Week 0 - Topic 9

Authors: Sai Gayatri Vadali | Julián Darío Miranda.

`Numpy`, Scientific and numeric library

`Numerical Python` popularly known as `Numpy` is core library for scientific computing. It provides high
object and tools for working with these objects.

In the following sections, we will be working on the `Numpy` library and its uses. We will compare the pe
with their classic `Python` equivalent and discuss the uses of the library. Let us begin!

1. Comparing `Python` List vs `Numpy` array

Consider the following snippet adapted from https://webcourses.ucf.edu/courses/. We start importin
as we have stated, enables scientific computing and high performance tasks executions with multi-di
`Timer` module from `timeit` library is used for the purpose of calculating the time that certain scripts

```
import numpy as np
from timeit import Timer
```

With imported libraries, we proceed to create two example arrays with numbers in the range [0, 9999]:

```
size_of_vec = 10000
X_list = range(size_of_vec)
Y_list = range(size_of_vec)
X = np.arange(size_of_vec)
Y = np.arange(size_of_vec)
```

To compare the performance of two scripts, one explicitly developed with `Python` and another with `N

```
def pure_python_version(): #Explicitly Python based with lists
    Z = []
    for i in range(len(X_list)):
        Z.append(X_list[i] + Y_list[i])

def numpy_version(): #Explicitly Numpy based with vectorization
    Z = X + Y
```

We call the developed functions and measure the time it takes to execute them once:

```
timer_obj1 = Timer("pure_python_version()",
                   "from __main__ import pure_python_version")
timer_obj2 = Timer("numpy_version()",
                   "from __main__ import numpy_version")

print("Pure python version:",timer_obj1.timeit(10))
print("Numpy version:",timer_obj2.timeit(10))
```

```
Pure python version: 0.07681977800001505
Numpy version: 0.0005145959999595107
```

As we can see, the vectorized sum approach with `Numpy` is much faster than the purely `Python` basec

2.Creating `Numpy` array from `Python` list

Let's start by creating a `numpy` array from a predefined list with te values 165, 170, 171, 180, 189, and

```
heights = [165, 170, 171, 180, 189, 178]
print(type(heights))

heights_np = np.array(heights)
print(type(heights_np))
```

```
<class 'list'>
<class 'numpy.ndarray'>
```

The type of object defined at first is a `list`. After conversion with the `.array()` function, the object is
means that we have created an array of multiple dimensions (`n` dimensions) from a `list`. In this cas
array.

Now let's see how to create a **two-dimensional array**:

```
weights = np.array([[50, 45, 56, 78],[78, 89, 59, 90],[89, 78, 69, 70],[67, 69, 89, 70],[90,8
print(weights)
```

```
[[50 45 56 78]
 [78 89 59 90]
 [89 78 69 70]
 [67 69 89 70]
 [90 89 80 84]
 [89 59 90 78]]
```

3. Exploring some of the key attributes of `ndarray` objects

Multidimensional arrays have the following important attributes:

- `ndim` : number of dimensions of the array
- `shape` : shape of the array in the format `(number_rows, number_columns)`
- `size` : total number of elements
- `dtypes` : type of data stored in the array
- `strides` : number of bytes that must be moved to store each row and column in memory, in the `number_bytes_columns)`

Let's see an example:

```
print("dimension:", weights.ndim)
print("shape:", weights.shape)
print("size:", weights.size)
print("dtype:", weights.dtype)
print("strides:", weights.strides)
```

```
dimension: 2
shape: (6, 4)
size: 24
dtype: int64
strides: (32, 8)
```

The exemplified arrangement has:

- `ndim` of 2 because it is a two-dimensional array.
- `shape` of (6, 4) as it is made up of 6 rows and 4 columns.
- `size` of 24 since it has 24 elements in total, 6 elements per column or what is the same, 4 elem
- int32 `dtypes` because each element of the array is a 32-bit (4-byte) integer
- `strides` of (16, 4) since 16 bytes (4 integers of 4 bytes in the rows) are needed to store each ro column) to store each column in memory.

Exercise 1

Convert the two-dimensional `ndarray` `weights` into a three-dimensional object without changing its s

```
# Answer
```

4. Exploring some of the key functions defined for `numpy` arrays

In this section we are going to explore some of the most important functions of `numpy` arrays:

1. `zeros(shape=(n,m))` : Allows to create a zero-array with the shape (`n` rows, `m` columns)
2. `arange(start=i, stop=j, step=u)` : creates a one-dimensional array whose first value is `i` incl value varies `s` steps from the previous.
3. `linspace(start=i, stop=j, num=n)` : creates a one-dimensional array whose first value is `i` inc contains `n` values in total. Each value differs from the previous one with the same magnitude th
4. `full(shape=(n,m), fill_value=f)` : Allows to create an array with the shape (`n` rows, `m` columr

Let's delve into each of them:

4.1. np.zeros()

The `zeros(shape=(n,m), dtypes)` function creates a zero-array with the shape (`n` rows, `m` columns)

```
x = np.zeros(shape=(3,5), dtype ="int32")
print(x)
```

```
[[0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]
```

As you can notice, we have created a two-dimensional array of zeros of three rows and five columns v

4.2. np.arange()

The `arange(start=i, stop=j, step=u)` function creates a one-dimensional array whose first value is each value varies `s` steps from the previous:

```
x = np.arange(start=100, stop=1000, step=100, dtype="int32")
print(x)
```

```
[100 200 300 400 500 600 700 800 900]
```

This function has allowed us to create a one-dimensional array that starts at 100, ends at 1000 (exclu with 32-bit integer values.

4.3. np.linspace()

The `linspace(start=i, stop=j, num=n)` function creates a one-dimensional array whose first value i
contains `n` values in total. Each value differs from the previous one with the same magnitude that dif
example:

```
x_lin = np.linspace(start=10, stop=50, num=30)
print(x_lin)
```

```
[10.         11.37931034 12.75862069 14.13793103 15.51724138 16.89655172
 18.27586207 19.65517241 21.03448276 22.4137931  23.79310345 25.17241379
 26.55172414 27.93103448 29.31034483 30.68965517 32.06896552 33.44827586
 34.82758621 36.20689655 37.5862069  38.96551724 40.34482759 41.72413793
 43.10344828 44.48275862 45.86206897 47.24137931 48.62068966 50.        ]
```

We have created a one-dimensional array that varies linearly from 10 to 50 inclusive, for a total of 30 f

4.4. np.full()

The `full(shape=(n,m), fill_value=f)` function allows to create an array with the shape (`n` rows, `m`
value `f`.

```
x_ful = np.full(shape=(5,6), fill_value=3)
print(x_ful)
```

```
[[3 3 3 3 3 3]
 [3 3 3 3 3 3]
 [3 3 3 3 3 3]
 [3 3 3 3 3 3]
 [3 3 3 3 3 3]]
```

We see that a two-dimensional array of 5 rows and 6 columns has been created, all with a value of 3 a

5. Exploring additional attributes and functions

Let's review three additional functions: `.reshape()`, `.flatten()` and `.ravel()`.

5.1. Reshaping the array

Let's reshape the `weights` numpy array. First take a look to the contant and the shape of `weights`:

```
weights
```

```
array([[50, 45, 56, 78],
       [78, 89, 59, 90],
       [89, 78, 69, 70],
       [67, 69, 89, 70],
       [90, 89, 80, 84],
       [89, 59, 90, 78]])
```

```
weights.shape
```

```
(6, 4)
```

The reshaping procedure is done using the `.reshape((n1,m1))` function, which receives as input para
that is, the new shape of the array to be created from the original array:

```
weights = weights.reshape((4,6))
weights
```

```
array([[50, 45, 56, 78, 78, 89],
       [59, 90, 89, 78, 69, 70],
       [67, 69, 89, 70, 90, 89],
       [80, 84, 89, 59, 90, 78]])
```

```
weights.shape
```

```
(4, 6)
```

Can you see the difference? We have changed the shape of the `weights` array, from 6 rows and 4 colu
the values are distributed in the new array is from left-to-right then top-to-bottom.

Let's add a new dimension through reshaping the current `weights` array:

```
weights = weights.reshape((2,6,2))
weights
```

```
array([[[50, 45],
        [56, 78],
        [78, 89],
        [59, 90],
        [89, 78],
        [69, 70]],

       [[67, 69],
        [89, 70],
        [90, 89],
        [80, 84],
        [89, 59],
        [90, 78]]])
```

```
weights.shape
```

```
(2, 6, 2)
```

Now, the array is three-dimensionally estructured. Two bi-dimensional (2D) arrays conform the new ar
rows and six columns.

5.2. Flattening the array

The `.flatten()` function returns a copy of an array collapsed into one dimension, no matter how mai
`weights` two-dimensional array:

```
weights
```

```
array([[[50, 45],
        [56, 78],
        [78, 89],
        [59, 90],
        [89, 78],
        [69, 70]],

       [[67, 69],
        [89, 70],
        [90, 89],
        [80, 84],
        [89, 59],
        [90, 78]]])
```

If we flatten the array, we are re-organizing their elements in a one-dimensional array, as follows:

```
weights_flattened = weights.flatten()
weights_flattened
```

```
array([50, 45, 56, 78, 78, 89, 59, 90, 89, 78, 69, 70, 67, 69, 89, 70, 90,
       89, 80, 84, 89, 59, 90, 78])
```

5.3. Raveling the array

The `.ravel()` function returns a flattened view of an array collapsed into one dimension. It works ide
although a copy in memory is not achieve, just a flatten view of the final result.

Consider the `weights` two-dimensional array:

```
weights_raveld = weights.ravel()
```

```
weights_raveld
```

```
array([50, 45, 56, 78, 78, 89, 59, 90, 89, 78, 69, 70, 67, 69, 89, 70, 90,
       89, 80, 84, 89, 59, 90, 78])
```

Some key differences between flattening and raveling the array are:

- `ravel()` function simply returns a flattened view of Numpy array. If you try to modify this view, y the original array. `flatten()` function returns a flattened copy in memory of the array, so that ne the original array.
- `ravel()` does not occupy memory, being faster than `flatten()`, which occupies memory when

Exercise 2

Create an array of 51 elements starting at 100 and ending at 500, using the two functions `np.linspac` the same content, with the names `array_lin` and `array_ara`, respectively. Verify that the arrays have `np.array_equal()` function.

```
# Answer
```

6.Array indexing

To access the content of an array we can use indexing through brackets `[ ]`. When using the bracket by:

1. Using a **positive single index** starting from 0
2. Using a **negative single index** starting from -1
3. Using **positive index intervals** using the `start:end:step` notation to specify starting and ending
4. Using **negative index intervals** using the `start:end:step` notation to specify negative index sta step.

Occasionally, we can:

- Get ride of the the `step` value as `sart:end`, so that by default we slice the data with a `step` of 1
- Get ride of the `start` value as `:fin:step`, and hence our `start` index will be 0, by default.
- Omit the `end` value as `start::step`, specifying the final position as the `end` index by default.
- Specify the range as `::step`, and hence the `start` position will be 0 and the end position will be

Let's see how indexing works using the `weights` and `heights_np` arrays:

```
weights
```

```
array([[[50, 45],
        [56, 78],
        [78, 89],
        [59, 90],
        [89, 78],
        [69, 70]],

       [[67, 69],
        [89, 70],
        [90, 89],
        [80, 84],
        [89, 59],
        [90, 78]]])
```

```
weights_or = weights.reshape((6,4))
weights_or
```

```
array([[50, 45, 56, 78],
       [78, 89, 59, 90],
       [89, 78, 69, 70],
       [67, 69, 89, 70],
       [90, 89, 80, 84],
       [89, 59, 90, 78]])
```

```
heights_np
```

```
array([165, 170, 171, 180, 189, 178])
```

6.1 Using a positive single index

When using positive indexing, it is important to consider the first position of the array to be 0:

```
print("Accessing single element in 1D array:", heights_np[2])
print("Accessing single element in 2D array:", weights_or[1][3])
```

```
Accessing single element in 1D array: 171
Accessing single element in 2D array: 90
```

```
heights_np[7]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-35-c1b0cf84b168> in <module>()
```

Why are we getting this error message?

Well guessed! It is because position 7 does not exist in the `heights_np` array, it is totally out of the bo size. The array has 6 elements, the last element being in position 5.

6.2 Using a negative single index starting

When using negative indexing, it is important to consider the last position of the array to be -1:

```
print("Accessing single element in 1D array:", heights_np[-4])
print("Accessing single element in 2D array:", weights_or[-5][-1])
```

```
Accessing single element in 1D array: 171
Accessing single element in 2D array: 90
```

```
heights_np[-8]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-38-7c81311360da> in <module>()
----> 1 heights_np[-8]

IndexError: index -8 is out of bounds for axis 0 with size 6
```

SEARCH STACK OVERFLOW

Why are we getting this error message again?

Well guessed! It is because position -8 does not exist in the `heights_np` array, it is totally out of the bo size. The array has 6 elements, the last element being in position -1 and the first element being in pos

6.3 Using positive index intervals

When using positive interval indexing `start:end:step`, the starting value is inclusive and the ending v examples:

```
heights_np[:2] # The default start value is 0
```

```
array([165, 170])
```

```
heights_np[2:] # The default end value is the last value of the array
```

```
array([171, 180, 189, 178])
```

```
heights_np[2:3] # The ending value is exlusive
```

```
array([171])
```

```
weights[:2, ::2]
```

```
array([[[50, 45],
        [78, 89],
        [89, 78]],

       [[67, 69],
        [90, 89],
        [89, 59]]])
```

```
weights[:3, 3::]
```

```
array([[[59, 90],
        [89, 78],
        [69, 70]],

       [[80, 84],
        [89, 59],
        [90, 78]]])
```

```
weights[:3, :3, :1]
```

```
array([[[50],
        [56],
        [78]],

       [[67],
        [89],
        [90]]])
```

6.4 Using negative index intervals

When using positive interval indexing `start:end:step`, the negative starting value is inclusive and the
are somoe examples:

```
heights_np[:-4] # Equivalent to heights_np[:2]
```

```
array([165, 170])
```

```
heights_np[-4:] # Equivalent to heights_np[2:]
```

```
array([171, 180, 189, 178])
```

```
heights_np[-4:-3] # Equivalent to heights_np[2:3]
```

```
array([171])
```

```
weights[:2, -3::] # Equivalent to weights[:3, 3::]
```

```
array([[[59, 90],
        [89, 78],
        [69, 70]],

       [[80, 84],
        [89, 59],
        [90, 78]]])
```

```
weights[:3, :-3, :-1] # Equivalent to weights[:3, :3, :1]
```

```
array([[[50],
        [56],
        [78]],

       [[67],
        [89],
        [90]]])
```

Exercise 3

Consider the `weights` array:

1. Select all the values that are in the even positions in the rows and in the odd positions in the col `weights_custom1` with these values.
2. Express the `weights_custom1` array flattened with an in-memory copy. Call the new array `weight`
3. Select items in positions 2 to 4 inclusive with negative indexing. Name the output array as `weig`

```
# Answer #1
```

```
# Answer #2
```

```
# Answer #3
```

7.Manipulating `Numpy` arrays

Arrays can be manipulated using arithmetic, logical, or relational operations in an element-wise way. L
using our arrays `weights_or`, `heights_np`, and and some other arrays that we will create.

7.1. Arithmetic operations

We are going to operate the content of the arrays with the four traditional arithmetic operations, addit
Let's first define our arrays again:

```
weights_or
```

```
array([[50, 45, 56, 78],
       [78, 89, 59, 90],
       [89, 78, 69, 70],
       [67, 69, 89, 70],
       [90, 89, 80, 84],
       [89, 59, 90, 78]])
```

```
heights_2 = np.array([165, 175, 180, 189, 187, 186])
print('heights_np:', heights_np)
print('heights_2: ', heights_2)
```

```
heights_np: [165 170 171 180 189 178]
heights_2:  [165 175 180 189 187 186]
```

Let's add the content of the two arrays element-wise:

```
heights_add = heights_np + heights_2
heights_add
```

```
array([330, 345, 351, 369, 376, 364])
```

The `np.add()` function allows adding the content of arrays element-wise:

```
added = np.add(heights_2, heights_np)
added
```

```
array([330, 345, 351, 369, 376, 364])
```

Exercise 4

Since we have seen how to add element-wise elements of one-dimensional arrays:

1. Calculate the subtraction, multiplication and division between the `heights_np` and `heights_2` a functions: `np.subtract()`, `np.multiply()`, and `np.divide()`.
2. Calculate the product element-wise of the multiplicative inverses $(1/x)$ between the arrays `hei` functions. For instante, if an element in `heights_np` $x1 = 5$ and an element in `heights_2` $y1 =$ $z = (1/x1) * (1/y1) = 1/20 = 0.05$.

```
# Answer substraction
```

```
# Answer multiplication
```

```
# Answer division
```

```
# Answer multiplicative inverse
```

7.2. Logical operations

Logical operations are mathematical expressions whose result is a Boolean value of 0 (False) or 1 (Tr operations are the disjunction `or`, conjunction `and`, and negation `not` operations, among others. Let's

```
x = np.array([True, True, False, False])
y = np.array([True, False, True, False])
```

```
np.logical_or(x,y)
```

```
array([ True,  True,  True, False])
```

```
np.logical_and(x,y)
```

```
array([ True, False, False, False])
```

```
np.logical_not(x)
```

```
array([False, False,  True,  True])
```

7.3.Comparison - Relational operators

The comparison operators allow us to compare the values of the content of `numpy` arrays element-wi operator `np.equal()`, (b) less than operator `np.less()` / `np.less_equal()`, (c) greater than operator r

(d) difference operator `np.not_equal()`. It is important to note that the output will always be Boolean

Let's see some examples:

```
x = np.array([1, 8, 3, 7, 3, 21])
y = np.array([4, 8, 1, 7, 6, 9])
```

```
np.equal(x,y)
```

```
array([False,  True, False,  True, False, False])
```

```
np.not_equal(x,y)
```

```
array([ True, False,  True, False,  True,  True])
```

```
np.less_equal(x,y)
```

```
array([ True,  True, False,  True,  True, False])
```

```
np.greater_equal(x,y)
```

```
array([False,  True,  True,  True, False,  True])
```

```
np.array_equal(x,y) # Comparing the entire content of both arrays
```

```
False
```

```
x = np.array([1, 8, 3, 7, 3, 21])
y = np.array(list((1, 8, 3, 7, 3, 21)))
np.array_equal(x,y) # Comparing the entire content of both arrays
```

```
True
```

8. Broadcasting

`numpy` has the ability of operating arrays of different shapes during arithmetic operations using **broad**
arrays are done on corresponding elements. The boradcasting operation replicates one of the arrays a
a mismatch of shapes. Consider the following arrays:

```
heights_np = heights_np.reshape((6,1))
heights_np
```

```
array([[165],
       [170],
       [171],
       [180],
       [189],
       [178]])
```

```
weights
```

```
array([[[50, 45],
        [56, 78],
        [78, 89],
        [59, 90],
        [89, 78],
        [69, 70]],

       [[67, 69],
        [89, 70],
        [90, 89],
        [80, 84],
        [89, 59],
        [90, 78]]])
```

We are going to add the elements of both arrays:

```
broad_np = heights_np + weights
broad_np
```

```
array([[[215, 210],
        [226, 248],
        [249, 260],
        [239, 270],
        [278, 267],
        [247, 248]],

       [[232, 234],
        [259, 240],
        [261, 260],
        [260, 264],
        [278, 248],
        [268, 256]]])
```

Although the arrays have different dimensions, `numpy` makes a sum for the corresponding elements i
way that the elements of the column vector `heights_np` are added with each column of the two-dime

Let's look at one more example:

```
x = np.ones((3,4))
y = np.random.random((5,1,4))
```

```
x
```

```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

```
y
```

```
array([[[0.58650573, 0.8991317 , 0.11794428, 0.95529092]],

       [[0.90979584, 0.51435102, 0.54402438, 0.91886755]],

       [[0.67730039, 0.08366353, 0.03533193, 0.72036147]],

       [[0.32999286, 0.05300852, 0.70901024, 0.48472041]],

       [[0.8319779 , 0.46926   , 0.40650511, 0.49699316]]])
```

```
z = x + y
z
```

```
array([[[1.58650573, 1.8991317 , 1.11794428, 1.95529092],
        [1.58650573, 1.8991317 , 1.11794428, 1.95529092],
        [1.58650573, 1.8991317 , 1.11794428, 1.95529092]],

       [[1.90979584, 1.51435102, 1.54402438, 1.91886755],
        [1.90979584, 1.51435102, 1.54402438, 1.91886755],
        [1.90979584, 1.51435102, 1.54402438, 1.91886755]],

       [[1.67730039, 1.08366353, 1.03533193, 1.72036147],
        [1.67730039, 1.08366353, 1.03533193, 1.72036147],
        [1.67730039, 1.08366353, 1.03533193, 1.72036147]],

       [[1.32999286, 1.05300852, 1.70901024, 1.48472041],
        [1.32999286, 1.05300852, 1.70901024, 1.48472041],
        [1.32999286, 1.05300852, 1.70901024, 1.48472041]],

       [[1.8319779 , 1.46926   , 1.40650511, 1.49699316],
        [1.8319779 , 1.46926   , 1.40650511, 1.49699316],
        [1.8319779 , 1.46926   , 1.40650511, 1.49699316]]])
```

Here each row in array `y` has been paired with rows in array `x`, since they have the same amount of c rows of array `y` and the same number of columns.

Exercise 5

Propose an array `y` such that the operation $x + y$ results in the array `z`.

```
x = [[14, 15, 18],
     [62, 90, 98],
     [71, 73, 90],
```

```
     [40, 24, 17],
     [11, 81, 14],
     [26, 81, 31]]

 z = [[24,  40,  58],
     [72, 115, 138],
     [81,  98, 130],
     [50,  49,  57],
     [21, 106,  54],
     [36, 106,  71]]
```

```
# Answer
```

9.Matrix multiplication

Let's delve into the element-wise and dot product multiplication between matrices (two-dimensional a

```
A = np.array([[1,1,8],[0,1,9],[9,0,8]])
print("Matrix A:\n", A, '\n')

B = np.array([[2,0,0],[3,4,9],[7,8,9]])
print('MATRIX B:\n', B, '\n')
```

```
Matrix A:
 [[1 1 8]
 [0 1 9]
 [9 0 8]]

MATRIX B:
 [[2 0 0]
 [3 4 9]
 [7 8 9]]
```

The product between the two matrices can be executed with the classic arithmetic operator `*`:

```
print("Element wise multiplication:\n", A*B, '\n')
```

```
Element wise multiplication:
 [[ 2  0  0]
 [ 0  4 81]
 [63  0 72]]
```

The dot product of matrices can be executed with the `@` operator or with the numpy `np.dot()` functic

```
print("Matrix product:\n", A@B, '\n') # matrix A = (2 ,3) , matrix B= (3,4), output matrix =(
print("Dot product:\n", A.dot(B), '\n')
```

```
Matrix product:
 [[61 68 81]
 [66 76 90]
 [74 64 72]]

Dot product:
 [[61 68 81]
 [66 76 90]
 [74 64 72]]
```

10. Arrays with `random` numbers

A random number is a result of a variable combination specified by a distribution function. When no d
the continuous uniform distribution in the interval [0,1) is used. Some functions for generating randon

- `np.random.random()` : returns random floats in the half-open interval [0.0, 1.0)
- `np.random.randint(low, high)` : returns random integers from low (inclusive) to high (exclusive
- `np.random.normal()` : returns random samples from a normal (Gaussian) distribution.

Let's see some examples:

```
np.random.random((4,3))
```

```
array([[0.11798973, 0.62090018, 0.25967593],
       [0.43534172, 0.39078729, 0.29537843],
       [0.62527037, 0.36664864, 0.90304918],
       [0.80110352, 0.06759967, 0.24346256]])
```

```
np.random.randint(10, 20, size=(2, 4))
```

```
array([[18, 14, 15, 16],
       [15, 13, 15, 19]])
```

```
np.random.normal(size=10)
```

```
array([ 0.0446794 ,  0.04824569, -0.34208871,  0.50114241,  1.35975059,
       -0.56957288,  1.0875766 , -0.09560878,  0.51696095,  1.74689523])
```

Ee can also specify a `seed`, so that the sequence of random numbers is repeatable (if you execute th

```
from numpy.random import seed
from numpy.random import rand

# Seed random number generator
seed(42)

# Generate random numbers between 0-1
values = rand(10)
print(values)
```

```
[0.37454012 0.95071431 0.73199394 0.59865848 0.15601864 0.15599452
 0.05808361 0.86617615 0.60111501 0.70807258]
```

11. Concatenate, and stack `Numpy` arrays

The **concatenation** `np.concatenate()` is a process of joining several arrays to form one, on the same joining arrays on a new axis. Let's dive a little bit more on this concepts with practical examples:

```
my_array = np.array([1,2,34,5])
x = np.array([1,4,5,6])
print('x: \t  ', x)
print('my_array: ', my_array)
```

```
x:          [1 4 5 6]
my_array:  [ 1  2 34  5]
```

```
print('Append:\n',np.append(my_array,x))
y = np.append(my_array, x)

# Concatentate `my_array` and `x`
print('\nConcatenate:\n',np.concatenate((my_array,x)))
```

```
Append:
 [ 1  2 34  5  1  4  5  6]

Concatenate:
 [ 1  2 34  5  1  4  5  6]
```

```
# Stack arrays vertically (row-wise)
print("Stack row wise:")
print(np.vstack((my_array, x)))
```

```
Stack row wise:
[[ 1  2 34  5]
 [ 1  4  5  6]]
```

```
# Stack arrays horizontally
print("Stack horizantally:")
print(np.hstack((my_array,x)))

print("\nAnother way:")
print(np.r_[my_array,x])
```

```
Stack horizantally:
[ 1  2 34  5  1  4  5  6]

Another way:
[ 1  2 34  5  1  4  5  6]
```

```
# Stack arrays column-wise
print("Stack column wise:")
print(np.column_stack(( my_array,x)))

print("\nColumn wise repeat:")
print(np.c_[ my_array,x])
```

```
Stack column wise:
[[ 1  1]
 [ 2  4]
 [34  5]
 [ 5  6]]

Column wise repeat:
[[ 1  1]
 [ 2  4]
 [34  5]
 [ 5  6]]
```

As you have seen, when we concatenate the arrays, we do it on the same axis. When stacking arrays,

12. Visualize `Numpy` array

To visualize the content of a numpy array we can make use of the `matplotlib.pyplot` library, which a distributions, among many other functions.

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

Let's specify an initial state for the Mersenne Twister number generator, a pseudo-random number ge

```
rng = np.random.RandomState(10)
```

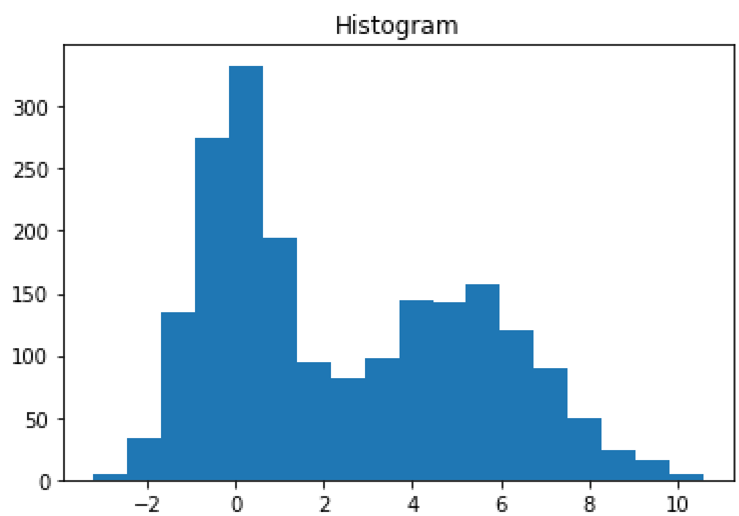
Now we generate random values of two normal distributions with different mean and standard deviat of mean 5, stacking them in a single array horizontally:

```
a = np.hstack((rng.normal(size=1000),rng.normal(loc=5, scale=2, size=1000)))
a
```

```
array([ 1.3315865 ,  0.71527897, -1.54540029, ...,  5.74446677,
        6.22449239,  8.42055014])
```

Let's visualize the data of the number arrangement of the two distributions, in a histogram with the he which we have aliased `plt`:

```
plt.hist(a, bins='auto')
plt.title("Histogram")
plt.show()
```



As can be seen, this graph denotes the distribution of the two normal distributions with a mean of 0 a

As an additional example, we are creating a meshgrid `np.meshgrid()` with values generated from an value of -5 (exclusive) and step of 0.01. We have calculated the value of `z` which corresponds to the g generate the graph shown below:

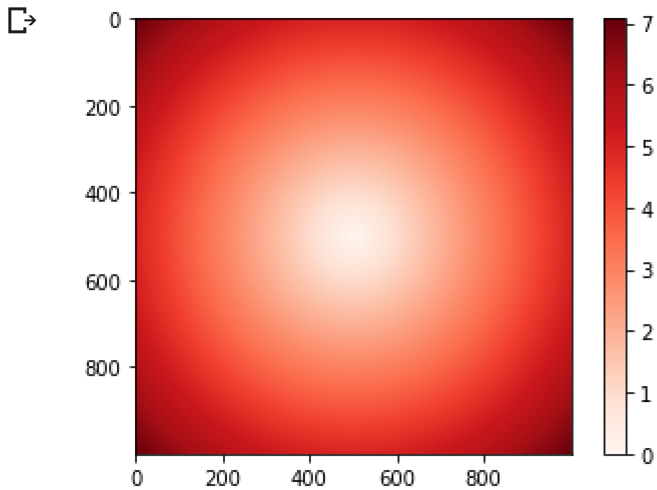
```
# Create an array
points = np.arange(-5, 5, 0.01)

# Make a meshgrid
xs, ys = np.meshgrid(points, points)
z = np.sqrt(xs ** 2 + ys ** 2)

# Display the image on the axes
plt.imshow(z, cmap=plt.cm.Reds)
```

```
# Draw a color bar
plt.colorbar()

# Show the plot
plt.show()
```



13. Save the numpy ndarray object into a npy file

Finally, one of the most important parts of the entire analysis process, the storage of the results. We c
function:

```
import numpy as np
x = np.arange(0.0,5.0,1.0)
np.savetxt('test.txt', x, delimiter=',')
```

Conclusions

We have learned the fundamentals of the `numpy` library for scientific computing, which allows us to cr
them using arithmetic, logical, and relational operators. We have also learned how to restructure arrar
basic tools for visualization.

In the next case study, we will see what is related to the `pandas` library, so we are ready to do data ana