Medical Images
Course 2023-2024

Lab session 1
Tools and work procedures

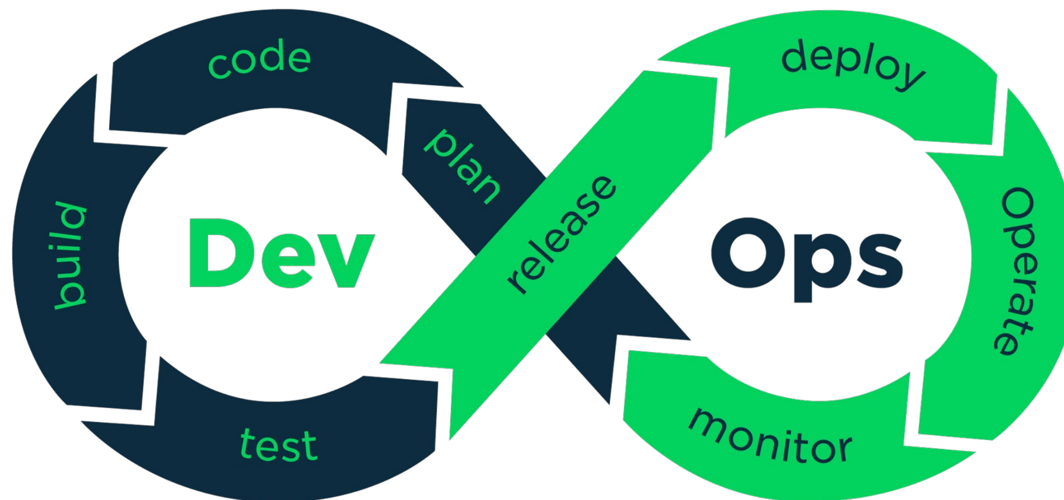
Dani Tost

Summary

- Throughout the course you'll be working on medical imaging **software development** projects
- We'll use **python3** as a programming language and the libraries: **scikit-image, numpy, matplotlib, pyqt and vtk**. You will have to install these tools in your computers. Refresh your python knowledge, please.
- On the lab's computer, we'll use **Ubuntu**. On your computers, specially if you use other operating systems, make sure that you have the required versions of the programming language and libraries.
- To manage the software projects, we will use gitlab as a version control system (vcs), concretely https://gitlab-gie.cs.upc.edu/.
- Today's class is aimed at getting familiar or refreshing the typical software development workflow and python.
- We'll start the implementation of a vectorial image

Tools

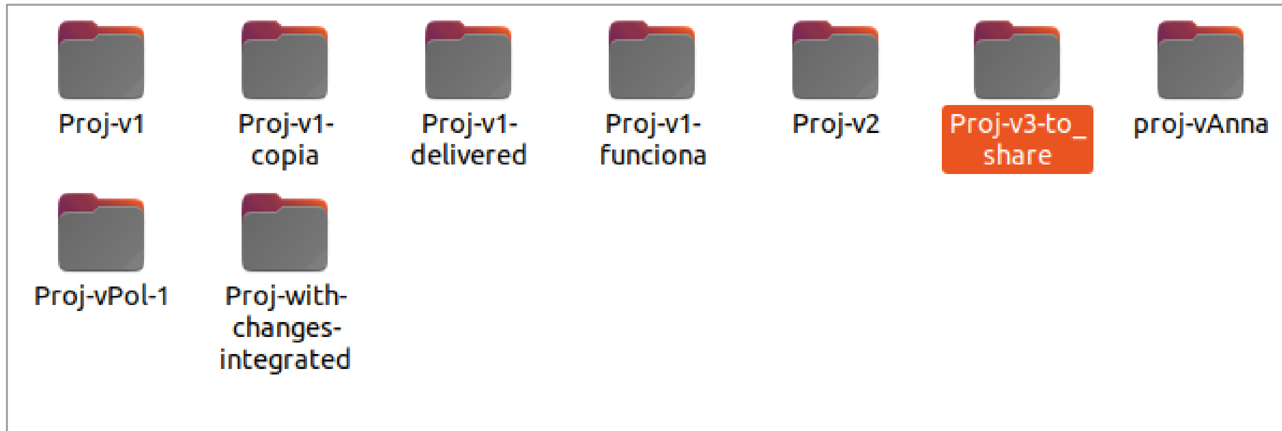
- Throughout the course you'll be working on your software projects in teams
- To manage software projects, we need to use DevOps tools that will facilitate and shorten the development life cycle and provide continuous delivery with high software quality.
- For that, we'll use gitlab and specifically our teaching instance: https://gitlab-gie.cs.upc.edu/



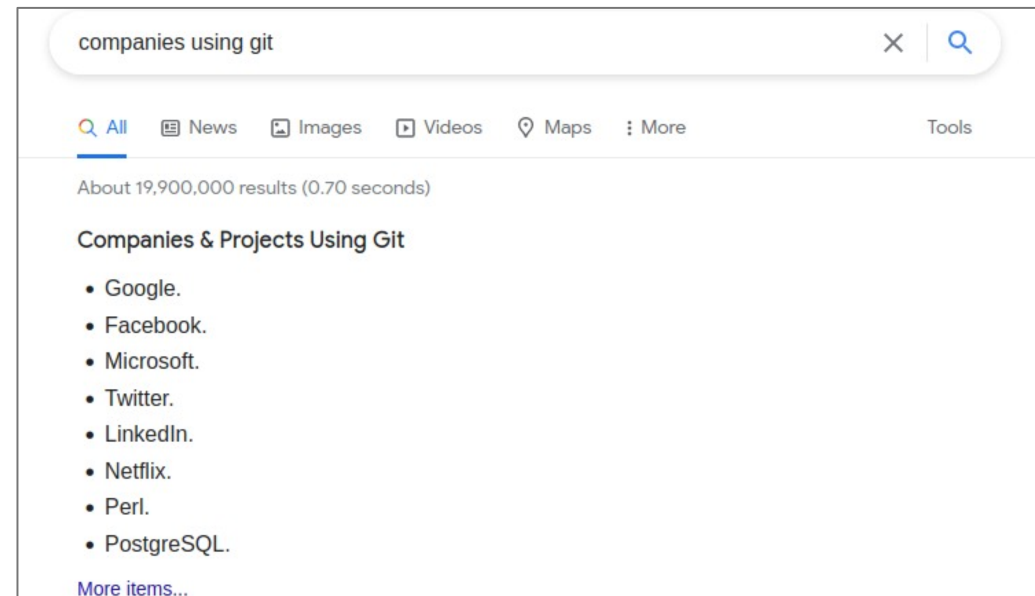
Git

Why should we use Git?

- To avoid this →



- Because it is used worldwide
- Because it is a very valued know-how
- Because it has a GNU license



Git

Installation

In linux: `sudo apt-get install git`

In windows or Mac: follow the steps
https://docs.gitlab.com/ee/gitlab-basics/start-using-git.html

Homework: Install git in your labtops

Tools

Open a **terminal** (in windows a windows terminal or a cmd window): a window in which we can type commands of the operating system.

In a **terminal**, clone the project:

```
https://gitlab-gie.cs.upc.edu/dani/medical_images_2023_2024.git
```

It is open to every one. I will create accounts for you to access to private projects starting from this session.

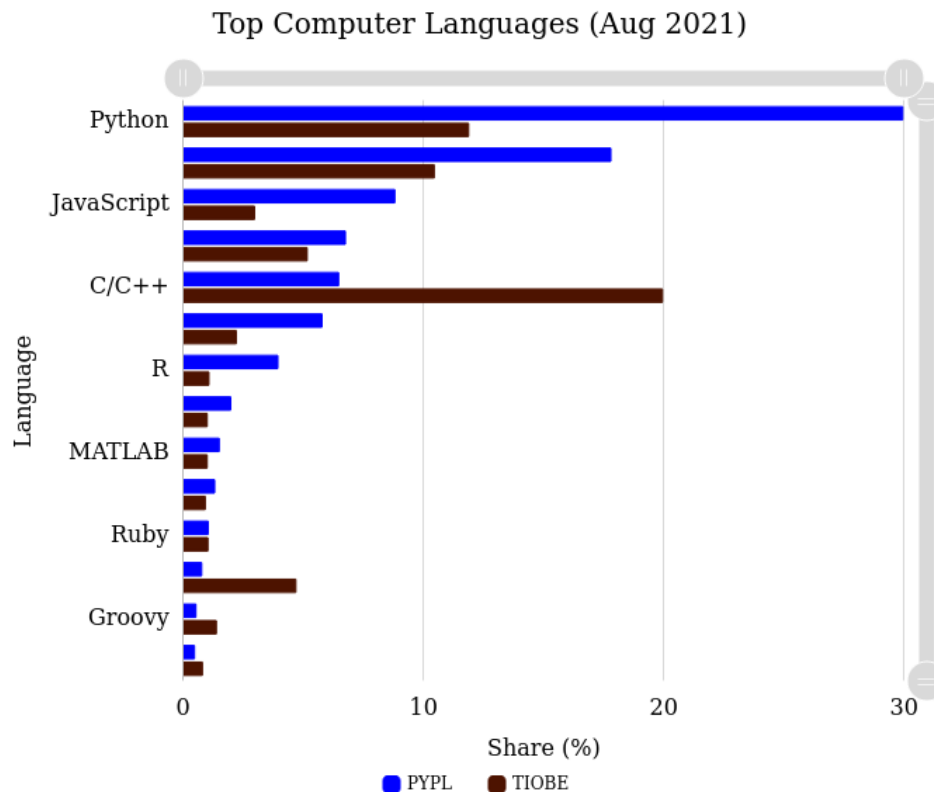
You should have a local copy of the project composed of 1 file and 1 test file.

Python

- A general-purpose programming language

- Becoming the first/second most used programming language

- Development 3-5 times shorter to develop than equivalent Java programs



Homework:
Install python3.8.10 in your laptops

Python

- Object-oriented programming with high-level data types

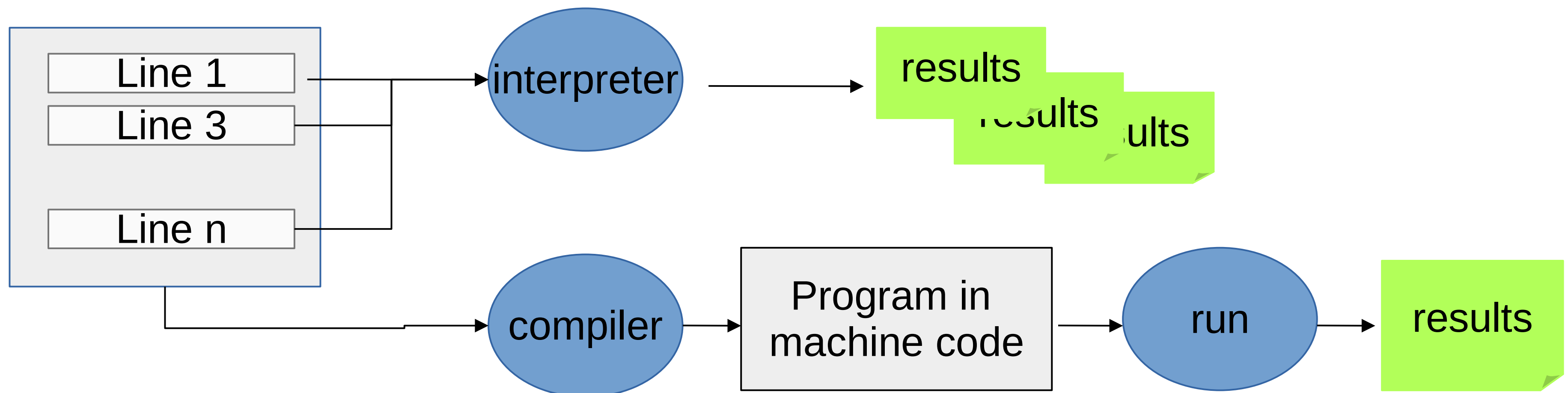
- Interpreted (no need to compile)

- Dynamic typing (the variable type is determined only during runtime, no need to pre-define the type of variable)

```
a = 34
a = 'hello'
```

- Many high-level libraries for all kinds of applications

Docs: https://docs.python.org/3/
Tutorial: https://docs.python.org/3/tutorial/index.html



Terminal and python interpreter

A **terminal** (in windows a windows terminal or a cmd window) is a window in which we can type commands of the operating system.

In Ubuntu, to open it, just click on the corresponding icon.

Each line starts with a **prompt**. You can start writing after this prompt. For instance:

```
pepe@room5.5:~pepe/IM/S1$

pepe@room5.5:~pepe/IM/S1$
```

To simplify, in the slides, we'll use the prompt `$`. You don't have to copy this symbol when you write the commands!!

A python **interpreter** or **shell** is a python environment in which you can write python commands. To open it, in the terminal write python3.

```
$ python3
>>>
```

In the python interpreter, the prompt is `>>>`

To exit: `ctrl-D` or `exit()`

Do you understand the difference between the terminal prompt and the python interpreter prompt?

Python interpreter

Once is the python interpreter, you can write python code.

Use `dir(type)` to list all methods of a class and `help(type.method)` to get help

Basic instructions

Python code is generally written in files with the extension **py** files (name.py).
These files are called python **modules**
The command blocks are delimited by **indents**

Basic commands:

- Definition of a variable and assignement: `varname = expression`
- Import a module that contains a function:

```
import module
```

or

```
from module import function
```

- Call a function:

```
module_name.function_name(parameters)
```

or

```
function_name(parameters)
```

- Define a function:

```
def function_name(parameters):
    ──────→ instructions
```

Built-in data types

Basic types: **int, float, bool, str, tuple, list, dict**

Immutable Can't be modified: int, float, bool, str, tuple

Mutable Can be modified: list, dict

listname = [val1, val2, valn]

tuplename = (val1, val2, valn)

strname = 'blablabla'

Indexation: name[i] (i = 0, 1.., n-1)

dictname = {key1: value, key2: value2, keyn: valuen}

Indexation: dictname[key]

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

Compositions

Basic compositions

Conditional composition

if expression1:
    Actions 1
elif expression2:
    Actions 2
elif expression3**:**
    Actions 3
else:
    Actions n

Iterative compositions

for i **in** range(n):
    actions

for element **in** iterable:
    actions

while condition**:**
    actions

Example

```
>>> L= ['a', 45, 2.5, True]
>>> len(L)
4
>>> for element in L:
...     print(element)
...
a
45
2.5
True
>>> for i in range(len(L)):
...     print(i, L[i])
...
0 a
1 45
2 2.5
3 True
```

Creation of python modules

We've lost all the work we've done so far. We need to save our code in modules. From now on we will edit our code with a text editor or an IDE. We will be able to use it from an interpreter or run it in a terminal.

Editors: emacs, gedit, ...
Integrated development environment IDE): spider, iddle, pycharm .

Structure of a python program

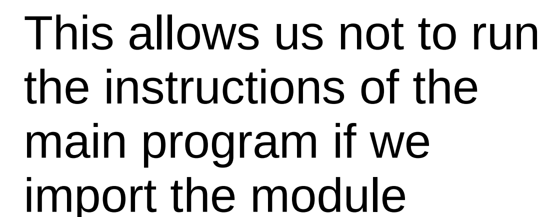
```
import library1
from library2 import class1, function2
def function1(parameters):
     code
     code
def function_main(parameters):
    code
    code

if __name__=='__main__':
      main_code
      function_main(parameters)
```

To run the program, in a terminal:

```
python3 name_module.py
```

To call a function, in a python interpreter:

```
>>> from mymodule import function1
```



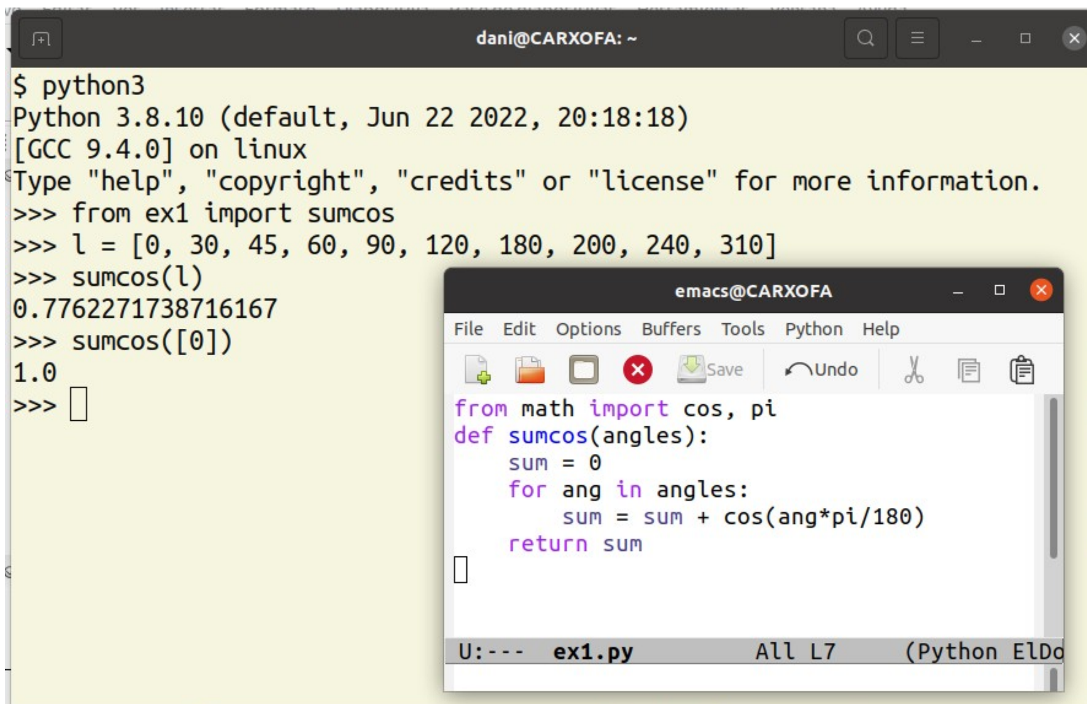
Exercise

Create the function *sumcos* after the following **specification:**

function ex1.sumcos(angles)
parameters:
mat: a list angles in degrees
returns:
the sum of the cosinus of all the angles
value return type: float

Name of the python file: ex1.py

Name of the function



Python tests

To verify that the code is clean we can use various programs: *pylint* and *black* for instance.

```
$ pylint ex1.py
************* Module ex1
ex1.py:1:0: C0114: Missing module docstring (missing-module-docstring)
ex1.py:3:4: W0622: Redefining built-in 'sum' (redefined-builtin)
ex1.py:2:0: C0116: Missing function or method docstring (missing-function
-docstring)

------------------------------------------------------------------
Your code has been rated at 5.00/10 (previous run: 5.00/10, +0.00)

$
```

I'd better change my code, add docstrings and change the name sum ...

```
$ pylint ex1.py

--------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)
```

```
$ black ex1.py
reformatted ex1.py
All done! ✨ 🍰 ✨
1 file reformatted.
```

```python
"""
MUEI/MUNR Medical Images
Course 2022-2023
First example
"""
from math import cos, pi


def sumcos(angles):
    """
    Returns the sum of the cosinus
    of the angles list expressed in degrees
    """
    sumc = 0
    for ang in angles:
        sumc = sumc + cos(ang * pi / 180)
    return sumc
```

Python tests

In addition to testing the format, we can test the functionality with tests. We'll use doctests. We'll write the tests that we would pass in a terminal in a text file and in a terminal do the command:

```
python3 -m doctest nametextfile -v |more
```

```
$ python3 -m doctest tests-ex1.txt -v |more
Trying:
    from ex1 import sumcos
Expecting nothing
ok
Trying:
    sumcos([0])
Expecting:
    1.0
ok
Trying:
    sumcos([0, 180])
Expecting:
    0.0
ok
Trying:
    round(sumcos([0, 180, 45]), 2)
Expecting:
    0.71
ok
Trying:
    round(sumcos([0, 30, 45, 180, 60]), 2)
Expecting:
    2.07
ok
1 items passed all tests:
   5 tests in tests-ex1.txt
5 tests in 1 items.
5 passed and 0 failed.
Test passed.
```

tests-ex1.txt

```
>>> from ex1 import sumcos
>>> sumcos([0])
1.0
>>> sumcos([0, 180])
0.0
>>> round(sumcos([0, 180, 45]), 2)
0.71
>>> round(sumcos([0, 30, 45, 180, 60]), 2)
2.07
```

← Here, no errors

Run a main program

Edit the module ex1.py and add to it a main program that will compute the sum of a specific list: [0, 30, 60].

Run the program in a terminal:

```
$ python3 ex1.py
0.881217650772308
```

```python
"""
MUEI/MUNR Medical Images
Course 2022-2023
First example
"""
from math import cos, pi


def sumcos(angles):
    """
    Returns the sum of the cosinus
    of the angles list expressed in degrees
    """
    sumc = 0
    for ang in angles:
        sumc = sumc + cos(ang * pi / 180)
    return sumc


if __name__=='__main__':
    print(sumcos([0, 30, 170]))
```

Classes

All objects in python are organized into **classes.** An object of a class is an **instance** of the class. It has **attributes** and **methods**.

Instanciation (creation of an instance, a variable of a class)

```
from module import classname
instance_name = classname(parameters)
```

Access to an attribute

```
instance_name.attribute_name
```

Assignment of a value to an attribute

```
instance_name.atribute_name = value
```

Invocation of a method:

```
instance_name.method_name(parameters)
```

Specification of a class

The specification of a class describes how to use and how to implement it.

Example:

Name of the module

Name of the class

```
class point2D.Point2D(x, y):
    Attributes:
        x: the x coordinate
        y: the y coordinate
    Methods:
        dist(q)
            Returns the distance between the point and q
        draw_matplotlib(plt)
            Renders the point as a small circle in the given plot
    This class supports the function str() and the comparison   ==.
```

Instantiation parameters

Use of a class

Given the specification of a class you should be able to use it, without knowing the actual implemention

```
>>> from point2D import Point2D
>>> p1 = Point2D(2, 3)
>>> str(p1)
'Point (2, 3)'
>>> p2 = Point2D(0, 0)
>>> p1 == p2
False
>>> p3 = Point2D(2.00000001, 2.99999999)
>>> p1 == p3
True
>>> p1.dist(p2)
3.605551275463989
>>> p2.dist(p1)
3.605551275463989
```

Create an instance

Invoke the special method __str__

Invoke the special method __eq__

Invoke the method dist. Obvioulsy it is symmetric

Example

Example: the class `numpy.ndarray` to represent n-dimensional arrays

numpy.ndarray

class **numpy.ndarray**(*shape*, *dtype=float*, *buffer=None*, *offset=0*, *strides=None*, *order=None*) [source]

An array object represents a multidimensional, homogeneous array of fixed-size items.

Attributes

dtype : *dtype object*
Data-type of the array's elements.

size : *int*
Number of elements in the array.

Methods

all([axis, out, keepdims, where])	Returns True if all elements evaluate to True.
any([axis, out, keepdims, where])	Returns True if any of the elements of *a* evaluate to True.

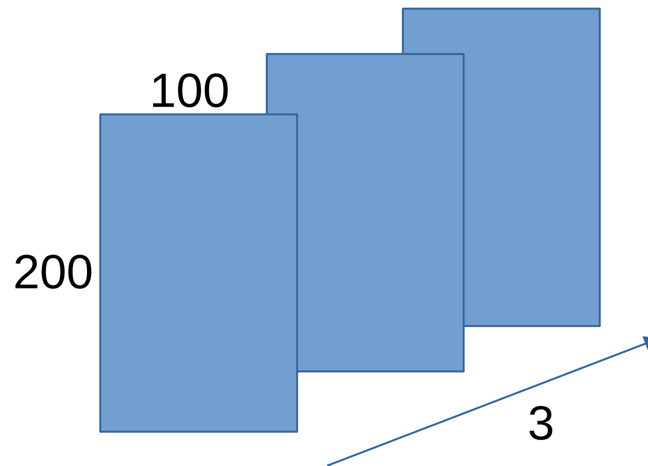
Read the **specification** in the documentation.
You should be able to know how to **create an instance**, access and modify **attributes** and i**nvoke methods.**

Example

Example: the class `numpy.ndrray` to represent n-dimensional arrays

```
>>> import numpy as np

>>> ima = np.ndarray(shape=(100, 200, 3), dtype=np.uint8)

>>> ima.size

60000

>>> ima.ndim

3

>>> ima.fill(8)

>>> ima[0,0,0]

8
```

ima is a ndarray of 100 x 200x 3 and 1 byte per value. How does this relate to images?



https://numpy.org/devdocs/reference/generated/numpy.ndarray.html?highlight=numpy%20ndarray#numpy.ndarray

https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html

Implementation of classes

During the course you'll have to implement or modify the implementation of classes. To do so, you have to read the **specification** and create the code to make the implemented class behave as described in the specification (same **attributes** and **methods**, same functionality).

The simplified structure of a class definition is:

```
class name:

   def __init__(self, parameters):
          ....
   def method1(self, parameters):
           ....
```

Some methods start and end with __ . They are **special** methods, invoked in a different way. The method **`__init__`** is invoked when creating an instance. It creates and initializes the attributes.

The headers of the methods have a first parameter «**`self`**» that represents the instance with which the method is invoked.

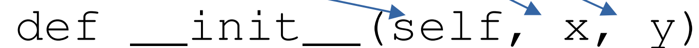
Example

Look at the implementation of the class Point2D. You'll find it in the repository you have just cloned.

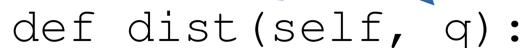
Do you understand? We need a way to refer to the instances that we will create and on which we will invoke methods. We refer to them as **self.**

Thus, the headers of all the methods have a first parameter «**`self`**» that represents the instance with which the method is invoked

```
>>> from point2D import Point2D
>>> p1 = Point(3, 4)
```

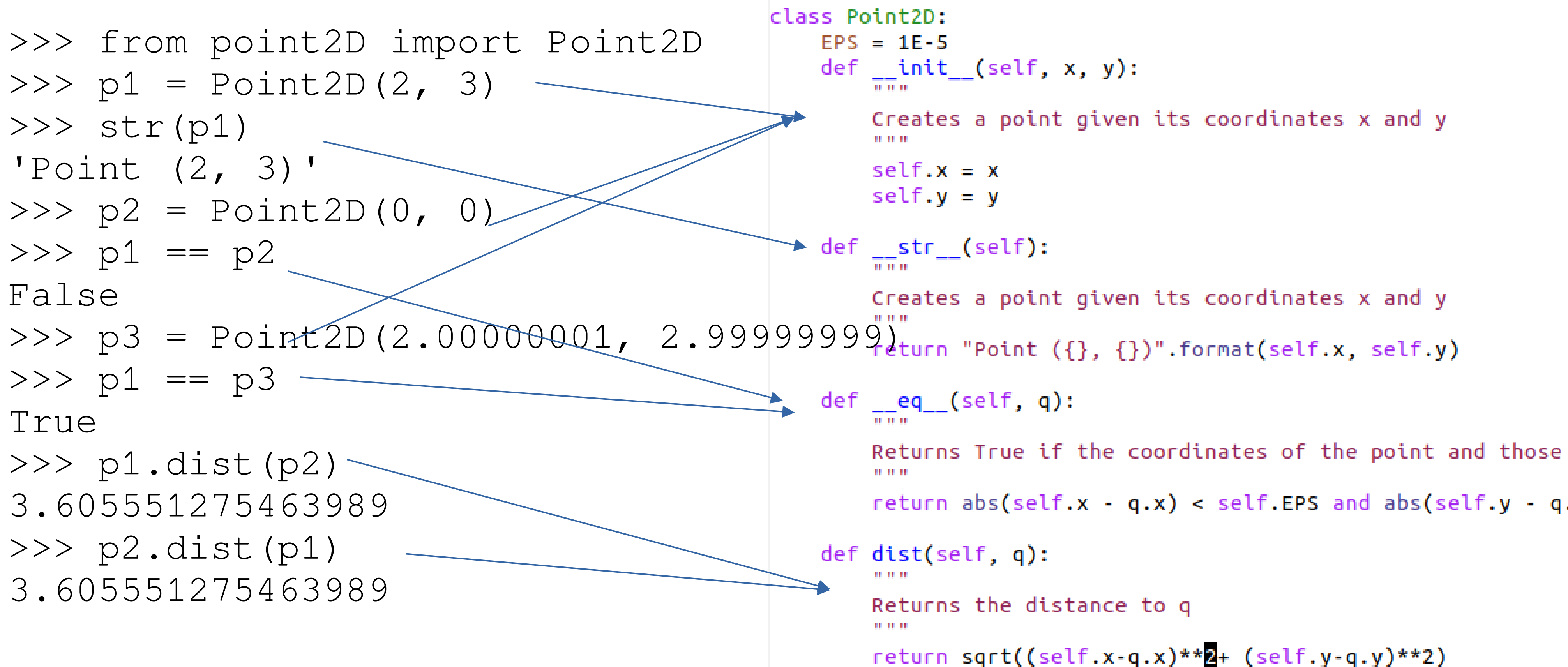



```
>>> p1.dist(p2)
```



Example

What does «supports the function» mean?

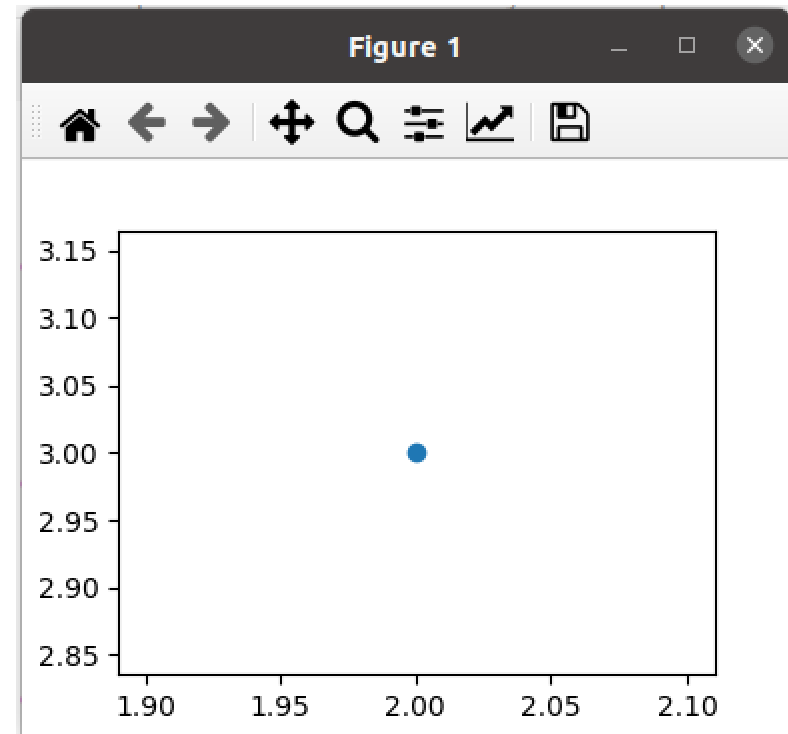


For that, we need to implement other **special** methods.

Draw a point

Points are vectorial primitives. To draw them we can use the library matplotlib.pyplot. Check the method `draw_matplotlib`.

```
def draw_matplotlib(self, plt):
    """
    Draws the point as a circle in a the given plot
    """
    plt.plot(self.x, self.y, marker="o")
```



To use the method:

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(3, 3))
<Figure size 300x300 with 0 Axes>
>>> p1.draw_matplotlib(plt)
>>> plt.show()
```

Test the code

Test files contain tests of the code.

To pass them

```
python3 -m doctest -v tests/test_point2D.txt |more
```

```
Trying:
    from point2D import Point2D
Expecting nothing
ok
Trying:
    p1 = Point2D(2, 3)
Expecting nothing
ok
Trying:
    str(p1)
Expecting:
    'Point (2, 3)'
ok
...
1 items passed all tests:
   9 tests in tests_point2D.txt
9 tests in 1 items.
9 passed and 0 failed.
Test passed.
```

Other special methods

Mèthod	Use
a.__getitem__ (expr)	a[expr]
a.__setitem__(expr, val)	a[expr] = val
a.__len__()	len(a)
a.__str__()	str(a)
a.__add__(b)	a + b
a.__sub__(b)	a – b
a.__contains__(x)	x in a
a.__le__(b) (lt/eq/ne/gt/ge)	a <=b (<,==, !=, >, >=)
a.__iter__()	iter(a)

Try to implement polygon following the specification

Class polygon.Polygon(color, x, y)

Creates a polygon filled of the given color (string) given the 2 arrays of coordinates

Attributes

col: the color of the polygon

Methods

add(v)

adds the vertex v after the last vertex of the polygon

draw_matplotlib(plt)

draws the polygon in plt

Operations

p[i]	that returns the ith vertex of the polygon
p[i] = v	assigns the vertex v to the ith vertex of the polygon
len(p)	returns the number of vertices of the polygon
str(p)	returns the string ‘Polygon of n vertices’

Use the tests!!