Computación Numérica / Numerical Computing

(https://www.unioviedo.es/compnum/compnum_NG/)

Notions of Numpy and Matplotlib

Contents

Interaction with Python

In this course, we shall interact in three ways with Python:

- **`spyder`**: is an Interactive Development Environment (IDE) for `Python` that provides edition and debugging features in a simple and light-weighted software. We use it for almost all the work related to **coding exercises**.
- **`terminal`**: is the essential interaction with Python. We will use it for some simple examples in class.
- **`jupyter notebook`**: is an interactive computational environment, in which you can combine code execution, rich text, mathematics, plots and rich media. We use it mainly to write these labs. These notebooks are used in Google Colab (https://colab.research.google.com/?hl=en). For example, this notebook (https://colab.research.google.com/drive/1ttW51L_gXPGDGfk8HPUPP-YvvlBhIwsv?usp=sharing).

Running the code (assuming that the paths are correctly set),

- **`spyder`**: There should be some icon linking to Spyder execution in your Desktop or in your Applications menu. In any case, you always can run spyder from command line just by running (writing) **`spyder`**.
- **`terminal`**: Just open a console and run **`python`**. Then, you enter to python command line. Another use of the console is to run a specific program. In this case you run in console **`python myscript.py`**.
- **`jupyter notebook`**: open a terminal, go tot the folder that contains the code and run **`jupyter notebook`**. The file extension is `ipynb`.

We will use Anaconda distribution (https://www.anaconda.com/download) with Python 3 for these labs.

Back to contents

Modules and packages

Python is a very versatile language that can perform very different tasks. In this course we shall focus in some **numerical algorithms** you can implement through Python.

Python is modular. A module is a script containing definitions of variables and functions, among others. Very often, modules are gathered into packages attending to some specific application. For instance, we shall make an extensive use of

- NumPy (numerical python).
- Matplotlib (python plotting).
- SciPy (scientific python).

A standard python script starts importing modules from packages. For instance

```
import numpy as np
```

Here we have imported all the `numpy` contents, to which we shall refer to with the prefix `np` :

```
print(np.pi)
```

```
3.141592653589793
```

There are variations of the `import` command. For instance, if we want to load just the definition of π, we use

```
from numpy import pi as PI
print(PI)
```

```
3.141592653589793
```

Back to contents

NumPy

The **`numpy`** package (module) is used in almost all numerical computation using Python. It is a package that provides high-performance vector, matrix and higher-dimensional data structures for Python. It is implemented in C and Fortran so when calculations are vectorized (formulated with vectors and matrices), performance is very good.

There are many ways of creating a numpy array. For instance,

```
a = np.array([1, 2, 3, 4])
b = np.array([(1.5, 2, 3), (4, 5, 6)])
c = np.zeros((3, 4))
d = np.ones((2, 3))
e = np.arange(1, 10, 2)
f = np.arange(1., 10, 2)
g = np.linspace(1, 9, 5)
```

```
print('a\n',a,'\n')
print('b\n',b,'\n')
print('c\n',c,'\n')
print('d\n',d,'\n')
print('e\n',e,'\n')
print('f\n',f,'\n')
print('g\n',g)
```

```
a
 [1 2 3 4]

b
 [[1.5 2.  3. ]
 [4.  5.  6. ]]

c
 [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

d
 [[1. 1. 1.]
 [1. 1. 1.]]

e
 [1 3 5 7 9]

f
 [1. 3. 5. 7. 9.]

g
 [1. 3. 5. 7. 9.]
```

Indexes begin with zero. `-1` index is for the last element of the array.

In `arange`, the values are generated within the half-open interval [1, 10), that is 1 is included but 10 is not. The last index is the step. It is similar to `range` but this last generates a list while `arange` generates a numpy array.

```
print('a[0]    = ', a[0], '\nb[0,0]  = ',b[0,0], '\nb[0][0] = ',b[0][0])
print('e[-1]   = ',e[-1])
```

```
a[0]    =  1
b[0,0]  =  1.5
b[0][0] =  1.5
e[-1]   =  9
```

`len` : number of elements in a one-dimension numpy array (vector), number of rows in a two-dimension numpy array (matrix).

`ndim` : dimension number of an array (one for a vector, two for a matrix, at least three for a multidimensional matrix)

`shape` : number of elements in every dimension, (number rows, number columns) for a two-dimension matrix.

```
print('len a = ', len(a),'; dim b = ', b.ndim, '; shape b = ', b.shape)
```

```
len a =  4 ; dim b =  2 ; shape b =  (2, 3)
```

Basic operations

Arithmetic operators on arrays apply element-wise. This is different than for Python lists!

```
a = np.array([1, 2, 3, 4])
b = np.array([(1.5, 2, 3, 5)])

a1 = [1, 2, 3, 4]
b1 = [1.5, 2, 3, 5]
```

```
print('a+b numpy array ')
print(a+b)
print('\na1+b1 list')
print(a1+b1)
```

```
a+b numpy array
[[2.5 4.  6.  9. ]]

a1+b1 list
[1, 2, 3, 4, 1.5, 2, 3, 5]
```

On the other hand

```
print('a = \n', a)
print('\n3+a = \n', 3+a)
print('\n3*a = \n', 3*a)
print('\n\n3/a = \n', 3/a)
print('\na/2 = \n', a/2)
```

```
a =
 [1 2 3 4]

3+a =
 [4 5 6 7]

3*a =
 [ 3  6  9 12]


3/a =
 [3.   1.5  1.   0.75]

a/2 =
 [0.5 1.  1.5 2. ]
```

Unlike in many matrix languages, the product operator `*` operates element-wise in numpy arrays. The matrix product can be performed using the `dot` function or method:

```
A = np.array( [[1, 1], [0, 1]] )
B = np.array( [[2, 3], [1, 4]] )
```

```
print('A')
print(A)
print('\nB')
print(B)
print('\nA*B')
print(A*B)
print('\nAB')
print(np.dot(A,B))
```

```
A
[[1 1]
 [0 1]]

B
[[2 3]
 [1 4]]

A*B
[[2 3]
 [0 4]]

AB
[[3 7]
 [1 4]]
```

Indexing

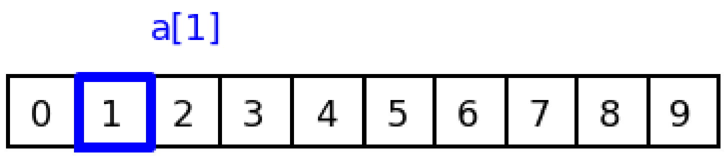
It is similar to the Python indexing. For instance

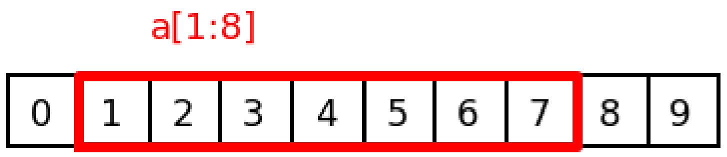
```
a = np.arange(10)
```

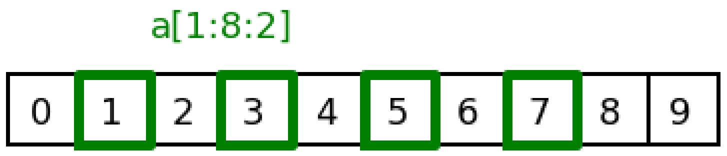
```
print('a\n', a)
print('\na[1]\n', a[1])
print('\na[1:8]\n', a[1:8])
print('\na[1:8:2]\n', a[1:8:2])
print('\na[1:]\n', a[1:])
print('\na[:8]\n', a[:8])
print('\na[::2]\n', a[::2])
print('\na[-1]\n', a[-1])
print('\na[:-1]\n', a[:-1])
print('\na[0]\n', a[0])
print('\na[1:]\n', a[1:])
print('\na[::-1]\n', a[::-1])
print('\na[::-2]\n', a[::-2])
print('\na[7:1:-2]\n', a[7:1:-2])
```

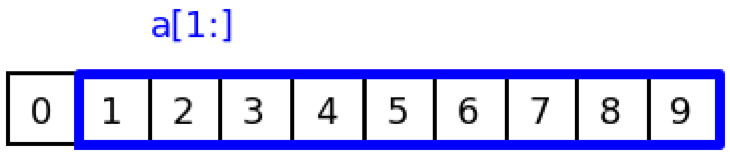
```
a
 [0 1 2 3 4 5 6 7 8 9]

a[1]
 1

a[1:8]
 [1 2 3 4 5 6 7]

a[1:8:2]
 [1 3 5 7]

a[1:]
 [1 2 3 4 5 6 7 8 9]

a[:8]
 [0 1 2 3 4 5 6 7]

a[::2]
 [0 2 4 6 8]

a[-1]
 9

a[:-1]
 [0 1 2 3 4 5 6 7 8]

a[0]
 0

a[1:]
 [1 2 3 4 5 6 7 8 9]

a[::-1]
 [9 8 7 6 5 4 3 2 1 0]

a[::-2]
 [9 7 5 3 1]

a[7:1:-2]
 [7 5 3]
```

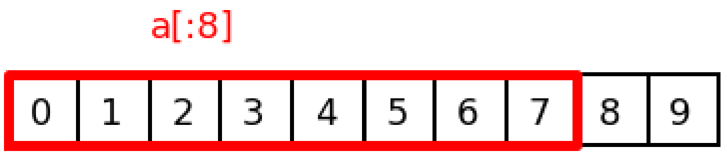
```
%run slicing1
```

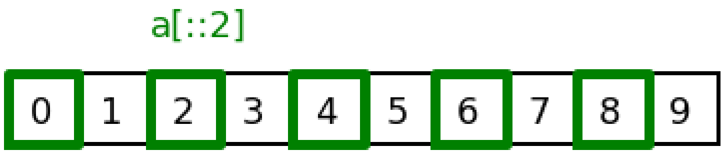

a[1]
0 1 2 3 4 5 6 7 8 9

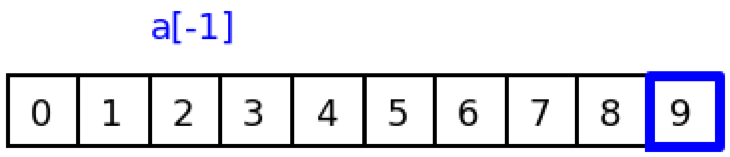

a[1:8]
0 1 2 3 4 5 6 7 8 9

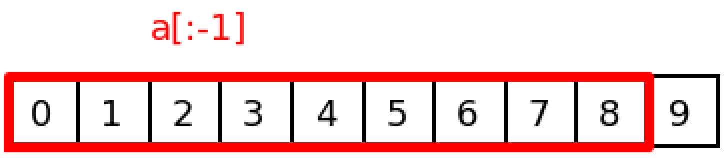

a[1:8:2]
0 1 2 3 4 5 6 7 8 9

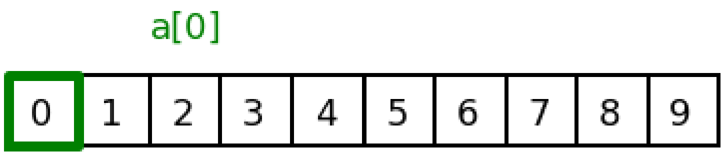

a[1:]
0 1 2 3 4 5 6 7 8 9

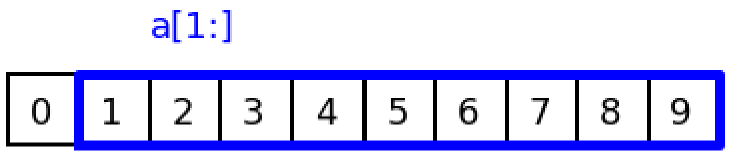

a[:8]
0 1 2 3 4 5 6 7 8 9

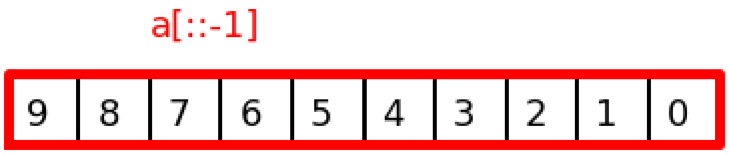

a[::2]
0 1 2 3 4 5 6 7 8 9

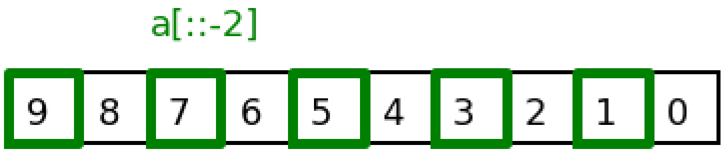

a[-1]
0 1 2 3 4 5 6 7 8 9

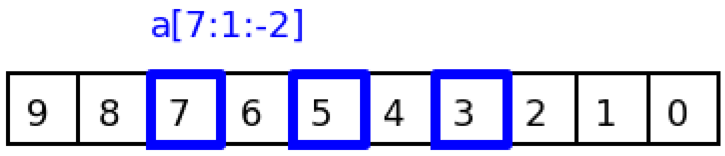












```python
b = np.array([0, 1, 5, -1])
print('\na[b]\n',a[b])
```

```
a[b]
 [0 1 5 9]
```

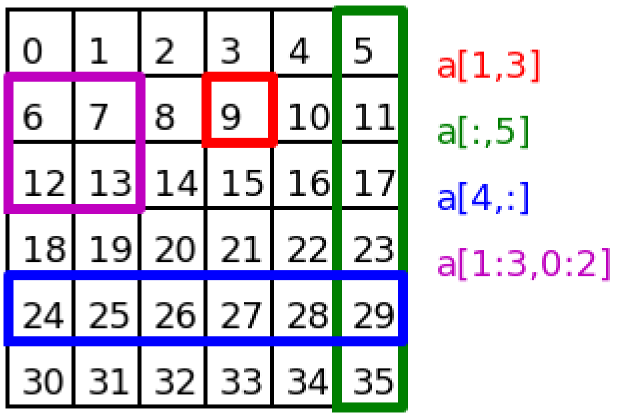
Also, for two-dimensional numpy arrays

```
n = 6
s = n*n
a = np.arange(s)
a = np.reshape(a,(n,n))
```

```
print('a = \n', a)
print('\033[91m \na[1,3] = \n', a[1,3])
print('\033[92m \na[:,5] = \n', a[:,5])
print('\033[94m \na[4,:] = \n', a[4,:])
print('\033[95m \na[1:3, 0:2] = \n', a[1:3, 0:2])
```

```
a =
 [[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 31 32 33 34 35]]

a[1,3] =
 9

a[:,5] =
 [ 5 11 17 23 29 35]

a[4,:] =
 [24 25 26 27 28 29]

a[1:3, 0:2] =
 [[ 6  7]
 [12 13]]
```

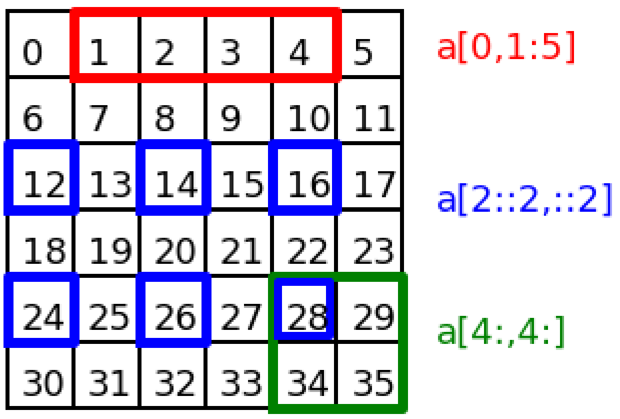
```
%run slicing2
```



```
print('\033[91m \na[0,1:5] = \n', a[0,1:5])
print('\033[92m \na[4:,4:] = \n', a[4:,4:])
print('\033[94m \na[2::2,::2] = \n', a[2::2,::2])
```

```
a[0,1:5] =
 [1 2 3 4]

a[4:,4:] =
 [[28 29]
 [34 35]]

a[2::2,::2] =
 [[12 14 16]
 [24 26 28]]
```

```
%run slicing3
```



Copies

To achieve high performance, assignments in Python usually do not copy the underlaying objects. This is important, for example, when objects are passed between functions, to avoid an excessive amount of memory copying when it is not necessary (technical term: pass by reference).

No Copy at All: Simple assignments make no copy of array objects or their data.

```
a = np.arange(12)
b = a
```

```
print('a[0] = ', a[0], '\nb[0] = ',b[0])
```

```
a[0] =  0
b[0] =  0
```

```
b[0] = 10
```

```
print('a[0] = ', a[0], '\nb[0] = ',b[0])
```

```
a[0] =  10
b[0] =  10
```

Deep copy: The `copy` method makes a complete copy of the array and its data.

```
b = a.copy()
```

```
print('a[0] = ', a[0], '\nb[0] = ',b[0])
```

```
a[0] =  10
b[0] =  10
```

```
b[0] = 0
```

```
print('a[0] = ', a[0], '\nb[0] = ',b[0])
```

```
a[0] =  10
b[0] =  0
```

Elementary functions

Numpy incorpores the elementary functions with their usual names. For instance

```
print(np.sin(PI/2))
print(np.exp(-1))
print(np.arctan(np.inf))
print(np.sqrt(4))
```

```
1.0
0.36787944117144233
1.5707963267948966
2.0
```

If they are applied to a numpy array, the result is a numpy array

```
a = np.linspace(2,4,5)
```

```
print('a =\n', a)
print('\nnp.sqrt(a) =\n', np.sqrt(a))
```

```
a =
 [2.  2.5 3.  3.5 4. ]

np.sqrt(a) =
 [1.41421356 1.58113883 1.73205081 1.87082869 2.        ]
```

Function definition

We can define functions:

- Using a **`lambda`** function.
- Using **`def`**.

```
f1 = lambda x: x ** 3
f2 = lambda x,y: x + y
```

```
print('f1(2) = ', f1(2))
print('f2(1,1) = ', f2(1,1))
```

```
f1(2) =  8
f2(1,1) =  2
```

```
def f3(x):
    if x > 2:
        return 0
    else:
        return 1
```

```
print('f3(-1) = ', f3(-1))
print('f3(3) = ', f3(3))
```

```
f3(-1) =  1
f3(3) =  0
```

For more information, see Numpy Quickstart tutorial (https://numpy.org/doc/stable/user/quickstart.html)

Back to contents

Matplotlib

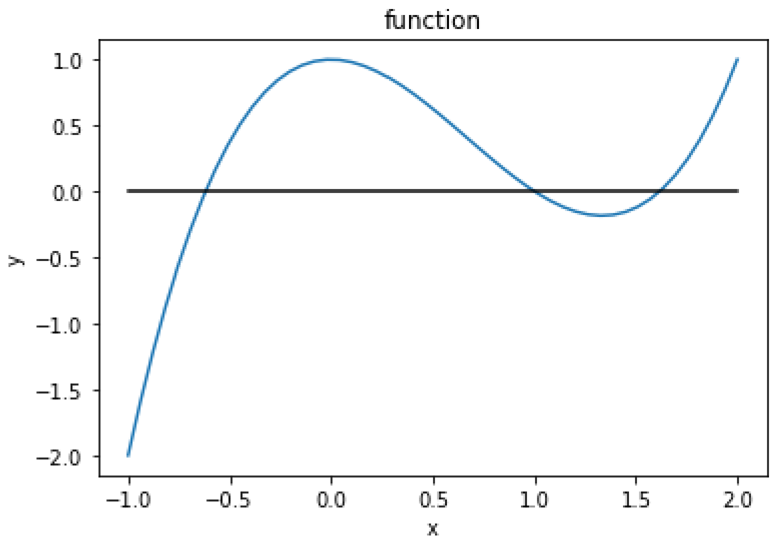
For basic plotting, we use the package **`matplotlib`**

```
import matplotlib.pyplot as plt
```

Simple one-dimensional plotting

```
x = np.linspace(-1,2)               # Define the grid
f = lambda x : x**3 - 2*x**2 + 1    # Define the function
OX = 0*x
```

```
plt.figure()
plt.plot(x,f(x))                    # Plot the function
plt.plot(x,OX,'k-')                 # Plot X axis
plt.xlabel('x')
plt.ylabel('y')
plt.title('function')
plt.show()
```



Back to contents

Two-variable function plotting

We create a grid for x and another for y. Twenty equispaced points for each one.

```
xgrid = np.linspace(-1,1,20)
ygrid = np.linspace(-1,1,20)
```

We construct a bidimensional grid

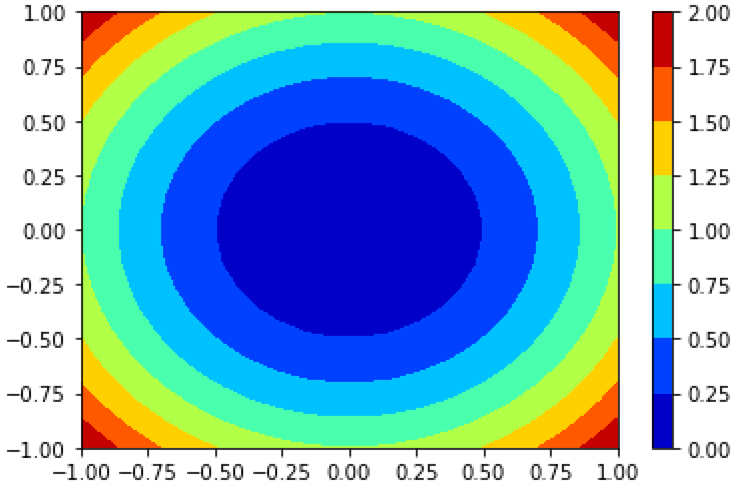
```
X, Y = np.meshgrid(xgrid,ygrid)
```

We define a two-variable function

```
g = lambda x,y: x**2 + y**2
```

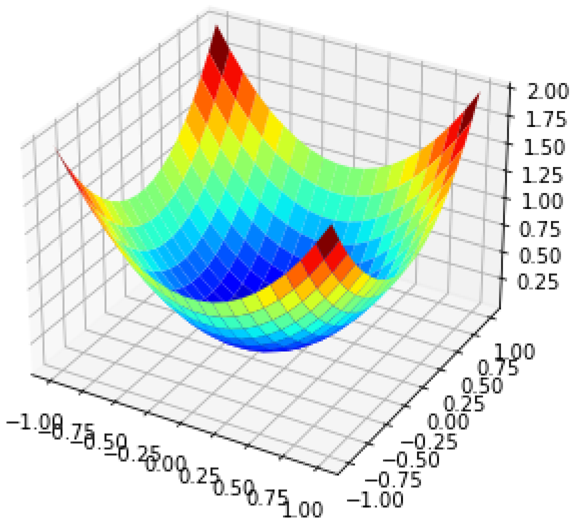
We represent the function with colors

```
plt.figure()
plt.contourf(X,Y,g(X,Y), cmap='jet')
plt.colorbar()
plt.show()
```



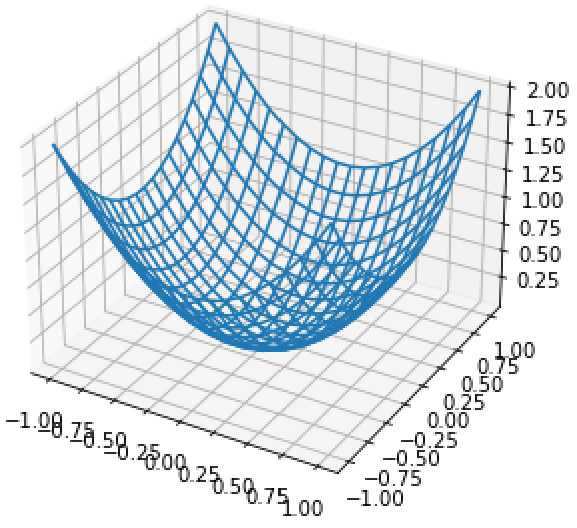
We represent the function as a surface

```
from mpl_toolkits.mplot3d import Axes3D

fig1  = plt.figure(figsize=(10,5))
ax1 = plt.axes(projection ='3d')
ax1.plot_surface(X, Y, g(X,Y), cmap='jet')
plt.show()
```



Or as a wireframe

```
fig2  = plt.figure(figsize=(10,5))
ax2 = plt.axes(projection ='3d')
ax2.plot_wireframe(X, Y, g(X,Y))
plt.show()
```



Back to contents

Exercises

Exercise 1

Create and print the following numpy vectors and matrices:

$$a = (1,\ 3,\ 7) \qquad b = \begin{pmatrix} 2 & 4 & 3 \\ 0 & 1 & 6 \end{pmatrix} \qquad c = (1,\ 1,\ 1) \qquad d = (0,\ 0,\ 0,\ 0)$$

$$e = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \qquad f = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

```
%run Exercise1
```

```
a =
[1 3 7]

b =
[[2 4 3]
 [0 1 6]]

c =
[1. 1. 1.]

d =
[0. 0. 0. 0.]

e =
[[0. 0.]
 [0. 0.]
 [0. 0.]]

f =
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

Exercise 2

Using the `arange` command and then `linspace` command, create the vectors:

- $a = (7,\ 9,\ 11,\ 13,\ 15)$
- $b = (10,\ 9,\ 8,\ 7,\ 6)$
- $c = (15,\ 10,\ 5,\ 0)$
- d is a vector that starts in 0 and ends in 1 and contains 11 equispaced points.
- e is a vector that starts in -1 and ends in 1 and its points divide $[-1, 1]$ into 10 equal intervals.
- f is a vector that starts in 1 and ends in 2 with step 0.1 between points.

Use

```
np.set_printoptions(precision=2,suppress=True)
```

```
%run Exercise2
```

```
a =  [ 7  9 11 13 15]
b =  [10  9  8  7  6]
c =  [15 10  5  0]
d =  [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
e =  [-1.  -0.8 -0.6 -0.4 -0.2 -0.   0.2  0.4  0.6  0.8  1. ]
f =  [1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2. ]


a =  [ 7.  9. 11. 13. 15.]
b =  [10.  9.  8.  7.  6.]
c =  [15. 10.  5.  0.]
d =  [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
e =  [-1.  -0.8 -0.6 -0.4 -0.2  0.   0.2  0.4  0.6  0.8  1. ]
f =  [1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2. ]
```

Exercise 3

- Using `arange`, create the vector

$$v = (0.0,\ 1.1,\ 2.2,\ 3.3,\ 4.4,\ 5.5,\ 6.6,\ 7.7,\ 8.8,\ 9.9,\ 11.0,\ 12.1)$$

- Using *slicing* obtain, from v, the vector v_i

$$v_i = (12.1,\ 11.0,\ 9.9,\ 8.8,\ 7.7,\ 6.6,\ 5.5,\ 4.4,\ 3.3,\ 2.2,\ 1.1,\ 0.0)$$

- Using *slicing* obtain, from v, the vectors v_1 and v_2

$$\begin{aligned} v_1 &= (0.0,\ 2.2,\ 4.4,\ 6.6,\ 8.8,\ 11.) \\ v_2 &= (1.1,\ 3.3,\ 5.5,\ 7.7,\ 9.9,\ 12.1) \end{aligned}$$

- Using *slicing* obtain, from v, the vectors v_1, v_2 and v_3

$$\begin{aligned} v_1 &= (0.0,\ 3.3,\ 6.6,\ 9.9) \\ v_2 &= (1.1,\ 4.4,\ 7.7,\ 11.0) \\ v_3 &= (2.2,\ 5.5,\ 8.8,\ 12.1) \end{aligned}$$

- Using *slicing* obtain, from v, the vectors v_1, v_2, v_3 and v_4.

$$\begin{aligned} v_1 &= (0.0,\ 4.4,\ 8.8) \\ v_2 &= (1.1,\ 5.5,\ 9.9) \\ v_3 &= (2.2,\ 6.6,\ 11.0) \\ v_4 &= (3.3,\ 7.7,\ 12.1) \end{aligned}$$

```
%run Exercise3
```

```
v =  [ 0.   1.1  2.2  3.3  4.4  5.5  6.6  7.7  8.8  9.9 11.  1
2.1]

vi =  [12.1 11.   9.9  8.8  7.7  6.6  5.5  4.4  3.3  2.2  1.1
0. ]

v1 =  [ 0.   2.2  4.4  6.6  8.8 11. ]
v2 =  [ 1.1  3.3  5.5  7.7  9.9 12.1]

v1 =  [0.  3.3 6.6 9.9]
v2 =  [ 1.1  4.4  7.7 11. ]
v3 =  [ 2.2  5.5  8.8 12.1]

v1 =  [0.  4.4 8.8]
v2 =  [1.1 5.5 9.9]
v3 =  [ 2.2  6.6 11. ]
v4 =  [ 3.3  7.7 12.1]
```

Exercise 4

Starting with the vector $a=(1,2,3)$ create a vector $b=(0,1,2,3,0)$

1. Using np.append (https://numpy.org/doc/stable/reference/generated/numpy.append.html) twice: add a zero at the end and then flip the vector, add the other zero and flip it again.
2. Create the vector $b=(0,0,0,0,0)$ and inserting the vector a with slicing.
3. Using np.concatenate (https://numpy.org/doc/stable/reference/generated/numpy.concatenate.html) using a vector $c=(0)$.

```
%run Exercise4
```

```
1.
b =  [0. 1. 2. 3. 0.]

2.
b =  [0. 1. 2. 3. 0.]

3.
b =  [0. 1. 2. 3. 0.]
```

Exercise 5

Given the matrix

$$A=\begin{pmatrix} 2 & 1 & 3 & 4\\ 9 & 8 & 5 & 7\\ 6 & -1 & -2 & -8\\ -5 & -7 & -9 & -6 \end{pmatrix}$$

From A, get the following matrices:

$$a=\begin{pmatrix} 2\\ 9\\ 6\\ -5 \end{pmatrix} \qquad b=\begin{pmatrix} 6 & -1 & -2 & -8 \end{pmatrix} \qquad c=\begin{pmatrix} 2 & 1\\ 9 & 8 \end{pmatrix}$$

$$d=\begin{pmatrix} -2 & -8\\ -9 & -6 \end{pmatrix} \qquad e=\begin{pmatrix} 8 & 5\\ -1 & -2 \end{pmatrix} \qquad f=\begin{pmatrix} 1 & 3 & 4\\ 8 & 5 & 7\\ -1 & -2 & -8\\ -7 & -9 & -6 \end{pmatrix} \qquad g=\begin{pmatrix} 8 & 5\\ -1 & -2\\ -7 & -9 \end{pmatrix}$$

```
%run Exercise5
```

```
A =
[[ 2  1  3  4]
 [ 9  8  5  7]
 [ 6 -1 -2 -8]
 [-5 -7 -9 -6]]

a =
[ 2  9  6 -5]

b =
[ 6 -1 -2 -8]

c =
[[2 1]
 [9 8]]

d =
[[-2 -8]
 [-9 -6]]

e =
[[ 8  5]
 [-1 -2]]

f =
[[ 1  3  4]
 [ 8  5  7]
 [-1 -2 -8]
 [-7 -9 -6]]

g =
[[ 8  5]
 [-1 -2]
 [-7 -9]]
```

Exercise 6

Write the following functions and compute their value at the points indicated:

$f(x) = x\,e^x$ at $x = 2$ $\qquad g(z) = \dfrac{z}{\sin z \cos z}$ at $z = \pi/4$ $\qquad h(x,y) = \dfrac{xy}{x^2+y^2}$ a

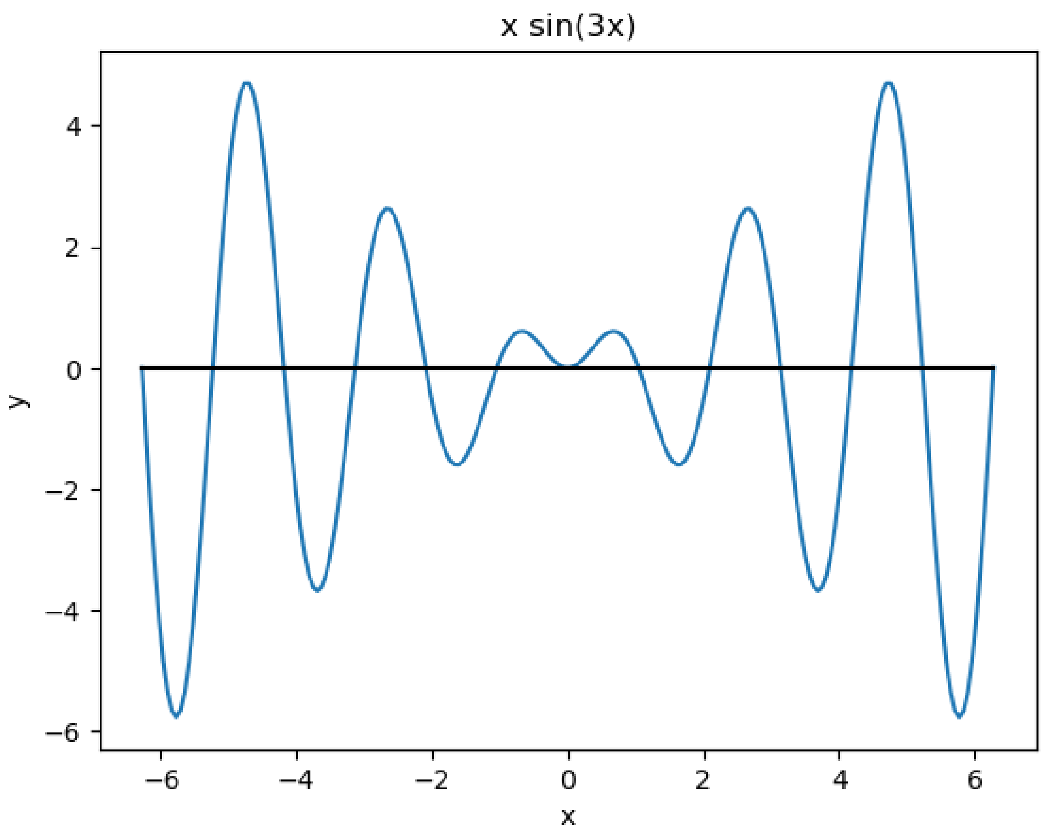
```
%run Exercise6
```

```
f(2) =  14.7781121978613

g(pi/4) =  1.5707963267948966

h(2,4) =  0.4
```

Exercise 7

Plot the function $f(x) = x\,\sin(3x)$ in the interval $[-2\pi, 2\pi]$

```
%run Exercise7
```



References

- Style Guide for Python Code (https://www.python.org/dev/peps/pep-0008)
- NumPy Quickstart tutorial (https://numpy.org/doc/stable/user/quickstart.html)
- Download Anaconda Distribution (https://www.anaconda.com/download)