NumPy_Essentials_and_Beyond

Author: Nikos Papakostas
Created: September 2024
GitHub: https://github.com/papaknik
LinkedIn: https://www.linkedin.com/in/nikos-papakostas/

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

- Installing NumPy library if not available in your Jupyter notebook environment

```
!pip install numpy
```

```
Requirement already satisfied: numpy in c:\users\papak\anaconda3\lib\site-packages (1.26.4)
```

- Importing the NumPy library into the current Python session and assigning it the alias 'np'

```
import numpy as np
```

In general, NumPy arrays (ndarrays) are designed to hold elements of a single, fixed data type. When you attempt to create an array with mixed data types, NumPy will automatically promote the data types to a common type according to specific rules.
By default, all elements in a NumPy array are of the same type, which optimizes performance and memory usage.
If you try to create an array with mixed types (such as integers and strings), NumPy will convert the elements to a common type (e.g., converting integers to strings) to maintain a uniform data type across the array.
While you can set the argument dtype=object to build an array for storing heterogeneous data types, this goes beyond the scope of this tutorial, as our focus will be on numerical arrays. After all, NumPy stands for Numerical Python!
Additionally, if you need to work with heterogeneous datasets, you can use the pandas library, which is built upon NumPy and offers powerful tools for data manipulation.

Building blocks of nd arrays

- Building a 1-dimensional array

```
array_1 = np.array([1,3,5,7])
print(array_1)
```

```
[1 3 5 7]
```

- Building a 2-dimensional array

```
array_2 = np.array([[1,2,3,4],[5,6,7,8]])
print(array_2)
```

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

- Extracting the number of dimensions of the array with `.ndim` attribute

```
array_2.ndim
```

```
2
```

- Extracting the shape of the array (rows, columns) with `.shape` attribute

```
array_2.shape
```

```
(2, 4)
```

The result is (2,4) meaning that the array consists of 2 rows and 4 columns

- Getting the type of the array with `.dtype` attribute

```
array_1.dtype
```

```
dtype('int32')
```

The result shows that the data type of the elements of array_1 is 'int32'.

You can explicitly define the data type of the elements of the array when you built it like below:

```
array_1 = np.array([1,3,5,7], dtype='int8')
```

```
array_1.dtype
```

```
dtype('int8')
```

Or you can also change the data type after the array creation using `.astype` function

```
array_1 = array_1.astype('int16')
array_1.dtype
```

```
dtype('int16')
```

- Extracting the total number of elements of an array with `.size` attribute

```
array_1.size
array_2.size
```

```
4
8
```

- 1D array(aka vector)

```
array_1 = np.array([1,3,5,7])
print(array_1)
array_1.ndim
array_1.shape
len(array_1)
array_1.size
```

```
[1 3 5 7]
```

```
1
(4,)
4
4
```

This 1D array has a shape (4,), which means that the size of its unique dimension/axis is 4.
Only in the case of 1D arrays the length and the size coincide and illustrate the number of elements inside the array.

- 2D array(aka matrix)

```
array_2 = np.array([[1,2,3,4],[5,6,7,8]])
print(array_2)
array_2.ndim
array_2.shape
len(array_2)
array_2.size
```

```
[[1 2 3 4]
 [5 6 7 8]]
2
(2, 4)
2
8
```

In this example, the resulting shape of the array is (2, 4), meaning the array consists of 2 rows and 4 columns, representing a matrix data structure.
The `len()` function returns the size of the first dimension/axis (the rows), which is 2, while the `.size` attribute results in the total number of elements inside the array which is 8.

- 3D array(aka 3D tensor)

```
array_3 = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])
print(array_3)
array_3.ndim
array_3.shape
array_3.size
```

```
len(array_3)
```

```
[[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]]
```

```
3
```

```
(2, 2, 3)
```

```
12
```

```
2
```

The shape of array_3 results in (2, 2, 3), meaning there are 2 layers (the first dimension), each having 2 rows (the second dimension), and each row having 3 columns (the third dimension). You can visualize these 2 layers as 2 matrices in 3D space, one after the other. Each layer, in essence, is a separate 2*3 matrix.
So, in other words a 3D array/tensor visualizes matrices in 3D space, just like a matrix visualizes vectors (row or column vectors) in 2D space.
The number of dimensions is 3, as indicated by the ndim attribute. The `len()` function returns 2, representing the size of the first outermost axis (the layers), while the size attribute gives us the total number of elements inside the 3D array/tensor. You can visualize the full structure of this data object (3D array/tensor) as a cube within 3D space.
The cube can be flat, representing just one layer (a matrix in 3D space), or it can consist of multiple layers of matrices stacked together to form a cube-like shape.

- 4D array(aka 4D tensor)

```
array_4 = np.array([[[[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, 36]]]])
print(array_4)
array_4.ndim
array_4.shape
array_4.size
len(array_4)
```

```
[[[[ 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 36]]]]
4
(3, 3, 2, 2)
36
3
```

Here comes the difficult part where we are called to explain and describe, or even 'worse' to 'visualize' a fourth dimension while living in a 3D world. The code informs us of 4 dimensions inside the array, having 36 elements in total and the length of the tensor (*the outermost dimension of the tensor*) has a size of 3, indicating there are 3 separate 3D objects.
The resulting shape is (3, 3, 2, 2). This can be thought of, as 3 different, independent 3d rectangular objects (the first dimension), where each object consists of 3 matrices (the second dimension), each matrix has 2 rows (the third dimension), and each row contains 2 columns (the fourth dimension). These objects are described as

independent because they occupy different "coordinates" along the fourth dimension and, therefore, do not intersect. In an abstract sense, you can imagine this as a set of 3D objects existing in parallel, in a four-dimensional space, with no overlap or intersection between them.

Manipulating np.arrays (Fancy Indexing, Boolean Masking and more)

```
array = np.array([[[1,2,3,4,5],[6,7,8,9,10]],[[11,12,13,14,15],[16,17,18,19,20]]])
```

```
print(array)
array.ndim
array.shape
array.size
```

```
[[[ 1  2  3  4  5]
  [ 6  7  8  9 10]]

 [[11 12 13 14 15]
  [16 17 18 19 20]]]
```

```
3
```

```
(2, 2, 5)
```

```
20
```

Before accessing and modifying elements in an np.array, note that NumPy uses 0-based indexing, and the indices follow the order of the axes as shown in the `.shape` attribute, starting from the outermost dimension (first axis) and moving inward to the last innermost axis to the right

Accessing the element of the array on the first matrix (index 0), on the second row (index 1), on the second column (index 1)

```
array[0,1,1]
```

```
7
```

Accessing all array elements of the second matrix (index 1) in the first row (index 0). By Using ':' we select all columns

```
array[1,0,:]
```

```
array([11, 12, 13, 14, 15])
```

Accessing all array elements of the first matrix (index 0), using ':' for all the rows, of the fourth column (index 3)

```
array[0,:,3]
```

```
array([4, 9])
```

Accessing all array elements in both matrices (using ':'), all the rows (using again ':'), that are on the first 3 columns (using '0:3')
Using 0:3 we mean the range from 0 up to 3 with the starting point/index being inclusive and the ending point/index being exclusive.
Therefore 0:3 results in indices 0,1,2 which correspond to columns 1,2 and 3.

```
array[:,:,0:3]
```

```
array([[[ 1,  2,  3],
        [ 6,  7,  8]],

       [[11, 12, 13],
        [16, 17, 18]]])
```

Accessing all array elements in both matrices (using':'), on the first row (index 0), specifically on the third and fifth column (using the tuple (2,4) to refer to the corresponding indices-Index 2 for column 3 and index 4 for column 5)

```
# This is also called 'fancy indexing' because of the use of a tuple to restrict our data slicing on specific elements
array[:,0,(2,4)]
```

```
array([[ 3,  5],
       [13, 15]])
```

Accessing the last element of each row on each matrix
The index -1 refers to the last element in the specified axis, allowing us to easily retrieve the final column without needing to know the total number of columns.
This is particularly useful for dynamic arrays where the size may vary, or for arrays where the number of rows or columns may be big and counting from the end (negative indexing) might be more convenient

```
array[:,:,-1]
```

```
array([[ 5, 10],
       [15, 20]])
```

Modifying the element on the 2 second matrix, the second row, the third element by changing it to 0

```
array[1,1,2] = 0
print(array)
```

```
[[[ 1  2  3  4  5]
  [ 6  7  8  9 10]]

 [[11 12 13 14 15]
  [16 17  0 19 20]]]
```

Modifying the elements on the 2 second matrix, the first row, on the second and fifth column by changing them to -1

```
array[1,0,(1,4)] = -1
print(array)
```

```
[[[ 1  2  3  4  5]
  [ 6  7  8  9 10]]

 [[11 -1 13 14 -1]
  [16 17  0 19 20]]]
```

Modify the first, third and fifth element of the second row on the first matrix to 10.
Using the slice 0:5:2 on the column axis, will help us start at index 0 (the first element), go up to index 5(not included), using a step size of 2, which selects every second element. This slice on the columns will result on indices 0,2 and 4 that correspond to the first, third and fifth column.

```
array[0,1,0:5:2] = 10
print(array)
```

```
[[[ 1  2  3  4  5]
  [10  7 10  9 10]]

 [[11 -1 13 14 -1]
  [16 17  0 19 20]]]
```

Modify all the elements of the first row on the second matrix to -50

```
array[1,0,:] = -50
print(array)
```

```
[[[  1   2   3   4   5]
  [ 10   7  10   9  10]]

 [[-50 -50 -50 -50 -50]
  [ 16  17   0  19  20]]]
```

Replace the value of the third column, on the first row on the first matrix with -1000, and the element on the fifth column on the second row on the second matrix with 1000

```
b = array.copy()
b
b[(0,1),(0,1),(2,4)] = (-1000, 1000)
b
```

```
array([[[  1,   2,   3,   4,   5],
        [ 10,   7,  10,   9,  10]],

       [[-50, -50, -50, -50, -50],
        [ 16,  17,   0,  19,  20]]])
array([[[    1,     2, -1000,     4,     5],
        [   10,     7,    10,     9,    10]],

       [[  -50,   -50,   -50,   -50,   -50],
        [   16,    17,     0,    19,  1000]]])
```

- replacing values meeting specific criteria with the use of 'np.where()'

```
T = np.array([[11,7,9],[5,15,9],[14,5,7]])
T
T[np.where(T > 10)]
T[np.where(T > 10)]=0
T
```

```
array([[11,  7,  9],
       [ 5, 15,  9],
       [14,  5,  7]])
array([11, 15, 14])
array([[0, 7, 9],
       [5, 0, 9],
       [0, 5, 7]])
```

```
T = np.array([[11,7,9],[5,15,9],[14,5,7]])
T
np.where(T<10, 0, T)
```

```
array([[11,  7,  9],
       [ 5, 15,  9],
       [14,  5,  7]])
```

```
array([[11,  0,  0],
       [ 0, 15,  0],
       [14,  0,  0]])
```

- Boolean Indexing

```
array
array[array > 10]
bool1 = array > 5
bool2 = array < 10
array[bool1 & bool2]
```

```
array([[[  1,   2,   3,   4,   5],
        [ 10,   7,  10,   9,  10]],

       [[-50, -50, -50, -50, -50],
        [ 16,  17,   0,  19,  20]]])
```

```
array([16, 17, 19, 20])
```

```
array([7, 9])
```

- Unique values of an array

Returns the unique elements of an array along with the count of their occurrences if you choose so

```
array
np.unique(array, return_counts=True)
```

```
array([[[  1,   2,   3,   4,   5],
        [ 10,   7,  10,   9,  10]],

       [[-50, -50, -50, -50, -50],
        [ 16,  17,   0,  19,  20]]])
```

```
(array([-50,   0,   1,   2,   3,   4,   5,   7,   9,  10,  16,  17,  19,
         20]),
 array([5, 1, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1, 1, 1], dtype=int64))
```

- Flattening an array

```
array
array.shape
array.flatten()
array.flatten().shape
```

```
array([[[  1,   2,   3,   4,   5],
        [ 10,   7,  10,   9,  10]],

       [[-50, -50, -50, -50, -50],
        [ 16,  17,   0,  19,  20]]])
```

```
(2, 2, 5)
```

```
array([  1,   2,   3,   4,   5,  10,   7,  10,   9,  10, -50, -50, -50,
       -50, -50,  16,  17,   0,  19,  20])
```

```
(20,)
```

- Reshaping an array

```
array
array.shape
np.reshape(array, (4,5))
np.reshape(array, (4,1,5))
```

```
array([[[  1,   2,   3,   4,   5],
        [ 10,   7,  10,   9,  10]],

       [[-50, -50, -50, -50, -50],
        [ 16,  17,   0,  19,  20]]])
```

```
(2, 2, 5)
```

```
array([[  1,   2,   3,   4,   5],
       [ 10,   7,  10,   9,  10],
       [-50, -50, -50, -50, -50],
       [ 16,  17,   0,  19,  20]])
```

```
array([[[  1,   2,   3,   4,   5]],

       [[ 10,   7,  10,   9,  10]],

       [[-50, -50, -50, -50, -50]],

       [[ 16,  17,   0,  19,  20]]])
```

- Flipping an array

```
ar = np.array([[1,2,3],[4,5,6],[7,8,9]])
ar
# Flipping all axis of the array
flipped_ar = np.flip(ar)
flipped_ar

# Flipping the row axis of the array
flipped_0_ar = np.flip(ar, axis=0)
flipped_0_ar

# Flipping the column axis of the array
flipped_1_ar = np.flip(ar, axis=1)
flipped_1_ar
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
array([[9, 8, 7],
       [6, 5, 4],
       [3, 2, 1]])
```

```
array([[7, 8, 9],
       [4, 5, 6],
       [1, 2, 3]])
```

```
array([[3, 2, 1],
       [6, 5, 4],
       [9, 8, 7]])
```

- Transposing an array

```
ar = np.array([[1,2,3],[4,5,6],[7,8,9]])
ar
t = np.transpose(ar)
t
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

- Vertically stacking arrays

```
a1 = np.array([1,2,3])
a2 = np.array([4,5,6])
a3 = np.array([7,8,9])
a4 = np.array([10,11,12])
v = np.vstack([a1,a2,a3,a4])
v
v.shape
```

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

```
(4, 3)
```

- Horizontally stacking arrays

```
b1 = np.full((3,3), 4)
b1
b2 = np.full((3,2), -1)
b2
h = np.hstack([b1,b2])
h
h.shape
```

```
array([[4, 4, 4],
       [4, 4, 4],
       [4, 4, 4]])
```

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

```
array([[ 4,  4,  4, -1, -1],
       [ 4,  4,  4, -1, -1],
       [ 4,  4,  4, -1, -1]])
```

```
(3, 5)
```

- Concatenating arrays

For concatenation to work, all dimensions of the input arrays, except for the concatenation axis, must match exactly. In this example, r1 and r2 cannot be concatenated along axis=1 initially because the sizes along axis=0 differ: r1 has a size of 4 along axis=0, while r2 has a size of 2 along axis=0.
This is also clearly stated in the ValueError we get.

```
r1 = np.array([[1,2],[2,4],[3,5],[4,7]])
r2 = np.array([[5,15],[6,17]])
r1
```

```
r2
np.concatenate((r1,r2), axis=0)
np.concatenate((r1,r2), axis=1)
```

```
array([[1, 2],
       [2, 4],
       [3, 5],
       [4, 7]])
array([[ 5, 15],
       [ 6, 17]])
array([[ 1,  2],
       [ 2,  4],
       [ 3,  5],
       [ 4,  7],
       [ 5, 15],
       [ 6, 17]])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[40], line 6
      4 r2
      5 np.concatenate((r1,r2), axis=0)
----> 6 np.concatenate((r1,r2), axis=1)

ValueError: all the input array dimensions except for the concatenation axis must match exactly, but along dimension 0, the array at index 0 has s
ize 4 and the array at index 1 has size 2
```

To fix the above ValueError, we reshape r1 so that the size along axis=0 matches the size of r2 (2 in this case).

This will make both arrays have the same size along axis=0, allowing concatenation along axis=1.

```
r1 = np.reshape(r1, (2,4))
np.concatenate((r1,r2),axis=1)
```

```
array([[ 1,  2,  2,  4,  5, 15],
       [ 3,  5,  4,  7,  6, 17]])
```

- sorting an array

```
array_5 = np.array([[[11,2,35],[4,51,6]],[[74,8,9],[100,15,12]]])
```

```
array_5
# Sorting the elements in ascending order along axis=0(representing the outermost layer of the tensor), the matrices
np.sort(array_5, axis=0)

# Sorting the elements in ascending order with respect to axis=1 which represents the rows of each matrix in the tensor
np.sort(array_5, axis=1)

# Sorting the elements in ascending order with respect to axis=2 which represents the columns of each matrix in the tensor
np.sort(array_5, axis=2)
```

```
array([[[ 11,   2,  35],
        [  4,  51,   6]],

       [[ 74,   8,   9],
        [100,  15,  12]]])
array([[[ 11,   2,   9],
        [  4,  15,   6]],

       [[ 74,   8,  35],
        [100,  51,  12]]])
array([[[  4,   2,   6],
        [ 11,  51,  35]],

       [[ 74,   8,   9],
        [100,  15,  12]]])
array([[[  2,  11,  35],
        [  4,   6,  51]],

       [[  8,   9,  74],
        [ 12,  15, 100]]])
```

- Splitting an array into multiple arrays

```
array_5
np.split(array_5, 2, axis=0)
np.split(array_5, 3, axis=2)
```

```
array([[[ 11,   2,  35],
        [  4,  51,   6]],

       [[ 74,   8,   9],
        [100,  15,  12]]])
```

```
[array([[[11,  2, 35],
         [ 4, 51,  6]]]),
 array([[[ 74,   8,   9],
         [100,  15,  12]]])]
```

```
[array([[[ 11],
         [  4]],

        [[ 74],
         [100]]]),
 array([[[ 2],
         [51]],

        [[ 8],
         [15]]]),
 array([[[35],
         [ 6]],

        [[ 9],
         [12]]])]
```

Building special np.arrays

- Building the Identity matrix

```
np.identity(4)
```

```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

- Building all 1s array

```python
np.ones(shape=3)
```

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

```python
np.ones(shape=(3,3))
```

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

- Building all 0s array

```python
np.zeros(5)
```

```
array([0., 0., 0., 0., 0.])
```

```python
np.zeros((3,3))
```

```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

```python
np.zeros((3,2,3))
np.zeros((3,2,3)).dtype
np.zeros((3,2,3)).astype('int32').dtype
```

```
array([[[0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.]]])
dtype('float64')
dtype('int32')
```

- Populating an array with a specific number

```
np.full((3,2), 5, dtype='int64')
```

```
array([[5, 5],
       [5, 5],
       [5, 5]], dtype=int64)
```

- Building a specific number array using the shape from another created array

```
# We will build an array with the same shape as of array_3 we built above, and populate all its elements with -1
```

```
np.full_like(array_3, -1)
```

```
array([[[-1, -1, -1],
        [-1, -1, -1]],

       [[-1, -1, -1],
        [-1, -1, -1]]])
```

- Building a 1D array on a specific range

```
np.arange(10)
x = np.arange(start=-1, stop=1, step=0.25)
x
x.reshape(4,2)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
array([-1.  , -0.75, -0.5 , -0.25,  0.  ,  0.25,  0.5 ,  0.75])
```

```
array([[-1.  , -0.75],
       [-0.5 , -0.25],
       [ 0.  ,  0.25],
       [ 0.5 ,  0.75]])
```

Randomly generated arrays

- Setting seed for code reproducibility

```
# Setting a seed ensures that the randomly generated numbers are the same each time, which is essential for debugging or replicating results
rng1 = np.random.default_rng(seed=100)
rng1.integers(low=0, high=100, size = 5)

rng2 = np.random.default_rng(seed=100)
rng2.integers(low=0, high=100, size = 5)

rng3 = np.random.default_rng(seed=100)
rng3.integers(low=0, high=100, size = 5)
```

```
array([76, 83, 12, 59,  8], dtype=int64)
array([76, 83, 12, 59,  8], dtype=int64)
array([76, 83, 12, 59,  8], dtype=int64)
```

- Building a random integer array

By default, the lower bound is inclusive and the upper bound is exclusive. If you need both ends of the range inclusive you have to set the argument 'endpoint' to True

```
rng = np.random.default_rng(seed=123)
rng.integers(low=-1, high=1, size = (10,10), endpoint=True)
```

```
array([[-1,  1,  0, -1,  1, -1, -1, -1,  0, -1],
       [ 0,  1,  0,  1,  0, -1,  1,  1,  1,  1],
       [-1,  0, -1, -1, -1,  1,  1, -1,  0,  1],
       [-1,  0,  0,  1,  1, -1,  1,  1, -1,  0],
       [ 1, -1, -1, -1, -1,  0, -1,  0,  0, -1],
       [ 0, -1, -1,  0, -1,  1, -1,  1, -1,  0],
       [ 0,  1,  0,  1,  0, -1, -1,  1, -1,  1],
       [ 0, -1,  0,  1,  0,  1,  1, -1,  0,  0],
       [ 0, -1,  0,  0,  1, -1,  0,  1,  1, -1],
       [ 1, -1,  0, -1,  1, -1,  0,  0,  1,  0]], dtype=int64)
```

- Populating an array with random floats in the range [0, 1) following the “continuous uniform” distribution

```
rng = np.random.default_rng(seed=123)
rng.random(size=(5,))
```

```
array([0.68235186, 0.05382102, 0.22035987, 0.18437181, 0.1759059 ])
```

- Populating an array with random elements drawn from the standard normal distribution (z-distribution, mean=0 and sd=1)

```
rng = np.random.default_rng(seed=123)
rng.standard_normal(size=100)
```

```
array([-0.98912135, -0.36778665,  1.28792526,  0.19397442,  0.9202309 ,
        0.57710379, -0.63646365,  0.54195222, -0.31659545, -0.32238912,
        0.09716732, -1.52593041,  1.1921661 , -0.67108968,  1.00026942,
        0.13632112,  1.53203308, -0.65996941, -0.31179486,  0.33776913,
       -2.2074711 ,  0.82792144,  1.54163039,  1.12680679,  0.75476964,
       -0.14597789,  1.28190223,  1.07403062,  0.39262084,  0.00511431,
       -0.36176687, -1.2302322 ,  1.22622929, -2.17204389, -0.37014735,
        0.16438007,  0.85988118,  1.76166124,  0.99332378, -0.29152143,
        0.72812756, -1.26160032,  1.42993853, -0.15647532, -0.67375915,
       -0.6390601 , -0.06136133, -0.39278492,  2.28990995, -0.71818115,
        0.03260774,  0.0280499 ,  0.02827212,  0.05534586, -0.48156286,
       -0.5834075 , -0.8621605 , -1.48817461,  0.21630683,  0.98437635,
       -0.54308414, -0.55861504, -0.31648283, -0.46063974, -1.43626975,
        1.36510803,  0.43899989, -0.71169503,  0.29717176, -0.43845727,
       -0.21163743,  0.36396383,  0.95296449,  1.51952413,  1.70390945,
       -0.24885871, -0.49974859,  0.0995975 ,  0.12834321, -0.73422189,
       -0.62047529,  0.81327372,  1.64180101, -0.22650085, -0.64796521,
       -0.28337121, -0.99513136, -0.27287177,  0.42244414, -0.08134296,
        1.2345776 ,  0.15088803,  0.48111953, -0.14875753,  1.31566571,
       -1.2223456 , -0.30359134, -1.17368868,  0.82627351,  0.85032229])
```

- Populate an array with random elements generated from the normal distribution

```
# The 'loc' argument stands for location and represents the mean of the distribution, while the 'scale' argument represents the standard deviation
rng = np.random.default_rng(seed=123)
```

```
rng.normal(loc=100, scale=10, size=(4,4))
```

```
array([[ 90.1087865 ,  96.32213349, 112.87925261, 101.93974419],
       [109.202309  , 105.77103791,  93.63536354, 105.4195222 ],
       [ 96.83404549,  96.77610884, 100.97167319,  84.74069593],
       [111.92166104,  93.28910325, 110.0026942 , 101.36321124]])
```

- Populate an array with values drawn from the Poisson distribution having a specific rate (average value of occurrences)

```
rng = np.random.default_rng(seed=123)
x = rng.poisson(lam=5, size = (5,5))
x
```

```
array([[3, 7, 8, 4, 9],
       [5, 4, 3, 1, 7],
       [4, 8, 5, 6, 6],
       [4, 7, 4, 6, 5],
       [4, 5, 2, 4, 4]], dtype=int64)
```

- Populate an array with values drawn from the Uniform distribution

```
rng = np.random.default_rng(seed=123)
x = rng.uniform(0,1,(20,5))
x
```

```
array([[0.68235186, 0.05382102, 0.22035987, 0.18437181, 0.1759059 ],
       [0.81209451, 0.923345  , 0.2765744 , 0.81975456, 0.88989269],
       [0.51297046, 0.2449646 , 0.8242416 , 0.21376296, 0.74146705],
       [0.6299402 , 0.92740726, 0.23190819, 0.79912513, 0.51816504],
       [0.23155562, 0.16590399, 0.49778897, 0.58272464, 0.18433799],
       [0.01489492, 0.47113323, 0.72824333, 0.91860049, 0.62553401],
       [0.91712257, 0.86469025, 0.21814287, 0.86612743, 0.73075194],
       [0.27786529, 0.79704355, 0.86522171, 0.2994379 , 0.52704208],
       [0.07148681, 0.58323841, 0.2379064 , 0.76496365, 0.17363164],
       [0.31274226, 0.01447448, 0.03255192, 0.49670184, 0.46831253],
       [0.12769032, 0.2575625 , 0.00318111, 0.38106775, 0.57587308],
       [0.42729877, 0.83510235, 0.61649125, 0.26608391, 0.81102211],
       [0.49948675, 0.75881032, 0.56608909, 0.43744036, 0.39615444],
       [0.02223529, 0.46935079, 0.6235584 , 0.94611342, 0.43532608],
       [0.4856414 , 0.51911514, 0.40859098, 0.57879572, 0.07035067],
       [0.48838383, 0.61014483, 0.74387911, 0.42983032, 0.30280213],
       [0.00589003, 0.75647897, 0.07757597, 0.48998804, 0.3043611 ],
       [0.84082216, 0.95047586, 0.31887458, 0.89776829, 0.33752905],
       [0.81211211, 0.7988436 , 0.65528518, 0.22870345, 0.13767446],
       [0.42437114, 0.15153875, 0.87327295, 0.17912676, 0.03029466]])
```

- Random sampling from an array

```
rng = np.random.default_rng(seed=123)
x

# Sampling a random row from the array by choosing axis=0
rng.choice(x, size=1, replace=False, axis=0)

# Sampling a random column from the array by choosing axis=1
rng.choice(x, size=1, replace=False, axis=1)

# For sampling individual elements from the array we first need to flatten the array to 1D and then sample the number of elements we intend to
rng.choice(x.flatten(), size=3, replace=False)
```

```
array([[0.68235186, 0.05382102, 0.22035987, 0.18437181, 0.1759059 ],
       [0.81209451, 0.923345  , 0.2765744 , 0.81975456, 0.88989269],
       [0.51297046, 0.2449646 , 0.8242416 , 0.21376296, 0.74146705],
       [0.6299402 , 0.92740726, 0.23190819, 0.79912513, 0.51816504],
       [0.23155562, 0.16590399, 0.49778897, 0.58272464, 0.18433799],
       [0.01489492, 0.47113323, 0.72824333, 0.91860049, 0.62553401],
       [0.91712257, 0.86469025, 0.21814287, 0.86612743, 0.73075194],
       [0.27786529, 0.79704355, 0.86522171, 0.2994379 , 0.52704208],
       [0.07148681, 0.58323841, 0.2379064 , 0.76496365, 0.17363164],
       [0.31274226, 0.01447448, 0.03255192, 0.49670184, 0.46831253],
       [0.12769032, 0.2575625 , 0.00318111, 0.38106775, 0.57587308],
       [0.42729877, 0.83510235, 0.61649125, 0.26608391, 0.81102211],
       [0.49948675, 0.75881032, 0.56608909, 0.43744036, 0.39615444],
       [0.02223529, 0.46935079, 0.6235584 , 0.94611342, 0.43532608],
       [0.4856414 , 0.51911514, 0.40859098, 0.57879572, 0.07035067],
       [0.48838383, 0.61014483, 0.74387911, 0.42983032, 0.30280213],
       [0.00589003, 0.75647897, 0.07757597, 0.48998804, 0.3043611 ],
       [0.84082216, 0.95047586, 0.31887458, 0.89776829, 0.33752905],
       [0.81211211, 0.7988436 , 0.65528518, 0.22870345, 0.13767446],
       [0.42437114, 0.15153875, 0.87327295, 0.17912676, 0.03029466]])

array([[0.68235186, 0.05382102, 0.22035987, 0.18437181, 0.1759059 ]])
array([[0.18437181],
       [0.81975456],
       [0.21376296],
       [0.79912513],
       [0.58272464],
       [0.91860049],
       [0.86612743],
       [0.2994379 ],
       [0.76496365],
       [0.49670184],
       [0.38106775],
       [0.26608391],
       [0.43744036],
       [0.94611342],
       [0.57879572],
       [0.42983032],
       [0.48998804],
       [0.89776829],
       [0.22870345],
       [0.17912676]])
```

```
array([0.81209451, 0.81211211, 0.26608391])
```

You can find more information about Random Generators in NumPy's official documentation here: https://numpy.org/doc/stable/reference/random/generator.html

Mathematical Operations on arrays

- Addition / Subtraction / Multiplication / Division between arrays

The operations are performed element-wise as long as the arrays have the same shape

```python
a1 = np.array([1,2,3])
a2 = np.array([4,5,6])
a1+a2
a1-a2
a1*a2
a1/a2
```

```
array([5, 7, 9])
array([-3, -3, -3])
array([ 4, 10, 18])
array([0.25, 0.4 , 0.5 ])
```

For the operations you can alternatively use the following code

```
np.add(a1,a2)
np.subtract(a1,a2)
np.multiply(a1,a2)
np.divide(a1,a2)
```

```
array([5, 7, 9])
array([-3, -3, -3])
array([ 4, 10, 18])
array([0.25, 0.4 , 0.5 ])
```

- Addition / Subtraction / Multiplication / Division of arrays with a scalar

As expected, the division of an array with 0 cannot be performed

```
a1
a1+4
a1-7
a1*1.5
a1/4
```

```
array([1, 2, 3])
array([5, 6, 7])
array([-6, -5, -4])
array([1.5, 3. , 4.5])
array([0.25, 0.5 , 0.75])
```

- Raising an array to a power

```
a1
np.power(a1,3)
```

```
array([1, 2, 3])
array([ 1,  8, 27], dtype=int32)
```

- First array elements raised to powers from second array, element-wise

```
a1
a2
np.power(a1,a2)
```

```
array([1, 2, 3])
array([4, 5, 6])
array([  1,  32, 729])
```

- Modulo. Element-wise the remainder of division with a scalar or the remainder of the division between 2 arrays

```
a1
a2
np.mod(a1,2)
np.mod(a1,a2)
```

```
array([1, 2, 3])
array([4, 5, 6])
array([1, 0, 1], dtype=int32)
array([1, 2, 3])
```

- Exponential of all elements in the array

```
a1
np.exp(a1)
```

```
array([1, 2, 3])
array([ 2.71828183,  7.3890561 , 20.08553692])
```

- Natural logarithm of all elements in the array

```
a1
np.log(a1)
```

```
array([1, 2, 3])
array([0.        , 0.69314718, 1.09861229])
```

- Reciprocal element-wise of an array

For the reciprocal function to work properly you have to turn the data type of the array to float

```
a1
np.reciprocal(a1)
np.reciprocal(a1.astype('float32'))
```

```
array([1, 2, 3])
array([1, 0, 0])
array([1.        , 0.5       , 0.33333334], dtype=float32)
```

Mathematical Operations with Broadcasting

Earlier, we mentioned that vectorized arithmetic operations between arrays are possible only if the involved arrays have the same shape.
Broadcasting is the mechanism that makes mathematical operations between arrays of different shapes possible!
Broadcasting works by stretching the shape of the smaller array across the larger one to make the operation possible.
However, this only applies if the arrays are "broadcastable" meaning their shapes are compatible according to a specific rule.
The rule for broadcasting is that the dimensions must either be the same or one of the dimensions must be equal to 1.
This condition must be met for every dimension, starting from the rightmost dimension (right-to-left comparison).

One last but important note, is that the broadcasting operation is performed always row-wise, so in some cases we must reshape the involved arrays accordingly.

```
ar1 = np.array([[1,2,3,4,5],[6,7,8,9,10]])
ar2 = np.array([[0],[1]])
ar1.shape
ar2.shape
ar1
ar2
ar1 + ar2
```

```
(2, 5)
(2, 1)
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
array([[0],
       [1]])
array([[ 1,  2,  3,  4,  5],
       [ 7,  8,  9, 10, 11]])
```

```
ar3 = np.array([[1,2,3],[4,5,6],[7,8,9]])
ar4 = np.array([[1,2,3]])
ar3.shape
ar4.shape
ar3
ar4
ar3 * ar4
```

```
(3, 3)
(1, 3)
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
array([[1, 2, 3]])
array([[ 1,  4,  9],
       [ 4, 10, 18],
       [ 7, 16, 27]])
```

You can find more information about element-wise functions on nd arrays in the official documentation here: https://numpy.org/doc/stable/reference/ufuncs.html

Creating Your Own Universal Functions

There will be cases where the built-in functions NumPy provides, such as `np.exp()`, `np.tanh`, `np.log2` etc, will not cover the need you may have.
This is where custom ufuncs come into play.
You can create your own ufuncs using the `np.frompyfunc`.

```
def func(x,y):
    return x**3 + np.sqrt(x*y) + np.log(x/y)

custom_func = np.frompyfunc(func, 2, 1) # 2 for the number of input arguments and 1 the number of objects returned by func
rng = np.random.default_rng(seed=123)
a = rng.integers(low=10, high=100, size=10**7, endpoint=True)
b = rng.uniform(low=1, high=10, size=10**7)

import time
start = time.time()
custom_func(a,b)
print(f"Computing time result: {time.time() - start}")
```

```
array([1341.502979622934, 373273.99528489, 250061.9038935455, ...,
       2206.4529469680083, 474575.38320384716, 54888.5747162595],
      dtype=object)
Computing time result: 21.8875994682312
```

Although `np.frompyfunc` creates a custom ufunc and it can handle any data type, the main drawback is that it is not as fast as native NumPy ufuncs.
For truly high-performance custom ufuncs, we should use the Numba library, which allows us to create custom ufuncs that match the speed of native NumPy functions.
Numba runs in parallel, across multiple CPU cores, providing a significant speedup.
Using the `@vectorize` decorator, Numba can compile a pure Python function into a ufunc that operates over NumPy arrays as fast as traditional built-in ufuncs written in C. More information on Numba can be found on the official documentation here: https://numba.readthedocs.io/en/stable/user/vectorize.html

```
from numba import vectorize
import time
start = time.time()

# The data types specified in @vectorize allow the function to handle both integers and floats efficiently
# The `target='parallel' option enables the function to run across multiple CPU cores. This will result in significant speed improvements,
# especially on multi-core machines.

@vectorize(['int32(int32, int32)', 'float64(float64, float64)'], target='parallel')
def func(x,y):
    return x**3 + np.sqrt(x*y) + np.log(x/y)
func(a,b)
print(f"Computing Numba time result: {time.time() - start}")
```

```
array([  1341.50297962, 373273.99528489, 250061.90389355, ...,
         2206.45294697, 474575.38320385,  54888.57471626])
Computing Numba time result: 1.725527286529541
```

The performance boost by using Numba when building custom functions is more than obvious. While in the case of `np.frompyfunc` the process completed after 21 seconds, using `@vectorize` took us less than a second! This demonstrates the incredible efficiency gain achieved through parallelization.

Linear Algebra Operations

- Computing the norm/length of a 1D array(vector)

```
a = np.array([1,2,3])
a
np.linalg.norm(a)
```

```
array([1, 2, 3])
```

```
3.7416573867739413
```

- Dot product of two 1D arrays(vectors)

```
a = np.array([1,2,3])
b = np.array([2,4,6])
a
b
np.dot(a,b)
np.vdot(a,b)
a @ b
```

```
array([1, 2, 3])
```

```
array([2, 4, 6])
```

```
28
```

```
28
```

```
28
```

- Outer product of two 1D arrays(vectors)

```
a = np.array([1,2,3])
b = np.array([2,4,6])
outer_prod = np.outer(a,b)
a
b
outer_prod
outer_prod.shape
outer_prod.ndim
```

```
array([1, 2, 3])
```

```
array([2, 4, 6])
```

```
array([[ 2,  4,  6],
       [ 4,  8, 12],
       [ 6, 12, 18]])
```

```
(3, 3)
```

```
2
```

- Tensor product of two 2D arrays

In the example below we begin from 2D and end up to 4D. The resulting product comes from multiplying every element of the first 2D array with every element on the second 2D array. Having 6 elements on each 2D array it is logical to end with 6 2d arrays. As for the final shape of the tensor product, this derives from the concatenation of the shapes of the 2D arrays individually. This is why we moved from 2D to 4D with the final shape (2,3,2,3).

```
a = np.array([[1,2,3],[1,1,1]])
b = np.array([[4,5,6],[2,2,2]])
np.tensordot(a,b, axes=0)
np.tensordot(a,b, axes=0).shape
```

```
array([[[[ 4,  5,  6],
         [ 2,  2,  2]],

        [[ 8, 10, 12],
         [ 4,  4,  4]],

        [[12, 15, 18],
         [ 6,  6,  6]]],


       [[[ 4,  5,  6],
         [ 2,  2,  2]],

        [[ 4,  5,  6],
         [ 2,  2,  2]],

        [[ 4,  5,  6],
         [ 2,  2,  2]]]])
(2, 3, 2, 3)
```

Following the above example, we will compute the tensor product of an array with shape (3,3,2,1) and an array with shape (2,2,2).
According to the above observation the product will occupy the '7D world' with a shape (3,3,2,1,2,2,2) that comes from
the concatenation of the shapes of the original arrays.

```
a = np.random.randint(10, size=(3,3,2,1))
b = np.random.randint(20, size=(2,2,2))
np.tensordot(a,b, axes=0)
np.tensordot(a,b, axes=0).ndim
np.tensordot(a,b, axes=0).shape
```

```
array([[[[[[[  8,  32],
            [ 22,   8]],

           [[ 22,  34],
            [ 30,  36]]]],



         [[[[ 28, 112],
            [ 77,  28]],

           [[ 77, 119],
            [105, 126]]]]],



        [[[[[ 20,  80],
            [ 55,  20]],

           [[ 55,  85],
            [ 75,  90]]]],



         [[[[  8,  32],
            [ 22,   8]],

           [[ 22,  34],
            [ 30,  36]]]]],




        [[[[[ 36, 144],
            [ 99,  36]],

           [[ 99, 153],
            [135, 162]]]],
         [[[[ 28, 112],
            [ 77,  28]],

           [[ 77, 119],
            [105, 126]]]]]],




       [[[[[[  8,  32],
            [ 22,   8]],

           [[ 22,  34],
            [ 30,  36]]]],



         [[[[  4,  16],
            [ 11,   4]],

           [[ 11,  17],
            [ 15,  18]]]]],




        [[[[[ 32, 128],
            [ 88,  32]],

           [[ 88, 136],
            [120, 144]]]],



         [[[[  8,  32],
            [ 22,   8]],

           [[ 22,  34],
            [ 30,  36]]]]],



        [[[[[  8,  32],
            [ 22,   8]],

           [[ 22,  34],
            [ 30,  36]]]],



         [[[[ 24,  96],
            [ 66,  24]],

           [[ 66, 102],
            [ 90, 108]]]]]],





       [[[[[[ 36, 144],
            [ 99,  36]],

           [[ 99, 153],
            [135, 162]]]],



         [[[[ 12,  48],
            [ 33,  12]],

           [[ 33,  51],
            [ 45,  54]]]]],



        [[[[[  4,  16],
            [ 11,   4]],

           [[ 11,  17],
            [ 15,  18]]]],



         [[[[ 20,  80],
            [ 55,  20]],

           [[ 55,  85],
            [ 75,  90]]]]],




        [[[[[ 32, 128],
            [ 88,  32]],

           [[ 88, 136],
            [120, 144]]]],



         [[[[ 36, 144],
            [ 99,  36]],

           [[ 99, 153],
            [135, 162]]]]]]])
```

```
7
```

```
(3, 3, 2, 1, 2, 2, 2)
```

- Determinant and Trace of a square matrix

```
A = np.array([[1,1],[2,3]])
A
trace = np.trace(A)
determinant = np.linalg.det(A)
trace
```

```
determinant
```

```
array([[1, 1],
       [2, 3]])
```

```
4
```

```
1.0
```

- Matrix multiplication

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

```
array([[1, 1],
       [2, 2],
       [3, 3]])
```

```
array([[1, 1, 1],
       [4, 4, 4]])
```

```
array([[ 5,  5,  5],
       [10, 10, 10],
       [15, 15, 15]])
```

```
array([[ 5,  5,  5],
       [10, 10, 10],
       [15, 15, 15]])
```

- Inverse of a square matrix

```
A = np.array([[1,2],[0,1]])
A
inv_A = np.linalg.inv(A)
np.matmul(A, inv_A)
```

```
array([[1, 2],
       [0, 1]])
```

```
array([[1., 0.],
       [0., 1.]])
```

- Pseudo Inverse of a rectangular matrix

```
A = np.array([[1,2],[3,4],[5,6]], dtype='float64')
A
pseudo_invA = np.linalg.pinv(A)
pseudo_invA

# Checking if the conditions for the pseudo inverse matrix are met
np.matmul(np.matmul(A, pseudo_invA),A)
np.matmul(np.matmul(pseudo_invA, A), pseudo_invA)
np.matmul(A, pseudo_invA) - np.transpose(np.matmul(A, pseudo_invA))
np.matmul(pseudo_invA, A) - np.transpose(np.matmul(pseudo_invA, A))
```

```
array([[1., 2.],
       [3., 4.],
       [5., 6.]])
```

```
array([[-1.33333333, -0.33333333,  0.66666667],
       [ 1.08333333,  0.33333333, -0.41666667]])
```

```
array([[1., 2.],
       [3., 4.],
       [5., 6.]])
```

```
array([[-1.33333333, -0.33333333,  0.66666667],
       [ 1.08333333,  0.33333333, -0.41666667]])
```

```
array([[ 0.00000000e+00,  7.21644966e-16,  2.99760217e-15],
       [-7.21644966e-16,  0.00000000e+00,  1.99840144e-15],
       [-2.99760217e-15, -1.99840144e-15,  0.00000000e+00]])
```

```
array([[ 0.00000000e+00, -2.66453526e-15],
       [ 2.66453526e-15,  0.00000000e+00]])
```

- Singular Value Decomposition(SVD) of a matrix

```
A = np.array([[3,2,2],[2,3,-2]])
A
```

```python
U, S, V = np.linalg.svd(A, full_matrices=False)
U
S = np.diag(S)
S
V
np.matmul(U@S, V)
```

```
array([[ 3,  2,  2],
       [ 2,  3, -2]])
```

```
array([[ 0.70710678, -0.70710678],
       [ 0.70710678,  0.70710678]])
```

```
array([[5., 0.],
       [0., 3.]])
```

```
array([[ 7.07106781e-01,  7.07106781e-01,  3.88578059e-16],
       [-2.35702260e-01,  2.35702260e-01, -9.42809042e-01]])
```

```
array([[ 3.,  2.,  2.],
       [ 2.,  3., -2.]])
```

- Solving an overdetermined system of linear equations using two different methods.

1. The least squares method
2. The pseudo-inverse matrix method

```
# Solving A*x=b -the overdetermined case
# Least squares method
A = np.array([[3, -2, 2], [2, -3, 4], [1,-6,7], [-1,3,-3], [7,5,-1]])
b = np.array([1,2,3,-1,4])
A
b
A.shape
b.shape
x, res, rank, s = np.linalg.lstsq(A,b, rcond=None)
x

# Pseudo-inverse method
pseudo_invA = np.linalg.pinv(A)
x = pseudo_invA @ b
x
```

```
array([[ 3, -2,  2],
       [ 2, -3,  4],
       [ 1, -6,  7],
       [-1,  3, -3],
       [ 7,  5, -1]])
```

```
array([ 1,  2,  3, -1,  4])
```

```
(5, 3)
```

```
(5,)
```

```
array([0.13780168, 0.82150188, 1.09880388])
```

```
array([0.13780168, 0.82150188, 1.09880388])
```

For more information on Linear Algebra operations using the NumPy library you can check the official documentation: https://numpy.org/doc/stable/reference/routines.linalg.html

Descriptive Statistics using NumPy

- Minimum

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

```python
# The minimum value across all elements inside the tensor as if the array is flattened
np.min(B)

# The minimum values across the matrices inside the tensor
np.min(B, axis=0)

# The minimum values across the rows in each matrix inside the tensor
np.min(B, axis=1)

# The minimum values across the columns in each matrix inside the tensor
np.min(B, axis=2)
```

```
array([[[ 1,  5, -9],
        [ 2, -4,  6],
        [-3,  7, 11]],

       [[ 4, -5, 12],
        [10, -1,  5],
        [ 8,  3,  6]]])
```

```
-9
```

```
array([[ 1, -5, -9],
       [ 2, -4,  5],
       [-3,  3,  6]])
```

```
array([[-3, -4, -9],
       [ 4, -5,  5]])
```

```
array([[-9, -4, -3],
       [-5, -1,  3]])
```

- Maximum

```python
B
# The maximum value across all elements inside the tensor as if the array is flattened
np.max(B)

# The maximum values across the matrices inside the tensor
np.max(B, axis=0)

# The maximum values across the rows in each matrix inside the tensor
```

```
np.max(B, axis=1)

# The maximum values across the columns in each matrix inside the tensor
np.max(B, axis=2)
```

```
array([[[ 1,  5, -9],
        [ 2, -4,  6],
        [-3,  7, 11]],

       [[ 4, -5, 12],
        [10, -1,  5],
        [ 8,  3,  6]]])
```

```
12
```

```
array([[ 4,  5, 12],
       [10, -1,  6],
       [ 8,  7, 11]])
```

```
array([[ 2,  7, 11],
       [10,  3, 12]])
```

```
array([[ 5,  6, 11],
       [12, 10,  8]])
```

- Sum

```
B
# The sum of the values across all elements inside the tensor as if the array is flattened
np.sum(B)

# The sum of the values across the matrices inside the tensor
np.sum(B, axis=0)

# The sum of the values across the rows in each matrix inside the tensor
np.sum(B, axis=1)

# The sum of the values across the columns in each matrix inside the tensor
np.sum(B, axis=2)
```

```
array([[[ 1,  5, -9],
        [ 2, -4,  6],
        [-3,  7, 11]],

       [[ 4, -5, 12],
        [10, -1,  5],
        [ 8,  3,  6]]])
```

```
58
```

```
array([[ 5,  0,  3],
       [12, -5, 11],
       [ 5, 10, 17]])
```

```
array([[ 0,  8,  8],
       [22, -3, 23]])
```

```
array([[-3,  4, 15],
       [11, 14, 17]])
```

- Mean

```
B
# The mean of the values across all elements inside the tensor as if the array is flattened
np.mean(B)

# The mean of the values across each matrix inside the tensor
np.mean(B, axis=0)

# The mean of the values across the rows in each matrix inside the tensor
np.mean(B, axis=1)

# The mean of the values across the columns in each matrix inside the tensor
np.mean(B, axis=2)
```

```
array([[[ 1,  5, -9],
        [ 2, -4,  6],
        [-3,  7, 11]],

       [[ 4, -5, 12],
        [10, -1,  5],
        [ 8,  3,  6]]])
```

```
3.2222222222222223
```

```
array([[ 2.5,  0. ,  1.5],
       [ 6. , -2.5,  5.5],
       [ 2.5,  5. ,  8.5]])
```

```
array([[ 0.        ,  2.66666667,  2.66666667],
       [ 7.33333333, -1.        ,  7.66666667]])
```

```
array([[-1.        ,  1.33333333,  5.        ],
       [ 3.66666667,  4.66666667,  5.66666667]])
```

- Median

```
B
# The median of the values across all elements inside the tensor as if the array is flattened
np.median(B)

# The median of the values across each matrix inside the tensor
np.median(B, axis=0)

# The median of the values across the rows in each matrix inside the tensor
np.median(B, axis=1)

# The median of the values across the columns in each matrix inside the tensor
np.median(B, axis=2)
```

```
array([[[ 1,  5, -9],
        [ 2, -4,  6],
        [-3,  7, 11]],

       [[ 4, -5, 12],
        [10, -1,  5],
        [ 8,  3,  6]]])
```

```
4.5
```

```
array([[ 2.5,  0. ,  1.5],
       [ 6. , -2.5,  5.5],
       [ 2.5,  5. ,  8.5]])
```

```
array([[ 1.,  5.,  6.],
       [ 8., -1.,  6.]])
```

```
array([[1., 2., 7.],
       [4., 5., 6.]])
```

- Standard deviation

```
B
# The Standard deviation of the values across all elements inside the tensor as if the array is flattened
np.std(B)

# The Standard deviation of the values across each matrix inside the tensor
np.std(B, axis=0)

# The Standard deviation of the values across the rows in each matrix inside the tensor
np.std(B, axis=1)

# The Standard deviation of the values across the columns in each matrix inside the tensor
np.std(B, axis=2)
```

```
array([[[ 1,  5, -9],
        [ 2, -4,  6],
        [-3,  7, 11]],

       [[ 4, -5, 12],
        [10, -1,  5],
        [ 8,  3,  6]]])
```

```
5.652487707545291
```

```
array([[ 1.5,  5. , 10.5],
       [ 4. ,  1.5,  0.5],
       [ 5.5,  2. ,  2.5]])
```

```
array([[2.1602469 , 4.78423336, 8.49836586],
       [2.49443826, 3.26598632, 3.09120617]])
```

```
array([[5.88784058, 4.10960934, 5.88784058],
       [6.94422222, 4.49691252, 2.05480467]])
```

- percentiles

```
B
```

```python
# The 75 percentile of the values across all elements inside the tensor as if the array is flattened
np.percentile(B, q=75)

# The 75 percentile of the values across each matrix inside the tensor
np.percentile(B, q=75, axis=0)

# The 75 percentile of the values across the rows in each matrix inside the tensor
np.percentile(B, q=75, axis=1)

# The 75 percentile of the values across the columns in each matrix inside the tensor
np.percentile(B, q=75, axis=2)
```

```
array([[[ 1,  5, -9],
        [ 2, -4,  6],
        [-3,  7, 11]],

       [[ 4, -5, 12],
        [10, -1,  5],
        [ 8,  3,  6]]])
```

```
6.75
```

```
array([[ 3.25,  2.5 ,  6.75],
       [ 8.  , -1.75,  5.75],
       [ 5.25,  6.  ,  9.75]])
```

```
array([[1.5, 6. , 8.5],
       [9. , 1. , 9. ]])
```

```
array([[3. , 4. , 9. ],
       [8. , 7.5, 7. ]])
```

More information on statistics using the NumPy library can be found in the official documentation here: https://numpy.org/doc/stable/reference/routines.statistics.html

Performance Benefits of using NumPy and its vectorization mechanism

One of the primary reasons for using NumPy in mathematical operations, instead of relying on traditional Python loops, is the significant performance boost provided by vectorization. In NumPy, arithmetic operations are applied directly to each element of an array without the need for explicit loops. This approach is much faster compared to iterating over elements in a Python list with a for loop.

We start by building an array of 10 million random floats in the interval [0,1)

```
rng = np.random.default_rng(seed=123)
a = rng.random(size=10000000)
```

```
a
type(a)
len(a)
```

```
array([0.68235186, 0.05382102, 0.22035987, ..., 0.30556449, 0.16787996,
       0.0722172 ])
```

```
numpy.ndarray
```

```
10000000
```

We populate also a list with the elements of the array

```
list_a = a.tolist()
type(list_a)
len(list_a)
```

```
list
```

```
10000000
```

The sigmoid function serves as a nice example to illustrate the performance difference. We will compute the sigmoid function for each element in both the list and the NumPy array.

```
def sigmoid(x):
    sig = 1.0 / (1 + np.exp(-x))
    return sig
```

- Using a for loop for the list

```
import time
from math import exp
start = time.time()
b = [sigmoid(x) for x in list_a]
```

```
print(f"Loop time result: {time.time() - start}")
```

```
Loop time result: 13.666921854019165
```

- Using the vectorized operation on the array

```
import time
start = time.time()
b = sigmoid(a)
print(f"Vectorized time result: {time.time() - start}")
```

```
Vectorized time result: 0.32981371879577637
```

The difference in execution time between the two methods is immense. Using a for loop with a Python list took approximately 13 seconds, while applying the vectorized operation on a NumPy array took only 0.3 seconds. This highlights the efficiency of NumPy's vectorization mechanism, especially when utilized in Big Data.

Memory Layouts in NumPy: C-contiguous Arrays

Continuing from the comments on the previous section about the performance benefits of using NumPy, another critical factor contributing to the library's efficiency is the **memory layout of NumPy arrays**. This layout refers to how the elements of the array are stored in the computer's memory, and it directly impacts how efficiently operations can access and manipulate the data.
The default layout in which all np.arrays are built is the C-contiguous (Row-major order), where rows of the array are stored sequentially in memory, and is optimized for row-wise operations.
To ensure that an array is in C-contiguous order, we can use the function `np.ascontiguousarray()`. This function converts the array into the correct memory layout if needed, though as already mentioned, NumPy arrays are built by default in this layout. To check the memory layout of an array, we can inspect its flags using `ndarray.flags`.

Populating an array with 10 million random floats in the interval [0,1)

```
rng = np.random.default_rng(seed=123)
a = rng.random(size=10000000)
```

```
a.flags
```

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
```

More information on memory layout of ndarrays can be found in the official documentation here: https://numpy.org/doc/stable/reference/arrays.ndarray.html