

Object Oriented Programming  
Using  
Python

Teemu Matilainen  
[teemu.matilainen@savonia.fi](mailto:teemu.matilainen@savonia.fi)

## **Lecture 2#**

You realize what a class is

You are able to define a class

You realize what an object is

You are able to create instances of a class

You realize what an attribute is

You are able to create attributes into a class

You realize what a method is

## **Classes and objects**

In object-oriented programming (OOP), classes and objects are fundamental concepts. They provide a way to structure and organize code in a more modular and reusable manner. Let's explore the concepts of classes and objects in Python.

## Classes:

A class is a blueprint or a template (**abstract**: dog) for creating objects. It defines a set of attributes (data) and methods (functions) that the objects of the class will have. In Python, a class is created using the **class** keyword.

## Simple class:

Now we create a simple skeleton of a class. Class does not do anything... yet.

```
class Simple_class:  
    pass
```

The provided code indicates to Python that a class called `Simple_class` is being defined. Although the class currently lacks any functionality, we can still instantiate an object based on it.

Now, consider a program in which two variables, 'name' and 'age,' are added to a `Simple_class` object. Any variables associated with an object are referred to as its attributes, specifically, data attributes or, at times, instance variables.

These attributes linked to an object can be retrieved through the object

# Simple Class continues:

---

```
class Simple_class:
    pass

sc = Simple_class()
sc.name = "my first class"
sc.age = 1

print(sc.name)
print(sc.age)
print("...")
```

Output:

```
my first class
1
...
```

# In this example:

---

- **Dog** is a class with attributes **name** and **age**, and a method **bark**.
- The `__init__` method is a special method called the constructor, which initializes the object's attributes when an instance is created.
- **my\_dog** is an instance (object) of the **Dog** class.

```
class dog:
    def __init__(self, name: str, age: int ):
        self.name = name
        self.age = age

    def bark(self):
        print(f"Dog {self.name} says woof")

my_dog = dog(name = "Nino", age = 12)
my_dog.bark()
```

Exercise:

- Create a class cat with the name and age. Cat class has a method Miaou and parameter what makes the cat miaou as many time as the input argument defines.



# One solution:

---

```
class Cat:
    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    def miau(self, c):
        for i in range(c):
            print(f"Miau {i}")

cat = Cat("Jaakko", 5)
cat.miau(5)
```

Output

```
Miau 0
Miau 1
Miau 2
Miau 3
Miau 4
```



## Objects:

An object is an instance of a class. It is a concrete realization of the class, with its own unique state (attribute values). Objects can perform actions (methods) defined by the class. In the example above, **my\_dog** is an object of the **Dog** class.

In Python, an object is an instance of a class, and a class is a blueprint for creating objects. Objects can have associated methods, which are functions that are defined within the class and can operate on the object's attributes.

Methods are essentially functions that are bound to the object and can access and modify its state.

## **Instantiation:**

The process of creating an object from a class is called instantiation. In the example, `my_dog = Dog(name='Nino', age=3)` is an instantiation of the **Dog** class.

## **Encapsulation:**

Encapsulation is the bundling of data and methods that operate on the data within a single unit, i.e., a class. The attributes and methods of a class are encapsulated within that class, providing a way to control access and modification.

## **Inheritance:**

Inheritance is a mechanism where a new class (subclass or derived class) can inherit attributes and methods from an existing class (base class or parent class). It promotes code reuse and extensibility.

## **Polymorphism:**

Polymorphism allows objects of different classes to be treated as objects of a common base class. It enables flexibility and extensibility in code.

These concepts collectively form the foundation of object-oriented programming and are widely used in Python and many other programming languages.

---

- **Compile-Time Polymorphism (Static Binding):**

- Achieved through method overloading and operator overloading.
- Python does not support method overloading in the same way as some statically-typed languages like Java or C++, but you can achieve similar behavior using default argument values or variable-length argument lists.
- Example of method overloading in Python (using default argument values):

```
class Cat:
    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    def miau(self, d, a=None, b=None, c=None):
        for i in range(d):
            if(a != None and b != None and c != None):
                print(f"Miau Loud {i}")
            elif(a != None and b != None):
                print(f"Miau Normal {i}")
            elif(a != None):
                print(f"Miau Silent {i}")
            else:
                print(f"Miau {i}")

cat = Cat("Jaakko", 5)
cat.miau(5)
cat.miau(5,1)
cat.miau(5,1,1)
cat.miau(5,1,1,1)
```

# Encapsulation example:

---

- In this example, **Dog** is a class with two methods: **bark** and **celebrate\_birthday**. The
- **\_\_init\_\_** method is a special method called the constructor, which is executed when a new instance of the class is created.
- The **bark** method is a simple method that prints a message using the **name**
- attribute of the **Dog** object.
- The **celebrate\_birthday** method increments the **age** attribute by 1 and prints a message.

```
class dog:
    def __init__(self, name: str, age: int ):
        self.name = name
        self.age = age

    def bark(self):
        print(f"Dog {self.name} says woof")

    def celebrate_birthday(self):
        self.age += 1
        print(f"{self.name} is now {self.age} years old")

my_dog = dog(name = "Nino", age = 12)
my_dog.bark()
my_dog.celebrate_birthday()
```

When we create an instance of the **Dog** class (**my\_dog**), we can call its methods and access its attributes. The methods operate on the object's state, and each method has access to the object's attributes through the **self** parameter.

Understanding objects and methods is essential in object-oriented programming, as it allows you to model and encapsulate behavior within classes, leading to more organized and modular code.

# Function to Display Object Information:

---

- Functions that work with objects in Python can interact with the attributes and methods of those objects. Here are some examples of functions that work with objects:


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

def display_person_info(p: Person):
    print(f"Name: {p.name}, Age: {p.age}")

me = Person(name = "Humpty Dumpty", age = 1)
display_person_info(me)

brother = Person("Little brother", 0.5)
display_person_info(brother)
```

```
Name: Humpty Dumpty, Age: 1
Name: Little brother, Age: 0.5
```



# Function to Calculate Area of a Rectangle Object:

---

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def calculate_area(p):
        return p.length*p.width

r = Rectangle(20, 30)
area = calculate_area(r)
print(f"Area of the rectangle: {area}")
```

# Function to Compare Objects:

---

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

def compare(p1: Point, p2: Point):
    return p1.x == p2.x and p1.y == p2.y

p1 = Point(20, 31)
p2 = Point(20, 30)

if compare(p1, p2) == True:
    print(f"Points are equal!")
else:
    print(f"Points are not equal!")
```



Later in the program you can still change the initial values of the data attributes:

---

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

def compare(p1: Point, p2: Point):
    return p1.x == p2.x and p1.y == p2.y

p1 = Point(20, 31)
p2 = Point(20, 30)

if compare(p1, p2) == True:
    print(f"Points are equal!")
else:
    print(f"Points are not equal!")

p1.x = 20
p1.y = 30

if compare(p1, p2) == True:
    print(f"Points are equal!")
else:
    print(f"Points are not equal!")
```

These examples illustrate how functions can take objects as parameters, access their attributes, and invoke their methods to perform various actions. This is a common pattern in object-oriented programming, promoting modularity and code organization.

Working with an object of type e.g. `date` you may have noticed that there is a slight difference between how the variables contained in the object are accessed, as opposed to how the methods attached to the objects are used:

```
import datetime

m_today = datetime.datetime(2024, 1, 11)

# method
weekday = m_today.isoweekday()

# variable
my_month = m_today.month

print("the date:", weekday)
print("The month:", my_month)
```

---

- **Assignment 3#:**

- Write a function named `list_all_years(dates: list)` which takes a list of date type objects as its argument. The function should return a new list, which contains the *years in the original list in chronological order*, from earliest to latest.

- An example of the function in action:

```
from datetime import date
```

```
d1 = date(2018, 2, 1)
d2 = date(2023, 10, 20)
d3 = date(1988, 1, 7)

years = list_all_years([d1, d2, d3])
print(years)
```

```
[1988, 2018, 2023]
```

---

- **Assignment 4#:**

- This **ShoppingList** class has methods for adding items, removing items, getting the count of unique items, getting the total units, and displaying the current shopping list etc.

```
# Example usage:
shopping_list = ShoppingList()

shopping_list.add_item("Apple", 3)
shopping_list.add_item("Banana", 2)
shopping_list.add_item("Orange", 4)
shopping_list.display_list()

print(f"Total unique items: {shopping_list.item_count()}")
print(f"Total units: {shopping_list.unit_count()}")

shopping_list.remove_item("Banana", 1)
shopping_list.display_list()

#Display one item (Banana) by using the index
print(shopping_list.item(1))
```

```
Total unique items: 3
Total units: 9
Shopping List:
- Apple: 3
- Banana: 1
- Orange: 4
Banana
```

---

- Create a ShoppingList class which has several methods item\_count, add\_item, unit\_count etc.

- Here is partially created shopping list class. Fill the gaps...

```
class ShoppingList:
    def __init__(self):
        self.items = {} # Dictionary to store items and their quantities

    # Get the item name by using the index value
    def item(self, i: int):
        if len(self.items) > i:
            ???
            return self.items[i]

    # Add an item to the shopping list.
    def add_item(self, item, quantity=1):
        if item in ???

    # Remove a specified quantity of an item from the shopping list.
    def remove_item(self, item, quantity=1):
        if item in self.items:
            ???

    # Get the total count of unique items on the shopping list.
    def item_count(self):
        return ???

    # Get the total count of all units (quantities) of items on the shopping list.
    def unit_count(self):
        return sum(???)

    # Display the current shopping list.
    def display_list(self):
        print("Shopping List:")
        ???
```