Python libraries

(Courtesy of Filippo Aleotti, https://filippoaleotti.github.io/website/)

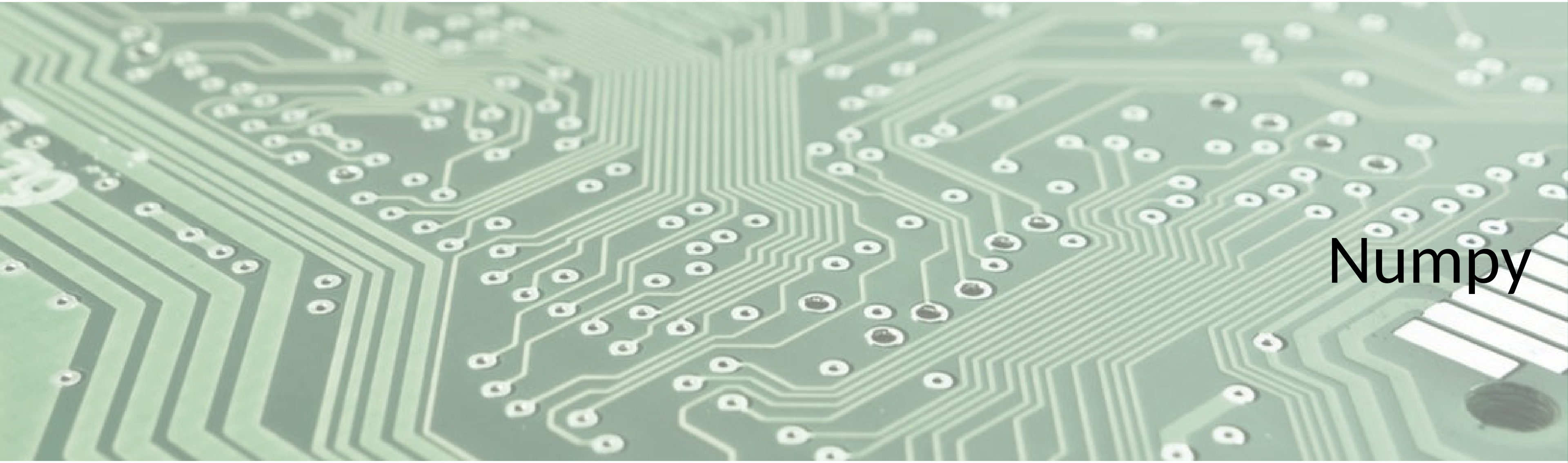
Digital Systems M, Module 2
Matteo Poggi, Università di Bologna

Python libraries

One of the key factors behind the raising popularity of Python is the availability of many **libraries** for some of the hottest computer science topics, such as computer vision and machine learning

Some examples: NumPy, Matplotlib, OpenCV, Scikit-learn, etc.

In this lesson, we will introduce some of the basic libraries to deal with multi-dimensional data (images and more)



Numpy

Python libraries

NumPy is a scientific computation package

It offers many functions and utilities to work with N-Dimension arrays

Largely used by other libraries such as OpenCV, TensorFlow and PyTorch to deal with multi dimensional arrays (e.g., tensors or images)

Python libraries

How to install

We can easily install NumPy using pip by running

pip install numpy

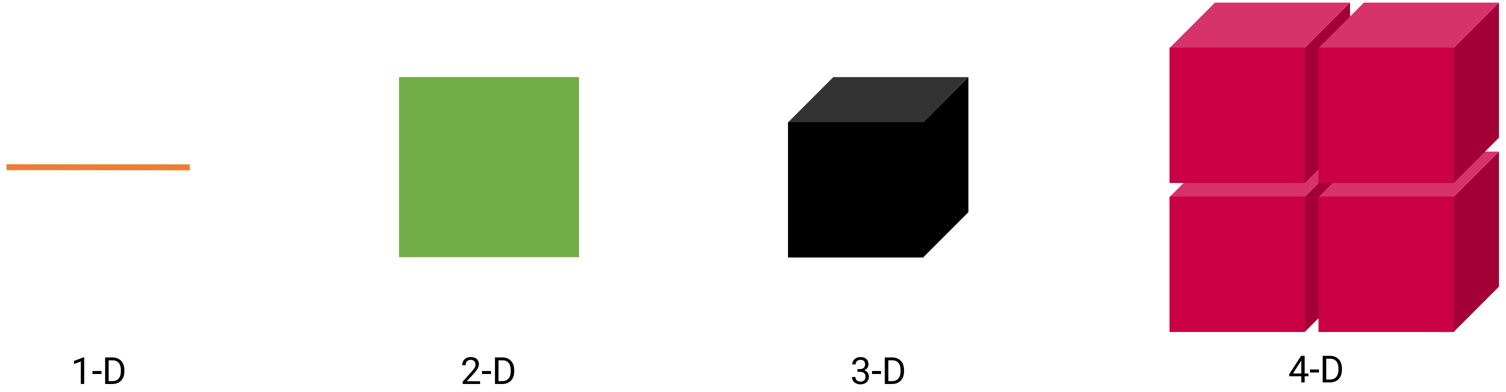
Then, we can import it in our python script

```
import numpy as np
```

Python libraries

N-dimensional arrays

Numpy supports N-dimensional array such as, for instance:



Python libraries

In order to create an array, we can use the **array** function, passing a list of values and optionally the type of data

```
# create an array from a list of values
list_of_values = [20.,2.,5.]
x = np.array(list_of_values)

more_values = [[[20],[2],[5]]]
y = np.array(more_values, dtype=np.int32)
```

NOTE: NumPy arrays must be *homogeneous* (all elements have the same type)

NOTE: if the type is not set, NumPy will set it to *float64* as default

Python libraries

NumPy offers standard functions to easily create arrays. Some of them are **ones**, **zeros**, **ones_like**, **zeros_like** and **eye**, but many more exist

We use the functions **zeros** and **ones** to create an array with given shapes in which each element is, respectively, 0 or 1

```
# create a int32 array of zeros with shape (20,2)
zeros = np.zeros((20,2), dtype=np.int32)

# create a float32 array of ones with shape (5,2,1)
ones = np.ones((5,2,1), dtype=np.float32)
```

Python libraries

Given a NumPy array with a certain shape, **zeros_like** and **ones_like** allow to create a 0 and 1 arrays with the same shape

```
x = np.array([5,5])
zeros = np.zeros_like(x) # [0 0]

ones = np.ones_like(x, dtype=np.float32) # [1. 1.]
```

Python libraries

Using the function arange(**start**, stop, step), we obtain a NumPy array containing all elements from **start** to stop, using a step spacing consecutive elements

```
# create an array containing [0,1,2,3,4]
x = np.arange(5)

# create an array containing [2,3,4]
y = np.arange(2,5)

# create an array containing [2,4]
z = np.arange(2,5,2)
```

Notice that the values are generated within the half-open set [**start**, stop[, so stop is not included

Python libraries

With **eye** we create an *identity* matrix, so a matrix full of zeros except for the diagonal

```
x = np.eye(5)
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

Sometimes we need random values. We can obtain an array filled with random values calling the **rand** function

```
# create an random matrix 3x5
x = np.random.rand(3,5)
```

Python libraries

Finally, we can create an array that contains a single scalar value using **full**

```
# create an matrix 2x2 matrix in which each element is 7
x = np.full((2,2), 7)
```

We can obtain the same result using **ones**/**ones_like** function

```
# create an 2x2 matrix in which each element is 7
x = np.ones((2,2)) * 7
```

Python libraries

Each array has got **attributes**, such as dtype or shape**.** Attributes contain important information related to that particular array

dtype allows to know the type of the array

```
x = np.array([2., 5., 3.])
y = np.array([2., 5., 3.], dtype=np.int32)

print(x.dtype) # float64
print(y.dtype) # int32
```

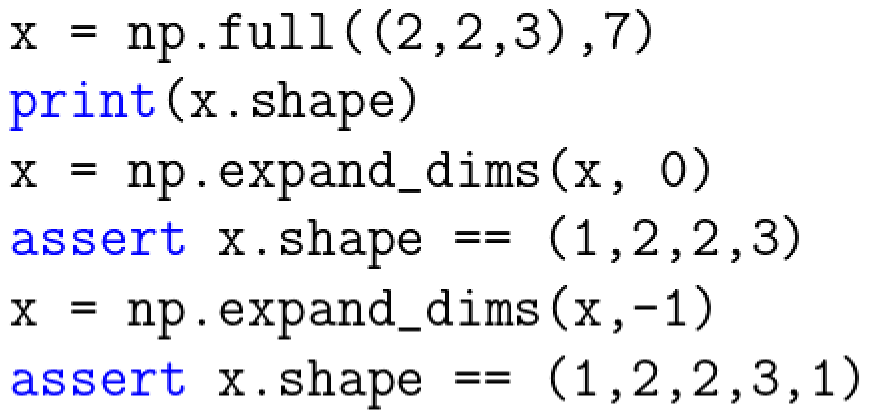
Python libraries

shape give you back the size of the array along each dimension

```
x = np.array([2., 5., 3.])
y = np.ones((2,4,1,2,3))

print(x.shape) # (3,)
print(y.shape) # (2, 4, 1, 2, 3)
```

Python libraries

Given an array, we can add a new dimension using **expand_dims**



```
x = np.full((2,2,3),7)
print(x.shape)
x = np.expand_dims(x, 0)
assert x.shape == (1,2,2,3)
x = np.expand_dims(x,-1)
assert x.shape == (1,2,2,3,1)
```

Python libraries

Using **squeeze**, we can remove all the single dimensional entries of the array

```
x = np.full((20,1,1), 5)
y = np.squeeze(x)
assert y.shape == (20,)
```

However, squeeze allows also to specify the axis to delete (**scalar**, **tuple** or **None**. Default is **None**)

```
x = np.full((20,1,1), 5)
y = np.squeeze(x, axis=1)
assert y.shape == (20,1)
```

Python libraries

A common operation consist in changing the **reshaping** a given array. For instance, we can turn a 10 elements array into a 2x5 using the **reshape** function

```
x = np.arange((10)) # [0 1 2 3 4 5 6 7 8 9]
y = np.reshape(x, (2,5)) #[[0 1 2 3 4],[5 6 7 8 9]]
assert y.shape == (2,5)
```

Given a partial shape completed by **-1**, NumPy can complete it

```
x = np.arange((20))
y = np.reshape(x, (2,-1,2))
assert y.shape == (2,5,2)
```

Python libraries

This operation works if a single dimension is missing. otherwise, if more dimensions are unknown, NumPy will throw **ValueError**

```
x = np.arange((20))
y = np.reshape(x, (2,-1,-1)) # ValueError: can only specify one unknown dimension
```

Python libraries

Given an array, you can access to its elements by index notation

```
# get the 10th element of the array
x = np.arange(20)
element = x[10] #10
```

Elements can be retrieved also using **item** function

```
# get the 10th element of the array
x = np.arange(20)
element = x.item(10) #10
```

Python libraries

Slice notation (the same used for python strings) is valid also for arrays

```
# get elements with index in [10,15[
x = np.arange(20)
element = x[10:15] #[10,11,12,13,14]

# get elements with index in [10: len(array)-7[
more_elements = x[10:-7] #[10 11 12]
assert np.array_equal(more_elements, x[10:13])

# get every element whose index is multiple of 3, starting from index 0
array = x[::3] #[0 3 6 9 12 15 18]
```

Python libraries

Given two array, we can concatenate them together to obtain a single array as output thanks to the **concatenate** function

```
x = np.full((5,2), 3)
y = np.full((5,1), 4)
z = np.concatenate([x,y], axis=-1)
print(z) #[[3 3 4],[3 3 4],[3 3 4],[3 3 4],[3 3 4]]
assert z.shape == (5,3)
```

Python libraries

NumPy math

NumPy is a scientific package and thus it offers both simple and complex math functions that we can apply to arrays.

In the following, we are going to see some of them

Python libraries

Given two arrays, we can sum or subtract them just using **+** and **-** operators

```
x = np.full((4,2,3), 8) # 4x2x3 array, full of 7
y = np.ones_like(x) # 4x2x3 array, full of 1

# sum two arrays
array_sum = x + y
assert np.array_equal(array_sum, np.ones_like(x)*9)

# subtract two arrays
array_sub = x - y
assert np.array_equal(array_sub, np.ones_like(x)*7)
```

In this case, both arrays have the same shape, so the operations are performed **element-wise**

Python libraries

Sometimes, our arrays have not the same shape, but NumPy is smart enough and try to “fit” the arrays. This operation is called **broadcasting**

```
x = np.full((4,2,3), 8) # 4x2x3 array, full of 7
y = 1

# sum two arrays
array_sum = x + y
assert np.array_equal(array_sum, np.ones_like(x)*9)

# subtract two arrays
array_sub = x - y
assert np.array_equal(array_sub, np.ones_like(x)*7)
```

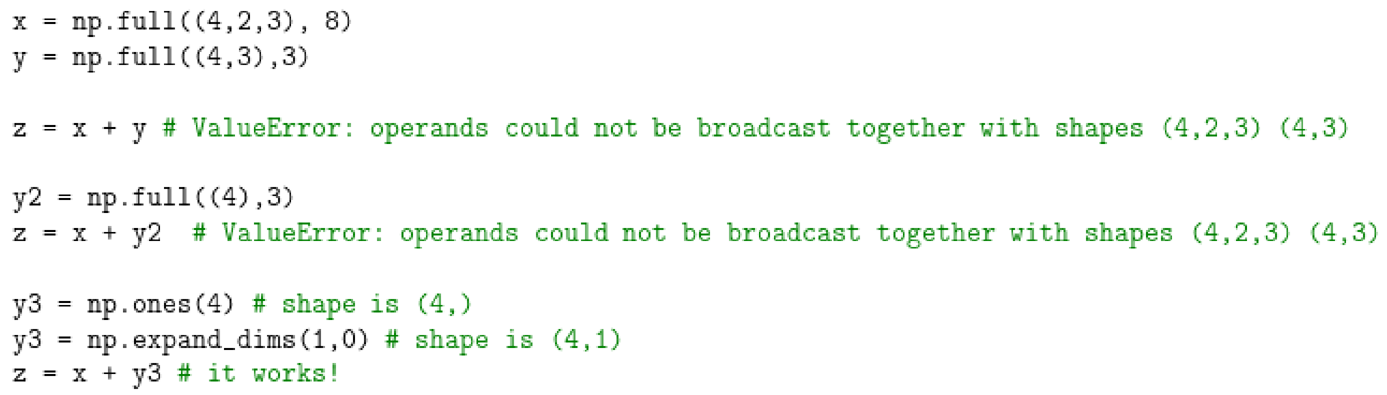
Notice that **y** is a scalar, but both **array_sum** and **array_sub** have shapes (4,2,3)

Python libraries

Broadcasting does not work for all the cases.

Two shapes are eligible for broadcasting if both have the same dimensions or if they differ by a single dimension, that is **1** for one of the two shapes. The final shape will have the maximum of corresponding dimensions of the two.

If broadcasting can't be applied, **ValueError** would be raised



```
x = np.full((4,2,3), 8)
y = np.full((4,3),3)

z = x + y # ValueError: operands could not be broadcast together with shapes (4,2,3) (4,3)

y2 = np.full((4),3)
z = x + y2  # ValueError: operands could not be broadcast together with shapes (4,2,3) (4,3)

y3 = np.ones(4) # shape is (4,)
y3 = np.expand_dims(1,0) # shape is (4,1)
z = x + y3 # it works!
```

Python libraries

Given two NumPy arrays, we can perform **element-wise** multiplication using * or **multiply** function

```
x = np.full((4,2,3), 8) # 4x2x3 array, full of 8
y = np.ones_like(x)*2 # 4x2x3 array, full of 2

# multiplicate element-wise two arrays
mul = x * y
assert np.array_equal(mul, np.ones_like(x)*16)

# subtract two arrays
mul2 = np.multiply(x, y)
assert np.array_equal(mul2, np.ones_like(x)*16)
```

Python libraries

Given two NumPy arrays, we can perform **matrix** multiplication using **matmul** function

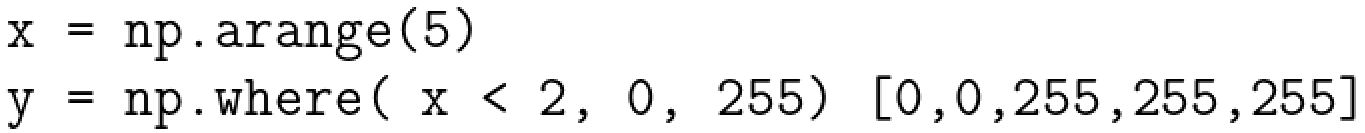
```
x1 = np.full((4,2,3), 8) # 4x2x3 array, full of 8
x2 = np.full((3,3), 7) # 3x3 array, full of 7
y = np.eye(3) # 3x3 diagonal array

# matrix multiplication of two arrays
mul = np.matmul(x1,y)
assert np.array_equal(mul, x1)

# matrix multiplication
mul = np.matmul(x2,y)
assert np.array_equal(mul, x2)
```

For N-Dimensional arrays (N>2), **matmul** applies broadcasting, treating the array as a stack of matrices

Python libraries

Conditions

We can use **where** function to apply a condition.Given an input array, a condition and two arrays x and y, **for each element** we sample from x if the condition is verified, from y otherwise



```
x = np.arange(5)
y = np.where( x < 2, 0, 255) [0,0,255,255,255]
```

Python libraries

NumPy is quite **optimized**, so when possible try to used “**native**” NumPy way instead of much more traditional programming paradigms (e.g. loops)

```
x = np.random.rand(4,640,480,3)
batch, height, width, channels = x.shape
start = time()
y = np.ones_like(x)
for b in range(batch):
    for h in range(height):
        for w in range(width):
            for c in range(channels):
                y[b,h,w,c] = 0 if x[b,h,w,c] < 0.05 else 255
duration = time() - start # 4.982 sec
```

```
x = np.random.rand(4,640,480,3)
start = time()
y = np.where( x < 0.05, 0, 255)
duration = time() - start # 0.0283 sec
```

Python libraries

NumPy provides a set of functions to write and read directly from the filesystem

Plain text (.txt) and csv (.csv) can be loaded using **loadtxt** function, providing the path to the file, the data type and the delimiter

```
data = np.loadtxt('your_file.txt',dtype=np.float32, delimiter=',')
```

Python libraries

Input/output

NumPy provides a set of functions to write and read directly from the filesystem

Plain text (.txt) and csv (.csv) can be loaded using **loadtxt** function, providing the path to the file, the data type and the delimiter

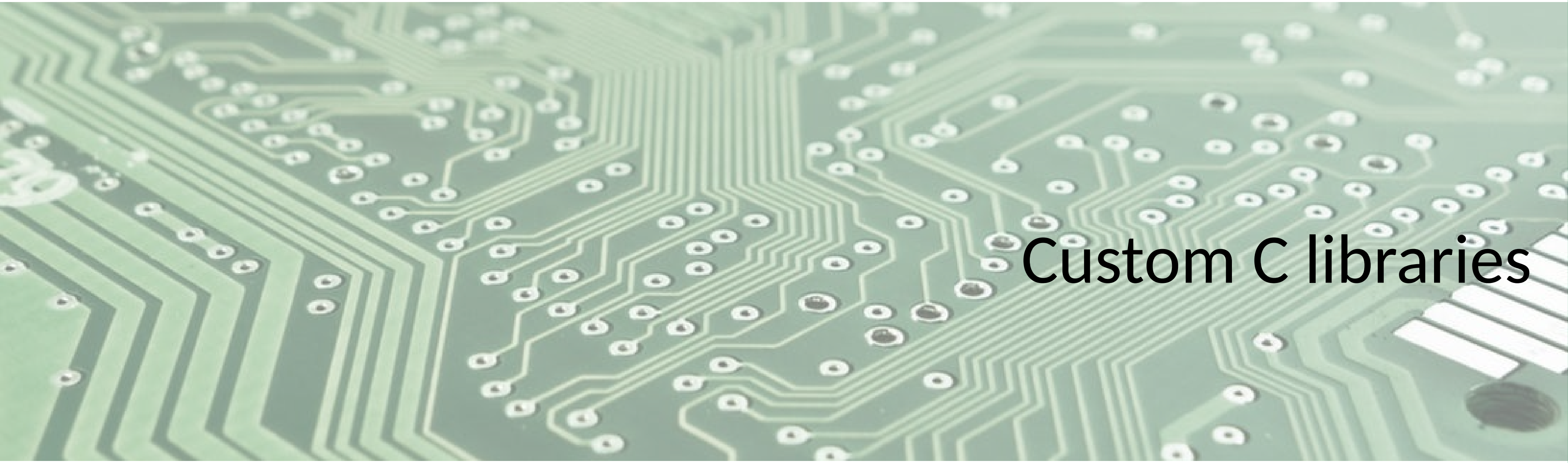
```
data = np.loadtxt('your_file.txt',dtype=np.float32, delimiter=',')
```

Python libraries

We can store NumPy arrays in two ways:

- binary file: using **np.save** we are able to serialize our arrays in the local file system. We will obtain a **.npy** file containing the array
- txt: using **savetxt** function we will store our 1D or 2D array in a new txt

Finally, **.npy** files can be read using **np.load** function



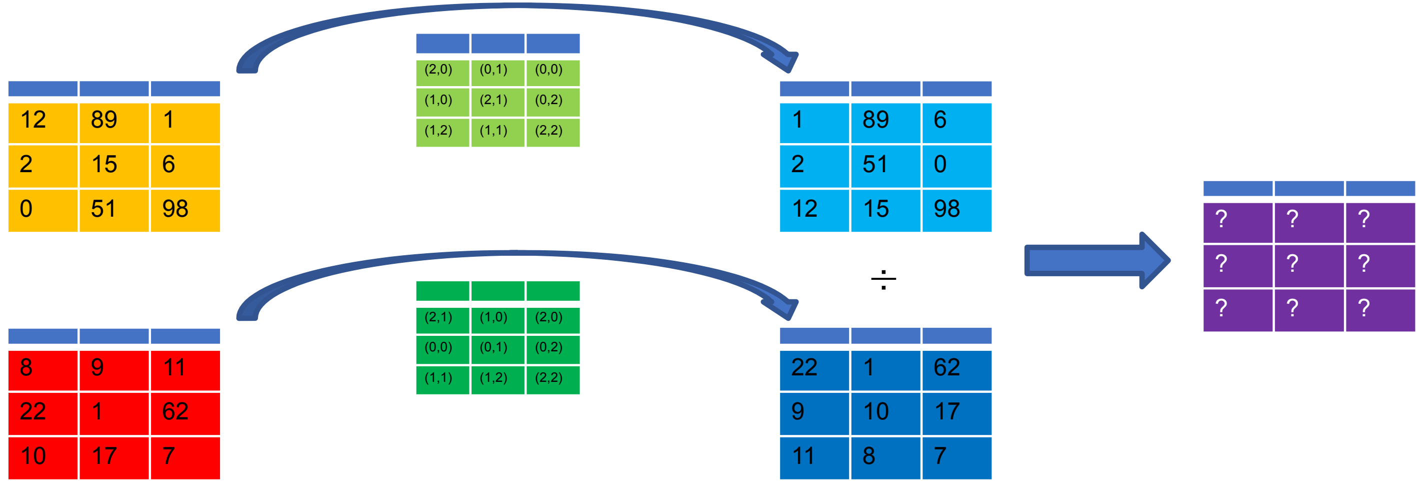
Custom C libraries

Python libraries

Numpy support to matrices operations is great!

… but what about some more complex functions?

For instance, we want to assign the elements of a matrix A_1 to a matrix C_1 at the coordinates stored in a tensor B_1, the elements of A_2 to C_2 at coordinates B_2 and finally compute an element-wise division between C_1 and C_2



12	89	1
2	15	6
0	51	98

(2,0)	(0,1)	(0,0)
(1,0)	(2,1)	(0,2)
(1,2)	(1,1)	(2,2)

1	89	6
2	51	0
12	15	98

÷

8	9	11
22	1	62
10	17	7

(2,1)	(1,0)	(2,0)
(0,0)	(0,1)	(0,2)
(1,1)	(1,2)	(2,2)

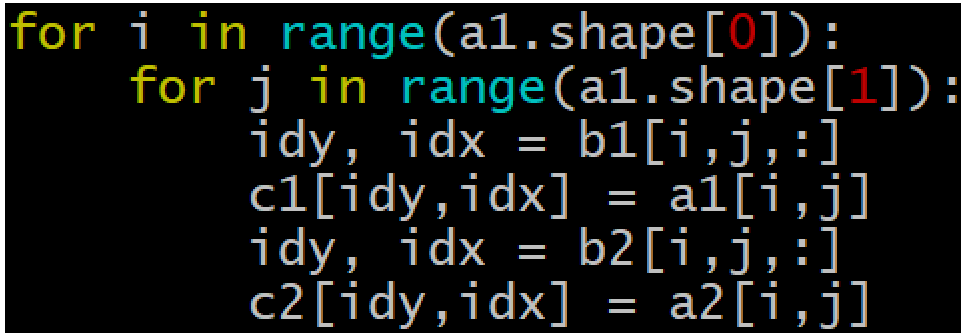
22	1	62
9	10	17
11	8	7

?	?	?
?	?	?
?	?	?

Python libraries

How to do it in Numpy?

No built-in function allows for it, then let's go over a **for loop**



```
for i in range(a1.shape[0]):
    for j in range(a1.shape[1]):
        idy, idx = b1[i,j,:]
        c1[idy,idx] = a1[i,j]
        idy, idx = b2[i,j,:]
        c2[idy,idx] = a2[i,j]
```

Everything is fine for 3x3 matrices…
… but what happens when our matrices become **bigger**?

Python libraries

If our custom function is designed to run images, on 0.3 Mpx frames it would take about **1.17 seconds** on Intel i7 processor to run.

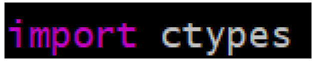
On 4K images, it would take about **20 seconds**.

The very same function, implemented in standard C, would take respectively 0.005 and 0.10 seconds (about **200x** speed up)

Python libraries

The **ctypes** library provides C-compatible data types and allows calling functions in DLLs or shared libraries.

It can be used to wrap these libraries in **pure Python**.

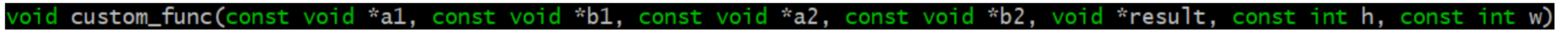


```
import ctypes
```

We can implement our function in standard C and call it inside our Python script with *few lines* of extra code

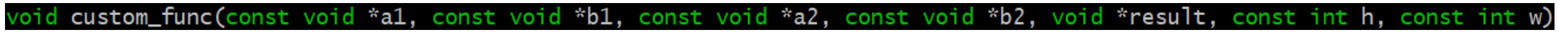
Python libraries

Let’s start by creating a **digitali.c** source file and defining our function by means of its signature



```c
void custom_func(const void *a1, const void *b1, const void *a2, const void *b2, void *result, const int h, const int w)
```

Python libraries

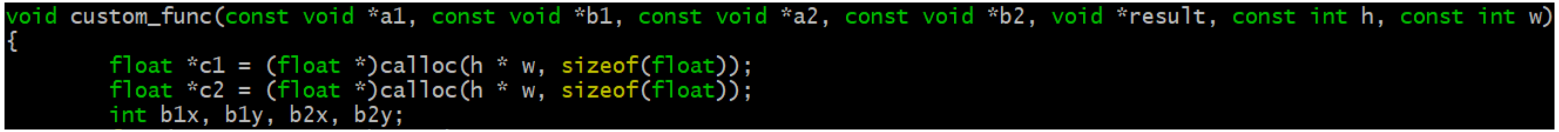


```
void custom_func(const void *a1, const void *b1, const void *a2, const void *b2, void *result, const int h, const int w)
```

We pass A_1, A_2, B_1 and B_2 to our function through **a1**, **b1**, **a2** and **b2**. Since they are (multi-dimensional) arrays, we pass them by their **references.**

Our matrices a1 and a2 have both dimensions **h** x **w**, passed as inputs to the function, while the index matrices b1 and b2 have dimensions **h** x **w x 2**

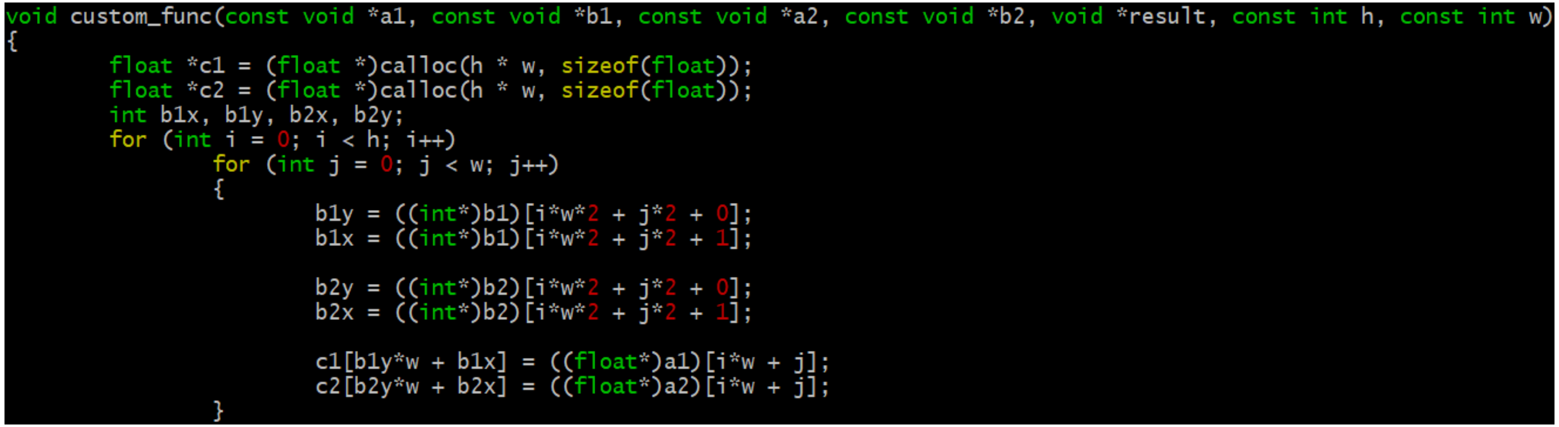
Python libraries



```
void custom_func(const void *a1, const void *b1, const void *a2, const void *b2, void *result, const int h, const int w)
{
        float *c1 = (float *)calloc(h * w, sizeof(float));
        float *c2 = (float *)calloc(h * w, sizeof(float));
        int b1x, b1y, b2x, b2y;
```

Then, we allocate two intermediate data structures **c1** and **c2** respectively for C_1 and C_2, where we store the values retrieved from **a1** and **a2**

Python libraries

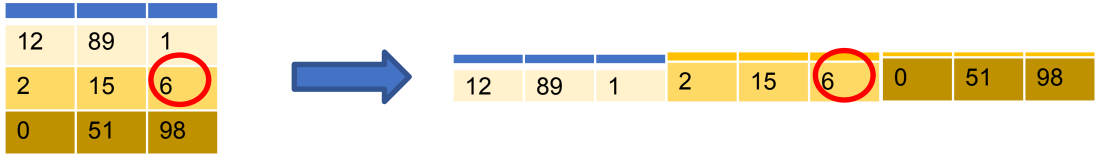


```
void custom_func(const void *a1, const void *b1, const void *a2, const void *b2, void *result, const int h, const int w)
{
        float *c1 = (float *)calloc(h * w, sizeof(float));
        float *c2 = (float *)calloc(h * w, sizeof(float));
        int b1x, b1y, b2x, b2y;
        for (int i = 0; i < h; i++)
                for (int j = 0; j < w; j++)
                {
                        b1y = ((int*)b1)[i*w*2 + j*2 + 0];
                        b1x = ((int*)b1)[i*w*2 + j*2 + 1];

                        b2y = ((int*)b2)[i*w*2 + j*2 + 0];
                        b2x = ((int*)b2)[i*w*2 + j*2 + 1];

                        c1[b1y*w + b1x] = ((float*)a1)[i*w + j];
                        c2[b2y*w + b2x] = ((float*)a2)[i*w + j];
                }
```

Now, we can loop.
We retrieve x,y coordinates from **b1** and **b2**, then we use them to assign **a1** and **a2** values to **c1** and **c2**

Python libraries

N-dimensional arrays are stored in **flatten** form, e.g. a 2D matrix made of H rows and W columns is stored as a 1D vector of size HxW. This is common to many libraries (e.g., C/C++ OpenCV)



Every time we access to a data structure using N-dimensional indexing, it is converted to a 1D index. For a 2D matrix, accessing to coordinates [Y,X] means means accessing to cell **[YxW + X]**

$A_1[1,2] = A_1.\text{data}[1x3+2] = A_1.\text{data}[5]$

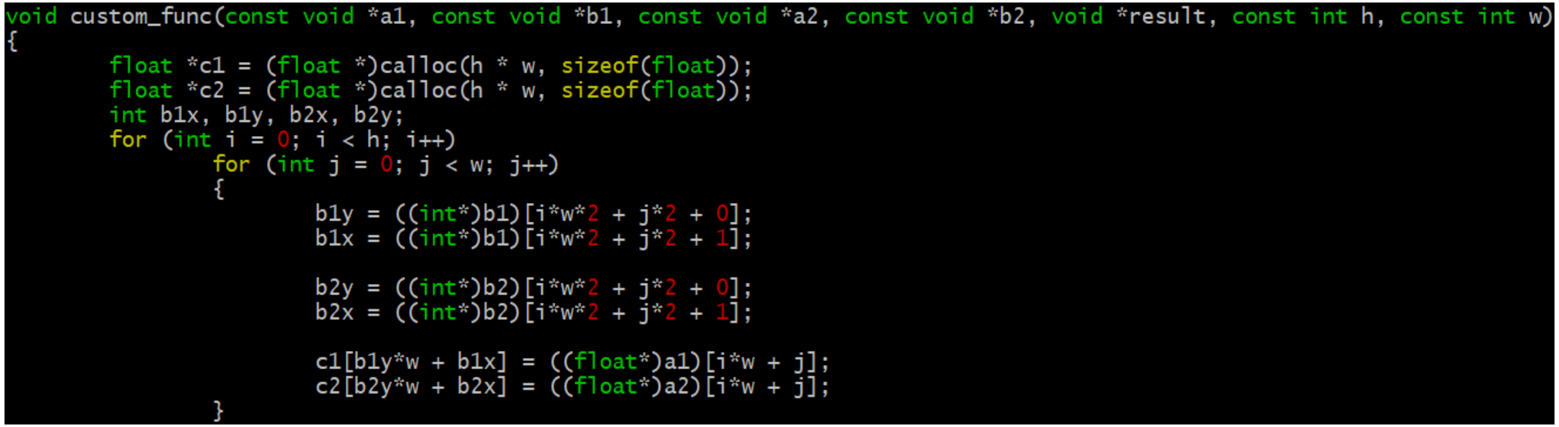
Python libraries

In case of a 3-dimensional array of shape DxHxW, indices **[Z,Y,X]** will be converted into **[ZxHxW + YxW + X]**

In general, when accessing to N-dimensional arrays of shape

coordinates $[C_0,C_1,\ldots,C_{N-1},C_N]$ will be converted to

Python libraries

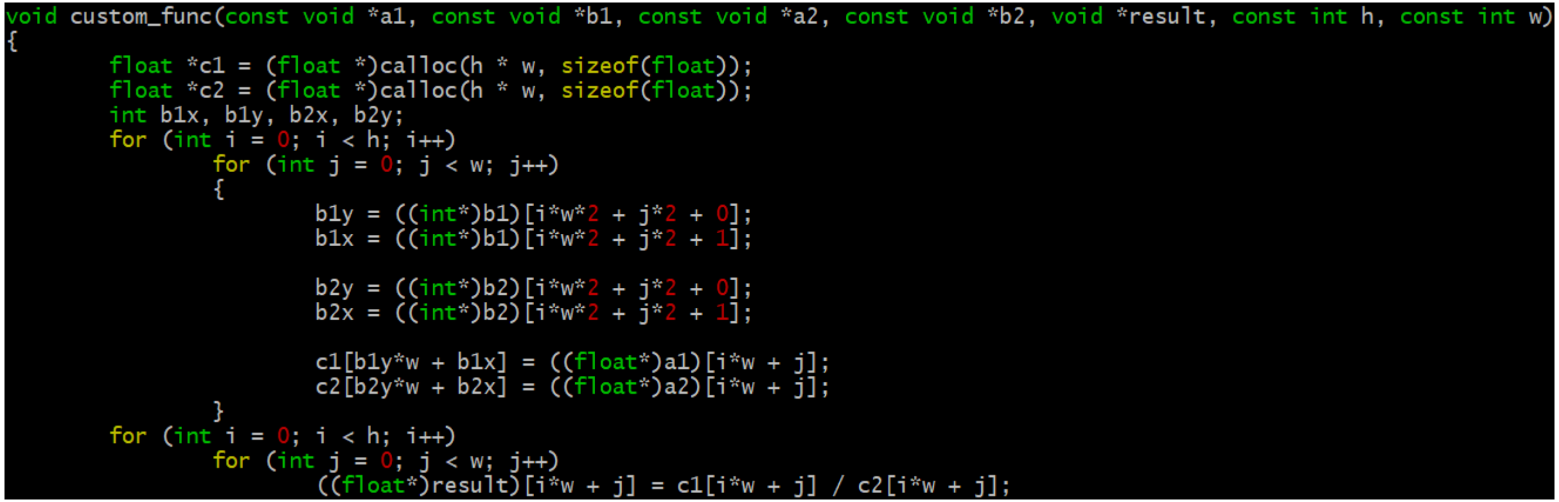


```
void custom_func(const void *a1, const void *b1, const void *a2, const void *b2, void *result, const int h, const int w)
{
    float *c1 = (float *)calloc(h * w, sizeof(float));
    float *c2 = (float *)calloc(h * w, sizeof(float));
    int b1x, b1y, b2x, b2y;
    for (int i = 0; i < h; i++)
        for (int j = 0; j < w; j++)
        {
            b1y = ((int*)b1)[i*w*2 + j*2 + 0];
            b1x = ((int*)b1)[i*w*2 + j*2 + 1];

            b2y = ((int*)b2)[i*w*2 + j*2 + 0];
            b2x = ((int*)b2)[i*w*2 + j*2 + 1];

            c1[b1y*w + b1x] = ((float*)a1)[i*w + j];
            c2[b2y*w + b2x] = ((float*)a2)[i*w + j];
        }
```

Let's go back to our function.
In our case, **a1** and **a2** are 2D matrices, thus [i,j] becomes [i*w+j], while **b1** and **b2** are 3D matrices (HxWx2), thus [i,j,0] becomes [i*w*2+j*2+0].

Finally, **c1** and **c2** are 2D matrices (allocated in flatten form)

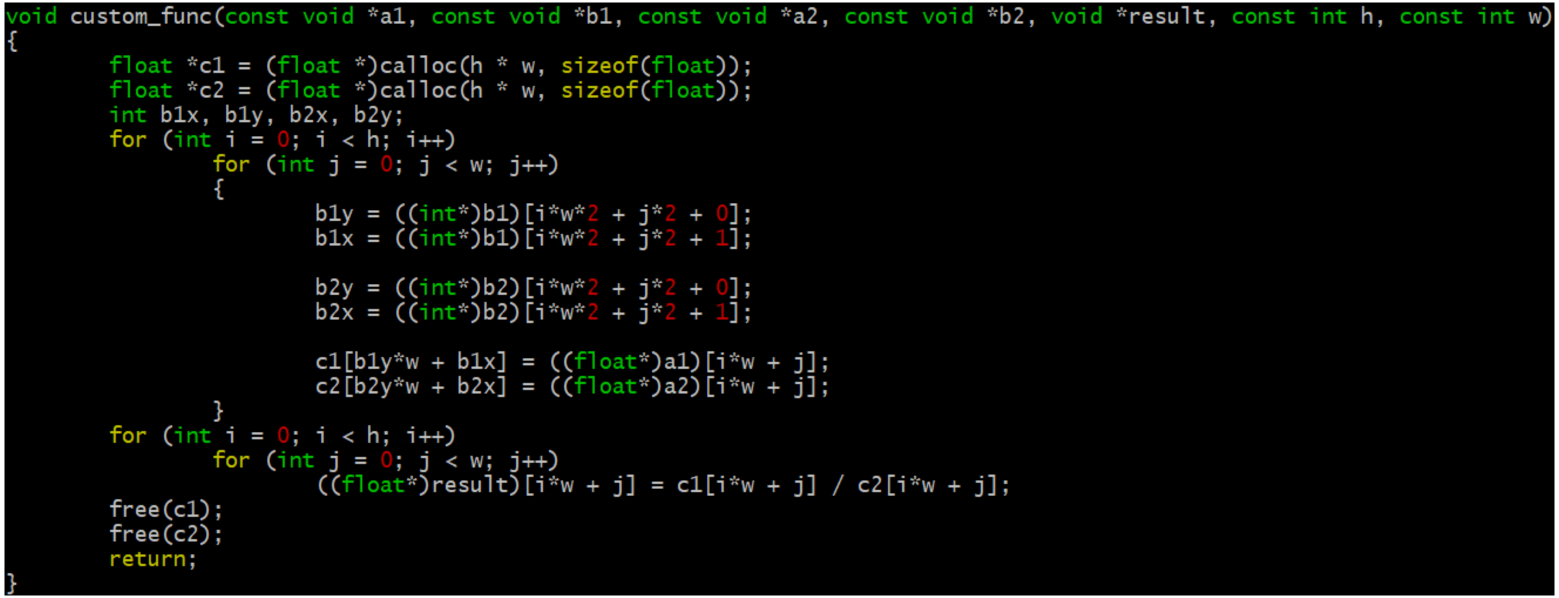
Python libraries



```
void custom_func(const void *a1, const void *b1, const void *a2, const void *b2, void *result, const int h, const int w)
{
        float *c1 = (float *)calloc(h * w, sizeof(float));
        float *c2 = (float *)calloc(h * w, sizeof(float));
        int b1x, b1y, b2x, b2y;
        for (int i = 0; i < h; i++)
                for (int j = 0; j < w; j++)
                {
                        b1y = ((int*)b1)[i*w*2 + j*2 + 0];
                        b1x = ((int*)b1)[i*w*2 + j*2 + 1];

                        b2y = ((int*)b2)[i*w*2 + j*2 + 0];
                        b2x = ((int*)b2)[i*w*2 + j*2 + 1];

                        c1[b1y*w + b1x] = ((float*)a1)[i*w + j];
                        c2[b2y*w + b2x] = ((float*)a2)[i*w + j];
                }
        for (int i = 0; i < h; i++)
                for (int j = 0; j < w; j++)
                        ((float*)result)[i*w + j] = c1[i*w + j] / c2[i*w + j];
```

Once **c1** and **c2** have been filled, we can get the final result as **c1/c2**

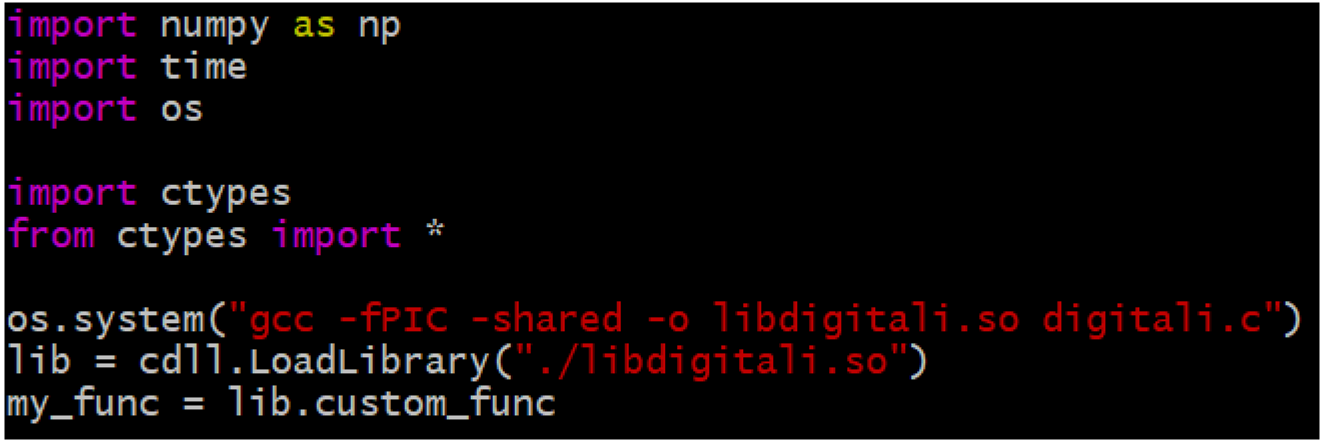
Python libraries



```c
void custom_func(const void *a1, const void *b1, const void *a2, const void *b2, void *result, const int h, const int w)
{
        float *c1 = (float *)calloc(h * w, sizeof(float));
        float *c2 = (float *)calloc(h * w, sizeof(float));
        int b1x, b1y, b2x, b2y;
        for (int i = 0; i < h; i++)
                for (int j = 0; j < w; j++)
                {
                        b1y = ((int*)b1)[i*w*2 + j*2 + 0];
                        b1x = ((int*)b1)[i*w*2 + j*2 + 1];

                        b2y = ((int*)b2)[i*w*2 + j*2 + 0];
                        b2x = ((int*)b2)[i*w*2 + j*2 + 1];

                        c1[b1y*w + b1x] = ((float*)a1)[i*w + j];
                        c2[b2y*w + b2x] = ((float*)a2)[i*w + j];
                }
        for (int i = 0; i < h; i++)
                for (int j = 0; j < w; j++)
                        ((float*)result)[i*w + j] = c1[i*w + j] / c2[i*w + j];
        free(c1);
        free(c2);
        return;
}
```

Before returning, remember to **free memory!**

Python libraries

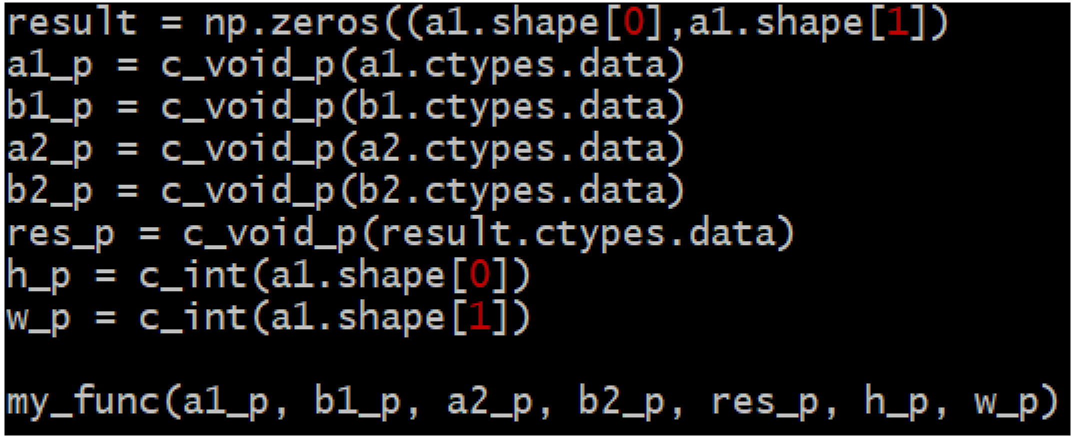


```
import numpy as np
import time
import os

import ctypes
from ctypes import *

os.system("gcc -fPIC -shared -o libdigitali.so digitali.c")
lib = cdll.LoadLibrary("./libdigitali.so")
my_func = lib.custom_func
```

Let’s go to the Python side.
First, we need to import ctypes and load the library. Then, we can access to the function itself

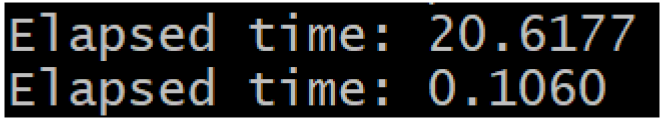
We can optionally compile C source files using **os.system()** if we did not yet

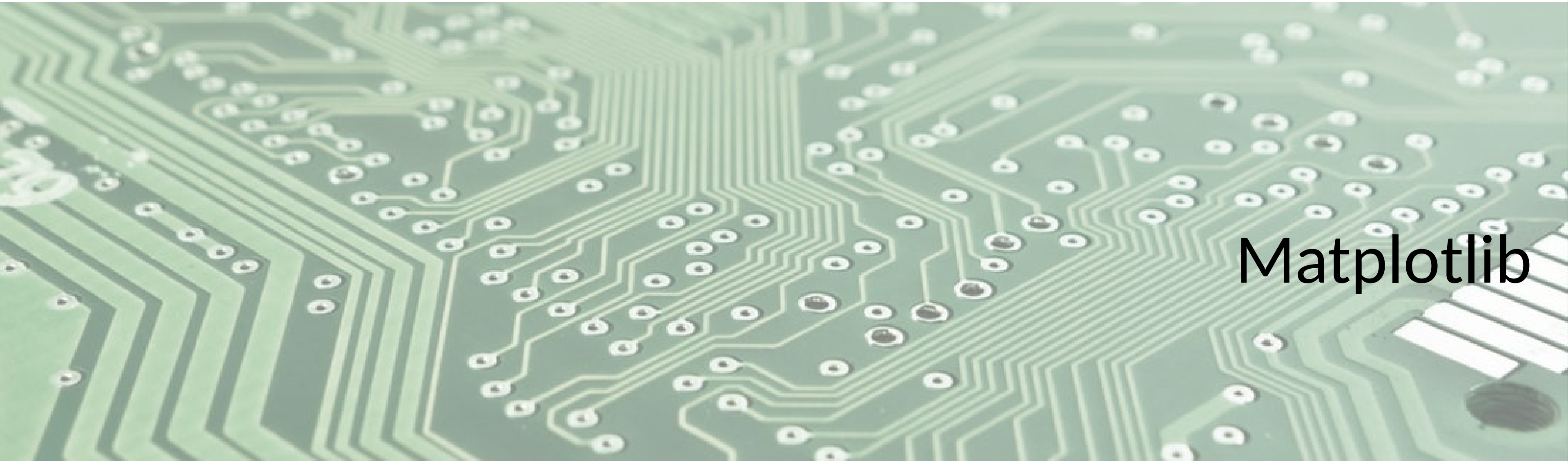
Python libraries


```
result = np.zeros((a1.shape[0],a1.shape[1])
a1_p = c_void_p(a1.ctypes.data)
b1_p = c_void_p(b1.ctypes.data)
a2_p = c_void_p(a2.ctypes.data)
b2_p = c_void_p(b2.ctypes.data)
res_p = c_void_p(result.ctypes.data)
h_p = c_int(a1.shape[0])
w_p = c_int(a1.shape[1])

my_func(a1_p, b1_p, a2_p, b2_p, res_p, h_p, w_p)
```

Before calling for our function, we need to cast our data to ctypes.

Finally, we can *transparently* call the function!


```
Elapsed time: 20.6177
Elapsed time: 0.1060
```



Matplotlib

Python libraries

Sometimes, we want to display values in a human readable form.

Suppose for instance you have got temperature values collected by a sensor, one measurement per hour for a week. Display such values in a chart would simplify largely the reading!

We can visualize NumPy arrays using some chart libraries, like Matplotlib

Python libraries

Matplotlib is an open source plotting library able to produce high quality graphs and charts. Easy to install using pip (just “*pip install matplotlib*”)

It offers a large set of plot types (e.g., histogram, scatter, line, 3D and more), and uses NumPy arrays to handle data

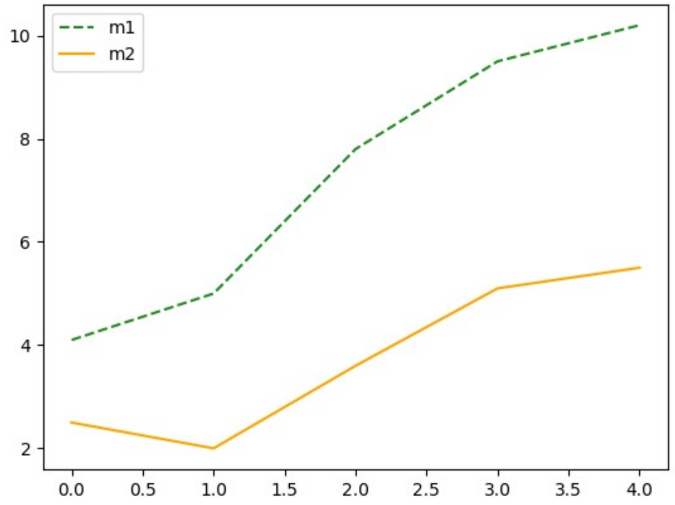
It can run also on interactive environments such as Jupyter

Python libraries

Plots

Given two collection of values, **m1** and **m2**, we can visualize them using Matplotlib

```python
import numpy as np
import matplotlib.pyplot as plt

labels = ['7', '8', '9', '10', '11']
x = np.arange(5)
m1 = np.array([4.1,5.0,7.8,9.5,10.2])
m2 = np.array([2.5,2.,3.6,5.1,5.5])

plt.plot(x,m1, color='forestgreen',label='m1', linestyle='dashed')
plt.plot(x,m2, color='orange', label='m2')
plt.legend()
plt.savefig('chart.png')
```

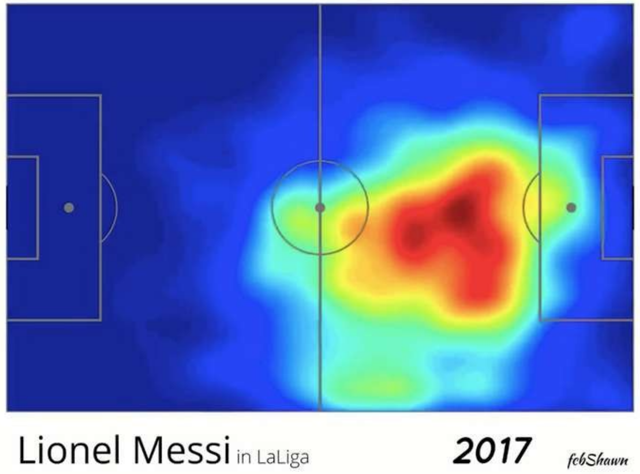


Python libraries

Colormaps

Let’s now consider more complex data.
For instance, we want to study in which part of the football field a player (e.g. Lionel Messi) spent most of its time in a specific season (e.g., 2017)

In this case, we deal with a 2D data structure (field coordinates) and the value we want to plot (player presence). An intuitive way to visualize the players’ behavior is to create an **heatmap**

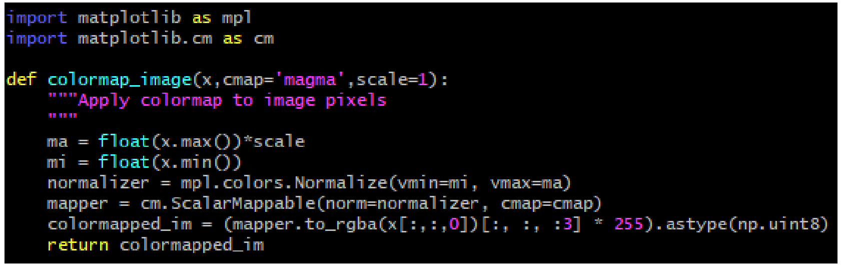


Source: www.givemesport.com

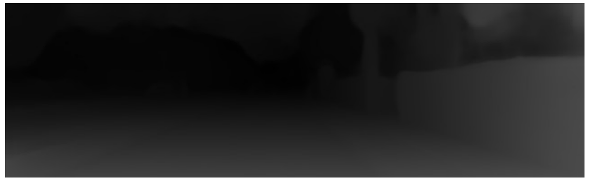
Python libraries

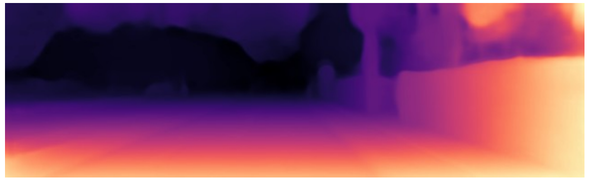
Another example concerns with **depth maps**, that are images in which each pixel encodes the distance of a 3D point from the camera.

Applying a **colormap** can greatly ease the understanding of depth maps



```
import matplotlib as mpl
import matplotlib.cm as cm

def colormap_image(x,cmap='magma',scale=1):
    """Apply colormap to image pixels
    """
    ma = float(x.max())*scale
    mi = float(x.min())
    normalizer = mpl.colors.Normalize(vmin=mi, vmax=ma)
    mapper = cm.ScalarMappable(norm=normalizer, cmap=cmap)
    colormapped_im = (mapper.to_rgba(x[:,:,0])[:, :, :3] * 255).astype(np.uint8)
    return colormapped_im
```

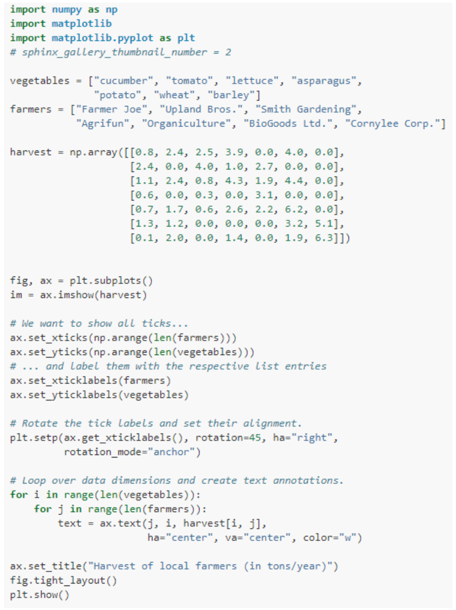




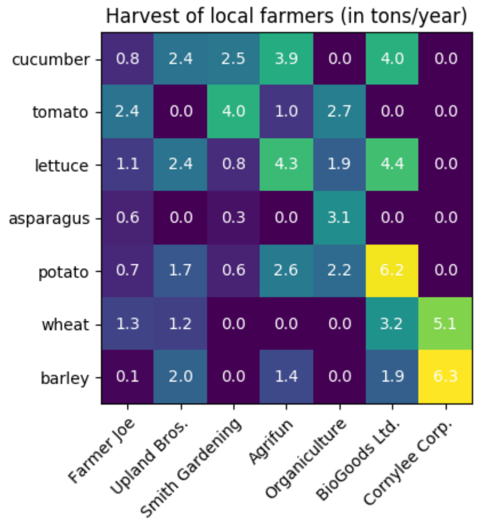
Python libraries

Categorical heatmap

Let’s consider a set of farmers, a set of vegetables and a 2D array, **harvest**, collecting the quantity of each vegetable harvested by each of the farmers.



```python
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
# sphinx_gallery_thumbnail_number = 2

vegetables = ["cucumber", "tomato", "lettuce", "asparagus",
              "potato", "wheat", "barley"]
farmers = ["Farmer Joe", "Upland Bros.", "Smith Gardening",
           "Agrifun", "Organiculture", "BioGoods Ltd.", "Cornylee Corp."]

harvest = np.array([[0.8, 2.4, 2.5, 3.9, 0.0, 4.0, 0.0],
                    [2.4, 0.0, 4.0, 1.0, 2.7, 0.0, 0.0],
                    [1.1, 2.4, 0.8, 4.3, 1.9, 4.4, 0.0],
                    [0.6, 0.0, 0.3, 0.0, 3.1, 0.0, 0.0],
                    [0.7, 1.7, 0.6, 2.6, 2.2, 6.2, 0.0],
                    [1.3, 1.2, 0.0, 0.0, 0.0, 3.2, 5.1],
                    [0.1, 2.0, 0.0, 1.4, 0.0, 1.9, 6.3]])


fig, ax = plt.subplots()
im = ax.imshow(harvest)

# We want to show all ticks...
ax.set_xticks(np.arange(len(farmers)))
ax.set_yticks(np.arange(len(vegetables)))
# ... and label them with the respective list entries
ax.set_xticklabels(farmers)
ax.set_yticklabels(vegetables)

# Rotate the tick labels and set their alignment.
plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
         rotation_mode="anchor")

# Loop over data dimensions and create text annotations.
for i in range(len(vegetables)):
    for j in range(len(farmers)):
        text = ax.text(j, i, harvest[i, j],
                       ha="center", va="center", color="w")

ax.set_title("Harvest of local farmers (in tons/year)")
fig.tight_layout()
plt.show()
```

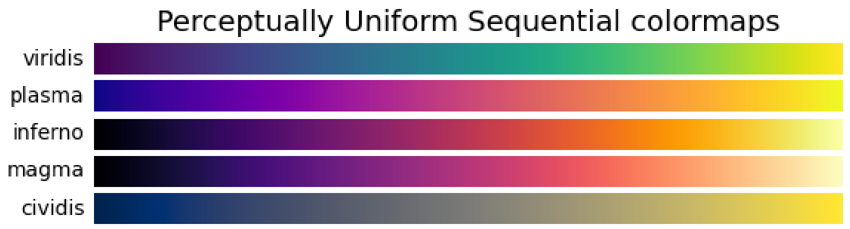


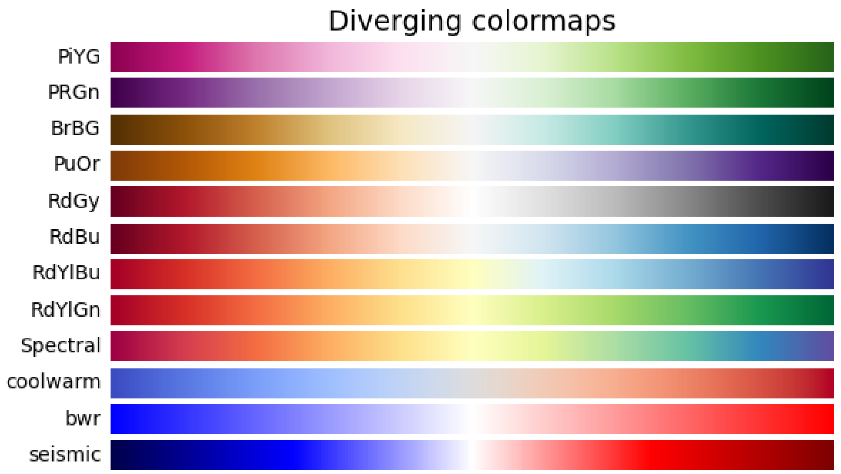
	Farmer Joe	Upland Bros.	Smith Gardening	Agrifun	Organiculture	BioGoods Ltd.	Cornylee Corp.
cucumber	0.8	2.4	2.5	3.9	0.0	4.0	0.0
tomato	2.4	0.0	4.0	1.0	2.7	0.0	0.0
lettuce	1.1	2.4	0.8	4.3	1.9	4.4	0.0
asparagus	0.6	0.0	0.3	0.0	3.1	0.0	0.0
potato	0.7	1.7	0.6	2.6	2.2	6.2	0.0
wheat	1.3	1.2	0.0	0.0	0.0	3.2	5.1
barley	0.1	2.0	0.0	1.4	0.0	1.9	6.3

More examples at:
https://matplotlib.org/3.1.1/gallery/images_contours_and_fields/image_annotated_heatmap.html

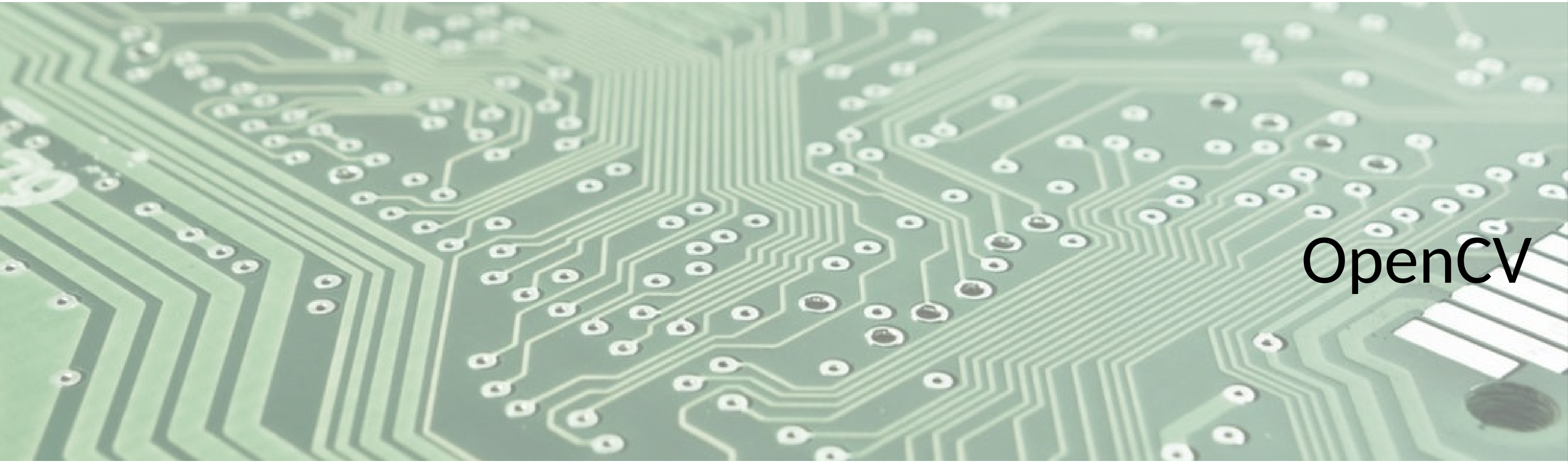
Python libraries

Matplotlib provides plenty of colormaps





More colormaps at:
https://matplotlib.org/3.2.2/tutorials/colors/colormaps.html



OpenCV

Python libraries

OpenCV is an open source Computer Vision library. It allows to develop complex Computer Vision and Machine Learning applications, offering a wide set of functions.

Originally developed in C/C++, now OpenCV has handlers also for Java and Python

It can be used also for deploying iOS and Android apps.

In Python, OpenCV and NumPy are **tightly connected**

Python libraries

Some of the functions offered by OpenCV are:

- Image Handling (read an image, write an image etc)
- Image manipulation (filtering, segmentation, etc.
- Corner Detection (Harris, Shi-Tomasi etc)
- Camera Calibration
- And more complex vision tasks
    - Features Detection and Description (ORB, SIFT, SURF etc)
    - K-Nearest Neighbour
    - Depth estimation (Block Matching, SGM etc)
    - Optical Flow (Lucas-Kanade)

    - … and many more!

Python libraries

We can install OpenCV directly by pip, calling

pip install opencv-python

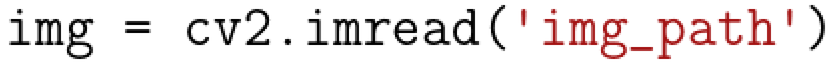
Then, in our Python script, we can import it as follows:

```
import cv2
```

Python libraries

As we said, OpenCV offers functions to read and write images.

We can open an image using the **imread** function:



```
img = cv2.imread('img_path')
```

Moreover, it handles various image format (png, jpeg etc) and data types (8bit, 16 bit etc)

Python libraries

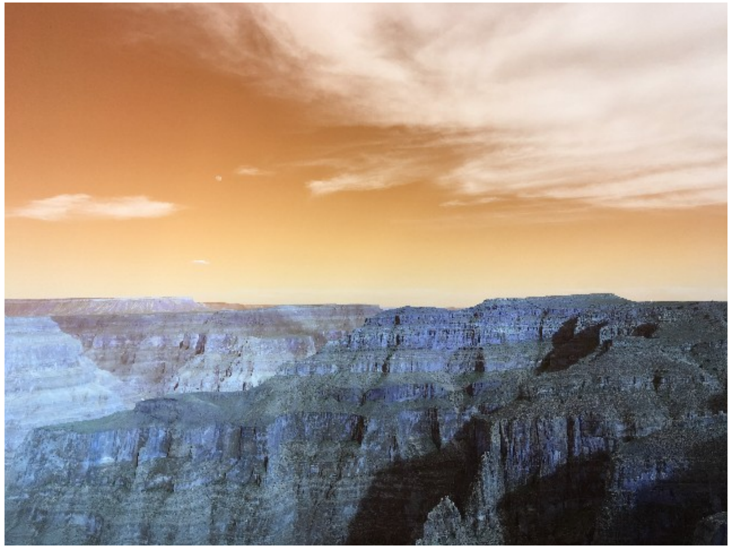
Once opened, OpenCV **returns a numpy array** that stores the image (each value of the array is a pixel)

OpenCV default format is **BGR**, so we need to swap the first and the last channels in order to manage **RGB** images. We can do it manually or invoking the cvtColor function

```
img = cv2.imread('path_to_your_image')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

cvtColor helps in converting colored images (**BGR** or **RGB**) to grayscale just using as options **cv2.BGR2GRAY** or **cv2.RGB2GRAY**

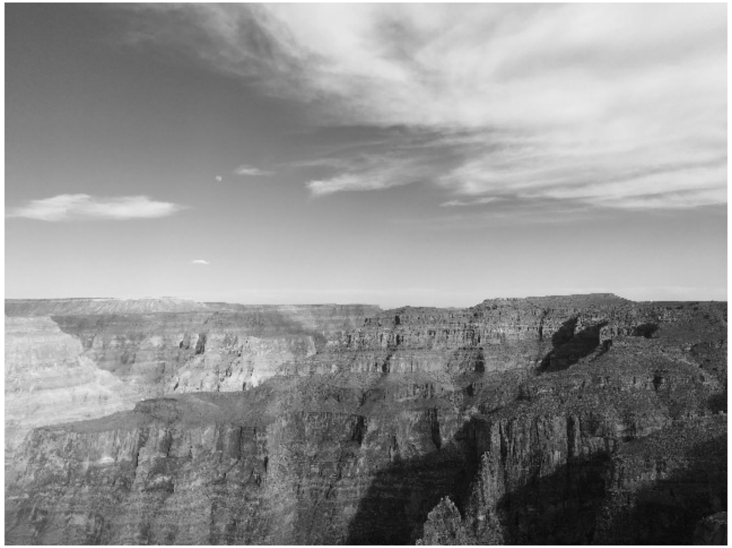
Python libraries



On left: the original RGB image.

On right: the same picture saved after being converted to BGR format

Python libraries



We can use cv.cvtColor function also to obtain grayscale images

```
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Python libraries

Instead, we can write an image in our file system using the **imwrite** function

```
cv2.imwrite('saving_path', img)
```

However, OpenCV expects a **BGR** image, so if *img* is in **RGB** format we must convert it to **BGR** first using cv2.cvtColor

```
bgr_img = cv2.cvtColor(rgb_img, cv2.COLOR_RGB2BGR)
```

Python libraries

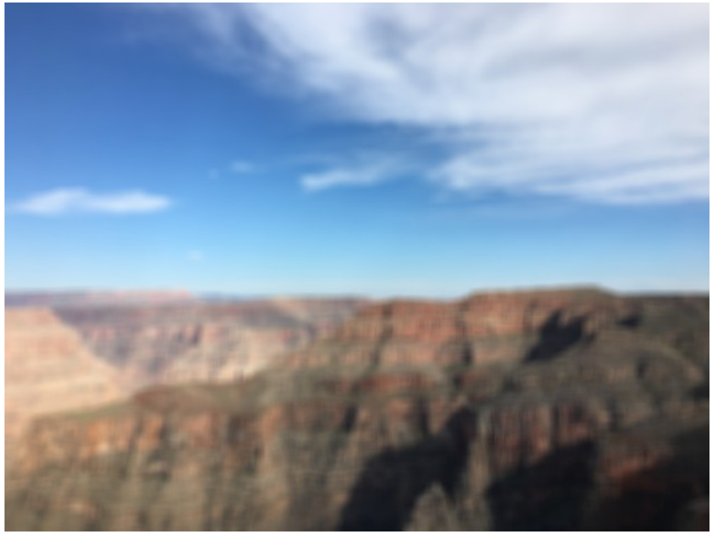
OpenCV allows to resize images using the **resize** function. It takes the image and the new shape

```
img = cv2.imread('image/img.png')
resized_img = cv2.resize(img, (320, 240))
```

Python libraries

2D Convolution in OpenCV is straightforward: you have just to call the **filter** function, passing as input the image and the kernel

```
kernel = np.ones((9,9),np.float32)/81
dst = cv2.filter2D(img,-1,kernel)
```

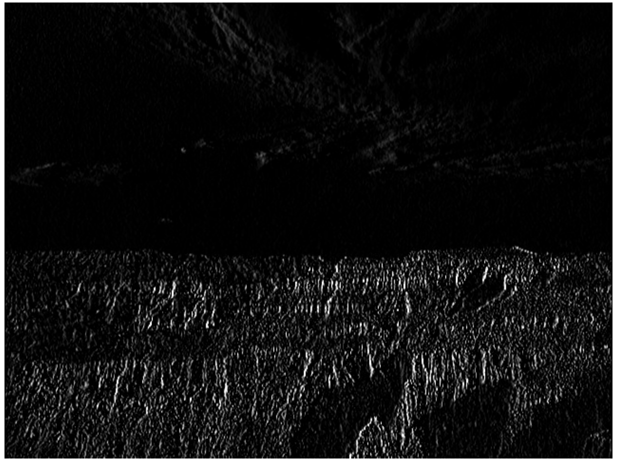


Python libraries

Changing the filter, we would obtain different results. For instance, **high-pass** filter can be obtained through a zero-sum kernel

```
kernel = np.array([[-1,0,1], [-2,0,2], [-1,0,1]], dtype=np.float32)
dst = cv2.filter2D(img,-1,kernel)
```

-1	0	1
-2	0	2
-1	0	1

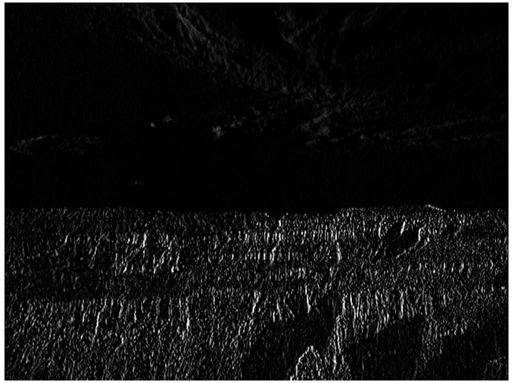


Python libraries

We can obtain the same result using **separable** filters

```
kernel1 = np.array([[-1,0,1]], dtype=np.float32)
kernel2 = np.array([[1],[2],[1]], dtype=np.float32)
dst = cv2.filter2D(img,-1, kernel1)
dst = cv2.filter2D(dst,-1, kernel2)
```

$$* \begin{matrix}-1 & 0 & 1\end{matrix} * \begin{matrix}1 \\ 2 \\ 1\end{matrix}$$



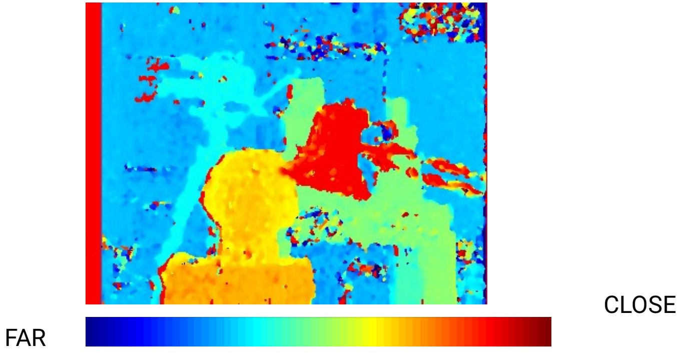
Python libraries

Example: stereo matching

```
left = cv2.cvtColor(cv2.imread('image/left.png'),cv2.COLOR_BGR2GRAY)
right = cv2.cvtColor(cv2.imread('image/right.png'), cv2.COLOR_BGR2GRAY)
stereo_matcher = cv2.StereoSGBM_create(numDisparities = 16, blockSize = 5)
disparity = stereo_matcher.compute(left, right)
disparity = (disparity).astype(np.uint8)
disparity = cv2.applyColorMap(disparity, cv2.COLORMAP_JET)
cv2.imwrite('./disparity.png',disparity)
```

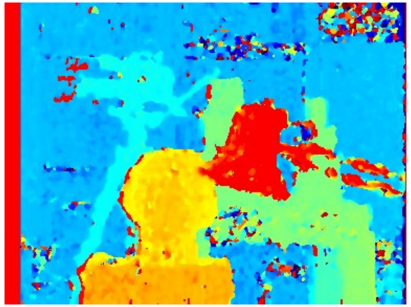
(opencv has colormaps too!)

from Middlebury Dataset

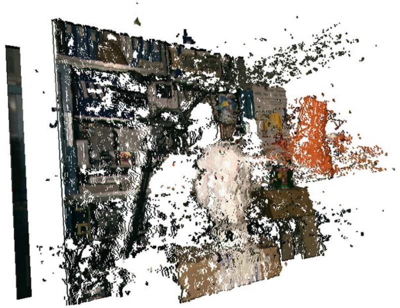


Python libraries

Example: 3D reconstruction from stereo



from Middlebury Dataset



```
img = cv2.imread('image/left.png')
left = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
right = cv2.cvtColor(cv2.imread('image/right.png'), cv2.COLOR_BGR2GRAY)
stereo_matcher = cv2.StereoSGBM_create(numDisparities = 16, blockSize = 5)
disparity = stereo_matcher.compute(left, right)
focal_length=1.
ppm =  np.float32([[1,0,0,0],[0,-1,0,0], [0,0,focal_length,0],[0,0,0,1]])
points_3D = cv2.reprojectImageTo3D(disparity, ppm)
colors = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
generate_pointcloud(colors,points_3D,'3D.ply')
```

Python libraries

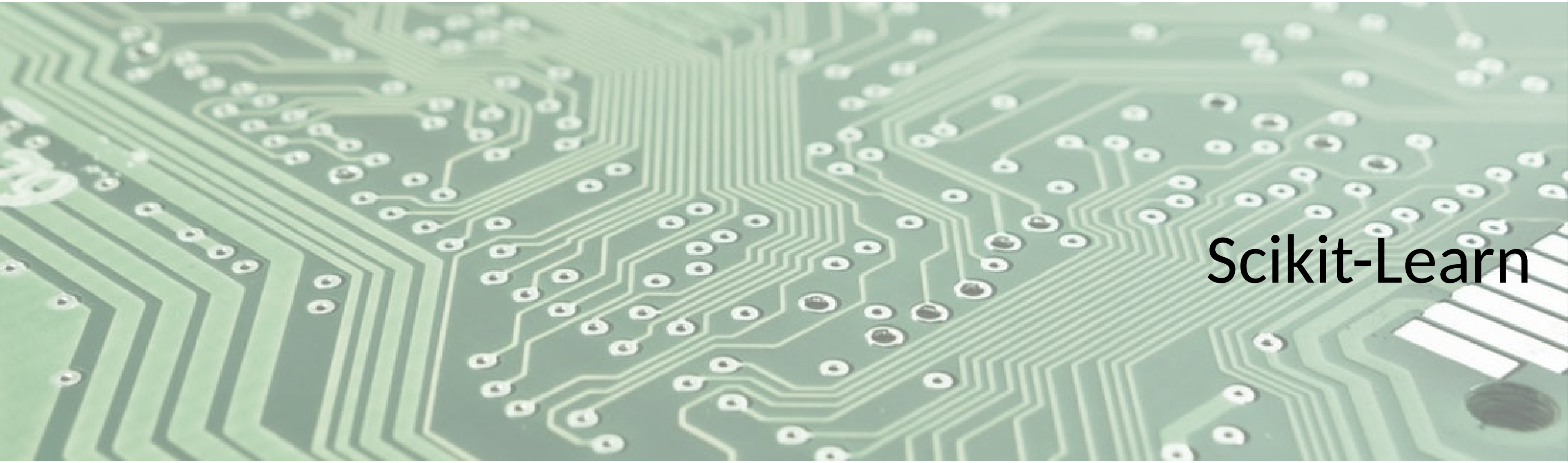
The computer vision libraries zoo:

OpenCV

pillow

Mahotas

scikit-image
image processing in python



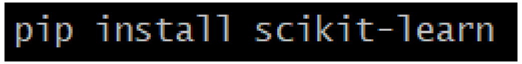
Scikit-Learn

Python libraries

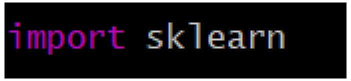
Scikit-learn is an open source library devoted to Machine Learning.

It provides several out-of-the-box algorithms for **classification**, **regression** and **clustering**

Tightly connected with Numpy! (and Scipy)



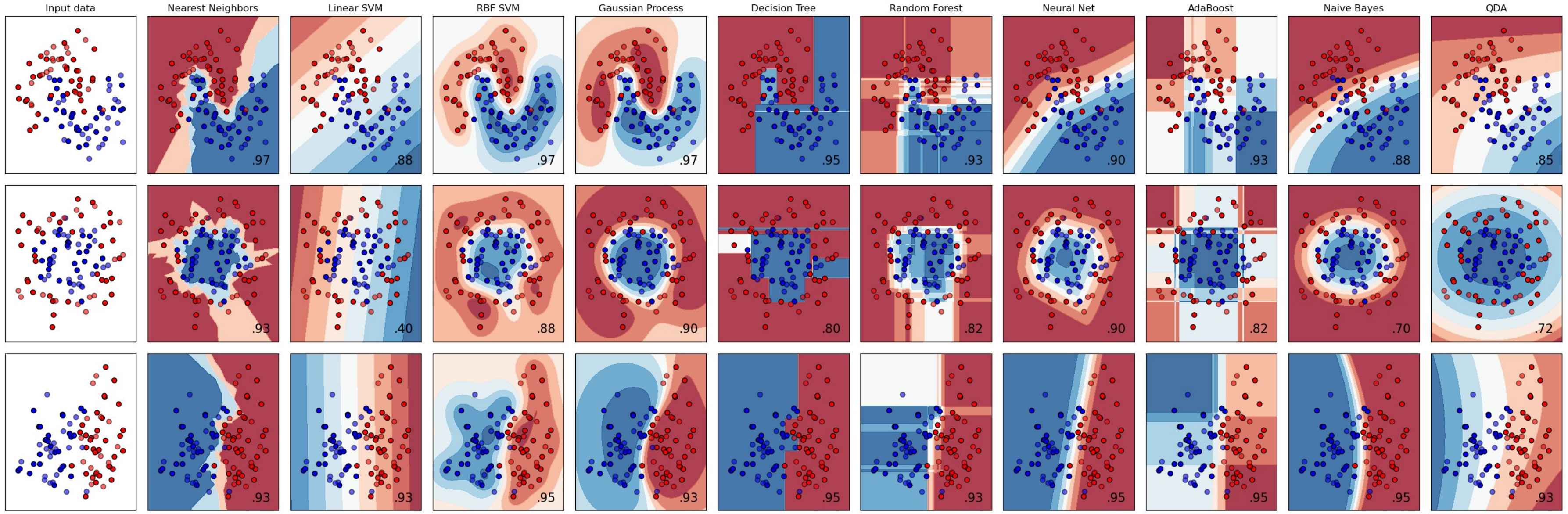
```
pip install scikit-learn
```



```
import sklearn
```

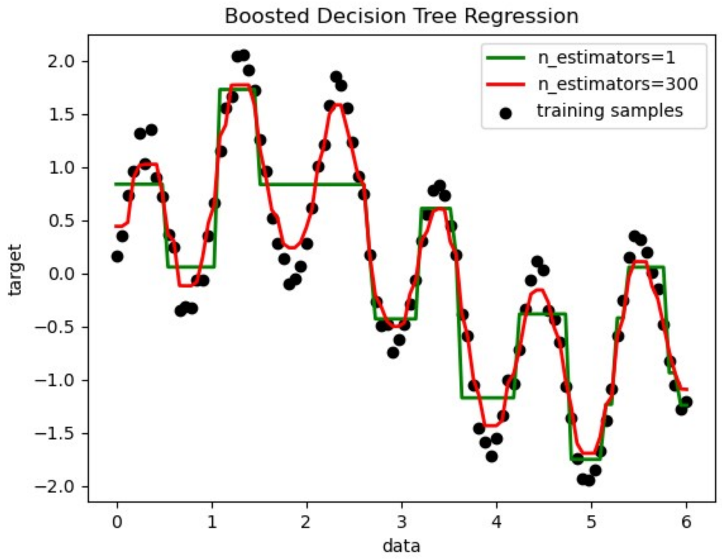
Python libraries

Classification:
We want to assign a **class label** to any observation. We usually know the classes in our problem (e.g., in image classification)



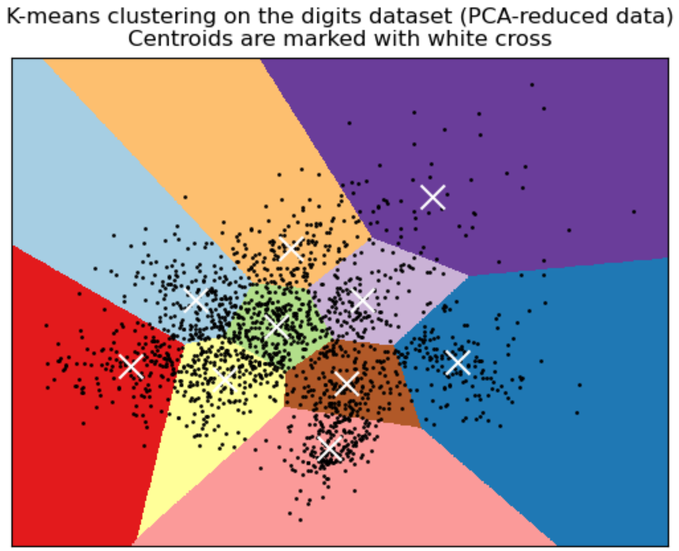
Python libraries

Regression:
We want to learn a **function** of the input data



Python libraries

Clustering:
We want to group data in **clusters** according to common properties. Conversely to classification, in this case we usually **don’t know** any prior about the clusters



Python libraries

Some of the machine learning algorithms available in scikit-learn:

- Tree classifiers
- Nearest-Neighbor (NN) classifiers
- Support Vector Machines
- Ensemble methods
- Etc…

(complete list here)

Conversely to deep networks, these classifiers are much more efficient. On the other hand, they often require explicit **features engineering** by the developer.

Python libraries

A toy example (from here)

Banknote classification (**fake** or **authentic**) according to features extracted from the banknote image itself

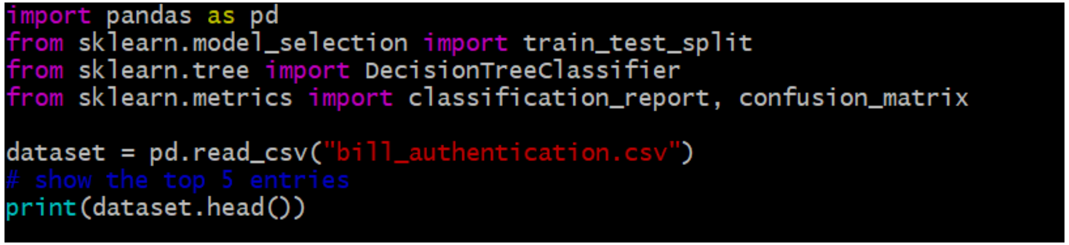
For each banknote in our dataset, we compute 4 features using image processing techniques and store them in a .csv file

We can download a toy dataset in this format from here

(credits: https://stackabuse.com/decision-trees-in-python-with-scikit-learn/)

Python libraries

A toy example



```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, confusion_matrix

dataset = pd.read_csv("bill_authentication.csv")
# show the top 5 entries
print(dataset.head())
```

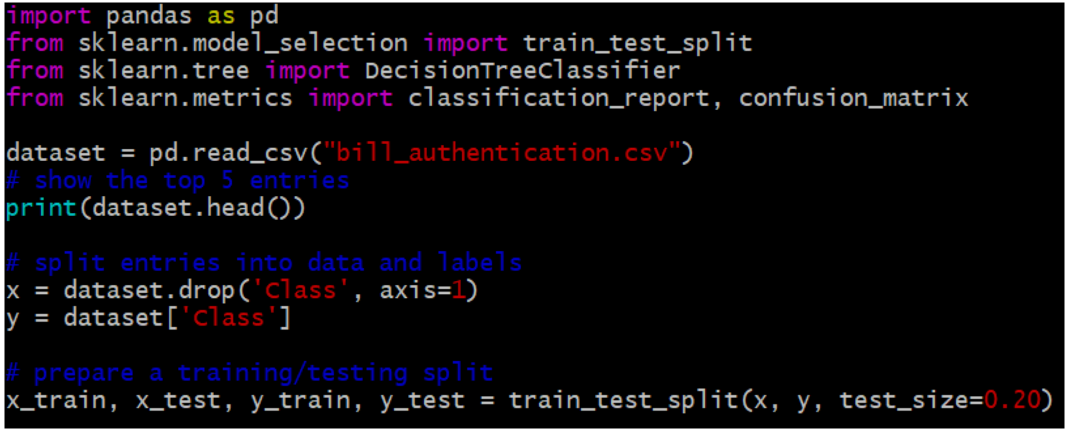
Pandas provides higher-level functions for data analysis

We can read a .csv file containing the attributes name in the first row and an entry in each row, returning a **Dataframe** structure

We can print the top-5 entries with the **.head()** method

Python libraries

A toy example



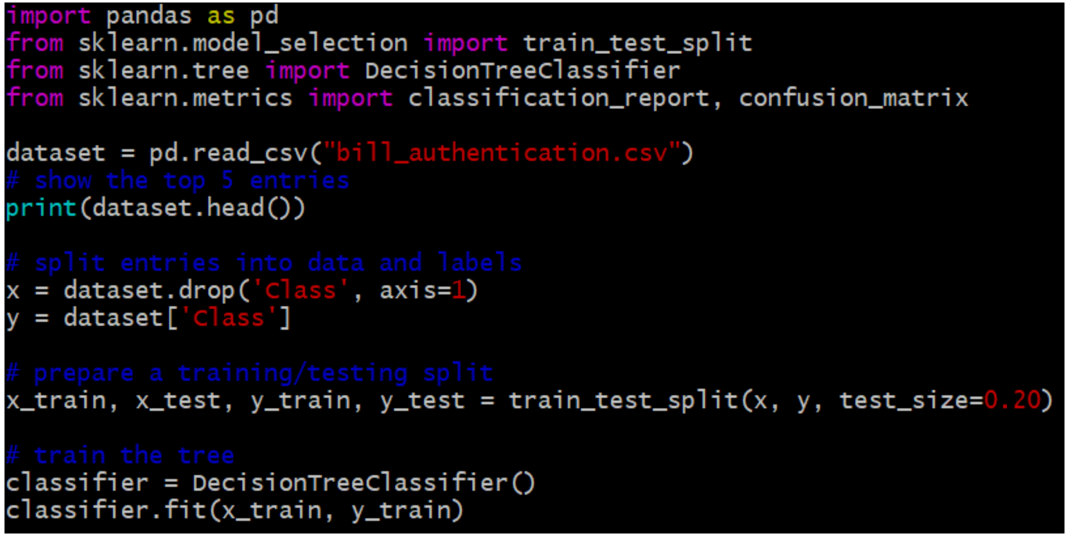
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, confusion_matrix

dataset = pd.read_csv("bill_authentication.csv")
# show the top 5 entries
print(dataset.head())

# split entries into data and labels
x = dataset.drop('Class', axis=1)
y = dataset['Class']

# prepare a training/testing split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.20)
```

We also divide our entries into **training data** and **testing data**

The latter are **not used** for training. We will measure on them the accuracy of our classifier

Python libraries

A toy example

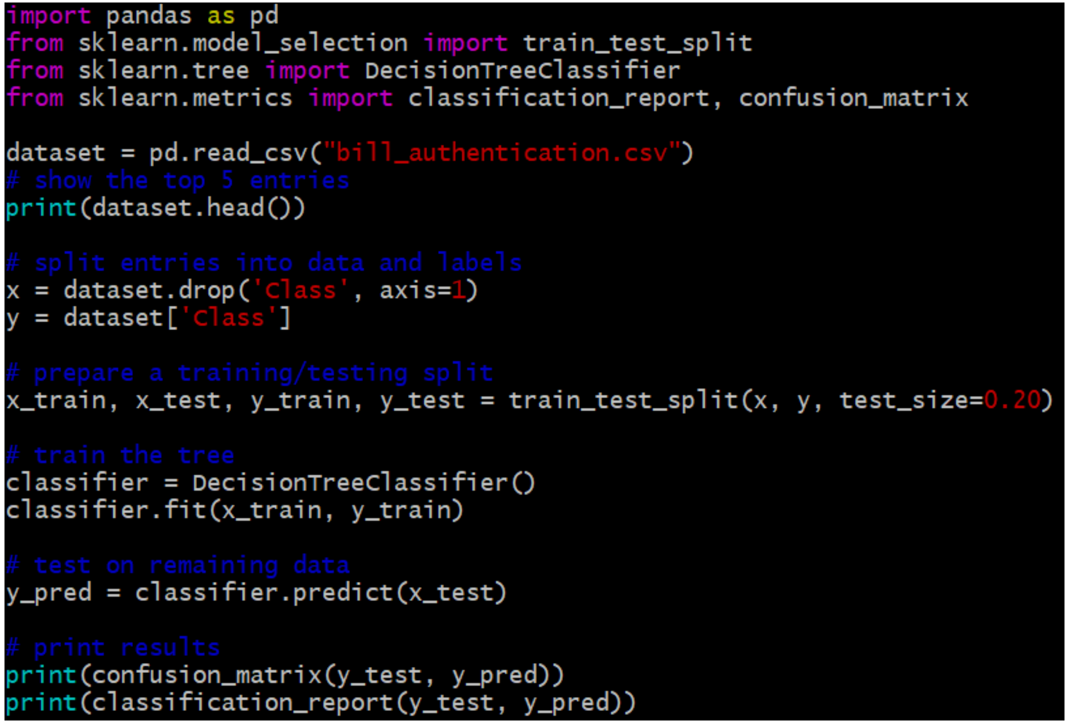


```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, confusion_matrix

dataset = pd.read_csv("bill_authentication.csv")
# show the top 5 entries
print(dataset.head())

# split entries into data and labels
x = dataset.drop('Class', axis=1)
y = dataset['Class']

# prepare a training/testing split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.20)

# train the tree
classifier = DecisionTreeClassifier()
classifier.fit(x_train, y_train)
```

Training a classifier is **easy**! Just create a DecisionTreeClassifier object and call the .fit method!

Python libraries

A toy example

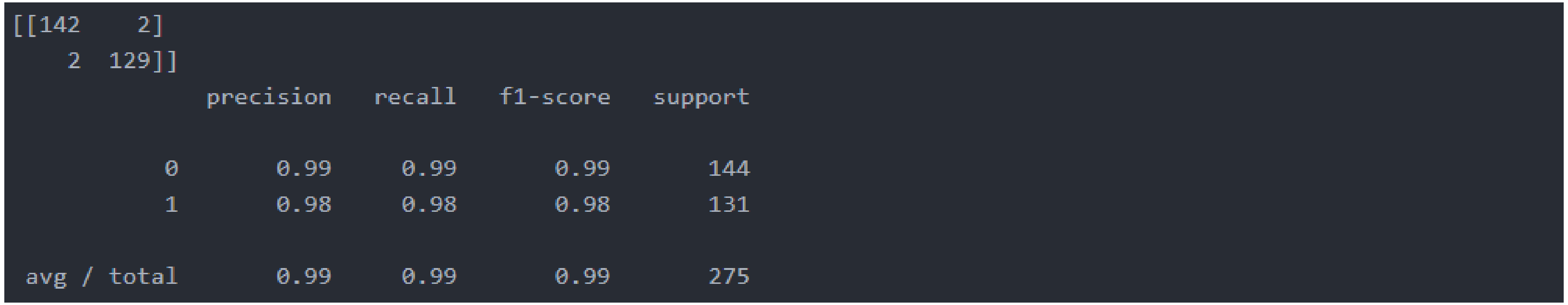


```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, confusion_matrix

dataset = pd.read_csv("bill_authentication.csv")
# show the top 5 entries
print(dataset.head())

# split entries into data and labels
x = dataset.drop('Class', axis=1)
y = dataset['Class']

# prepare a training/testing split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.20)

# train the tree
classifier = DecisionTreeClassifier()
classifier.fit(x_train, y_train)

# test on remaining data
y_pred = classifier.predict(x_test)

# print results
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

Finally, we test the classifier on our data and print the results

Python libraries

Showing results



```
[[142    2]
    2  129]]
             precision    recall  f1-score   support

          0       0.99      0.99      0.99       144
          1       0.98      0.98      0.98       131

avg / total       0.99      0.99      0.99       275
```

Confusion matrix: a 2D matrix showing for each class (rows) the distribution of labels assigned by the classifier (columns). Main diagonal = **correct labels**

Classification report: a set of scores measuring the effectiveness of the classifier (**precision**, **recall**, **f1-score**)

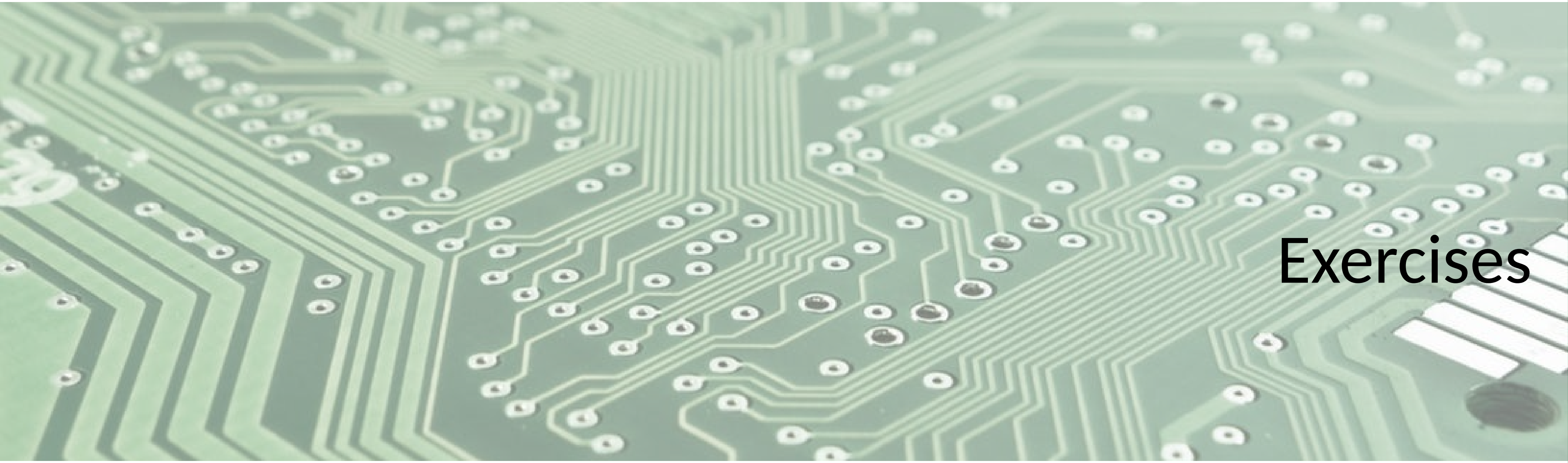
Python libraries

Showing results

In case of a binary classifier, we can count True Positives (TP, 1 predictions that are correct), True Negatives (TN, 0 predictions that are correct), False Positives (FP, 1 predictions that are wrong) and False Negatives (FN, 0 predictions that are wrong.

Precision (P) = TP / (TP+FN)
Recall (R) = TP / (TP+FP)
F1-score = 2xPxR/(P+R)

Others:
Accuracy = (TP+TN)/All



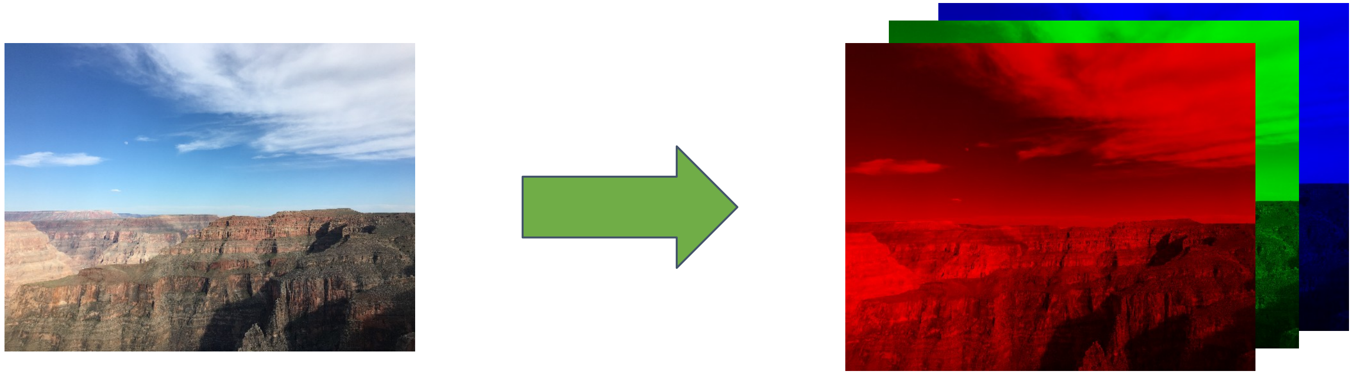
Exercises

Python libraries

Exercise 1

Given the image *canyon.png* load, load with, split the channels and save each channel in a new image called as the channel.

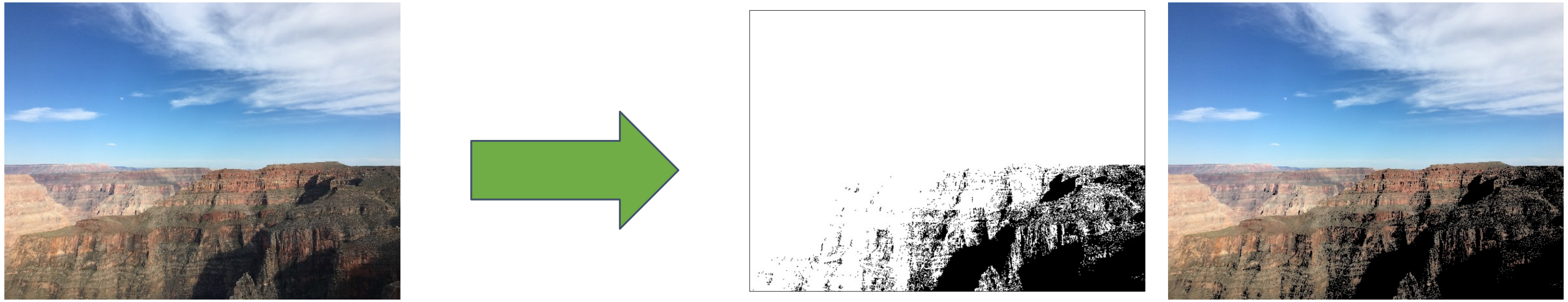
For instance, the red channel will be saved as *red.png*



Python libraries

Exercise 2

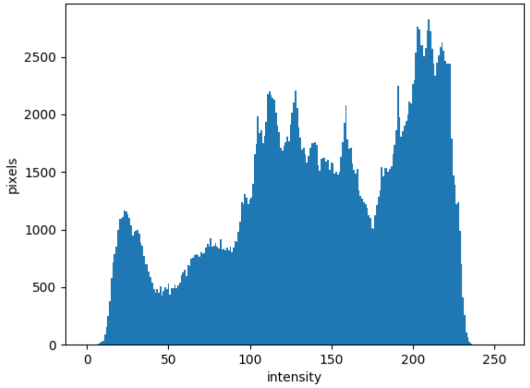
Using the same image of the previous exercise, load it as gray-scale and replace all pixels with intensity lower than 80 with 0, 1 otherwise. Save it both as a new image, called *mask.png*, and as npy. Finally, apply the mask to the original image, keeping the original value where the mask is 1, 0 otherwise



Python libraries

Exercise 3

Using Matplotlib, display intensity values of *canyon.png* (loaded as grayscale image) in a bar chart. For each intensity value, the height of the column is given by the number of pixels that have such intensity value


pixels
intensity