TECNICHE DI RAPPRESENTAZIONE E MODELLIZZAZIONE DEI DATI

— Part 1 —

(2 CFU out of 6 total CFU)

Link moodle: https://moodle2.units.it/course/view.php?id=11703

Teams code: 0ftoqj8

Python functions

A **function** is a block of code which corresponds to a set of instructions and only runs when it is called.

Main **aim** of functions: split the script in logical blocks.

Syntax of functions:

```
# the function starts here
def function_name( 1st_parameter, 2nd_parameter ):
    statements
# end of function
```

Python functions

A **function** is a block of code which corresponds to a set of instructions and only runs when it is called.

Main **aim** of functions: split the script in logical blocks.

Syntax of functions:

```
# the function starts here
def function_name( 1st_parameter, 2nd_parameter ):
    statements
# end of function
```

keyword def

function name

( ) enclosing names of the parameters (if there)

Python functions

A **function** is a block of code which corresponds to a set of instructions and only runs when it is called.

Main **aim** of functions: split the script in logical blocks.

Syntax of functions:

```
# the function starts here
def function_name( 1st_parameter, 2nd_parameter ):
    statements
# end of function
```

keyword def

function name

set of statements which are executed every time the function is called

indentation: statements must be indented

Python functions

A **function** is a block of code which corresponds to a set of instructions and only runs when it is called.

Main **aim** of functions: split the script in logical blocks.

Syntax of functions:

```
# the function starts here
def function_name( 1st_parameter, 2nd_parameter ):
    statements
# end of function
```

keyword def

function name

: token, which begins the body block of the function

the function ends when I leave the indentation block

Python functions

A script can have as many functions as the user wants.

Functions can be called only after they've been defined (i.e., below).

Functions can call other functions.

Names of the function arguments are independent of those outside the function.

Functions help the script readability.

Python functions

Functions can be broadly split into two subgroups:

void functions and **functions returning values**

Examples of void functions:

```
>>> def greetings():
...     print("Hello!")
...     print("Have a nice day!")
...
>>> greetings()
Hello!
Have a nice day!
>>>
```

Python functions

Functions can be broadly split into two subgroups:

void functions and **functions returning values**

Examples of void functions:

```
>>> def greetings():
...     print("Hello!")
...     print("Have a nice day!")
...
>>> greetings()
Hello!
Have a nice day!
>>>
```

```
>>> def greetings(name):
...     print("Hello, {}".format(name))
...     print("Have a nice day!")
...
>>> greetings("John")
Hello, John
Have a nice day!
>>> greetings("Anna")
Hello, Anna
Have a nice day!
```

Void functions perform an action but do not return any computed value to the caller (they actually return None).

Python functions

Functions can be broadly split into two subgroups:

void functions and **functions returning values**

Example of a function returning a value:

```
>>> def sum_of_numbers(a, b):
...     result = a + b
...     return result
...
>>> first_number = 3
>>> second_number = 5
>>> final_result = sum_of_numbers(first_number, second_number)
>>> print("The result is: {}".format(final_result))
The result is: 8
```

Python functions

Functions can be broadly split into two subgroups:

void functions and **functions returning values**

Example of a function returning a value:

```
>>> def sum_of_numbers(a, b):
...     result = a + b
...     return result
...
>>> first_number = 3
>>> second_number = 5
>>> final_result = sum_of_numbers(first_number, second_number)
>>> print("The result is: {}".format(final_result))
The result is: 8
```

function header
temporary variable
return temporary variable

Variables created within functions only exist within them

Function arguments are **local variables**, too

Python functions

Some examples.

```
import numpy as np

#Example 1: function that computes the volume of a sphere
def sphere_vol(radius):

    vol = (4./3.)*np.pi*(radius**3)

    return vol

# ============================================================================

#Main program: function calls

#We set radius value to 2 and call the first function
r = 2
v = sphere_vol(r)
print('The volume of a sphere of radius {} is {}'.format(r, v))
```

```
(base) milena:Desktop milenavalentini$ python Function_examples.py
The volume of a sphere of radius 2 is 33.510321638291124
```

Python functions

```python
#Example 2: same function as before, but the value of radius defaults to 1 if not provided
def sphere_vol_default(radius = 1):

    print('This is example number 2: the chosen value for the radius is {}'.format(radius))

    vol = (4./3.)*np.pi*(radius**3)

    return vol

#We set radius value to 2 and call the second function: same result as before
r = 2
v = sphere_vol_default(r)
print('The volume of a sphere of radius {} is {}'.format(r, v))

#We give no radius value and call the second function: it defaults to 1
#WARNING: first function would have returned error. Try this.
v = sphere_vol_default()
print('The volume of a sphere of radius {} is {}'.format('?', v))
```

```
(base) milena:Desktop milenavalentini$ python Function_examples.py
The volume of a sphere of radius 2 is 33.510321638291124
This is example number 2: the chosen value for the radius is 2
The volume of a sphere of radius 2 is 33.510321638291124
This is example number 2: the chosen value for the radius is 1
The volume of a sphere of radius ? is 4.1887902047863905
```

Python functions

```python
#Example 3: function with no parameters. This function returns the volume of a sphere of
    radius 5 (hardcoded) and takes no input.
def sphere_vol_no_input():

    radius = 5

    print('This is example number 3: the chosen value for the radius is {}'.format(radius))

    vol = (4./3.)*np.pi*(radius**3)

    return vol
#We call the third function with no arguments
v = sphere_vol_no_input()
print('The volume of a sphere of radius {} is {}'.format('?', v))

#WARNING: variable names used in main program and in functions are separate!
#We define a variable named 'radius' exactly as the one used in the functions
#Then we call function number 3 without arguments. Even if we define a variable 'radius'
    before calling the function, it is ignored and the value inside the function is used
radius = 20
v = sphere_vol_no_input()
print('The volume of a sphere of radius {} is {}'.format('?', v))
```

```
(base) milena:Desktop milenavalentini$ python Function_examples.py
This is example number 3: the chosen value for the radius is 5
The volume of a sphere of radius ? is 523.5987755982989
This is example number 3: the chosen value for the radius is 5
The volume of a sphere of radius ? is 523.5987755982989
```

Python functions

```python
#Example 4: void function
def sphere_vol_void(radius = 1):

    print('This is example number 4: the chosen value for the radius is {}'.format(radius))

    vol = (4./3.)*np.pi*(radius**3)

# ============================================================================

#Main program: function calls

#We set radius value to 2 and call the fourth function: returns None
r = 2
v = sphere_vol_void(r)
print('The volume of a sphere of radius {} is {}'.format(r, v))

```

```
(base) milena:Desktop milenavalentini$ python Function_examples.py
This is example number 4: the chosen value for the radius is 2
The volume of a sphere of radius 2 is None
(base) milena:Desktop milenavalentini$
```

Python functions

The “““ Test here to document ””” string within a function is called **docstring**.

Placed at the very top of the function body, it acts as a documentation on the function.

This string gets printed out when you call help( ) on the function.

```
>>> def sum_three_numbers(a, b, c):
...     """sum function that takes three numbers as input and returns their sum"""
...     result = a + b + c
...     return result
...
>>> help(sum_three_numbers)
```

```
Help on function sum_three_numbers in module __main__:

sum_three_numbers(a, b, c)
    sum function that takes three numbers as input and returns their sum
(END)
```

Python functions

Example.

```
import numpy as np

#Example 1: function that computes the volume of a sphere
def sphere_vol(radius):
    """
    This function computes the volume of a sphere given its radius. It also provides an
        example of function documentation for the creation of a manual.

    Parameters
    ----------
    radius : float
        The radius of the sphere for which the volume has to be computed.

    Returns
    -------
    vol: float
        The volume of the sphere.
    """

    vol = (4./3.)*np.pi*(radius**3)

    return vol

```

Python functions

Exercises.

1. Write a function that computes the volume of a cylinder given radius and height. Make it so that radius and height default to 1 if they are not given.

2. Write another function that prints a sentence (‘Hello World’) and takes no input. Make it a void function.

3. Write one function that for each radius of a list of at least four radii computes diameter, circumference and area (via a single call to function).
   Then, use the function and print a sentence like
   “Radius xx has: diameter yy, circumference zz, and area ww”.

Python: numpy

Search the docs ...

GETTING STARTED

- What is NumPy?
- Installation
- NumPy quickstart
- NumPy: the absolute basics for beginners

FUNDAMENTALS AND USAGE

- NumPy fundamentals
- NumPy for MATLAB users
- NumPy Tutorials
- NumPy how-tos

ADVANCED USAGE AND INTEROPERABILITY

- Building from source
- Using NumPy C-API
- F2PY user guide and reference manual

NumPy

What is NumPy?

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

At the core of the NumPy package, is the *ndarray* object. This encapsulates *n*-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an *ndarray* will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.
- A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today's scientific/mathematical Python-based software, just knowing how to use Python's built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

Python: numpy

The N-dim array (ndarray) object and operations with arrays

```
In [2]: import numpy as np

In [3]: my_array_1 = np.array([1, 2, 3, 5, 10, 20, 30])

In [4]: my_array_1
Out[4]: array([ 1,  2,  3,  5, 10, 20, 30])

In [5]: type(my_array_1)
Out[5]: numpy.ndarray

In [6]: my_array_2 = np.array([5, 11, 6, 5, 10, 25, 60])

In [7]: my_array_1+my_array_2
Out[7]: array([ 6, 13,  9, 10, 20, 45, 90])
```

Python: numpy

The N-dim array (ndarray) object and operations with arrays

```
In [2]: import numpy as np

In [3]: my_array_1 = np.array([1, 2, 3, 5, 10, 20, 30])

In [4]: my_array_1
Out[4]: array([ 1,  2,  3,  5, 10, 20, 30])

In [5]: type(my_array_1)
Out[5]: numpy.ndarray

In [6]: my_array_2 = np.array([5, 11, 6, 5, 10, 25, 60])

In [7]: my_array_1+my_array_2
Out[7]: array([ 6, 13,  9, 10, 20, 45, 90])
```

```
In [9]: np.log10((my_array_1+my_array_2)/my_array_1)
Out[9]:
array([0.77815125, 0.81291336, 0.47712125, 0.30103   , 0.30103   ,
       0.35218252, 0.47712125])
```

How to create an N-dim matrix

numpy.ones creates an array and fills it with 1.

```
In [12]: my_matrix = np.ones((2,5,3))

In [13]: my_matrix
Out[13]:
array([[[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]],

       [[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]]])

In [14]: my_matrix*2
Out[14]:
array([[[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]],

       [[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]]])
```

Python: numpy

What are the dimensions of the matrix?

```
In [15]: my_matrix.shape
Out[15]: (2, 5, 3)
```

```
In [12]: my_matrix = np.ones((2,5,3))

In [13]: my_matrix
Out[13]:
array([[[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]],

       [[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]]])

In [14]: my_matrix*2
Out[14]:
array([[[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]],

       [[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]]])
```

Python: numpy

```
In [12]: my_matrix = np.ones((2,5,3))

In [13]: my_matrix
Out[13]:
array([[[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]],

       [[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]]])
```

```
In [15]: my_matrix.shape
Out[15]: (2, 5, 3)
```

```
In [3]: a = [1,2,3]

In [4]: a.shape
Traceback (most recent call last):

  Cell In[4], line 1
    a.shape

AttributeError: 'list' object has no attribute 'shape'
```

The shape attribute only works with arrays

An attribute is a ~feature of the data structure
that you can access with the .
(if the method is present for that data structure)

numpy.shape

`numpy.shape(a)` [source]

Return the shape of an array.

Parameters: **a** : *array_like*

Input array.

Returns: **shape** : *tuple of ints*

The elements of the shape tuple give the lengths of the corresponding array dimensions.

See also

`len`

`len(a)` is equivalent to `np.shape(a)[0]` for N-D arrays with `N>=1`.

`ndarray.shape`

Equivalent array method.

len( ) also works with e.g. lists

numpy.matrix.shape

attribute

`matrix.shape`

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with `numpy.reshape`, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

Operations with matrices

```
In [21]: my_matrix = np.ones((3,4,2))*3

In [22]: my_matrix
Out[22]:
array([[[3., 3.],
        [3., 3.],
        [3., 3.],
        [3., 3.]],

       [[3., 3.],
        [3., 3.],
        [3., 3.],
        [3., 3.]],

       [[3., 3.],
        [3., 3.],
        [3., 3.],
        [3., 3.]]])

In [23]: np.sum(my_matrix)
Out[23]: 72.0

In [24]: my_matrix.shape
Out[24]: (3, 4, 2)
```

Python: numpy

Operations with matrices

```
In [21]: my_matrix = np.ones((3,4,2))*3

In [22]: my_matrix
Out[22]:
array([[[3., 3.],
        [3., 3.],
        [3., 3.],
        [3., 3.]],

       [[3., 3.],
        [3., 3.],
        [3., 3.],
        [3., 3.]],

       [[3., 3.],
        [3., 3.],
        [3., 3.],
        [3., 3.]]])

In [23]: np.sum(my_matrix)
Out[23]: 72.0

In [24]: my_matrix.shape
Out[24]: (3, 4, 2)
```

```
In [25]: sum_axis_0 = np.sum(my_matrix, axis = 0)

In [26]: sum_axis_0
Out[26]:
array([[9., 9.],
       [9., 9.],
       [9., 9.],
       [9., 9.]])

In [27]: sum_axis_0.shape
Out[27]: (4, 2)

In [28]: sum_axis_1 = np.sum(my_matrix, axis = 1)

In [29]: sum_axis_1
Out[29]:
array([[12., 12.],
       [12., 12.],
       [12., 12.]])

In [30]: sum_axis_1.shape
Out[30]: (3, 2)

In [31]: sum_axis_2 = np.sum(my_matrix, axis = 2)

In [32]: sum_axis_2
Out[32]:
array([[6., 6., 6., 6.],
       [6., 6., 6., 6.],
       [6., 6., 6., 6.]])

In [33]: sum_axis_2.shape
Out[33]: (3, 4)
```

Python: numpy

numpy.where

```
In [47]: my_array = np.array([1, 2, 5, 5, 3, 10, 5])

In [48]: my_array
Out[48]: array([ 1,  2,  5,  5,  3, 10,  5])

In [49]: my_array.shape
Out[49]: (7,)

In [50]: my_array_id = np.where(my_array == 5)[0]

In [51]: my_array_id
Out[51]: array([2, 3, 6])
```

numpy.where

```
In [47]: my_array = np.array([1, 2, 5, 5, 3, 10, 5])

In [48]: my_array
Out[48]: array([ 1,  2,  5,  5,  3, 10,  5])

In [49]: my_array.shape
Out[49]: (7,)

In [50]: my_array_id = np.where(my_array == 5)[0]

In [51]: my_array_id
Out[51]: array([2, 3, 6])

In [52]: my_array_id = np.where(my_array == 5)

In [53]: my_array_id
Out[53]: (array([2, 3, 6]),)
```

Python: numpy

numpy.where

```
In [47]: my_array = np.array([1, 2, 5, 5, 3, 10, 5])

In [48]: my_array
Out[48]: array([ 1,  2,  5,  5,  3, 10,  5])

In [49]: my_array.shape
Out[49]: (7,)

In [50]: my_array_id = np.where(my_array == 5)[0]

In [51]: my_array_id
Out[51]: array([2, 3, 6])

In [52]: my_array_id = np.where(my_array == 5)

In [53]: my_array_id
Out[53]: (array([2, 3, 6]),)

In [54]: type((7))
Out[54]: int

In [55]: type((7,))
Out[55]: tuple
```

Python: numpy arrays

numpy.where

```
In [47]: my_array = np.array([1, 2, 5, 5, 3, 10, 5])

In [48]: my_array
Out[48]: array([ 1,  2,  5,  5,  3, 10,  5])

In [49]: my_array.shape
Out[49]: (7,)

In [50]: my_array_id = np.where(my_array == 5)[0]

In [51]: my_array_id
Out[51]: array([2, 3, 6])
```

```
In [58]: my_result = np.where(my_array == 5, -100, 100)

In [59]: my_result
Out[59]: array([ 100,  100, -100, -100,  100,  100, -100])
```

Python: numpy

Are array operations convenient?
Which speed up can I achieve?

```
import numpy as np
import time

#Define large matrices
large_matrix_1 = np.ones((1000, 1000))*5
large_matrix_2 = np.ones((1000, 1000))*2

#Empty matrix to be filled
result_large_matrix_1 = np.zeros((1000, 1000))

#Operate on them with for loop
t0 = time.time()
for i in range(1000):
    for j in range(1000):

            result_large_matrix_1[i, j] = large_matrix_1[i, j]**2 + np.log10(large_matrix_1[i, j]) + np.sqrt(large_matrix_2[i, j]) + large_matrix_2[i, j]**3

print('For loop takes {0} seconds'.format(time.time()-t0))

#Operate on them with array operations
t0 = time.time()
result_large_matrix_2 = large_matrix_1**2 + np.log10(large_matrix_1) + np.sqrt(large_matrix_2) + large_matrix_2**3

print('Array operation takes {0} seconds'.format(time.time()-t0))

print('The result is the same: {0}'.format(np.all(result_large_matrix_1 == result_large_matrix_2)))
```

Python: numpy

Are array operations convenient?
Which speed up can I achieve?

```python
import numpy as np
import time

#Define large matrices
large_matrix_1 = np.ones((1000, 1000))*5
large_matrix_2 = np.ones((1000, 1000))*2

#Empty matrix to be filled
result_large_matrix_1 = np.zeros((1000, 1000))

#Operate on them with for loop
t0 = time.time()
for i in range(1000):
    for j in range(1000):

            result_large_matrix_1[i, j] = large_matrix_1[i, j]**2 + np.log10(large_matrix_1[i, j]) +
                np.sqrt(large_matrix_2[i, j]) + large_matrix_2[i, j]**3

print('For loop takes {0} seconds'.format(time.time()-t0))

#Operate on them with array operations
t0 = time.time()
result_large_matrix_2 = large_matrix_1**2 + np.log10(large_matrix_1) + np.sqrt(large_matrix_2) + large_matrix_2**3

print('Array operation takes {0} seconds'.format(time.time()-t0))

print('The result is the same: {0}'.format(np.all(result_large_matrix_1 == result_large_matrix_2)))
```

```
For loop takes 2.521970272064209 seconds
Array operation takes 0.018016815185546875 seconds
The result is the same: True
```

Python: numpy

Other relevant features of numpy

```
>>> import numpy as np
>>>
>>> a = np.array([2, 5, 6, 9, 12, 18, 21])
>>> min_a = np.min(a)
>>> max_a = np.max(a)
>>> mean_a = np.mean(a)
>>> std_a = np.std(a)
>>> print('Min = {}, max = {}, mean = {}, std = {}'.format(min_a, max_a, mean_a, std_a))
Min = 2, max = 21, mean = 10.428571428571429, std = 6.477590885002648
```

Python: numpy

Other relevant features of numpy

```
>>> import numpy as np
>>>
>>> a = np.array([2, 5, 6, 9, 12, 18, 21])
>>> min_a = np.min(a)
>>> max_a = np.max(a)
>>> mean_a = np.mean(a)
>>> std_a = np.std(a)
>>> print('Min = {}, max = {}, mean = {}, std = {}'.format(min_a, max_a, mean_a, std_a))
Min = 2, max = 21, mean = 10.428571428571429, std = 6.477590885002648
```

```
>>> b = np.array([7, 5, 3, 2, 6, 1, 4, 8])
>>> sorted_b = np.sort(b)
>>> print(sorted_b)
[1 2 3 4 5 6 7 8]
>>> print(sorted_b[::-1])
[8 7 6 5 4 3 2 1]
```

Python: numpy

Other relevant features of numpy

```
>>> import numpy as np
>>>
>>> a = np.array([2, 5, 6, 9, 12, 18, 21])
>>> min_a = np.min(a)
>>> max_a = np.max(a)
>>> mean_a = np.mean(a)
>>> std_a = np.std(a)
>>> print('Min = {}, max = {}, mean = {}, std = {}'.format(min_a, max_a, mean_a, std_a))
Min = 2, max = 21, mean = 10.428571428571429, std = 6.477590885002648
```

```
>>> b = np.array([7, 5, 3, 2, 6, 1, 4, 8])
>>> sorted_b = np.sort(b)
>>> print(sorted_b)
[1 2 3 4 5 6 7 8]
>>> print(sorted_b[::-1])
[8 7 6 5 4 3 2 1]
```

```
>>> val = np.arange(0, 5, 0.3)
>>> print(val)
[0.  0.3 0.6 0.9 1.2 1.5 1.8 2.1 2.4 2.7 3.  3.3 3.6 3.9 4.2 4.5 4.8]
```

Python: exercise

Exercise 1.

Create two arrays (for instance with numpy.arange).
Let's call them array_1 and array_2.
Put some zeros in at least 10 entries of array_2.

Goal: compute array_1 / array_2

Intermediate steps to perform:

I. — use numpy.where to identify the zeros in array_2, and then compute array_1 / array_2 only when the zero entries are not involved.

II. — Use numpy.where to replace the zeros in array_2 with 0.001, and then compute array_1 / array_2 for all the entries.

III. — Try to compute array_1 / array_2 (with array_1 and array_2 being the original arrays without modifications) and print the indices of the resulting array where the division by zero occurs (use numpy.isinf).

Python: libraries

A Python library is a collection of codes and modules.

It contains bunches of code that can be used repeatedly in different programs for specific operations.

We use libraries so that we don't need to write the code again in our program that is already available.

Python: libraries

matplotlib Plot types User guide Tutorials Examples Reference Contribute Releases

Latest stable release
3.8.0: docs | release notes

Last release for Python 2
2.2.5: docs | changelog

Matplotlib 3.8.0 documentation

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations.

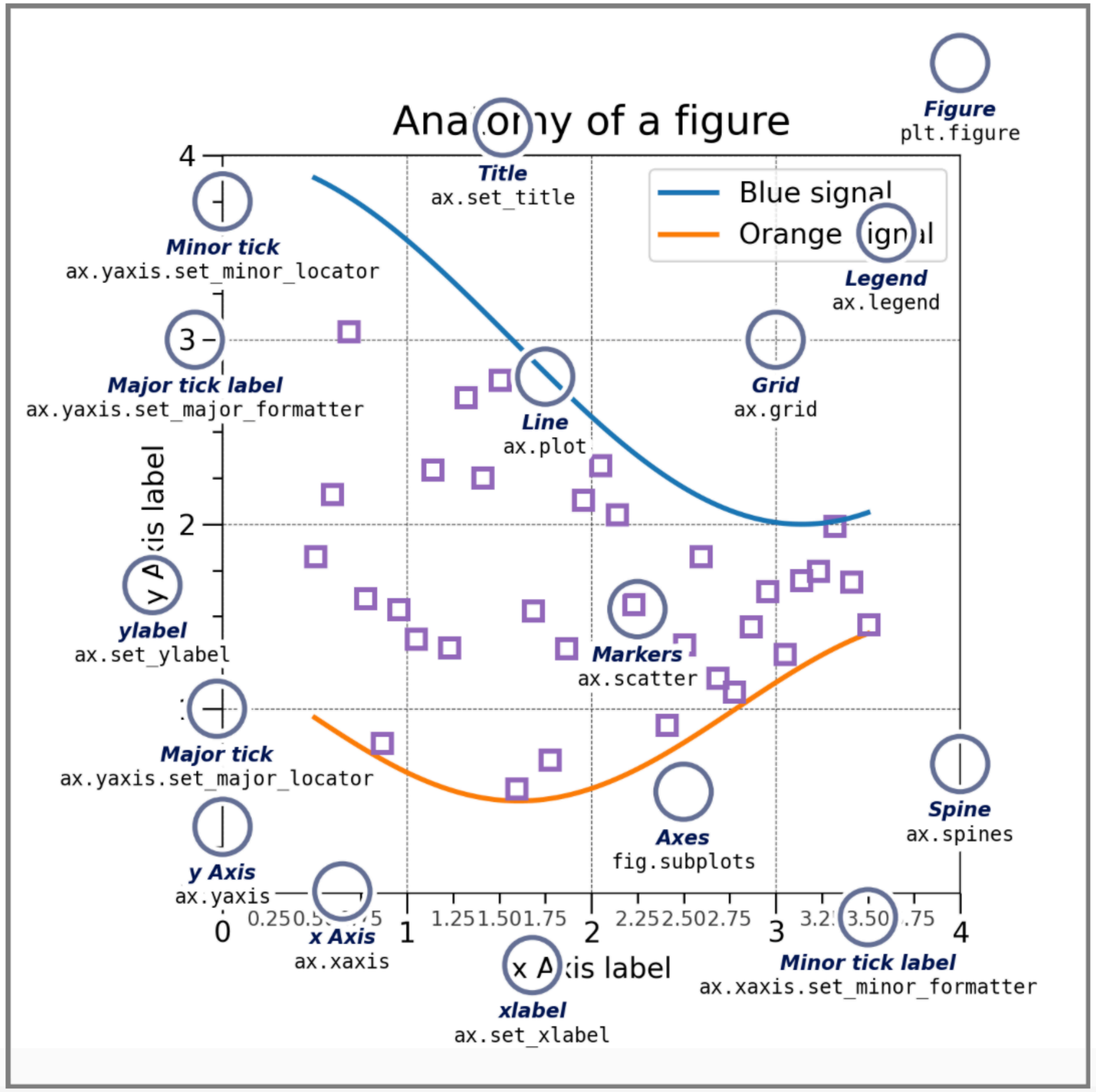
Install

pip conda other

```
pip install matplotlib
```

Parts of a Figure

Here are the components of a Matplotlib Figure.



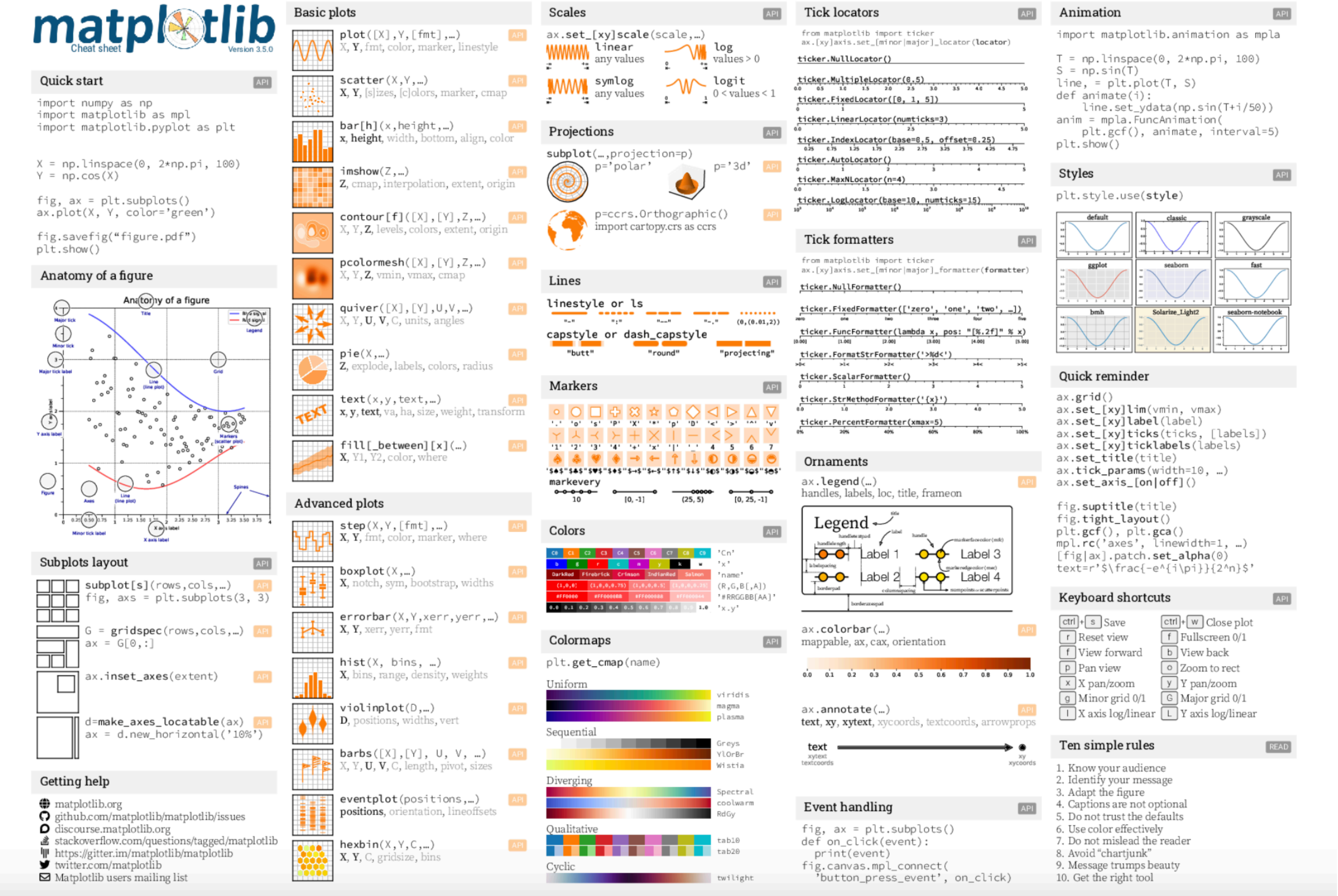
matplotlib

A Python library is a collection of codes and modules.

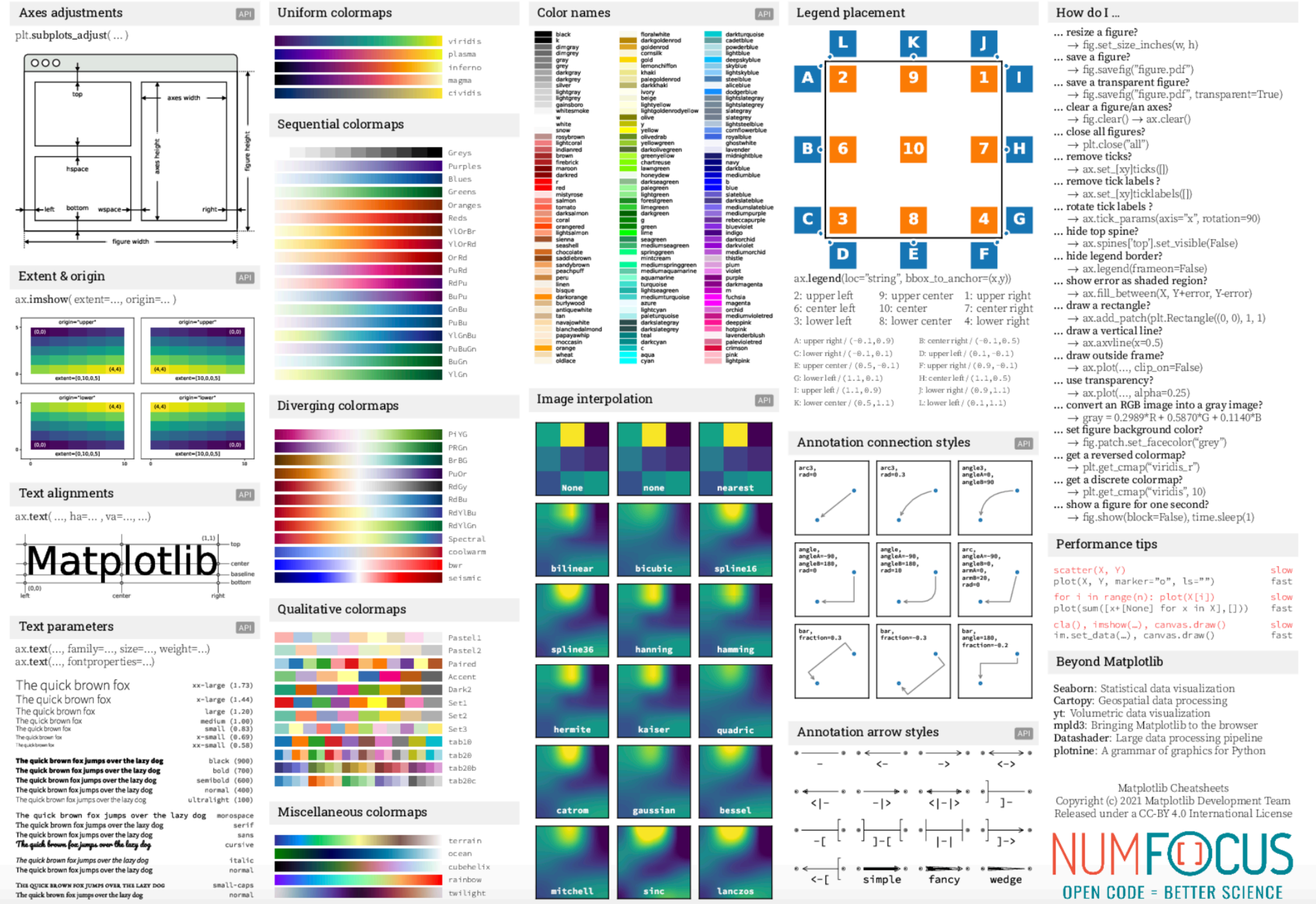
It contains bunches of code that can be used repeatedly in different programs for specific operations.

We use libraries so that we don't need to write the code again in our program that is already available.

Python: libraries



Python: libraries

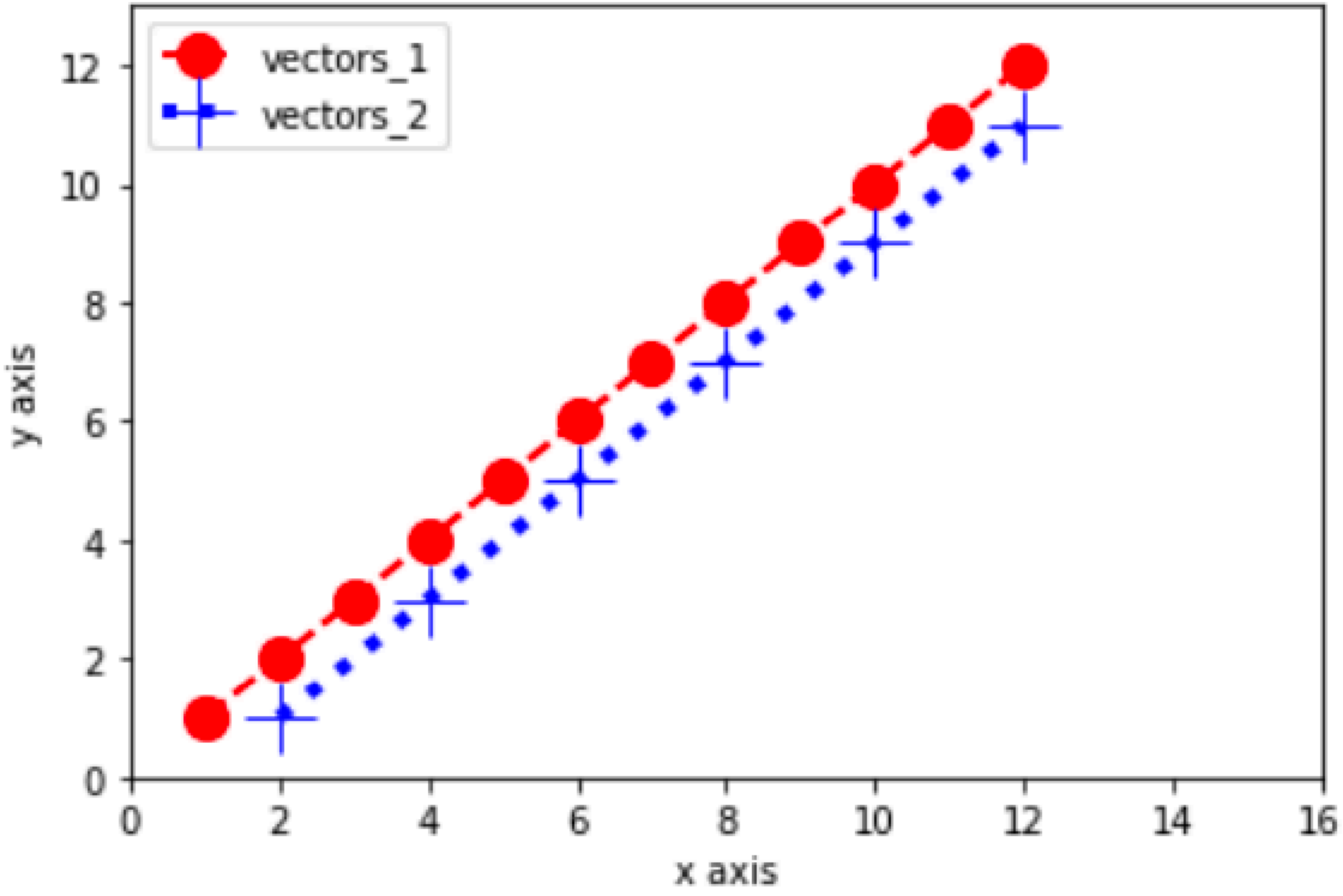


Python: plotting examples

```python
@author: milenavalentini
"""


import numpy as np
import matplotlib.pyplot as mpl

x_vector_1 = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
y_vector_1 = np.array([1,2,3,4,5,6,7,8,9,10,11,12])

x_vector_2 = np.array([2,4,6,8,10,12])
y_vector_2 = np.array([1,3,5,7,9,11])

mpl.plot(x_vector_1, y_vector_1, color='red', marker='o', linestyle='dashed',
     linewidth=2, markersize=12, label='vectors_1')

mpl.plot(x_vector_2, y_vector_2, color='blue', marker='+', linestyle='dotted',
     linewidth=4, markersize=20, label='vectors_2')

mpl.xlim(0,16)
mpl.ylim(0,13)

mpl.xlabel('x axis')
mpl.ylabel('y axis')

mpl.legend()

mpl.show()
mpl.savefig('a_first_plot.png')
```

Python: plotting examples

```python
@author: milenavalentini
"""


import numpy as np
import matplotlib.pyplot as mpl

x_vector_1 = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
y_vector_1 = np.array([1,2,3,4,5,6,7,8,9,10,11,12])

x_vector_2 = np.array([2,4,6,8,10,12])
y_vector_2 = np.array([1,3,5,7,9,11])

mpl.plot(x_vector_1, y_vector_1, color='red', marker='o', linestyle='dashed',
         linewidth=2, markersize=12, label='vectors_1')

mpl.plot(x_vector_2, y_vector_2, color='blue', marker='+', linestyle='dotted',
         linewidth=4, markersize=20, label='vectors_2')

mpl.xlim(0,16)
mpl.ylim(0,13)

mpl.xlabel('x axis')
mpl.ylabel('y axis')

mpl.legend()

mpl.show()
mpl.savefig('a_first_plot.png')
```

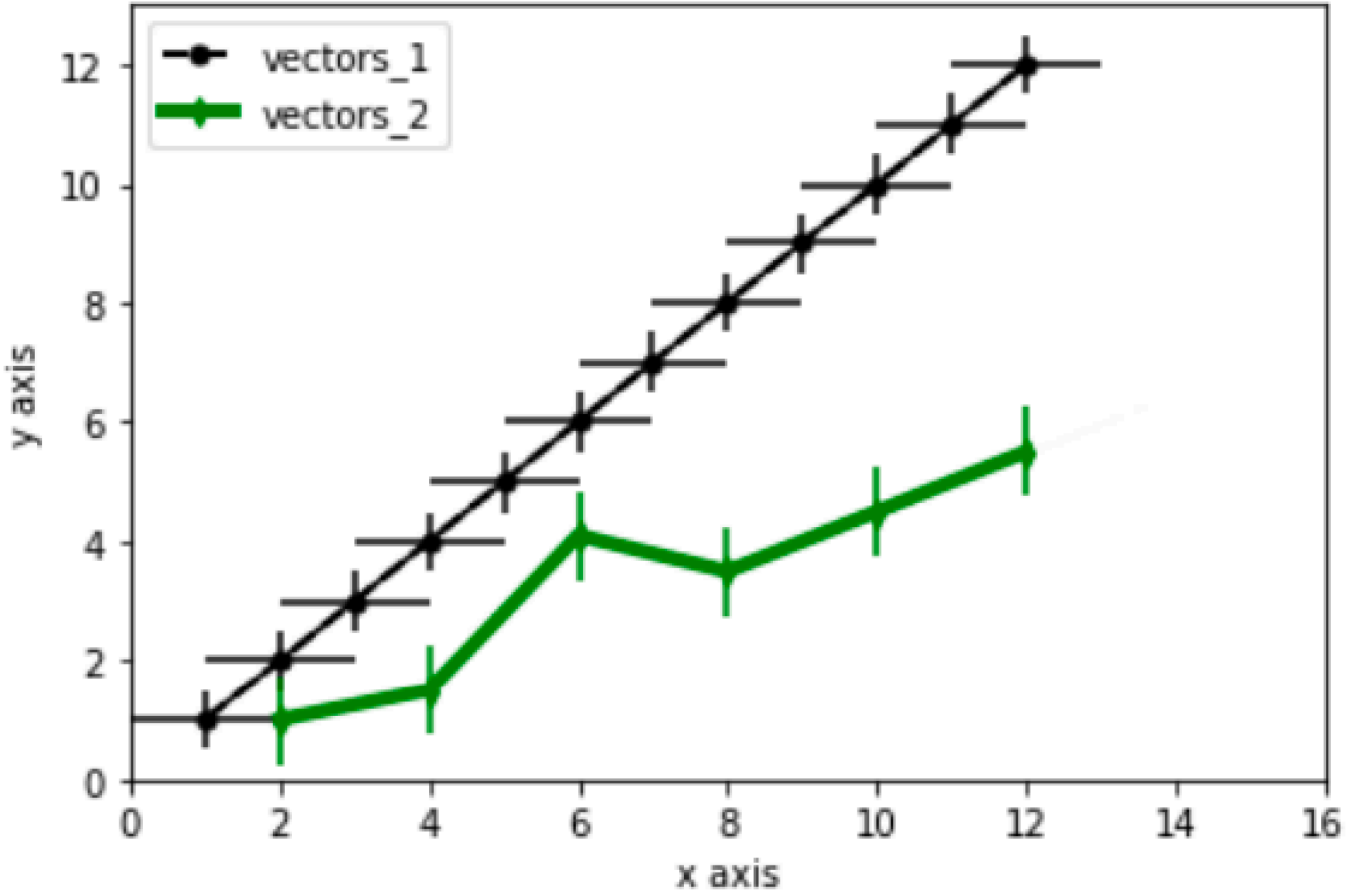


Python: plotting examples

```
@author: milenavalentini
"""

import numpy as np
import matplotlib.pyplot as mpl

x_vector_1 = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
y_vector_1 = np.array([1,2,3,4,5,6,7,8,9,10,11,12])

x_vector_2 = np.array([2,4,6,8,10,12])
y_vector_2 = np.array([1,1.5,4.1,3.5,4.5,5.5])

mpl.plot(x_vector_1, y_vector_1, color='black', marker='o', linestyle='dashed',
     linewidth=2, markersize=5, label='vectors_1')

mpl.plot(x_vector_2, y_vector_2, color='green', marker='d', linestyle='solid',
     linewidth=4, markersize=5, label='vectors_2')

mpl.errorbar(x_vector_1, y_vector_1, yerr=0.5, xerr=1, color='black')

mpl.errorbar(x_vector_2, y_vector_2, yerr=.75, color='green')

mpl.xlim(0,16)
mpl.ylim(0,13)

mpl.xlabel('x axis')
mpl.ylabel('y axis')

mpl.legend()

mpl.show()
mpl.savefig('a_first_plot.png')
```

Python: plotting examples

```
@author: milenavalentini
"""

import numpy as np
import matplotlib.pyplot as mpl

x_vector_1 = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
y_vector_1 = np.array([1,2,3,4,5,6,7,8,9,10,11,12])

x_vector_2 = np.array([2,4,6,8,10,12])
y_vector_2 = np.array([1,1.5,4.1,3.5,4.5,5.5])

mpl.plot(x_vector_1, y_vector_1, color='black', marker='o', linestyle='dashed',
    linewidth=2, markersize=5, label='vectors_1')

mpl.plot(x_vector_2, y_vector_2, color='green', marker='d', linestyle='solid',
    linewidth=4, markersize=5, label='vectors_2')

mpl.errorbar(x_vector_1, y_vector_1, yerr=0.5, xerr=1, color='black')

mpl.errorbar(x_vector_2, y_vector_2, yerr=.75, color='green')

mpl.xlim(0,16)
mpl.ylim(0,13)

mpl.xlabel('x axis')
mpl.ylabel('y axis')

mpl.legend()

mpl.show()
mpl.savefig('a_first_plot.png')
```

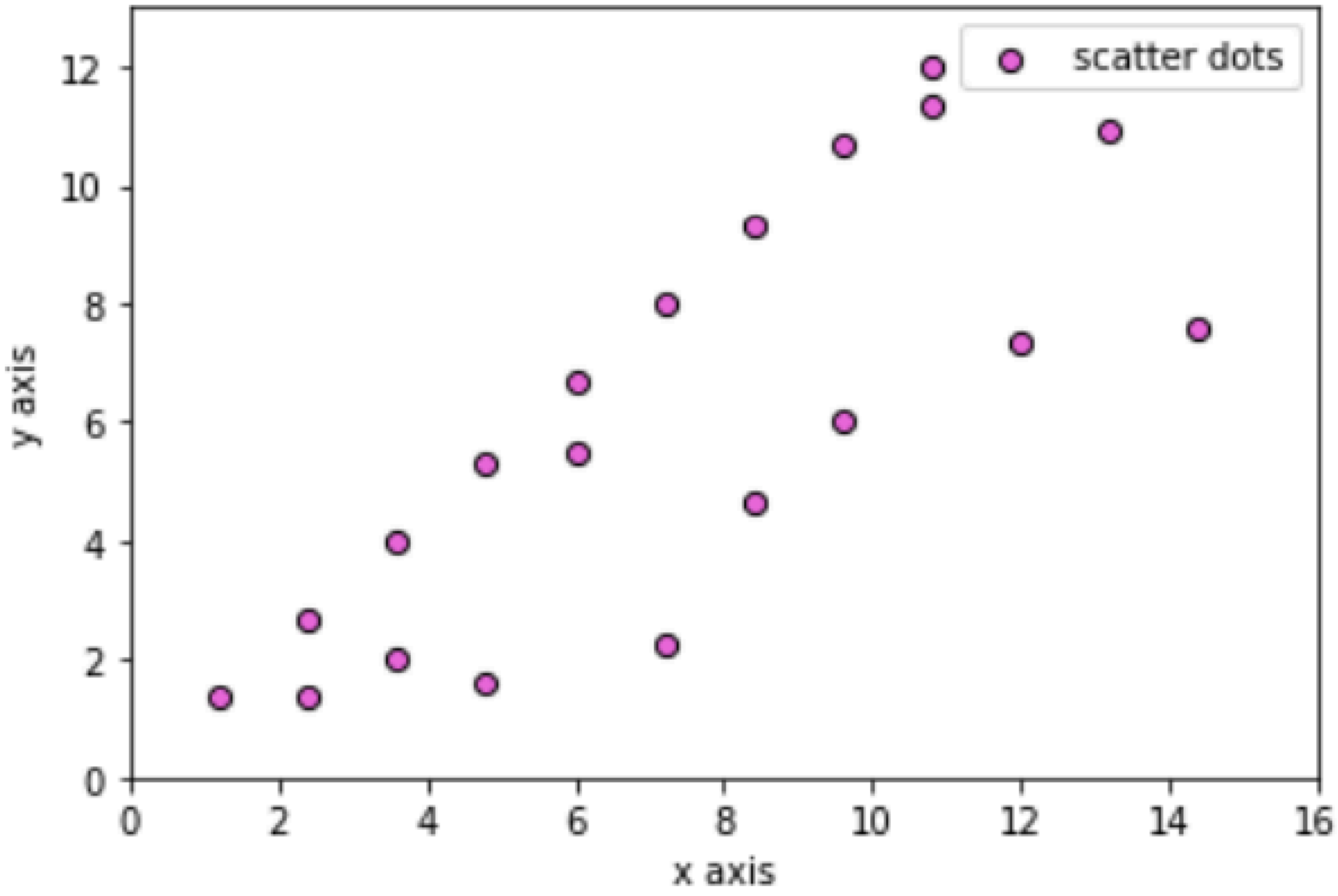


Python: plotting examples

```python
@author: milenavalentini
"""

import numpy as np
import matplotlib.pyplot as mpl

list_1 = [1,2,3,4,5,6,7,8,9,10,11,12]
list_2 = [1,2,3,4,5,6,7,8,9,10,11,12]

list_3 = [2,3,4,5,6,7,8,9,10,11,12]
list_4 = [1,1.5,1.2,4.1,1.7,3.5,4.5,8.5,5.5,8.2,5.7]

x_vector_1 = np.array(list_1 + list_3) * 1.2
y_vector_1 = np.array(list_2 + list_4) / 0.75

mpl.scatter(x_vector_1, y_vector_1, c='orchid', edgecolors='black', marker='.',
s=150, label='scatter dots')

mpl.xlim(0,16)
mpl.ylim(0,13)

mpl.xlabel('x axis')
mpl.ylabel('y axis')

mpl.legend()

mpl.show()
mpl.savefig('a_scatter_plot.png')
```

Python: plotting examples

```python
@author: milenavalentini
"""

import numpy as np
import matplotlib.pyplot as mpl

list_1 = [1,2,3,4,5,6,7,8,9,10,11,12]
list_2 = [1,2,3,4,5,6,7,8,9,10,11,12]

list_3 = [2,3,4,5,6,7,8,9,10,11,12]
list_4 = [1,1.5,1.2,4.1,1.7,3.5,4.5,8.5,5.5,8.2,5.7]

x_vector_1 = np.array(list_1 + list_3) * 1.2
y_vector_1 = np.array(list_2 + list_4) / 0.75

mpl.scatter(x_vector_1, y_vector_1, c='orchid', edgecolors='black', marker='.',
s=150, label='scatter dots')

mpl.xlim(0,16)
mpl.ylim(0,13)

mpl.xlabel('x axis')
mpl.ylabel('y axis')

mpl.legend()

mpl.show()
mpl.savefig('a_scatter_plot.png')
```



Python: plotting examples

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as sopt

#We generate random data, drawn from a gaussian centered at 2.5, standard deviation of 5.5
original_mu = 2.5
original_sigma = 5.5
my_data = np.random.normal(original_mu, original_sigma, 10000)

#We create an histogram of these data
#We do not use weights, but we ask for the probability density function to be returned. The integral of this curve will be 1.
#We use 100 bins, we could have defined bins ourselves
gauss_hist, bin_edges = np.histogram(my_data, bins = 100, density = True)

…

#Fit curve to data using scipy.optimize
#We need to define a function to use for the fit. In our case, we fit a gaussian
#I place the function here for example, but this should really go at the top of the script
#This function returns the probability density of a Gaussian curve, i.e. consistent with our 'observed' values (our histogram)
def gauss_for_fit(xvals, mu, sigma):

    p_of_x = (1./np.sqrt(2*np.pi*sigma**2))*np.exp(-(((xvals-mu)**2)/(2*sigma**2)))

    return p_of_x

#We perform the fit: scipy.optimize is only one of the many methods to do this!
fit_params, fit_covariance_matrix = sopt.curve_fit(gauss_for_fit, bin_centers, gauss_hist)
bestfit_mu, bestfit_sigma = fit_params

…
```

Python: plotting examples