Object Oriented Programming

Using

Python

Teemu Matilainen

teemu.matilainen@savonia.fi

Lecture 3#

- **You are able to change attributes**
- **Youe are able to call class attributes**
- **You are able to create a dynamic method bound to an instance of a class**
- **You are able to create a static method**
- **You realize what data encapsulation means**
- **You realize what is an accessor (Getter, Setter)**
- **You realize what is an mutator (Getter, Setter)**
- **You realize what a mutator is used for**
- **You realize what an accessor is used for**
- **You realize what access control attributes are**
- **You are able to restrict visibility of attributes**
- **You are able to restrict visibility of methods**

Adding a constructor:

Here's an example of a simple **Bird** class with the constructor:

- In this example:
  - We define a class named **Bird**.
  - The class has a class attribute **species**.
  - The **____init____** method is used to initialize instance attributes (**name** and **color**).
  - There are two instance methods, **description** and **make_sound**.
  - We create two instances of the **Bird** class, **bird1** and **bird2**, and then access their attributes and call methods.

```
class Bird:
    species = "Unknown"

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

    def description(self):
        return f"{self.name} is a {self.color} bird"

    def make_sound(self, sound = "Chirp"):
        return f"{self.name } says {sound}"


bird1 = Bird(name="Robin", color="Red");
bird2 = Bird(name = "Sparrow", color = "Brown");


print(f"Bird name {bird1.name} is a {bird1.species}")
print(bird2.description())
print(bird1.make_sound())
```

Changing initial values of the data attributes:

You can add a method to the **Bird** class that allows you to change the initial values of the data attributes:

In this example, I added a **change_attributes** method to the **Bird** class, which takes new values for the **name** and **color** attributes and updates them accordingly. The example then demonstrates how to use this method to change the attributes of an existing **bird_instance**.

```python
class Bird:
    species = "Unknown"

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

    def description(self):
        return f"{self.name} is a {self.color} bird"

    def make_sound(self, sound = "Chirp"):
        return f"{self.name } says {sound}"

    def change_attribute(self, new_name, new_color):
        self.name = new_name
        self.color = new_color

bird1 = Bird(name="Robin", color="Red");
bird2 = Bird(name = "Sparrow", color = "Brown");

print(f"Bird name {bird1.name} is a {bird1.species}")
print(bird2.description())
print(bird1.make_sound())

bird1.change_attribute(new_name="updated bird", new_color="grey")
print(bird1.description())
```

```
Bird name Robin is a Unknown
Sparrow is a Brown bird
Robin says Chirp
updated bird is a grey bird
```

Class using different attribute types:

A **Bird** class with a birth date and a list of baby birds as parameters in the constructor:

```
Bird name Robin is a Unknown
Sparrow is a Brown bird and is born 2024-01-16
Robin says Chirp
Robin's babies
Sparrow is a Brown bird and is born 2024-01-16
```

```python
class Bird:
    species = "Unknown"

    def __init__(self, name, color, birthday, baby_birds=None):
        self.name = name
        self.color = color
        self.birthday = birthday
        self.baby_birds = baby_birds if baby_birds is not None else []

    def description(self):
        return f"{self.name} is a {self.color} bird and is born {self.birthday}"

    def make_sound(self, sound = "Chirp"):
        return f"{self.name } says {sound}"

    def change_attribute(self, new_name, new_color):
        self.name = new_name
        self.color = new_color

    def add_baby_bird(self, baby):
        self.baby_birds.append(baby)
```

```python
bird1 = Bird(name="Robin", birthday="2000-01-18", color="Red")
bird2 = Bird(name = "Sparrow",  birthday="2024-01-16", color = "Brown")

print(f"Bird name {bird1.name} is a {bird1.species}")
print(bird2.description())
print(bird1.make_sound())

#bird1.change_attribute(new_name="updated bird", new_color="grey")
#print(bird1.description())

bird1.add_baby_bird(bird2)

print(f"{bird1.name}'s babies")
for baby in bird1.baby_birds:
    print(baby.description())
```

```
class Bird:
    species = "Unknown"
```

Class attributes

In Python, you can access a **class attribute** from within the class by using the class name. Class attributes are shared among all instances of the class, and you can reference them using the **ClassName.attribute** syntax.

```
    def change_species(self, new_species):
        Bird.species = new_species
```

```
bird1.change_species("BAT")
print(f"Bird name {bird1.name} is a {bird1.species}")
```

```
Bird name Robin is a BAT
```

Local variables

- In a Python class like Bird, you can use **local variables** within methods to perform computations or store temporary values

```python
from datetime import datetime, date

class Bird:
    species = "Unknown"

    def __init__(self, name, color, birthday: date, baby_birds=None):
        self.name = name
        self.color = color
        self.birthday = birthday
        self.baby_birds = baby_birds if baby_birds is not None else []

    def description(self):
        return f"{self.name} is a {self.color} bird and is born {self.birthday}"

    def make_sound(self, sound = "Chirp"):
        return f"{self.name } says {sound}"

    def change_attribute(self, new_name, new_color):
        self.name = new_name
        self.color = new_color

    def add_baby_bird(self, baby):
        self.baby_birds.append(baby)

    def __str__(self):
        return f"{self.name} is a {self.color} bird and is born {self.birthday}"

    def __repr__(self):
        return f"{self.name} is a {self.color} bird and is born {self.birthday}"
```

```python
    def change_species(self, new_species):
        Bird.species = new_species

    def age(self):
        calculated_age = datetime.now() - self.birthday
        return calculated_age

bird1 = Bird(name="Robin", birthday=datetime.strptime("2000-01-18", "%Y-%M-%d"), color="Red")
bird2 = Bird(name = "Sparrow",  birthday=datetime.strptime("2024-01-16", "%Y-%M-%d"), color = "Brown")

print(f"Bird name {bird1.name} is a {bird1.species}")
print(bird2.description())
print(bird1.make_sound())

#bird1.change_attribute(new_name="updated bird", new_color="grey")
#print(bird1.description())

print(f"Age in days: {bird1.age().days} days")

bird1.add_baby_bird(bird2)
print(f"{bird1.name}'s \"babies\" ")
for baby in bird1.baby_birds:
    print(baby.description())
```

```
Bird name Robin is a Unknown
Sparrow is a Brown bird and is born 2024-01-16 00:01:00
Robin says Chirp
Age in days: 8773 days
Robin's "babies"
Sparrow is a Brown bird and is born 2024-01-16 00:01:00
```

In this:

- The **Bird** class now takes four parameters in the constructor: **name**, **color**, **birth_date**, and **baby_birds**. The **baby_birds** parameter is optional and defaults to an empty list if not provided.
- The **add_baby_bird** method is added to add a baby bird to the list of baby birds for a parent bird.

What is the **if baby_birds is not None else []??**

The **if baby_birds is not None else []** part in the constructor is a ternary conditional expression, also known as a **ternary operator**. It is a concise way to write an **if-else** statement in a single line. Here's how it works:

- **baby_birds if baby_birds is not None else []** checks if **baby_birds** is not **None**.
- If **baby_birds** is not **None**, it evaluates to **baby_birds**.
- If **baby_birds** is **None**, it evaluates to an empty list**[]**. So, in the context of the **Bird** class constructor:

```
self.baby_birds = baby_birds if baby_birds is not None else []
```

This line sets the **baby_birds** attribute of the **Bird** instance. If a value for **baby_birds** is provided during the object creation (**Bird(..., baby_birds=some_value)**), it will be used. Otherwise, it defaults to an empty list (**[]**). This is a way to handle the case where **baby_birds** might not be provided, ensuring that **self.baby_birds** is always a list.

How to print an object?

When you have created a class and defined it by yourself, the default thing would be calling the print command with that object. As you can see it’s not very informative:

```python
class Bird:
    species = "Unknown"

    def __init__(self, name, color, birthday, baby_birds=None):
        self.name = name
        self.color = color
        self.birthday = birthday
        self.baby_birds = baby_birds if baby_birds is not None else []

    def description(self):
        return f"{self.name} is a {self.color} bird and is born {self.birthday}"

    def make_sound(self, sound = "Chirp"):
        return f"{self.name } says {sound}"

    def change_attribute(self, new_name, new_color):
        self.name = new_name
        self.color = new_color

    def add_baby_bird(self, baby):
        self.baby_birds.append(baby)

bird1 = Bird(name="Robin", birthday="2000-01-18", color="Red")
bird2 = Bird(name = "Sparrow",  birthday="2024-01-16", color = "Brown")

print(bird1)
```

```
<__main__.Bird object at 0x0000012CF3DBE240>
```

__str__ method

- If we want to have more control over what is printed out. The easiest way to do this is to use special __str__ method to the class definition. Its purpose is to return a snapshot of the state of the object in string format!!! If the class definition contains a __str__ method, the value returned by the method is the one printed out when the print command is executed.

```
class Bird:
    species = "Unknown"

    def __init__(self, name, color, birthday, baby_birds=None):
        self.name = name
        self.color = color
        self.birthday = birthday
        self.baby_birds = baby_birds if baby_birds is not None else []

    def description(self):
        return f"{self.name} is a {self.color} bird and is born {self.birthday}"
```

...

```
    def __str__(self):
        return f"{self.name} is a {self.color} bird and is born {self.birthday}"

bird1 = Bird(name="Robin", birthday="2000-01-18", color="Red")
bird2 = Bird(name = "Sparrow",  birthday="2024-01-16", color = "Brown")

print(bird1)
```

```
Robin is a Red bird and is born 2000-01-18
```

- The __str__ method is more often used for formulating a string representation of the object with the str function.

```
bird1 = Bird(name="Robin", birthday="2000-01-18", color="Red")
bird2 = Bird(name = "Sparrow",  birthday="2024-01-16", color = "Brown")

string = str(bird1)
print(string)
```

__repr__ method

- There more special underscored methods which can be defined for classes. One rather similar is the __repr__ method. Its purpose is to provide a technical **representation** of the state of the object.

Note: Run the code in a Debugging mode and test the software without __str__ method!!!

```
    #def __str__(self):
    #    return f"{self.name} is a {self.color} bird and is born {self.birthday}"

    def __repr__(self):
        return f"{self.name} is a {self.color} bird and is born {self.birthday}"

bird1 = Bird(name="Robin", birthday="2000-01-18", color="Red")
bird2 = Bird(name = "Sparrow",  birthday="2024-01-16", color = "Brown")

string = repr(bird1)
print(string)

string = str(bird1)
print(str(bird1))
```

```
Robin is a Red bird and is born 2000-01-18
Robin is a Red bird and is born 2000-01-18
```

Helper methods

The class definition for MathOperations contains separate validator methods which ascertain that the arguments passed are valid.

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

    def subtract(self, x=None, y=None):
        if(x == None or y == None):
            return self.x - self.y
        elif(x != None and y != None):
            if self.is_number_ok(x, y) == True:
                return x - y
            else:
                return 0

    def is_number_ok(self, x, y):
        if isinstance(x, (int, float)) == True and isinstance(y, (int, float)) == True:
            return True
        else:
            return False

# Using static methods
math_object = MathOperations(4,6)

print("Subtract result:", math_object.subtract())
print(f"Subtract result: {math_object.subtract(4,6)}")
print(f"Subtract result: {math_object.subtract("4",6)}") # This gives an error if the nubm
```

Static methods

- **Static** method is a method that belongs to a class rather than an instance of the class. It is defined using the **@staticmethod** decorator and does not have access to the instance or its attributes. Static methods are similar to class methods, but they do not receive the class itself as the first parameter.

```python
class MathOperations:
    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
    def multiply(x, y):
        return x * y

# Using static methods
result_add = MathOperations.add(3, 5)
result_multiply = MathOperations.multiply(4, 6)

print("Addition result:", result_add)
print("Multiplication result:", result_multiply)
```

Key characteristics of static methods:

- They do not have access to the instance (self) or its attributes.
- They do not have access to the class itself (cls) as the first parameter (unlike class methods).
- They are defined using the **@staticmethod** decorator.
- They are called on the class rather than an instance of the class.
- Static methods are useful when a method is related to the class but does not require access to instance-specific data. They are often used for utility functions that are related to the class but don't involve manipulating instance attributes.

Yetunde

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

    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
    def multiply(x, y):
        return x * y

    def subtract(self, x=None, y=None):
        if(x == None or y == None):
            return self.x - self.y

        elif(x != None and y != None):
            return x - y

# Using static methods
math_object = MathOperations(4,6)

print("Addition result:", MathOperations.add(3, 5))
print("Multiplication result:", MathOperations.multiply(3, 5))
print(f"Multiplication result: {math_object.multiply(3,5)}")
print("Subtract result:", math_object.subtract())
print(f"Subtract result: {math_object.subtract(4,6)}")
#print(f"Subtract result: {MathOperations.subtract(4,6)}") # This gives an error!
```

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

    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
    def multiply(x, y):
        return x * y

    @staticmethod
    def is_number_ok(x, y):
        if isinstance(x, (int, float)) == True and isinstance(y, (int, float)) == True:
            return True
        else:
            return False

    def subtract(self, x=None, y=None):
        if(x == None or y == None):
            #if MathOperations.is_number_ok(x, y) == True: # this works
            if self.is_number_ok(x, y) == True:
                return self.x - self.y
            else:
                return 0
        elif(x != None and y != None):
            return x - y
```

Creating a dynamic method bound to an instance of a class

- How to create a dynamic method bound to an instance of a class by using the types module, specifically the **types.MethodType** class. This allows you to dynamically add a method to an instance of a class at runtime.

```python
import types

class Car:
    def __init__(self, make):
        self.make = make

def dynamic_method(self):
    print(f"This is a dynamic method for {self.make}")

# Create an instance of the Car class
my_car = Car("Toyota")

# Bind the dynamic method to the instance
my_car.dynamic_method = types.MethodType(dynamic_method, my_car)

# Call the dynamic method
my_car.dynamic_method()
```

```
This is a dynamic method for Toyota
```

Data encapsulation

Data encapsulation is one of the fundamental principles of object-oriented programming (OOP) and refers to the bundling of data (attributes) and methods (functions) that operate on that data into a single unit known as a class. Encapsulation helps in hiding the internal state of an object and restricting direct access to it from outside the class. The primary goals of encapsulation are data protection, code organization, and promoting a clean interface.

Here are key concepts related to data encapsulation:

- **Attributes (Data Members)**: These are the variables that store the state of an object. They are typically declared as **private** or **protected** within the class to restrict direct access from outside.

- **Methods (Member Functions)**: These are functions that operate on the attributes of the class. Methods are used to perform operations, modify the state of the object, or provide information about the object. They are defined within the class and can access the private attributes.

- **Access Modifiers**: In many programming languages (including Python, Java, and C++), access modifiers such as **public**, **private**, and **protected** are used to control the visibility of attributes and methods. For encapsulation, **it's common to make attributes private (accessible only within the class) and provide public methods to interact with them.**

- **Getters and Setters**: These are methods that are used to access (get) or modify (set) the values of private attributes. By using getters and setters, you can control how the data is accessed and modified, enforcing validation or business rules if needed.

In object-oriented programming, access modifiers determine the visibility of class members (attributes and methods) from outside the class. There are typically three main access modifiers: **public**, **private**, and **protected**. Here's a brief overview of each:

Public:

Members declared as public are accessible from anywhere, both within the class and from external code. There are no restrictions on accessing public members.

Private:

Members declared as private are only accessible from within the same class. External code cannot directly access or modify private members.

Protected:

Members declared as protected are accessible within the same class and its subclasses (derived classes or child classes). External code, which is not a subclass, typically cannot access or modify protected members.

In Python, the access modifiers **public, private, and protected** are conventions rather than strict rules, as Python is a language that emphasizes readability and simplicity. The convention is to use a **single underscore _ to indicate a protected** member, and a **double underscore __ to indicate a private member**.

It's important to note that Python does not enforce strict encapsulation or access control like some other languages. The use of a **single underscore** (_) is a convention to indicate that a variable or method should be treated as **protected**, and developers are expected to follow this convention. However, external code can still access protected members if needed. Private members (those with **double underscores** (__) are name-mangled to make them less accessible from outside the class, but they are not entirely inaccessible.

Here's a modified version of the Car class example with **private** attributes:

```python
class Car:
    def __init__(self, make, model):
        self.__make = make  # Private attribute
        self.__model = model  # Private attribute
        self.make2 = make
        self.model2 = model

    def get_make(self):
        return self.__make  # Getter method

    def set_make(self, make):
        if make:
            self.__make = make  # Setter method with validation

    def get_model(self):
        return self.__model  # Getter method

    def set_model(self, model):
        if model:
            self.__model = model  # Setter method with validation


car = Car("Toyota", "Corolla")
print(car.get_make())
print(car.make2)
```

```
Toyota
Toyota
```

Here's a modified version of the Car class example with **protected** attributes:

```python
class Car:
    def __init__(self, make, model):
        self._make = make  # Protected attribute
        self._model = model  # Protected attribute

    def get_make(self):
        return self._make  # Getter method

    def set_make(self, make):
        if make:
            self._make = make  # Setter method with validation

    def get_model(self):
        return self._model  # Getter method

    def set_model(self, model):
        if model:
            self._model = model  # Setter method with validation


car = Car("Toyota", "Corolla")
print("Car ", car.get_make(), car.get_model())
print("Car ", car._make, car._model)
```

```
Car  Toyota Corolla
Car  Toyota Corolla
```

Visibility of methods

It is controlled using the same conventions as attributes. There are no explicit access modifiers like public, private, or protected as in some other programming languages. Instead, Python relies on naming conventions to indicate the intended visibility of methods:

Public Methods:

Methods without any special naming convention are considered public and can be accessed from anywhere.
Example: def public_method(self):

Protected Methods:

Methods intended to be protected are prefixed with a single underscore _.
Example: def _protected_method(self):

Private Methods:

Methods intended to be private are prefixed with a double underscore __.
Example: def __private_method(self):

- Just like with attributes, these conventions are not enforced by the Python interpreter, and they are more about signaling the intended use of the methods to other developers. **It's still possible to call "protected" and "private" methods from outside the class.**

```python
class Car:
    def __init__(self, make, model):
        self._make = make  # Protected attribute
        self.__model = model  # Private attribute

    def get_make(self):
        return self._make

    def set_make(self, make):
        if make:
            self._make = make

    def _protected_method(self):
        print("This is a protected method.")

    def __private_method(self):
        print("This is a private method.")

    def public_method(self):
        print("This is a public method.")
        self._protected_method()
        self.__private_method()

# Example usage
car = Car("Toyota", "Corolla")
car.public_method()
car._protected_method()
car.__private_method()
```

```
This is a public method.
This is a protected method.
This is a private method.
This is a protected method.
```

The concept of a "**mutator**" method is not explicitly defined or recognized as a standard term. However, a mutator, in the context of object-oriented programming, is often associated with methods that modify the state of an object. These methods are sometimes referred to as **setter methods**.

A **setter method**, or **mutator** method, is responsible for modifying the value of an attribute. It provides a controlled way to update the internal state of an object. In Python, you can use a setter method to implement mutator functionality.

In this example:

- The **@property** decorator is used for the make method, indicating that it is a property (**getter**).
- The **@make.setter** decorator is used to define a **setter** method for the make property.
- The make property can be accessed and modified like an attribute, providing a clean interface for both getting and setting the attribute value.

```python
class Car:
    def __init__(self, make):
        self._make = make  # Private attribute

    @property
    def make(self):
        return self._make  # Getter method using @property

    @make.setter
    def make(self, new_make):
        if new_make:
            self._make = new_make  # Setter method (mutator) with validation

# Example usage
my_car = Car("Toyota")
print(my_car.make)  # Accessing the make attribute using the property

my_car.make = "Honda"  # Updating the make attribute using the setter (mutator)
print(my_car.make)  # Accessing the updated make attribute
```

```
Toyota
Honda
```

Assignment 5#

Implement a stopwatch class to represent a simple stopwatch that can measure elapsed time. Define all the attributes as a private. Test your code also with static methods!

```
sw = stopwatch()

i = 0
sw.start()
while i < 5:
    time.sleep(1)
    print(sw.elapsed_time())
    i += 1

sw.start()
time.sleep(5)  # Simulate some activity for 5 seconds while the stopwatch is running
print(sw.elapsed_time())
sw.reset()
print(sw.elapsed_time())  # Should print 0 seconds after reset
```

- **The class should have the following methods:**

- **__init__(self)**
- **Initializes the stopwatch with an initial state.**

- **start(self)**
- **Starts the stopwatch. If the stopwatch is already running, do nothing.**

- **stop(self)**
- **Stops the stopwatch. If the stopwatch is not running, do nothing.**

- **reset(self)**
- **Resets the stopwatch to its initial state.**

- **elapsed_time(self)**
- **Returns the elapsed time in seconds. If the stopwatch is running, it should return the time elapsed up to the current moment. If the stopwatch is stopped or reset, it should return the total elapsed time.**

Output example:

```
1.0126705169677734
2.0132105350494385
3.0141782760620117
4.014995098114014
5.016038417816162
10.016729354858398
0
```