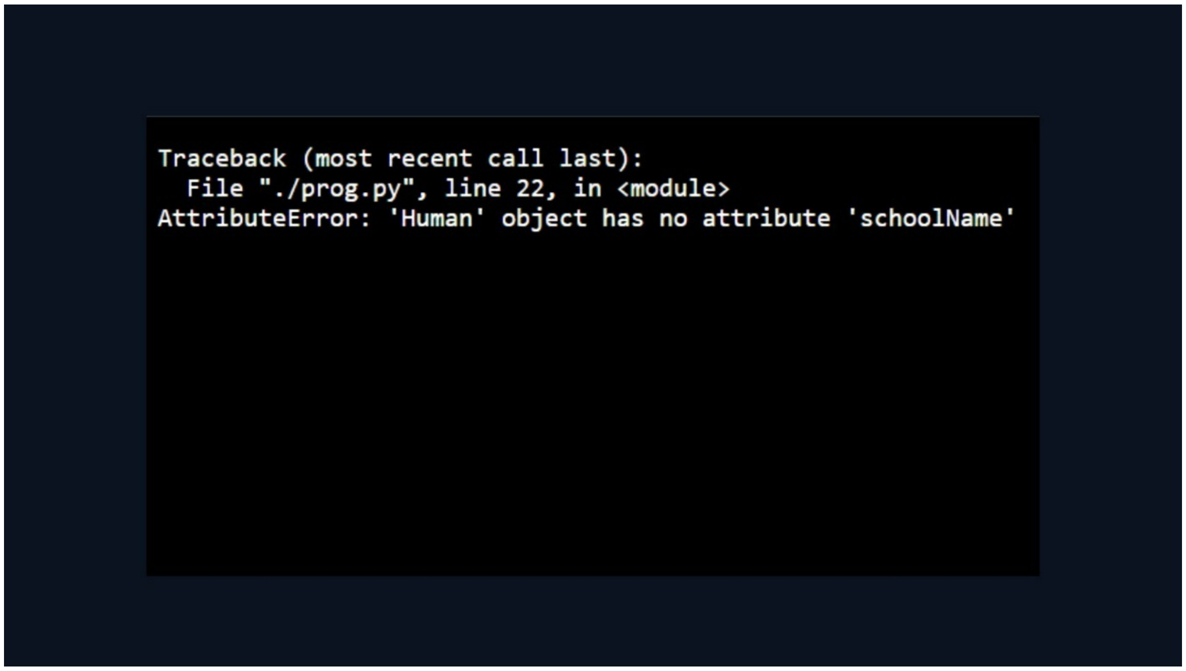
The parent class Human is inherited by the child class Boy in the example above. Because Boy is inheriting from Human, we can access all of its methods and properties when we create an instance of the Boy class.

In the Boy class, a method called schoolName has also been defined. The parent class object is unable to access the method schoolName. The schoolName method can, however, be called by making a child class object (Boy).

Let's look at the problem we run into if we try to use the object from the parent class to invoke the methods of a child class.



```
#Edcorner Learning Python OOPS

class Human:
    def __init__(self,name,age,gender):
        self.name = name
        self.age = age
        self.gender = gender
    def description(self):
        print(f"Hey! My name is {self.name}, I'm a
{self.gender} and I'm {self.age} years old")

class Girl(Human):
    def schoolName(self,schoolName):
        print("I study in {schoolName}")


h = Human('Edcorner',20,'girl') # h is the object of the
parent class - Human
h.description()
h.schoolName('ABC Academy') #cannot access child class's
method using parent class's object
```

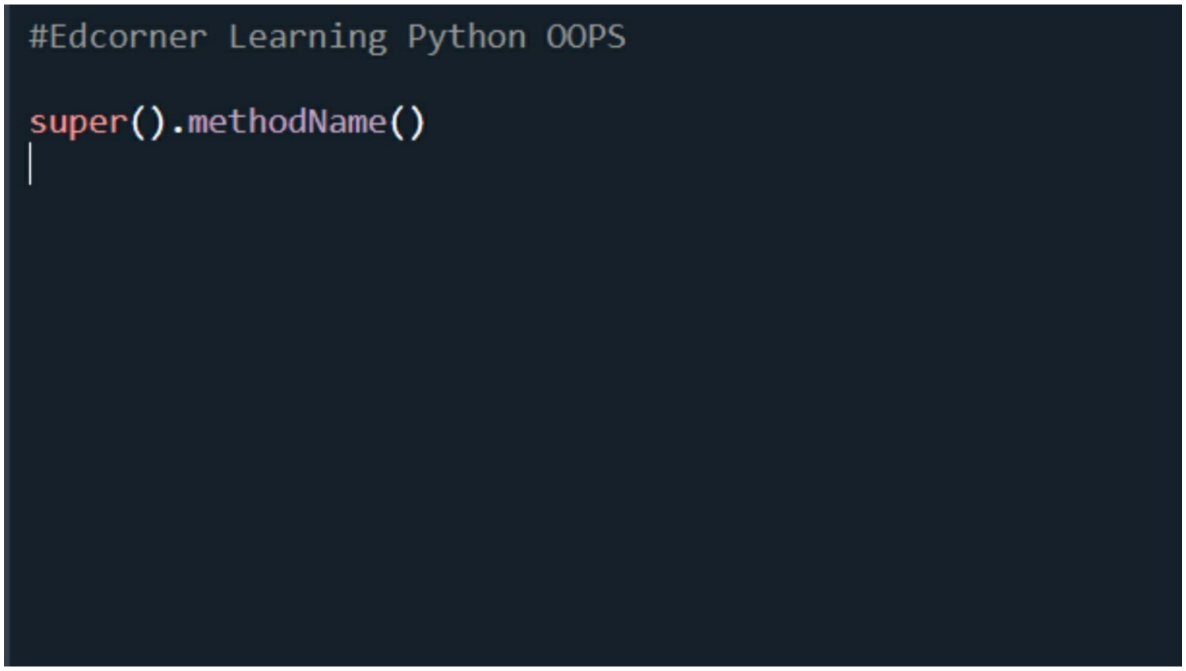


```
Traceback (most recent call last):
  File "./prog.py", line 22, in <module>
AttributeError: 'Human' object has no attribute 'schoolName'
```

Therefore, the AttributeError: 'Human' object has no attribute'schoolName' is returned in this case. because it is not possible for child classes to access parent class data and properties.

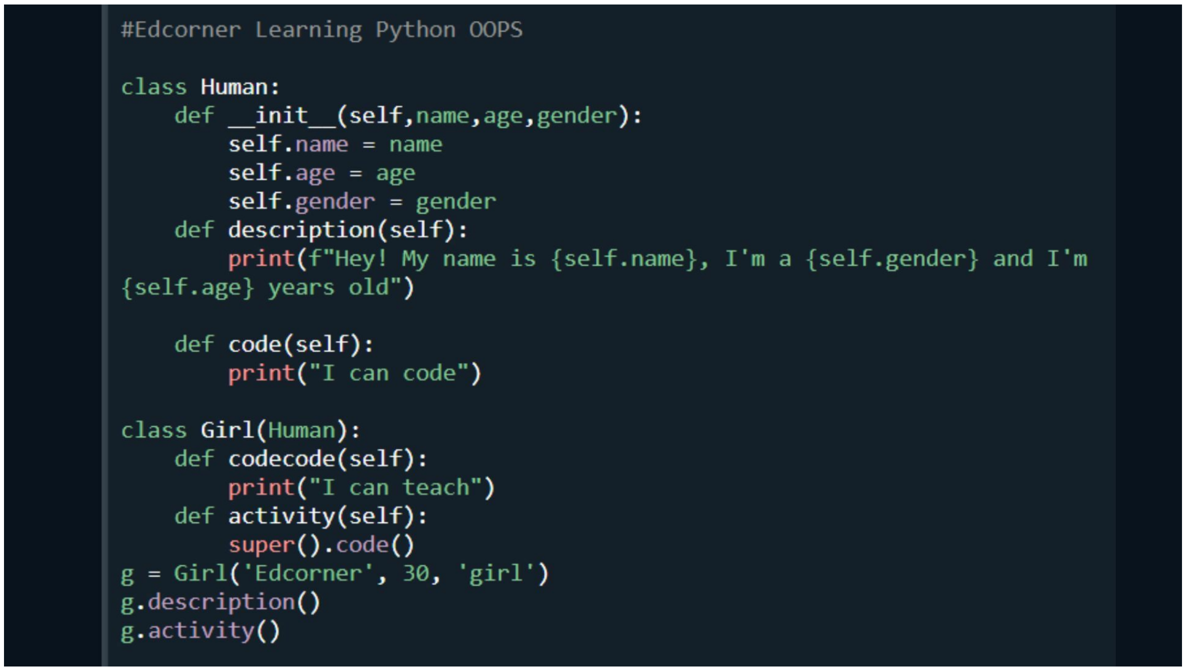
12 Super()

The parent class is referred to in the inheritance-related method super(). It can be used to locate a certain method in a superclass of an object. It serves a very important purpose. Let's examine its operation —

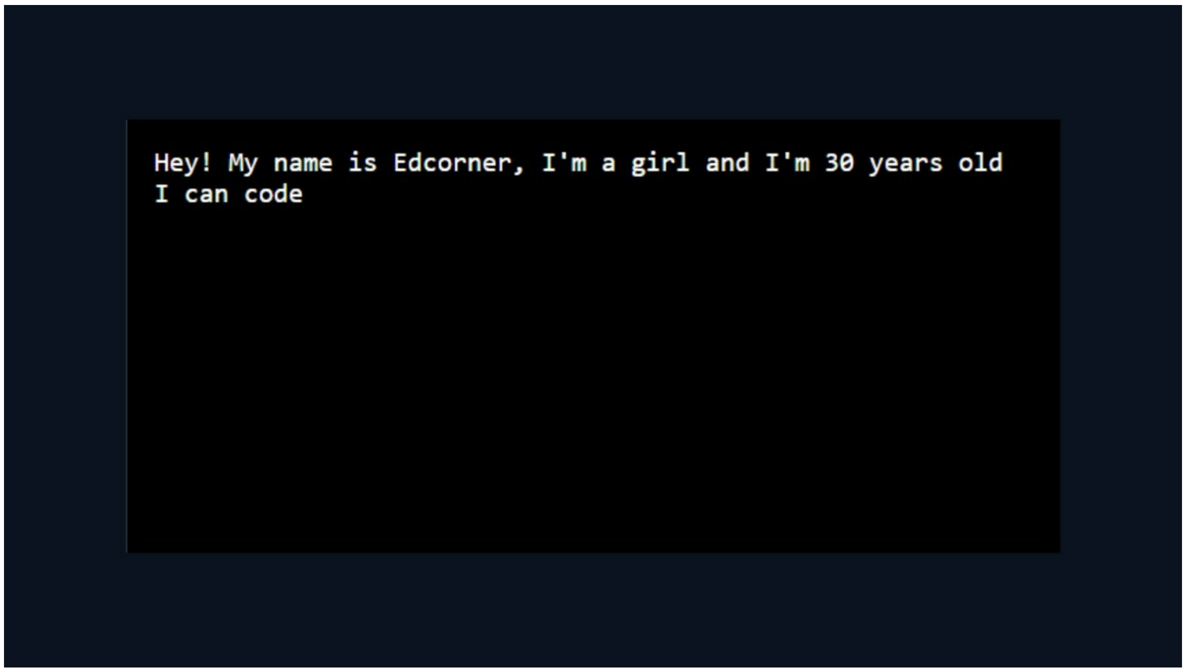


```
#Edcorner Learning Python OOPS

super().methodName()
```

The super function's syntax is as follows. After the super() keyword, we write the name of the parent class function we want to refer to.



```
#Edcorner Learning Python OOPS

class Human:
    def __init__(self,name,age,gender):
        self.name = name
        self.age = age
        self.gender = gender
    def description(self):
        print(f"Hey! My name is {self.name}, I'm a {self.gender} and I'm
{self.age} years old")

    def code(self):
        print("I can code")

class Girl(Human):
    def codecode(self):
        print("I can teach")
    def activity(self):
        super().code()
g = Girl('Edcorner', 30, 'girl')
g.description()
g.activity()
```



```
Hey! My name is Edcorner, I'm a girl and I'm 30 years old
I can code
```

Here, the classes Human and Girl have definitions for the Code() method. But as you can see, each method has a separate implementation. The Code technique in Human class reads "I can Code," whereas the Code method in Girl class reads "I can teach." So let's call the Code technique from the child class to the parent class.

As you can see, we are using super to call the Code method at line 19. ().

Code(). This will invoke the Human class' Code method. Thus, "I can Code" is printed. Nevertheless, Code() was already implemented in Girl (at line 15).

If a method with the same name already exists in the subclass, super() will still call the method in the superclass.

13 Polymorphism

Let's say you are using your phone to browse the Instagram feeds. You opened Spotify and began playing your favourite song because you suddenly had the urge to listen to some music as well. After a while, you received a call, so you paused whatever you were doing in the background to answer it. Your friend called and requested that you text them a certain person's phone number. So you sent him the phone number through SMS and carried on with your tasks.

Have you picked up on anything? With just one device—your mobile phone—you could surf through feeds, listen to music, take and make phone calls, and message.

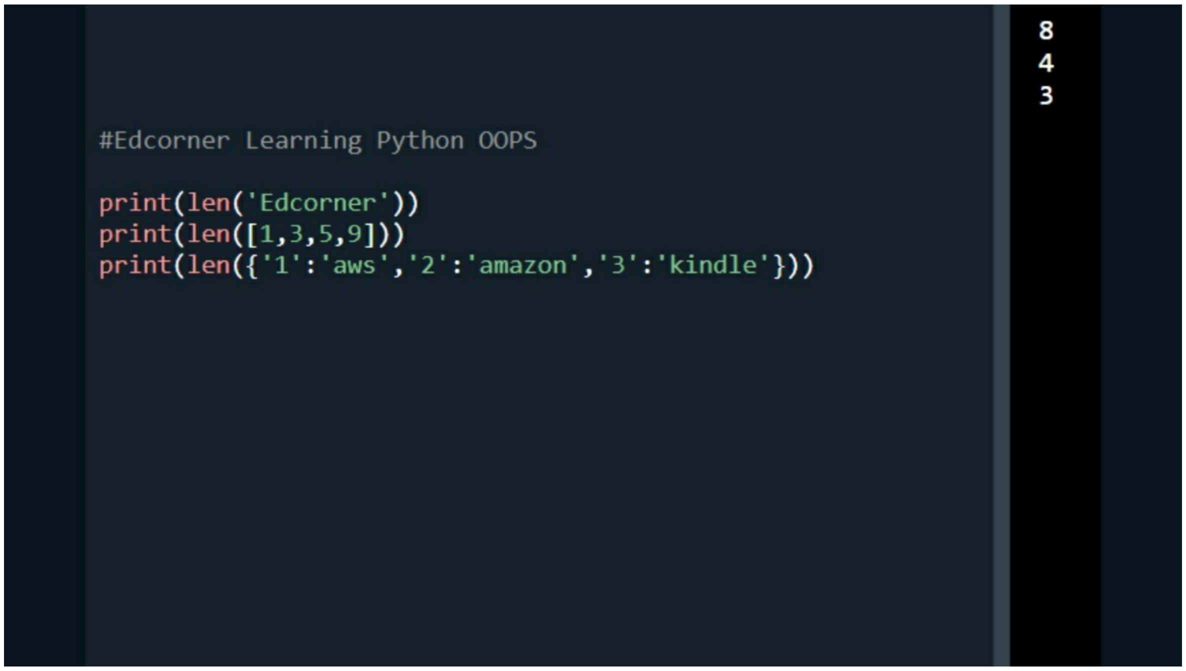
Therefore, polymorphism is comparable to that. Poly means numerous, and morph denotes different forms. Therefore, polymorphism as a whole refers to something with various forms. Or "something" that, depending on the circumstance, can exhibit a variety of behaviours.

In OOPS, polymorphism describes functions with the same names but different behaviours. Alternatively, a different function signature with the same function name (parameters passed to the function).

All of a child class's properties are inherited from the parent class's methods. But occasionally, it tries to change the techniques by adding its own implementation. In Python, there are many different ways to use polymorphism.

An illustration of an inbuilt polymorphic function

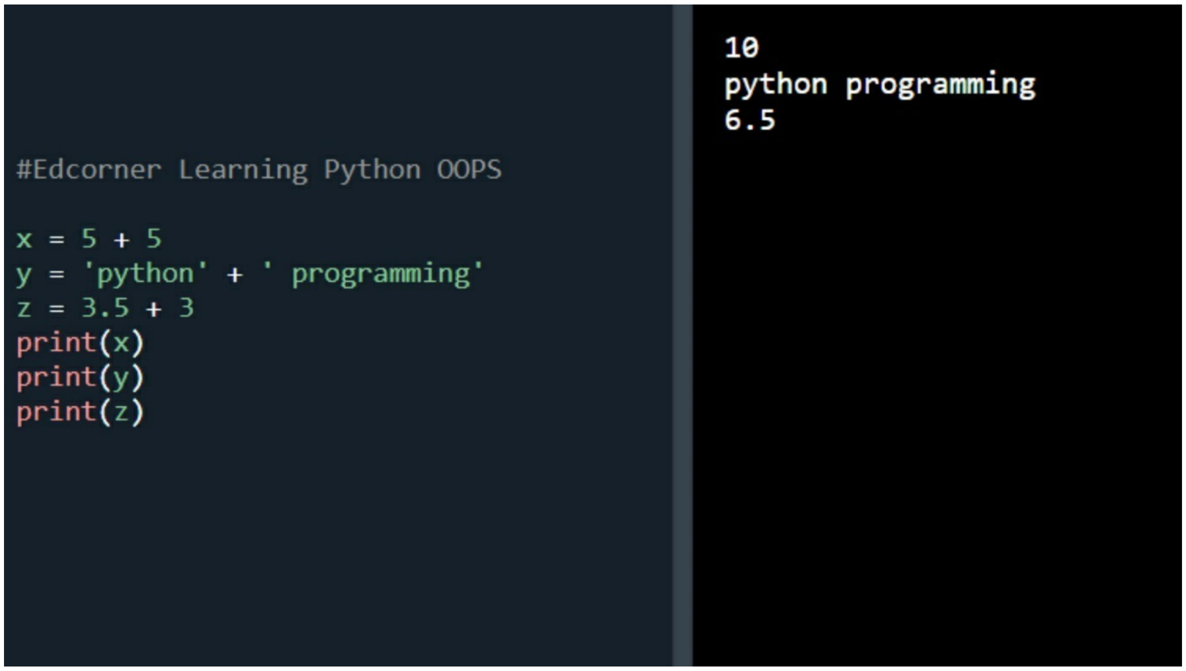
An example of an inbuilt polymorphism function is len(). It will just compute the value and return because we can use it to calculate the length of various kinds like strings, lists, tuples, and dictionaries.



```
#Edcorner Learning Python OOPS

print(len('Edcorner'))
print(len([1,3,5,9]))
print(len({'1':'aws','2':'amazon','3':'kindle'}))
```

```
8
4
3
```

Here, the len function was given a string, list, and dictionary, and it computed the answer. It serves as an illustration of an inbuilt polymorphic function.



```
#Edcorner Learning Python OOPS

x = 5 + 5
y = 'python' + ' programming'
z = 3.5 + 3
print(x)
print(y)
print(z)
```

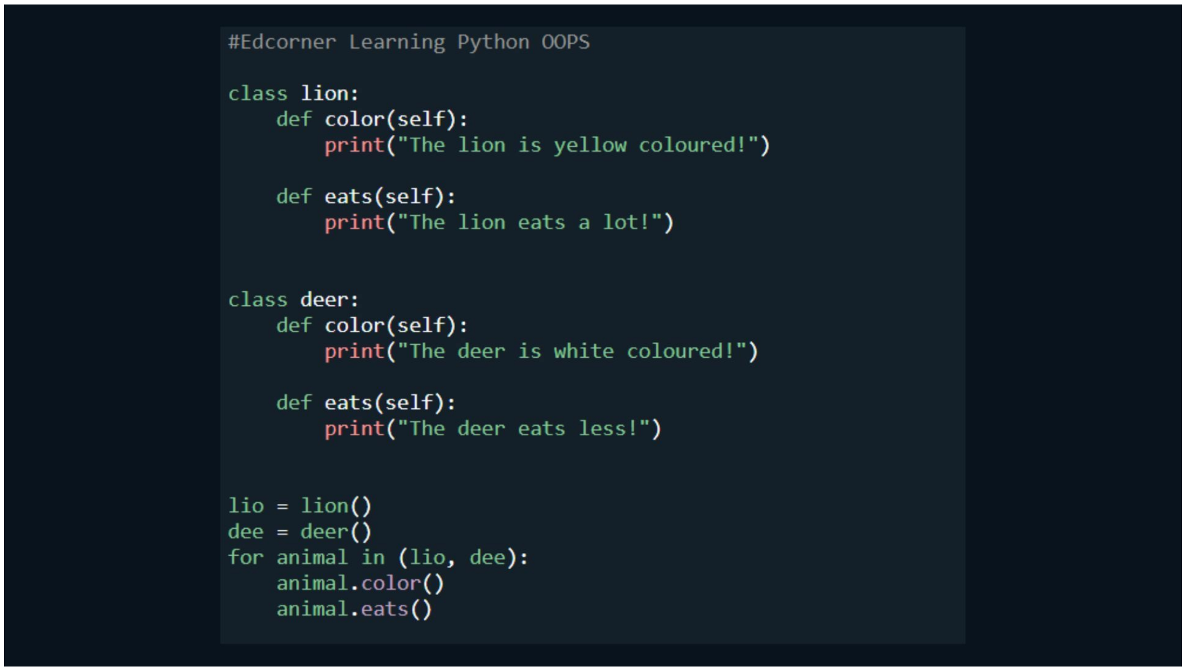
```
10
python programming
6.5
```

With the addition operator "+," polymorphism is also a possibility. It can be used to "add" integers, floats, or any other arithmetic operation. On the other side, it conducts the "concatenation" process with String.

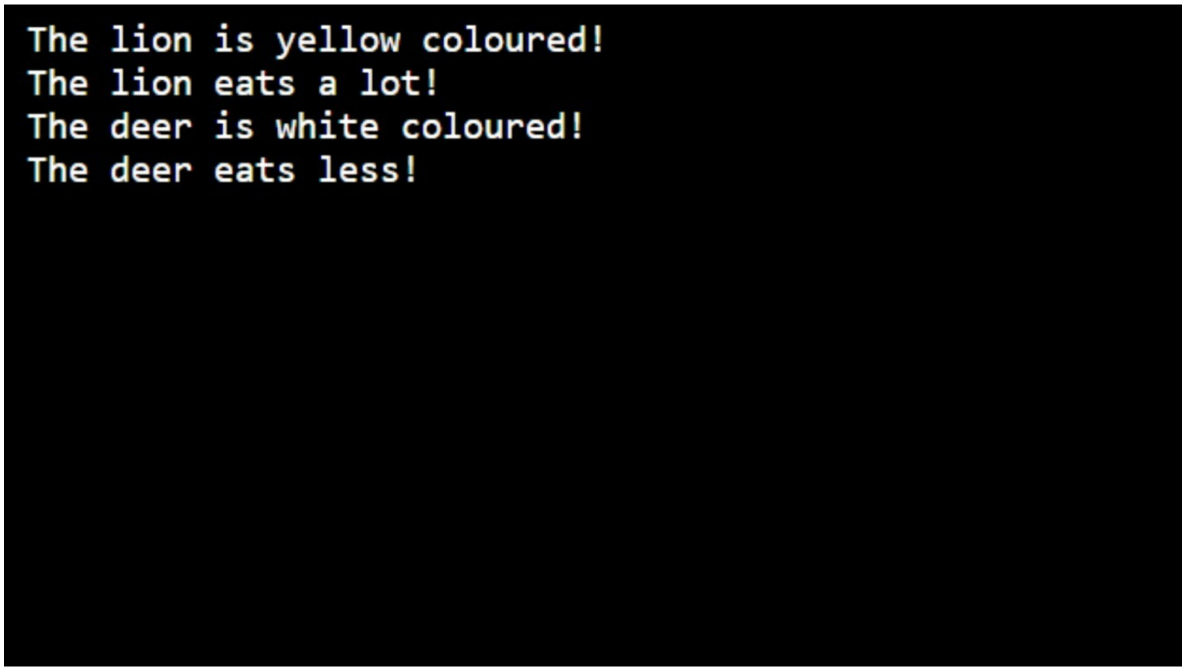
As a result, we can observe that different operations have been performed for various data types using the same operator, "+."

14 Class Methods and polymorphism

With the class methods, polymorphism is possible. Observe how:



```
#Edcorner Learning Python OOPS

class lion:
    def color(self):
        print("The lion is yellow coloured!")

    def eats(self):
        print("The lion eats a lot!")


class deer:
    def color(self):
        print("The deer is white coloured!")

    def eats(self):
        print("The deer eats less!")


lio = lion()
dee = deer()
for animal in (lio, dee):
    animal.color()
    animal.eats()
```

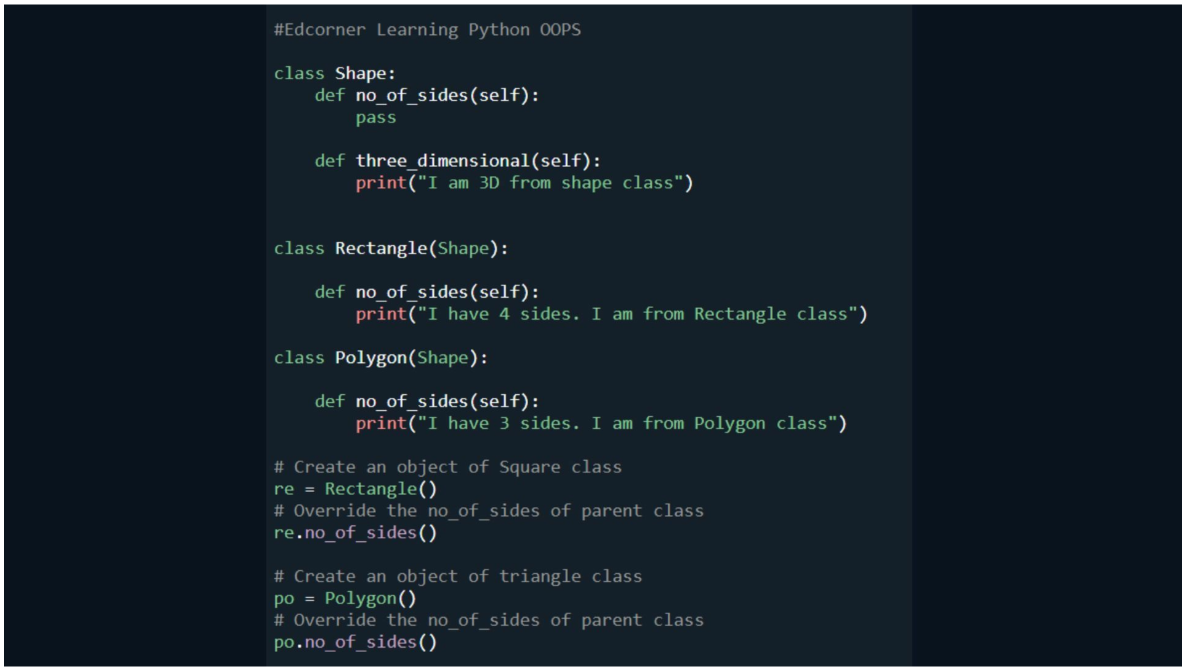


```
The lion is yellow coloured!
The lion eats a lot!
The deer is white coloured!
The deer eats less!
```

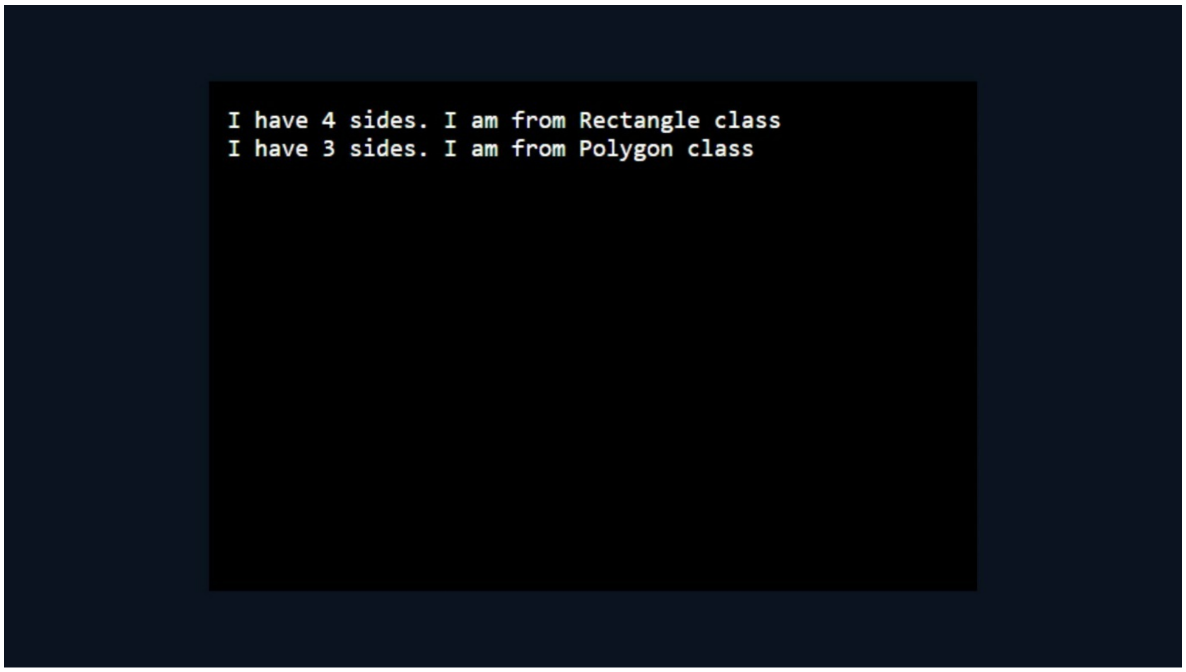
Using the variable animal, we may loop over the objects of the Lion and Deer, calling their respective instance methods. As a result, the behaviour (colour() & eats()) of both the Lion and the Deer are represented here by a single variable called animal. Therefore, it is according to the polymorphism rules!

15 Inheritance and Polymorphism

Polymorphism and inheritance are both possible. A method that a child class has inherited from a parent class may be changed to include the child class's own implementation. Method overriding is the practise of re-implementing a method in the child class. Here is a case where polymorphism with inheritance is demonstrated.



```
#Edcorner Learning Python OOPS

class Shape:
    def no_of_sides(self):
        pass

    def three_dimensional(self):
        print("I am 3D from shape class")


class Rectangle(Shape):

    def no_of_sides(self):
        print("I have 4 sides. I am from Rectangle class")

class Polygon(Shape):

    def no_of_sides(self):
        print("I have 3 sides. I am from Polygon class")

# Create an object of Square class
re = Rectangle()
# Override the no_of_sides of parent class
re.no_of_sides()

# Create an object of triangle class
po = Polygon()
# Override the no_of_sides of parent class
po.no_of_sides()
```



```
I have 4 sides. I am from Rectangle class
I have 3 sides. I am from Polygon class
```

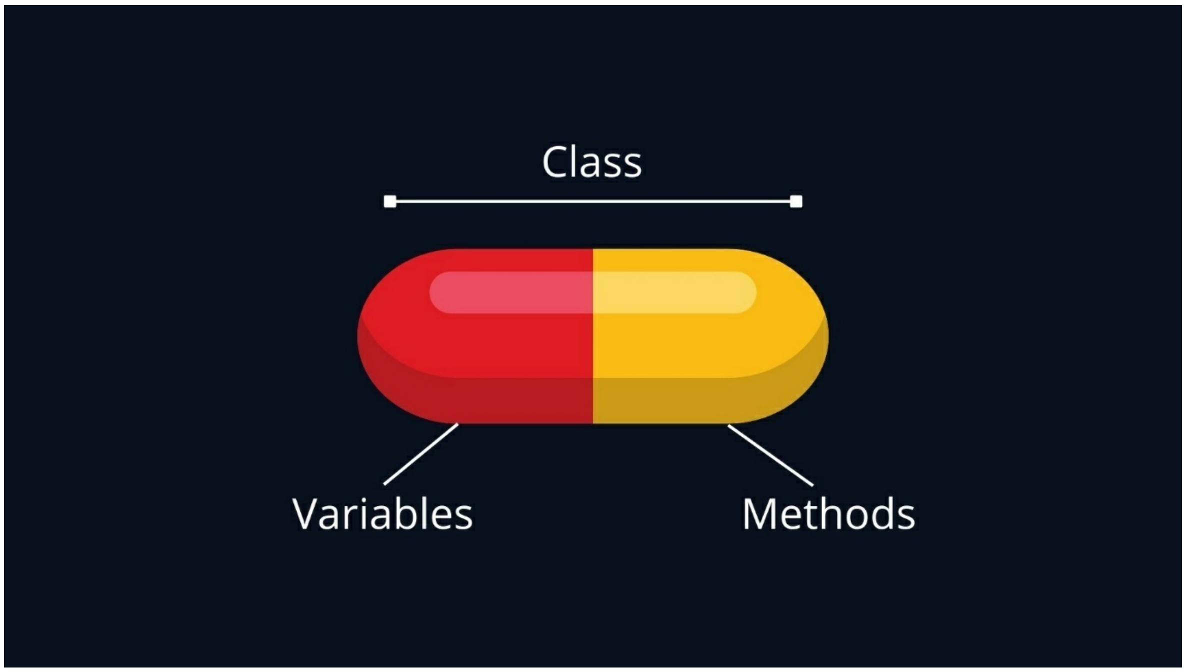
Here, the method of the shape class has been replaced by the Rectangle and Polygon class. As a result, there are many implementations of the method no of sides here depending on the form. Therefore, it is consistent with polymorphism.

16 Encapsulation

You must have encountered pill forms for medications where the entire contents are kept sealed inside the capsule’s lid. In essence, a capsule contains many drug combinations.

In programming, the variables and the methods are similarly kept in a container known as a “class”! Yes, we have learned a lot about classes in Python, and we are already aware that every variable and function we create in OOP stays inside the class.

Encapsulation is the term used in Python to describe the act of combining data with the matching methods (behaviour) into a single entity.

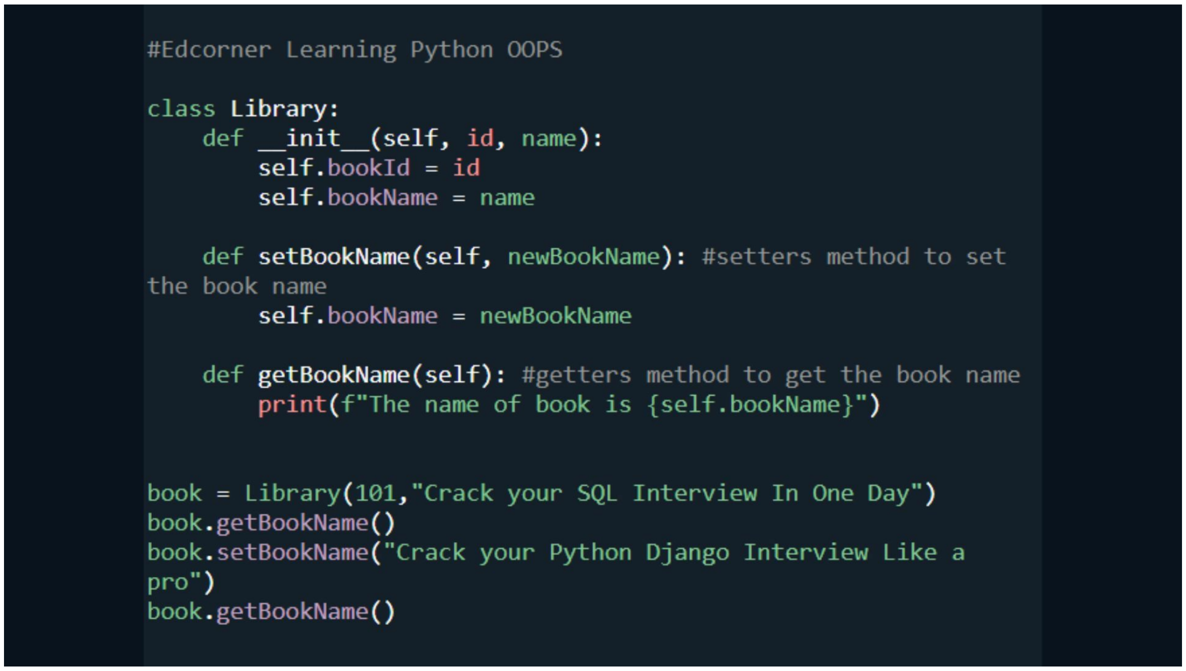


Encapsulation, then, is a programming approach that ties the class's variables and methods together and makes it impossible for other classes to access them. It is an example of an OOPS concept in Python.

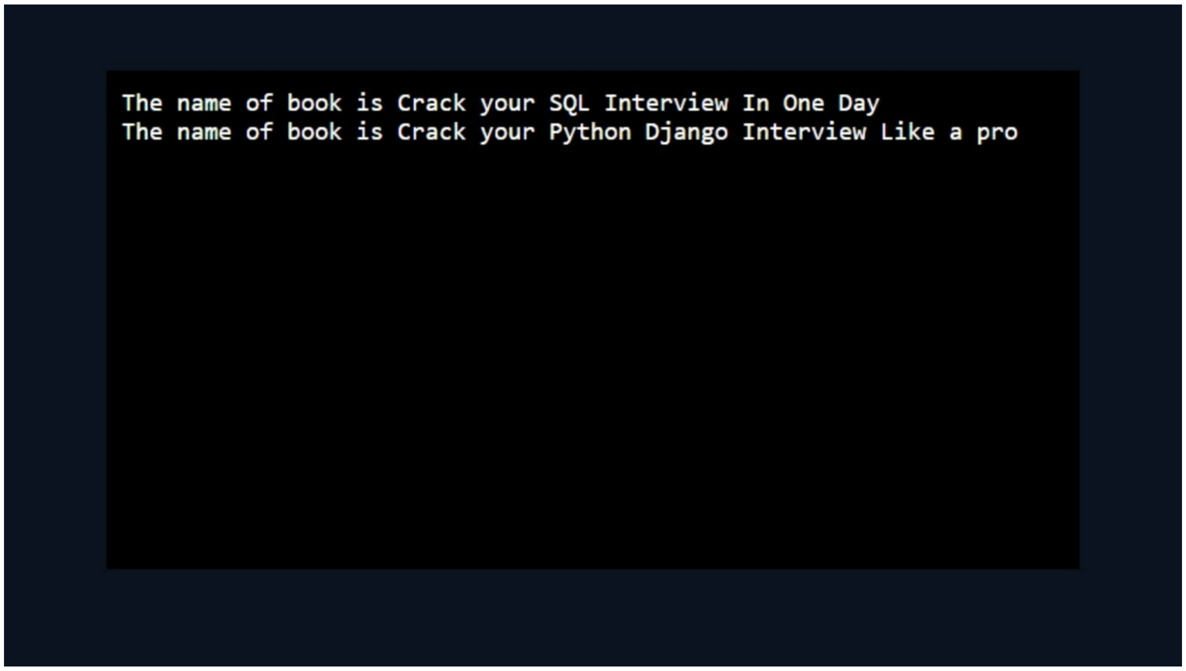
Security is possible via encalpsulation. It protects the data from outside access. By encapsulating an object or piece of information, a company can prevent clients or other unauthorised individuals from accessing it without permission.

17 A getter and a setter

Encapsulation is mostly used for Data Hiding. To do this, we give our classes getter and setter methods. Anyone who needs some data must call the getter method in order to obtain it. Additionally, users must utilise the setter method if they wish to add a value to the data; otherwise, they won't be able to. However, the internal operation of these getter and setter methods is hidden from the outside world.



```
#Edcorner Learning Python OOPS

class Library:
    def __init__(self, id, name):
        self.bookId = id
        self.bookName = name

    def setBookName(self, newBookName): #setters method to set
the book name
        self.bookName = newBookName

    def getBookName(self): #getters method to get the book name
        print(f"The name of book is {self.bookName}")


book = Library(101,"Crack your SQL Interview In One Day")
book.getBookName()
book.setBookName("Crack your Python Django Interview Like a
pro")
book.getBookName()
```



```
The name of book is Crack your SQL Interview In One Day
The name of book is Crack your Python Django Interview Like a pro
```

For getting and setting the book names, respectively, we defined the getter getBookName() and setter setBookName() in the example above. As a result, we can no longer directly access or edit any data; instead, we can only get and set the book names when using the methods. As a result, our data is kept in a highly secure manner because other people are not fully aware of how the following methods are used (if their access are restricted).

Using access modifiers, we can additionally encourage the security of our data. Check out what access modifiers are.

18 Modifiers of Access

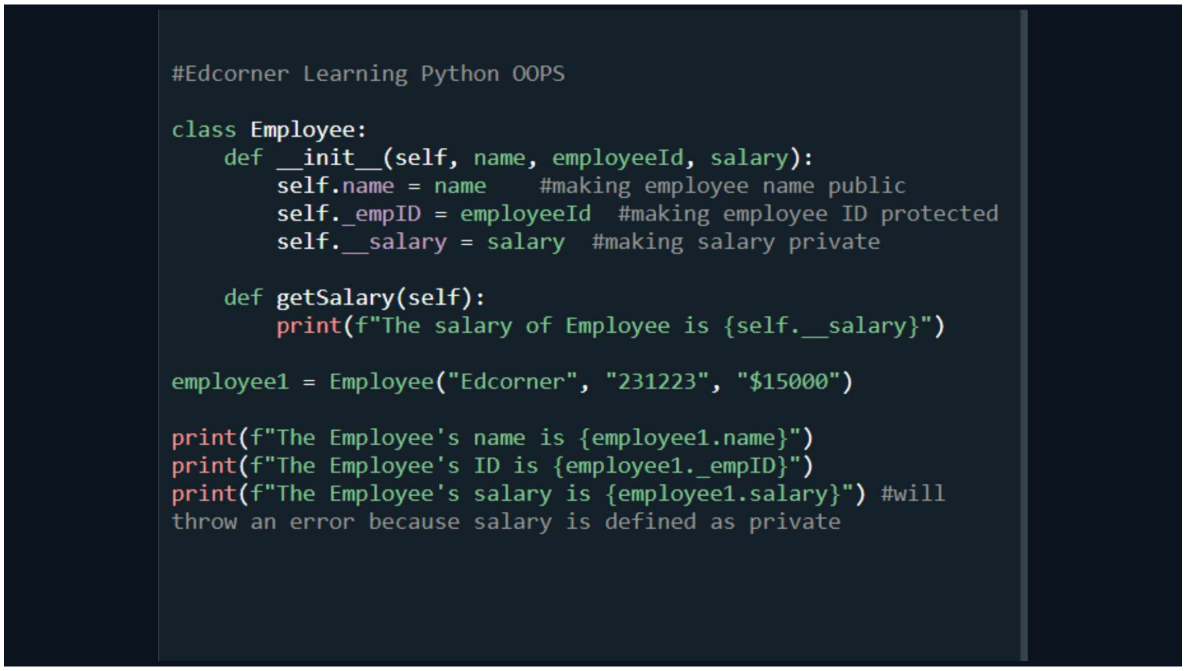
Access modifiers control who can access a class's variables and methods. Private, public, and protected are the three types of access modifiers offered by Python.

Direct access modifiers like public, private, and protected don't exist in Python. Utilizing single and double underscores will help us do this.

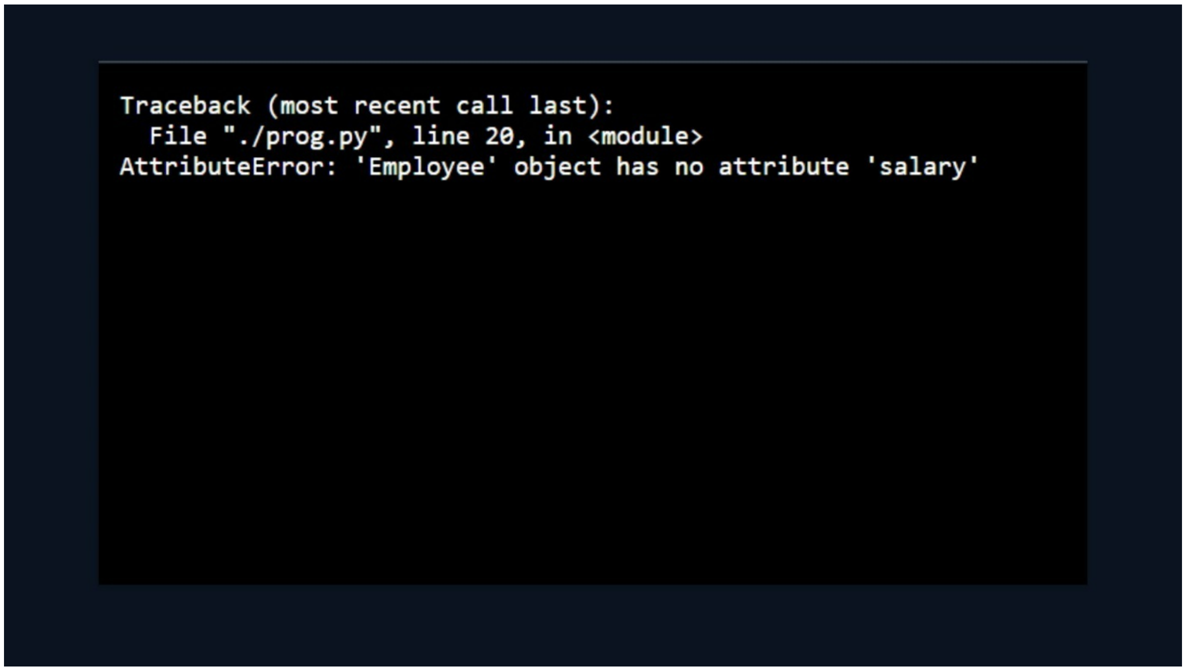
- Accessible from anyplace outside of the class as a public member.
- Private Member: Only available to members of the class
- Accessible within the class and its subclasses; protected member

Protected class is represented by a single underscore ( ). Private class is denoted by a double underscore, .

Let's say we try to create a class for employees.



```python
#Edcorner Learning Python OOPS

class Employee:
    def __init__(self, name, employeeId, salary):
        self.name = name    #making employee name public
        self._empID = employeeId  #making employee ID protected
        self.__salary = salary  #making salary private

    def getSalary(self):
        print(f"The salary of Employee is {self.__salary}")

employee1 = Employee("Edcorner", "231223", "$15000")

print(f"The Employee's name is {employee1.name}")
print(f"The Employee's ID is {employee1._empID}")
print(f"The Employee's salary is {employee1.salary}") #will
throw an error because salary is defined as private
```

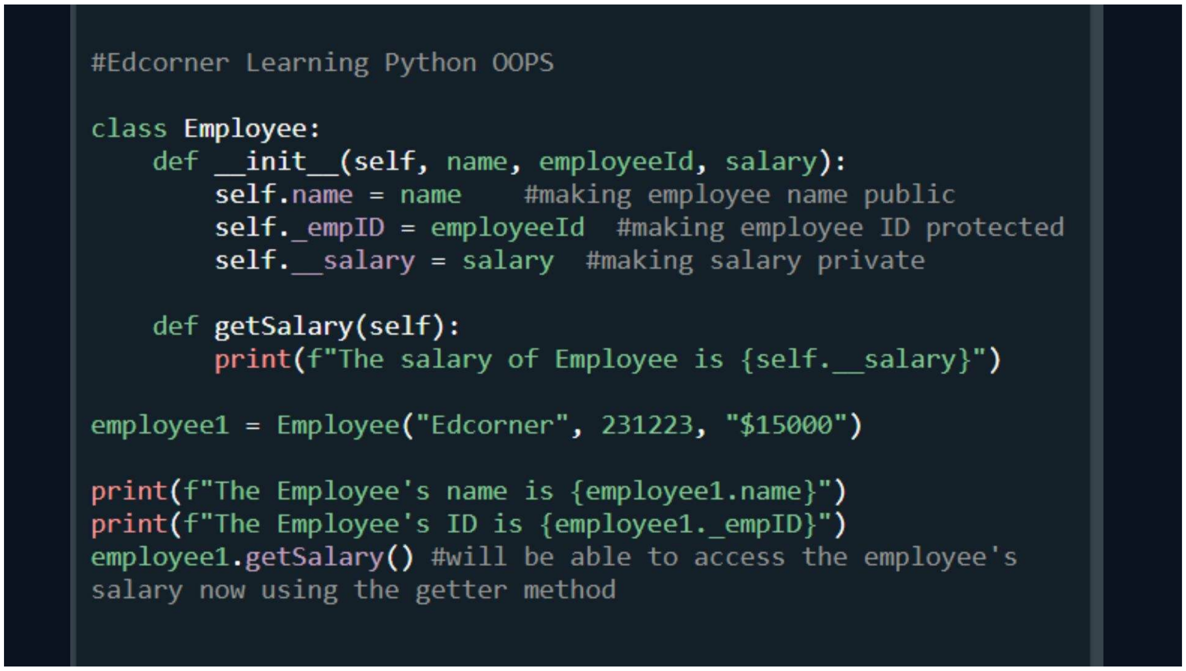


```
Traceback (most recent call last):
  File "./prog.py", line 20, in <module>
AttributeError: 'Employee' object has no attribute 'salary'
```

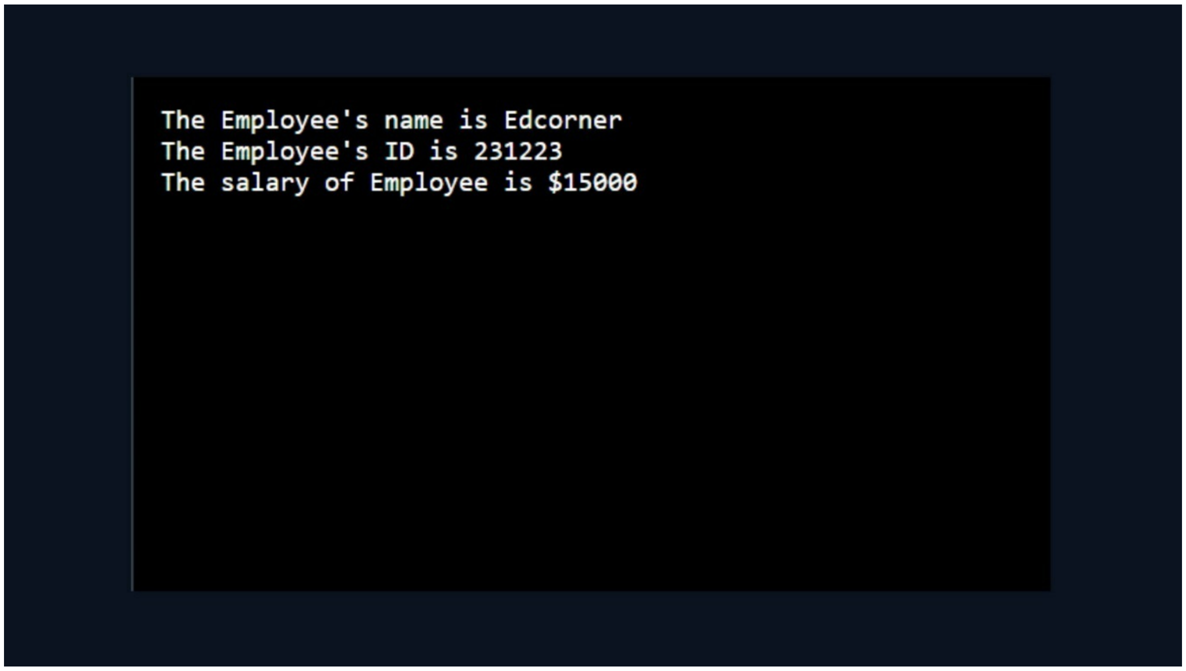
In this case, the employee's name is made public, their ID is preserved, and their compensation is kept confidential. Let's say we attempt to print every value. We may now access the employee's name or ID, but not his or her salary (because it is private).

However, we may call the getter method getSalary() we established in our Employee class to retrieve the employee's pay.

By this point, you may have realised how important getters and setters are, as well as how access modifiers limit who may access your data.



```
#Edcorner Learning Python OOPS

class Employee:
    def __init__(self, name, employeeId, salary):
        self.name = name    #making employee name public
        self._empID = employeeId  #making employee ID protected
        self.__salary = salary  #making salary private

    def getSalary(self):
        print(f"The salary of Employee is {self.__salary}")

employee1 = Employee("Edcorner", 231223, "$15000")

print(f"The Employee's name is {employee1.name}")
print(f"The Employee's ID is {employee1._empID}")
employee1.getSalary() #will be able to access the employee's
salary now using the getter method
```



```
The Employee's name is Edcorner
The Employee's ID is 231223
The salary of Employee is $15000
```

By establishing a public method to access private members, we can access a class's private members from outside of it (just like we did above). Another way to gain access is through name mangling.

When inheritance is utilised and you want the data members to be accessible only to the child classes, you use a protected data member.

Encapsulation thus guards against unwanted access to an object. To avoid unintentional data change, private and secured access levels are available.

19 Abstraction

You most likely use a laptop, phone, or tablet to read this book. While reading it, you are also presumably taking notes, underlining key passages, and possibly saving some information to your personal files. All you can see when you read this is a "screen" with the data that is being displayed to you. You just see the keyboard's keys as you type, so you don't have to worry about internal subtleties like how pushing a key can cause that word to appear onscreen. Alternatively, how pressing a button on your screen may launch a new tab!

Therefore, whatever we may see in this situation is abstract. We can only see the outcome it is producing and not the inside details (which actually matters to us).

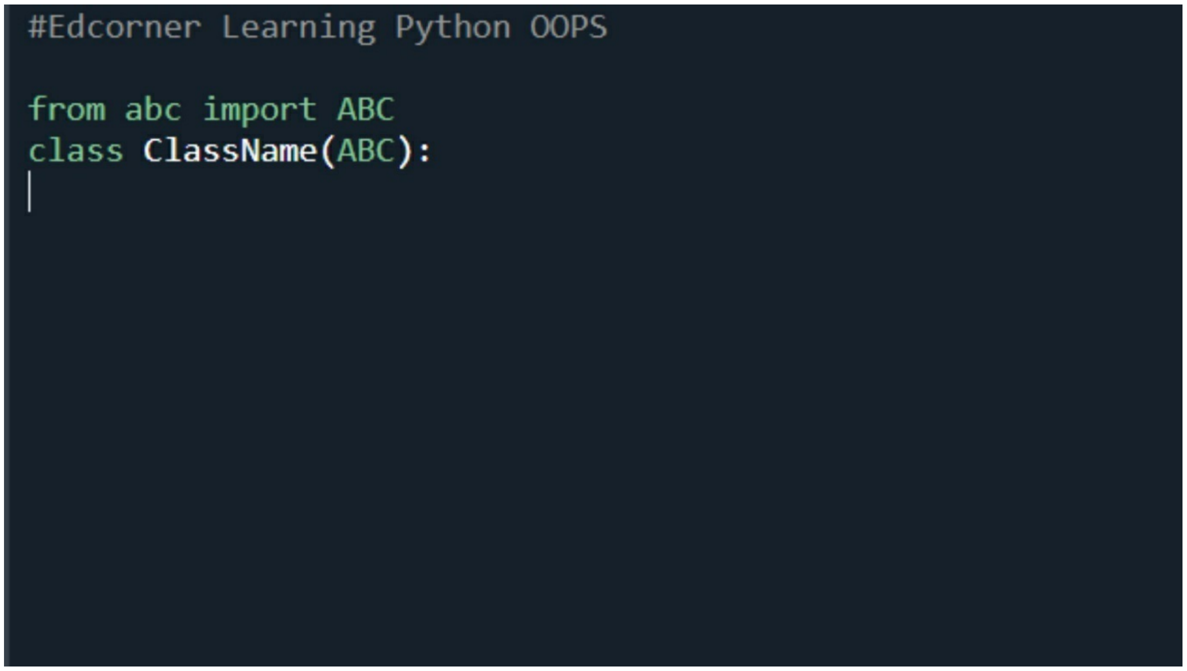
Similar to this, abstraction only reveals the functions that anything possesses while concealing any implementations or internal details.

Abstraction's major objective is to conceal background information and any extraneous data implementation so that people only see what they need to see. It aids in managing the codes' complexity.

20 Important aspects of abstract classes

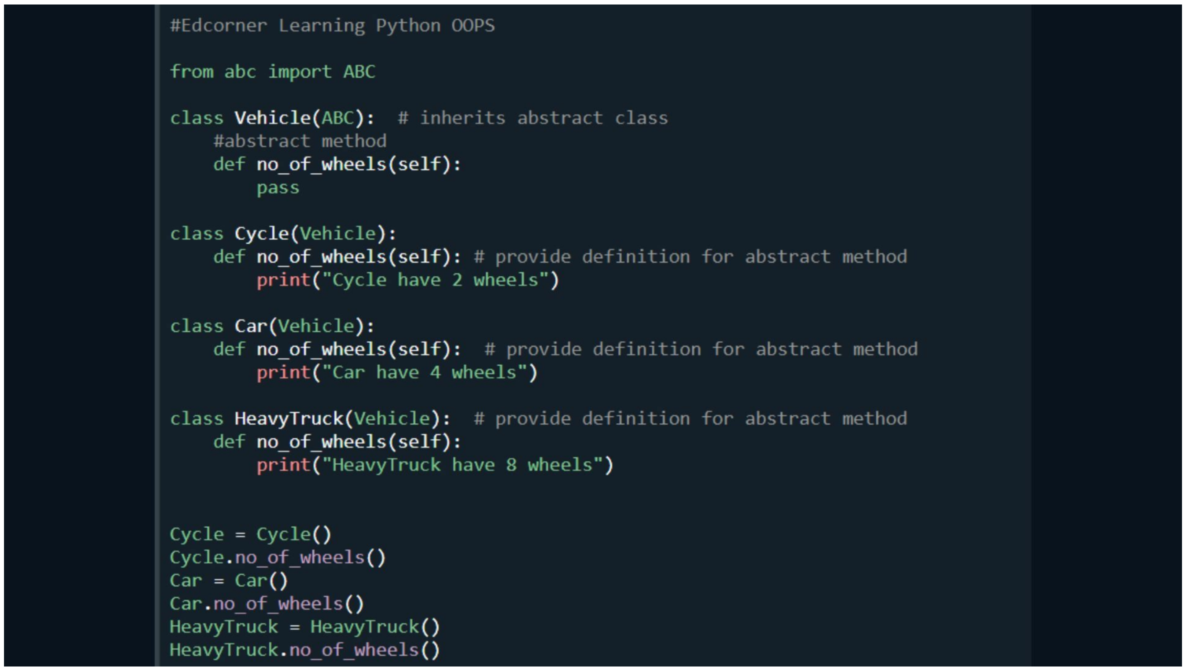
- Abstraction is used to obscure background information or any other extraneous data implementation so that people only see what they need to see.
- Utilizing abstract classes in Python, one can achieve abstraction.
- The term "abstract class" refers to a class that contains one or more abstract methods.
- There is no implementation for abstract methods on their own.
- Any subclass may inherit from an abstract class. The implementations for the abstract methods are provided by the subclasses that inherit the abstract classes.
- When building complicated functions, abstract classes might serve as a prototype for other classes. Additionally, the subclass that receives their inheritance can make use of the abstract methods to implement the features.
- Python has an abstraction module called ABC.

Python Abstract Class Syntax

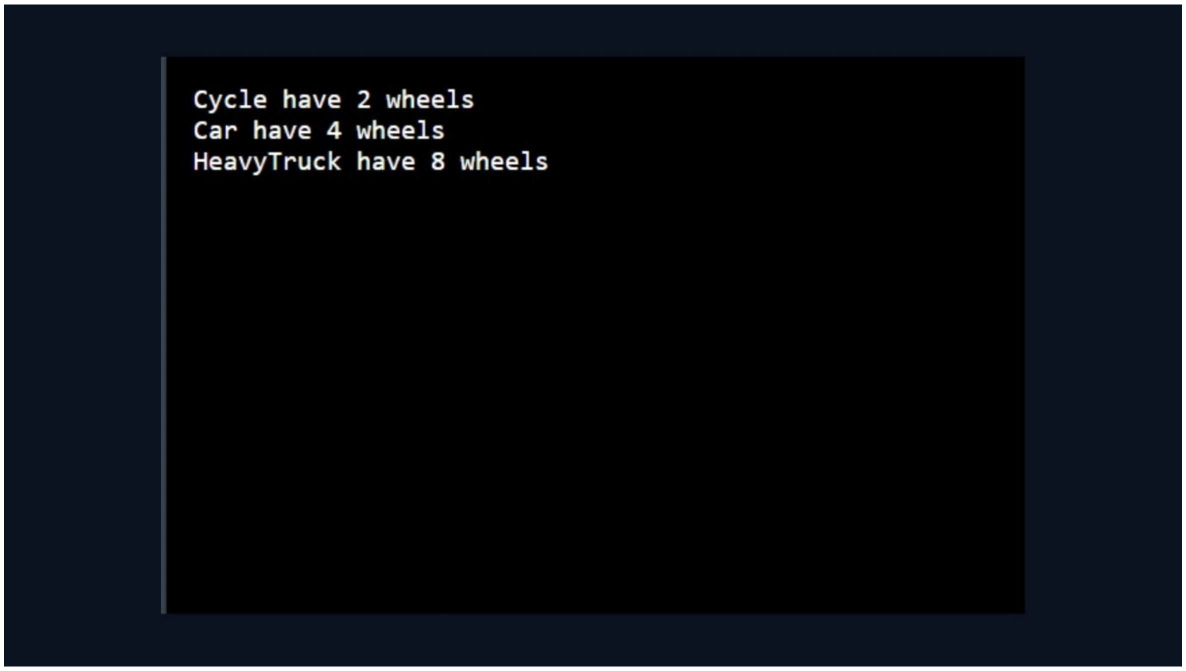


```
#Edcorner Learning Python OOPS

from abc import ABC
class ClassName(ABC):
```

We must import ABC class from the ABC module in order to use abstraction.



```
#Edcorner Learning Python OOPS

from abc import ABC

class Vehicle(ABC):  # inherits abstract class
    #abstract method
    def no_of_wheels(self):
        pass

class Cycle(Vehicle):
    def no_of_wheels(self): # provide definition for abstract method
        print("Cycle have 2 wheels")

class Car(Vehicle):
    def no_of_wheels(self):  # provide definition for abstract method
        print("Car have 4 wheels")

class HeavyTruck(Vehicle):  # provide definition for abstract method
    def no_of_wheels(self):
        print("HeavyTruck have 8 wheels")


Cycle = Cycle()
Cycle.no_of_wheels()
Car = Car()
Car.no_of_wheels()
HeavyTruck = HeavyTruck()
HeavyTruck.no_of_wheels()
```



```
Cycle have 2 wheels
Car have 4 wheels
HeavyTruck have 8 wheels
```

A vehicle-related abstract class is present here. Because it inherits from the abstract class ABC, it is abstract. No of wheels is an abstract method in the class Vehicle that lacks a definition since abstract methods are not declared (or abstract methods remain empty, However, other classes that derive from the Vehicle class, such as Cycle, Car, and HeavyTruck, define the method no of wheels and offer their own implementation. Assuming the Cycle has two wheels, the inherited abstract method no of wheels prints "Cycle have two wheels." The Car and HeavyTruck classes both offer their own implementations in a similar manner.

21 Certain noteworthy aspects of Abstract classes are:

- It is impossible to instantiate abstract classes. Simply put, we are unable to build objects for abstract classes.
- Both conventional and abstract methods can be found in an abstract class. We do not include any definitions or programming in the abstract methods. However, when using standard methods, we supply the method's implementation of the necessary code.

22 OOPS's benefits in Python

- Python's support for OOPS ideas has many benefits that make it suitable for the development of significant software. Let's examine a couple of them:
- Effective problem-solving is possible because we create classes that do the necessary actions for each mini-problem. The next problem can be solved even more quickly because we can reuse those classes.
- flexibility in having different versions of the same class thanks to polymorphism
- high level of code complexity was reduced through abstraction.
- By encapsulating, high security and data privacy are achieved.
- code reuse caused by a child class's inheritance of parent class properties.
- Instead than sifting through hundreds of lines of code to locate a single problem, modularity of programming makes debugging simple.

Lets starts Python OOPS Programming Exercises

Module 1 Class Method - Decorator

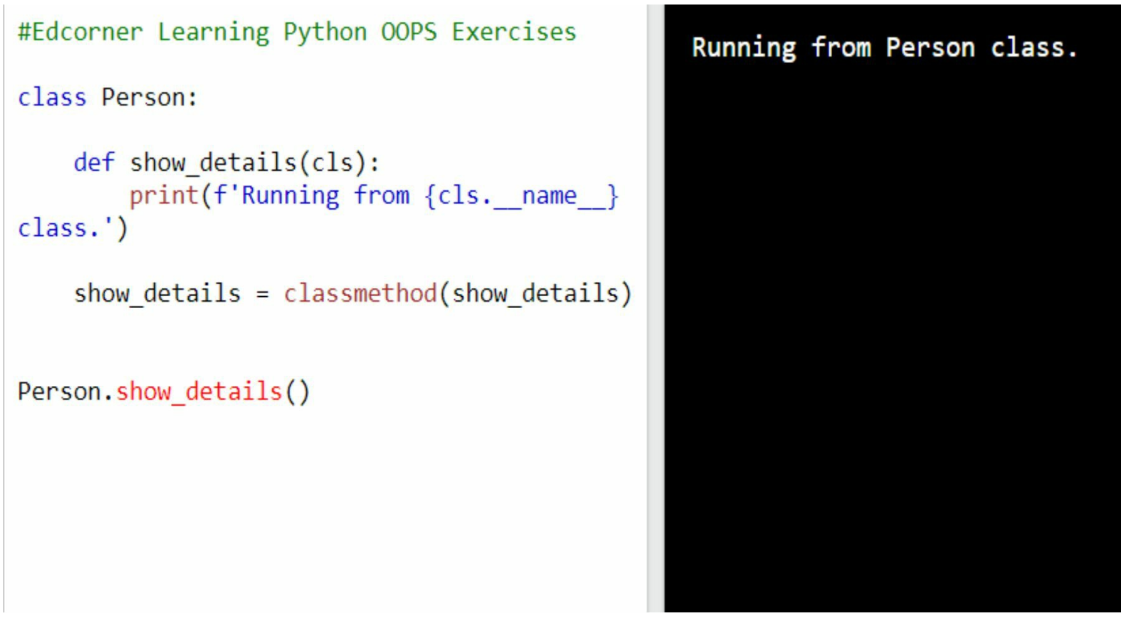
1. Using the classmethod class (do it in the standard way) implement a class named Person that has a class method named show_details() which displays the following text to the console:

'Running from Person class.'

Try to pass the class name using the appropriate attribute of the Person class. In response, call the show_details() class method.

Expected result:

Running from Person class.



```
#Edcorner Learning Python OOPS Exercises

class Person:

    def show_details(cls):
        print(f'Running from {cls.__name__}
class.')

    show_details = classmethod(show_details)


Person.show_details()
```

```
Running from Person class.
```

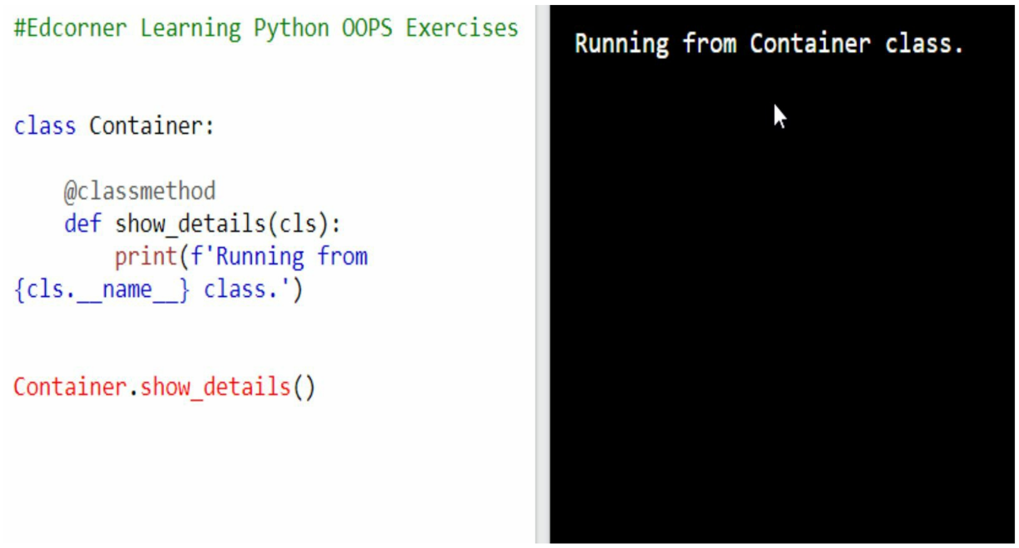
2. Using the classmethod class (do it in the standard way) implement a class named Person that has a class method named show_details() which displays the following text to the console:

'Running from Container class.'

Try to pass the class name using the appropriate attribute of the Person class. In response, call the show_details() class method.

Expected result:

'Running from Container class.'



```
#Edcorner Learning Python OOPS Exercises


class Container:

    @classmethod
    def show_details(cls):
        print(f'Running from
{cls.__name__} class.')


Container.show_details()
```

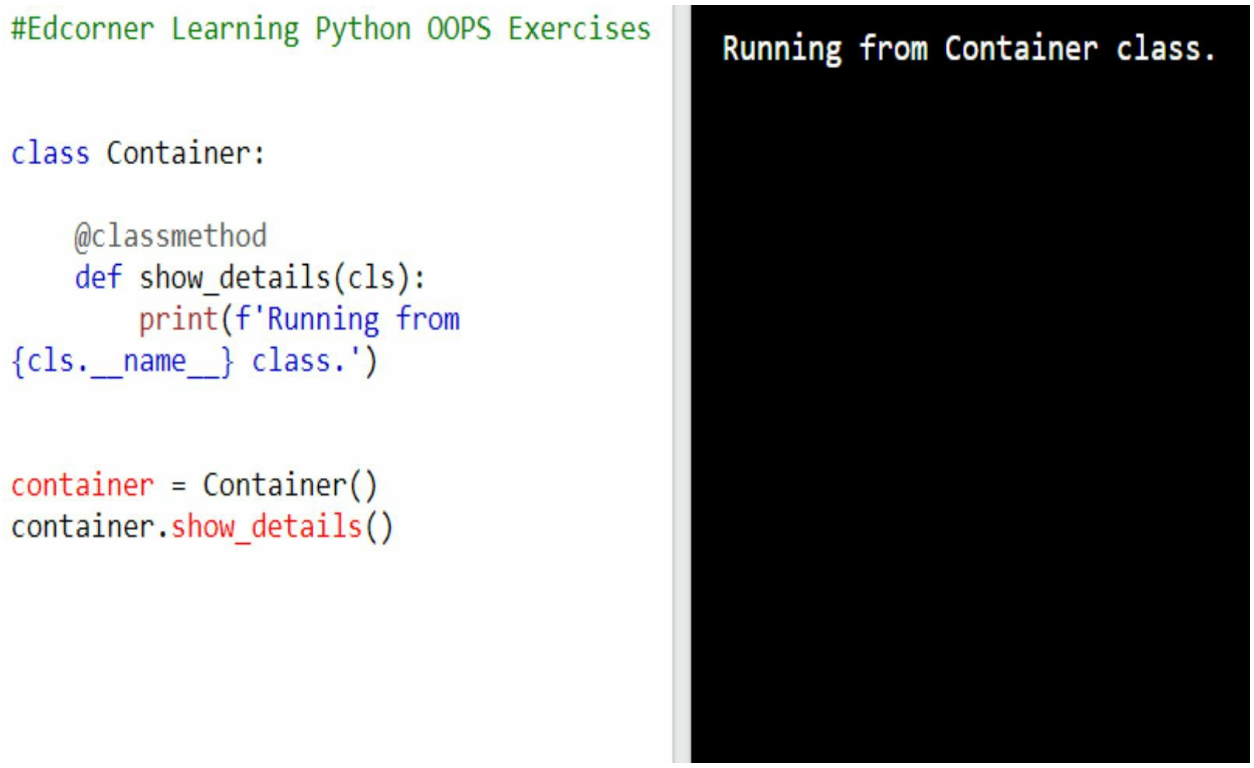
```
Running from Container class.
```

3. The Container class is given. Create an instance of this class named container and call the show_details( ) method from this instance.

Expected result:

Running from Container class.

```
class Container:
    @classmethod
    def show_details(cls):
```



```
#Edcorner Learning Python OOPS Exercises


class Container:

    @classmethod
    def show_details(cls):
        print(f'Running from
{cls.__name__} class.')


container = Container()
container.show_details()
```

```
Running from Container class.
```

print(f'Running from {cls.__name__} class.')

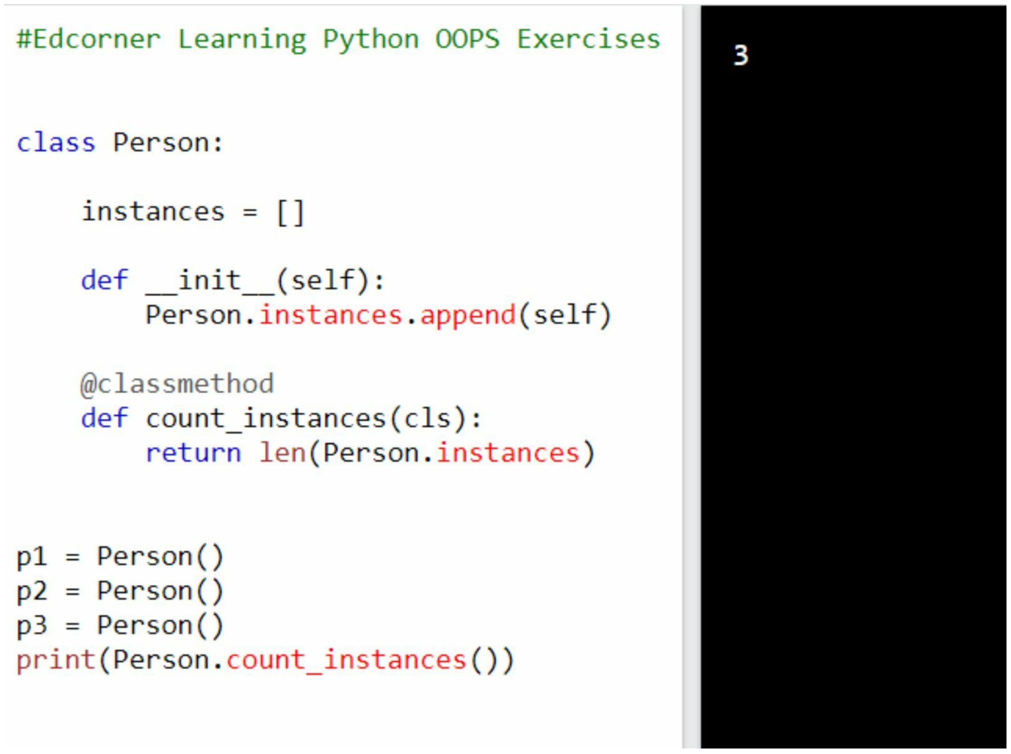
4. Implement a class named Person which has a class attribute named instances as an empty list. Then, each time you create an instance of the Person class, add it to the Person.instances list (use the init () method for this).

Also implement a class method called count_instances( ) that returns the number of Person objects created (the number of items in the Person.instances list).

Create three instances of the Person class. Then call the count_instances( ) class method and print result to the console.

Expected result:

3



```python
#Edcorner Learning Python OOPS Exercises


class Person:

    instances = []

    def __init__(self):
        Person.instances.append(self)

    @classmethod
    def count_instances(cls):
        return len(Person.instances)


p1 = Person()
p2 = Person()
p3 = Person()
print(Person.count_instances())
```

```
3
```

5. A class named Person is given. Modify the _init() method so that you can set two

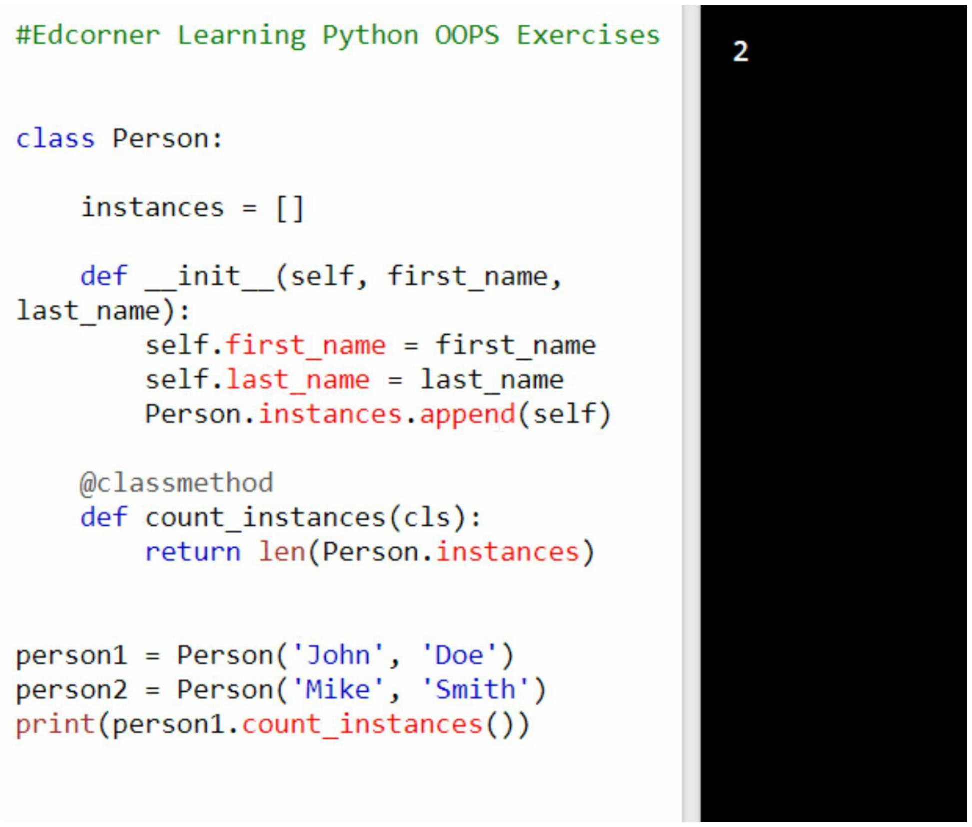
instance attributes: firstname and lastname (bare attributes, without any validation).

Create two instances of the Person class. Then call the count_instances() class method and print result to the console.

Expected result:

2

```
class Person:

    instances = []

    def __init__(self):
        Person.instances.append(self)

    @classmethod
    def count_instances(cls):
        return len(Person.instances)
```



```python
#Edcorner Learning Python OOPS Exercises


class Person:

    instances = []

    def __init__(self, first_name,
last_name):
        self.first_name = first_name
        self.last_name = last_name
        Person.instances.append(self)

    @classmethod
    def count_instances(cls):
        return len(Person.instances)


person1 = Person('John', 'Doe')
person2 = Person('Mike', 'Smith')
print(person1.count_instances())
```

```
2
```

Module 2 Static Method - Decorator

6. Define a Container class that has a static method (use the staticmethod class - do it in the standard way) named get_current_time( ) returning the current time in the format

‘ %H : %M : %S ' , e.g. '09:45:10' .

Tip: Use the built-in time module.

Solution:

```
import time


class Container:


    def get_current_time():
        return time.strftime('%H:%M:%S', time.localtime())


    get_current_time = staticmethod(get_current_time)
```

7. Define a Container class that has a static method (use the @staticmethod decorator) named get_current_time() returning the current time in the format '%h:%m:%s' , e.g. '09:45:10' .

Tip: Use the built-in time module.

Solution :

```
import time



class Container:

    @staticmethod
    def get_current_time():
        return time.strftime('%H:%M:%S', time.localtime())
```

8. Complete the implementation of the Book class. In the _init_ ( ) method, set the bare attributes of the instance with names:

- title
- author
- book_id

Set the instance book_id attribute using the uuid module. Exactly the uuid. uuid4( ) function from this module. An example of using this function:

import uuid

str(uuid.uuid4().fields[-1])[: 6]

Returns a 6-element string. This will be the value of the bookjd attribute.

Using the above code, create a static method of the Book class (use the @staticmethod decorator) called get_id() L which will generate a 6-digit str object (the value of the bookjd field).

Then create an instance of the class named bookl with the following arguments:

- title=' Python Object Oriented Programming Exercises Volume 2'
- author='Edcorner Learning'

In response, print all the _dict_ attribute keys of book1 to the console.

Expected result:

dict_keys(['book_id', 'title', 'author'])

```
import uuid


class Book:


    def __init__(self, title, author):
        pass
```

Solution:

```
import uuid

class Book:

    def __init__(self, title, author):
        self.book_id = self.get_id()
        self.title = title
        self.author = author




    @staticmethod
    def get_id():
        return str(uuid.uuid4().fields[-1])[:6]

book1 = Book(' Python Object Oriented Programming Exercises Volume 2', 'Edcorner Learning')
print(book1.__dict__.keys())
```

9. The Book class is implemented. Add a _repr_( ) method to the Book class that represents an instance of this class (see below).

Then create an instance of the class named book 1 passing the following arguments:

- title= ‘Python Object Oriented Programming Exercises Volume 2'
- author='Edcorner Learning'

In response, print the instance book1 to the console.

Expected result:

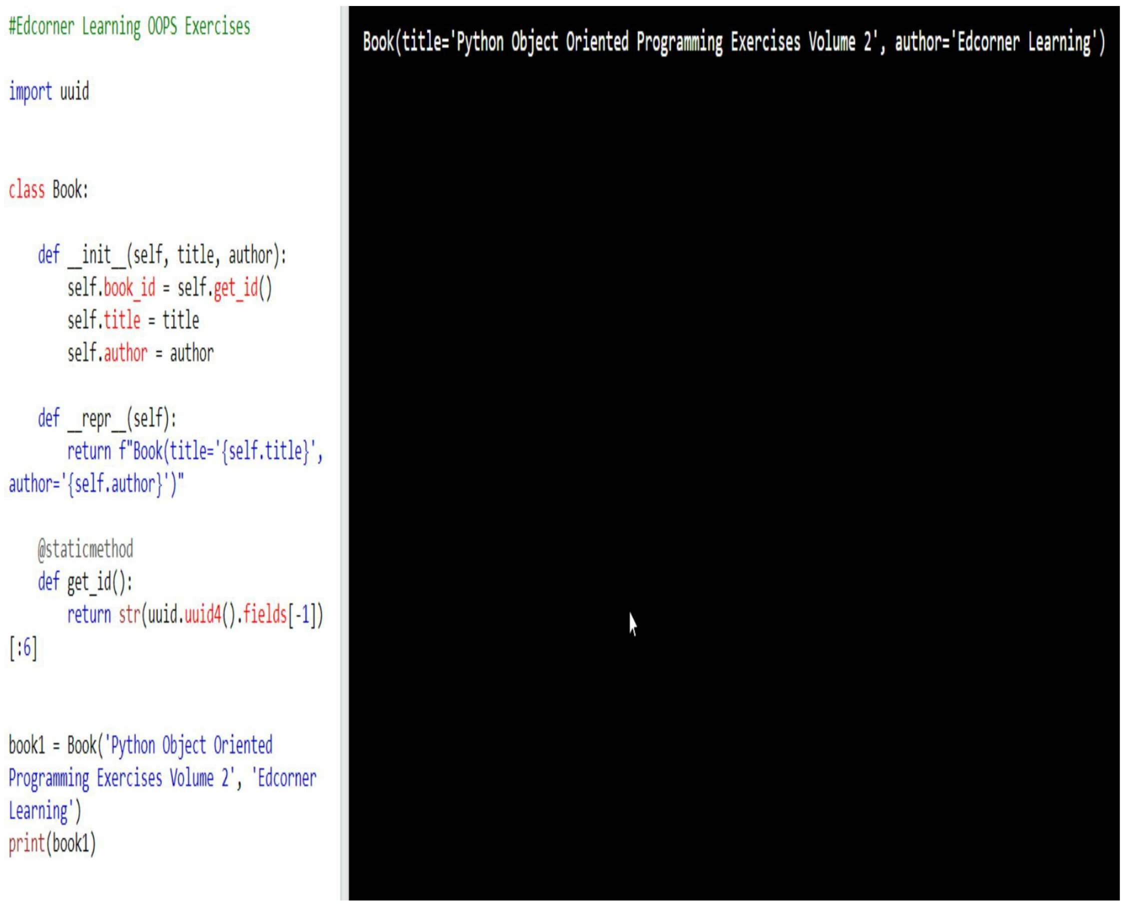
Book(title=' Python Object Oriented Programming Exercises Volume 2', author='Edcorner Learning')

```
import uuid
class Book:

    def __init__(self, title, author):
        self.book_id = self.get_id()
        self.title = title
        self.author = author
    @staticmethod
    def get_id():
        return str(uuid.uuid4().fields[-1])[:6]
```

Solution:

```
import uuid
class Book:

    def __init__(self, title, author):
        self.book_id = self.get_id()
        self.title = title
        self.author = author

    def __repr__(self):
        return f"Book(title='{self.title}', author='{self.author}')"

    @staticmethod
    def get_id():
        return str(uuid.uuid4().fields[-1])[:6]
book1 = Book(' Python Object Oriented Programming Exercises
Volume 2', author='Edcorner Learning')
```

print(book1)



```
#Edcorner Learning OOPS Exercises

import uuid


class Book:

    def __init__(self, title, author):
        self.book_id = self.get_id()
        self.title = title
        self.author = author

    def __repr__(self):
        return f"Book(title='{self.title}',
author='{self.author}')"

    @staticmethod
    def get_id():
        return str(uuid.uuid4().fields[-1])
[:6]


book1 = Book('Python Object Oriented
Programming Exercises Volume 2', 'Edcorner
Learning')
print(book1)
```

```
Book(title='Python Object Oriented Programming Exercises Volume 2', author='Edcorner Learning')
```

Module 3 Special Methods

10. Define a Person class that takes two bare attributes: fname (first name) and Iname (last name).

Then implement the _repr_ ( ) special method to display a formal representation of the

Person object as shown below:

[IN]: person = Person('John', 'Doe')

[IN]: print(person)

[OUT]: Person(fname='John', lname='Doe')

Create an instance of the Person class with the given attributes:

fname = 'Mike'

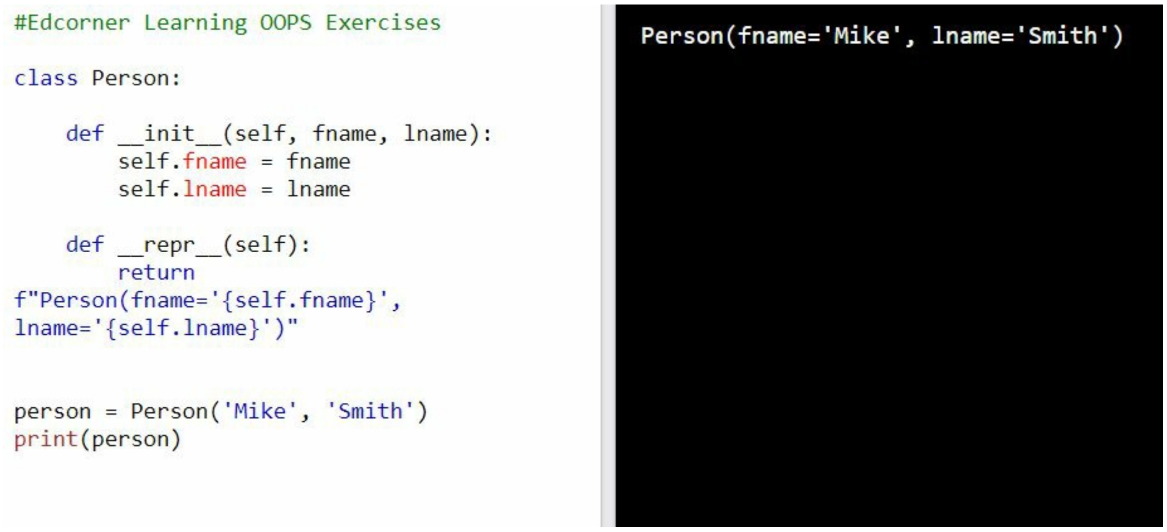
• lname = 'Smith'

and assign it to the variable person. In response, print the person instance to the console.

Expected result:

Person(fname='Mike', lname='Smith')

Solution:



```
#Edcorner Learning OOPS Exercises

class Person:

    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname

    def __repr__(self):
        return
f"Person(fname='{self.fname}',
lname='{self.lname}')"


person = Person('Mike', 'Smith')
print(person)
```

```
Person(fname='Mike', lname='Smith')
```

11. The Person class is implemented. Add a special method _str_ () to return an informal

representation of an instance of the Person class.

Example:

[IN]: person = Person('Mike', 'Smith')

[IN]: print(person)

First name: Mike

Last nane: Smith

Then create an instance named person with the following values:

- fname = 'Edcorner'
- lname = 'Learning'

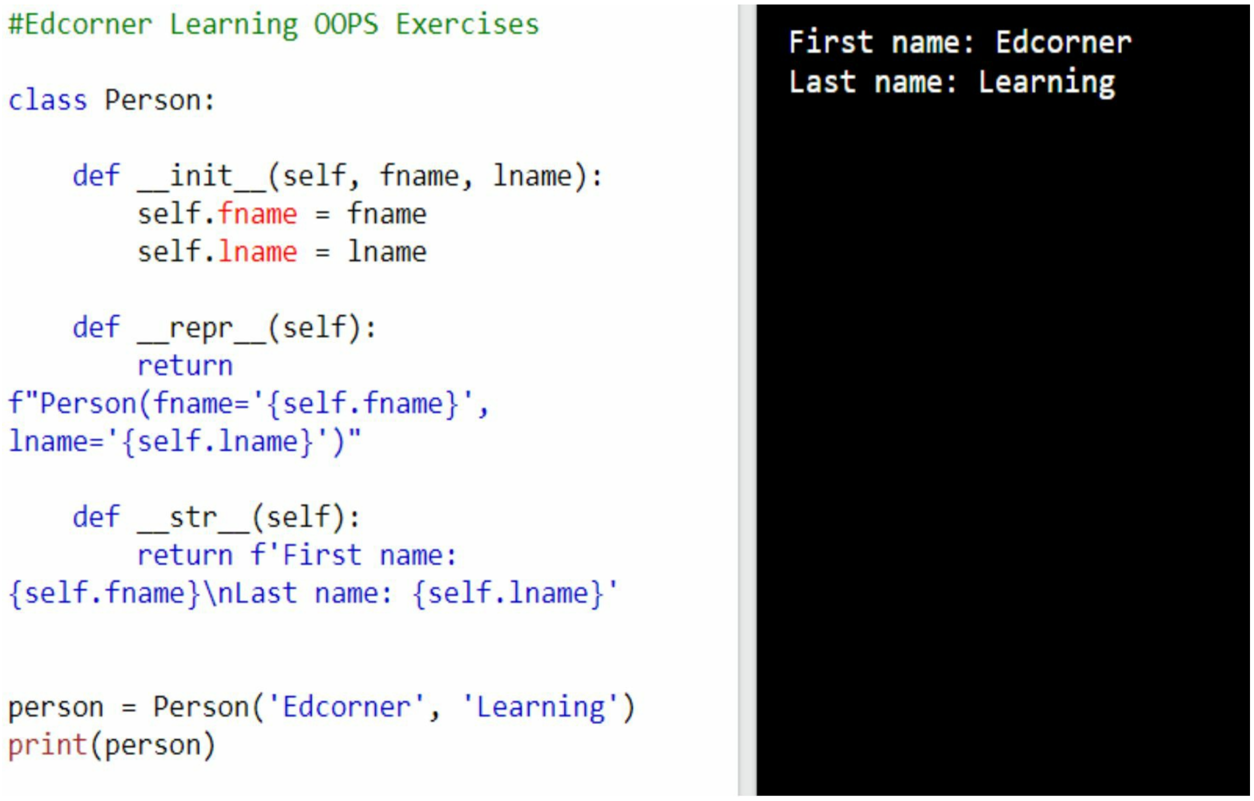
In response, print the person instance to the console.

Expected result:

First name: Edcorner

Last name: Learning

```
    class Person:


  def __init__(self, fname, lname):
      self.fname = fname
      self.lname = lname



def __repr__(self):
return f"Person(fname='{self.fname}', lname='{self.lname}')"
```



```
#Edcorner Learning OOPS Exercises

class Person:

    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname

    def __repr__(self):
        return
f"Person(fname='{self.fname}',
lname='{self.lname}')"

    def __str__(self):
        return f'First name:
{self.fname}\nLast name: {self.lname}'


person = Person('Edcorner', 'Learning')
print(person)
```

```
First name: Edcorner
Last name: Learning
```

Solution:

12. Implement a class named Vector which, when creating an instance takes any number of arguments (vector coordinates in n-dimensional space - without any validation) and assign it to an attribute named components.

Then implement the _repr_( ) special method to display a formal representation of Vector as

shown below:

[IN]: vl = Vector(l, 2)

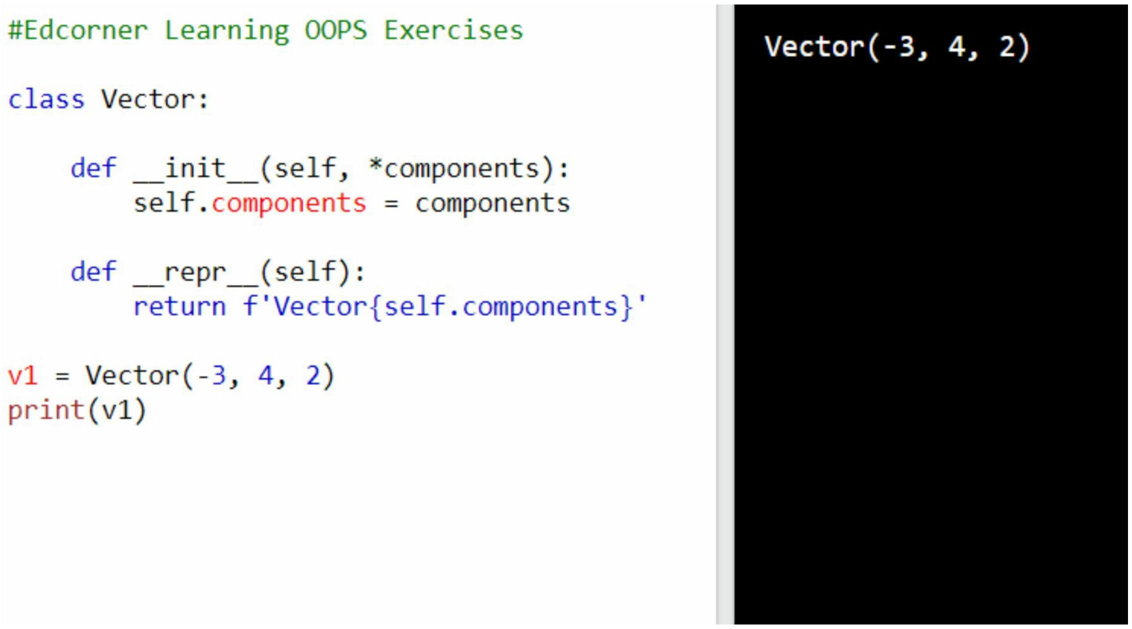
[IN]: print(vl)

[Out]: Vector(l, 2)

Create a vector from the R^3 space with coordinates: (-3,4,2) and assign it to the variable v7. In response, print the variable vl to the console.

Expected result:

Vector(-3, 4, 2)



```
#Edcorner Learning OOPS Exercises

class Vector:

    def __init__(self, *components):
        self.components = components

    def __repr__(self):
        return f'Vector{self.components}'

v1 = Vector(-3, 4, 2)
print(v1)
```

```
Vector(-3, 4, 2)
```

13. An implementation of the Vector class is given. Implement the _str_()

display an informal representation of a Vector instance as shown below:

[IN]: vl = Vector(l, 2)

[IN]: print(vl)

[Out]: (1, 2)

special method to

Create a vector from the RA3 space with coordinates: (-3,4,2) and assign it to the variable vl.

In response, print the variable vl to the console.

Expected result:

(-3, 4, 2)

```
class Vector:


    def __init__(self, *components):
        self.components = components
    def __repr__(self):
        return f'Vector{self.components}'
```

Solution:

```
class Vector:

    def __init__(self, *components):
        self.components = components

    def __repr__(self):
        return f'Vector{self.components}'

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

v1 = Vector(-3, 4, 2)
print(v1)
```

14. An implementation of the Vector class is given. Implement the _len_ ( ) special method to return the number of vector coordinates.

Example:

[IN]: vl = Vector(3, 4, 5)

[IN]: print(len(vl))

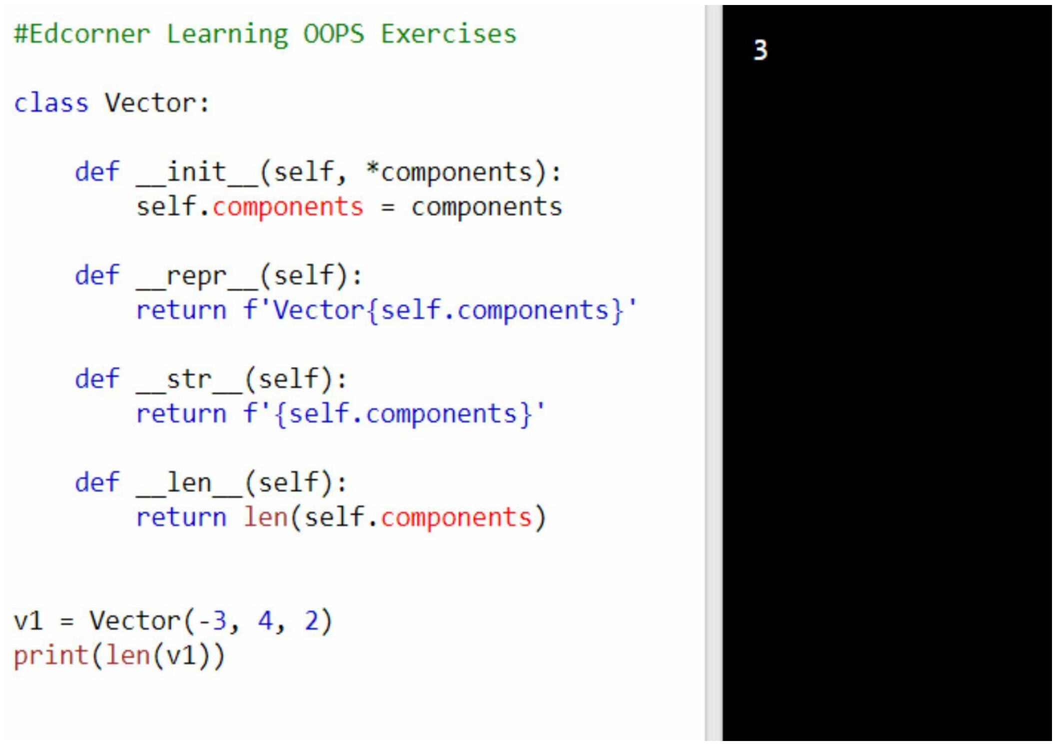
[Out]: 3

Create a vector from the RA3 space with coordinates: (-3,4,2) and assign it to the variable v7. In response, print the number of coordinates of this vector using the built-in len() function.

Expected result:

3

```
class Vector:
    def __init__(self, *components):
        self.components = components


    def __repr__(self):
        return f'Vector{self.components}'


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



```
#Edcorner Learning OOPS Exercises

class Vector:

    def __init__(self, *components):
        self.components = components

    def __repr__(self):
        return f'Vector{self.components}'

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

    def __len__(self):
        return len(self.components)


v1 = Vector(-3, 4, 2)
print(len(v1))
```

```
3
```

15. An implementation of the Vector class is given. Implement the _bool_( ) special method to return the logical value of vector:

- False in case the first coordinate is zero
- on the contrary True

If the user doesn't pass any argument, return the logical value False.

Example

[IN]: vl = Vector(0, 4, 5)

[IN]: print(bool(vl))

[Out]: False

Then create the following instances:

- vl = Vector()
- v2 = Vector(3, 2)

v3 = Vector(0, -3, 2)

v4 = Vector(5, 0, -1)

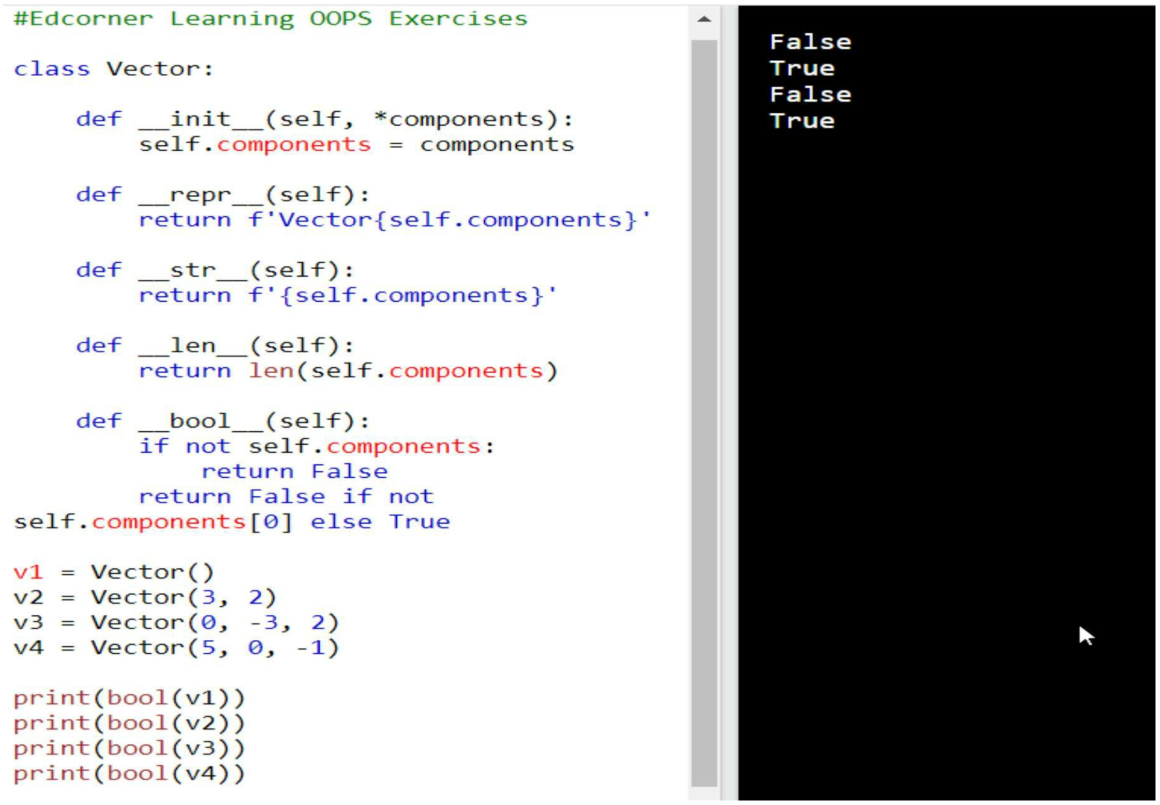
In response, print the logical value of the given instances to the console.

Expected result:

```
False

True

False

True
```

```
class Vector:


    def __init__(self, *components):
        self.components = components


    def __repr__(self):
        return f'Vector{self.components}'
```

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

def __len__(self):
    return len(self.components)
```



```
#Edcorner Learning OOPS Exercises

class Vector:

    def __init__(self, *components):
        self.components = components

    def __repr__(self):
        return f'Vector{self.components}'

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

    def __len__(self):
        return len(self.components)

    def __bool__(self):
        if not self.components:
            return False
        return False if not
self.components[0] else True

v1 = Vector()
v2 = Vector(3, 2)
v3 = Vector(0, -3, 2)
v4 = Vector(5, 0, -1)

print(bool(v1))
print(bool(v2))
print(bool(v3))
print(bool(v4))
```

```
False
True
False
True
```

16. An implementation of the Vector class is given. Create the following instances of this class:

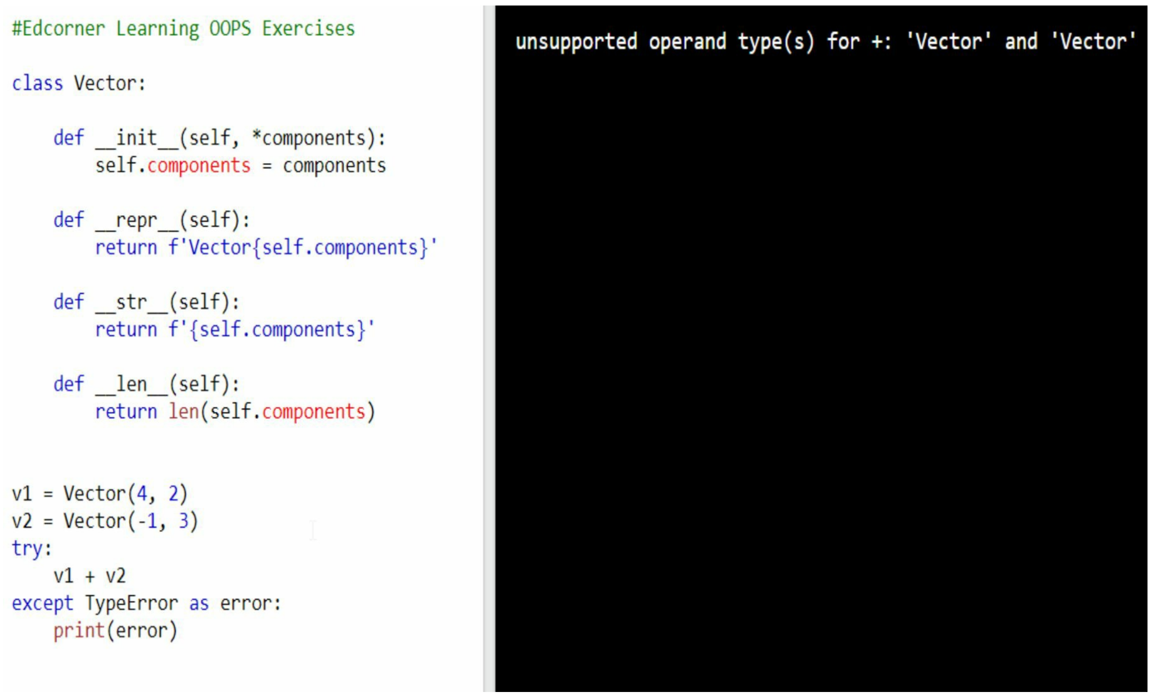
- vl = Vector(4, 2)
- v2 = Vector(-l, 3)

Then try to add these instances, i.e. perform the operation v1 + v2 . If there is an error, print the error message to the console. Use a try ... except ... clause in your solution.

Expected result:

unsupported operand type(s) for +: 'Vector' and 'Vector'

```
class Vector:

    def __init__(self, *args):
        self.components = args

    def __repr__(self):
        return f"Vector{self.components}"

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

    def __len__(self):
        return len(self.components)
```



```
#Edcorner Learning OOPS Exercises

class Vector:

    def __init__(self, *components):
        self.components = components

    def __repr__(self):
        return f'Vector{self.components}'

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

    def __len__(self):
        return len(self.components)


v1 = Vector(4, 2)
v2 = Vector(-1, 3)
try:
    v1 + v2
except TypeError as error:
    print(error)
```

```
unsupported operand type(s) for +: 'Vector' and 'Vector'
```

17. An implementation of the Vector class is given. Implement the _add_( ) special method to add Vector instances (by coordinates). For simplicity, assume that the user adds vectors of the same length. Then create two instances of the Vector class:

vl = Vector(4, 2)

v2 = Vector(-l, 3)

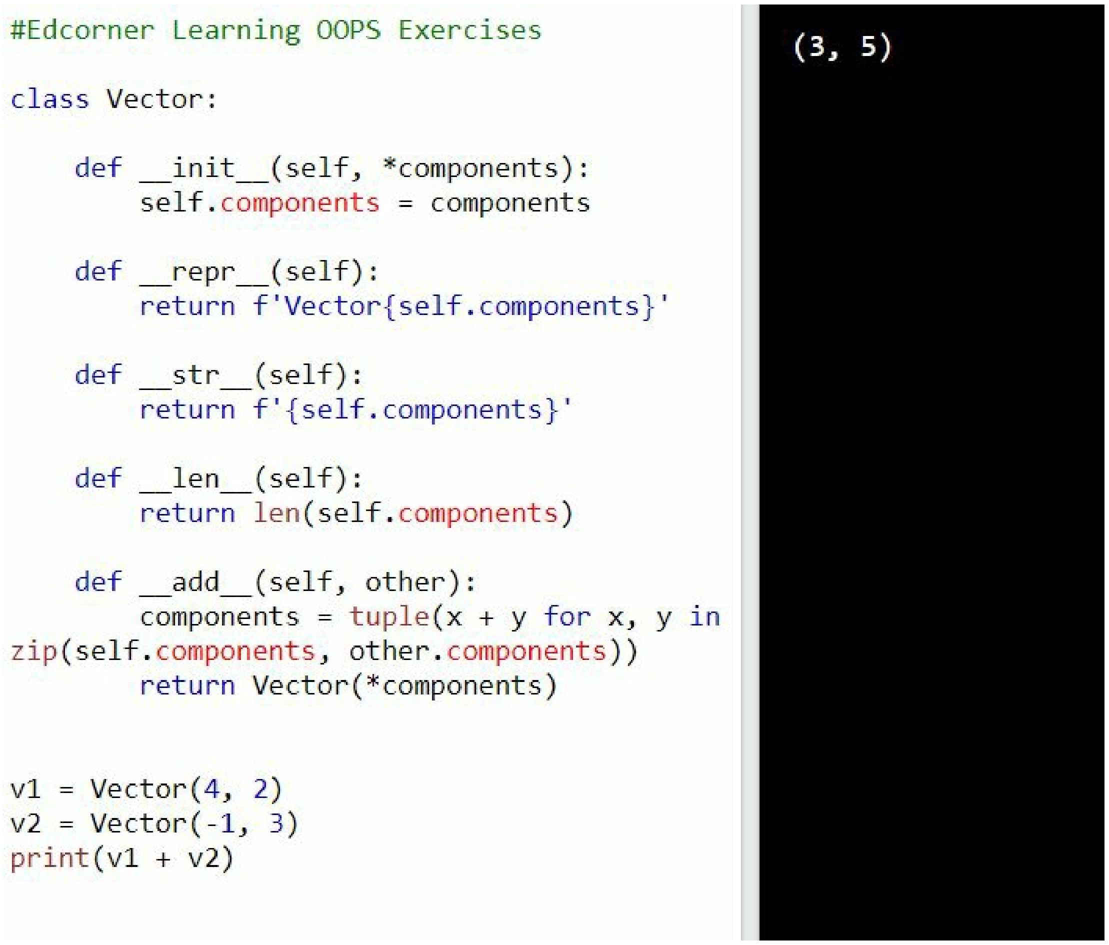
and perform the addition of these vectors. Print the result to the console.

Expected result:

(3,5)

class Vector:

```
def __init__(self, *components):
    self.components = components

def __repr__(self):
    return f'Vector{self.components}'

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

def __len__(self):
    return len(self.components)
```



```
#Edcorner Learning OOPS Exercises

class Vector:

    def __init__(self, *components):
        self.components = components

    def __repr__(self):
        return f'Vector{self.components}'

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

    def __len__(self):
        return len(self.components)

    def __add__(self, other):
        components = tuple(x + y for x, y in
zip(self.components, other.components))
        return Vector(*components)


v1 = Vector(4, 2)
v2 = Vector(-1, 3)
print(v1 + v2)
```

```
(3, 5)
```

18. An implementation of the Vector class is given. Create the following instances of this class:

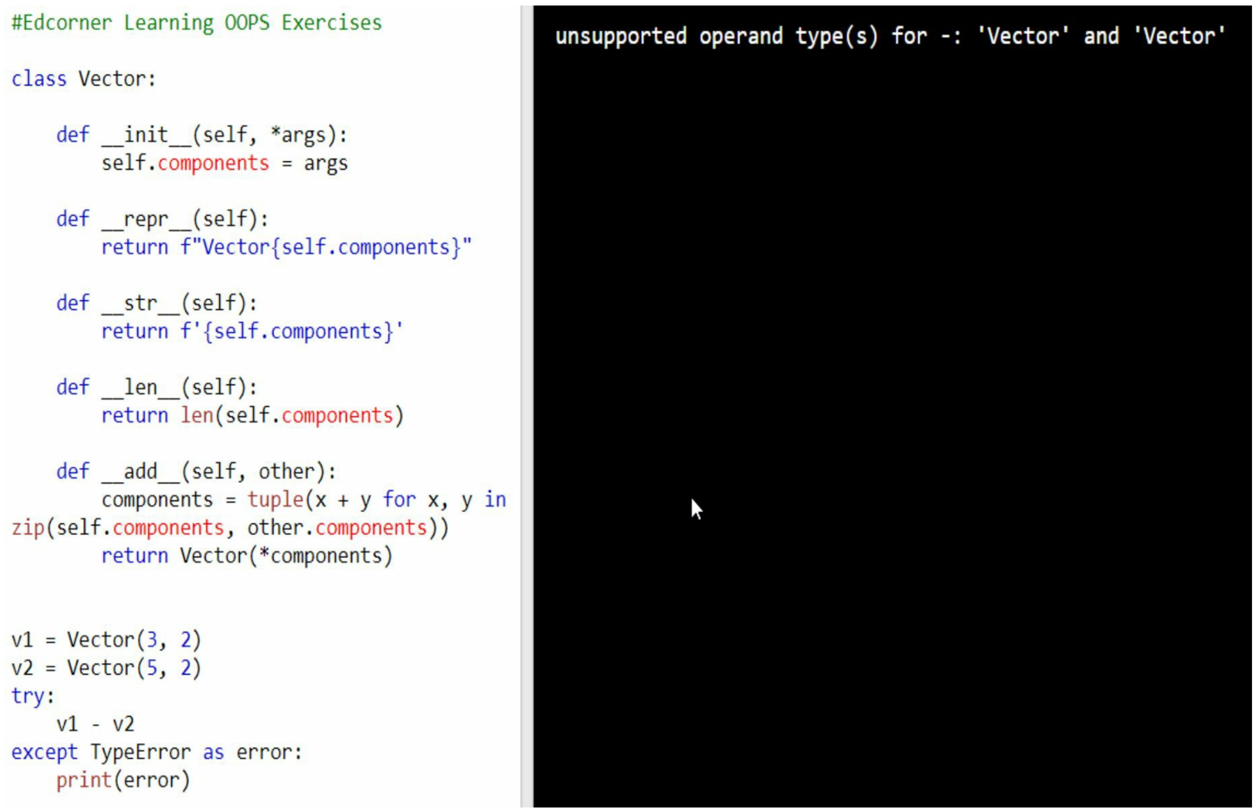
- vl = Vector(4, 2)
- v2 = Vector(-l, 3)

Then try to subtract these instances (perform the v1 - v2 operation). If there is an error, print the error message to the console. Use a try ... except ... clause in your solution.

Expected result:

unsupported operand type(s) for 'Vector' and 'Vector'

```
class Vector:

    def __init__(self, *args):
        self.components = args

    def __repr__(self):
        return f"Vector{self.components}"

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

    def __len__(self):
        return len(self.components)

            def __add__(self, other):
        components = tuple(x + y for x, y in zip(self.components,
other.components))
        return Vector(*components)
```



```
#Edcorner Learning OOPS Exercises

class Vector:

    def __init__(self, *args):
        self.components = args

    def __repr__(self):
        return f"Vector{self.components}"

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

    def __len__(self):
        return len(self.components)

    def __add__(self, other):
        components = tuple(x + y for x, y in
zip(self.components, other.components))
        return Vector(*components)


v1 = Vector(3, 2)
v2 = Vector(5, 2)
try:
    v1 - v2
except TypeError as error:
    print(error)
```

```
unsupported operand type(s) for -: 'Vector' and 'Vector'
```

Solution:

19. An implementation of the Vector class is given. Implement the _sub_ () special method that

subtracts Vector instances (by coordinates). For simplicity, assume that the user subtracts vectors of the same length. Then create two instances of this class:

- vl = Vector(4, 2)
- v2 = Vector(-l, 3)

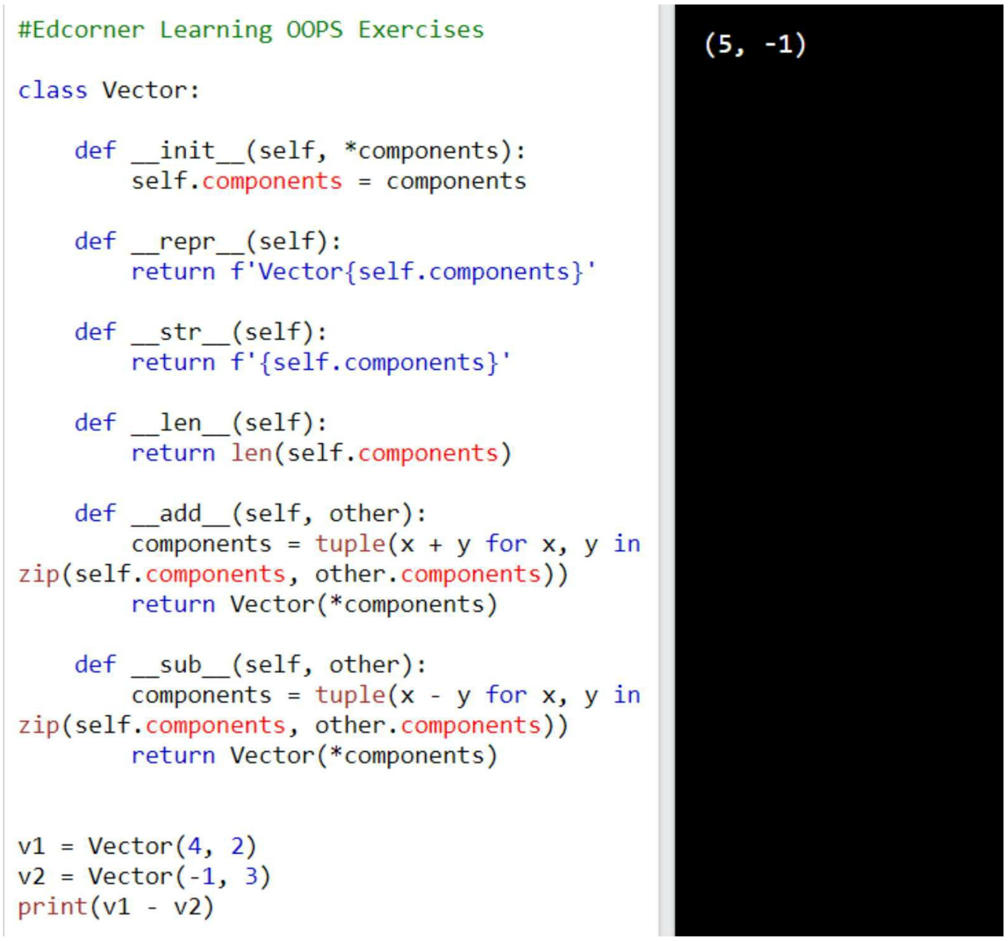
and perform the subtraction of these vectors. Print the result to the console.

Expected result:

```
(5, -1)
```

```
class Vector:


    def __init__(self, *components):
        self.components = components


    def __repr__(self):
        return f'Vector{self.components}'


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


    def __len__(self):
         return len(self.components)
            def __add__(self, other):
                components = tuple(x + y for x, y in zip(self.components,
         other.components))
                return Vector(*components)
```

Soluton:



```
#Edcorner Learning OOPS Exercises

class Vector:

    def __init__(self, *components):
        self.components = components

    def __repr__(self):
        return f'Vector{self.components}'

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

    def __len__(self):
        return len(self.components)

    def __add__(self, other):
        components = tuple(x + y for x, y in
zip(self.components, other.components))
        return Vector(*components)

    def __sub__(self, other):
        components = tuple(x - y for x, y in
zip(self.components, other.components))
        return Vector(*components)


v1 = Vector(4, 2)
v2 = Vector(-1, 3)
print(v1 - v2)
```

```
(5, -1)
```

20. An implementation of the Vector class is given. Implement the _mui ( ) special method that allows you to multiply Vector instances (by coordinates). For simplicity, assume that the user multiplies vectors of the same length. Then create two instances of this class:

- vl = Vector(4, 2)
- v2 = Vector(-l, 3)

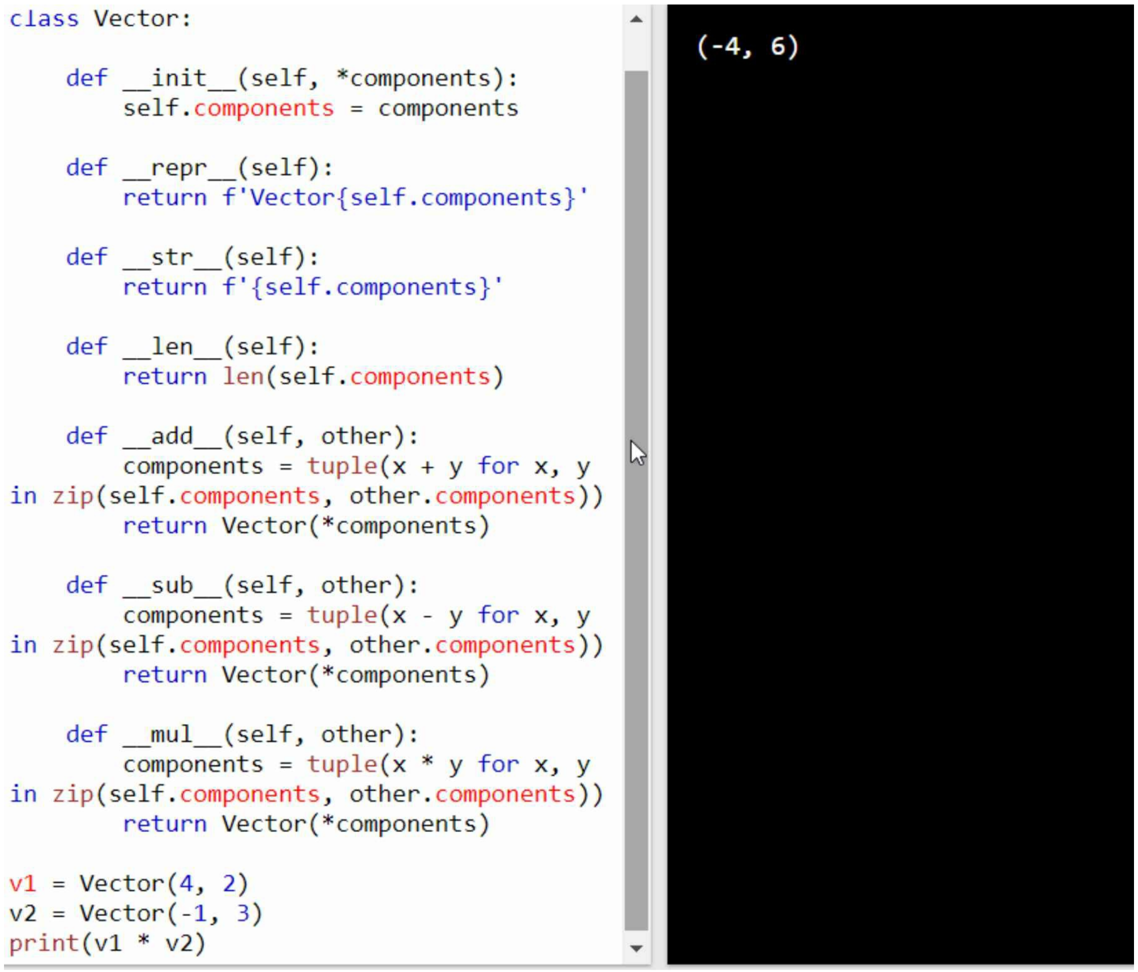
and perform the multiplication of these vectors. Print the result to the console.

Expected result:

(-4,6)

```
class Vector:

    def __init__(self, *components):
        self.components = components

    def __repr__(self):
        return f'Vector{self.components}'

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

    def __len__(self):
return len(self.components)

def __add__(self, other):
    components = tuple(x + y for x, y in zip(self.components,
other.components))
    return Vector(*components)

def __sub__(self, other):
    components = tuple(x - y for x, y in zip(self.components,
other.components))
    return Vector(*components)
```

Solution:



```
class Vector:

    def __init__(self, *components):
        self.components = components

    def __repr__(self):
        return f'Vector{self.components}'

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

    def __len__(self):
        return len(self.components)

    def __add__(self, other):
        components = tuple(x + y for x, y
in zip(self.components, other.components))
        return Vector(*components)

    def __sub__(self, other):
        components = tuple(x - y for x, y
in zip(self.components, other.components))
        return Vector(*components)

    def __mul__(self, other):
        components = tuple(x * y for x, y
in zip(self.components, other.components))
        return Vector(*components)

v1 = Vector(4, 2)
v2 = Vector(-1, 3)
print(v1 * v2)
```

```
(-4, 6)
```

21. An implementation of the Vector class is given. Implement the _truediv( ) special method

which allows you to divide Vector instances (division by coordinates). For simplicity, assume that the user divides vectors of the same length and the coordinates of the second vector are not zeros. Then create two instances of this class:

- vl = Vector(4, 2)
- v2 = Vector(-l, 4)

and perform the division of these vectors. Print the result to the console.

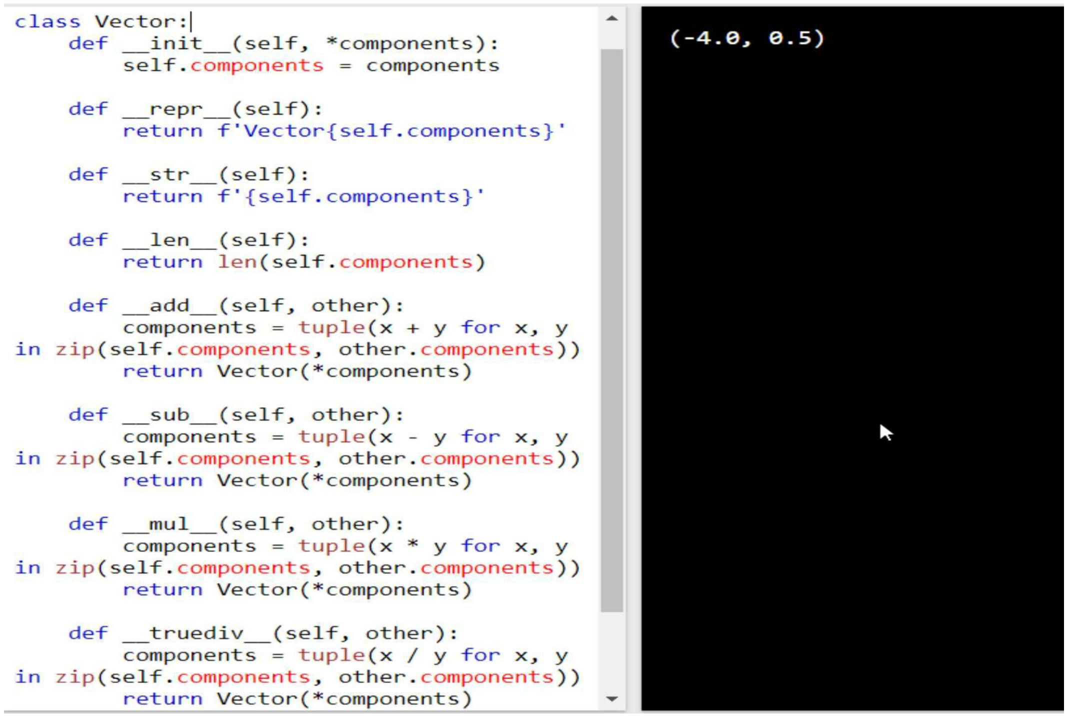
Expected result:

(-4.0, 0.5)

```
class Vector:

    def __init__(self, *components):
        self.components = components

    def __repr__(self):
        return f'Vector{self.components}'

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

        def __len__(self):
            return len(self.components)

        def __add__(self, other):
            components = tuple(x + y for x, y in zip(self.components,
         other.components))
            return Vector(*components)

        def __sub__(self, other):
            components = tuple(x - y for x, y in zip(self.components,
         other.components))
            return Vector(*components)
        def __mul__(self, other):
            components = tuple(x * y for x, y in zip(self.components,
         other.components))
            return Vector(*components)
```

Soluton:

```
class Vector:

    def __init__(self, *components):
        self.components = components

    def __repr__(self):
        return f'Vector{self.components}'

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

def __len__(self):
    return len(self.components)

def __add__(self, other):
    components = tuple(x + y for x, y in zip(self.components,
other.components))
    return Vector(*components)

def __sub__(self, other):
        components = tuple(x - y for x, y in zip(self.components, other.components))
        return Vector(*components)

    def __mul__(self, other):
        components = tuple(x * y for x, y in zip(self.components, other.components))
        return Vector(*components)

    def __truediv__(self, other):
        components = tuple(x / y for x, y in zip(self.components, other.components))
        return Vector(*components)


v1 = Vector(4, 2)
v2 = Vector(-1, 4)
```



```
class Vector:
    def __init__(self, *components):
        self.components = components

    def __repr__(self):
        return f'Vector{self.components}'

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

    def __len__(self):
        return len(self.components)

    def __add__(self, other):
        components = tuple(x + y for x, y
in zip(self.components, other.components))
        return Vector(*components)

    def __sub__(self, other):
        components = tuple(x - y for x, y
in zip(self.components, other.components))
        return Vector(*components)

    def __mul__(self, other):
        components = tuple(x * y for x, y
in zip(self.components, other.components))
        return Vector(*components)

    def __truediv__(self, other):
        components = tuple(x / y for x, y
in zip(self.components, other.components))
        return Vector(*components)
```

```
(-4.0, 0.5)
```

print(v1 / v2)

22. An implementation of the Vector class is given. Implement the _f loordiv_ ( ) special method

to do the integer division of Vector instances (division by coordinates). For simplicity, assume that the user divides vectors of the same length and the coordinates of the second vector are not zeros. Then create two instances of this class:

- vl = Vector(4, 2)
- v2 = Vector(-1, 4)

and perform an integer division for these vectors. Print the result to the console.

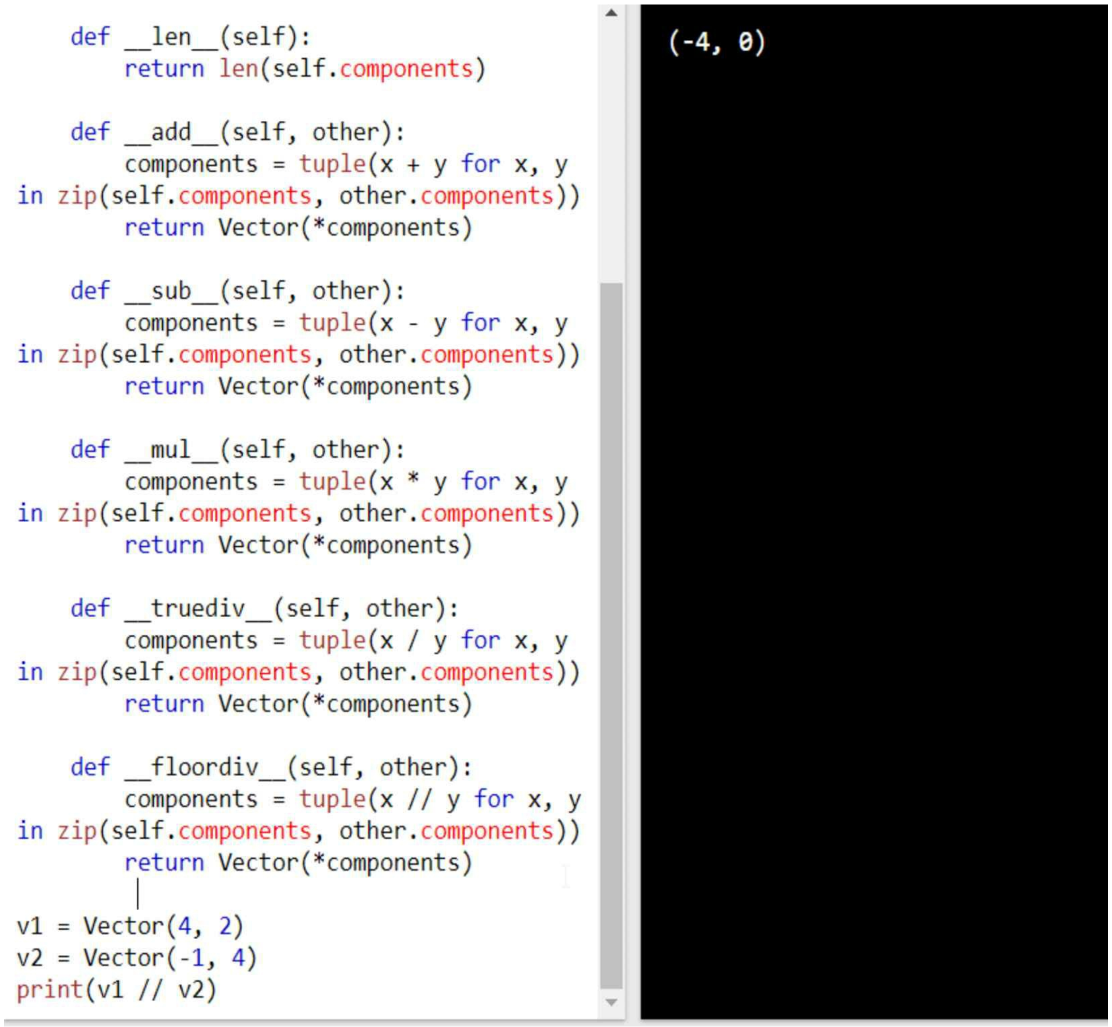
Expected result:

(-4, 0)

```
class Vector:

    def __init__(self, *components):
        self.components = components

    def __repr__(self):
        return f'Vector{self.components}'

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

    def __len__(self):
        return len(self.components)

    def __add__(self, other):
        components = tuple(x + y for x, y in zip(self.components,
other.components))
        return Vector(*components)

    def __sub__(self, other):
        components = tuple(x - y for x, y in zip(self.components,
other.components))
        return Vector(*components)
    def __mul__(self, other):
        components = tuple(x * y for x, y in zip(self.components,
other.components))
        return Vector(*components)

    def __truediv__(self, other):
        components = tuple(x / y for x, y in zip(self.components,
other.components))
        return Vector(*components)
```

Solution:

```
class Vector:

    def __init__(self, *components):
        self.components = components

    def __repr__(self):
        return f'Vector{self.components}'

    def __str__(self):
        return f'{self.components}'
    def __len__(self):
        return len(self.components)

    def __add__(self, other):
        components = tuple(x + y for x, y in zip(self.components,
other.components))
        return Vector(*components)

    def __sub__(self, other):
        components = tuple(x - y for x, y in zip(self.components,
other.components))
            return Vector(*components)

        def __mul__(self, other):
            components = tuple(x * y for x, y in zip(self.components,
        other.components))
            return Vector(*components)

        def __truediv__(self, other):
            components = tuple(x / y for x, y in zip(self.components,
        other.components))
            return Vector(*components)

        def __floordiv__(self, other):
            components = tuple(x // y for x, y in zip(self.components,
        other.components))
            return Vector(*components)


        v1 = Vector(4, 2)
        v2 = Vector(-1, 4)
        print(v1 // v2)
```



```
    def __len__(self):
        return len(self.components)

    def __add__(self, other):
        components = tuple(x + y for x, y
in zip(self.components, other.components))
        return Vector(*components)

    def __sub__(self, other):
        components = tuple(x - y for x, y
in zip(self.components, other.components))
        return Vector(*components)

    def __mul__(self, other):
        components = tuple(x * y for x, y
in zip(self.components, other.components))
        return Vector(*components)

    def __truediv__(self, other):
        components = tuple(x / y for x, y
in zip(self.components, other.components))
        return Vector(*components)

    def __floordiv__(self, other):
        components = tuple(x // y for x, y
in zip(self.components, other.components))
        return Vector(*components)

v1 = Vector(4, 2)
v2 = Vector(-1, 4)
print(v1 // v2)
```

```
(-4, 0)
```

23. The following Doc class is implemented for storing text documents. Implement the _add_ ( ) special method to add Doc instances with a space character.

Example:

[IN]: docl = Doc('Object')

```
[IN]: doc2 = Doc('Oriented')
[IN]: print(docl + doc2)


[OUT]: Object Oriented
```

Then create two instances of the Doc class for the following documents:

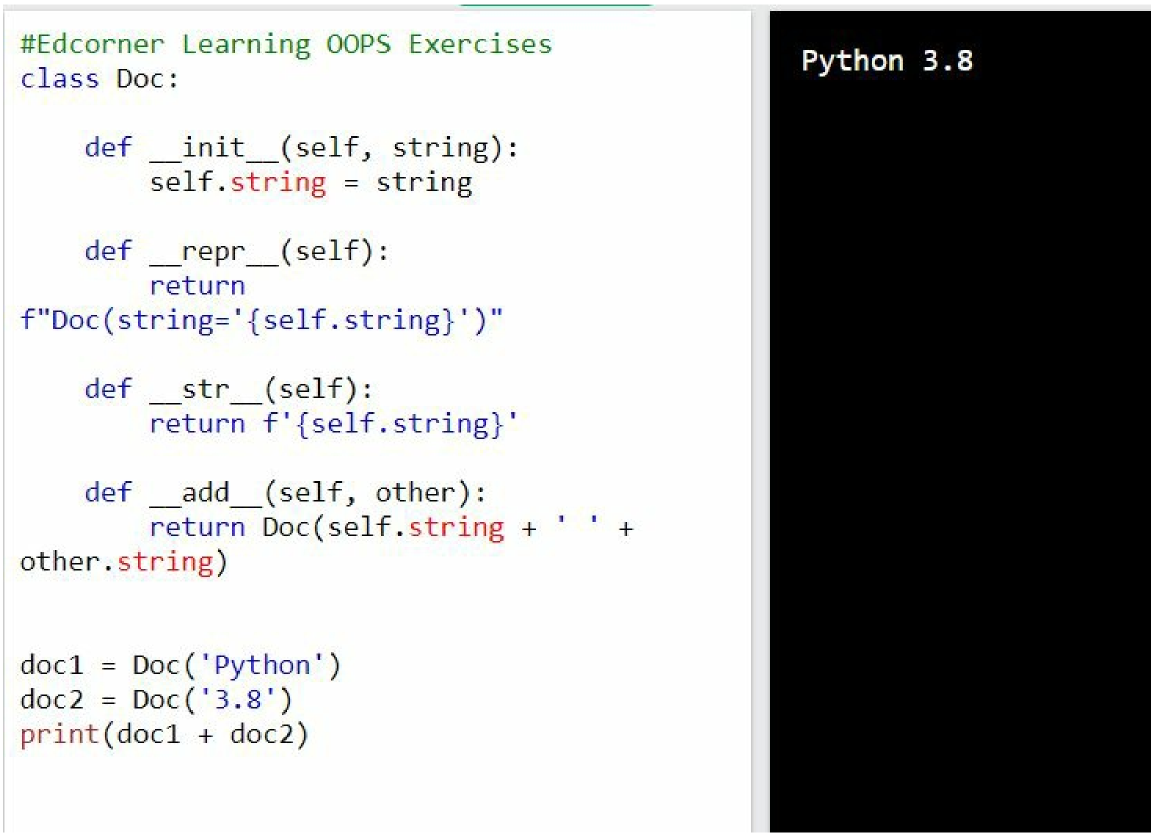
- 'Python'
- '3.8'

In response, print the result of adding these instances to the console.

Expected result:

Python 3.8

```
class Doc:


    def __init__(self, string):
        self.string = string




    def __repr__(self):
        return f"Doc(string='{self.string}')"


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



```
#Edcorner Learning OOPS Exercises
class Doc:

    def __init__(self, string):
        self.string = string

    def __repr__(self):
        return
f"Doc(string='{self.string}')"

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

    def __add__(self, other):
        return Doc(self.string + ' ' +
other.string)


doc1 = Doc('Python')
doc2 = Doc('3.8')
print(doc1 + doc2)
```

```
Python 3.8
```

24. The following Hashtag class is implemented for storing text documents - hashtags. Implement the _add_ () special method to add (concatenate) Hashtag instances using a space character as shown below (take into account the appropriate number of ■ # ■ characters at the beginning of the new object).

Example:

[IN]: hashtagl = Hashtag('sport')

[IN]: hashtag2 = Hashtag('travel1) [IN]: print(hashtagl + hashtag2)

[OUT]: »sport »travel

Then create three Hashtag instances for the following text documents:

- python

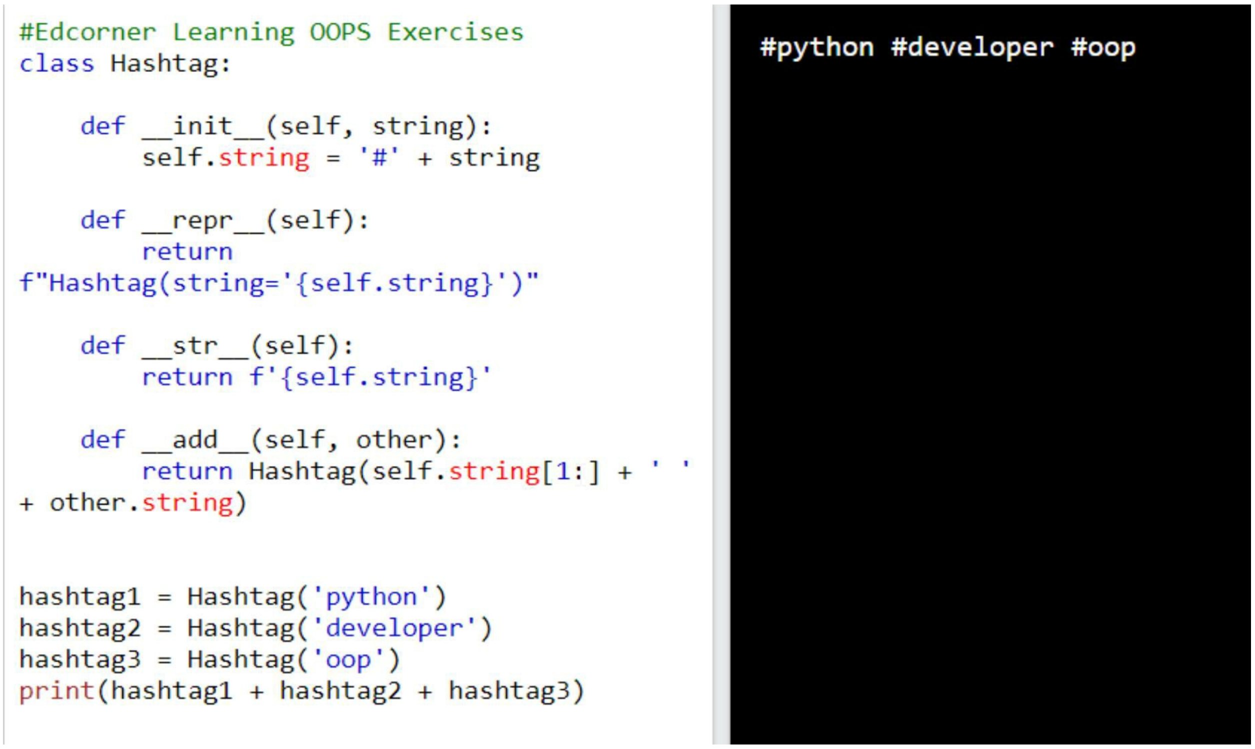
- developer
- oop

In response, print the result of adding these instances.

Expected result:

#python #developer #oop

```
class Hashtag:

    def __init__(self, string):
        self.string = '#' + string



def __repr__(self):
    return f"Hashtag(string='{self.string}')"

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



```
#Edcorner Learning OOPS Exercises
class Hashtag:

    def __init__(self, string):
        self.string = '#' + string

    def __repr__(self):
        return
f"Hashtag(string='{self.string}')"

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

    def __add__(self, other):
        return Hashtag(self.string[1:] + ' '
+ other.string)


hashtag1 = Hashtag('python')
hashtag2 = Hashtag('developer')
hashtag3 = Hashtag('oop')
print(hashtag1 + hashtag2 + hashtag3)
```

```
#python #developer #oop
```

Solution:

25. The following Doc class is implemented for storing text documents. Implement the _eq_( )special method to compare Doc instances. Class instances are equal when they have identical string attribute values.

Example:

[IN]: docl = Doc('Finance')

[IN]: doc2 = Doc('Finance')

[IN]: print(docl == doc2)

[OUT]: True

Then create two instances of the Doc class for the following documents:

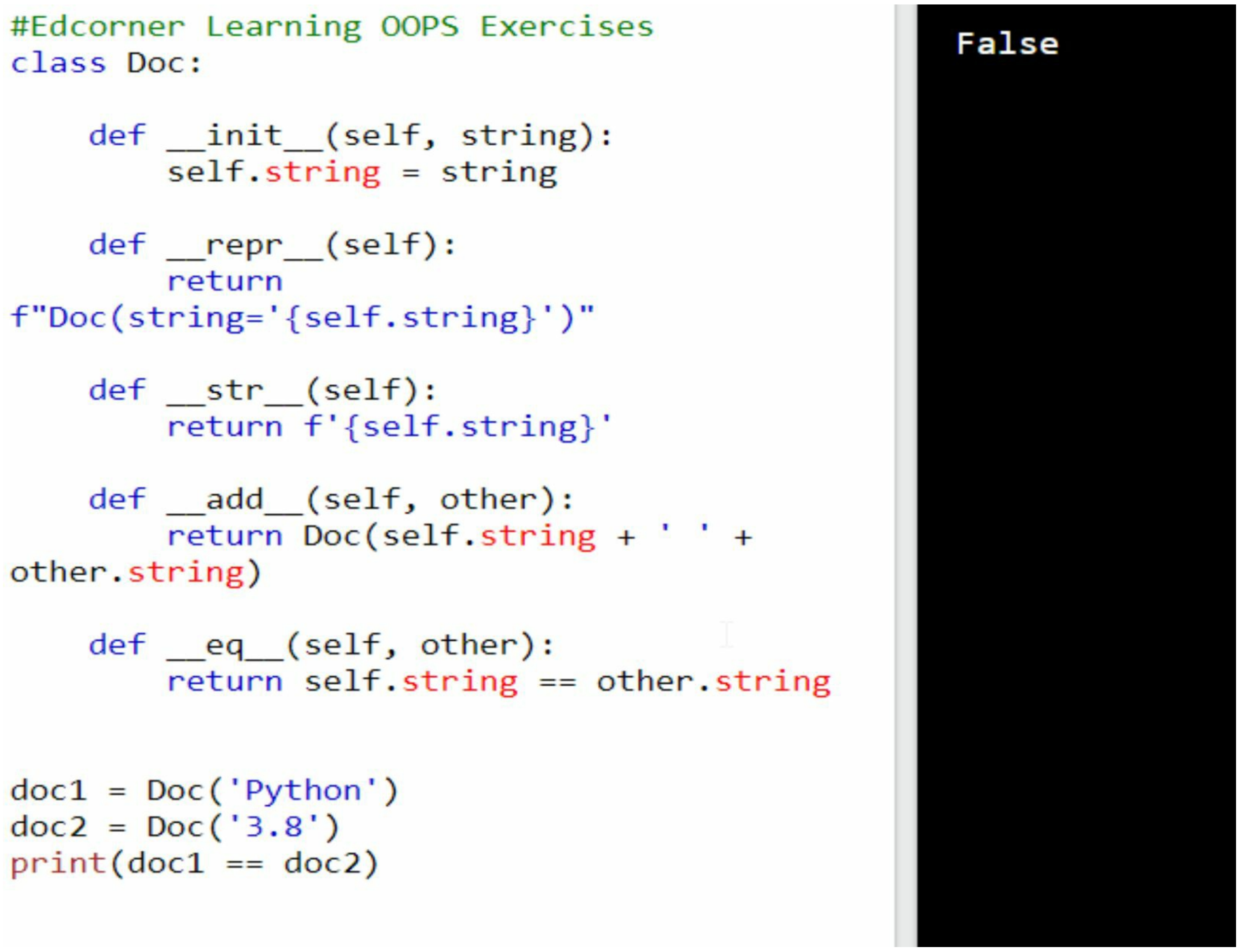
- 'Python'
- '3.8'

In response, print the result of comparing these instances.

Expected result:

False

```
class Doc:

     def __init__(self, string):
          self.string = string

     def __repr__(self):
          return f"Doc(string='{self.string}')"

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

def __add__(self, other):
     return Doc(self.string + ' ' + other.string)
```



```
#Edcorner Learning OOPS Exercises
class Doc:

    def __init__(self, string):
        self.string = string

    def __repr__(self):
        return
f"Doc(string='{self.string}')"

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

    def __add__(self, other):
        return Doc(self.string + ' ' +
other.string)

    def __eq__(self, other):
        return self.string == other.string


doc1 = Doc('Python')
doc2 = Doc('3.8')
print(doc1 == doc2)
```

```
False
```

Solution:

26. The following Doc class is implemented for storing text documents. Implement the _it_( )

special method to compare Doc instances. A class instance is 'smaller1 than another instance when the string attribute is shorter.

Example:

[IN]: docl = Doc('Finance')

[IN]: doc2 = Doc('Education')

```
[IN]: print(docl < doc2)
   [OUT]: True
```

Then create two instances of the Doc class for the following documents:

- 'sport'
- 'activity'

and assign to the variables:

doc1

doc2

In response, print the result of comparing these instances (perform doc1 < doc2).

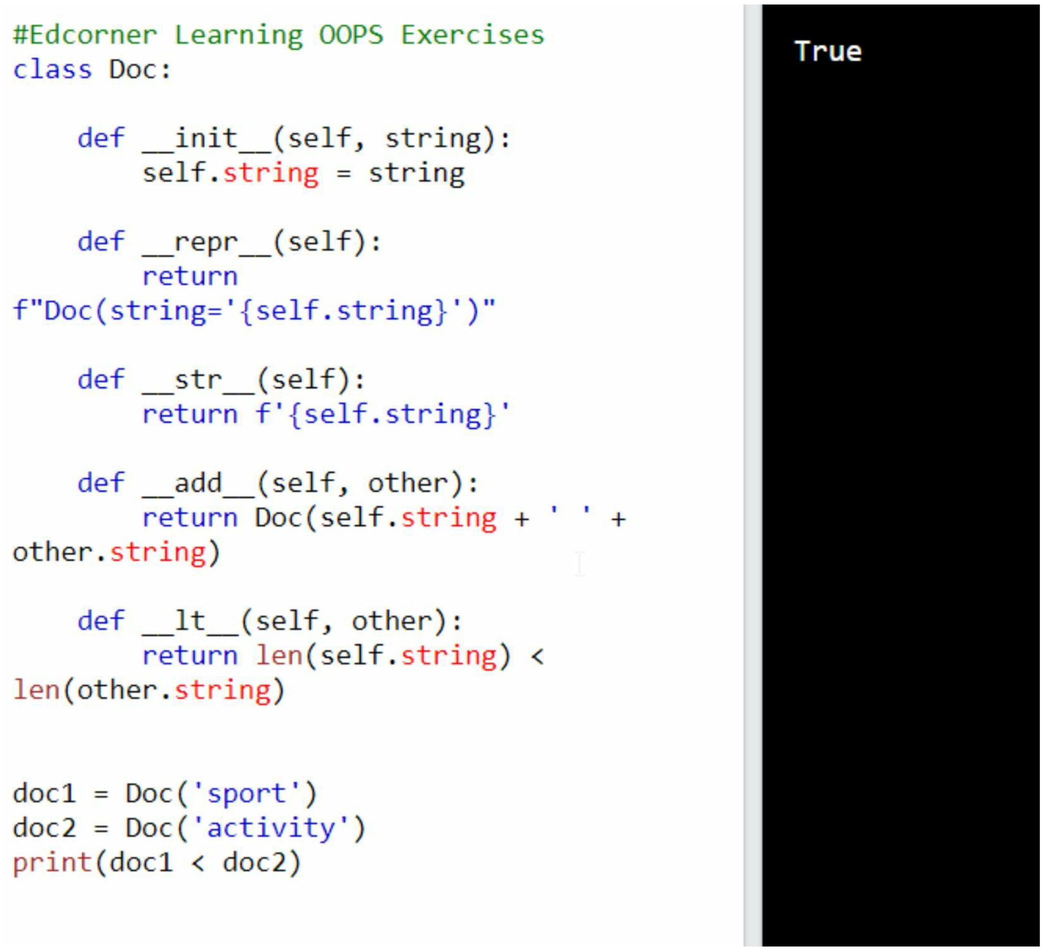
Expected result:

True

```
class Doc:


    def __init__(self, string):
        self.string = string


    def __repr__(self):
        return f"Doc(string='{self.string}')"
```

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

def __add__(self, other):
    return Doc(self.string + ' ' + other.string)
```

Solution:



```
#Edcorner Learning OOPS Exercises
class Doc:

    def __init__(self, string):
        self.string = string

    def __repr__(self):
        return
f"Doc(string='{self.string}')"

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

    def __add__(self, other):
        return Doc(self.string + ' ' +
other.string)

    def __lt__(self, other):
        return len(self.string) <
len(other.string)


doc1 = Doc('sport')
doc2 = Doc('activity')
print(doc1 < doc2)
```

```
True
```

27. The following Doc class is implemented for storing text documents. Implement the _iadd_() special method to perform extended assignments. Concatenate two instances with the string ‘ & ‘

Example:

[IN]:

[IN]:

[IN]:

docl = Doc(1 Finance')

doc2 = Doc('Accounting1)

```
docl += doc2
[IN]: print(docl)
[OUT]: Finance & Accounting
```

Then create two instances of the Doc class for the following documents:

- 'sport'
- 'activity'

and assign according to the variables:

- docl

doc2

Perform extended assignment

- docl + = doc2

Print docl instance to the console.

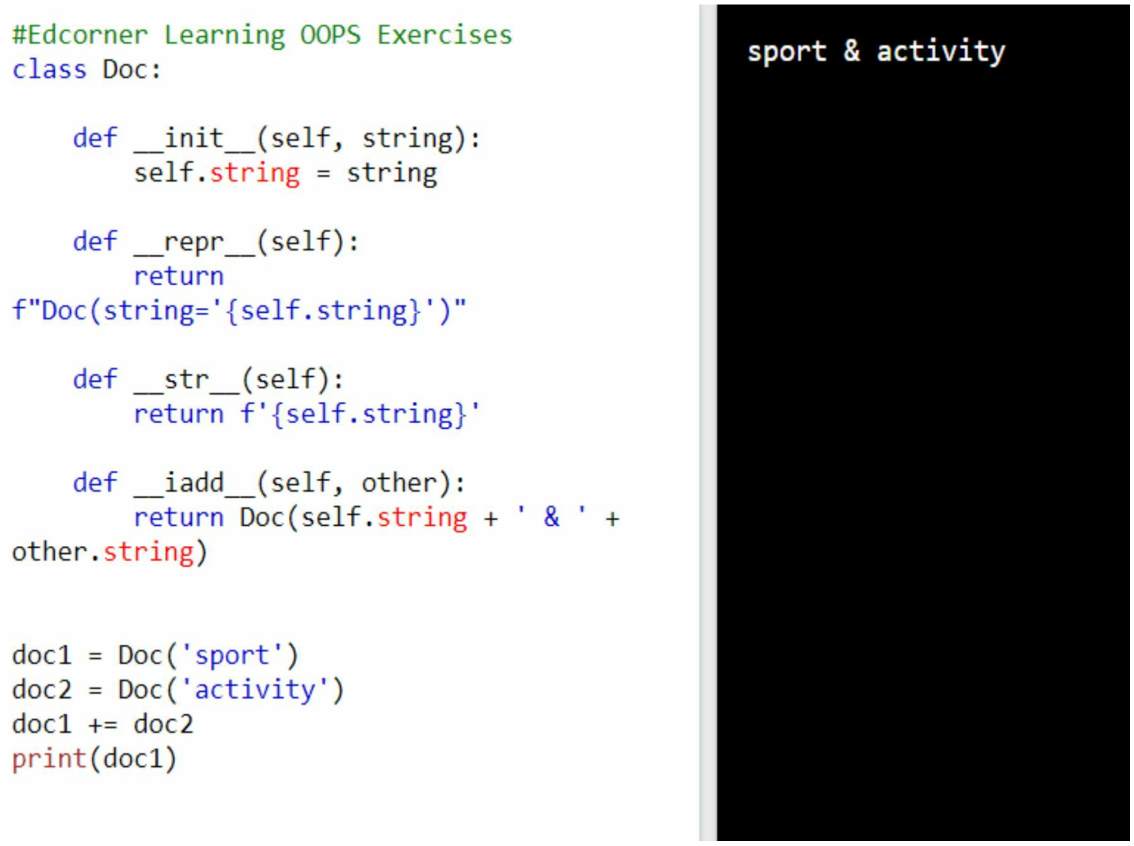
Expected result:

sport & activity

```
class Doc:

    def __init__(self, string):
        self.string = string

    def __repr__(self):
        return f"Doc(string='{self.string}')"
```

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

Solution:



```
#Edcorner Learning OOPS Exercises
class Doc:

    def __init__(self, string):
        self.string = string

    def __repr__(self):
        return
f"Doc(string='{self.string}')"

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

    def __iadd__(self, other):
        return Doc(self.string + ' & ' +
other.string)


doc1 = Doc('sport')
doc2 = Doc('activity')
doc1 += doc2
print(doc1)
```

```
sport & activity
```

28. The Book class is given. Implement the _str _() method to display an informal

representation of a Book instance (see below).

Example:

[IN]: bookl = Book('Python OOPS Vol2', 'Edcorner Learning’)

[IN]: print(bookl)

[OUT]: Book ID: 214522 | Title: Python OOPS Vol2 | Author: Edcorner Learning

Then create an instance named book with arguments:

- title= ‘Python OOPS Vol2’
- author='Edcorner Learning’

In response, print the instance to the console.

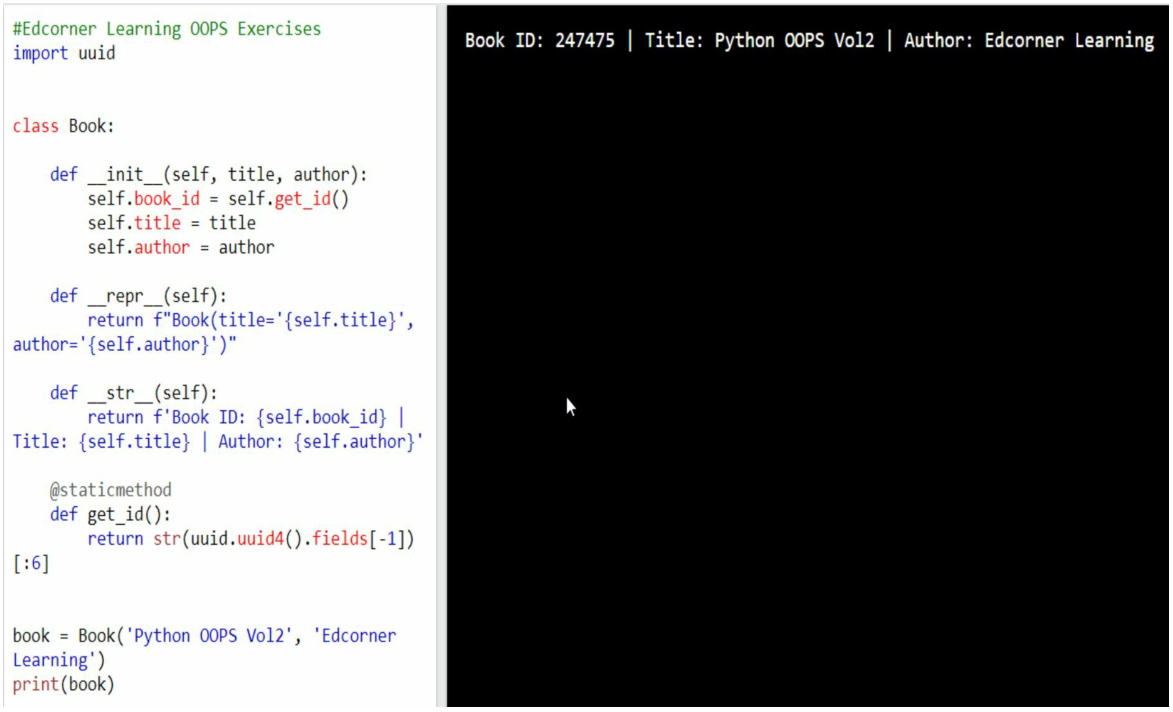
Expected result:

Book ID: 1234 | Title: Python OOPS Vol2 | Author: Edcorner Learning

Note: The Book ID value may vary.

```
import uuid


class Book:


    def __init__(self, title, author):
            self.book_id = self.get_id()
            self.title = title
        self.author = author

    def __repr__(self):
        return f"Book(title='{self.title}', author='{self.author}')"

    @staticmethod
    def get_id():
        return str(uuid.uuid4().fields[-1])[:6]
```

Solution:



```python
#Edcorner Learning OOPS Exercises
import uuid


class Book:

    def __init__(self, title, author):
        self.book_id = self.get_id()
        self.title = title
        self.author = author

    def __repr__(self):
        return f"Book(title='{self.title}',
author='{self.author}')"

    def __str__(self):
        return f'Book ID: {self.book_id} |
Title: {self.title} | Author: {self.author}'

    @staticmethod
    def get_id():
        return str(uuid.uuid4().fields[-1])
[:6]


book = Book('Python OOPS Vol2', 'Edcorner
Learning')
print(book)
```

```
Book ID: 247475 | Title: Python OOPS Vol2 | Author: Edcorner Learning
```

Module 4 Inheritance

29. The Container class was implemented. Implement two simple classes inheriting from the class Container with names respectively:

- PlasticContainer
- MetalContainer

```
class Container:
    pass
```

Solution:

```
class Container:
    pass
class PlasticContainer(Container):
    pass
class MetalContainer(Container):
    pass
```

30. The following classes are implemented:

- Container

- PlasticContainer
- MetalContainer
- CustomContainer

Using the issubciassQ built-in function, check if the classes:

PlasticContainer

MetalContainer

• CustomContainer

are subclasses of Container class. Print the result to the console as shown below:

```
True
True
False
```

```
class Container:
    pass
class PlasticContainer(Container):
    pass
class MetalContainer(Container):
    pass



class CustomContainer:
    pass
```

Solution:

class Container:

```
    pass
class PlasticContainer(Container):
    pass
class MetalContainer(Container):
    pass
class CustomContainer:
    pass
print(issubclass(PlasticContainer, Container))
print(issubclass(MetalContainer, Container))
print(issubclass(CustomContainer, Container))
```

31. The following classes are implemented:

- Vehicle
- LandVehicle
- AirVehicle

Define a _repr _( ) special method in the Vehicle class that returns a formal representation of

the objects of the classes Vehicle, LandVehicle, and AirVehicle.

Example: The code below:

```
instances = [Vehicle(), LandVehicle(), AirVehicle()]
```

for instance in instances: print(instance)

returns:

Vehicle(category='land vehicle')

LandVehicle(category='land vehicle1)

AirVehicle(category='air vehicle1)

Run the code below in response:

instances = [VehicleQ, LandVehicleQ, AirVehicleQ]

for instance in instances: print(instance)

Expected result:

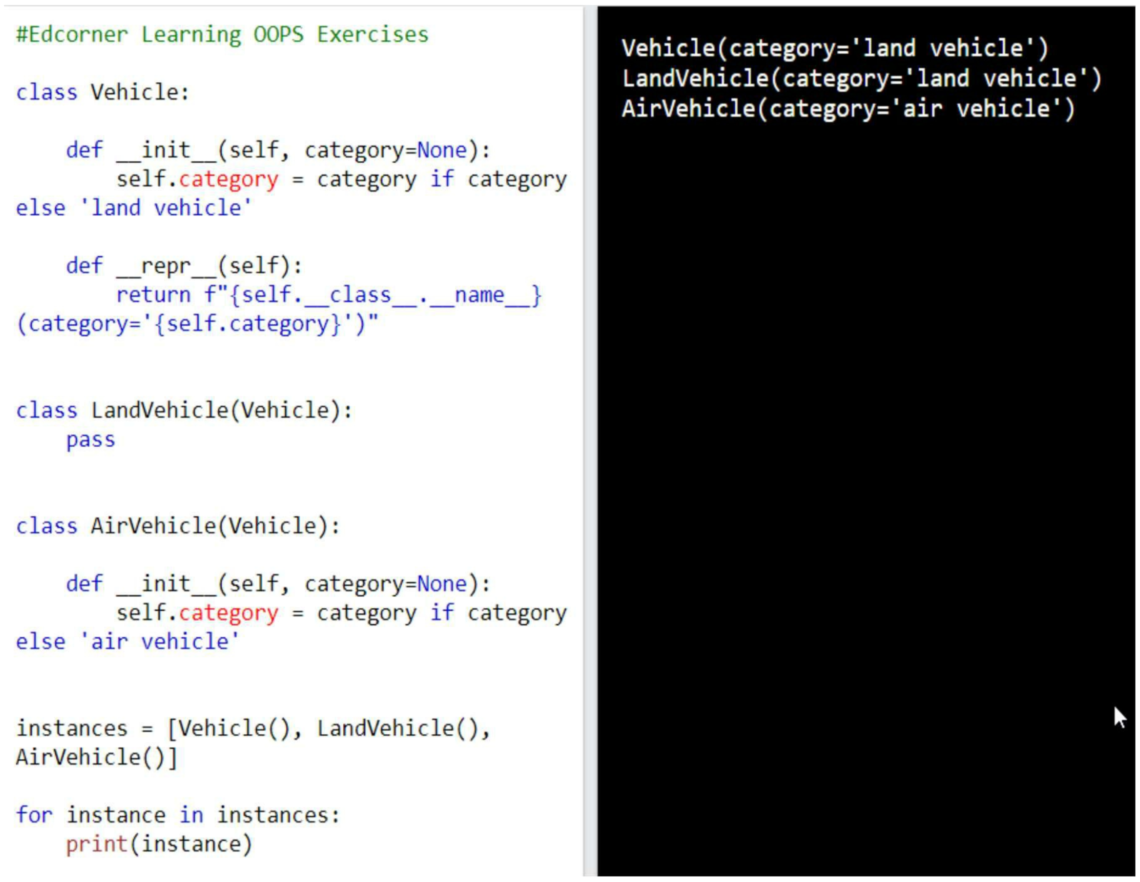
Vehicle(category='land vehicle')

LandVehicle(category='land vehicle1)

AirVehicle(category='air vehicle1)

```
class Vehicle:


    def __init__(self, category=None):
        self.category = category if category else 'land vehicle'
class LandVehicle(Vehicle):
    pass
class AirVehicle(Vehicle):
    def __init__(self, category=None):
        self.category = category if category else 'air vehicle'
```

```
instances = [Vehicle(), LandVehicle(), AirVehicle()]
for instance in instances:
    print(instance)
```



```
#Edcorner Learning OOPS Exercises

class Vehicle:

    def __init__(self, category=None):
        self.category = category if category
else 'land vehicle'

    def __repr__(self):
        return f"{self.__class__.__name__}
(category='{self.category}')"


class LandVehicle(Vehicle):
    pass


class AirVehicle(Vehicle):

    def __init__(self, category=None):
        self.category = category if category
else 'air vehicle'


instances = [Vehicle(), LandVehicle(),
AirVehicle()]

for instance in instances:
    print(instance)
```

```
Vehicle(category='land vehicle')
LandVehicle(category='land vehicle')
AirVehicle(category='air vehicle')
```

32. The following classes are implemented:

- Vehicle
- LandVehicle
- AirVehicle

Define a dispiay_info( ) method in the Vehicle class to display the class name along with the value of the category attribute. The method is supposed to work for all classes.

For example, the following code:

instances = [Vehicle(), LandVehtcle(), AirVehtcle()]

for instance in instances: print(instance)

returns:

Vehicle -> land vehicle

LandVehicle

land vehicle

AirVehicle -> air vehicle

Run the code below in response:

instances = [Vehicle(), LandVehicle(), AirVehicle()]

for instance in instances: print(instance)

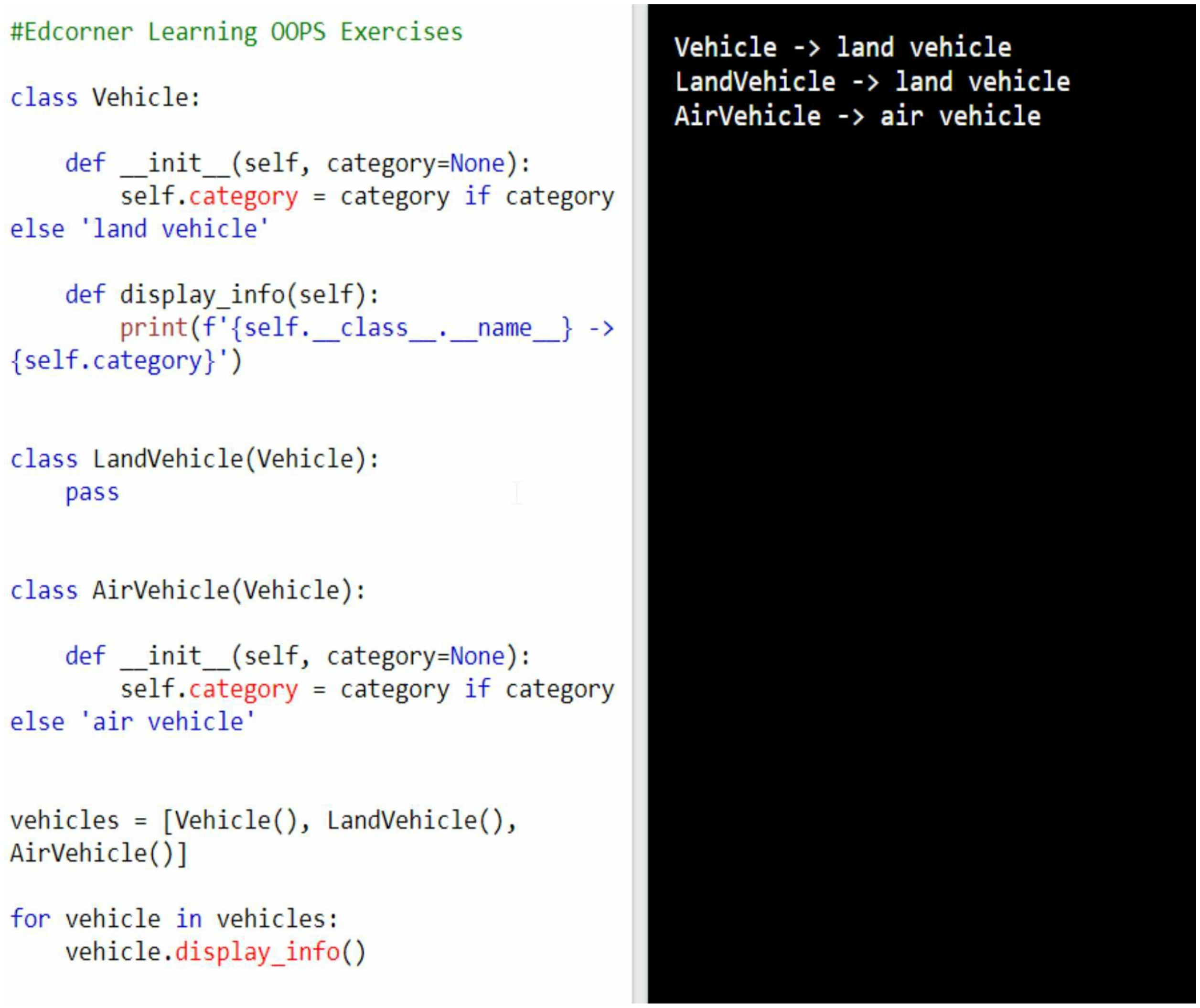
Expected result:

Vehicle -> land vehicle

LandVehlcle -> land vehicle

AlrVehlcle -> air vehicle

```
 class Vehicle:
    def __init__(self, category=None):
        self.category = category if category else 'land vehicle'

class LandVehicle(Vehicle):
    pass
class AirVehicle(Vehicle):

    def __init__(self, category=None):
        self.category = category if category else 'air vehicle'
vehicles = [Vehicle(), LandVehicle(), AirVehicle()]

for vehicle in vehicles:
    vehicle.display_info()
```



```python
#Edcorner Learning OOPS Exercises

class Vehicle:

    def __init__(self, category=None):
        self.category = category if category
else 'land vehicle'

    def display_info(self):
        print(f'{self.__class__.__name__} ->
{self.category}')


class LandVehicle(Vehicle):
    pass


class AirVehicle(Vehicle):

    def __init__(self, category=None):
        self.category = category if category
else 'air vehicle'


vehicles = [Vehicle(), LandVehicle(),
AirVehicle()]

for vehicle in vehicles:
    vehicle.display_info()
```

```
Vehicle -> land vehicle
LandVehicle -> land vehicle
AirVehicle -> air vehicle
```

Solution:

33. A Vehicle class is given that has three instance attributes:

- brand
- color
- year

Create a Car class that inherits from Vehicle class. Next, override the _init ( ) method so

that the Car class in the constructor takes four arguments:

- brand
- color
- year
- horsepower

and set them appropriately as instance attributes. Don't use super () in this case. Then create following instances:

- with the name vehicle and the attribute values: 'BMW', 'red', 2020
- with the name car and the attribute values: 'BMW', 'red', 2020, 300

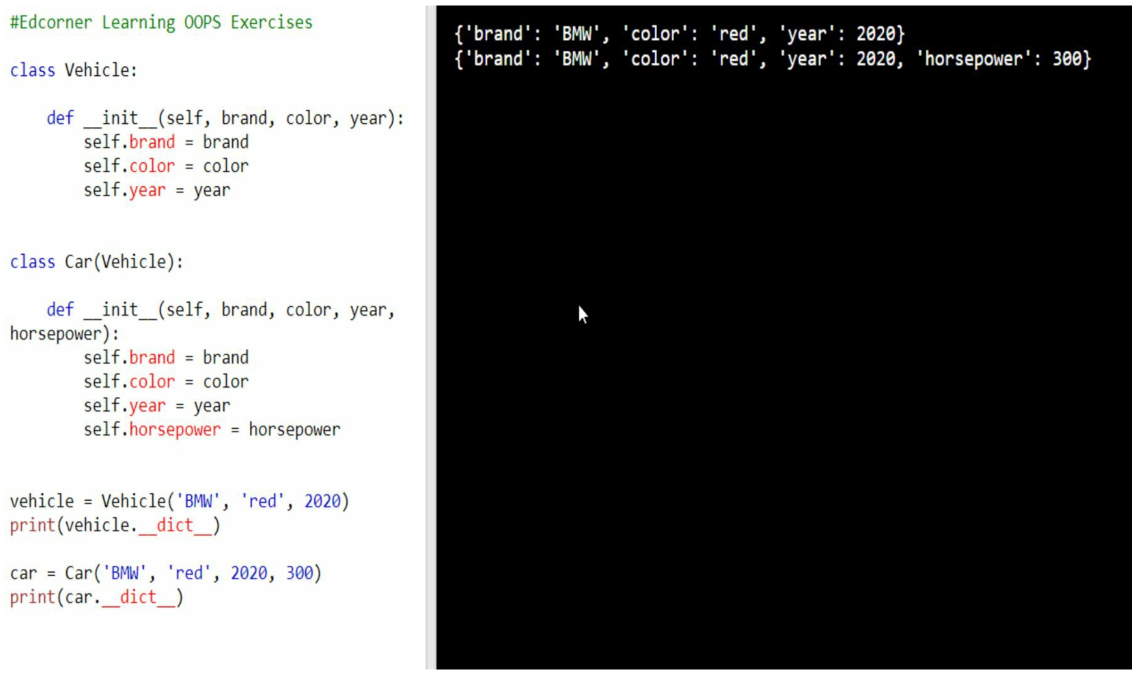
In response, print the value of the _dict_ attribute of the vehicle and car instances.

Expected result:

{'brand': ' BMW’, 'color': 'red', 'year': : 2020}

{1 brand' : ' BMW’, 'color': 'red', 'year': : 2020, 'horsepower': 300}

```
class Vehicle:

    def __init__(self, brand, color, year):
        self.brand = brand
        self.color = color
self.year = year
```



```
#Edcorner Learning OOPS Exercises

class Vehicle:

    def __init__(self, brand, color, year):
        self.brand = brand
        self.color = color
        self.year = year


class Car(Vehicle):

    def __init__(self, brand, color, year,
horsepower):
        self.brand = brand
        self.color = color
        self.year = year
        self.horsepower = horsepower


vehicle = Vehicle('BMW', 'red', 2020)
print(vehicle.__dict__)

car = Car('BMW', 'red', 2020, 300)
print(car.__dict__)
```

```
{'brand': 'BMW', 'color': 'red', 'year': 2020}
{'brand': 'BMW', 'color': 'red', 'year': 2020, 'horsepower': 300}
```

Solution:

34. The Vehicle and Car classes are listed below. Implement a method named dispiay_attrs() in the base class Vehicle, which displays the instance attributes and their values. For example, for the Vehicle class:

vehicle = Vehlcle('BMW', 'red', 2020) vehicle.dlsplay_attrs()

brand -> BMW

color -> red

year -> 2020

And for the Car class:

car = Car(‘BMW’, 'red', 2020, 190) car.dlsplay_attrs()

brand -> BMW color -> red

year -> 2020

horsepower -> 190

Then create an instance of the Car class named car with the attribute values: 1 Opel ', 'black', 2018, 160

In response, call dispiay_attrs( ) on the car instance.

Expected result:

brand -> Opel

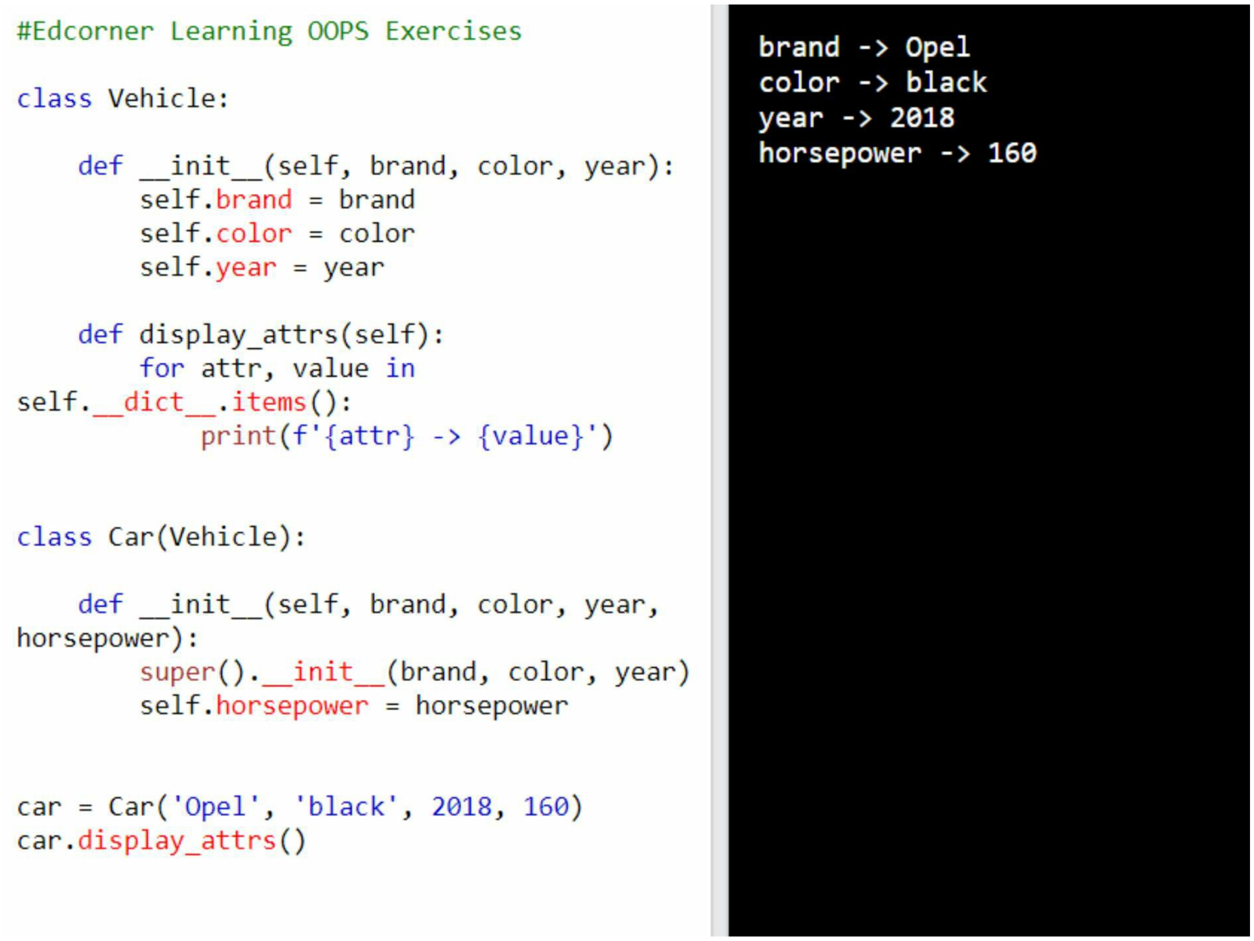
color -> black

year -> 2018

horsepower -> 160

```
class Vehicle:

    def __init__(self, brand, color, year):
        self.brand = brand
        self.color = color
        self.year = year


class Car(Vehicle):
```

```
def __init__(self, brand, color, year, horsepower):
    super().__init__(brand, color, year)
    self.horsepower = horsepower
```



```python
#Edcorner Learning OOPS Exercises

class Vehicle:

    def __init__(self, brand, color, year):
        self.brand = brand
        self.color = color
        self.year = year

    def display_attrs(self):
        for attr, value in
self.__dict__.items():
            print(f'{attr} -> {value}')


class Car(Vehicle):

    def __init__(self, brand, color, year,
horsepower):
        super().__init__(brand, color, year)
        self.horsepower = horsepower


car = Car('Opel', 'black', 2018, 160)
car.display_attrs()
```

```
brand -> Opel
color -> black
year -> 2018
horsepower -> 160
```

Solution:

35. The Vehicle and Car classes are listed below. Extend the dispiay_attrs() method in the Car class so that the following information is displayed before displaying the attributes: ' calling from class: Car' and then the rest of the attributes with their values. Use super () for this. For example, for the Car class:

car = Car('BMW', 'red', 2020, 190) car.dlsplay_attrs()

returns:

Calling from class: Car brand -> BMW

color -> red

year -> 2020

horsepower ->

190

Then create an instance of the class Car named car with the attribute values: ' BMW’

'black', 2018, 260

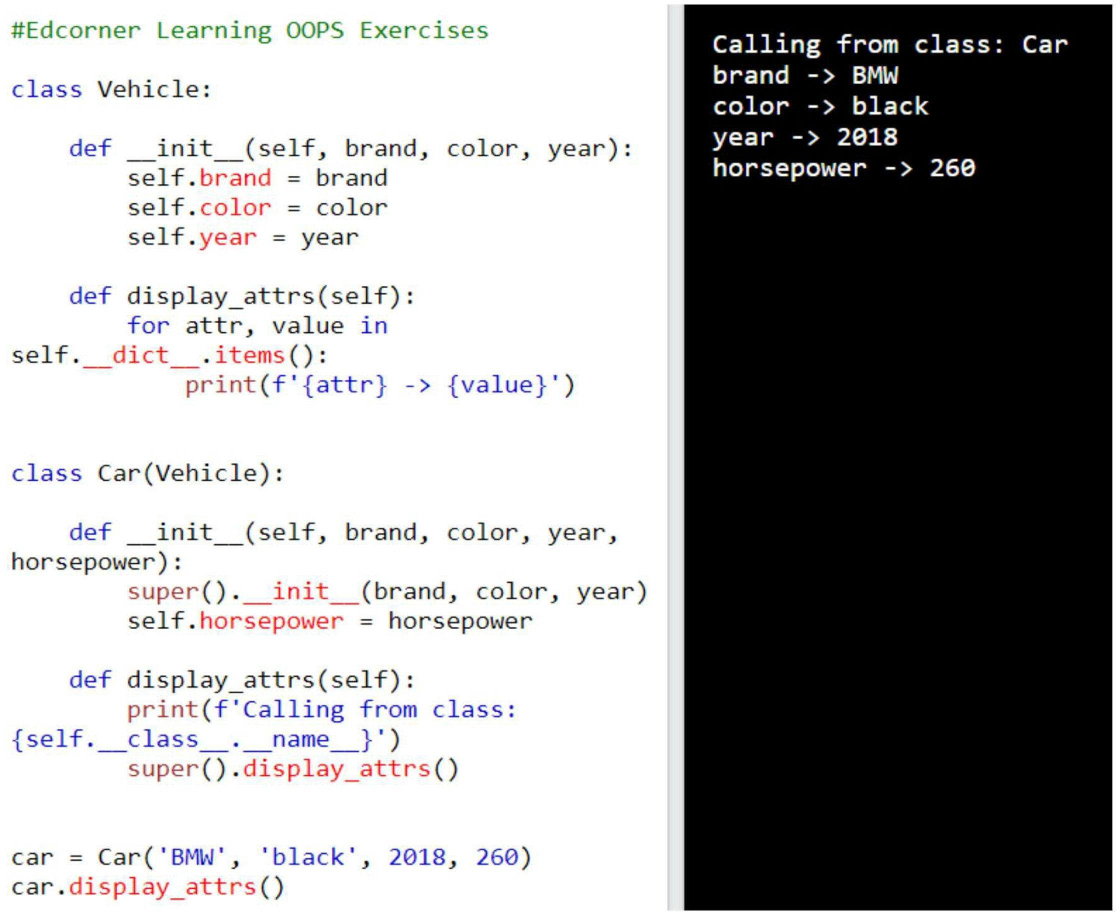
In response, call display_attrs( ) on the car instance.

Expected result:

```
Calling from class: Car
brand -> BMW
color -> black
year -> 2018
horsepower -> 260
```

```
class Vehicle:

    def __init__(self, brand, color, year):
        self.brand = brand
        self.color = color
        self.year = year

    def display_attrs(self):
        for attr, value in self.__dict__.items():
            print(f'{attr} -> {value}')


class Car(Vehicle):

    def __init__(self, brand, color, year, horsepower):
        super().__init__(brand, color, year)
        self.horsepower = horsepower
```

Solution:



```
#Edcorner Learning OOPS Exercises

class Vehicle:

    def __init__(self, brand, color, year):
        self.brand = brand
        self.color = color
        self.year = year

    def display_attrs(self):
        for attr, value in
self.__dict__.items():
            print(f'{attr} -> {value}')


class Car(Vehicle):

    def __init__(self, brand, color, year,
horsepower):
        super().__init__(brand, color, year)
        self.horsepower = horsepower

    def display_attrs(self):
        print(f'Calling from class:
{self.__class__.__name__}')
        super().display_attrs()


car = Car('BMW', 'black', 2018, 260)
car.display_attrs()
```

```
Calling from class: Car
brand -> BMW
color -> black
year -> 2018
horsepower -> 260
```

36. Implement simple classes with the following structure:

• Container

• TemperatureControlledContainer • RefrigeratedContainer

The TemperatureControlledContainer class inherits from the Container class and the RefrigeratedContainer class inherits from TemperatureControlledContainer.

Solution:

```
class Container:
    pass

class TemperatureControlledContainer(Container):
    pass

class RefrigeratedContainer(TemperatureControlledContainer):
    pass
```

37. Simple classes with the following structure are implemented:

• Container

• TemperatureControlledContainer • RefrigeratedContainer Using the built-in issubclass() function, check if:

• TemperatureControlledContainer is a class derived from Container

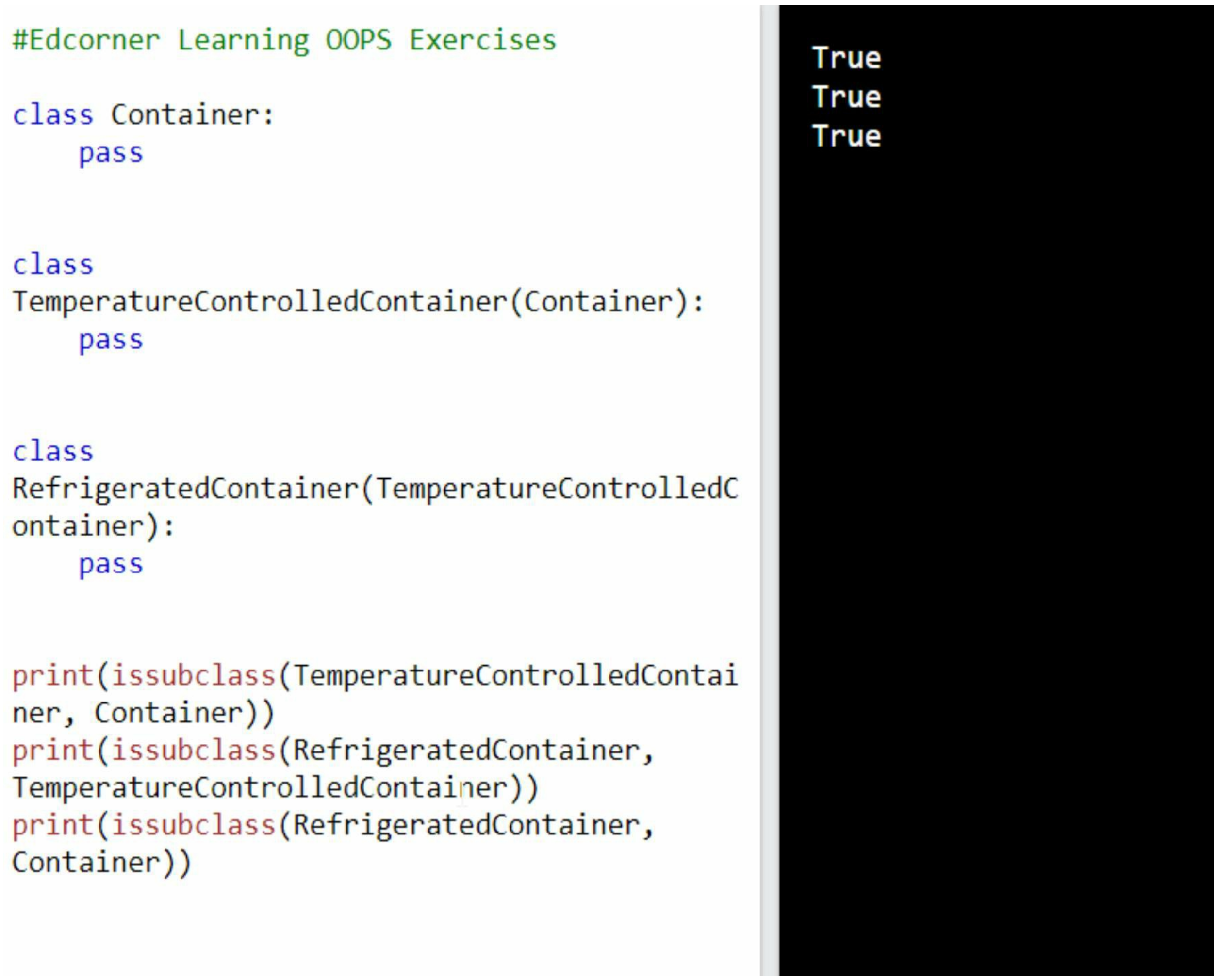
• RefrigeratedContainer is a class derived from TemperatureControlledContainer

• RefrigeratedContainer is a class derived from Container and print the obtained logical values to the console.

Expected result:

```
True
True
True
```

```
 class Container:
    pass
class TemperatureControlledContainer(Container):
    pass
class RefrigeratedContainer(TemperatureControlledContainer):
    pass
```



```
#Edcorner Learning OOPS Exercises

class Container:
    pass


class
TemperatureControlledContainer(Container):
    pass


class
RefrigeratedContainer(TemperatureControlledC
ontainer):
    pass


print(issubclass(TemperatureControlledContai
ner, Container))
print(issubclass(RefrigeratedContainer,
TemperatureControlledContainer))
print(issubclass(RefrigeratedContainer,
Container))
```

```
True
True
True
```

Solution:

38. Simple classes with the following structure are implemented:

• Container

• TemperatureControlledContainer • RefrigeratedContainer

The TemperatureControlledContainer class inherits from the Container class and the RefrigeratedContainer class inherits from

TemperatureControlledContainer.

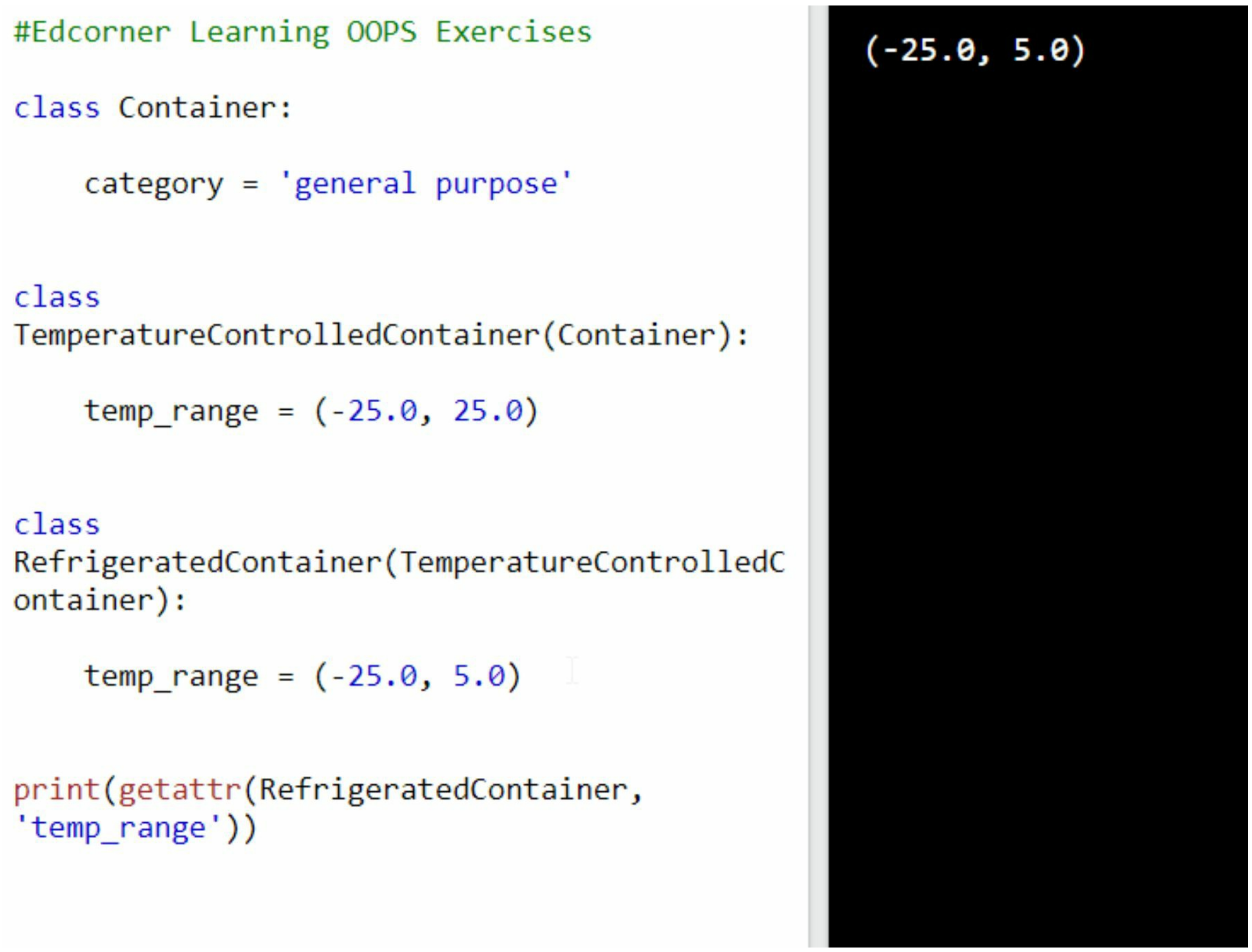
Add a class attribute called temp_range to the TemperatureControlledContainer class that stores the tuple (-25.0, 25.0) , and to the RefrigeratedContainer class add a class attribute with the same name and value (-25.0, 5.0) .

Then, using the getattr( ) function, read the value of the tempjange attribute of the RefrigeratedContainer class and print to the console.

Expected result:

(-25.0, 5.0)

```
class Container:
    category = 'general purpose'
class TemperatureControlledContainer(Container):
    pass
class RefrigeratedContainer(TemperatureControlledContainer):
    pass
```



```
#Edcorner Learning OOPS Exercises

class Container:

    category = 'general purpose'


class
TemperatureControlledContainer(Container):

    temp_range = (-25.0, 25.0)


class
RefrigeratedContainer(TemperatureControlledC
ontainer):

    temp_range = (-25.0, 5.0)


print(getattr(RefrigeratedContainer,
'temp_range'))
```

```
(-25.0, 5.0)
```

Solution:

39. Implement two simple classes named Person and Department. Then create a Worker class that inherits from the Person and Department classes in the given order (multiple inheritance).

Solution:

class Person:

```
    pass


class Department:
    pass


class Worker(Person, Department):
    pass
```

40. The following classes are defined. Add the _init_() method to the Person class, which sets three attributes:

- firstname
- lastname
- age

Then create an instance of the Worker class passing the following arguments:

- 'John'
- 'Doe'
- 35

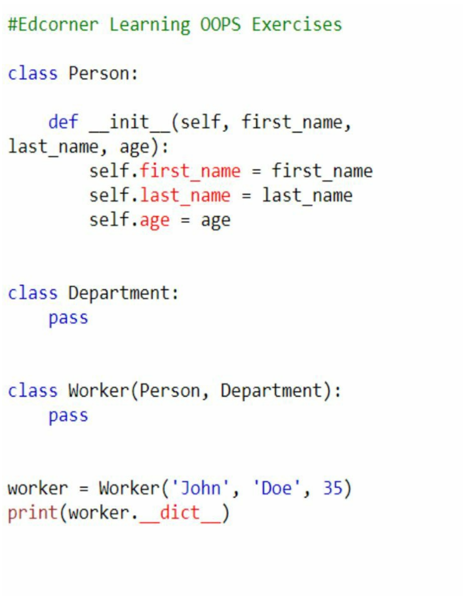
In response, print the value of the _dict_ attribute of this instance.

Expected result:

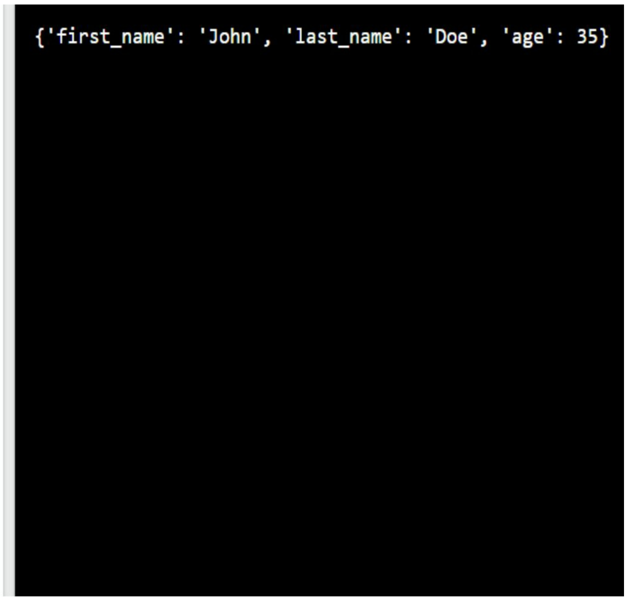
{‘first_name’ : 'John', 'last_name': 'Doe', 'age': 35}

```
class Person:
    pass
class Department:
    pass
class Worker(Person, Department):
    pass
```

Solution:



```
#Edcorner Learning OOPS Exercises

class Person:

    def __init__(self, first_name,
last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age


class Department:
    pass


class Worker(Person, Department):
    pass


worker = Worker('John', 'Doe', 35)
print(worker.__dict__)
```



```
{'first_name': 'John', 'last_name': 'Doe', 'age': 35}
```

41. The following classes are defined. Add a init ( ) method to the Department class that sets

the following attributes:

- deptname (department name)
- short_dept_name (department short name)

Then create an instance of the Department class with arguments:

- 'Information Technology'
- 'IT'

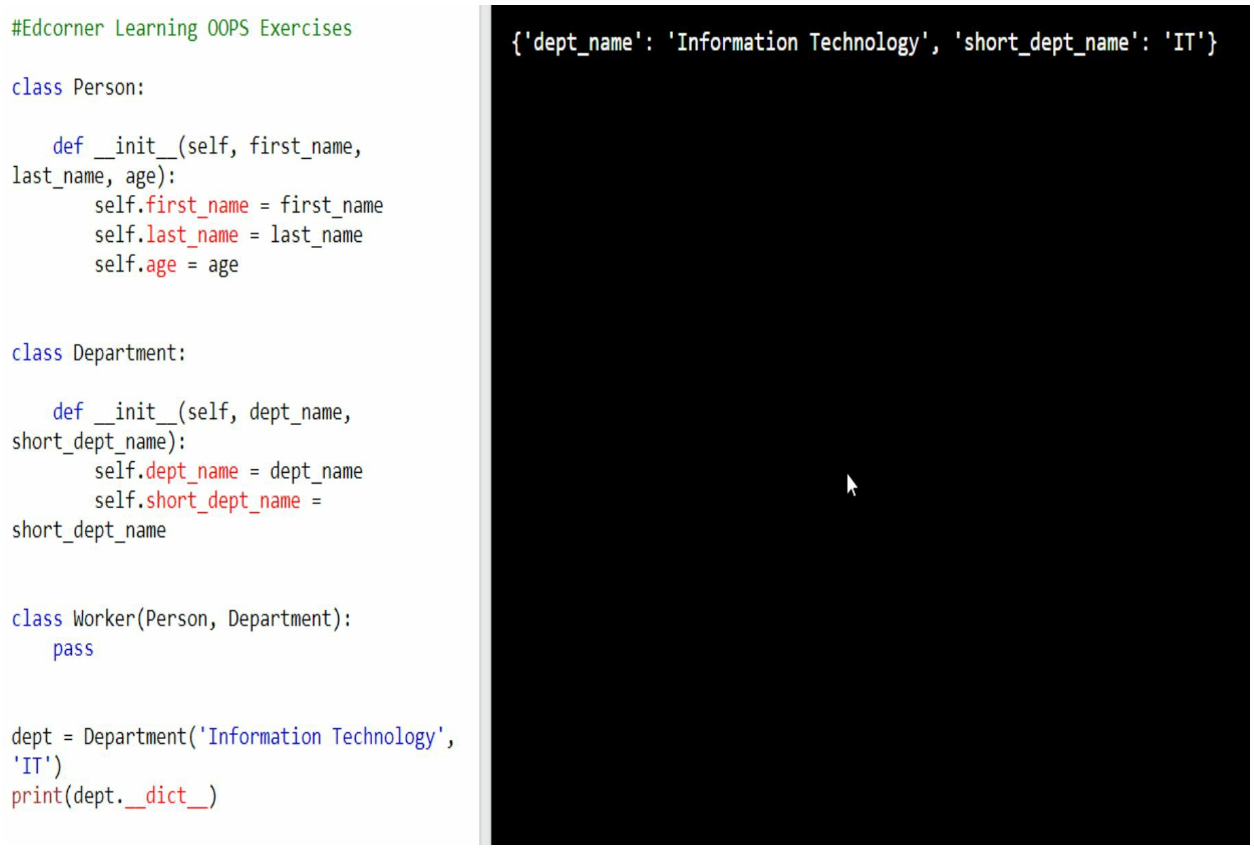
In response, print the value of the _dict_ attribute of this instance.

Expected result:

{'dept_name': 'Information Technology', 'short_dept_name': 'IT'}

```
class Person:


    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
            class Department:
            pass
```

```
class Worker(Person, Department):
    pass
```



```
#Edcorner Learning OOPS Exercises

class Person:

    def __init__(self, first_name,
last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age


class Department:

    def __init__(self, dept_name,
short_dept_name):
        self.dept_name = dept_name
        self.short_dept_name =
short_dept_name


class Worker(Person, Department):
    pass


dept = Department('Information Technology',
'IT')
print(dept.__dict__)
```

```
{'dept_name': 'Information Technology', 'short_dept_name': 'IT'}
```

Solution:

42. The following classes are defined. Add the _init_( ) method to the Worker

class to set all the attributes from the Person and Department classes.

Then create an instance of the Worker class passing the following arguments:

- 'John'
- 'Doe'
- 30
- 'Information Technology’
- 'IT'

In response, print the value of the _dict_ attribute of this instance.

.

Expected Result:

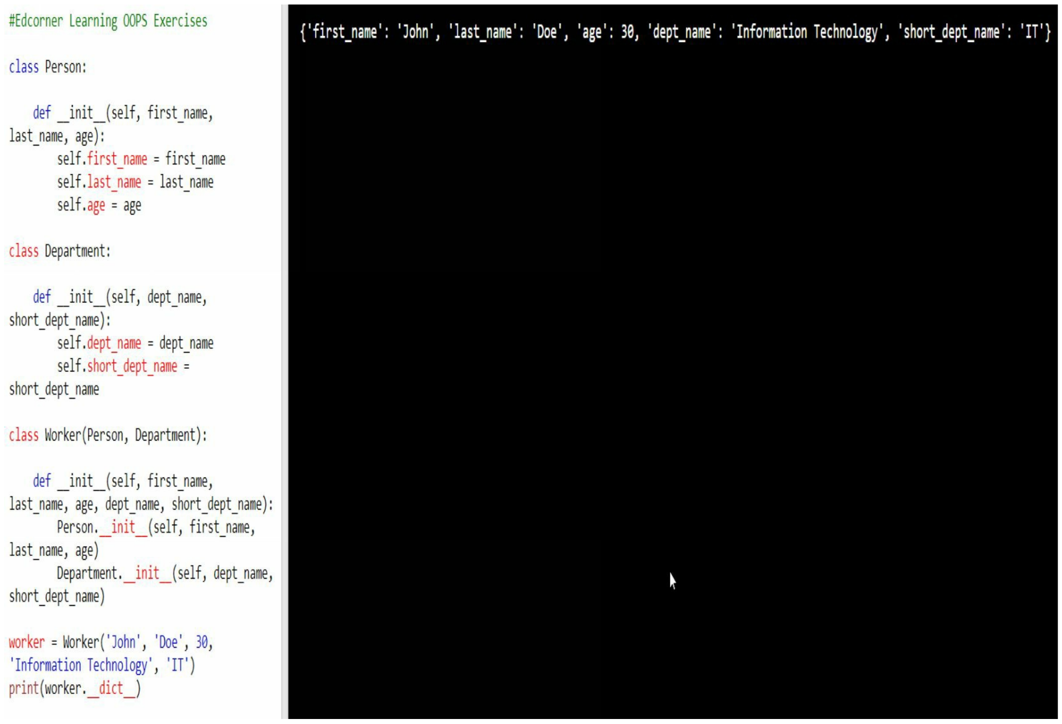
{'first_name': 'John', 'last_name': 'Doe', 'age': 30, 'dept_name': 'Information Technology', 'short_dept_name': 'IT'}

```
class Person:


  def __init__(self, first_name, last_name, age):
    self.first_name = first_name
    self.last_name = last_name
    self.age = age
class Department:
  def __init__(self, dept_name, short_dept_name):
    self.dept_name = dept_name


     self.short_dept_name = short_dept_name
     class Worker(Person, Department):
       pass
```

Solution:

```
class Person:


    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
class Department:
    def __init__(self, dept_name, short_dept_name):
        self.dept_name = dept_name
        self.short_dept_name = short_dept_name
class Worker(Person, Department):
    def __init__(self, first_name, last_name, age, dept_name,
short_dept_name):
        Person.__init__(self, first_name, last_name, age)
        Department.__init__(self, dept_name, short_dept_name)




worker = Worker('John', 'Doe', 30, 'Information Technology', 'IT')
print(worker.__dict__)
```

Solution:



```
#Edcorner Learning OOPS Exercises

class Person:

    def __init__(self, first_name,
last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

class Department:

    def __init__(self, dept_name,
short_dept_name):
        self.dept_name = dept_name
        self.short_dept_name =
short_dept_name

class Worker(Person, Department):

    def __init__(self, first_name,
last_name, age, dept_name, short_dept_name):
        Person.__init__(self, first_name,
last_name, age)
        Department.__init__(self, dept_name,
short_dept_name)

worker = Worker('John', 'Doe', 30,
'Information Technology', 'IT')
print(worker.__dict__)
```

```
{'first_name': 'John', 'last_name': 'Doe', 'age': 30, 'dept_name': 'Information Technology', 'short_dept_name': 'IT'}
```

43. The following classes are defined. Display the MRO - Method Resolution Order for the Worker class.

Note: The solution that the user passes is in a file named exercise.py, while the checking code (which is invisible to the user) is executed from a file named evaluate.py from the level where the classes are imported. Therefore, instead of the name of the module _main_ , the response will be the name of the module in which this class is implemented, in this case exercise .

Expected result:

[<class 'exercise.Worker’>, <class 'exercise.Person'>, <class

'exercise.Departnent', <class 'object'>]

```
class Person:

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
class Department:

    def __init__(self, dept_name, short_dept_name):
        self.dept_name = dept_name
        self.short_dept_name = short_dept_name

class Worker(Person, Department):


    def __init__(self, first_name, last_name, age, dept_name):
        Person.__init__(self, first_name, last_name, age)
        Department.__init__(self, dept_name)
```

Solution:

```
class Person:
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
class Department:
    def __init__(self, dept_name, short_dept_name):
        self.dept_name = dept_name
        self.short_dept_name = short_dept_name
class Worker(Person, Department)


   def __init__(self, first_name, last_name, age, dept_name):
        Person.__init__(self, first_name, last_name, age)
        Department.__init__(self, dept_name)
print(Worker.mro())
```

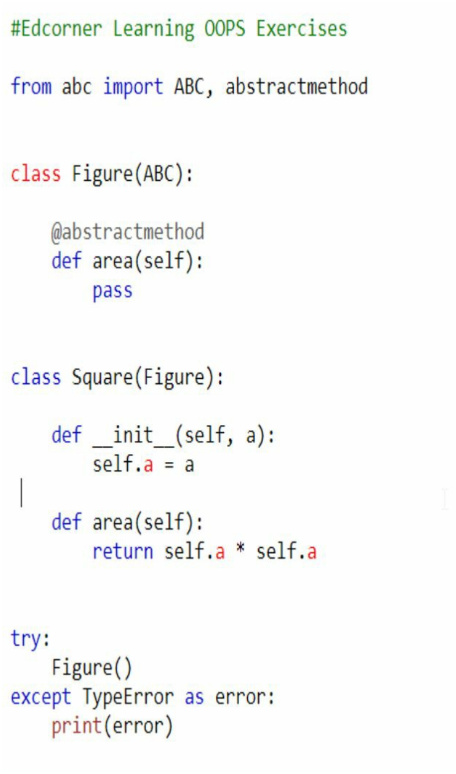
Module 5 Abstract Classes

44. Create an abstract class named Figure with the abstract method named area. Then create a Square class that inherits from the Figure class, which sets the side length of the square in the constructor. Implement the area method that allows you to calculate the area of a square.

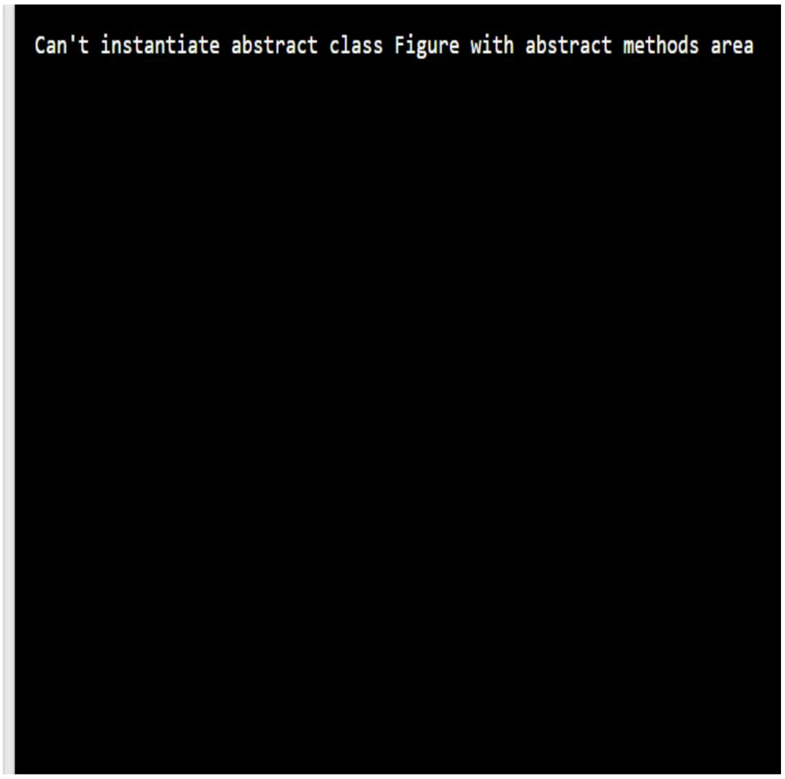
Then try to create an instance of the Figure class, in case of an error, print the error message to the console.

Expected result:

Can't instantiate abstract class Figure with abstract methods area



```
#Edcorner Learning OOPS Exercises

from abc import ABC, abstractmethod


class Figure(ABC):

    @abstractmethod
    def area(self):
        pass


class Square(Figure):

    def __init__(self, a):
        self.a = a

    def area(self):
        return self.a * self.a


try:
    Figure()
except TypeError as error:
    print(error)
```



```
Can't instantiate abstract class Figure with abstract methods area
```

Solution:

45. An implementation of the Figure and Square classes is given. Add

an abstract method called perimeterQ to the Figure class, then implement it in the Square class. The perimeter() method should return the perimeter of the square.

Create an instance of the Square class with side 10 and using the area() and perimeter() methods display the area and perimeter of the created instance to the console.

Expected result:

100

```
from abc import ABC, abstractmethod
class Figure(ABC):
    @abstractmethod
    def area(self):
        pass
class Square(Figure):
    def __init__(self, a):
        self.a = a

    def area(self):
        return self.a * self.a
```

Solution:

from abc import ABC, abstractmethod

```
class Figure(ABC):
    @abstractmethod
    def area(self):
        pass
    @abstractmethod
    def perimeter(self):
        pass
class Square(Figure):

    def __init__(self, a):
        self.a = a

    def area(self):
        return self.a * self.a

    def perimeter(self):
        return 4 * self.a



square = Square(10)
print(square.area())
print(square.perimeter())
```

46. Create an abstract class named Taxpayer. In the _init_() method set an instance attribute

(without validation) called salary. Then add an abstract method called calculate_tax() (use the @abstractmethod decorator).

Solution:

from abc import ABC, abstractmethod

class TaxPayer(ABC):

```
def __init__(self, salary):
    self.salary = salary

@abstractmethod
def calculate_tax(self):
    pass
```

47. An implementation of the Taxpayer abstract class is given. Create a class derived from Taxpayer named StudentTaxPayer that implements the caicuiate_tax( ) method that calculates the 15% salary tax (salary attribute).

Then create an instance of the StudentTaxPayer class named student and salary 40,000. In response, by calling caicuiate_tax( ) print the calculated tax to the console.

Expected result:

```
6000.0
```

```
from abc import ABC, abstractmethod


class TaxPayer(ABC):
    def __init__(self, salary):
        self.salary = salary

    @abstractmethod
    def calculate_tax(self):
        pass
```

Solution:

```
 from abc import ABC, abstractmethod

class TaxPayer(ABC):

    def __init__(self, salary):
        self.salary = salary

    @abstractmethod
    def calculate_tax(self):
        pass
class StudentTaxPayer(TaxPayer):
```

```
**def calculate_tax(self):
    return self.salary * 0.15
 student = StudentTaxPayer(40000)
 print(student.calculate_tax())**
```

48. An implementation of the Taxpayer abstract class is given. Create a class derived from the TaxPayer class named DisabledTaxPayer that implements the caicuiate_tax( ) method that calculates the minimum value of the following two:

- 12% salary tax (salary attribute)
- 5000.0

Then create an instance of DisabledTaxPayer class named disabled and salary 50,000. In response, by calling caicuiate_tax(), print the calculated tax value to the console.

Expected result:

5000.0

```
from abc import ABC, abstractmethod
class TaxPayer(ABC):
    def __init__(self, salary):
        self.salary = salary

    @abstractmethod
    def calculate_tax(self):
        pass
class StudentTaxPayer(TaxPayer):
    def calculate_tax(self):
        return self.salary * 0.15
```

Solution:

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

    def __init__(self, salary):
        self.salary = salary

    @abstractmethod
    def calculate_tax(self):
        pass
class StudentTaxPayer(TaxPayer):

    def calculate_tax(self):
        return self.salary * 0.15
class DisabledTaxPayer(TaxPayer):
    def calculate_tax(self):
        return min(self.salary * 0.12, 5000.0)
disabled = DisabledTaxPayer(50000)
print(disabled.calculate_tax())
```

49. An implementation of the Taxpayer abstract class is given. Create a class derived from the TaxPayer class named WorkerTaxPayer that implements the caicuiate_tax( ) method that calculates the tax value according to the rule:

- up to the amount of 80,000 -> 17% tax rate
- everything above 80,000 -> 32% tax rate

Then create two instances of WorkerTaxPayer named workerl and worker2 and salaries of 70,000 and 95,000 respectively. In response, by calling caicuiate_tax() print the calculated tax for both instances to the console.

Expected result:

```
11900.0

18400.0
```

```
from abc import ABC, abstractmethod

class TaxPayer(ABC):

    def __init__(self, salary):

        self.salary = salary

    @abstractmethod

    def calculate_tax(self):

  pass

class StudentTaxPayer(TaxPayer):
    def calculate_tax(self):
        return self.salary * 0.15
class DisabledTaxPayer(TaxPayer):

    def calculate_tax(self):
        return self.salary * 0.12
```

Solution:

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

    def __init__(self, salary):
        self.salary = salary

    @abstractmethod
    def calculate_tax(self):
        pass

class StudentTaxPayer(TaxPayer):

    def calculate_tax(self):
        return self.salary * 0.15

         class DisabledTaxPayer(TaxPayer):
```

```
    def calculate_tax(self):
        return self.salary * 0.12

class WorkerTaxPayer(TaxPayer):

    def calculate_tax(self):
        if self.salary < 80000:
            return self.salary * 0.17
        else:
            return 80000 * 0.17 + (self.salary - 80000) * 0.32

worker1 = WorkerTaxPayer(70000)
worker2 = WorkerTaxPayer(95000)
print(worker1.calculate_tax())
print(worker2.calculate_tax())
```

50. The following classes are given:

- StudentTaxPayer
- DisabledTaxPayer
- WorkerTaxPayer

Create a list named tax_payers and assign four instance to it, respectively:

- an instance of the StudentTaxPayer class with a salary of 50,000
- an instance of the DisabledTaxPayer class with a salary of 70,000
- an instance of the WorkerTaxPayer class with a salary of 68,000
- an instance of the WorkerTaxPayer class with a salary of 120,000

Then, iterating through the list, call caicuiate_tax() method on the given instance and print the tax amount to the console.

Expected result:

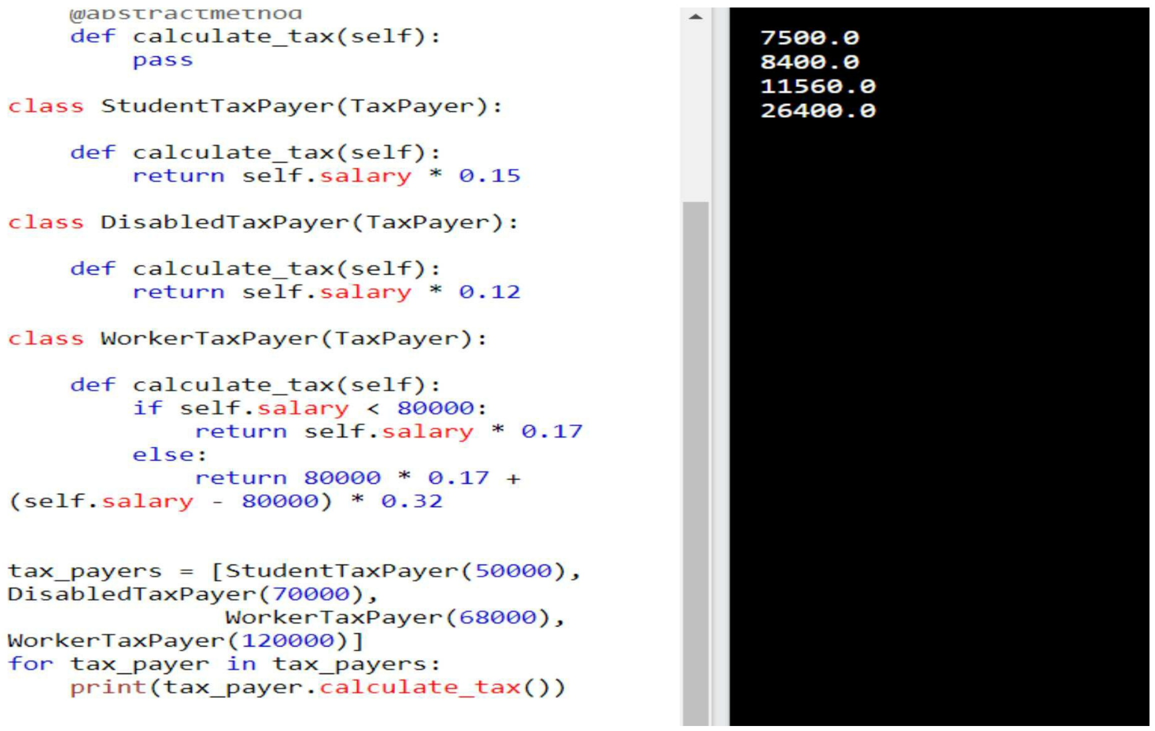
```
7500.0
8400.0
11560.0
26400.0
```

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



    def __init__(self, salary):
        self.salary = salary
    @abstractmethod
    def calculate_tax(self):
        pass
class StudentTaxPayer(TaxPayer):
    def calculate_tax(self):
        return self.salary * 0.15
class DisabledTaxPayer(TaxPayer):
    def calculate_tax(self):
        return self.salary * 0.12
class WorkerTaxPayer(TaxPayer):

    def calculate_tax(self):
        if self.salary < 80000:
            return self.salary * 0.17
        else:
            return 80000 * 0.17 + (self.salary - 80000) * 0.32
```

Solution:

```
from abc import ABC, abstractmethod
class TaxPayer(ABC):
    def __init__(self, salary):
        self.salary = salary
    @abstractmethod
    def calculate_tax(self):
        pass
class StudentTaxPayer(TaxPayer):
    def calculate_tax(self):
        return self.salary * 0.15
```

```
class DisabledTaxPayer(TaxPayer):
    def calculate_tax(self):
        return self.salary * 0.12
class WorkerTaxPayer(TaxPayer):
    def calculate_tax(self):
        if self.salary < 80000:
            return self.salary * 0.17
        else:
            return 80000 * 0.17 + (self.salary - 80000) * 0.32



tax_payers = [StudentTaxPayer(50000),
DisabledTaxPayer(70000),
              WorkerTaxPayer(68000), WorkerTaxPayer(120000)]
for tax_payer in tax_payers:
    print(tax_payer.calculate_tax())
```



```
    @abstractmethod
    def calculate_tax(self):
        pass

class StudentTaxPayer(TaxPayer):

    def calculate_tax(self):
        return self.salary * 0.15

class DisabledTaxPayer(TaxPayer):

    def calculate_tax(self):
        return self.salary * 0.12

class WorkerTaxPayer(TaxPayer):

    def calculate_tax(self):
        if self.salary < 80000:
            return self.salary * 0.17
        else:
            return 80000 * 0.17 +
(self.salary - 80000) * 0.32


tax_payers = [StudentTaxPayer(50000),
DisabledTaxPayer(70000),
              WorkerTaxPayer(68000),
WorkerTaxPayer(120000)]
for tax_payer in tax_payers:
    print(tax_payer.calculate_tax())
```

```
7500.0
8400.0
11560.0
26400.0
```

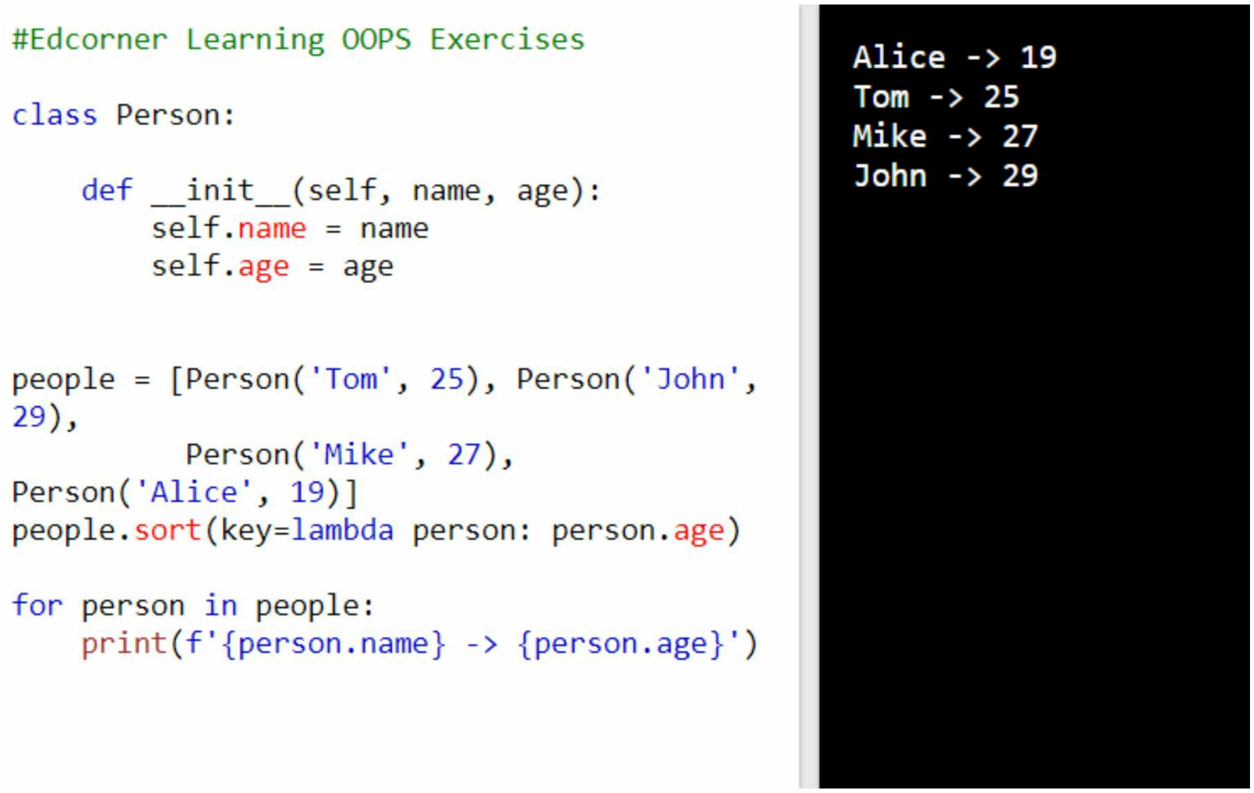
Module 6 Miscelleanuoes Exercises

51. The people list is given. Sort the objects in the people list ascending by age. Then print the name and age to the console as shown below.

Expected result:

```
Alice-> 19
Tom ->25
Mike -27
John -> 29
```

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
people = [Person('Tom', 25), Person('John', 29),
          Person('Mike', 27), Person('Alice', 19)]
```



```
#Edcorner Learning OOPS Exercises

class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age


people = [Person('Tom', 25), Person('John',
29),
          Person('Mike', 27),
Person('Alice', 19)]
people.sort(key=lambda person: person.age)

for person in people:
    print(f'{person.name} -> {person.age}')
```

```
Alice -> 19
Tom -> 25
Mike -> 27
John -> 29
```

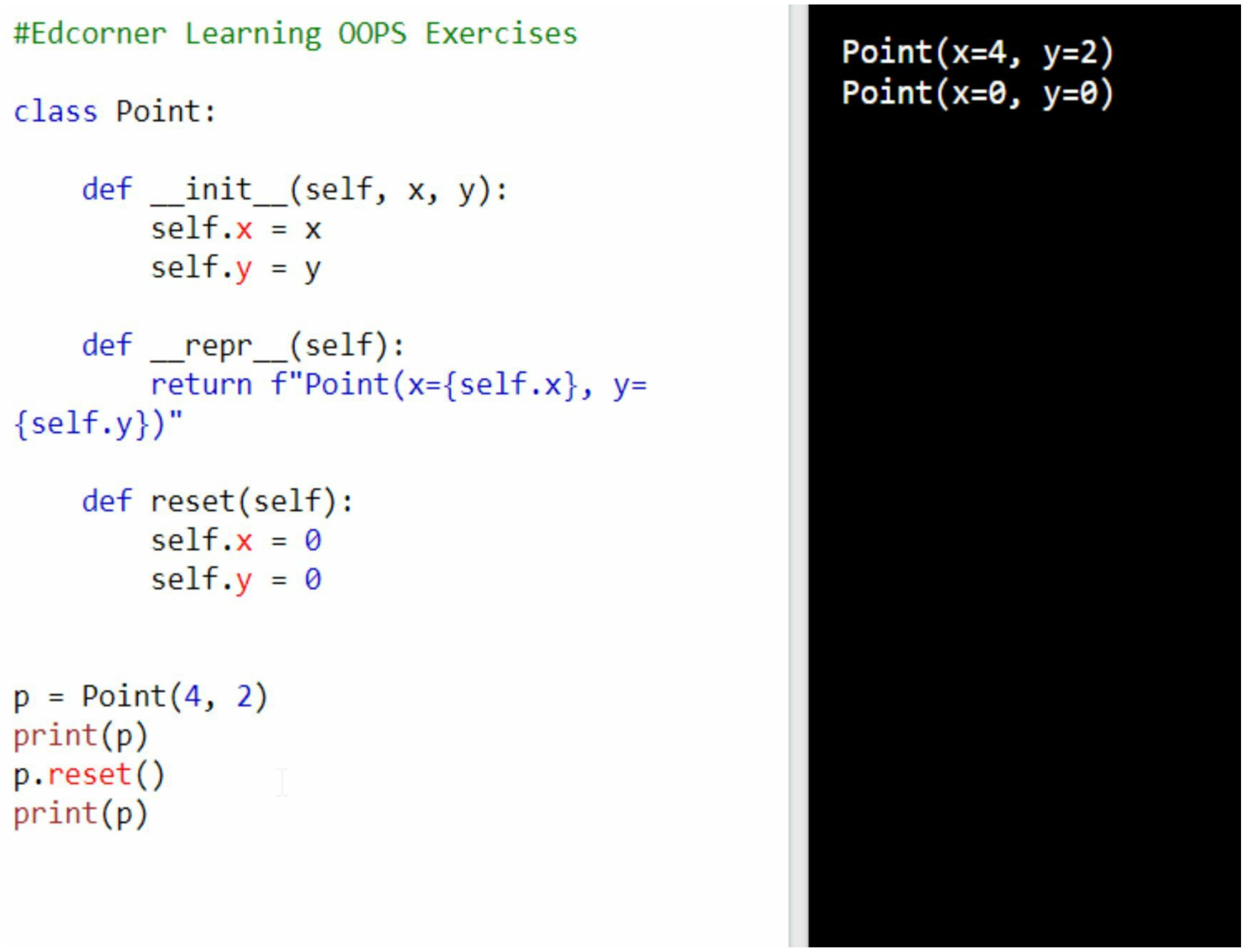
Solution:

52. The following Point class is given. Implement a reset ( ) method that allows you to set the values of the x and y attributes to zero. Then create an instance of the Point class with coordinates (4, 2) and print it to the console. Call the reset ( ) method on this instance and print the instance to the console again.

Expected result:

```
Point(x=4, y=2)
Point(x=0, y=0)
```

```
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Point(x={self.x}, y={self.y})"
```



```
#Edcorner Learning OOPS Exercises

class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Point(x={self.x}, y=
{self.y})"

    def reset(self):
        self.x = 0
        self.y = 0


p = Point(4, 2)
print(p)
p.reset()
print(p)
```

```
Point(x=4, y=2)
Point(x=0, y=0)
```

Solution:

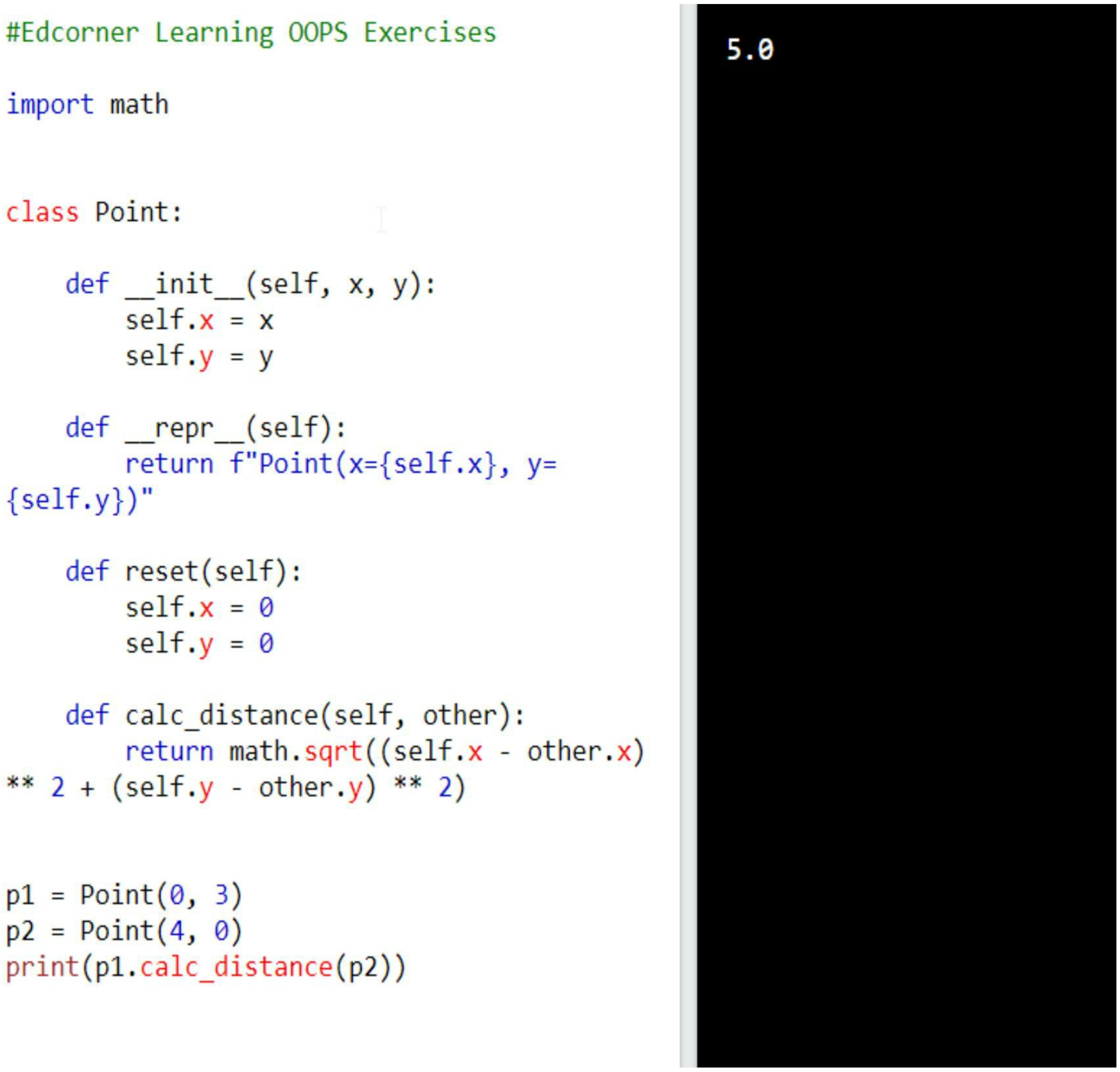
53. The following Point class is given. Implement the caic_distance( ) method that calculates the euclidean distance of two points.

Create two instances of the Point class with the coordinates (0, 3) and (4, 0) and calculate the distance of these points (use the caic_distance( ) method).

Expected result:

5.0

```
import math
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Point(x={self.x}, y={self.y})"

    def reset(self):
        self.x = 0
        self.y = 0
```



```python
#Edcorner Learning OOPS Exercises

import math


class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Point(x={self.x}, y=
{self.y})"

    def reset(self):
        self.x = 0
        self.y = 0

    def calc_distance(self, other):
        return math.sqrt((self.x - other.x)
** 2 + (self.y - other.y) ** 2)


p1 = Point(0, 3)
p2 = Point(4, 0)
print(p1.calc_distance(p2))
```

```
5.0
```

Solution:

54. Implement a class called Note that describes a simple note. When creating Note objects, an instance attribute called content will be set with the contents of the note. Also add instance attribute called creationjime that stores the creation time (use the given date format: '%m-%d- %Y %H:%M:%S').

Next, create two instances of the Note class named notel and note2, and assign the following contents:

'My first note.'

'My second note.'

Solution:

```
import datetime
class Note:

    def __init__(self, content):
        self.content = content
        self.creation_time = datetime.datetime.now().strftime('%m-%d-%Y
%H:%M:%S')
note1 = Note('My first note.')
note2 = Note('My second note.')
```

55. The Note class is given. Implement a find( ) method that checks if a given word is in the note (case sensitive). The method should return True or False, respectively.

Then create an instance named notel with the contents of the note:

'Object Oriented Programming in Python.'

On the notel instance call the find() method to check if the note contains the following words:

- 'python'
- 'Python'

Print the result to the console.

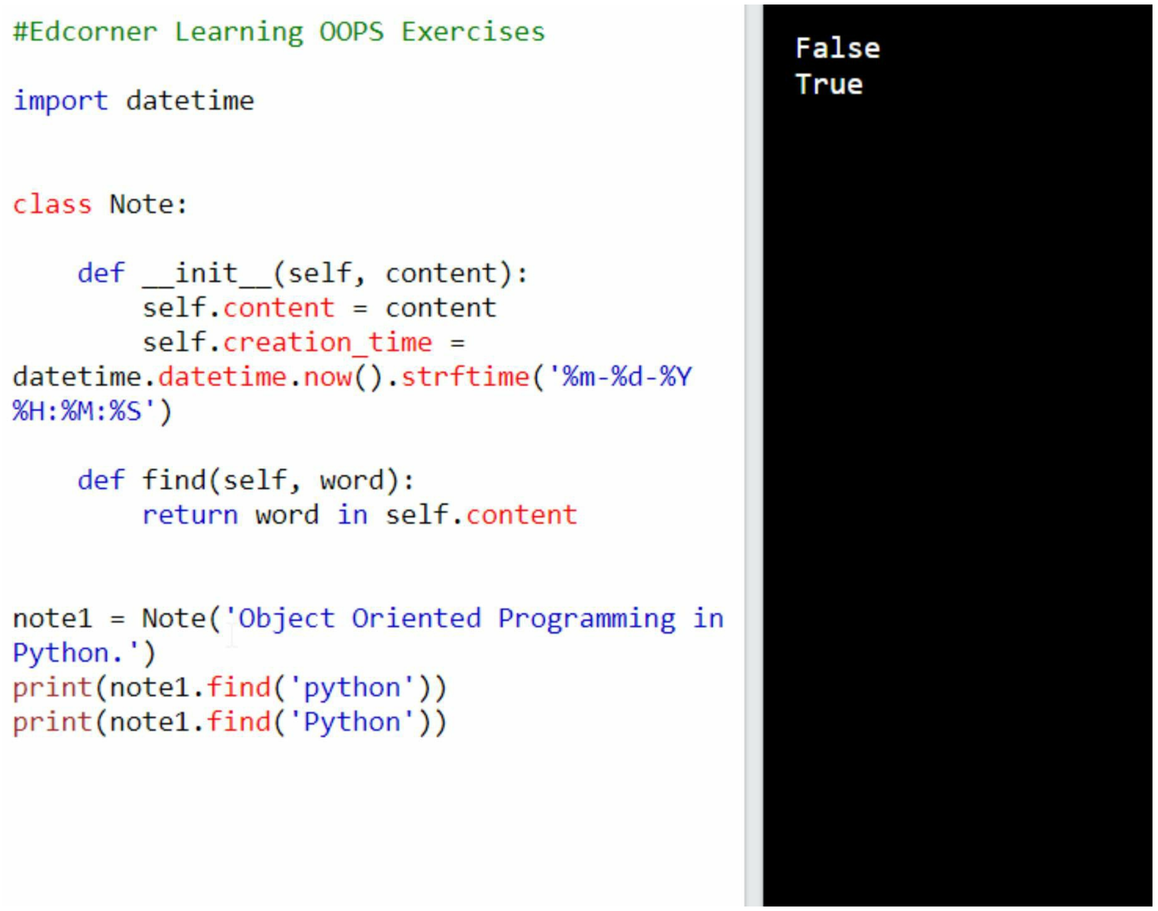
Expected result:

False

True

```
import datetime
class Note:


    def __init__(self, content):
        self.content = content
        self.creation_time = datetime.datetime.now().strftime('%m-%d-%Y
%H:%M:%S')
```

Solution:



```
#Edcorner Learning OOPS Exercises

import datetime


class Note:

    def __init__(self, content):
        self.content = content
        self.creation_time =
datetime.datetime.now().strftime('%m-%d-%Y
%H:%M:%S')

    def find(self, word):
        return word in self.content


note1 = Note('Object Oriented Programming in
Python.')
print(note1.find('python'))
print(note1.find('Python'))
```

```
False
True
```

56. The Note class is given. Implement a find( ) method that checks if a given word is in the note (case insensitive). The method should return True or False, respectively.

Then create an instance named notel with the contents of the note:

'Object Oriented Programming in Python.'

On the note7 instance call the find() method to check if the note contains the following words:

'python'

'Python'

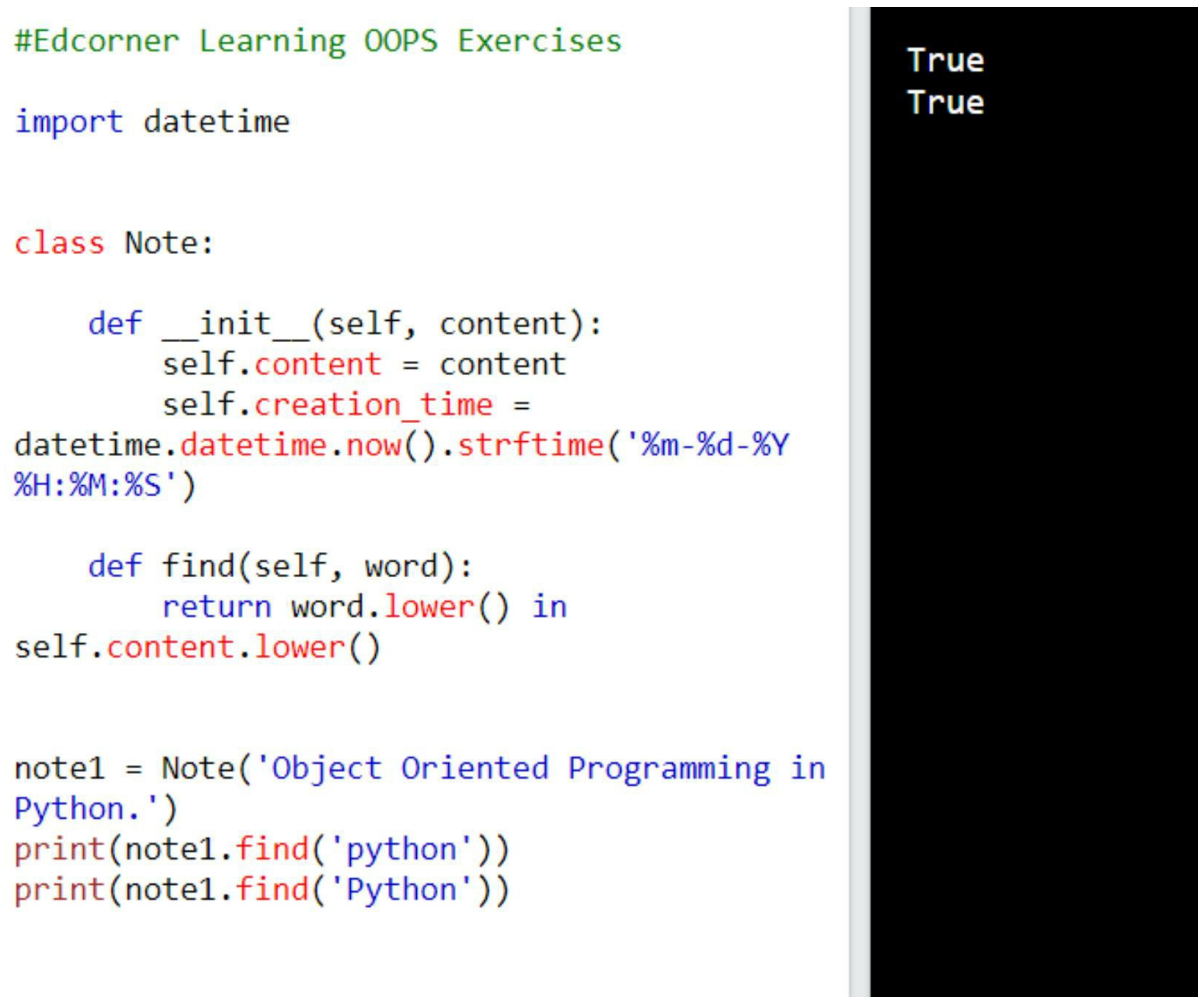
Print the result to the console.

Expected result:

True

True

```
import datetime
class Note:
def __init__(self, content):
self.content = content
self.creation_time = datetime.datetime.now().strftime('%m-%d-%Y
%H:%M:%S')
```



```
#Edcorner Learning OOPS Exercises

import datetime


class Note:

    def __init__(self, content):
        self.content = content
        self.creation_time =
datetime.datetime.now().strftime('%m-%d-%Y
%H:%M:%S')

    def find(self, word):
        return word.lower() in
self.content.lower()


note1 = Note('Object Oriented Programming in
Python.')
print(note1.find('python'))
print(note1.find('Python'))
```

```
True
True
```

Solution:

57. The Note class (representation of a note) is given. Implement the Notebook class (representation of a notebook with notes) with two methods:

- init ( ) for creating an instance attribute of the Notebook class named notes (an

empty list where the notes will be stored).

- new_note( ) for creating a new Note object and adding it to the notes list

Create an instance of the Notebook class named notebook. Then, using the new_note() method add two notes to the notebook with the following content:

'My first note.'

'My second note.'

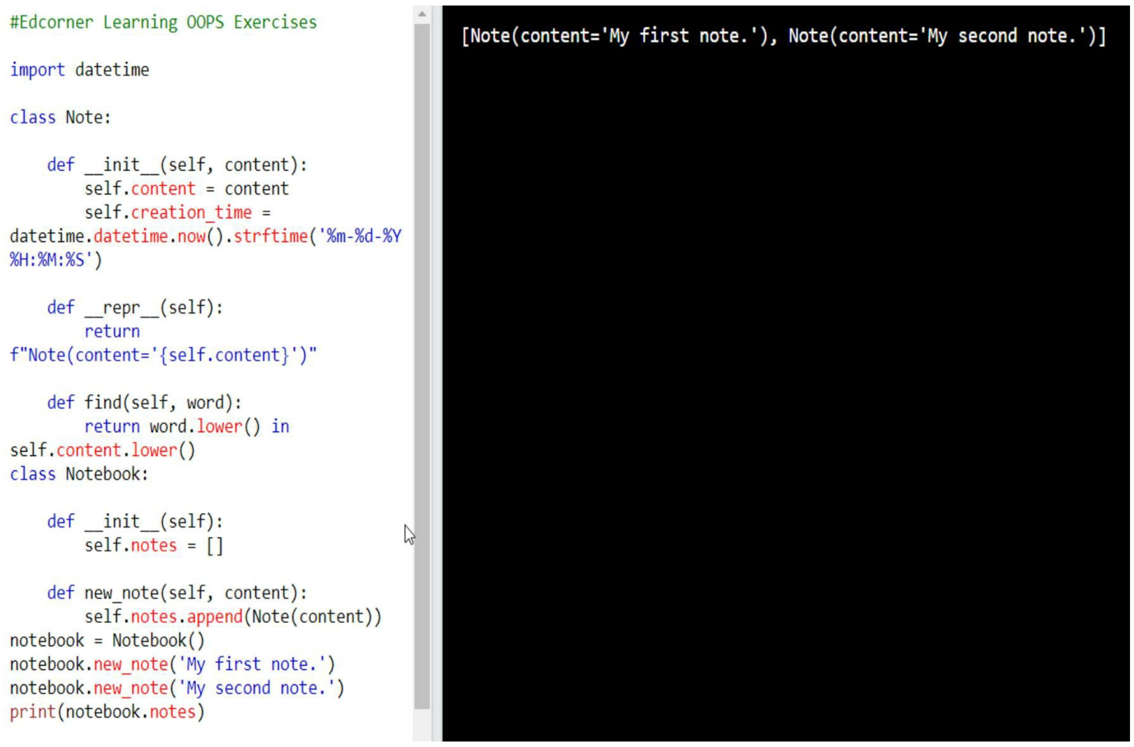
In response, print the content of the notes attribute to the console.

Expected result:

[Note(content='My first note.'), Note(content='My second note.')]

```
 import datetime
  class Note:
    def __init__(self, content):
        self.content = content
        self.creation_time = datetime.datetime.now().strftime('%m-%d-
%Y %H:%M:%S')
    def __repr__(self):
        return f"Note(content='{self.content}')"


def find(self, word):
    return word.lower() in self.content.lower()
```

Solution:



```
#Edcorner Learning OOPS Exercises

import datetime

class Note:

    def __init__(self, content):
        self.content = content
        self.creation_time =
datetime.datetime.now().strftime('%m-%d-%Y
%H:%M:%S')

    def __repr__(self):
        return
f"Note(content='{self.content}')"

    def find(self, word):
        return word.lower() in
self.content.lower()
class Notebook:

    def __init__(self):
        self.notes = []

    def new_note(self, content):
        self.notes.append(Note(content))
notebook = Notebook()
notebook.new_note('My first note.')
notebook.new_note('My second note.')
print(notebook.notes)
```

```
[Note(content='My first note.'), Note(content='My second note.')]
```

58. Implementations of the Note and Notebook class are given. Implement a method named dispiay_notes( ) in the Notebook class to display the content of all notes of the notes instance attribute to the console.

Create an instance of the Notebook class named notebook. Then, using the new_note() method add two notes to the notebook with the following content:

- 'My first note.'
- 'My second note.'

In response, call dispiay_notes() method on the notebook instance.

Expected result:

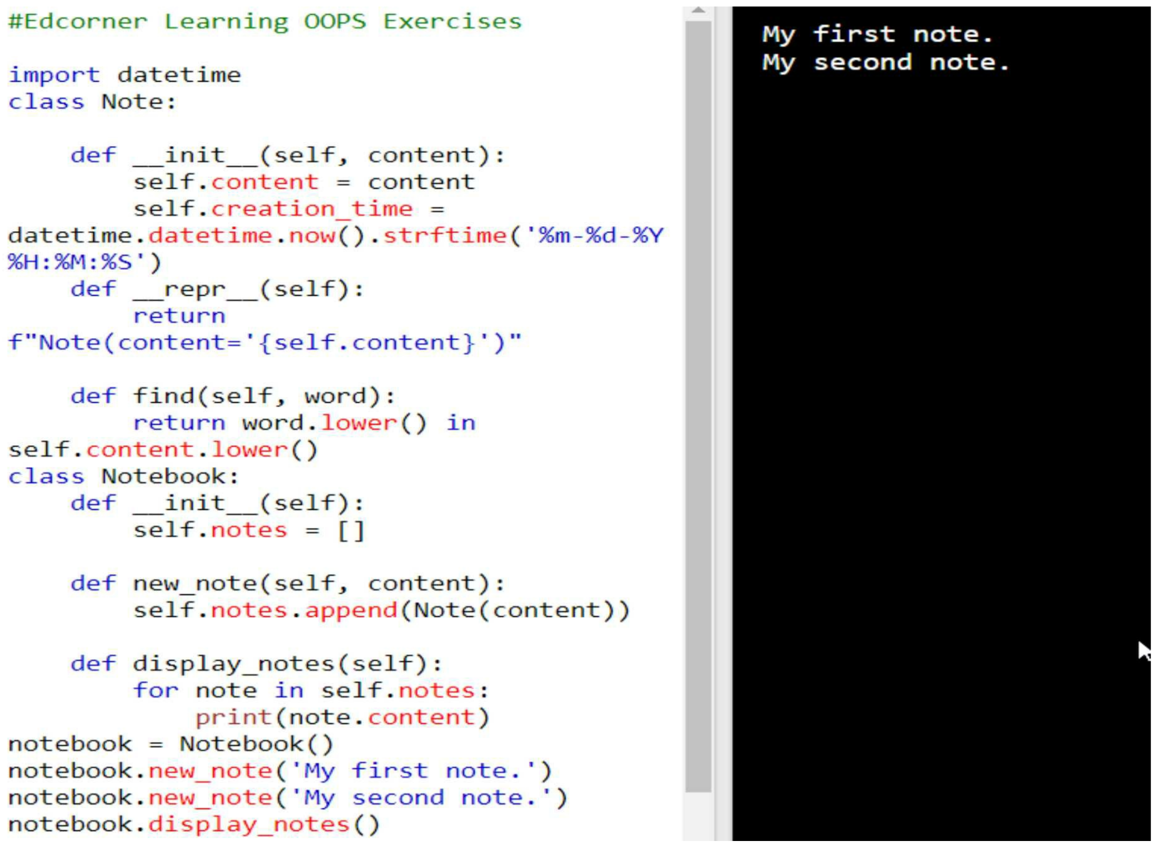
My first note.

My second note.

```
  import datetime
class Note:
   def __init__(self, content):
      self.content = content
      self.creation_time = datetime.datetime.now().strftime('%m-%d-
%Y %H:%M:%S')
   def __repr__(self):
      return f"Note(content='{self.content}')"
   def find(self, word):
return word.lower() in self.content.lower()
  class Notebook:



def __init__(self):
    self.notes = []
 def new_note(self, content):
    self.notes.append(Note(content))
```

Solution:

Solution:



```
#Edcorner Learning OOPS Exercises

import datetime
class Note:

    def __init__(self, content):
        self.content = content
        self.creation_time =
datetime.datetime.now().strftime('%m-%d-%Y
%H:%M:%S')
    def __repr__(self):
        return
f"Note(content='{self.content}')"

    def find(self, word):
        return word.lower() in
self.content.lower()
class Notebook:
    def __init__(self):
        self.notes = []

    def new_note(self, content):
        self.notes.append(Note(content))

    def display_notes(self):
        for note in self.notes:
            print(note.content)
notebook = Notebook()
notebook.new_note('My first note.')
notebook.new_note('My second note.')
notebook.display_notes()
```

```
My first note.
My second note.
```

59. Implementations of the Note and Notebook class are given. Implement a method called search() in the Notebook class that allows you to return a list of notes containing a specific word (passed as an argument to the method, case insensitive). You can use the Note.find method for this.

Create an instance of the Notebook class named notebook. Then, using the new_note() method add notes to the notebook with the following content:

- 'Big Data'
- 'Data Science'
- 'Machine Learning'

In response, call the search() method on the notebook instance looking for notes that contain the Word 'data' .

Expected result:

[Note(content='Big Data'), Note(content='Data Science')]

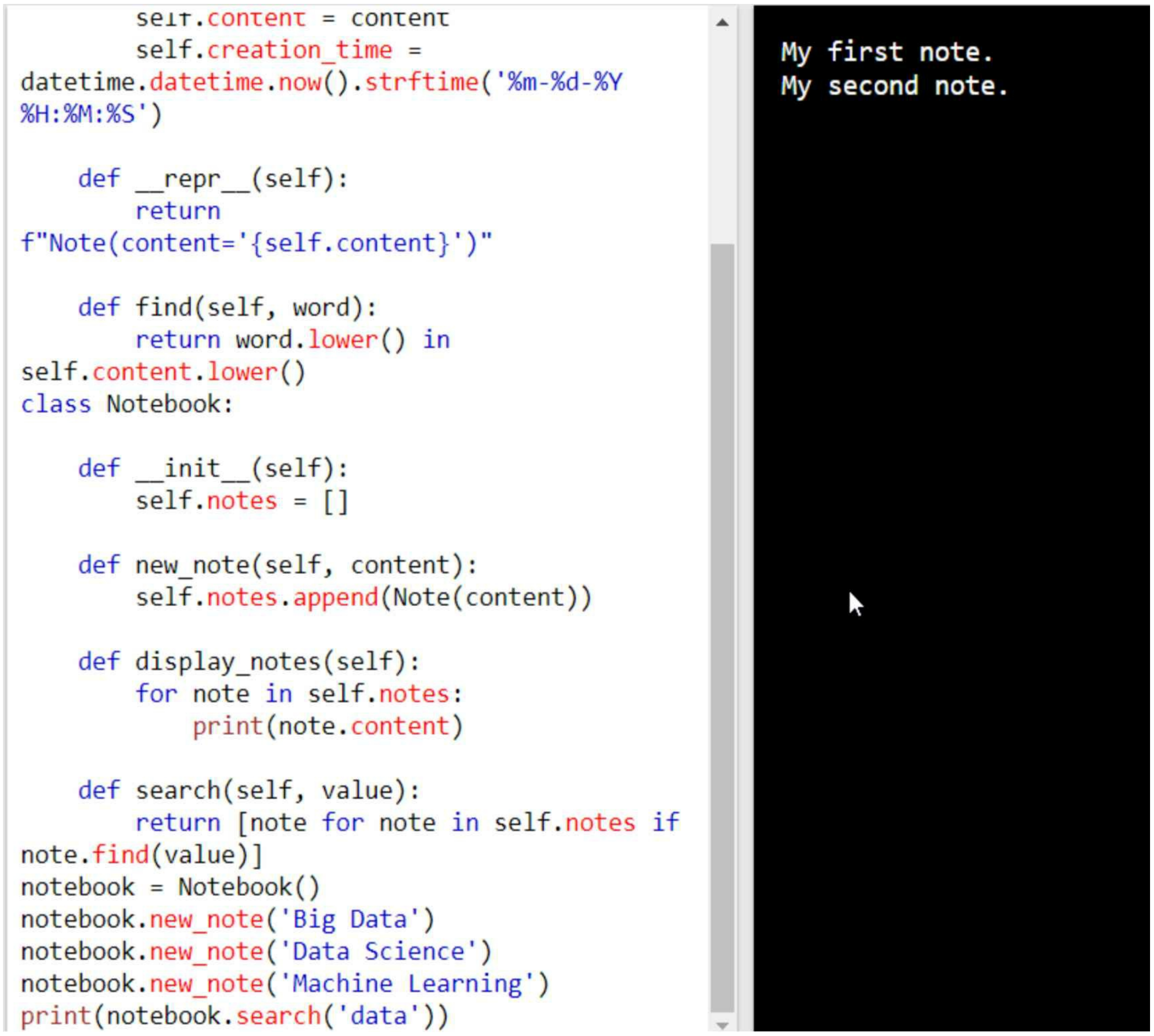
```
import datetime

class Note:


    def __init__(self, content):

        self.content = content

        self.creation_time = datetime.datetime.now().strftime('%m-%d-%Y
%H:%M:%S')


            def __repr__(self):
```

```
        return f"Note(content='{self.content}')"


    def find(self, word):
        return word.lower() in self.content.lower()
class Notebook:
    def __init__(self):
        self.notes = []
    def new_note(self, content):
        self.notes.append(Note(content))
    def display_notes(self):
        for note in self.notes:
            print(note.content)
```

Solution:

```
import datetime
class Note:


    def __init__(self, content):
        self.content = content
        self.creation_time = datetime.datetime.now().strftime('%m-%d-%Y %H:%M:%S')
    def __repr__(self):
        return f"Note(content='{self.content}')"



    def find(self, word):
        return word.lower() in self.content.lower()
```

```
class Notebook:

    def __init__(self):
        self.notes = []
    def new_note(self, content):
        self.notes.append(Note(content))
    def display_notes(self):
        for note in self.notes:
            print(note.content)
    def search(self, value):
        return [note for note in self.notes if note.find(value)]
notebook = Notebook()
notebook.new_note('Big Data')
notebook.new_note('Data Science')
notebook.new_note('Machine Learning')
print(notebook.search('data'))
```



```
        self.content = content
        self.creation_time =
datetime.datetime.now().strftime('%m-%d-%Y
%H:%M:%S')

    def __repr__(self):
        return
f"Note(content='{self.content}')"

    def find(self, word):
        return word.lower() in
self.content.lower()
class Notebook:

    def __init__(self):
        self.notes = []

    def new_note(self, content):
        self.notes.append(Note(content))

    def display_notes(self):
        for note in self.notes:
            print(note.content)

    def search(self, value):
        return [note for note in self.notes if
note.find(value)]
notebook = Notebook()
notebook.new_note('Big Data')
notebook.new_note('Data Science')
notebook.new_note('Machine Learning')
print(notebook.search('data'))
```

```
My first note.
My second note.
```

60. Implement a class named Client which has a class attribute named all_clients (as a list). Then the _init_( ) method sets two instance attributes (no validation):

- name
- email

Add this instance to the alLclients list (Client class attribute). Also add a _repr_( ) method the Client class (see below).

Create three clients by executing the following code:

Clientl = Client(‘Tom’, ‘sample@gmail.com’)

client2 = Client(‘Donald', ‘sales@yahoo.com’)

client3 = Client('Mike’, ‘sales-contact@yahoo.com’)

In response, print the all_cients attribute of the Client class.

Expected Result:

[Client(name='Tom', email='sample@gmail.com'), Client(name='Donald', email='sales@yahoo.com'), Client(name='Mike', email='sales-contact@yahoo.com')]

Solution:

```
class Client:


 all_clients = []


 def __init__(self, name, email):
         self.name = name
         self.email = email
         Client.all_clients.append(self)
```

```
    def __repr__(self):
        return f"Client(name='{self.name}', email='{self.email}')"

client1 = Client('Tom', 'sample@gmail.com')
client2 = Client('Donald', 'sales@yahoo.com')
client3 = Client('Mike', 'sales-contact@yahoo.com')
print(Client.all_clients)
```

61. The Client class is implemented. Note the class attribute all_clients. Try to implement a special class extending the built-in list class called ClientList, which in addition to the standard methods for the built-in class list will have a search_emaii() method that allows you to return a list of Client class instances containing the text (value argument) in the email address.

For example, the following code:

```
Clientl = Client('Tom', 'sample@gmail.com')
client2 = Client('Donald', 'sales@yahoo.com')
client3 = Client('Mike', 'sales-contact@yahoo.com')
client4 = Client(1 Lisa1, 'info@gmail.com' )
print(Client.all_clients.search_email('sales'))
```

```
class ClientList(list)
    def search_email(self, value):
        pass
class Client:
    all_clients = ClientList()
    def __init__(self, name, email):
        self.name = name
                self.email = email
                Client.all_clients.append(self)



            def __repr__(self):
                return f"Client(name='{self.name}', email='{self.email}')"
```

Solution:

```
     class ClientList(list):

         def search_email(self, value):
             result = [client for client in self if value in client.email]
             return result

     class Client:

         all_clients = ClientList()
```

```
    def __init__(self, name, email):
        self.name = name
        self.email = email
        Client.all_clients.append(self)

    def __repr__(self):
        return f"Client(name='{self.name}', email='{self.email}')"


client1 = Client('Tom', 'sample@gmail.com')
client2 = Client('Donald', 'sales@gmail.com')
client3 = Client('Mike', 'sales@yahoo.com')
client4 = Client('Lisa', 'info@gmail.com')
print(Client.all_clients.search_email('sales'))
```

62. The Client class is implemented. Create the following four instances of the Client class:

For example, the following code:

```
Clientl = Client(‘Tom’, ‘sample@gmail.com’)
client2 = Client(‘Donald', ‘sales@yahoo.com’)
client3 = Client('Mike’, ‘sales-contact@yahoo.com’)
client4 = Client(1 Lisa1, 'info@gmail.com' )
```

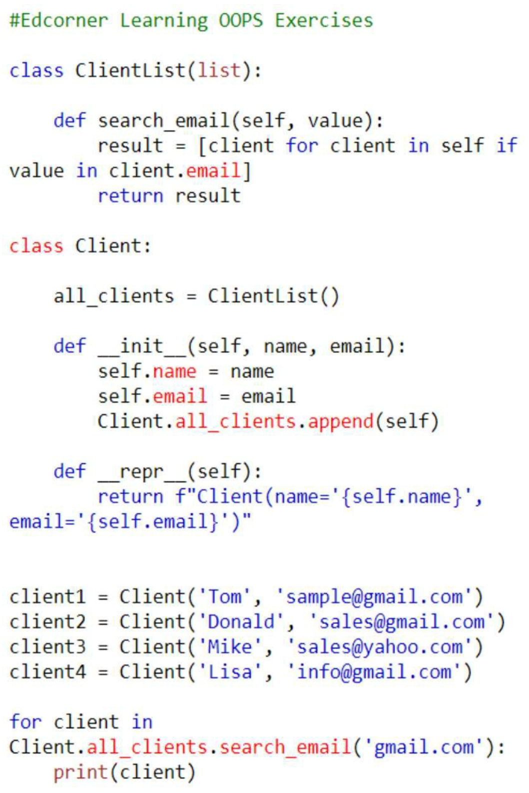
Then search for all customers who have a gmail account (‘gmail ' in email address). In response, print result to the console as shown below.

Expected result:

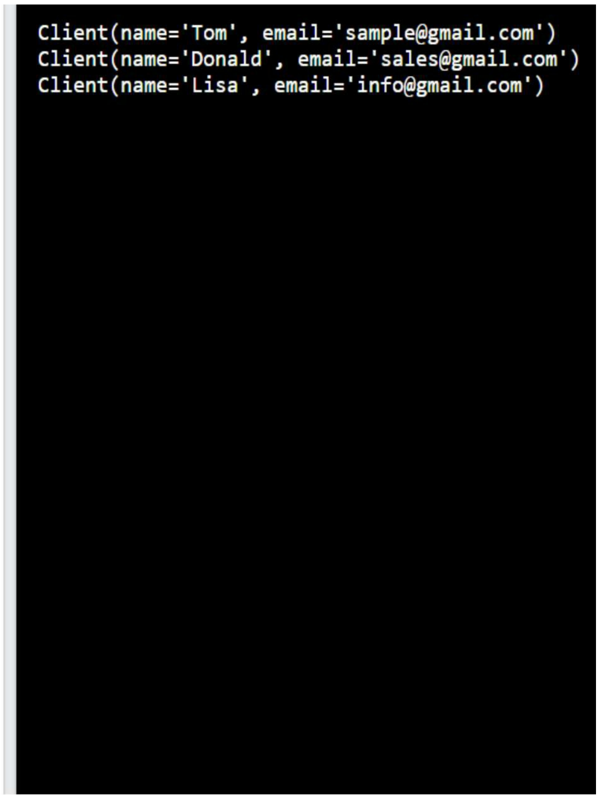
```
Client(name='Tom', email='sample@gmail.com')
Client(name='Donald', email='sales@gmail.com')
Client(name='Lisa', email='info@gmail.com')
```

```
class ClientList(list):

def search_email(self, value):
        result = [client for client in self if value in client.email]
        return result


class Client:

    all_clients = ClientList()

    def __init__(self, name, email):
        self.name = name
        self.email = email
        Client.all_clients.append(self)

    def __repr__(self):
        return f"Client(name='{self.name}', email='{self.email}')"
```

Solution:



```
#Edcorner Learning OOPS Exercises

class ClientList(list):

    def search_email(self, value):
        result = [client for client in self if
value in client.email]
        return result

class Client:

    all_clients = ClientList()

    def __init__(self, name, email):
        self.name = name
        self.email = email
        Client.all_clients.append(self)

    def __repr__(self):
        return f"Client(name='{self.name}',
email='{self.email}')"


client1 = Client('Tom', 'sample@gmail.com')
client2 = Client('Donald', 'sales@gmail.com')
client3 = Client('Mike', 'sales@yahoo.com')
client4 = Client('Lisa', 'info@gmail.com')

for client in
Client.all_clients.search_email('gmail.com'):
    print(client)
```



```
Client(name='Tom', email='sample@gmail.com')
Client(name='Donald', email='sales@gmail.com')
Client(name='Lisa', email='info@gmail.com')
```

63. The Client class is implemented. The following four instances of the Client class:

For example, the following code:

Clientl = Client('Tom', 'sample@gmail.com')

```
client2 = Client(‘Donald', ‘sales@yahoo.com’)
client3 = Client('Mike’, ‘sales-contact@yahoo.com’)
client4 = Client(1 Lisa1, 'info@gmail.com' )
```

Search for all customers with the word ‘sales ' email address. In response, print the names

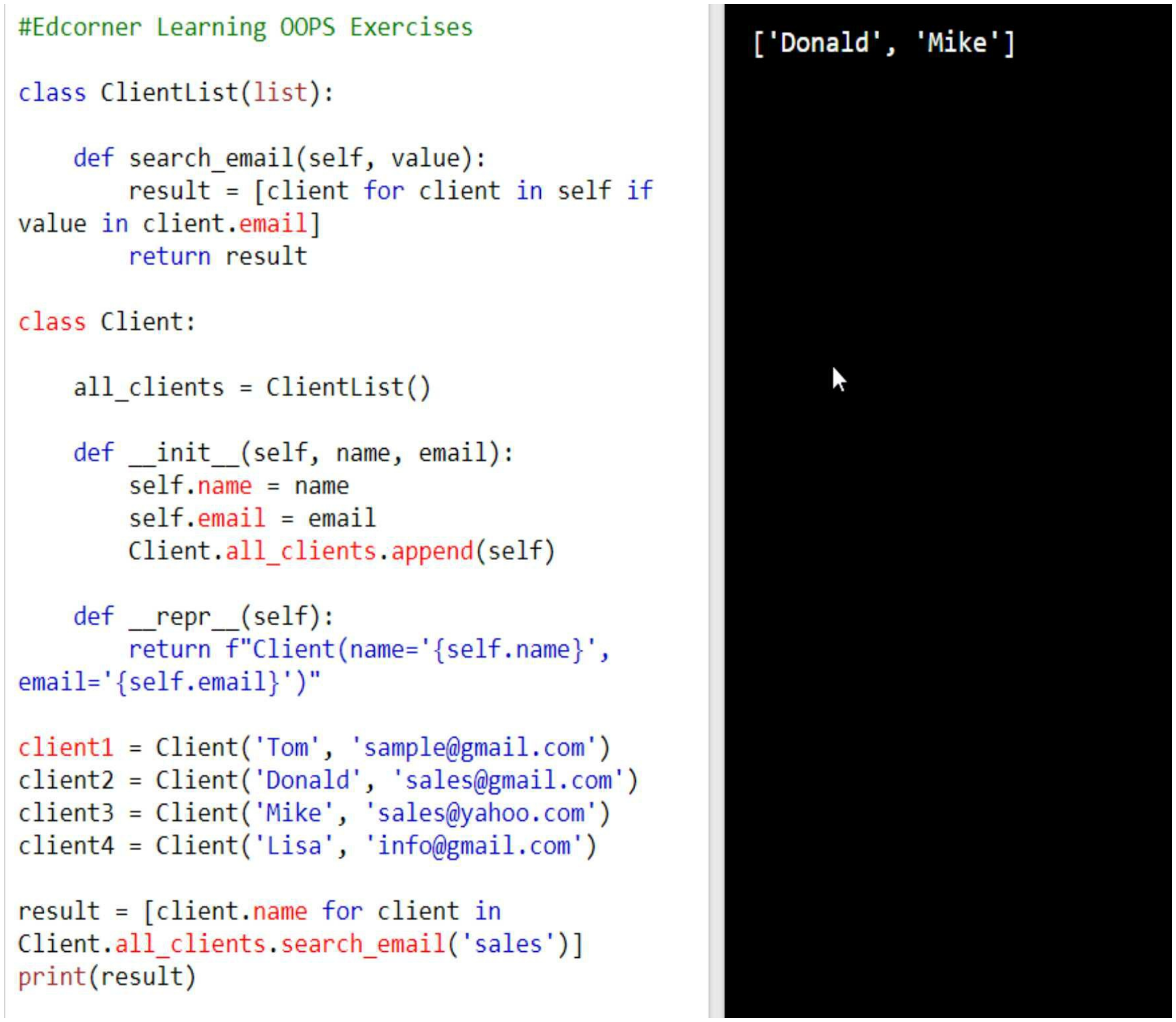
of the customers as a list to the console.

Expected result:

['Donald', 'Mike']

```
class ClientList(list):

    def search_email(self, value):
        result = [client for client in self if value in client.email]
        return result

class Client:

        all_clients = ClientList()

        def __init__(self, name, email):
            self.name = name
            self.email = email
            Client.all_clients.append(self)

        def __repr__(self):
            return f"Client(name='{self.name}', email='{self.email}')"
```

```
client1 = Client('Tom', 'sample@gmail.com')
client2 = Client('Donald', 'sales@gmail.com')
client3 = Client('Mike', 'sales@yahoo.com')
client4 = Client('Lisa', 'info@gmail.com')
```



```
#Edcorner Learning OOPS Exercises

class ClientList(list):

    def search_email(self, value):
        result = [client for client in self if
value in client.email]
        return result

class Client:

    all_clients = ClientList()

    def __init__(self, name, email):
        self.name = name
        self.email = email
        Client.all_clients.append(self)

    def __repr__(self):
        return f"Client(name='{self.name}',
email='{self.email}')"

client1 = Client('Tom', 'sample@gmail.com')
client2 = Client('Donald', 'sales@gmail.com')
client3 = Client('Mike', 'sales@yahoo.com')
client4 = Client('Lisa', 'info@gmail.com')

result = [client.name for client in
Client.all_clients.search_email('sales')]
print(result)
```

```
['Donald', 'Mike']
```

Solution:

64. Create a class named CustomDict that extends the built-in diet class. Add a method named is_any_str_vaiue() that returns a boolean value:

• True in case the created dictionary contains at least one value of str type

- otherwise False.

Example I:

```
[IN]: cd = CustonDict(python=’mid')
  [IN]: print(cd.ls_any_str_value())
```

returns:

```
[OUT]: True
```

Example II:

```
    [IN]: cd = CustomDict(price=119.99)
   [IN]: print(cd.ls_any_str_value())
```

returns:

```
[OUT]: False
```

You only need to implement the CustomDict class.

Solution:

```
   class CustomDict(dict):


  def is_any_str_value(self):
      flag = False
      for key in self:
```

```
    if isinstance(self[key], str):
        flag = True
        break
return flag
```

65. Create a class named StringListOnly that extends the built-in list class. Modify the behavior of the append () method so that only objects of str type can be added to the list. If you try to add a different type of object raise TypeE rror with message:

'Only objects of type str can be added to the list.'

Then create an instance of the StringListOnly class and add the following objects with the append() method:

'Data'

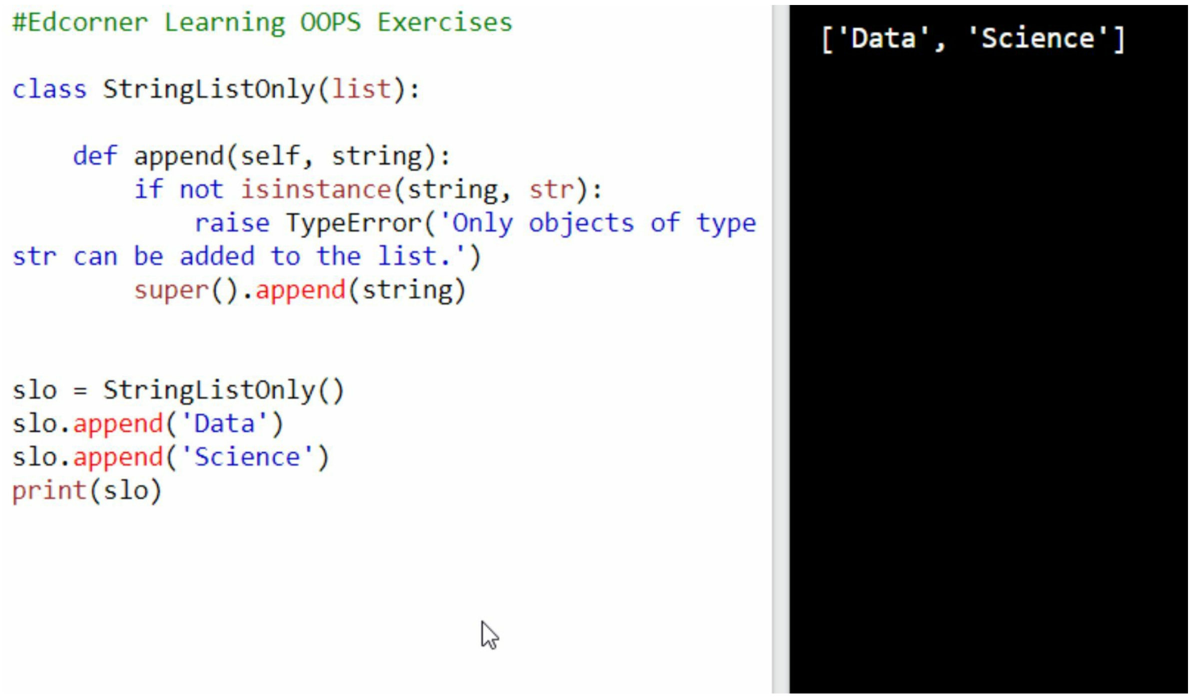
'Science'

In response, print result to the console.

Expected result:

['Data', 'Science']

Solution:



```
#Edcorner Learning OOPS Exercises

class StringListOnly(list):

    def append(self, string):
        if not isinstance(string, str):
            raise TypeError('Only objects of type
str can be added to the list.')
        super().append(string)


slo = StringListOnly()
slo.append('Data')
slo.append('Science')
print(slo)
```

```
['Data', 'Science']
```

66. Create a class named StringListOnly that extends the built-in list class. Modify the behavior of the append() method so that only objects of str type an be added to the list. Replace all uppercase letters with lowercase before adding the object to the list. If you try to add a different type of object raise TypeE rror with message:

'Only objects of type str can be added to the list.'

Then create an instance of the StringListOnly class and add the following objects with the append() method:

'Data'

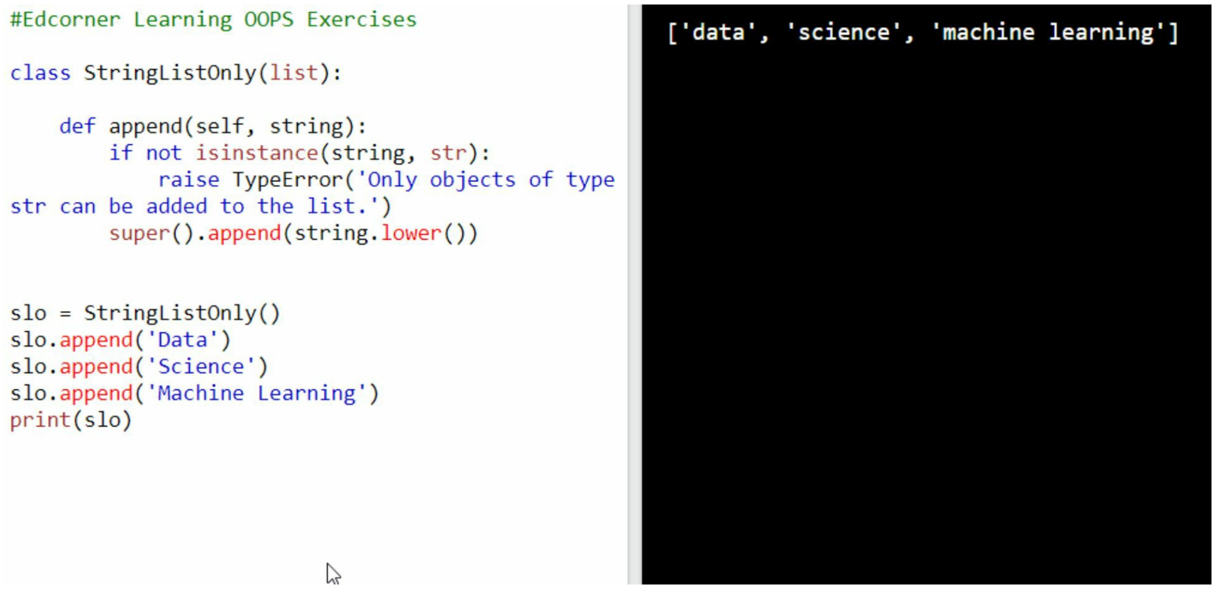
- 'Science'
- 'Machine Learning'

In response, print result to the console.

Expected result:

['data', 'science', 'machine learning']

Solution:



```python
#Edcorner Learning OOPS Exercises

class StringListOnly(list):

    def append(self, string):
        if not isinstance(string, str):
            raise TypeError('Only objects of type 
str can be added to the list.')
        super().append(string.lower())


slo = StringListOnly()
slo.append('Data')
slo.append('Science')
slo.append('Machine Learning')
print(slo)
```

```
['data', 'science', 'machine learning']
```

67. An implementation of the Product class is given. Implement a class named Warehouse which in the init ( ) method sets an instance attribute of the Warehouse class named products to

an empty list.

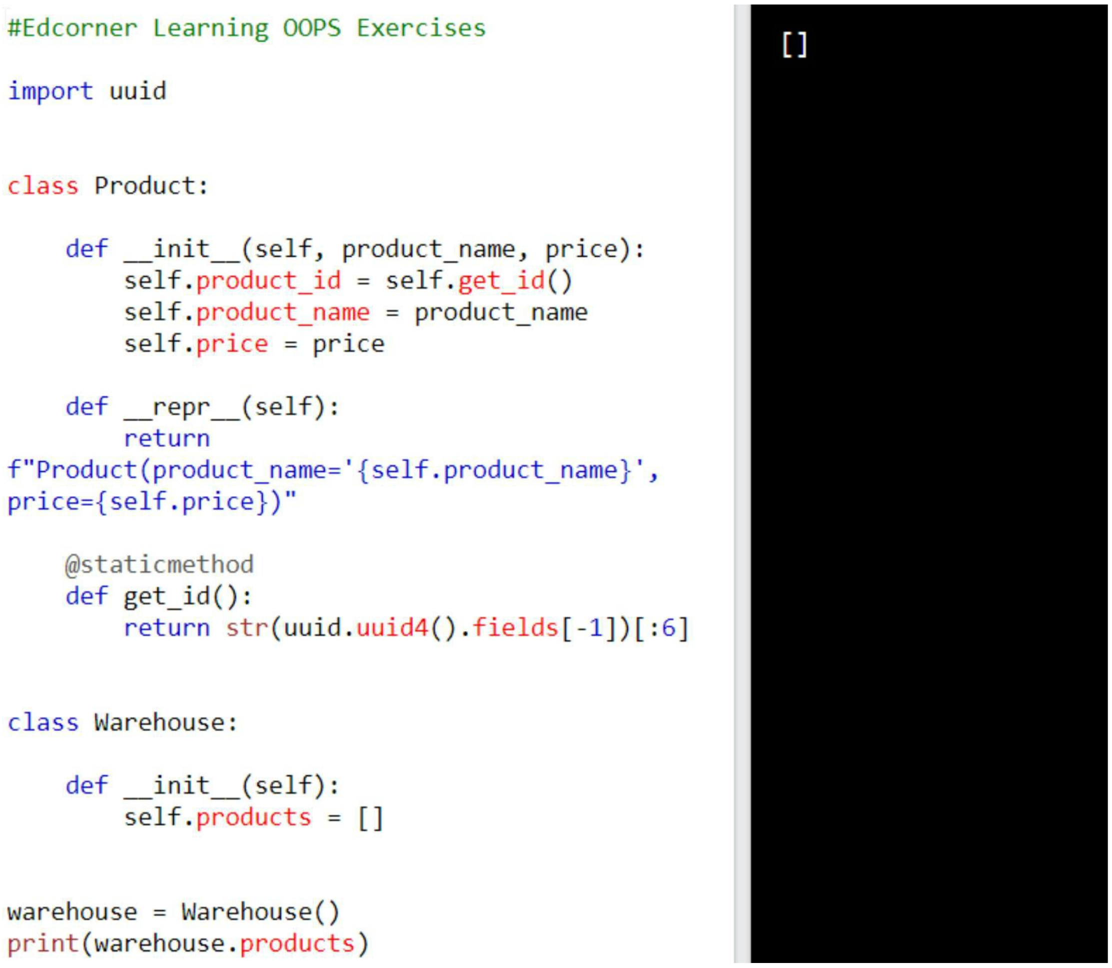
Then create an instance of the Warehouse class named warehouse and display the value of the products attribute to the console.

Expected result:

```
 []
```

```
import uuid



class Product:


    def __init__(self, product_name, price):
        self.product_id = self.get_id()
        self.product_name = product_name
        self.price = price







def __repr__(self):
return f"Product(product_name='{self.product_name}', price=
{self.price})"

    @staticmethod
    def get_id():
        return str(uuid.uuid4().fields[-1])[:6]
```

Solution:



```
#Edcorner Learning OOPS Exercises

import uuid


class Product:

    def __init__(self, product_name, price):
        self.product_id = self.get_id()
        self.product_name = product_name
        self.price = price

    def __repr__(self):
        return
f"Product(product_name='{self.product_name}',
price={self.price})"

    @staticmethod
    def get_id():
        return str(uuid.uuid4().fields[-1])[:6]


class Warehouse:

    def __init__(self):
        self.products = []


warehouse = Warehouse()
print(warehouse.products)
```

```
[]
```

68. The implementation of the classes: Product and Warehouse is given. To the Warehouse class, add a method named add_product( ) that allows you to add an instance of the Product class to the products list. If the product name is already in the products list, skip adding the product.

Next, create an instance of the Warehouse class named warehouse. Using the add_product() method add the following products:

'Laptop', 3900.0

'Mobile Phone', 1990.0

'Mobile Phone', 1990.0

Note that the second and third products are duplicates. The add_product() method should avoid adding duplicates. Print the products attribute of the warehouse instance to the console.

Expected result:

[Product(product_name='Laptop', price=3900.0), Product(product_name='Mobile Phone', price=1990.0)]

```
import uuid

class Product:


    def __init__(self, product_name, price):
        self.product_id = self.get_id()
        self.product_name = product_name
        self.price = price




    def __repr__(self):
        return f"Product(product_name='{self.product_name}', price=
{self.price})"


    @staticmethod
    def get_id():
        return str(uuid.uuid4().fields[-1])[:6]
```

```
class Warehouse:

    def __init__(self):
        self.products = []
```

Solution:

```
import uuid


class Product:

    def __init__(self, product_name, price):
        self.product_id = self.get_id()
self.product_name = product_name
self.price = price
```

```
    def __repr__(self):
        return f"Product(product_name='{self.product_name}', price=
{self.price})"

    @staticmethod
    def get_id():
        return str(uuid.uuid4().fields[-1])[:6]


class Warehouse:

    def __init__(self):
        self.products = []

    def add_product(self, product_name, price):
        product_names = [product.product_name for product in
self.products]
        if not product_name in product_names:
            self.products.append(Product(product_name, price))


warehouse = Warehouse()
warehouse.add_product('Laptop', 3900.0)
warehouse.add_product('Mobile Phone', 1990.0)
print(warehouse.products)
```

69. The implementation of the classes: Product and Warehouse is given. To the Warehouse class, add a method named remove_product( ) that allows you to remove an instance of the Product class from the products list with a given product name. If the product name is not in the products list, just skip.

Next, create an instance of the Warehouse class named warehouse. Using the add_product() method add the following products:

'Laptop', 3900.0

'Mobile Phone', 1990.0

• 'Camera', 2900.0

Then, using the remove_product() method, remove the product named 'Mobile Phone' . In response, print the products attribute of the warehouse instance to the console.

Expected result:

```
[Product(product_name='Laptop', price=3900.0),
Product(product_name='Camera', price=2900.0)]
```

```
import uuid
```

```
class Product:

    def __init__(self, product_name, price):
        self.product_id = self.get_id()
        self.product_name = product_name
        self.price = price

            def __repr__(self):
              return f"Product(product_name='{self.product_name}', price=
{self.price})"

              @staticmethod
              def get_id():
                  return str(uuid.uuid4().fields[-1])[:6]


          class Warehouse:

              def __init__(self):
                  self.products = []

              def add_product(self, product_name, price):
                  product_names = [product.product_name for product in
          self.products]
                  if not product_name in product_names:
                      self.products.append(Product(product_name, price))
```

Solution:

```
import uuid



class Product:

    def __init__(self, product_name, price):
        self.product_id = self.get_id()
        self.product_name = product_name
        self.price = price

    def __repr__(self):
        return f"Product(product_name='{self.product_name}', price=
{self.price})"

    @staticmethod
    def get_id():
        return str(uuid.uuid4().fields[-1])[:6]
```

```
class Warehouse:


    def __init__(self):
        self.products = []


    def add_product(self, product_name, price):
        product_names = [product.product_name for product in
self.products]
        if not product_name in product_names:
            self.products.append(Product(product_name, price))


    def remove_product(self, product_name):
        for product in self.products:
            if product_name == product.product_name:
                self.products.remove(product)
warehouse = Warehouse()
warehouse.add_product('Laptop', 3900.0)
warehouse.add_product('Mobile Phone', 1990.0)
warehouse.add_product('Camera', 2900.0)
warehouse.remove_product('Mobile Phone')
print(warehouse.products)
```

70. The implementation of the classes: Product and Warehouse is given. To the Product class, add a _str_( ) method that is an informal representation of the Product class.

An example of how the _str_( ) method works. The code below:

returns:

product = Product(1 Laptop', 3900.0) print(product)

Then create an instance of the Product class named product with the arguments passed:

• 'Mobile Phone', 1990.0

In response, print the product instance to the console.

Expected result:

Product Name: Mobile Phone | Price: 1990.0

```
import uuid
class Product:


    def __init__(self, product_name, price):
        self.product_id = self.get_id()
        self.product_name = product_name
        self.price = price


    def __repr__(self):
        return f"Product(product_name='{self.product_name}', price=
{self.price})"


            @staticmethod
```

```
 def get_id():
return str(uuid.uuid4().fields[-1])[:6]



 class Warehouse:

    def __init__(self):
        self.products = []

    def add_product(self, product_name, price):
        product_names = [product.product_name for product in
 self.products]
        if not product_name in product_names:
            self.products.append(Product(product_name, price))

    def remove_product(self, product_name):
        for product in self.products:
            if product_name == product.product_name:
                self.products.remove(product)
```

Solution:

import uuid

class Product:

```
    def __init__(self, product_name, price):
        self.product_id = self.get_id()
        self.product_name = product_name
        self.price = price

    def __repr__(self):
        return f"Product(product_name='{self.product_name}', price=
{self.price})"

    def __str__(self):
        return f'Product Name: {self.product_name} | Price: {self.price}'

    @staticmethod
    def get_id():
        return str(uuid.uuid4().fields[-1])[:6]


class Warehouse:

    def __init__(self):
        self.products = []

    def add_product(self, product_name, price):
        product_names = [product.product_name for product in
self.products]
        if not product_name in product_names:
            self.products.append(Product(product_name, price))

        def remove_product(self, product_name):
            for product in self.products:
                if product_name == product.product_name:
                    self.products.remove(product)


product = Product('Mobile Phone', 1990.0)
print(product)
```

71. The implementation of the classes: Product and Warehouse is given. Add a method to the Warehouse class named dispiay_products() that displays all products in the products attribute of the Warehouse class.

Then create an instance of the Warehouse class named warehouse and execute the following code:

```
warehouse.add_product(1 Laptop', 3900.0)
warehouse.add_product('Mobile Phone', 1990.0)
warehouse.add_product('Cañera', 2900.0)
```

In response, call dispiay_products() method on the warehouse instance.

Expected result:

```
Product Name: Laptop | Price: 3900.0
Product Name: Mobile Phone | Price: 1990.0
Product Name: Camera | Price: 2900.0
```

```
 import uuid

class Product:

    def __init__(self, product_name, price):
        self.product_id = self.get_id()
        self.product_name = product_name
                self.price = price

            def __repr__(self):
                return f"Product(product_name='{self.product_name}', price=
        {self.price})"

            def __str__(self):
                return f'Product Name: {self.product_name} | Price: {self.price}'

            @staticmethod
            def get_id():
                return str(uuid.uuid4().fields[-1])[:6]


        class Warehouse:
```

```
    def __init__(self):
        self.products = []

    def add_product(self, product_name, price):
        product_names = [product.product_name for product in
self.products]
        if not product_name in product_names:
            self.products.append(Product(product_name, price))

    def remove_product(self, product_name):
        for product in self.products:
            if product_name == product.product_name:
                self.products.remove(product)
```

Solution:

```
import uuid
class Product:

    def __init__(self, product_name, price):
        self.product_id = self.get_id()
        self.product_name = product_name
        self.price = price

    def __repr__(self):
        return f"Product(product_name='{self.product_name}', price=
{self.price})"
```

```
def __str__(self):
    return f'Product Name: {self.product_name} | Price: {self.price}'

@staticmethod
def get_id():
            return str(uuid.uuid4().fields[-1])[:6]



    class Warehouse:

        def __init__(self):
            self.products = []

        def add_product(self, product_name, price):
            product_names = [product.product_name for product in
    self.products]
            if not product_name in product_names:
                self.products.append(Product(product_name, price))

        def remove_product(self, product_name):
            for product in self.products:
                if product_name == product.product_name:
                    self.products.remove(product)

        def display_products(self):
```

```
        for product in self.products:
            print(product)


warehouse = Warehouse()
warehouse.add_product('Laptop', 3900.0)
warehouse.add_product('Mobile Phone', 1990.0)
warehouse.add_product('Camera', 2900.0)
warehouse.display_products()
```

72. The implementation of the classes: Product and Warehouse is given. Add a method called sort_by_price( ) to the Warehouse class that returns an alphabetically sorted list of products. The sort_by_price( ) method also takes an argument ascending set to True by default, which means an ascending sort. If False is passed, reverse the sort order.

Then create an instance of the Warehouse class named warehouse and execute the following code:

```
warehouse.add_product(1 Laptop', 3900.0)
warehouse.add_product('Mobile Phone', 1990.0)
warehouse.add_product('Cañera', 2900.0)
warehouse.add_product('USB Cable', 24.9)
warehouse.add_product('House', 49.0)
```

In response, use the sort_by_price( ) method to print a sorted list of products to the console as shown below.

Expected result:

```
Product(product_name='USB Cable', price=24.9)
Product(product_name='Mouse', price=49.0)
Product(product_name='Mobile Phone', price=1990.0)
Product(product_name='Camera', price=2900.O)
Product(product_name='Laptop', price=3900.0)
```

```
import uuid
```

```
class Product:

        def __init__(self, product_name, price):
            self.product_id = self.get_id()
            self.product_name = product_name
            self.price = price

        def __repr__(self):
            return f"Product(product_name='{self.product_name}',
      price={self.price})"

        @staticmethod
        def get_id():
            return str(uuid.uuid4().fields[-1])[:6]



      class Warehouse:

        def __init__(self):
  self.products = []

    def add_product(self, product_name, price):
        product_names = [product.product_name for product in
   self.products]
        if not product_name in product_names:
            self.products.append(Product(product_name, price))

    def remove_product(self, product_name):
        for product in self.products:
            if product_name == product.product_name:
                self.products.remove(product)

    def display_products(self):
        for product in self.products:
            print(f'Product ID: {product.product_id} | Product name: '
                  f'{product.product_name} | Price: {product.price}')
```

Solution:

```
import uuid


class Product:

    def __init__(self, product_name, price):
        self.product_id = self.get_id()
        self.product_name = product_name
        self.price = price

    def __repr__(self):
        return f"Product(product_name='{self.product_name}', price=
{self.price})"

    @staticmethod
    def get_id():
        return str(uuid.uuid4().fields[-1])[:6]


class Warehouse:

    def __init__(self):
        self.products = []

    def add_product(self, product_name, price):
        product_names = [product.product_name for product in
self.products]
        if not product_name in product_names:
            self.products.append(Product(product_name, price))

    def remove_product(self, product_name):
        for product in self.products:
            if product_name == product.product_name:
                self.products.remove(product)

    def display_products(self):
        for product in self.products:
            print(f'Product ID: {product.product_id} | Product name: '
                  f'{product.product_name} | Price: {product.price}')

    def sort_by_price(self, ascending=True):
        return sorted(self.products, key=lambda product:
product.price,
                      reverse=not ascending)


warehouse = Warehouse()
warehouse.add_product('Laptop', 3900.0)
warehouse.add_product('Mobile Phone', 1990.0)
warehouse.add_product('Camera', 2900.0)
warehouse.add_product('USB Cable', 24.9)
warehouse.add_product('Mouse', 49.0)
for product in warehouse.sort_by_price():
    print(product)
```

73. The implementation of the classes: Product and Warehouse is given. Complete the implementation of the method named search_product( ) of the Warehouse class that allows you to return a list of products containing the specified name (query argument).

Then create an instance of the Warehouse class named warehouse and execute the following code:

warehouse.add_product(1 Laptop', 3900.0)

warehouse.add_product('Mobile Phone', 1990.0)

warehouse.add_product('Cañera', 2900.0)

warehouse.add_product(1 USB Cable', 24.9)

warehouse.add_product(1 Mouse1, 49.0)

In response, call search_product() method and find all products that contain the letter 'm'.

Expected Result:

[Product(product_name='Mobile Phone', price=1990.0), Product(product_name='Mouse', price=49.0)]

```
 import uuid
class Product:

    def __init__(self, product_name, price):
        self.product_id = self.get_id()
        self.product_name = product_name
        self.price = price



    def __repr__(self):
        return f"Product(product_name='{self.product_name}', price=
{self.price})"

    @staticmethod
    def get_id():
        return str(uuid.uuid4().fields[-1])[:6]


class Warehouse:

    def __init__(self):
        self.products = []

    def add_product(self, product_name, price):
        product_names = [product.product_name for product in self.products]
    if not product_name in product_names:
        self.products.append(Product(product_name, price))

def remove_product(self, product_name):
    for product in self.products:
        if product_name == product.product_name:
            self.products.remove(product)

        def display_products(self):
            for product in self.products:
                print(f'Product ID: {product.product_id} | Product name: '
                      f'{product.product_name} | Price: {product.price}')

        def sort_by_price(self, ascending=True):
            return sorted(self.products, key=lambda product: product.price,
                          reverse=not ascending)

        def search_product(self, query):
            pass
```

Solution:

```
import uuid
class Product:

    def __init__(self, product_name, price):
        self.product_id = self.get_id()
        self.product_name = product_name
        self.price = price

    def __repr__(self):
        return f"Product(product_name='{self.product_name}', price=
{self.price})"

    @staticmethod
    def get_id():
        return str(uuid.uuid4().fields[-1])[:6]



class Warehouse:

          def __init__(self):
           self.products = []
    def add_product(self, product_name, price):
        product_names = [product.product_name for product in
self.products]
        if not product_name in product_names:
            self.products.append(Product(product_name, price))

    def remove_product(self, product_name):
        for product in self.products:
            if product_name == product.product_name:
                self.products.remove(product)

    def display_products(self):
        for product in self.products:
            print(f'Product ID: {product.product_id} | Product name: '
                  f'{product.product_name} | Price: {product.price}')

    def sort_by_price(self, ascending=True):
        return sorted(self.products, key=lambda product:
product.price,
                      reverse=not ascending)

    def search_product(self, query):
        return [prod for prod in self.products if query in
prod.product_name]
```

```
warehouse = Warehouse()
warehouse.add_product('Laptop', 3900.0)
warehouse.add_product('Mobile Phone', 1990.0)
warehouse.add_product('Camera', 2900.0)
warehouse.add_product('USB Cable', 24.9)
warehouse.add_product('Mouse', 49.0)
print(warehouse.search_product('M'))
```

ABOUT THE AUTHOR

“Edcorner Learning” and have a significant number of students on **Udemy** with more than **90000+ Student and Rating of 4.1 or above.**