Programming 2

Dries Decuyper
Serhat Erdogan
Janne Gilis
Bart Thumas

Operator Overloading

Static Methods

Inheritance

PBA Toegepaste Informatica

2023-2024

UCLL
HOGESCHOOL

AGENDA

- Operator Overloading
- Static Methods
- Inheritance

Operator Overloading

- Operator
- Dunder Methods
- (__str__)

OPERATOR

- Arithmetic

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x – y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor Division	x // y

- Assignment

Operator	Example	Same as
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
…	…	…

OPERATOR

- Comparison

Operator	Name	Example
==	Equal	x == y
>=	Greater than of equal to	x >= y
…	…	…

- Logical

Operator	Name	Example
and	Returns True if both statements are true	x < 5 and x < 10
…	…	…

- Identity

Operator	Name	Example
is	Returns True if both variables are the same object	x is y
…	…	…

OPERATOR

- Membership

Operator	Name	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
…	…	…

OPERATOR

```
5 + 3 # Adding numbers
a" + "b" # Adding strings
[1, 2] + [3, 4] # Adding lists
```

- 5 + 3: Mathematical addition of the numbers 5 and 3
- "a" + "b": Concatenating the strings "a" and "b" to form the new string "ab“
- [1, 2] + [3, 4]: Combining the lists [1, 2] and [3, 4] through list concatenation to produce the result [1, 2, 3, 4]

DUNDER METHODS

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

# Example usage:
vector1 = Vector(1, 2)
vector2 = Vector(3, 4)

result = vector1 + vector2
print(result)  # Results in unsupported operand
```

- The program doesn’t know how to add two instances or objects of our class together

DUNDER METHODS

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        else:
            raise TypeError("Unsupported operand type for +: " + str(type(other)))

    # …

# Example usage:
vector1 = Vector(1, 2)
vector2 = Vector(3, 4)

result = vector1 + vector2
print(result)  # Output: Vector(4, 6)
```

DUNDER METHODS

```
class Interval:
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def __add__(self, other):
        if isinstance(other, Interval):
            merged_start = min(self.start, other.start)
            merged_end = max(self.end, other.end)
            return Interval(merged_start, merged_end)
        else:
            raise TypeError("Unsupported operand type for +: " + str(type(other)))
```

- A "dunder" method, short for "double underscore" method, refers to special methods in Python that have double underscores at the beginning and end of their names
- The __add__ method, for example, is a dunder method that defines how an object should behave when the + operator is used with it
- Dunder methods provide a way to define behaviors for various operations on objects, making classes more powerful and flexible.

DUNDER METHODS

Operator	Method
+	__add__
-	__sub__
*	__mul__
/	__truediv__
//	__floordiv__
%	__mod__
**	__pow__

DUNDER METHODS

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        else:
            raise TypeError("Unsupported operand type for +: " +
str(type(other)))

    def __sub__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x - other.x, self.y - other.y)
        else:
            raise TypeError("Unsupported operand type for +: " +
str(type(other)))

    def __mul__(self, multiplier):
        return Vector(self.x * multiplier, self.y * multiplier)

    #...
```

```
# Example usage:
vector1 = Vector(1, 2)
vector2 = Vector(3, 4)

result1 = vector1 + vector2
print(result1)  # Output: Vector(4, 6)
result2 = vector2 - vector1
print(result2)  # Output: Vector(2, 2)
result3 = vector1 * 5
print(result3)  # Output: Vector(5, 10)
```

__STR__

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        else:
            raise TypeError("Unsupported operand type for +: " +
str(type(other)))

    def __str__(self):
        return f"Vector ({self.x}, {self.y})"
```

```
# Example usage:
vector1 = Vector(1, 2)
vector2 = Vector(3, 4)

result1 = vector1 + vector2
print(result1)  # Output: Vector(4, 6)
```

- The __str__() method returns a human-readable, or informal, string representation of an object. This method is called by the built-in print(), str(), and format() functions
- toString() method in Java

EXERCISE

- Try the following exercises

- 02-OO
  > 04-operator-overloading
    > 01-money

Static Methods

- Static
- Factory

STATIC

```python
class FootballTeam:
    def __init__(self, name, points):
        self.name = name
        self.points = points

    @staticmethod
    def compare_teams(team1, team2):
        return team1.points > team2.points
```

```python
# Example usage:
team1 = FootballTeam("Team A", 15)
team2 = FootballTeam("Team B", 12)

result = FootballTeam.compare_teams(team1, team2)
print(result)  # Output: True (because Team A has more points than Team B)
```

- **@staticmethod** decorator, annotation
- We call a static method directly on the class instead of creating an object first
- Also notice that a static method is missing the self parameter: self is meant to refer to the object a method is called on, but in this case, there is no object, so having a self parameter would make no sense

FACTORY

```python
class Shape:
    def draw(self):
        pass

class Circle(Shape):
    def draw(self):
        return "Drawing a Circle"

class Rectangle(Shape):
    def draw(self):
        return "Drawing a Rectangle"

class ShapeFactory:
    @staticmethod
    def create_shape(shape_type):
        if shape_type == "Circle":
            return Circle()
        elif shape_type == "Rectangle":
            return Rectangle()
        else:
            raise ValueError("Invalid shape type")
```

```python
# Example usage:
circle =
ShapeFactory.create_shape("Circle")
rectangle =
ShapeFactory.create_shape("Rectangle")

print(circle.draw())
# Output: Drawing a Circle
print(rectangle.draw())
# Output: Drawing a Rectangle
```

- A factory is a design pattern that provides an interface for creating objects but allows subclasses to alter the type of objects that will be created

FACTORY

- **Enhanced Flexibility:** Factories offer greater flexibility in object creation compared to constructors, as they can encapsulate complex instantiation logic
- **Decoupling:** Helps in decoupling client code from specific class implementations, promoting loose coupling and easier maintenance
- **Single Responsibility:** Separates the responsibility of creating an object from the responsibility of using the object, adhering to the Single Responsibility Principle

EXERCISE

- Try the following exercises

- 02-OO
  > 05-static-methods

Inheritance

- Inheritance
- Super

INHERITANCE

```
class Circle:
    def __init__(self, color, radius):
        self.color = color
        self.radius = radius

    def area(self):
        return 3.14 * self.radius**2

    def describe(self):
        return f"This is a {self.color} circle
with radius {self.radius}."
```

```
class Square:
    def __init__(self, color, side_length):
        self.color = color
        self.side_length = side_length

    def area(self):
        return self.side_length**2

    def describe(self):
        return f"This is a {self.color} square with
side length {self.side_length}."
```

- Inheritance in Python provides a mechanism for code reuse and structuring code hierarchies by allowing a subclass to inherit attributes and methods from a superclass
- It promotes code organization, reduces redundancy, and supports the creation of specialized classes that inherit and extend the functionality of a more general base class

INHERITANCE

```
class Shape:
    def __init__(self, color):
        self.color = color

    def describe(self):
        return f"This is a
{self.color} shape."
```

```
class Circle(Shape):
    def __init__(self, color, radius):
        super().__init__(color)
        self.radius = radius

    def area(self):
        return 3.14 * self.radius**2

    def describe(self):
        return f"This is a {self.color} circle
with radius {self.radius}."
```

```
class Square(Shape):
    def __init__(self, color, side_length):
        super().__init__(color)
        self.side_length = side_length

    def area(self):
        return self.side_length**2

    def describe(self):
        return f"This is a {self.color} square with
side length {self.side_length}."
```

- We can use inheritance by putting the parent class between parentheses after the child class
- Same as extends in Java

SUPER

```
class Circle(Shape):
    def __init__(self, color, radius):
        super().__init__(color)
        self.radius = radius

    def area(self):
        return 3.14 * self.radius**2

    def describe(self):
        return f"This is a {self.color} circle 
with radius {self.radius}."
```

```
class Shape:
    def __init__(self, color):
        self.color = color

    def describe(self):
        return f"This is a 
{self.color} shape."
```

- **super()** in Python is a built-in function used within a subclass to call a method or access an attribute from its superclass, facilitating code reuse and maintaining the inheritance hierarchy.

EXERCISE

- Try the following exercises

- 02-OO
  > 05-inheritance
    > 01-human