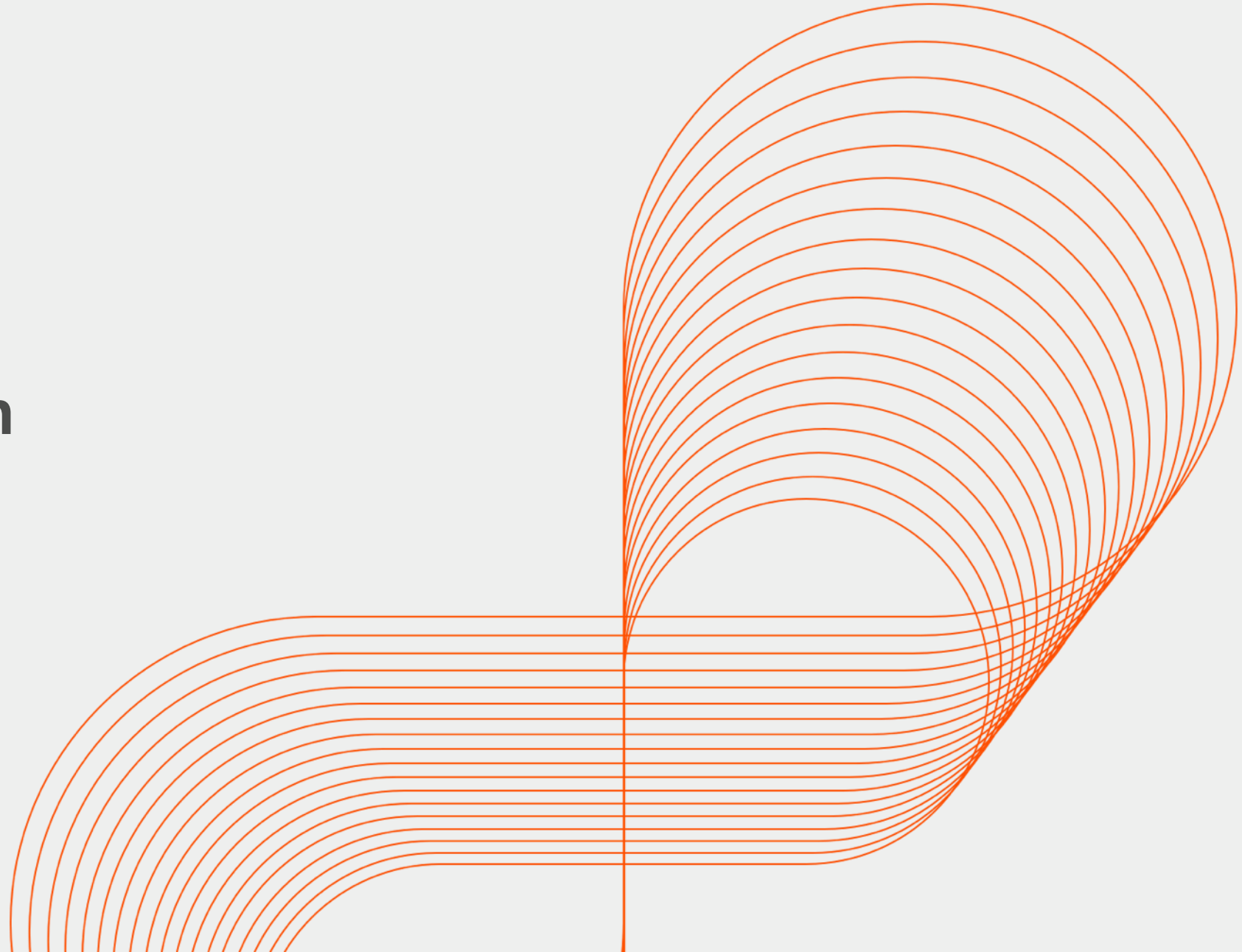
Persistent

OOP in Python



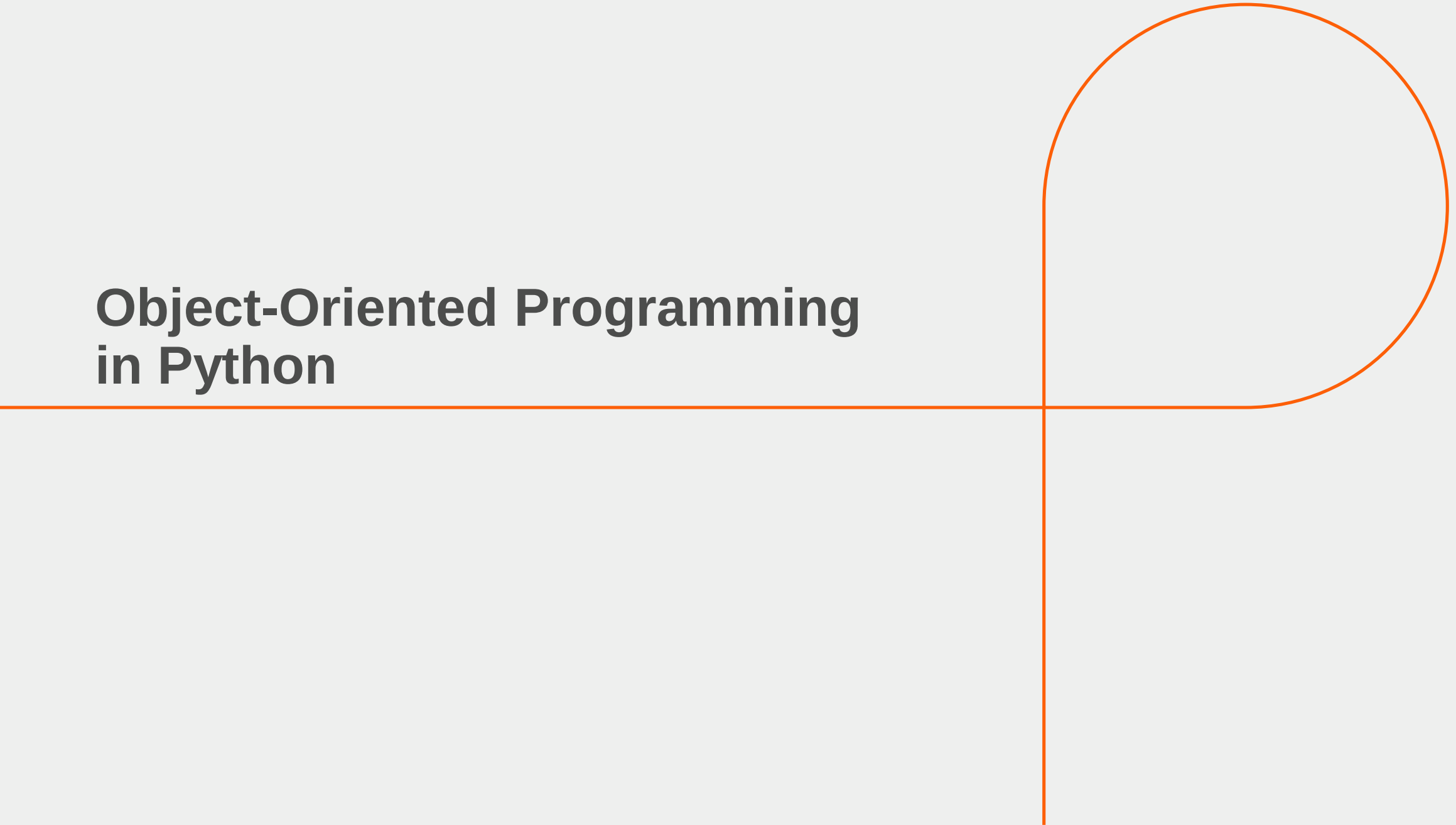
Objectives

At the end of this second session, you will be able to understand:

- Object-oriented programming in Python

Persistent

Object-Oriented Programming in Python



Objectives

At the end of this first session, you will be able to:

Learn basic fundamentals of Object Oriented Programming in Python

OOPs in Python

- Defining Classes
- Using instance variables
- Defining methods
- Defining class variables and methods
- Inheriting from other classes
- Inheriting from multiple classes

Introduction to OOPs

Python is primarily designed as an object-oriented programming language. The main two entities in Python object-oriented programming are classes and class instances.

Python classes, which can be used in a manner analogous to C structures, but which can also be used in a full object-oriented manner.

A **class** is defined with the class statement as shown:

```
class MyClass:

    body
```

A **new object** of the class type (an instance of the class) can be created by calling the class name as a function:

```
instance = MyClass()
```

Creating instance objects and instance variables

Example:

```
>>> class SayHello:
        def __init__(self):
            self.greeting = "Hello, how are you“
>>> hello1 = SayHello()
>>> print (hello1.greeting)
Hello, how are you
```


Persistent

Methods

- A method is a function associated with a particular class
- In Python, methods are defined as part of the class definition, but can be invoked by an instance
- In other words, the path one must take to finally be able to call a method goes like this: (1) define the class and the methods, (2) create an instance, and finally, (3) invoke the method from that instance
- Here is an example class with a method:

```
class MyClassWithMethod:          # define the class
    def sayHello(self):                    # define the method
        print ('You invoked sayHello()!')
```

The following code instantiates the class and invokes the method once an instance is created

```
>>> myObj = MyClassWithMethod()          # create the instance
>>> myObj. sayHello()                    # now invoke the method
'You invoked sayHello()!
```

Class variables

- A class variable is a variable associated with a class, not an instance of a class, and is accessed by all instances of the class, in order to keep track of some class-level information, such as how many instances of the class have been created at any point in time
- A class variable is created by an assignment in the class body, not in the __init__ function. It can be seen by all instances of the class after it has been created

Example:

```
class Circle:
    pi = 3.14159
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return self.radius * self.radius * Circle.pi
```

Class variables (contd.)

- Note the two uses of radius in the example. self.radius is the instance variable called radius. radius by itself is the local function variable called radius. The two aren't the same!
- The class variable can be accessed and used as shown below:

```
>>> Circle.pi
3.1415899999999999
>>> Circle.pi = 4
>>> Circle.pi
4
```

- Notice in this example that Circle.pi is accessed before any circle instances have been created. Obviously, Circle.pi exists independently of any specific instances of the Circle class

Static methods

- Python classes can also have methods that correspond explicitly to static methods in a language such as Java. Just as in Java, static methods can be invoked using the class name even though no instance of that class has been created, although they can be called using a class instance.
- To create a static method, use the **@staticmethod decorator** in the class Circle shown earlier.

Example:

```
@staticmethod
def total_area():
    total = 0
    for c in Circle.all_circles:
        total = total + c.area()
        return total
```


Persistent

Class methods

- Class methods are similar to static methods, in that they can be invoked before an object of the class has been instantiated or by using an instance of the class. But class methods are implicitly passed the class they belong to as their first parameter.
- To create a static method, use the **@classmethod decorator** in the class Circle shown earlier. The class parameter is traditionally cls . cls can be used instead of self.__class__.

Example:

```
@classmethod
def total_area(cls):
    total = 0
    for c in cls.all_circles:
        total = total + c.area()
        return total
```


Persistent

Inheritance

- Inheritance describes how the attributes of base classes are “inherited" to a derived class. A subclass inherits attributes of any of its base classes whether they be data attributes or methods.

Example:

```
class Parent:                    # parent class
    '''Parent class'''
    def __init__(self):
        print ('created an instance of', \
                self.__class__.__name__)
class Child(Parent):             # child class
    pass
```


Persistent

Overriding methods through inheritance

- Let Parent have another function that will be overridden in its child class as shown below.

```
class Parent:

    def greet(self):

        print('Hi, I am Parent-greet()')

        p = Parent()


p.greet()

#Hi, I am Parent-foo()
```

- Now let us create the child class Child, subclassed from parent Parent as shown below.

```
class Child(Parent):

    def greet(self):

        print('Hi, I am Child-greet()')
```

Multiple inheritance

- Python allows for subclassing from multiple base classes. This feature is commonly known as “multiple inheritance”

- Python supports a limited form of multiple inheritance whereby a depth-first searching algorithm is employed to collect the attributes to assign to the derived class. Multiple inheritance takes the first name that is found

- The example below consists of a pair of parent classes, a pair of child classes and one grandchild class.

Multiple inheritance Example

```
class Parent1:                  # parent class 1
    def foo(self):
        print ('called Parent1-foo()')
class Parent2:                  # parent class 2
    def foo(self):
        print ('called Parent2-foo()')
    def bar(self):
        print ('called Parent2-bar()')
```

Multiple inheritance Example

- The example below consists of a pair of parent classes, a pair of child classes and one grandchild class.

```
class Child1(Parent1,Parent2):          # child 1 derived from Parent1, Parent2
    pass
class Child2(Parent1,Parent2):          # child 2 derived from Parent1, Parent2
    def foo(self):                      #over riding methods
        print ('called Child2-foo()')
    def bar(self):
        print ('called Child2-bar()')
class GrandChild(Child1,Child2):        # define grandchild class
    pass                                # derived from Child1 and Child2
```

Multiple inheritance Example

- Upon executing the above declarations , it can be confirmed that only the first attributes encountered are used as shown below.

```
gc = GrandChild()

gc.foo()          #called Child2-foo()

gc.bar()          #called Child2-bar()
```

- Again, a specific method can always be called by invoking the method using its fully-qualified name and providing a valid instance as shown below

```
Parent1.foo(gc)     #called Parent1-foo()
```

Customizing classes with special methods

- The other important matter is that there are two special methods which provide the functionality of constructors and destructors, namely __init__() and __del__() respectively
- Some of the special methods allow for a powerful form of extending classes in Python. In particular, they allow for Overloading operators
- Special methods enable classes to emulate standard types by overloading standard operators such as +, * etc.

- __str__() or __repr__(): it can display something meaningful rather than the generic Python object string (<object type at id>) and display an ordered pair (tuple) with the current data values in the object
- The following code implements the __str__() so that the ordered pair is displayed

```
def __str__(self):          # str() string representation
    return str(self.data) # convert tuple to string
__repr__ = __str__
# repr() string representation
```

Customizing classes with special methods

- The following example shows a simple class consisting of an ordered pair (x, y) of numbers. This data will be represented as a 2-tuple.
- The example defines the class with a constructor that takes a pair of values and stores them as the data attribute of oPair class.

```
class oPair:                        # ordered pair
    def __init__(self, obj1, obj2):  # constructor
        self.data = (obj1, obj2)     # assign attribute
```

- Using this class, the objects can be instantiated as follows

```
>>> myPair = oPair(6, -4)           # create instance
>>> myPair                          # calls repr()
<oPair instance at 92bb50>
>>> print (myPair)                  # calls str()
<oPair instance at 92bb50>
```

Customizing classes with special methods (Cont'd)

- As shown above, neither print using str() nor the actual object's string representation using repr() reveals much useful details about the object.
- One good idea would be to implement either __str__() or __repr__(), or both so that it can display what the object looks like.
- In other words, it can display something meaningful rather than the generic Python object string (<object type at id>) and display an ordered pair (tuple) with the current data values in the object.
- The following code implements the __str__() so that the ordered pair is displayed.

```
def __str__(self):                  # str() string representation
    return str(self.data)           # convert tuple to string
__repr__ = __str__                  # repr() string representation
```

- The output shown in the following slide is greatly improved with the above customization.

Customizing classes with special methods (Cont'd)

- The following shows the output after the customization.

```
>>> myPair = oPair(-5, 9)          # create instance
>>> myPair                         # repr() calls __repr__()
(-5, 9)
>>> print (myPair)                 # str() calls __str__()
(-5, 9)
```

- Continuing with the above let's say the oPair objects need to interact with each other for an addition operation of two oPair objects (x1, y1) and (x2, y2) to be the sum of each individual component.
- Therefore, the "sum" of two oPair objects is defined as a new object with the values (x1 + x2, y1 + y2).
- The __add__() special method can be implemented in such a way that the individual sums are calculated first, then the class constructor is called to return a new object as shown in the following slide.

Customizing classes with special methods (Cont'd)

- The following shows the overloaded definition of addition operation.

```
def __add__(self, other):    # add two oPair objects
    return self.__class__(self.data[0] + other.data[0],self.data[1] + other.data[1])
```

- As shown above the addition operator is overloaded and the output is as shown below.

```
>>> pair1 = oPair(6, -4)
>>> pair2 = oPair(-5, 9)
>>> pair1 + pair2
(1, 5)
```

Assignments

1. Create Employee Class

Maintain class level variable “empCount” and write function “displayCount()” to display the total empCount

Define instance variables Name and salary

Also write instance method “displayEmployee() to display all employee details (Name and Salary)

Override __str__() and __repr__() methods to display Employee details (Name and Salary), instead of default string representation

Summary:

With this we have come to an end of this session, where we discussed…

- Object-oriented programming in Python

In the next session we will discuss about

- Python Functional Programming

Reference material

- http://www.tutorialspoint.com/python
- http://www.learnpython.org/

Questions

Key contacts

Sakshi Jamgaonkar

sakshi_jamgaonkar@persistent.com

Asif Immanad

asif_immanad@persistent.co.in

Persistent

Persistent

Thank you!