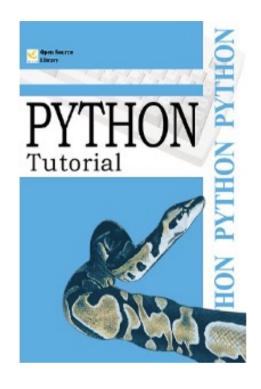
# Operators Overloading



Dra. Ma Dolores Rodríguez Moreno





## Objectives

#### **Specific Objectives**

- Understanding what is Operator Overloading
- Main Magic Methods for Operator Overloading

#### **Source**

- <a href="https://docs.python.org/3/reference/">https://docs.python.org/3/reference/</a>
- https://ellibrodepython.com/
- Python Tutorial Tapa blanda. GuidoVan Rossum (2012)





### Outline

- Introduction
- Magic Methods
- Best Practices



#### Introduction

- Customizing how operators work with user-defined classes
- Allows objects to behave like built-in types
- Makes code more intuitive and readable
- Operator overloading allows operators to have different meanings based on their operands
- Implemented using special methods (magic methods)
- For example:
  - `+` for addition of numbers, concatenation of strings, or merging of lists



### Outline

- Introduction
- Magic Methods
- Best Practices



## Magic Methods

- Arithmetic Operators
  - \_\_add\_\_(self, other) for `+`
  - \_\_sub\_\_(self, other) for `-`
  - \_\_mul\_\_(self, other) for `\*`
  - \_\_truediv\_\_(self, other) for `/`
- Comparison Operators
  - \_\_eq\_\_(self, other) for `==`
  - \_\_lt\_\_(self, other) for `<`
  - <u>\_gt\_(self, other) for</u> `>`
  - <u>\_\_le\_\_(self, other) for `<=`</u>
- Indexation: \_\_getitem\_\_(self, other) for `[]`





## Example: Añadir función sumar a la clase

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





## Example: Add two Vector

```
Example: +
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def sumar(self, other):
        return Vector(self.x + other.x, self.y + other.y)

v1 = Vector(2, 3)
v2 = Vector(4, 5)
v3 = v1.sumar(v2)
```



```
Example: +
class Vector:
    def init (self, x, y):
        self.x = x
        self.y = y
    def add (self, other):
        return Vector(self.x + other.x, self.y + other.y)
v1 = Vector(2, 3)
v2 = Vector(4, 5)
v3 = v1. add (v2) #Explicit call
```





```
Example: +
class Vector:
    def init (self, x, y):
        self.x = x
        self.y = y
    def add (self, other):
        return Vector(self.x + other.x, self.y + other.y)
v1 = Vector(2, 3)
v2 = Vector(4, 5)
v3 = v1 + v2 \# Output: < main .Vector object at 0x7ee28d312830>
```



### Example: str

```
Example: str
```

```
class Vector:
   def init (self, x, y):
       self.x = x
       self.y = y
   def add (self, other):
       return Vector(self.x + other.x, self.y + other.y)
   def str (self):
       return f"({self.x}, {self.y})" # Formato deseado
       #return "(" + str(self.x) + "," + str(self.y) + ")"
```



```
Example: + & str

v1 = Vector(2, 3)

v2 = Vector(4, 5)

v3 = v1 + v2

print(v3) # Output: (6, 8)
```





#### Example: + with different types

```
v1 = Vector(2, 3)
v2 = Vector(4, 5)
#What happen if we add a number?
v3 = v1 + 4
```



## Example: \_\_add\_\_ with different types

# Example: add a number

```
class Vector:
    def add (self, other):
        if isinstance (other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        elif isinstance(other, (int, float)):
            return Vector(self.x + other, self.y + other)
        else:
            print("Unsupported operand type")
```





## Example: commutative `+` Operator

#### Example: + with different types

```
v1 = Vector(2, 3)

v2 = Vector(4, 5)

#What happen if we add a number?

v5 = 4 + v2 # ERROR: unsupported operand type(s) for +: 'int' and 'Vector'
```





## Example: \_\_radd\_\_

- Used to handle cases where the + operator appears with the object to the right of the operator, such as  $4 + v_2$
- Python first tries int.\_\_add\_\_, and if this fails, it calls Vector.\_\_radd\_\_.

```
Example: add a number in both sides of +
class Vector:
...

def __radd__(self, other):
    return self.__add__(other)

v5= v2.__radd__(4)
```





### Outline

- Introduction
- Magic Methods
- Best Practices





#### **Best Practices**

- Follow Python's operator semantics (e.g., + should add, not subtract)
- Implement related operators together (if you implement <, implement >)
- Return *NotImplemented* for unsupported operand types
- Maintain consistency with built-in types
- Document the behavior of overloaded operators
- Consider implementing reverse (conmutative) operations (radd, rsub, etc.)



## Best Practices: example

• Return NotImplemented for unsupported operand types

```
Example: NotImplemented

def __add__(self, other):
    if isinstance(other, Vector):
        return Vector(self.x + other.x, self.y + other.y)
    return NotImplemented
```





### Practical Exercises

- Create a Rectangle class and overload the \* operator to multiply areas: use \_\_mul\_\_
- 2. Try the code:

```
rect1 = Rectangle(4, 5) #area: 20
rect2 = Rectangle(3, 6) #área: 18
result = rect1 * rect2 # Multiply the áreas of both rectangles
print(result) #The vaule is: 360
```

