Introduction to Python NumPy library

Abdesselam Filali
infos@filali.net
https://github.com/Abdou-fi/python_libraries

February 05, 2026

This is a Jupyter notebook converted to LaTeXusing the nbconvert tool. It contains explanation, code snippets and outputs related to ***NumPy*** array manipulations. For any question or remark please email me or send a request throught my github.

1 Numpy Array Creation Methods

1.1 Create a NumPy array from a python list

[1]:
```
import pandas as pd
import numpy as np
```

To use NumPy, we first need to import it. The common convention is to import it as `np`. We also import the pandas library as `pd` for later use.

To create a NumPy array from a Python list, we can use the `np.array()` function. This function takes a list (or a list of lists) as input and converts it into a NumPy array. in the example bellow, we have a list of numbers and we convert it into a NumPy array:

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

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

We can see that the output is a NumPy array containing the elements of the original list. The output shows that the list `[1, 2, 3]` has been successfully converted into a NumPy array.

[3]:
```
type(np.array(a))
```

[3]:
```
numpy.ndarray
```

The `array()` function can also handle more complex structures, such as lists of lists, which will be converted into multi-dimensional arrays. For example, if we have a list of lists, it will be converted into a 2D array:

[4]:
```
a = [[1,2,3], [4,5,6]]
np.array(a)
```

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

In this case, the list of lists `[[1, 2, 3], [4, 5, 6]]` has been converted into a 2D NumPy array with two rows and three columns. We can also specify the data type of the array elements using the `dtype` parameter. For example, if we want to create an array of type `float32`, we can do it like:

[5]:
```
a = [[1,2,3], [4,5,6]]
np.array(a, dtype=np.float32)
```

[5]:
```
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
```

In this example, the original list of lists has been converted into a 2D NumPy array with the specfied data type of `float32`. `float32` means that each element in the array is stored as a 32-bit floating-point number, which allows for decimal values and can save memory compared to the default `float64` type.

1.2 Create a NumPy array of Zeros

We can also create arrays with specific shapes using the `np.zeros()` and `np.ones()` functions. For example, to create a 1D array of zeros with 5 elements, we can do:

[6]:
```
np.zeros(5)
```

[6]:
```
array([0., 0., 0., 0., 0.])
```

To create a 2x3 array of zeros, we can do it like:

[7]:
```
np.zeros((2,3))
```

[7]:
```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

1.3 Create a NumPy array of Ones

To create a 2x3 array of ones , we can do:

[8]:
```
np.ones((2,3))
```

[8]:
```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

1.4 Create an Identity Numpy Array

To create an identity matrix (also known as a square matrix with ones on the diagonal and zeros elsewhere), we can use the `np.eye()` function:

[9]:
```
np.eye(3)
```

[9]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

1.5 Create an equally spaced Numpy Array with a specific step

We can create an equally spaced array of numbers using the `np.arange()` function. This function takes a start value, a stop value, and an optional step value. For example, to create an array of numbers from 0 to 9, we can do:

[10]:
```
np.arange(10)
```

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

Examples of using `np.arange()` with different start, stop, and step values:

[11]:
```
np.arange(5,11)
```

[11]:
```
array([ 5,  6,  7,  8,  9, 10])
```

[12]:
```
np.arange(5,11,2)
```

[12]:
```
array([5, 7, 9])
```

1.6 Create an equally spaced Numpy Array with a specific array size

We can create an array of evenly spaced values using the `np.linspace()` function. This function takes a start value, a stop value, and the number of values to generate. For example, to create an array of 5 evenly spaced values between 10 and 20, we can do:

[13]:
```
np.linspace(start=10, stop=20, num=5)
```

[13]:
```
array([10. , 12.5, 15. , 17.5, 20. ])
```

1.7 Generate a random numpy array

We can generate a random numpy array using the `np.random` module. For example, to generate an array of 5 random integers between 5 and 15, we can do:

[14]:
```
np.random.randint(low=5, high=16, size=5)
```

[14]:
```
array([13, 10,  6, 12, 12], dtype=int32)
```

To generate an array of 10 random floating-point numbers between 0 and 1, we can do:

[15]:
```
np.random.random(size=10)
```

[15]:
```
array([0.27012108, 0.89988154, 0.24439995, 0.69737164, 0.41265013,
       0.04315854, 0.68070023, 0.14625646, 0.73165926, 0.46900252])
```

1.8 Generate NumPy Array from a Pandas Series

We can also create a NumPy array from a pandas Series using the `np.array()` or `np.asarray()` functions. For example, if we have a pandas Series, we can convert it to a NumPy array like this:

[16]:
```
s= pd.Series([1,2,3,4], name="col")
np.array(s)
```

[16]:
```
array([1, 2, 3, 4])
```

[17]:
```
s= pd.Series([1,2,3,4], name="col")
np.asarray(s)
```

[17]:
```
array([1, 2, 3, 4])
```

2 NumPy Array Manipulation Methods

2.1 Shape of the NumPy Array

We can get the shape of a NumPy array using the `shape` attribute. For example, if we have a 2x3 array, we can get its shape like this:

[18]:
```
a = np.ones((2,3))
print("Shape of the array - Method 1:", np.shape(a))
print("Shape of the array - Method 2:", a.shape)
```

```
Shape of the array - Method 1: (2, 3)
Shape of the array - Method 2: (2, 3)
```

2.2 Reshape the NumPy Array

In order to reshape a NumPy array, we can use the `reshape()` method. This method allows us to change the shape of an array without changing its data. For example, if we have a 1D array of 10 elements, we can reshape it into a 2D array with 2 rows and 5 columns like this:

[19]:
```
a = np.arange(10)
a
```

[19]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[20]:
```
a.reshape((2,5))
```

[20]:
```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

2.3 Transpose the NumPy Array

We can transpose a NumPy array using the `transpose()` method. This method returns a new array with the rows and columns swapped. For example, if we have a 6x2 array, we can transpose it into a 2x6 array like this:

[21]:
```
a = np.arange(12).reshape((6,2))
a
```

[21]:
```
array([[ 0,  1],
       [ 2,  3],
       [ 4,  5],
       [ 6,  7],
       [ 8,  9],
       [10, 11]])
```

[22]:
```
a.transpose()
```

[22]:
```
array([[ 0,  2,  4,  6,  8, 10],
       [ 1,  3,  5,  7,  9, 11]])
```

2.4 Concatenate multiple NumPy arrays to form one NumPy Array

We can concatenate multiple NumPy arrays to form one NumPy array using the `concatenate()` method. This method takes a tuple of arrays and an axis argument. The axis argument specifies the axis along which the arrays should be concatenated.

Bellow some examples of concatenating two 2D arrays row-wise (axis=0) and column-wise (axis=1).

In this example, we have two 2D arrays `a` and `b`. The `concatenate()` function is used to concatenate them row-wise (axis=0), resulting in a new array with three rows and two columns. The first two rows come from array `a`, and the third row comes from array `b`.

[23]:
```
# Concatenate Row-wise

a = np.array([[1,2], [3,4]])
b = np.array([[5,6]])
np.concatenate((a,b), axis=0)
```

[23]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In the second example, we concatenate the same two arrays `a` and `b` column-wise (axis=1). The `b.T` is used to transpose array `b` from a 1x2 array to a 2x1 array, allowing it to be concatenated with array `a` along the columns. The resulting array has two rows and three columns, with the first two columns from array `a` and the third column from array `b`.

[24]:
```
# Concatenate Column-wise

a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b.T), axis=1)
```

[24]:
```
array([[1, 2, 5],
       [3, 4, 6]])
```

In the third example, we concatenate the same two arrays `a` and `b` without specifying an axis (axis=None). This results in a flt array that contains all the elements from both arrays in a single dimension.

[25]:
```
# Concatenate to generate a flat NumPy Array

a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), axis=None)
```

[25]:
```
array([1, 2, 3, 4, 5, 6])
```

- axis=0 is same as `np.vstack()`.
- axis=1 is same as `np.hstack()`.

[26]:
```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), axis=0)
```

[26]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

[27]:
```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.vstack((a, b))
```

[27]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

[28]:
```
a = np.array([1, 2, 3, 4])
b = np.array([5, 6])
c = np.hstack((a, b))
c
```

[28]:
```
array([1, 2, 3, 4, 5, 6])
```

2.5 Flatten NumPy Array

Assuming we have a 2D NumPy array and we want to flatten it into a 1D array, we can use the `flatten()` method. This method returns a copy of the array collapsed into one dimension.

[29]:
```
a = np.array([[1,2], [3,4]])
a.flatten()
```

[29]:
```
array([1, 2, 3, 4])
```

2.6 Unique Elements of a NumPy Array

To find the unique elements of a NumPy array, we can use the `unique()` function. This function returns the sorted unique elements of an array. For example, if we have a 2D array with some duplicate values, we can find the unique elements like this:

[30]:
```
a = np.array([[1,2],[2,3]])
np.unique(a)
```

[30]:
```
array([1, 2, 3])
```

To find the unique rows of a 2D array, we can use the `unique()` function with the `axis` parameter set to 0. For example, if we have a 2D array with some duplicate rows, we can find the unique rows like this:

[31]:
```
# Return Unique Rows

a = np.array([[1, 2, 3], [1, 2, 3], [2, 3, 4]])
np.unique(a, axis=0)
```

[31]:
```
array([[1, 2, 3],
       [2, 3, 4]])
```

In the same way, to find the unique columns of a 2D array, we can use the `unique()` function with the `axis` parameter set to 1. For example, if we have a 2D array with some duplicate columns, we can find the unique columns like this:

[32]:
```
# Return Unique Columns

a = np.array([[1, 1, 3], [1, 1, 3], [1, 1, 4]])
np.unique(a, axis=1)
```

[32]:
```
array([[1, 3],
       [1, 3],
       [1, 4]])
```

2.7 Squeeze a NumPy Array

If you want to remove axes of length one from your numpy array, use the np.squeeze() method. For example, if you have a 3D array with an axis of length one, you can remove it like this:

[33]:
```
x = np.array([[[0], [1], [2]]])
```

[34]:
```
x.shape
```

[34]:
```
(1, 3, 1)
```

[35]:
```
np.squeeze(x)
```

[35]:
```
array([0, 1, 2])
```

[36]:
```
np.squeeze(x).shape
```

[36]:
```
(3,)
```

2.8 Transform NumPy Array to Python List

To transform a NumPy array to a Python list, we can use the `tolist()` method. This method converts the NumPy array into a nested list of Python scalars. For example, if we have a 2D NumPy array and we want to convert it to a list, we can do it like this:

[37]:
```
a = np.array([[1, 1, 3], [1, 1, 3], [1, 1, 4]])
a.tolist()
```

[37]:
```
[[1, 1, 3], [1, 1, 3], [1, 1, 4]]
```

3 Mathematical Operations on NumPy Arrays

We can perform various mathematical operations on NumPy arrays, including trigonometric functions, rounding functions, and more.

3.1 Trigonometric Functions

We can apply trigonometric functions to NumPy arrays using the `sin()`, `cos()`, and `tan()` functions from the NumPy library. These functions operate element-wise on the input array, returning a new array with the same shape containing the trigonometric values of each element. For example, if we have a NumPy array of angles in radians, we can compute the sine, cosine, and tangent of those angles like this:

[38]:
```
a = np.array([1,2,3])
print("Trigonometric Sine   :", np.sin(a))
print("Trigonometric Cosine :", np.cos(a))
print("Trigonometric Tangent:", np.tan(a))
```

```
Trigonometric Sine   : [0.84147098 0.90929743 0.14112001]
Trigonometric Cosine : [ 0.54030231 -0.41614684 -0.9899925 ]
Trigonometric Tangent: [ 1.55740772 -2.18503986 -0.14254654]
```

3.2 Rounding Functions

- Return the element-wise floor using the np.floor() method.
- Return the element-wise ceiling using the np.ceil() method.
- Round to nearest integer using the np.rint() method.

[39]:
```
a = np.linspace(1, 2, 5)
a
```

[39]:
```
array([1.  , 1.25, 1.5 , 1.75, 2.  ])
```

[40]:
```
np.floor(a)
```

[40]:
```
array([1., 1., 1., 1., 2.])
```

[41]:
```
np.ceil(a)
```

[41]:
```
array([1., 2., 2., 2., 2.])
```

[42]:
```
np.rint(a)
```

[42]:
```
array([1., 1., 2., 2., 2.])
```

To round to a given number of decimals, we can use the `round()` method from the NumPy library. This method takes two arguments: the array to be rounded and the number of decimal places to round to. For example, if we have a NumPy array of floating-point numbers and we want to round them to 2 decimal places, we can do it like this:

[43]:
```
# Round to given number of decimals using the np.round_() method:
import numpy as np
a = np.linspace(1,2,7)
b = np.round(a, 2)  # 2 decimal places
b
```

[43]:
```
array([1.  , 1.17, 1.33, 1.5 , 1.67, 1.83, 2.  ])
```

3.3 Exponents and logarithms

- Calculate the element-wise exponential using the np.exp() method.
- Calculate the element-wise natural logarithm using the np.log() method.

[44]:
```
a = np.arange(1,6)
a
```

[44]:
```
array([1, 2, 3, 4, 5])
```

[45]:
```
np.exp(a).round(2)
```

[45]:
```
array([  2.72,   7.39,  20.09,  54.6 , 148.41])
```

[46]:
```
np.log(a).round(2)
```

[46]:
```
array([0.  , 0.69, 1.1 , 1.39, 1.61])
```

3.4 Sum and Product

We can calculate the sum and product of elements in a NumPy array using the `sum()` and `prod()` methods, respectively. The `sum()` method returns the sum of all elements in the array, while the `prod()` method returns the product of all elements in the array. For example, if we have a NumPy array of numbers and we want to calculate the sum and product of those numbers, we can do it like this:

[47]:
```
a = np.array([[1, 2], [3, 4]])
a
```

[47]:
```
array([[1, 2],
       [3, 4]])
```

[48]:
```
np.sum(a, axis=0)
```

[48]:
```
array([4, 6])
```

[49]:
```
np.sum(a, axis=1)
```

[49]:
```
array([3, 7])
```

[50]:
```
np.sum(a, axis=None)
```

[50]:
```
np.int64(10)
```

[51]:
```
a = np.array([[1, 2], [3, 4]])
np.prod(a)
```

[51]:
```
np.int64(24)
```

[52]:
```
np.prod(a, axis=0)
```

[52]:
```
array([3, 8])
```

[53]:
```
np.prod(a, axis=1)
```

[53]:
```
array([ 2, 12])
```

3.5 Square Root

To calculate the square root of each element in a NumPy array, we can use the `sqrt()` method from the NumPy library. This method takes a NumPy array as input and returns a new array containing the square root of each element. For example, if we have a 2D NumPy array and we want to calculate the square root of each element, we can do it like this:

[54]:
```
a = np.array([[1,2], [3,4]])
np.sqrt(a)
```

[54]:
```
array([[1.        , 1.41421356],
       [1.73205081, 2.        ]])
```

4 Matrix and Vector Operations

Operations such as dot product and matrix product are fundamental in linear algebra and are widely used in various applications, including machine learning, data analysis, and scientific computing. NumPy provides built-in functions to perform these operations efficiently and accurately.

4.1 Dot Product

The dot product of two vectors is a scalar value that is calculated by multiplying the corresponding elements of the vectors and then summing the results. In NumPy, we can calculate the dot product of two vectors or matricies using the `dot()` function. The `dot()` function takes two arrays as input and returns the dot product of those arrays. If the input arrays are 1-D, it returns the inner product of the vectors.

[55]:
```
vector1= np.array([1, 2, 3])
vector2= np.array([4, 5, 6])
np.dot(vector1,vector2)
```

[55]:
```
np.int64(32)
```

The dot product is also defined for 2-D arrays (matrices) and higher-dimensional arrays, where it performs a matrix multiplication or a tensor dot product, respectively.

For example, if we have two matricies `a` and `b`, we can calculate their dot product as follows:

[56]:
```
a = np.array([[1, 2], [3, 4]])
b = np.array([[1, 1], [1, 1]])
np.dot(a,b)
```

[56]:
```
array([[3, 3],
       [7, 7]])
```

4.2 Matrix Product

The matrix product, also known as matrix multiplication, is a binary operation that takes a pair of matrices and produces another matrix. In NumPy, we can perform matrix multiplication using the `matmul()` function or the `@` operator. The `matmul()` function takes two arrays as input and returns the matrix product of those arrays. The `@` operator is a shorthand for the `matmul()` function and can be used to perform matrix multiplication in a more concise way. For example, if we have two matrices `a` and `b`, we can calculate their matrix product as follows:

[57]:
```
a = np.array([[1, 2], [3, 4]])
b = np.array([[1, 1], [1, 1]])
```

[58]:
```
np.matmul(a,b)
```

[58]:
```
array([[3, 3],
       [7, 7]])
```

[59]:
```
a@b
```

[59]:
```
array([[3, 3],
       [7, 7]])
```

4.3 Vector Norm

The vector norm is a measure of the length or magnitude of a vector. In NumPy, we can calculate the vector norm using the `linalg.norm()` function. The `linalg.norm()` function takes a vector as input and returns the norm of that vector. By default, it calculates the L2 norm, which is the square root of the sum of the squares of the elements in the vector. However, it can also calculate other types of norms, such as the L1 norm (the sum of the absolute values of the elements) and the infinity norm (the maximum absolute value of the elements), by specifying the `ord` parameter.

[60]:
```
a = np.arange(-4, 5)
a
```

[60]:
```
array([-4, -3, -2, -1,  0,  1,  2,  3,  4])
```

[61]:
```
np.linalg.norm(a)
```

[61]:
```
np.float64(7.745966692414834)
```

[62]:
```
np.linalg.norm(a,1)
```

[62]:
```
np.float64(20.0)
```

5 Sorting Methods

Arrays can be sorted in various ways, such as sorting based on rows, columns, or the flattened array. NumPy provides the `sort()` method to sort arrays in diffeerent ways.

5.1 Sort a NumPy Array

By default, the `sort()` method sorts the array along the last axis (i.e., the columns for a 2D array). However, we can specify the axis along which to sort the array using the `axis` parameter. If we set `axis=None`, the array will be flattened before sorting. For example, if we have a 2D NumPy array and we want to sort it based on rows, columns, or the flattened array, we can do it like this:

[63]:
```
a = np.array([[1,4],[3,1]])
a
```

[63]:
```
array([[1, 4],
       [3, 1]])
```

[64]:
```
np.sort(a) # sort based on rows
```

[64]:
```
array([[1, 4],
       [1, 3]])
```

[65]:
```
np.sort(a, axis=None) # sort the flattened array
```

[65]:
```
array([1, 1, 3, 4])
```

[66]:
```
np.sort(a, axis=0) # sort based on columns
```

[66]:
```
array([[1, 1],
       [3, 4]])
```

You can sort a NumPy array using one of two primary methods:

`numpy.sort(a)`: Returns a sorted copy of the array, leaving the original array unchanged.

`a.sort()`: Sorts the array in-place, modifying the original array directly.

[67]:
```
arr = np.array([3, 2, 0, 1])
sorted_arr = np.sort(arr)
print("arr: ", arr)                  # arr is still [3, 2, 0, 1]
print("sorted_arr: ", sorted_arr)    # sorted_arr is [0, 1, 2, 3]
```

```
arr:  [3 2 0 1]
sorted_arr:  [0 1 2 3]
```

[68]:
```
arr = np.array([3, 2, 0, 1])
arr.sort()      # arr is now [0, 1, 2, 3]
arr
```

[68]:
```
array([0, 1, 2, 3])
```

5.2 Order of Indices in a Sorted NumPy Array

We can use the `argsort()` method to get the indices of the elements in a NumPy array that would sort the array. This method returns an array of indices that can be used to sort the original array. For example, if we have a 2D NumPy array and we want to get the indices of the elements that would sort the array based on rows, columns, or the flattened array, we can do it like this:

[69]:
```
x = np.array([3,1,2])
np.argsort(x)
```

[69]:
```
array([1, 2, 0])
```

6 Searching Methods

NumPy provides several methods for searching arrays, including `argmax()`, `argmin()`, and `where()`.

6.1 Indices corresponding to Maximum Values

In a flattened array, the `argmax()` function returns the index of the maximum value in the array.

[70]:
```
a = np.random.randint(1, 20, 10).reshape(2,5)
a
```

[70]:
```
array([[12,  8,  8,  1, 17],
       [ 2, 14,  7,  2,  3]], dtype=int32)
```

[71]:
```
np.argmax(a) # index in a flattened array
```

[71]:
```
np.int64(4)
```

[72]:
```
a.flatten()
```

[72]:
```
array([12,  8,  8,  1, 17,  2, 14,  7,  2,  3], dtype=int32)
```

[73]:
```
np.argmax(a, axis=0) # indices along columns
```

[73]:
```
array([0, 1, 0, 1, 0])
```

[74]:
```
np.argmax(a, axis=1) # indices along rows
```

[74]:
```
array([4, 1])
```

To find the index in a non-flattened array, you can do the following:

[75]:
```
ind = np.unravel_index(np.argmax(a), a.shape)
ind
```

[75]:
```
(np.int64(0), np.int64(4))
```

6.2 Indices corresponding to Minimum Values

In the same way, in a flattened array, the `argmin()` function returns the index of the minimum value in the array.

[76]:
```
a = np.random.randint(1, 20, 10).reshape(2,5)
a
```

[76]:
```
array([[ 3, 14, 16, 10,  6],
       [ 9,  7,  8,  7, 18]], dtype=int32)
```

[77]:
```
np.argmin(a) # index in a flattend array
```

[77]:
```
np.int64(0)
```

[78]:
```
a.flatten()
```

[78]:
```
array([ 3, 14, 16, 10,  6,  9,  7,  8,  7, 18], dtype=int32)
```

[79]:
```
np.argmin(a, axis=0) # indices along columns
```

[79]:
```
array([0, 1, 1, 1, 0])
```

[80]:
```
np.argmin(a, axis=1) # indices along rows
```

[80]:
```
array([0, 1])
```

6.3 Search based on condition

The `where()` function returns the indices of elements in an array that satisfy a given condition.

[81]:
```
a = np.random.randint(-10, 10, 10)
a
```

[81]:
```
array([ 6, -5, -8,  5,  4,  5,  8, -1,  0,  5], dtype=int32)
```

The condition can be any boolean expression that evaluates to True or False for each element in the array. For example, we can find the indices of elements in the array that are greater than 0. np.where(a > 0). In the following example, we find the indices of elements in the array that are less than or equal to 0, and we replace them with 0.

[82]:
```
np.where(a <0, 0, a)
```

[82]:
```
array([6, 0, 0, 5, 4, 5, 8, 0, 0, 5], dtype=int32)
```

6.4 Indices of Non-Zero Elements

The `nonzero()` function returns the indices of elements in an array that are non-zero. The result is a tuple of arrays, one for each dimension of the input array, containing the indices of the non-zero elements along that dimension.

[83]:
```
a = np.array([[3, 0, 0], [0, 4, 0], [5, 6, 0]])
np.nonzero(a)
```

[83]:
```
(array([0, 1, 2, 2]), array([0, 1, 0, 1]))
```

7 Statistical Methods

NumPy provides several statistical methods for arrays, including `mean()`, `median()`, and `std()`.

7.1 Mean

The `mean()` function returns the mean of the elements in an array. By default, it calculates the mean of all elements in the array. However, we can specify the axis along which to calculate the mean using the `axis` parameter. For example, if we have a 2D NumPy array and we want to calculate the mean of the elements along rows or columns, we can do it like this:

[84]:
```
a = np.array([[1, 2], [3, 4]])
a
```

[84]:
```
array([[1, 2],
       [3, 4]])
```

[85]:
```
np.mean(a)    # mean of all elements in a
```

[85]:
```
np.float64(2.5)
```

[86]:
```
np.mean(a, axis=1) # along the row axis
```

[86]:
```
array([1.5, 3.5])
```

[87]:
```
np.mean(a, axis=0) # along the column axis
```

[87]:
```
array([2., 3.])
```

7.2 Median

The `median()` function returns the median of the elements in an array. By default, it calculates the median of all elements in the array. However, we can specify the axis along which to calculate the median using the `axis` parameter. For example, if we have a 2D NumPy array and we want to calculate the median of the elements along rows or columns, we can do it like this:

[88]:
```
a = np.array([[1, 2], [3, 4]])
a
```

[88]:
```
array([[1, 2],
       [3, 4]])
```

[89]:
```
np.median(a)   # median of all elements in a
```

[89]:
```
np.float64(2.5)
```

[90]:
```
np.median(a, axis=1) # along the row axis
```

[90]:
```
array([1.5, 3.5])
```

[91]:
```
np.median(a, axis=0) # along the column axis
```

[91]:
```
array([2., 3.])
```

7.3 Standard Deviation

The `std()` function returns the standard deviation of the elements in an array. By default, it calculates the standard deviation of all elements in the array. However, we can specify the axis along which to calculate the standard deviation using the `axis` parameter. For example, if we have

a 2D NumPy array and we want to calculate the standard deviation of the elements along rows or columns, we can do it like this:

[92]:
```
a = np.array([[1, 2], [3, 4]])
a
```

[92]:
```
array([[1, 2],
       [3, 4]])
```

[93]:
```
np.std(a)   # standard deviation of all elements in a
```

[93]:
```
np.float64(1.118033988749895)
```

[94]:
```
np.std(a, axis = 1) # along the row axis
```

[94]:
```
array([0.5, 0.5])
```

[95]:
```
np.std(a, axis = 0) # along the column axis
```

[95]:
```
array([1., 1.])
```