

Special Class Methods

**SPECIAL
BROADCAST**

Essentially Python's version of Special Forces

What are "Special Methods":

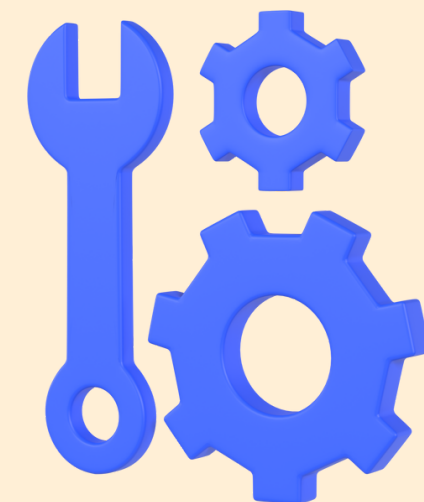
Special Methods are **automatically ran** when an object is created.

This is the **same thing** that happens with `__init__` as it's **automatically called** when an object is created.

Double-underscore methods make **operator overloading** possible.

This **gives extended meaning** beyond their predefined operational meaning

Special method \longrightarrow `__init__(self,)`



List of "Special Methods":

Syntax	What they do
<code>__init__(self,)</code>	Initializes -> Start. Used to build variables for the object
<code>__del__(self)</code>	Has the ability to delete an object
<code>__str__(self)</code>	Returns the object representation in a string format
<code>__len__(self)</code>	Allows you to return the length of an Object
<code>__eq__(self)</code>	Allows you to compare two Objects
<code>__add__(self)</code>	Allows you to add two objects

`__str__()` method:

This method provides a string representation of an object.

It's called by the built-in `str()` function and is commonly used to define a human-readable string representation of the object

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"Name: {self.name}, Age: {self.age}"
```

```
person = Person( "John", 25 )
print( person )
```

We construct a string using the **name** and **age** properties. The string is formatted in a readable format that represents the Person's information.

`__str__()` method:

This method provides a **string representation of an object**

It's called by the built-in **str()** function and is commonly used to define a human-readable string representation of the object

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"Name: {self.name}, Age: {self.age}"
```

```
person = Person("John", 25)
print(person)
```

When we **create an instance of the Person** class, such as **person**, and then print it using **print(person)**, The **__str__()** method is automatically called.

It returns the string representation of the **person** object, which is then displayed in our Terminal

`__str__()` method:

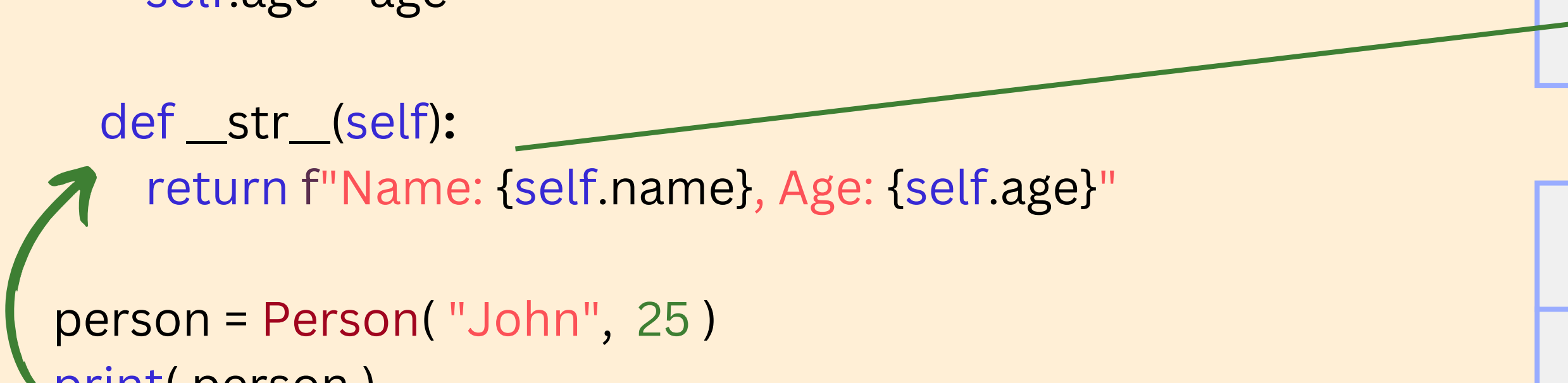
It's useful for providing a **concise representation of an object** when it is converted to a string.

Helps in debugging, logging, and **displaying object info in a human-readable format**

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"Name: {self.name}, Age: {self.age}"

person = Person("John", 25)
print(person)
```



Output in Terminal

Name: John, Age: 25

Without `__str__` method

<__main__.Person object
at 0x106378a00>

`__del__()` method:

Method that is called when an object is about to be destroyed. It's used to perform any final actions before the object is removed from memory

```
class File:
    def __init__(self, filename):
        self.filename = filename

    def __del__(self):
        print(f"Deleting file: {self.filename}")
```

```
file = File("data.txt")
del file ←
```

Useful for performing **cleanup** actions, like releasing resources or closing connections, **before an object is destroyed**

It can be **helpful** in situations where **manual cleanup** is necessary.

Use python's **del** statement on an object to use it

`__del__()` method:

Method that is called when an object is about to be destroyed. It's used to perform any final actions before the object is removed from memory

```
class File:
```

```
    def __init__(self, filename):
```

```
        self.filename = filename
```

```
    def __del__(self):
```

```
        print(f"Deleting file: {self.filename}")
```

```
file = File("data.txt")
```

```
del file
```

Output in Terminal

Deleting file: data.txt

`__add__()` method:

Special method that **allows objects of a class to be added together**

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

    def __add__(self, other):
        if isinstance(other, Vector):
            new_x = self.x + other.x
            new_y = self.y + other.y
            return Vector(new_x, new_y)
        else:
            raise TypeError("Unsupported type for +")
```

```
vector1 = Vector( 2, 3 )
vector2 = Vector( 4, 5 )
result = vector1 + vector2
print("New Object Values:", result.x, result.y)
```

Check if **other** is also an **object of the same class**

If **yes**, we **add** the corresponding **x** and **y** components of the two vectors

Creating a new Vector object with the result, and returning it

other -> python parameter that **represents another instance of this current class**

`__add__()` method:

Special method that **allows objects of a class to be added together**

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

    def __add__(self, other):
        if isinstance(other, Vector):
            new_x = self.x + other.x
            new_y = self.y + other.y
            return Vector(new_x, new_y)
        else:
            raise TypeError("Unsupported type for +")
```

```
vector1 = Vector( 2, 3 )
vector2 = Vector( 4, 5 )
result = vector1 + vector2
print("New Object Values:", result.x, result.y)
```

Output in Terminal

New Object Values: 6 8

`__sub__()` method:

Special method that **allows objects of a class to be subtracted together**

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

    def __sub__(self, other):
        if isinstance(other, Vector):
            new_x = self.x - other.x
            new_y = self.y - other.y
            return Vector(new_x, new_y)
        else:
            raise TypeError("Unsupported type for -")
```

```
vector1 = Vector(7, 4)
vector2 = Vector(5, 8)
result = vector1 - vector2
print("New Object Values:", result.x, result.y)
```

Exact opposite of `__add__`

Output in Terminal

New Object Values: 2 -4

Combining our Special Methods:

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

    def __sub__(self, other):
        if isinstance(other, Vector):
            new_x = self.x - other.x
            new_y = self.y - other.y
            return Vector(new_x, new_y)
        else:
            raise TypeError("Unsupported type for -")

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

vector1 = Vector(7, 4)
vector2 = Vector(5, 8)
res = vector1 - vector2
print(vector1)
print(res)
```

Output in Terminal

Object: 7, 4

Output in Terminal

Object: 2, -4

__eq__() method:

Special method that **allows us to check if two objects are equal**

```
class Rectangle:
```

```
    def __init__(self, width, height):
```

```
        self.width = width
```

```
        self.height = height
```

```
    def __eq__(self, other):
```

```
        if isinstance(other, Rectangle):
```

```
            return self.width == other.width and self.height == other.height
```

```
        else:
```

```
            return False
```

```
r1 = Rectangle(3, 4)
```

```
r2 = Rectangle(3, 4)
```

```
r3 = Rectangle(5, 2)
```

```
print(r1 == r2) # Output: True
```

```
print(r1 == r3) # Output: False
```

We can use the == operator to **compare two objects**.

Python **automatically calls** the __eq__ method

***Python will call the method on the object to its left

`__eq__()` method:

Special method that **allows us to check if two objects are equal**

```
class Rectangle:
```

```
    def __init__(self, width, height):  
        self.width = width  
        self.height = height
```

```
    def __eq__(self, other):  
        if isinstance(other, Rectangle):  
            return self.width == other.width and self.height == other.height  
        else:  
            return False
```

```
r1 = Rectangle(3, 4)  
r2 = Rectangle(3, 4)  
r3 = Rectangle(5, 2)
```

```
print(r1 == r2) # Output: True  
print(r1 == r3) # Output: False
```

Check if **other** is also an
object of the same class

If **yes**, check if objects have
same width/height

If **yes**, return **True**

`other` -> python parameter that **represents
another instance of this current class**



Can you implement these methods on your own?