Programming 2

Dries Decuyper
Serhat Erdogan
Janne Gilis
Bart Thumas

Classes
Access Control
Properties

GOALS

AGENDA

- Classes
- Access Control
- Properties

Classes

- Class
- Method
- Constructor
- Object

CLASS

```
class Vegetable:
    name = "Tomato"
    color = "Red"
    taste = "Sweet/Sour"

    # Methods…
```

- Classes are blueprints for creating objects, allowing us to model real-world entities and their behaviors in our code

METHOD

```
class Vegetable:

    name = "Tomato"
    color = "Red"
    taste = "Sweet/Sour"

    def describe_vegetable(self):
        print(f"I'm {self.name}, a {self.color} vegetable with a {self.taste} taste.")

    def cook(self, degrees):
        return f"I need to be boiled at {degrees} degrees"
```

- Functions defined within a class, representing actions or behaviors associated with the object
- **‘self’** in Python classes is like a mirror - it reflects the instance of the object, enabling access and manipulation of its attributes within the class methods. It's a way for the class to refer to itself

METHOD

```
class Vegetable:

    name = "Tomato"
    color = "Red"
    taste = "Sweet/Sour"

    def describe_vegetable(self):
        print(f"I'm {self.name}, a {self.color} vegetable with a {self.taste} taste.")

    def cook(self, degrees):
        return f"I need to be boiled at {degrees} degrees"
```

- The **return** statement is a powerful tool for enhancing the functionality of methods by providing meaningful output

CONSTRUCTOR

```
class Vegetable:

    name = "Tomato"
    color = "Red"
    taste = "Sweet/Sour"
```

```
class Vegetable:

    def __init__(self, name, color, taste):
        self.name = name
        self.color = color
        self.taste = taste

    #..
```

- It's quite rare in the real world to see a class that defines properties in the way we've been doing it
- It's much more practical to use a **constructor**

CONSTRUCTOR

```
class Vegetable:

    def __init__(self, name, color, taste):
        self.name = name
        self.color = color
        self.taste = taste

    #..
```

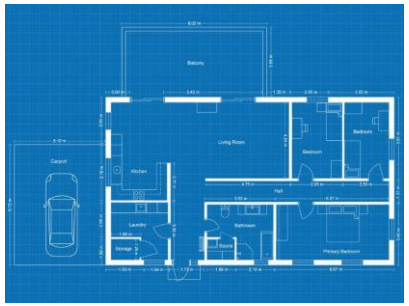
```
class Vegetable:

    def __init__(self, name, color, taste = "Sour"):
        self.name = name
        self.color = color
        self.taste = taste

    #..
```

```
class Vegetable:

    def __init__(self, name, color = "red", taste):
        self.name = name
        self.color = color
        self.taste = taste

    # Does not work
```

- Initializes the object with specified attributes
- Unlike some other programming languages, Python does not support multiple constructors with different parameter lists in the same class
- Workaround with default values. Must be last in the list of parameters

OBJECT

```
vegetable = Vegetable("Zesty Zucchini", "green", "zesty")
vegetable.color = "olive green"
vegetable.describe_vegetable()
```



- Think of it as a real-world entity that can perform actions and has characteristics, making code more modular and organized
- Objects are instances of classes, serving as the building blocks for structured and reusable code

EXERCISE

- Try the following exercises

- 02-OO
  > 01-classes
    > 01-class

    > 02-methods

    > 03-return-values

    > 05-constructor

    > 06-objects

Access Control

- Encapsulation

ENCAPSULATION

```
class Vegetable:

    def __init__(self, name, color, taste):
        self.__name = name
        self.color = color
        self.taste = taste

    #..
```

```
vegetable = Vegetable("Zesty Zucchini",
"green", "zesty")
print(vegetable.color) # Prints green
print(vegetable.__name) # Results in an error
```

- Encapsulation is the practice of hiding information inside a "black box" so that other developers working with the code don't have to worry about it
- And bundles data and methods, promoting cleaner code
- Reusability: Easily reuse classes across different parts of the program

EXERCISE

- Try the following exercises

- 02-OO
  > 02-access-control
    > 01-encapsulation

Properties

- Readonly
- Setter

READONLY

```
class Vegetable:

    def __init__(self, name, color, taste):
        self.__name = name
        self.color = color
        self.taste = taste

    @property
    def name(self):
        print("reading value")
        return self.__name
```

```
vegetable = Vegetable("Zesty Zucchini",
"green", "zesty")
print(vegetable.name) # prints reading value
# followed by the name of the vegetable
```

- As the term **readonly** states, can only be read and not changed
- **@property** decorator tells Python that whenever the user tries to read from an attribute, that it should call a method. Because of this, it is also called a getter method

SETTER

```
class Vegetable:

    def __init__(self, name, color, taste):
        self.name = name
        self.color = color
        self.taste = taste

    @property
    def name(self):
        print("reading value")
        return self.__name

    @name.setter
    def name(self, value):
        print("setting value")
        if value is None or len(value) is 0:
            raise ValueError('No valid name')
        self.__name = value
```

```
# Example 1
vegetable = Vegetable("Zesty Zucchini",
"green", "zesty")
vegetable.name = "" # Prints setting
value then raises the error

# Example 2
anotherVegetable = Vegetable(None,
"green", "zesty") # Prints setting
value then raises the error
```

- **@attribute.setter** tells Python that it's the attribute's setter method. The value parameter (you can choose whichever name you like, but as always, keep it descriptive) is the value the user is trying to assign
- Possibility to add extra intelligence or validation
- Don't forget to update the constructor to delegate to the attribute's setter

EXERCISE

- Try the following exercises

- 02-OO
  > 03-properties
    > 01-readonly

    > 02-setters