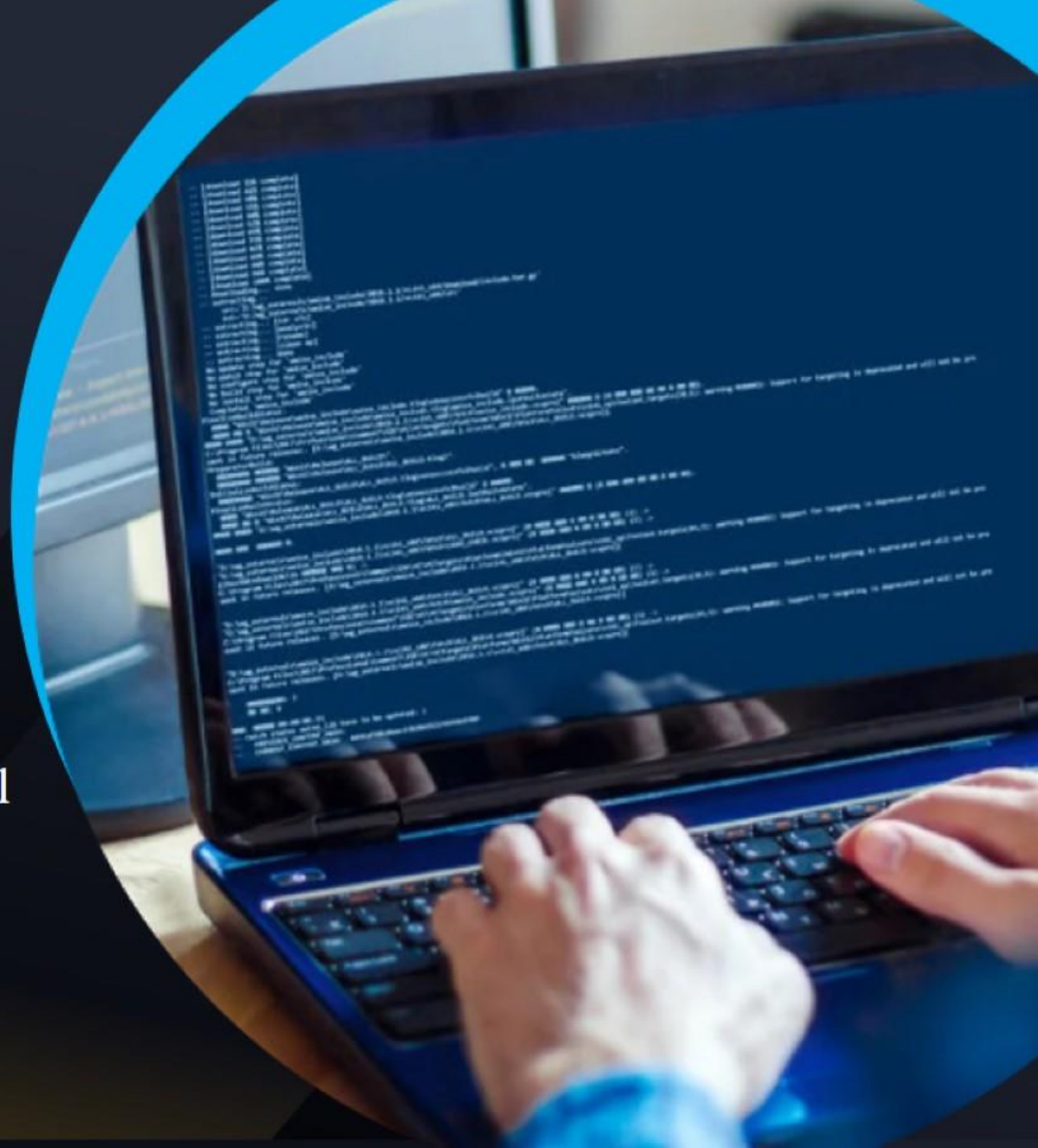


Programming Foundations in Python

Prepared by Dr. Hamada I. AbdulWakel

Lecture 9



Object-Oriented Programming II

Lecture 9

Prepared by
Dr. Hamada I. AbdulWakel

Lecture

09

RECAP: LECTURE 8

- Understanding the basics of Object-Oriented Programming.
- The building blocks and principles of OOP.
- Difference instance and class attributes.

Lecture

09

LECTURE MAIN POINTS

- Learning more about classes behaviors.
- The meaning of using the operator overloading.
- The meaning and uses of inheritance and polymorphism.

THE GETTER AND SETTER METHODS

- **getattr(obj, 'paramName')**: returns the current value of this parameter.
- **setattr(obj, 'paramName', value)**: allows you to set/change the value of an attribute.
- **hasattr(obj, 'Name')**: checks whether the object has this attribute or not.
- **isinstance(obj, className)**: returns True if the specified object is of the specified type, otherwise False .

THE GETTER AND SETTER METHODS

Cont'd

- Example:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
p1 = Point(1, 2)
print(getattr(p1, 'x'))
print(setattr(p1, 'y', 5))
print(hasattr(p1, 'z'))
print( isinstance(p1, Point) )
```

THE OPERATOR OVERLOADING

- Example:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
p1 = Point(1, 2)
p2 = Point(2, 3)
print(p1+p2)
```

THE OPERATOR OVERLOADING

Cont'd

- You can change the meaning of **any operator** in Python depending upon the operands used to be suitable to your class.
- For example, the `+` operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.
- But, if you perform it on two instances of a user-defined class it will give you a **TypeError** because these instances are unsupported operand type(s) for `+` operator.

THE OPERATOR OVERLOADING

Cont'd

- Among the special functions for operators are:
 - + by overloading `__add__(self, otherObj)` for **obj1 + obj2**
 - - by overloading `__sub__(self, otherObj)` for **obj1 - obj2**
 - * by overloading `__mul__(self, otherObj)` for **obj1 * obj2**
 - / by overloading `__div__(self, otherObj)` for **obj1 / obj2**
 - % by overloading `__mod__(self, otherObj)` for **obj1 % obj2**
 - ** by overloading `__pow__(self, otherObj)` for **obj1 ** obj2**

THE OPERATOR OVERLOADING

Cont'd

- Python does not limit operator overloading to arithmetic operators only. We can overload **comparison operators** as well.
- Among the special functions for these operators are:
 - < by overloading `__lt__(self, otherObj)` for `obj1 < obj2`
 - <= by overloading `__le__(self, otherObj)` for `obj1 <= obj2`
 - > by overloading `__gt__(self, otherObj)` for `obj1 > obj2`
 - >= by overloading `__ge__(self, otherObj)` for `obj1 >= obj2`
 - == by overloading `__eq__(self, otherObj)` for `obj1 == obj2`
 - != by overloading `__ne__(self, otherObj)` for `obj1 != obj2`

THE OPERATOR OVERLOADING

Cont'd

- Example:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x;    self.y = y
    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)
p1 = Point(1, 2);    p2 = Point(2, 3);
print(p1+p2)
```

THE OPERATOR OVERLOADING

Cont'd

- Example:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x;    self.y = y
    def __lt__(self, other):
        selfCords = (self.x ** 2) + (self.y ** 2)
        otherCords = (other.x ** 2) + (other.y ** 2)
        return selfCords < otherCords

p1 = Point(1, 2);    p2 = Point(2, 3);
print(p1 < p2)
```

THE OPERATOR OVERLOADING

Cont'd

- Example:

```
class Fruit:
    def __init__(self, weight, name = "none"):
        self.name = name
        self.weight = weight
    def __add__(self, x):
        if isinstance(x, Fruit):
            return Fruit(self.weight + x.weight)
        if isinstance(x, int):
            return Fruit(self.weight + x, self.name)
```

THE OPERATOR OVERLOADING

Cont'd

- Example (con't):

```
class Fruit:
    def __eq__(self, x):
        if self.name == x.name:
            print("Both are the same fruits")
        else:
            print(self.name, "and", x.name, "are different fruits.")
    def __str__(self):
        return "Name: {0}, Weight: {1}\n".format(self.name,
            self.weight)
```

THE OPERATOR OVERLOADING

Cont'd

- Example (con't):

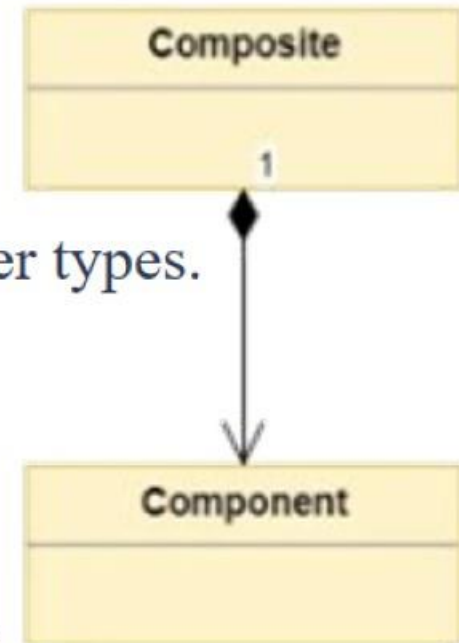
```
f1 = Fruit(5, "Strawberry")  
f2 = Fruit(100, "Watermelon")  
f3 = Fruit(20, "Mango")  
print( f1, f2, f3 )  
print( f1 + f2 + f3 )  
print( f1 == f2 )
```

CLASSES RELATIONSHIPS

- Classes relationships describe how classes and classes' objects interact with each other.
- There are two types of these relationships:
 - Inheritance.
 - Composition.
- Both enable **code reuse**, but they operate in **different ways**.

CLASS COMPOSITION

- **Composition:** is a concept that models a **has a** relationship.
- It enables creating complex types by combining objects of other types.
- This means that a class Composite can contain an object of another class Component.
- A composition relationship occurs if one class has an **instance attribute** which is an object from the other class (car has wheels).



CLASS COMPOSITION

Cont'd

- Example:

```
class Course:
    def __init__(self, name):
        self.cname = name;    self.hours = 3

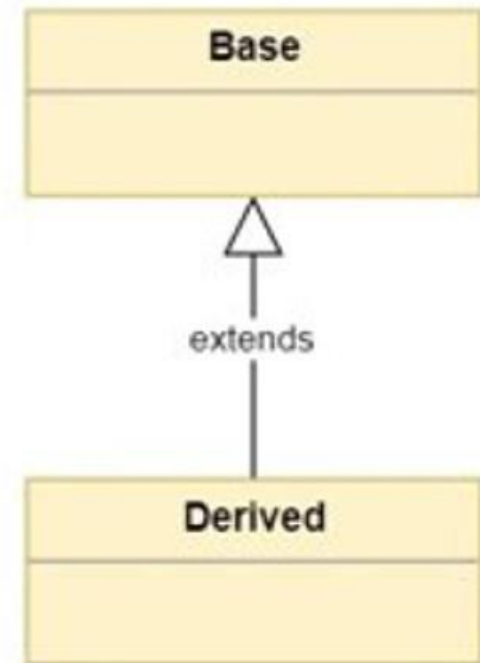
class Student:
    def __init__(self, name):
        self.course = Course("OOP")
        self.name = name

s = Student("ahmed")
print( s.course.cname )
```

INHERITANCE

- **Inheritance:** models what is called an **is a** relationship.
- A class can inherit **all** the methods and attributes of another class. These methods and attributes become available to the new class without the need to redefine them from scratch.
- A **derived class** that inherits from a **Base class**.

```
class DrivedClassName(BaseClassName):  
    # body
```



INHERITANCE EXAMPLE

- **Student Management System Case Study**



Student has:

- 1- Name
- 2- Address
- 3- NumofCourse
- 4- Hours
- 5- Grades []
- 6- getGrades()



Teacher has:

- 1- Name
- 2- Address
- 3- NumofCourse
- 4- Hours
- 5- PayRate
- 6- getSalary()

INHERITANCE EXAMPLE

Cont'd

DRY: Don't Repeat Yourself



Student has:

- 1- Grades []
- 2- getGrades()



Person has:

- 1- Name
- 2- Address
- 3- NumofCourse
- 4- Hours



Teacher has:

- 1- PayRate
- 2- getSalary()

INHERITANCE EXAMPLE

Cont'd

```
class Person:
    def __init__(self, name, address, numofcourses, hours):
        self.name = name
        self.address = address
        self.numofcourses = numofcourses
        self.hours = hours

    def getPersonInfo(self):
        print("The person name is ", self.name)
        print("The person address is ", self.address)
```

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

```
ahmed = Student("Ahmed", "1 XYZ street ", 3, 9)
ahmed.getPersonInfo()
```

INHERITANCE EXAMPLE

Cont'd

```
class Person:
    def __init__(self, name, address, numofcourses, hours):
        self.name = name
        self.address = address
        self.numofcourses = numofcourses
        self.hours = hours

    def getPersonInfo(self):
        print("The person name is ", self.name)
        print("The person address is ", self.address)
```

```
class Student(Person):
    def getGrades(self):
        print("The student grades are ", [35, 59, 80])
```

More behaviors
than parent !!

```
ahmed = Student("Ahmed", "1 XYZ street ", 3, 9)
ahmed.getPersonInfo()
ahmed.getGrades()
```

INHERITANCE EXAMPLE

Cont'd

- Example:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return f"The point is located in ({self.x}, {self.y})"
    def displayX(self):
        print("The point x coordinate = ", self.x)
```

INHERITANCE EXAMPLE

Cont'd

- Example (con't):

```
class Circle(Point):  
    def __init__(self, r, x, y):  
        Point.__init__(self, x, y)  
        self.radius = r  
    def __str__(self):  
        print(Point.__str__(self))  
        return f"The circle radius = {self.radius}"
```

INHERITANCE EXAMPLE

Cont'd

- Example (con't):

```
c1 = Circle(2.5, 2, 3)
print(c1)
c1.displayX()
print( isinstance(c1,Circle) )
issubclass( Circle, Point )
```

Lecture

09

POLYMORPHISM

- Polymorphism is taken from the Greek words **Poly** (many) and **morphism** (forms).
- it means that the methods with **the same name** can be executed on **many** objects or classes.
- There are two ways to perform polymorphism:
 - Using functions.
 - Using class methods.

Lecture

09

FUNCTION POLYMORPHISM

- You need to define a function that can **take any object**, for which the methods' calling in the function body will be applied.
- Previously, there are some functions in Python which are **compatible** to run with multiple data types.

```
L1 = [ 1, 2, 3, 4]
```

```
T1 = (1, 2, 3, 4)
```

```
print( len( T1 ) )
```

```
print( len( L1 ) )
```

FUNCTION POLYMORPHISM

Cont'd

- Example:

```
class Student:
    def workplace(self):
        print("At Midocean University")

class Employee:
    def workplace(self):
        print("In free-business or at an international company")
```

FUNCTION POLYMORPHISM

Cont'd

- Example (con't):

```
def applyPolymorphism( obj ):  
    obj.workplace()
```

```
Std1 = Student()
```

```
Emp1 = Employee()
```

```
applyPolymorphism(Std1)
```

```
applyPolymorphism(Emp1)
```

Lecture

09

CLASS POLYMORPHISM

- Polymorphism defines methods in **the child class** that have the same name as the methods in the parent class.
- The inherited child class method will modify the parent's implementation with an implementation that suitable for the child class.
- This process of re-implementing a method in the child class is known as **Method Overriding**.

CLASS POLYMORPHISM

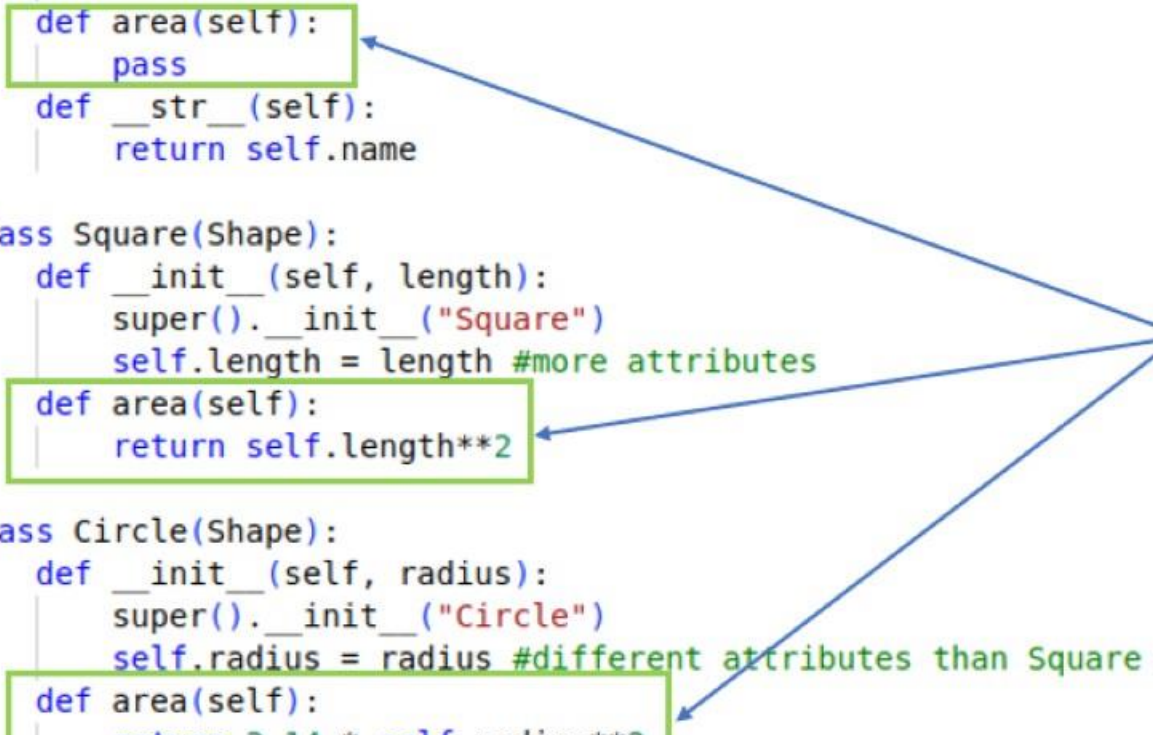
Cont'd

```
class Shape:
    def __init__(self, name):
        self.name = name
    def area(self):
        pass
    def __str__(self):
        return self.name
```

```
class Square(Shape):
    def __init__(self, length):
        super().__init__("Square")
        self.length = length #more attributes
    def area(self):
        return self.length**2
```

```
class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius #different attributes than Square
    def area(self):
        return 3.14 * self.radius**2
```

Same method with
different code !!!



CLASS POLYMORPHISM

Cont'd

- Example:

```
class Bird:
    def flight(self):
        print("Most of the birds can fly but some cannot")
class parrot(Bird):
    def flight(self):
        print("Parrots can fly")
class penguin(Bird):
    def flight(self):
        print("Penguins do not fly")
```

CLASS POLYMORPHISM

Cont'd

- Example:

```
obj_bird = Bird()  
obj_parr = parrot()  
obj_peng = penguin()
```

```
obj_bird.flight()  
obj_parr.flight()  
obj_peng.flight()
```

Lecture

09

QUESTIONS FOR ONLINE DISCUSSION

- Write code to create a **Vehicle** class with `max_speed` and `mileage` instance attributes, and add methods as you can in this class. Then, create a **Bus** class that inherits from the **Vehicle** class. Give the `capacity` argument a default value of 50 and define `seating_capacity()` inside **Bus** to modify the `capacity` value.



Thank You

