FRUIZIONE E UTILIZZO DEI MATERIALI DIDATTICI

- **E’ vietata** la **copia**, la **rielaborazione**, la **riproduzione** dei contenuti e immagini presenti nelle lezioni in qualsiasi forma

- **E’ inoltre vietata** la **diffusione**, la **redistribuzione** e la **pubblicazione** dei contenuti e immagini, incluse le registrazioni delle videolezioni con qualsiasi modalità e mezzo non autorizzati espressamente
dall'autore o da Unica

INDUSTRIAL SOFTWARE DEVELOPMENT

Introduction to OOP

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 OOP[1]

- as a way to improve **Information Hiding** $\Longrightarrow$ **Abstraction + Encapsulation**
- as a method to improve the representation of the entities (concrete or abstract) 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.

```
def create_time_slot(h=0, m=0):
# ...
def set_h_m(time_slot, h, m):
# ...
```

```
t1 = create_time_slot() # dictionary
set_h_m(t1, 2, 20)
print(get_m(t1))  # Expected value: 140
```

It would be more useful to have a `TimeSlot` data type whose variables can be manipulated in this way:

(OOP syntax)

```
t1 = TimeSlot()
t2 = TimeSlot()
t1.set_h_m( 2, 20)
t2.set_h_m( 1, 10)
```

(instead of)

```
t1 = create_time_slot()
t2 = create_time_slot()
set_h_m(t1, 2, 20)
set_h_m(t2, 1, 10)
```

Advantages:

Advantages:

- It is much clearer **which data** is manipulated (`t1.set_h_m(2,20)` acts on `t1`)

Advantages:

- It is much clearer **which data** is manipulated (`t1.set_h_m(2,20)` acts on `t1`)
- The presence of an **interface** is explicit. The interface is the set of functions that can be called using the `dot` notation (i.e. `t1.set_h_m()`)

Advantages:

- It is much clearer **which data** is manipulated (`t1.set_h_m(2,20)` acts on `t1`)
- The presence of an **interface** is explicit. The interface is the set of functions that can be called using the `dot` notation (i.e. `t1.set_h_m()`)
- We can have interface functions with the **same name** (and same *semantic*) that act on different data types with different mechanisms.

Example. Suppose we have a `TimeSlot` data type and an `Angle` data type. `Angle` manages the measure of an angle in degrees or radians. We need a function that **shows** the value of our data. The function depends on the data itself.

(OOP syntax)

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

(instead of)

```
t1 = create_time_slot()
a1 = create_angle()
show_time_slot(t1)
show_angle(a1)
```

Classes and Objects

In OOP, Objects are abstractions of elements of the world - typically, elements of the problem domain.

Formally, an **object** is a collection of **data** and associated **behaviors**.

Classes not only *describe* objects. They are like *blueprints* for creating an object. They are like *factories* that can build (*instantiate*) objects.

Attributes and behaviors

Objects are instances of classes. An object has its own set of *data* and *behaviors*.

Attributes[2] (data) represents 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. An object has its own set of *data* and *behaviors*.

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 on the instance. `self` represents the instantiated object on which the method acts.

Method definition: n parameters

```
def m(self, a, b, c)
```

Method calling: n parameters

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

```python
# Define a class

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

    def __init__(self):  # initialize an empty slot
        self.timeslot = {'h': 0, 'm': 0}
        self.name='I am a timeslot!'
        # timeslot is an instance attribute
        #          (attribute of the object)

    def set_h_m(self, h, m):
        # set_h_m() is an instance method
        #          (method of the object)
        self.timeslot['h'] = h
        self.timeslot['m'] = m

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

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()
```

the constructor `__new__` creates the new object, and automatically calls the `__init__` method.
The identifier `self`, that is the first argument of the `__init__()` method, denote the **new object**.

We can use the `__init__` method to initialize the new object, to add attributes, to perform operations, and so on.

Interface

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

The public interface is very important. It needs to be carefully designed, as it is difficult to change it in the future. Changing the interface will break any client objects that are calling it.

Interface

The public interface is the set of all *public* methods and attributes of the object.

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] "Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. 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/

Example

Interface: `name`, `set_h_m()`
Private slots: `_timeslot`, `_f()`

```
class TimeSlot:
    """A class to store time slot"""
    def __init__(self, h=0, m=0):
        # initialize an empty slot
        self._timeslot = {'h': h, '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)
```

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.

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 namespace of the 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._timeslot = {'h': 0, '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?

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`, `y`. The initial value is $(0, 0)$.

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


print(p.x) # 0
p.x=10
print(op.x) # 10
```

We introduce this constraint: **x must take only non-negative values**.
We are stipulating a *contract* with the client, assuring that x can only be accessed through methods (called **setters** and **getters**) and not directly. In this way we can validate the values assumed by x. x must therefore be non-public. We do the same thing for y, because later there may be constraints on y as well.

```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 first `@property` method will return the value held by the private attribute `_x`. The method decorated with `@x.setter` will run 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-> (%0.2f, %0.2f)' % (p.x, p.y))
# p-> (5.00, 0.00)
```

The first `@property` method will return the value held by the private attribute _x. The method decorated with `@x.setter` will run 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->(%0.2f,%0.2f)'%(p.x, p.y))
# p-> (5.00, 0.00)
```

Properties are an excellent way to achieve **command and query separation**: "a method of an object should either answer to something or do something, but not both" (**single responsibility principle**). Suppose that a method of an object is doing something and simultaneously returns a status answering a question. This can create confusion, making it harder for readers to understand the code's actual intention.

For example, let us consider the `set_email()` method. What is this code doing?

```
self.set_email("a@j.com")
```

Is it setting the email to a@j.com, or **is it checking if** the email is already set to that value?

The `@property` decorator is the query that will answer to something, and the `@<property_name>.setter` is the command that will do something.

Hint: Don't write custom `get_*` and `set_*` methods for all attributes. You can start with regular, 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 the method to add two 'time slots' together. Define public and private attributes or methods.
(Solution: `timeslot_3.py`)

```
t1 = TimeSlot('Carbonara')
t1.set_m(20)

t2 = TimeSlot('Tiramisu')
t2.set_m(30)

t_menu = t1.add(t2)

print(t_menu.get_h_m()) # 0:50
```

Exercise

Uses propretiers for h and m. Uses `get_h_m()` and `set_h_m()` to get / set hours and minutes. Uses the 'magic' method `__add__()` to implements the sum[5] (Solution: `timeslot_4.py`)

```
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)
```

[5] https://docs.python.org/3/reference/datamodel.html#object.__add__

Instance, Class, and Static Methods

```
class MyClass:
    def my_method(self):
        return 'instance method of the instance', self

    @classmethod
    def my_classmethod(cls):
        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'
```

```python
obj = MyClass()

# INSTANCE METHOD # my_method(self)

print(obj.my_method())
# 'instance method of the instance', <__main__.MyClass object at 0x...>

# CLASS METHOD # my_classmethod(cls)

print(obj.my_classmethod())
print(MyClass.my_classmethod())
#('class method of the class', <class '__main__.MyClass'>)
```

```python
# STATIC METHOD # my_staticmethod()

print(obj.my_staticmethod())
print(MyClass.my_staticmethod())
#static method - it is tied to the class,
# but has no access to the class attributes or its instances
```