Introduction to Python (MIPIA Programming Exercise)

Python is a highly portable dynamic programming language. If you are familiar with *Matlab*, then for starters you may consider Python as an environment that basically allows you adequate programming in a noncommercial way (so far). With the rise of deep learning it became really popular in the scientific field. It supports *object oriented programming* and you will find a lot of helpful libraries for your problems.
This is an introduction to Python. Later you will implement actual Python code files, but to get an idea of this programming language you can work through this so-called *Juypter notebook* in an easy fashion by solving each task step by step.

Syntax

Variables do not need a variable declaration (like in Matlab), but they do have a **type** (in contrast to Matlab where most variables are initialized as `double`). **Execute** the following lines by clicking into the field and then pressing `Ctrl + Enter` .

In [1]:
```
a = 1
a_type = type(a)

print(a)
print(a_type)
```

```
1
<class 'int'>
```

The *lack of semi-colons and parentheses* to finalize a command or command block, respectively, are characteristics of the Python syntax. Structure is generated by using **indentation** as shown in the following line. This time execute it with `Shift + Enter` .

In [2]:
```
if a > 0:
    print(a)
```

```
1
```

Loops can be used as follows:

In [3]:
```
# for-loop
print("for")
for i in range (5):
    print(i)

# while-loop
print("while")
i = 0
while i < 5:
    print(i)
    i = i + 1 # this line is also inside the loop

# foreach-loop
print("foreach")
array = [0, 1, 2, 3, 4]
for val in array:
    print(val)
```

```
for
0
1
2
3
4
while
0
1
2
3
4
foreach
0
1
2
3
4
```

Functions can be defined by using the keyword `def` and using indentation for the function block.

In [4]:
```
def divide(a , b):
    assert b != 0
    return a / b

res1 = divide(225,75)
print("The result is {}!".format(res1))
```

```
The result is 3.0!
```

Here you also see how a value can be inserted into a formatted string.

Now it is your turn!

Calculations with Python

Basics

Let's start with some simple calculations. For this, we need to **import Python's math library *NumPy* to our notebook**. Please look up in the internet how this can be achieved. Use the name `np` for the module to access its functions later on.

```
In [5]: import numpy as np
```

Now we can start computing by introducing two *variables* x and y (initialize them with values of your own choice).

```
In [6]: x = 6
        y = 8

        # or simply

        x,y = 6,8
```

For calculations, we need to call the functions of NumPy by accessing them via `np` . Let us calculate $\sqrt{x^2 + y^2}$ and save it to a variable called z.

```
In [7]: z = np.sqrt(np.square(x) + np.power(y,2))
```

Print this variable.

```
In [8]: print(z)

        10.0
```

Working with arrays and matrices

Define an *array* containing the integer numbers from 1 to 10.

```
In [9]: values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Get the *type* of the defined array.

```
In [10]: type(values)

Out[10]: list
```

You should see that it is a `list` . **Convert** the array to a NumPy array with `np.array(<args>)` and name it `v` . Check the type of the converted array.

```
In [11]: v = np.array(values)
         type(v)

Out[11]: numpy.ndarray
```

You should see that the type has changed. Now let us work with this array:

1. First, **print the complete array**.
2. Then print the array's **first, last and second to last element**.

Do this in a *generalizable* way, i.e. such that you can reuse your commands for an array with arbitrary length.
Hint: You can access the last element with the index -1.

In [12]:
```
print(v)
print("First element: {}".format(v[0]))
print("Last element: {}".format(v[-1]))
print("Second last element: {}".format(v[-2]))
```

```
[ 1  2  3  4  5  6  7  8  9 10]
First element: 1
Last element: 10
Second last element: 9
```

Numpy arrays allow easy array *operations*. **Subtract 1** from the whole array `v` , **multiply** its elements **by 5** in a single line and finally **print** for each element **if it is smaller than 20**.

In [13]:
```
v = v - 1
v = v*5
print(v < 20)
```

```
[ True  True  True  True False False False False False False]
```

The NumPy method `arange(<start>, <stop, excluded>, <step size, optional>)` creates an array of successive values in the range of the interval $[\text{start}, \text{stop})$. The default step size is 1.

Create

- one array `a1` in the range of $[1, 10)$ with step size 1,
- and another array `a2` in the range of $[0.5, 5)$ with a step size of 0.5. Print both to check the result.

In [14]:
```
a1 = np.arange(1, 10)
a2 = np.arange(0.5, 5, 0.5)

print(a1)
print(a2)
```

```
[1 2 3 4 5 6 7 8 9]
[0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5]
```

In medical imaging, working in 1-D is not enough, so let us switch to 2-D.

Create a **2-D array** `V` for a matrix $\boldsymbol{V}$ with dimensions 3×3 and fill it with increasing numbers from 0 to 8 (rowwise first).
Hint: Instead of typing the elements, you can use `np.arange(<args>)` in combination with `np.reshape(<args>)` .

In [15]:
```
V = np.array([ [0, 1, 2], [3, 4, 5], [6, 7, 8] ])
print(V)

# alternative
V = np.arange(9).reshape(3, 3)
print(V)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

Print the following:

1. the **transposed** matrix $\boldsymbol{V}^\mathsf{T}$,
2. the **size** of the matrix/array (= *total number of elements*),
3. the **dimension** of the array (which is 2 for a matrix),
4. the **shape** (dimensions of the matrix),
5. and the **last element** (independent of the dimensions).

For this, look up suitable methods and properties of the NumPy array class.

In [16]:
```
print("Transposed matrix:\n {}".format(V.T))
print("Size: {}".format(V.size))
print("Number of dimensions: {}".format(V.ndim))
print("Shape: {}".format(V.shape))
print("Last element: {}".format(V[-1][-1]))
```

```
Transposed matrix:
 [[0 3 6]
 [1 4 7]
 [2 5 8]]
Size: 9
Number of dimensions: 2
Shape: (3, 3)
Last element: 8
```

To access whole segments of an array, you can use the **colon** operator `:` .

1. Print the **elements 0 to 5** of the *1-D array* `v` .
2. Print the **last row** of the *2-D array* `V` .

In [17]:
```
print("1D: {}".format(v[0:5]))
print("2D: {}".format(V[-1,:]))
```

```
1D: [ 0  5 10 15 20]
2D: [6 7 8]
```

There are methods to create certain standard arrays (as you might know them from Matlab).
Create a 3×3 matrix filled with **zeros**, another one filled with **ones**, and as a third one the **identity matrix**.

In [18]:
```python
Z = np.zeros((3,3))
O = np.ones((3,3))
I = np.identity((3))
E = np.eye((3)) # alternative

print(Z)
print(O)
print(I)
print(E)
```

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

Last, let us do some **matrix operations**. As you know, there are *different kinds of multiplication* possible with matrices and vectors. You can do all of them with Python.
Symbolic operators that can be used are `*` for *element-wise multiplication*, and `@` for the *matrix product*. Earlier versions of Python did not inherit a specific operator for the array class to calculate the matrix product of two matrices. Therefore, the `@` operator was introduced.
NumPy also provides other products like the cross product `np.cross(<args>)` in 2-D and 3-D. For matrices it returns the cross product of the row vectors.

Print the results for

1. **element-wise multiplication** of the matrix $\boldsymbol{V}$ and its *transpose* $\boldsymbol{V}^{\mathsf{T}}$,
2. the **matrix product $\boldsymbol{V}\boldsymbol{V}^{\mathsf{T}}$**,
3. and the **cross product** of both matrices.

In [19]:
```python
# element-wise multiplication
print(V*V.T)

# matrix multiplication
print(V@V.T)
print(np.dot(V,V.T)) # alternative

# cross product
print(np.cross(V,V.T))
```

```
[[ 0  3 12]
 [ 3 16 35]
 [12 35 64]]
[[  5  14  23]
 [ 14  50  86]
 [ 23  86 149]]
[[  5  14  23]
 [ 14  50  86]
 [ 23  86 149]]
[[  0   0   0]
 [  8 -16   8]
 [ 16 -32  16]]
```

NumPy's built-in methods are usually applied on *all* of an input array's *elements* separately.

1. Print the 3×3 matrix where each element shows the **exponential value** of the corresponding element in $\boldsymbol{V}$.
2. Do the same computing the **square roots** of each element in $\boldsymbol{V}$.

In [20]:
```
print(np.exp(V))
print(np.sqrt(V))
```

```
[[1.00000000e+00 2.71828183e+00 7.38905610e+00]
 [2.00855369e+01 5.45981500e+01 1.48413159e+02]
 [4.03428793e+02 1.09663316e+03 2.98095799e+03]]
[[0.         1.         1.41421356]
 [1.73205081 2.         2.23606798]
 [2.44948974 2.64575131 2.82842712]]
```

Output plots

For plotting functions and showing images we can use the module `matplotlib.pyplot` . Simply execute the following line to continue.

In [21]:
```
import matplotlib.pyplot as plt
```

Plotting functions

The goal is to **plot** the following two functions in a single graph:

$$f(x) = e^{-x}$$
$$g(x) = x$$

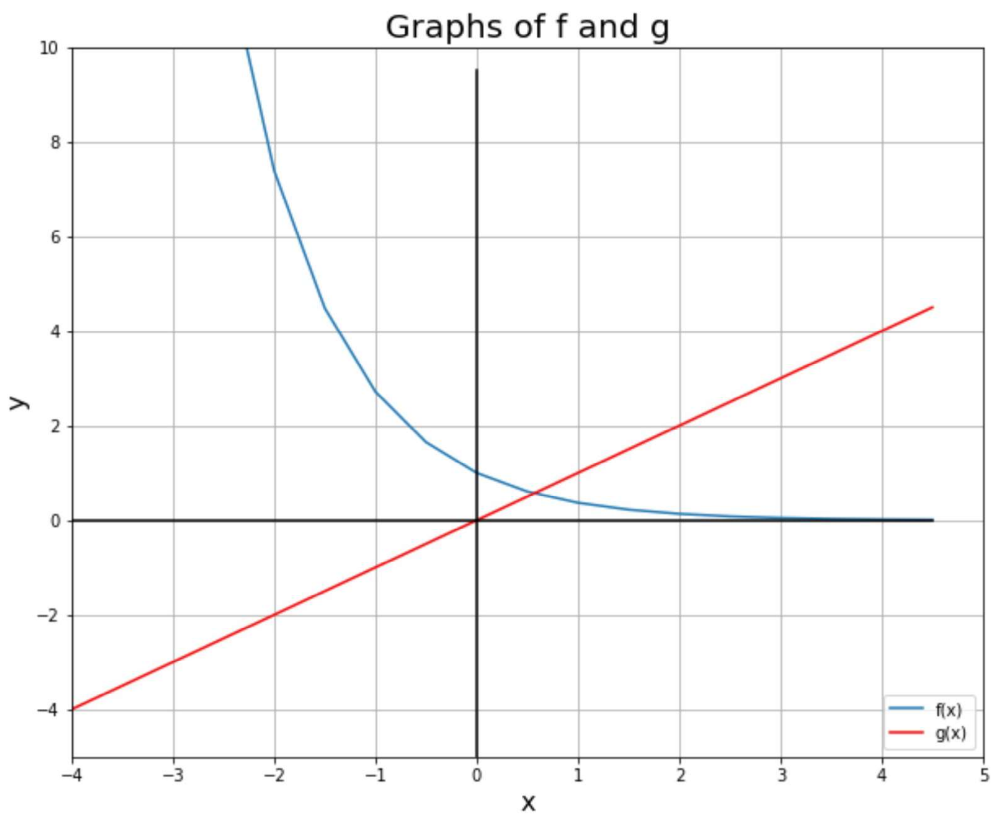
We want to visualize both functions where x is in the range of $[-4, 5]$ and y is in the range of $[-5, 10]$.

1. Create a 1-D array `x` with `np.arange(<args>)` in the given range and step size 0.5.
2. Create a 1-D array `y` in the same fashion.
3. Create arrays `f` and `h` which contain the function values at the sampled points `x` .

In [22]:
```
x = np.arange(-4,5,0.5)
y = np.arange(-5,10,0.5)
f = np.exp(-x)
g = x
```

Now **complete** the following code which plots the two functions using *Matplotlib*.

In [23]:
```
# define a new figure and set a size
fig = plt.figure(figsize=(10,8))

# plot the functions
plt.plot(x, f)
plt.plot(x, g, color='#ff0000')

# plot x- and y-axes (y=0, x=0)
plt.plot(x, np.zeros(len(x), int), color='#000000')
plt.plot(np.zeros(len(y), int), y, color='#000000')

# show grid lines
plt.grid()

# limit axes to the ranges mentioned above
plt.xlim([-4, 5])
plt.ylim([-5, 10])

# figure title
plt.title("Graphs of f and g", size=20)

# axes labels
plt.xlabel("x", size=16)
plt.ylabel("y", size=16)

# legend
plt.legend(["f(x)", "g(x)"], loc='lower right')

plt.show()
```



You can **save** this plot as PDF by:

In [24]:
```
fig.savefig("output/P0_plot.pdf", format='pdf', dpi=100);
```

Imaging

Now we want to do some **image processing**.
Start by **loading** the well-known test image *Lena* from disk (you find it besides this notebook's file in the same folder).
Use the method `imread(<imagename>)` from Matplotlib appropriately to open the 8 bit RGB color image (i.e. 8 bits in each channel).

In [25]:
```
I = plt.imread("lena.png")
```

Now **display** the image. Use the Matplotlib methods `imshow(<imagearray>)` followed by `show()` .

In [26]:
```
# scale and remove axes
plt.gcf().set_size_inches(10, 10)
plt.axis('off')

plt.imshow(I)
plt.show()
```

Convert the RGB image to grayscale. For this purpose use the `Image` module from `PIL` (*Pillow*, formerly *Python Imaging Library*). *Open* the image and *convert* it to grayscale with the **proper methods** from the `Image` module.
Remark: This is different to the way the image was opened for the plot above!

In [27]:
```
from PIL import Image

gray = Image.open("lena.png").convert('L')
type(gray)
```

Out[27]:
```
PIL.Image.Image
```

When using `PIL.Image` , the type of the image *changed* to `PIL.Image.Image` . **Convert it back** to a NumPy array using the `asarray(<Image>)` method.

In [28]:
```
gray = np.asarray(gray)
type(gray)
```

Out[28]:
```
numpy.ndarray
```

Take a look at the image. To set the grayscale, use the `cmap` attribute and set it to 'gray'.

In [29]:
```
# scale and remove axes
plt.gcf().set_size_inches(10, 10)
plt.axis('off')

plt.imshow(gray, cmap='gray')
plt.show()
```

To **save** the manipulated image on your disk you can use the `imsave(<args>)` method from Matplotlib.

In [30]:
```
plt.imsave("output/gray-lena.png", gray)
```

Investigate both the RGB image's and the grayscale image's **dimensions**.

In [31]:
```
height, width, channels = I.shape
print("Size of the colored image: {1} x {0}, {2} channels".format(height, width, channe
ls))
print("Size of the grayscale image: {}".format(gray.shape))
```

```
Size of the colored image: 512 x 512, 3 channels
Size of the grayscale image: (512, 512)
```

By using **subplots** images can be placed next to each other. Try placing the two images from above side by side.

In [32]:
```
fig = plt.figure(figsize=(13,6))

# subplot 1
plt.subplot(1,2,1)
plt.imshow(I)
plt.axis('off')
plt.title("Colored image", size=16);
# subplot 2
plt.subplot(1, 2, 2)
plt.imshow(gray, cmap='gray')
plt.axis('off')
plt.title("Grayscale image", size=16);

plt.show()
```

SVD (MIPIA Programming Exercise)

We want to compute the **singular value decomposition (SVD)** of a 3×3 matrix filled with *random* values. At the end, we also perform some **validity checks** on typical *properties* of the SVD to test if the computation was correct.

Decomposition

Create an array `X` with the given shape, filled with random values. Look up the `numpy.random` module for that.

In [33]:
```
X = np.random.rand(3,3) - 0.5
print(X)
```

```
[[ 0.0485257  -0.02855161 -0.33122353]
 [-0.40122905 -0.10697343  0.36572249]
 [-0.46884445 -0.0955264  -0.31703857]]
```

The SVD for a matrix $\boldsymbol{X} \in \mathbb{R}^{3\times3}$ is defined as

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^\mathsf{T},$$

where

- $\boldsymbol{U} \in \mathbb{R}^{3\times3}$ is a *unitary* matrix, i.e. $\boldsymbol{U}^{-1} = \boldsymbol{U}^\mathsf{T}$; the columns of $\boldsymbol{U}$ are the eigenvectors of $\boldsymbol{X}\boldsymbol{X}^\mathsf{T}$,
- $\boldsymbol{V} \in \mathbb{R}^{3\times3}$ is a unitary matrix; the rows of $\boldsymbol{V}$ are the eigenvectors of $\boldsymbol{X}^\mathsf{T}\boldsymbol{X}$,
- $\boldsymbol{S} \in \mathbb{R}^{3\times3}$ is an array containing the *singular values* of the matrix $\boldsymbol{X}$, sorted in descending order.

Use the method `svd(<matrix>)` from the `numpy.linalg` module to compute the SVD components `U`, `S`, `V` for the array `X`.

In [34]:
```
U, S, V = np.linalg.svd(X)
print(U)
print(S)
print(V)
```

```
[[-0.17554758  0.53924764  0.82364739]
 [ 0.76021638 -0.45732243  0.4614404 ]
 [ 0.62550308  0.70715499 -0.32966305]]
[0.63696931 0.58393729 0.04244018]
[[-0.95264105 -0.21360973  0.21643918]
 [-0.2087335  -0.05827169 -0.97623498]
 [ 0.22114557 -0.97517962  0.01092449]]
```

Hint: Consider that the outputs are implemented according to the definition above (i.e. the matrix $\boldsymbol{V}^\mathsf{T}$ is returned).

Validity checks

To *conclude* this exercise, let us do some **checks** to ensure that the SVD calculations were right.

First, the matrix **dimensions** should be 3×3 for `U`, `V`, and 3 for `S`. Print all three.

In [35]:
```
print("Size of U: {0} x {1}".format(len(U), len(U[0])))
print("Size of S: {0}".format(len(S)))
print("Size of V: {0} x {1}".format(len(V), len(V[0])))
```

```
Size of U: 3 x 3
Size of S: 3
Size of V: 3 x 3
```

`S` is given as a *vector of singular values*. Convert it into a **diagonal matrix** and then **check** if $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^\mathsf{T}$ holds.
Hint: You can use the NumPy method `allclose(<matrix1>,<matrix2>)` to check if two matrices are *'close to equal'* (slight differences are allowed).

```
In [36]: print(S)
         print(np.diag(S))
         print(np.allclose(X, np.dot(U, np.dot(np.diag(S), V))))
```

```
[0.63696931 0.58393729 0.04244018]
[[0.63696931 0.         0.        ]
 [0.         0.58393729 0.        ]
 [0.         0.         0.04244018]]
True
```

Check if `U` and `V` are **unitary matrices**, i.e. $\boldsymbol{U}\boldsymbol{U}^\top = \boldsymbol{U}^\top\boldsymbol{U} = \boldsymbol{I}$, $\boldsymbol{V}^\top\boldsymbol{V} = \boldsymbol{V}\boldsymbol{V}^\top = \boldsymbol{I}$. Do this by checking if the respective matrix products are close to identity $\boldsymbol{I}$.

```
In [37]: print(np.allclose(np.dot(V.T, V), np.identity(3)))
         print(np.allclose(np.dot(U, U.T), np.identity(3)))
```

```
True
True
```

Finally, check if the following property regarding the **eigenvectors and eigenvalues** of $\boldsymbol{X}^\top\boldsymbol{X}$ holds:

$$\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{v}_i = \lambda_i\boldsymbol{v}_i\,, \qquad \lambda_i = s_i^2\,, \quad s_i = (\boldsymbol{S})_{ii}\,, \quad \boldsymbol{v}_i = (\boldsymbol{V})_i\,, \quad i = 1, 2, 3.$$

```
In [38]: for i in range(3):
             eigvec = V[i,:]
             eigval = S[i]**2
             chkval = np.allclose(np.dot(np.dot(X.T, X), eigvec), eigval * eigvec)
             print(chkval)
```

```
True
True
True
```

Congratulations!

You have **completed** the *introduction to Python* and are now familiar with the most **important concepts of Python**. Now you can **proceed** with the *next* MIPIA programming exercises.