Lesson 8: Object-oriented programming
Inheritance

Implementation vs usage of classes

- Complementary concepts when coding that allows us to solve a problem.
- Two different perspectives.

Implementing an object type from a class		Use of a new object type in the code
Class definition	VS	Create a class instance
Attributes definition: What's the object Methods definition: How to use the object		Perform operations with instances: Manipulate attributes Call methods

Definition of a class vs instance of a class

Definition of a class of an object type

The name of the class is the **type**

```
class Coordinate()
```

The class is generic
We use `self` to refer to any instance that the class represents:

- We use `.` to access to data

```
(self.Cx - self.Cy)**2
```

- As a parameter when defining the methods

```
def distance (self, other):
```

The class defines all the data, (attributes) and all the methods that will be **common to all instances**.

VS

Instance of a class

One instance is an **specific object**

```
Coord = Coordinate(1,2)
```

Data attribute values vary between instances

```
C1 = Coordinate(1,2)
C2 = Coordinate(2,3)
```

- The attributes of `C1` and `C2` have diferent values `C1.x`,`C1.y`, `C2.x` i `C2.y` because they are different objects

The instance has the **class structure**

Why do we use OOP and classes?

- We want languages that are more expressive, that reduce the gap between the problem (reality) and the way to solve it (programs).
- We group objects that are part of the same type

Invoice Zara

Invoice H&M

Invoice Mango

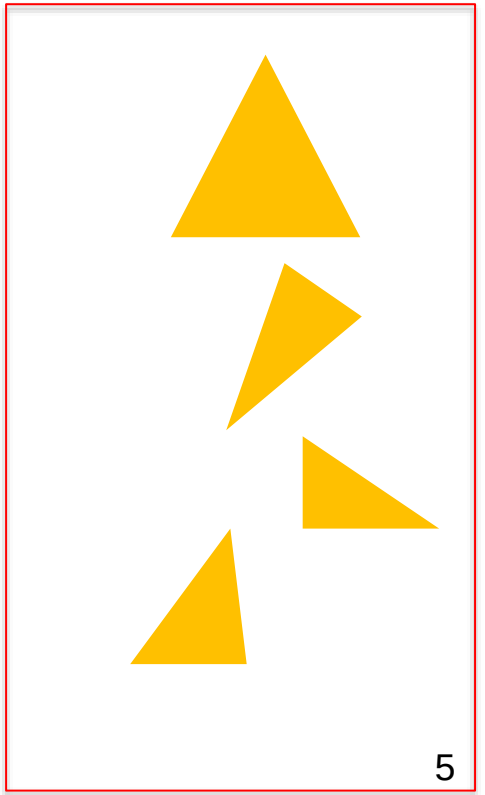
Flipper
4 years old

Bob
6 years old

Parquins
3 years old

Why do we use OOP and classes?

- We want languages that are more expressive, that reduce the gap between the problem (reality) and the way to solve it (programs).
- We group objects that are part of the same type



Classes: Groups of objects with attributes

- **Data Atributes**
  - What data represents the object?
  - What is?
    - For a *coordinate*: it is composed of x and y
    - For a *triangle*: three coordinates
    - For a *dog*: name and age
- **Procedural Atributes = methods**
  - How can we interact with the object?
  - What does it?
    - For one *coordinate*: calculate the distance between two coordinates
    - For a *triangle*: calculate its perimeter
    - For a *dog*: barking

How to define a class

- We create a class `Employee`.
- We store the attributes: name and surname.
- From the name and surname, we generate the email as: `<name>.<surname>@company.com`

```
class Employee():
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname
        self.email = name + "." + surname + "@email.com"
```

- To create an instance

```
e = Employee("Javier","Vilajosana")
```

1. Class definition
2. Class name
3. Method to create an instance
4. Variable to refer to an instance of the class
5. Parameters that contain data to initialize an `Employee`
6. Instance
7. Values to initialize one instance of the class `Employee`

Encapsulation of information

- We want to modify one attribute of an instance of the class `Employee`.

```
class Employee:
    def __init__ (self, name, surname):
        self.name = name
        self.surname = surname
        self.email = name + "." + surname + "@email.com"
```

- If we access the data of the attributes from outside the classes, errors can be generated

```
e = Employee("Javier","Vilajosana")

e.name = "Xavier"
```

```
In [4]: print(e.name,e.surname,e.email)
Xavier Vilajosana Javier.Vilajosana@email.com
```

Methods *Getter* and *Setter*

- To avoid problems and maintain the principle of **encapsulation**, data is always manipulated through methods.
- Changes or access should be made through specific methods.

Getter
Acces

Setter
Assign

```
class Employee:
    def __init__ (self, name, surname):
        self.name = name
        self.surname = surname
        self.email = name + "." + surname + "@email.com"

    def get_name (self):
        return (self.name)
    def get_surname (self):
        return (self.surname)
    def get_email (self):
        return (self.email)

    def set_name (self, name):
        self.name = name
        self.email = self.name + "." + self.surname + "@email.com"
    def set_surname (self, surname):
        self.surname = surname
        self.email = self.name + "." + self.surname + "@email.com"
```

```
In [6]: e = Employee("Javier","Vilajosana")

In [7]: e.set_name("Xavier")

In [8]: print(e.get_name(),e.get_surname(),e.get_email())
Xavier Vilajosana Xavier.Vilajosana@email.com
```

Why is encapsulation important?

- It allows us to implement the principle of abstraction and this involves:
  - Simplicity, we only make public what is needed to interact with the class
  - Ease of maintaining the code
  - Prevent errors, avoid accessing data from anywhere in the program
  - It makes it easy to test if the class is working properly

Python and encapsulation

- Python does not implement encapsulation very well
- It allows accessing to data from outside the class

```
In [9]: print (e.name)
Xavier
```

- It allows to write from outside the class

```
In [10]: e.name = "Xavier"
```

- It allows you to create new attributes in an instance from outside the class

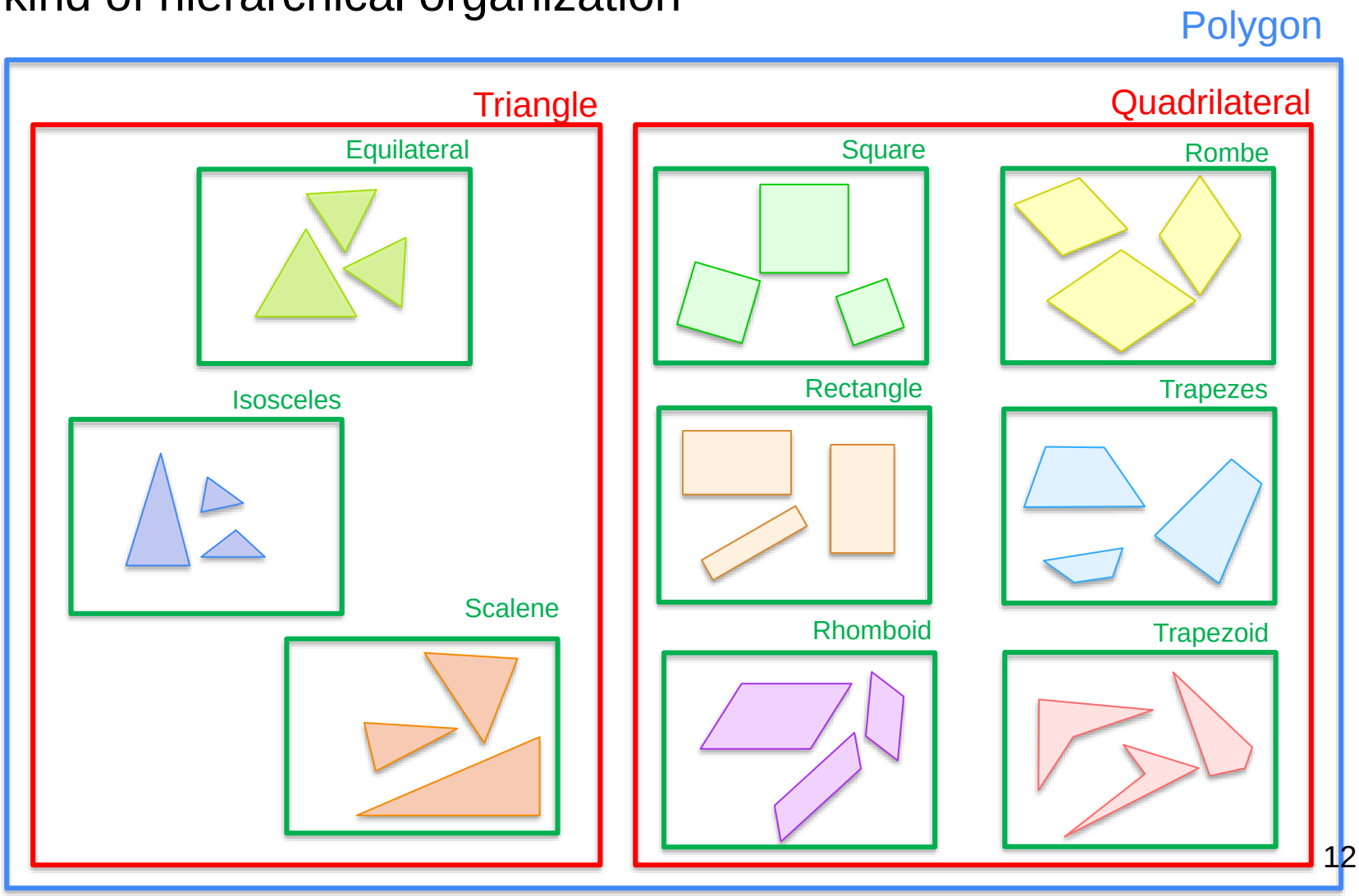
```
In [11]: e.office = "QC-1038"

In [12]: print(e.name,e.surname,e.office)
Xavier Vilajosana QC-1038
```


* In Python, a particular notation can partially implement encapsulation

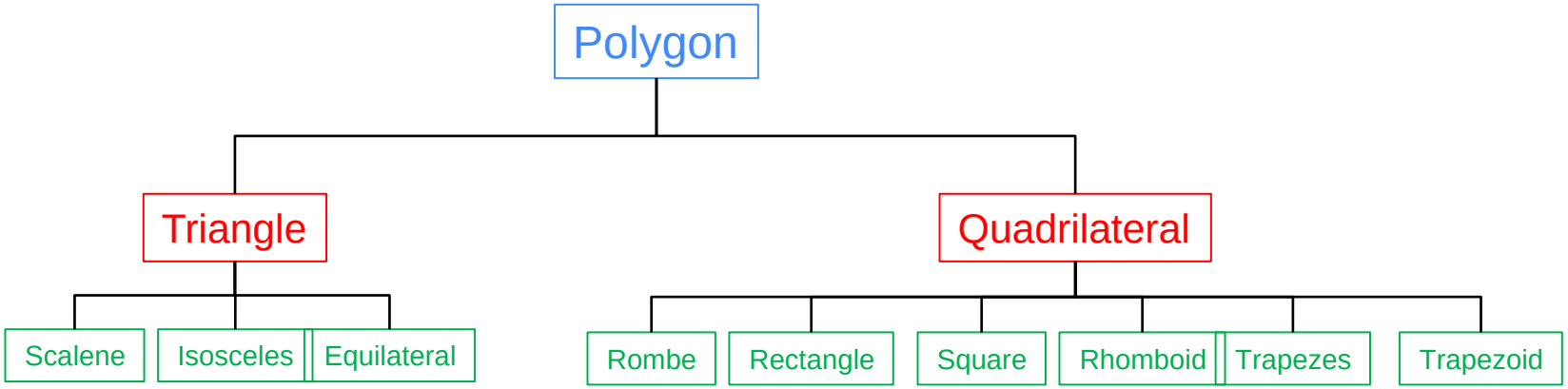
Hierarchies

- In many cases we will find that between classes there is some kind of hierarchical organization



Hierarchies

- In many cases we will find that between classes there is some kind of hierarchical organization



- In a hierarchy, we have:
  - The class “parent” (superclass)
  - The class “son” (subclass)
    - This class inherits the attributes and methods of the parent class
    - We can add more attributes
    - We can add more methods
    - We can redefine methods

Hierarchy: “Parent” class

- We have a class `Employee` with attributes and methods

```
class Employee:
    def __init__ (self, name, surname):
        self.name = name
        self.surname = surname
        self.email = name + "." + surname + "@email.com"

    def get_name (self):
        return (self.name)
    def get_surname (self):
        return (self.surname)
    def get_email (self):
        return (self.email)

    def set_name (self, name):
        self.name = name
        self.email = self.name + "." + self.surname + "@email.com"
    def set_surname (self, surname):
        self.surname = surname
        self.email = self.name + "." + self.surname + "@email.com"

    def __str__ (self):
        return "Employee: " + self.surname
```

Hierarchy: Subclass

- We want to create the class `Developer:`
  - with the same attributes than `Employee` (name, surname, email)
  - with an extra attribute: the programming language that the employee knows

```
class Developer(Employee):                                          (1)
    def __init__ (self, name, surname, prog_language):
        Employee.__init__(self, name, surname)                      (2) (3)
        self.prog_language = prog_language

    def get_prog_language(self):
        return (self.lleng_prog)

    def set_prog_language(self, prog_language):                     (4)
        self.prog_language = prog_language

    def __str__ (self):                                             (5)
        return "Dev: " + self.surname + "\n " + "Lang: " + self.prog_language
```

1. Parent class from which it inherits all atributes and methods
2. The method `__init__` uses the one from `Employee` and adds the new attributes
3. New attribute
4. New methods
5. The method `__str__` overwrites the one from `Employee` (Polymorphism)

Hierarchy: Subclass

- We want to create the class `projectManager`:
  - with the same attributes than `Employee` (name, surname, email)
  - with an extra attribute: a list of employees that he/she coordinates

```
class ProjectManager(Employee):
    def __init__ (self, name, surname, employees = None):          (1)
        Employee.__init__(self, name, surname)
        if (employees = None):
            self.employees = []
        else:
            self.employees = employees
    def add_employee(self, emp):
        if emp not in self.employees :
            self.employees.append(emp)
    def remove_employee(self, emp):
        if emp in self.employees :
            self.employees.remove(emp)
(2) def __str__ (self):
        return "PM: " + self.surname + "\n " + "Emp: " + self.employees
```

1. The default arguments are used in case the parameter is not passed when calling the method
2. The subclass can have methods with the same name as the superclass
   i. For an instance of a class, it looks for the method in the definition of the current class
   ii. If it is not found, it is searched in the hierarchy (parents, grandparents, etc.)
   iii. It uses the first method in the hierarchy that matches the method's name

Variables of a class

- Suppose we need a unique identifier for each instance of `Employee` and its subclasses

```
class Employee():
    identifier = 1                                    (1)
    def __init__ (self, name, surname):
        self.name = name
        self.surname = surname
        self.email = name + "." + surname + "@email.com"
        self.id = identifier
        identifier += 1                               (1)
    def get_name (self):
        return (self.name)
    def get_surname (self):
        return (self.surname)
    def get_email (self):
        return (self.email)
    def set_name (self, name):
        self.name = name
        self.email = self.name + "." + self.surname + "@email.com"
    def set_surname (self, surname):
        self.surname = surname
        self.email = self.name + "." + self.surname + "@email.com"
    def __str__ (self):
        return "Employee: " + self.surname
```

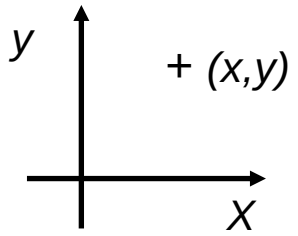
(1) Class variables and their values are shared among all instances of a class

In Python everything is an object

- The instruction `class` allows to define new data types.
- We define a coordinate in a plane:

```
class Coordinate(object):
    # Define Attributes
```

(1)



(1) The word `object` indicates that `Coordinate` is an object of Python and inherits all its attributes. `Coordinate` is a subclass of `object`

From Python version 3, it's not necessary to write `object`.

Object Oriented Programming

- Allows you to create your **own data** collections
- It allows to **organize** the information
- **Divide** the work
- Access information in a **consistent** manner
- Add **levels** of complexity
- Like functions, classes are mechanisms for applying the divide and conquer methodology: **partition** and **abstraction**.

Exercise: Type of Bank Accounts

- Type of Bank Accounts
  - Current Account: no return of any kind.
  - Remunerated Account: small return, paid monthly.
  - Fixed Account: higher return on savings deposited but, for a period of time, no funds can be withdrawn. Return is paid monthly.
- All classes have an account holder and an amount of money
- Remunerated and Fixed accounts have an interest
- Return:
  - Remunerated Account: formula of the compound capital
  - Fixed Account: formula of the simple capital
- Create an instance of each type
  - Current Account: 3.000€.
  - Remunerated Account : 3.000€, monthly interest: 0,25%
  - Fixed Account: 3.000€, monthly interest:1,25%, time period:48 months
- Implement and print the return at 12 months for the 3 accounts

Exercise: Type of Bank Accounts

```
class CurrentAccount():
    def __init__ (self, name, capital = 0):
        self.name = name
        self.capital = capital
    def get_name (self):
        return (self.name)
    def get_capital (self):
        return (self.capital)

    def set_name (self, name):
        self.name = name
    def set_capital (self,capital):
        self.capital = capital

    def compute_return (self, months):
        return (0)

    def __str__ (self):
        return "Account " + self.name + " has " + self.capital + "€"
```

Exercise: Type of Bank Accounts

```
class RemuneratedAccount(CurrentAccount):
    def __init__ (self, name, capital, interest):
        CurrentAccount.__init__(self, name, capital)
        self.interest = interest
    def get_interest (self):
        return (self.interest)

    def set_interest (self, interest):
        self.interest = interest

    def compute_return (self, months):
       r = self.capital * ( (1+(self.interest /100))**months - 1)
        return (r)


class FixedAccount(RemuneratedAccount):
    def __init__ (self, name, capital, interest, period):
        RemuneratedAccount.__init__(self, name, capital, interest)
        self.period = period
    def get_period (self):
        return (self.period)

    def set_period (self, period):
        self.period = period

    def compute_return (self, months):
        r = self.capital * ((self.interest /100)*months)
        return (r)
```

Exercise: Type of Bank Accounts

```
cc = CurrentAccount("Javier Vilajosana", 3000)
cr = RemuneratedAccount("Rodolfo Arias", 3000, 1.10)
cf = FixedAccount("Guillem Redeu", 3000, 1.25, 48)

print("CurrentA: ", cc.compute_return(12), "RemuneratedA: ",
       cr.compute_return(12), "FixedA: ", cf.compute_return(12))
```