OOP - Introduction

Instructors **Battista Biggio**, **Angelo Sotgiu** and **Leonardo Regano**

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence
University of Cagliari, Italy

With the `time slot` exercise we have taken a step in the direction of modularization.

We have defined a public interface that allows you to manipulate the data without knowing its internal structure.

In this section we introduce **Object-Oriented Programming (OOP)**[1]

- as a way to improve **Information Hiding** $\Longrightarrow$ **Abstraction + Encapsulation**
- as a method to improve the representation of (concrete or abstract) entities of the real world that we want to manipulate

---

[1] A complete discussion of OOP is beyond the scope of this course.

One of the limitations of the `time slot` solution: **data types** and **functions that work on the data types** are separate.

```
t1 = create_time_slot()   # dictionary
set_time_slot_h_m(t1, 2, 20)
print(get_time_slot_m(t1))
```

Consider a scenario where we manage data associated with **angles**.
Data structures and functions can be established in a similar fashion.

```
a1 = create_angle()   # dictionary
set_angle(a1, 2, 20)
print(get_angle(a1))
```

The semantics of the functions `create()`, `set()`, `get()` are quite similar for both angles and times.
However, it is essential to select distinct names based on the data type to which they are applied, *i.e.*,

`create_angle()`, `create_time()`

`set_angle()`, `set_time()`

`get_angle()`, `get_time()`

For better code comprehension and readability, it would be more helpful to have data types with variables that can be manipulated in this manner:

(OOP syntax)

```
t1 = TimeSlot()
a1 = Angle()

t1.set(2, 20)
a1.set(1, 10)
```

(instead of)

```
t1 = create_time_slot()
a1 = create_angle()
set_time_slot_h_m(t1, 2, 20)
set_angle(a1, 1, 10)
```

Comprehension and readability ensure the reduction of errors and facilitating easier code modifications, even by other programmers.

Advantages:

- It's significantly clearer **which data** is being manipulated and which data is merely a 'parameter'
  -> `t1.set_h_m(2,20)` operates on **t1**
- The existence of an interface is clear and explicit. This interface consists of functions accessible via the dot notation
  -> `t1.set_h_m()`
- Functions that operate on **different data types** using **different mechanisms** share the **same name** if they have the **same semantics**

  `t1.set()`, `t1.show()`, `a1.set()`, `a1.show()`

This is just one of the advantages of object-oriented programming.
Now I'm solely focusing on the modularity and readability of an OOP language, not on its capabilities of modeling reality.

Classes and Objects

In Object-Oriented Programming (OOP), objects represent abstractions of *concrete* or *conceptual* elements of the problem domain.

Formally, an object combines **data** and associated **behaviors**.

Classes not only *describe* objects but serve as *blueprints* for creating them.
They are like factories capable of building (*instantiating*) objects.

Attributes and behaviors

Objects are instances of classes. Each object has its own set of data, while the behavior is common to all objects of the same class.

Attributes[2] (the data) represent the individual characteristics of a certain object. The class defines the set of attributes of the object, but **any specific object can have different values for its attributes**.

- (i.e. all `TimeSlot` objects store hours and minutes, but different `TimeSlot` objects store different values).

---

[2] **Attributes** are called also **members** or **properties**. We don't use the term "properties", because the **property keyword** has a special meaning in Python.

Attributes and behaviors

Objects are instances of classes. Each object has its own set of data, while the behavior is common to all objects of the same class.

Methods[3] are functions that have direct access to the attributes of the object. We can use methods to manipulate the object. For example, `set_min()` sets the minutes.

---

[3] **Instance** methods, that is, methods of the object, to be more precise. There are also other types of methods (class methods, static methods).

Through the **self** parameter, instance methods can access attributes and other methods of the instance.
`self` represents the instantiated object on which the method operates.

Method definition: n parameters (*i.e.*, 4 parameters here)

```
def method_1(self, a, b, c):
  ...
```

Method call: n parameters (*i.e.*, **4** parameters here)

```
obj.method_1(a, b, c)
```

Let's define a class that creates Timeslot objects:

```
class NameOfTheClass:
  """DOCSTRING"""

  a = 10   # CLASS ATTRIBUTE

  def __init__(self):   # initialize
    self.x =  20   # INSTANCE ATTRIBUTE

  def f(self, y):
    self.y = self.x * y + 1
```

https://docs.python.org/3/tutorial/classes.html

Client code

```
# Instantiate (create) and use an object:
t1 = TimeSlot()
t1.set_h_m(2, 10)

print("total amount of minutes:", t1.get_m())  # Expected value: 140
print("hours %d minutes %d" % t1.get_h_m())  # Expected value: 2, 20

t1.set_m(130)
print("total amount of minutes:", t1.get_m())  # Expected value: 140
print("hours %d minutes %d" % t1.get_h_m())  # Expected value: 2, 20
```

Define the `TimeSlot` class:

```
class TimeSlot:
  """ DoCSTRING -> A class to store time slot"""

  def __init__(self, h=0, m=0):
    self.mins = m
    self.hours = h

  def set_m(self, m):
    self.hours= int(m / 60)
    self.mins = m %  60

    # the MAGIC NUMBER 60 must be replaced with a class variable
    # see next slides

    def set_h_m(self, h, m):
      self.hours = h
      self.mins = m

    def get_m(self):
      return self.hours * 60 + self.mins

    def get_h_m(self):
      return self.hours, self.mins
```

To understand exactly what we have done we can explore the created objects:

```
print(TimeSlot.__dict__)
print()
print(t1.__dict__)
```

The `__init__` method

Most object-oriented programming languages have the concept of a **constructor**, **a special method that *creates* and *initializes* the object when it is created**.

Python has a constructor `__new__` **and** an initializer `__init__`.

The `__init__` method

When we instantiate a new object `t1 = TimeSlot()` a special method, `__call__()`, is called.

`__call__()` method invokes the following:

- `__new__()` → the constructor
- `__init__()` → the initializer

The constructor `__new__` creates the new object.
The initializer `__init__` initialize the new object, add attributes, perform operations, and so on.

The identifier `self`, that is the first argument of the `__init__()` method, denotes the **new object**.

Interface

The **interface** is the collection of PUBLIC **attributes** and **methods** that other objects can use to interact with that object.

The public interface is crucial and requires careful design.
Altering the interface in the future can be challenging.
Any changes to the interface can impact client objects that uses it.

Interface

You can define a *non_public*[4] attribute or method using the prefix _ (underscore) in its name.
The underscore indicates that it is for **internal use only**.

Be careful! This is merely a "gentlemen agreement". Python **will not** prevent you from accessing the private members of an object.

---

[4] We need to decide if a class's methods and instance variables (collectively: "attributes") should be public or non-public. There are conflicting opinions.

"If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public." https://www.python.org/dev/peps/pep-0008/.

The other opinion is to set them as public and uses **properties** - SEE NEXT SLIDES

Example

Interface: `name`, `set_h_m()`

Private slots: `_h`, `_m`, `_f()`

```
class TimeSlot:
  """A class to store time slot"""

  def __init__(self, h=0, m=0):
    """initialize an empty slot"""

    self._h = h
    self._m = m
    self.name = "I am a timeslot"

    def set_h_m(self, h, m):
      ...

    def _f(self):
      ...
```

Example

```
# violation of the secret of the module!
print(t1._timeslot["h"])
t1._timeslot["h"] = 5
t1._f()

print(t1.name)   # OK! (public attribute)
```

Example

A 'true' private?[5] (the same is valid for ATTRIBUTES)

```
class C:
  def __init__(self):
    self.v1 = 10
    self._v2 = 20
    self.__v3 = 30

  def f1(self):
    print("I am f1")

  def _f2(self):
    print("I am f2")

  def __f3(self):
    print("I am f3")
```

```
obj = C()
print(obj.v1)
print(obj._v2)
print(obj.__v3)    # use this: _C__v3

obj.f1()
obj._f2()
obj.__f3()
```

---

[5] https://docs.python.org/3/tutorial/classes.html?highlight=mangling#private-variables

Class Attribute

A class attribute is a Python variable that belongs to a **class** rather than a particular object.
It is shared between all the objects of this class.

Class Attribute

```
class TimeSlot:
  """A class to store time slot"""
  minutes_in_hour = 60   # CLASS attribute

  def __init__(self):   # initialize an empty slot
    self.h = 0
    self.m = 0


t1 = TimeSlot()
print(t1.minutes_in_hour)

#  Python searches first in the namespace of the object.
#  If `minutes_in_hour` is not an instance attribute,
#  it searches in the class.
```

Try **accessing** and **changing** the class attribute `minutes_in_hour` using several objects `t1`, `t2`,... What happens?

Explore the `__dict__` attribute to understand what happened!

```
t1 = Timeslot()
t2 = Timeslot()

...

print(t1.minutes_in_hour)
print(t2.minutes_in_hour)

t1.minutes_in_hour = 10

print(t1.minutes_in_hour)
print(t2.minutes_in_hour)
```

When you access an attribute using the dot convention, Python searches first in the namespace of that object for that attribute name. If it is found, it returns the value; otherwise, it searches in the class's namespace.

Getters and Setters - Properties

Properties are used when we need to define *access control* to some attributes in an object.

Example. Write a class `P` with (instance) attributes `x` and `y`.

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


p1 = P(1, 2)
p2 = P(3, 4)
p1.x = p1.x + p2.x
```

Suppose we prefer not to grant direct access to the instance attributes `x` and `y`.
Instead, we want the client to interact with these attributes solely through specific methods known as **setters** and **getters**.

`set_x()`, `set_y()`, `get_x()`, `get_y()`

(We cannot entirely block the client from accessing x and y; we can only ask not to do so)

We introduce this constraint: **x must take only non-negative values**.
We can write setters that check the condition.

```python
class P:
  def __init__(self):
    self._x = 0
    self._y = 0

  def get_x(self):
    return self._x

  def set_x(self, x):
    if x >= 0:
      self._x = x

  # the same for y: get_y, set_y
```

```python
p = P()
print(p.get_x())   # 0
p.set_x(-10)
print(p.get_x())   # 0
p.set_x(10)
print(p.get_x())   # 10
```

The solution is simple, but it produces 'uncomfortable' code.
Let's compare the two versions with the public and the non-public attribute.

```
# public
p.x = q.x
r.x = p.x + q.x

# non-public
p.set_x(q.get_x())
r.set_x(p.get_x() + q.get_x())
```

The version with public attributes is more readable.
Python allows us to solve the problem using **properties**.

The method decorated with `@property` returns the (value held by the) private attribute `_x`.

The method decorated with `@x.setter` runs when `obj.x(val)` is called.

```
class P:
  def __init__(self):
    self._x = 0
    self._y = 0

  @property
  def x(self):
    return self._x

  @x.setter
    def x(self, x):
      if x >= 0:
        self._x = x

  # the same for y
```

```
p = P()
q = P()
r = P()

q.x = 5
r.x = -5
p.x = q.x + r.x

print(p.x, p.y)   # 5 0
```

I can use the property even in the `__init__` method.

Here, I'm not creating x or y attributes.

The code `self.x = x` will call the **setter** method

```
class P:
  def __init__(self, x=0, y=0):
    self.x = x  # the attribute is _x
    self.y = y  # the attribute is _y

  @x.setter
  ...
```

```
class P:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

  @property
  def x(self):
    return self._x

  @x.setter
  def x(self, value):
    self._x = value if value > 0 else 0

  @property
  def y(self):
    return self._y

  @y.setter
  def y(self, value):
    self._y  = value if value > 0 else 0
```

Hint: Don't write custom `get_*` and `set_*` methods for all attributes. You can start with public attributes.
If you need to modify the logic for when an attribute is retrieved or modified, then use *properties*.

If you use properties, changing the attributes from public to non-public or inserting validation logic does not change the object's interface.

Exercise

Implement the class `TimeSlot`. Each `TimeSlot` object has its name, that you must provide when instantiating the object. Write methods to **add** time slots. Uses properties.

Implement the following interface, which differs from the one used in the previous exercise.
`t.m` and `t.h` enable access to the stored minutes and hours. The entered values must be positive.

```
t1 = TimeSlot("Carbonara")
t1.m = 20
t2 = TimeSlot("Bistecca")
t2.m = 20
t3 = TimeSlot("Tiramisù")
t3.m = 30

t_menu = t1 + t2 + t3
print(t_menu)   # output: 1:10
```

Constraint: $m < 60$. If $m >= 60$, the value is set to 0 (or remains unchanged: you choose how to proceed).

Why this choice? If we decide to accept input values greater than 60 and to set separately hours and minutes, we have two possibilities. Both of them lead to an inconsistency. Consider `h = 2`, `m = 70`. The correct amount of time is 190 min (3 h, 10 min)

1. When `h` is placed in the `t.h` attribute and `m` is divided into minutes and hours, the hours are **inserted** into the `t.h` attribute, **overriding** the previous value. Not only is the stored value incorrect, but the result changes depending on the order of execution of the instructions, *i.e.*, `t.h = h`, `t.m = m` or vice versa.
2. When `h` is inserted into the `t.h` attribute and `m` is divided into minutes and hours, the hours are **added** to the previous value stored into `t.h`. In this case, the final value is correct, but the semantics of the operation are completely wrong. The meaning of `=` is an assignment, not an increment.

Please implement the different scenarios and highlight the inconsistencies.

Use the 'magic' method `__add__`[6] to implement the sum and the function `__repr__`[7] to obtain a printable representation of the object.

```
t1 = TimeSlot("Carbonara")
t1.m = 20

t2 = TimeSlot("Tiramisu")
t2.m = 30

t_menu = t1 + t2
print("t_menu-> ", t_menu.h, t_menu.m)
```

---

(6) https://docs.python.org/3/reference/datamodel.html#object.__add__

(7) https://docs.python.org/3/library/functions.html#repr

```
class TimeSlot:
  """A class to store timeslots"""
  _minutes_in_hour = 60

  def __init__(self, name="a generic timeslot", m=0):
    self.name = name
    self._h = 0
    self._m = 0

  @property
  def m(self):
    return self._m

  @m.setter
  def m(self, m):
    """if m > 60 I do not change the value"""
    if m >= self._minutes_in_hour:  # 60
      print("You put a value >= 60. The original value remains unchaged")
    else:
      self._m = m
```

Magic methods

```
# called when an object is instantiated
__new__(cls, ...)  # object creator
__init__(self, ...)  # object initializer

# called when using operators
__add__(self, other)  # +
__sub__(self, other)  # -
__mul__(self, other)  # *
__truediv__(self, other)  # /
__lt__(self, other)  # <
__eq__(self, other)  # ==
__ne__(self, other)  # !=
__gt__(self, other)  # >
__and__(self, other)  # &
__or__(self, other)  # |

# other magic methods
__int__(self)  # called when using int()
__str__(self)  # called when using str() - used for users
__repr__(self)  # called when using repr() - used for developing purpose
__len__(self)  # called when using len()
__contains__(self, item)  # called when using `in`
__getitem__(self, key)  # called when accessing item with indexing
```

Instance, Class, and Static Methods

```
class MyClass:
  class_attr = 10

  def my_method(self):
    print("the value of the class attribute is:", self.class_attr)
    return "instance method of the instance", self

  @classmethod
  def my_classmethod(cls):
    print("the value of the class attribute is:", cls.class_attr)
    return "class method of the class", cls

  @staticmethod
  def my_staticmethod():
    return "static method - " \
           "it is tied to the class, " \
           "but has no access to the class attributes or its instances"
```

```
obj = MyClass()

# INSTANCE METHOD -> my_method(self)
print(obj.my_method())
print()

# CLASS METHOD -> my_classmethod(cls)
print(MyClass.my_classmethod())  # obj.my_classmethod()
print()

# STATIC METHOD -> my_staticmethod()
print(MyClass.my_staticmethod())  # obj.my_staticmethod()
print()
```

Instance methods can access the object's attributes using the `self` identifier.

Class methods can access class attributes through the `cls` identifier.

Static methods do not have access to either the object's or the class's attributes but are linked to the class, which acts as a container.

The class's **dict** dictionary contains references to the code of the three method types.

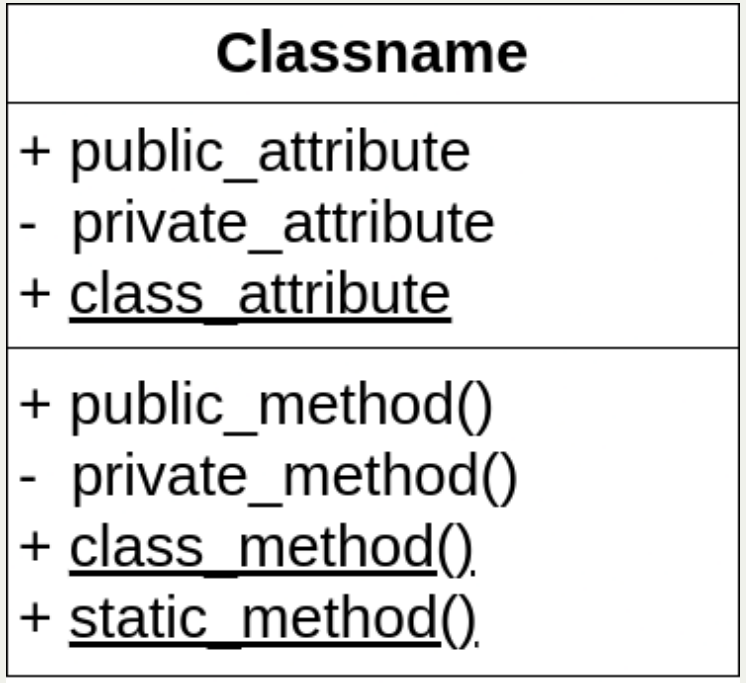
UML Class Diagram

UML (Unified Modeling Language) is a standardized modeling language composed of a set of integrated diagrams that can be used to model and visualize the design of a system under different perspectives (behavior, structure, interactions between components).

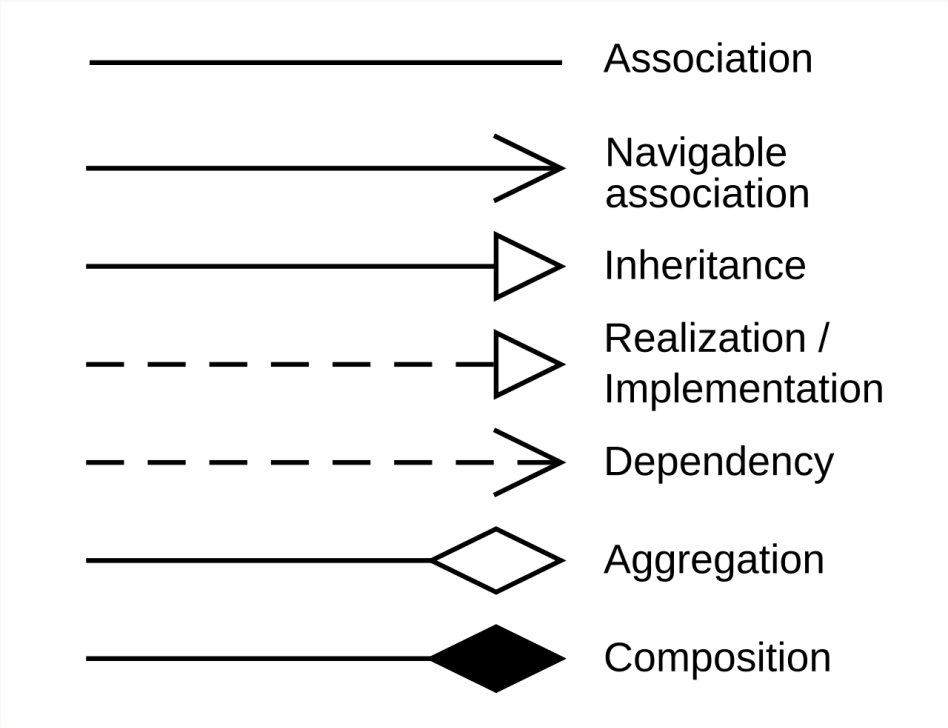
The **Class diagram** describes the system structure in terms of classes, each of which can have its own attributes and methods, and relationships among them.

Class diagrams are very useful to model object-oriented software systems, and can be easily translated into their code implementation.

A class is represented with a box containing three compartments



Connections between classes can be represented in different ways, depending on their logical relationships



By Yanpas https://en.wikipedia.org/wiki/Class_diagram#/media/File:Uml_classes_en.svg