

NumPy Handbook

“ Numerical Python, working with numerical data in Python, going from Python lists to Numpy arrays, CSV data files using Numpy, and etc.. “

By
Ahmad Jawabreh

Dec, 2023

1.0 Introduction.....	3
2.0 Going from Python lists to Numpy array.....	5
3.0 Operating on Numpy arrays.....	6
4.0 Multi-dimensional Numpy arrays.....	7
5.0 Working with CSV data files.....	8
6.0 Arithmetic operations, broadcasting and comparison.....	11
6.1 Array Broadcasting.....	12
6.2 Array Comparison.....	13
6.3 Array indexing and slicing.....	14
7.0 More NumPy Functions.....	15

1.0 Introduction

The "data" in Data Analysis typically refers to numerical data, e.g., stock prices, sales figures, sensor measurements, sports scores, database tables, etc. The Numpy library provides specialized data structures, functions, and other tools for numerical computing in Python. Let's work through an example to see why & how to use Numpy for working with numerical data.

Suppose we want to use climate data like the temperature, rainfall, and humidity to determine if a region is well suited for growing apples. A simple approach for doing this would be to formulate the relationship between the annual yield of apples (tons per hectare) and the climatic conditions like the average temperature (in degrees Fahrenheit), rainfall (in millimeters) & average relative humidity (in percentage) as a linear equation.

$$\text{yield_of_apples} = w1 * \text{temperature} + w2 * \text{rainfall} + w3 * \text{humidity}$$

We're expressing the yield of apples as a weighted sum of the temperature, rainfall, and humidity. This equation is an approximation since the actual relationship may not necessarily be linear, and there may be other factors involved. But a simple linear model like this often works well in practice.

Based on some statistical analysis of historical data, we might come up with reasonable values for the weights $w1$, $w2$, and $w3$. Here's an example set of values:

$$w1, w2, w3 = 0.3, 0.2, 0.5.$$

Given some climate data for a region, we can now predict the yield of apples. Here's some sample data:

Region	Temp (F)	Rainfall (mm)	Humaidity (%)
Jerusalem	73	67	43
Bethlehem	91	88	64
Hebron	87	134	85
Jericho	102	43	37
Nablus	69	96	70

figure(1): climate data for a region sample

To begin, we can define some variables to record climate data for a region.

Example:

```
jerusalem_temp = 73  
jerusalem_rainfall = 67  
jerusalem_humidity = 43
```

```
w1, w2, w3 = 0.3, 0.2, 0.5
```

```
jerusalem_yield_apples = jerusalem_temp * w1 + jerusalem_rainfall * w2 + jerusalem_humidity  
* w3
```

```
jerusalem_yield_apples
```

```
print(f"The expected yield of apples in Jerusalem region is {jerusalem_yield_apples} tons per  
hectare")
```

To make it slightly easier to perform the above computation for multiple regions, we can represent the climate data for each region as a vector, i.e., a list of numbers.

```
jerusalem = [73, 67, 43]  
bethlehem = [91, 88, 64]  
hebron = [87, 134, 58]  
jericho = [102, 43, 37]  
nablus = [69, 96, 70]
```

The three numbers in each vector represent the temperature, rainfall, and humidity data, respectively.

Why do we represent the climate data for each region as a vector, i.e., a list of numbers? Because if we assume that we are working on a real life project, we will have a huge amount of data, if we will be using the first way by having three variables for each in this case we will be having a huge amount of variables which will make the process of dealing with it very hard.

We can also represent the set of weights used in the formula as a vector.

```
weights = [w1, w2, w3]
```

We can now write a function `crop_yield` to calculate the yield of apples (or any other crop) given the climate data and the respective weights.

Example:

```
jerusalem = [73, 67, 43]
bethlehem = [91, 88, 64]
hebron = [87, 134, 58]
jericho = [102, 43, 37]
nablus = [69, 96, 70]
w1, w2, w3 = 0.3, 0.2, 0.5
weights = [w1, w2, w3]
```

```
def crop_yield(region, weights):
    result = 0
    for x, w in zip(region, weights):
        result += x * w
    return result

print(crop_yield(jerusalem, weights))
```

2.0 Going from Python lists to Numpy array

The calculation performed by the `crop_yield` (element-wise multiplication of two vectors and taking a sum of the results) is also called the dot product. Learn more about dot product here: <https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/dot-cross-products/v/vector-dot-product-and-vector-length>. The Numpy library provides a built-in function to compute the dot product of two vectors. However, we must first convert the lists into Numpy arrays.

element-wise multiplication of two vectors and taking a sum of the results:

$\text{Jerusalem} = \begin{bmatrix} 73 \\ 67 \\ 43 \end{bmatrix}$ $\text{Weights} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.5 \end{bmatrix}$	<p>element-wise multiplication</p> $\text{Jerusalem} * \text{Weights} = \begin{bmatrix} 73 * 0.3 \\ 67 * 0.2 \\ 43 * 0.5 \end{bmatrix}$ $\text{Jerusalem} * \text{Weights} = \begin{bmatrix} 21.9 \\ 13.4 \\ 12.5 \end{bmatrix}$ $\text{Sum}(\text{Jerusalem} * \text{Weights}) = 56.8$
--	--

figure(2): element-wise multiplication of two vectors

Let's install the Numpy library using the pip package manager.
`pip install numpy`

Next, let's import the numpy module. It's common practice to import numpy with the alias np
`import numpy as np`

We can now use the `np.array` function to create Numpy arrays.
`jerusalem = np.array([73, 67, 43])`
`weights = np.array([w1, w2, w3])`

The type of these two variables is `ndarray`, which is a numpy array and in this way we converted the python's list to numpy array.

Just like lists, Numpy arrays support the indexing notation `[]`.

```
weights[0] output: 0.3  
jerusalem [2] output: 43
```

3.0 Operating on Numpy arrays

We can now compute the dot product of the two vectors using the `np.dot` function
`np.dot(jerusalem, weights)`

We can achieve the same result with low-level operations supported by Numpy arrays: performing an element-wise multiplication and calculating the resulting numbers' sum
`(jerusalem * weights).sum()`

The `*` operator performs an element-wise multiplication of two arrays if they have the same size.

```
import numpy as np
```

```
jerusalem = np.array([73, 67, 43])  
weights = np.array([0.3, 0.2, 0.5])
```

```
crop = np.dot(jerusalem, weights)  
print(crop)
```

Benefits of using Numpy arrays:

- Ease of use: You can write small, concise, and intuitive mathematical expressions like `(jerusalem * weights).sum()` rather than using loops & custom functions like `crop_yield`.
 - Performance: Numpy operations and functions are implemented internally in C++, which makes them much faster than using Python statements & loops that are interpreted at runtime
-

4.0 Multi-dimensional Numpy arrays

We can now go one step further and represent the climate data for all the regions using a single 2-dimensional Numpy array.

Instead of:

```
jerusalem = [73, 67, 43]
bethlehem = [91, 88, 64]
hebron = [87, 134, 58]
jericho = [102, 43, 37]
nablus = [69, 96, 70]
```

It will be:

```
climate_data = np.array([[73, 67, 43],
                          [91, 88, 64],
                          [87, 134, 58],
                          [102, 43, 37],
                          [69, 96, 70]])
```

You may recognize the above 2-d array as a matrix with five rows and three columns. Each row represents one region, and the columns represent temperature, rainfall, and humidity, respectively.

Numpy arrays can have any number of dimensions and different lengths along each dimension. We can inspect the length along each dimension using the **.shape** property of an array.

If an array contains even a single floating point number, all the other elements are also converted to float. All the elements in a numpy array have the same data type. You can check the data type of an array using the **.dtype** property.

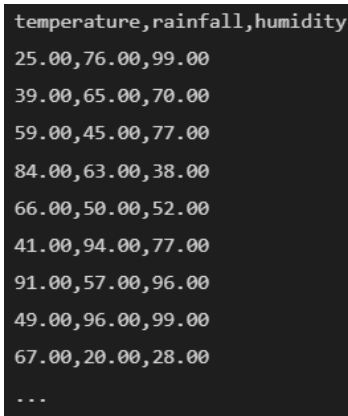
We can now compute the predicted yields of apples in all the regions, using a single matrix multiplication between `climate_data` (a 5x3 matrix) and `weights` (a vector of length 3).

```
weights = np.array([0.3, 0.2, 0.5])
climate_data = np.array([[73, 67, 43],
                          [91, 88, 64],
                          [87, 134, 58],
                          [102, 43, 37],
                          [69, 96, 70]])

crop = np.matmul(climate_data, weights)
print(crop)
```

5.0 Working with CSV data files

Numpy also provides helper functions reading from & writing to files. Let's download a file `climate.txt`, which contains 10,000 climate measurements (temperature, rainfall & humidity) in the following format:



```
temperature,rainfall,humidity
25.00,76.00,99.00
39.00,65.00,70.00
59.00,45.00,77.00
84.00,63.00,38.00
66.00,50.00,52.00
41.00,94.00,77.00
91.00,57.00,96.00
49.00,96.00,99.00
67.00,20.00,28.00
...
```

figure(3): climate.txt content

This format of storing data is known as comma-separated values or CSV.

So, now we need need to install the data file from the internet from this link:

<https://gist.githubusercontent.com/BirajCoder/a4ffcb76fd6fb221d76ac2ee2b8584e9/raw/4054f90adfd361b7aa4255e99c2e874664094cea/climate.csv>

There are two ways to do that, either installing it manually or using `urllib` library to make request

```
import urllib.request
```

```
urllib.request.urlretrieve( file_link_on_the_internet',
'saving_location_with_file_name_and_extension)
```


Then we need to load the data into a numpy array, here we will be using the [numpy.genfromtxt](#) function that is used to load data from a text file, such as a CSV (Comma Separated Values) file, into a NumPy array.

```
climate_data = np.genfromtxt('climate.txt', delimiter=',', skip_header=1)
```

First parameter is the file location and name with the extension

Second parameter is the separator between the data which is comma in our case

Third parameter is defining the header to skip it because it's just label not real data

If we try to print the value of `climate_data` we will see that the file content has been loaded into the numpy array successfully.

```
[ [25. 76. 99.]
  [39. 65. 70.]
  [59. 45. 77.]
  ...
  [99. 62. 58.]
  [70. 71. 91.]
  [92. 39. 76.]]
```

figure(4): climate.txt content into numpy array

We can now perform a matrix multiplication using the `@` operator or using the `numpy.matmul` function to predict the yield of apples for the entire dataset using a given set of weights.

```
climate_data = np.genfromtxt('climate.txt', delimiter=',', skip_header=1)
weights = np.array([0.3, 0.2, 0.5])
crop_yield = np.matmul(climate_data, weights)
print(crop_yield)
print(crop_yield.shape)
```

Let's add the yields to `climate_data` as a fourth column using the `numpy.concatenate`:

```
climate_results = np.concatenate((climate_data, crop_yield.reshape(10000, 1)), axis=1)
```

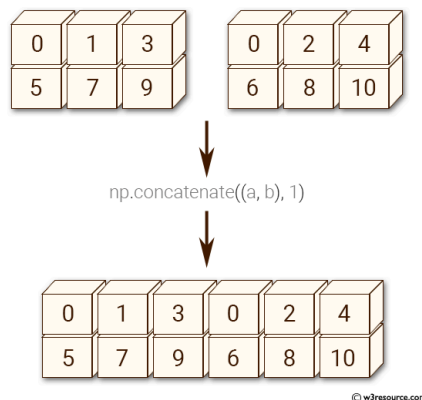
```
[ [25. 76. 99. 72.2]
  [39. 65. 70. 59.7]
  [59. 45. 77. 65.2]
  ...
  [99. 62. 58. 71.1]
  [70. 71. 91. 80.7]
  [92. 39. 76. 73.4]]
```

figure(5): climate.txt content into numpy array

There are a couple of subtleties here:

- Since we wish to add new columns, we pass the argument `axis=1` to `np.concatenate`. The `axis` argument specifies the dimension for concatenation.
- The arrays should have the same number of dimensions, and the same length along each except the dimension used for concatenation. We use the `np.reshape` function to change the shape of `yields` from `(10000,)` to `(10000,1)`.

Here's a visual explanation of `np.concatenate` along `axis=1` (can you guess what `axis=0` results in?):



figure(6): visual explanation of `np.concatenate`

Let's write the final results from our computation above back to a file using the `np.savetxt` function, so the full code becomes as below:

```
1 import numpy as np
2
3 # Loading the data file into numpy array
4 climate_data = np.genfromtxt('climate.txt', delimiter=',', skip_header=1)
5 weights = np.array([0.3, 0.2, 0.5]) # defining the weights array
6
7 # multiplying the climate data with the weights
8 crop_yield = np.matmul(climate_data, weights)
9
10 # adding the crop_yield as new column in the climate_data file
11 climate_results = np.concatenate((climate_data, crop_yield.reshape(10000, 1)), axis=1)
12
13 try:
14     np.savetxt('climate_results.txt',
15               climate_results,
16               fmt='%.2f',
17               delimiter=',',
18               header='temperature,rainfall,humidity,yield_apples',
19               comments='')
20     print("[INFO] Data successfully saved to 'climate_results.txt'")
21 except Exception as e:
22     print(f"[ERROR] Error: {e}")
```

figure(7): climate data example full code

Numpy provides hundreds of functions for performing operations on arrays. Here are some commonly used functions:

- Mathematics: np.sum, np.exp, np.round, arithmetic operators
 - Array manipulation: np.reshape, np.stack, np.concatenate, np.split
 - Linear Algebra: np.matmul, np.dot, np.transpose, np.eigvals
 - Statistics: np.mean, np.median, np.std, np.max
-

6.0 Arithmetic operations, broadcasting and comparison

Numpy arrays support arithmetic operators like +, -, *, etc. You can perform an arithmetic operation with a single number (also called scalar) or with another array of the same shape. Operators make it easy to write mathematical expressions with multi-dimensional arrays.

We will apply some arithmetic operation on the below arrays:

```
arr2 = np.array([[1, 2, 3, 4],  
                [5, 6, 7, 8],  
                [9, 1, 2, 3]])
```

```
arr3 = np.array([[11, 12, 13, 14],  
                [15, 16, 17, 18],  
                [19, 11, 12, 13]])
```

```
arr4 = np.array([4, 5, 6, 7])
```

- Adding a scalar: print(arr2 + 3)

```
[[ 4  5  6  7]  
 [ 8  9 10 11]  
 [12  4  5  6]]
```

- Element-wise subtraction: print(arr3 - arr2)

```
[[ 4  5  6  7]  
 [ 8  9 10 11]  
 [12  4  5  6]]
```

- Division by scalar: print(arr2 / 2)

```
[[0.5 1.  1.5 2. ]  
 [2.5 3.  3.5 4. ]  
 [4.5 0.5 1.  1.5]]
```

- Element-wise multiplication: `print(arr2 * arr3)`

```
[[ 11  24  39  56]
 [ 75  96 119 144]
 [171  11  24  39]]
```

- Modulus with scalar: `print(arr2 % 4)`

```
[[1 2 3 0]
 [1 2 3 0]
 [1 1 2 3]]
```

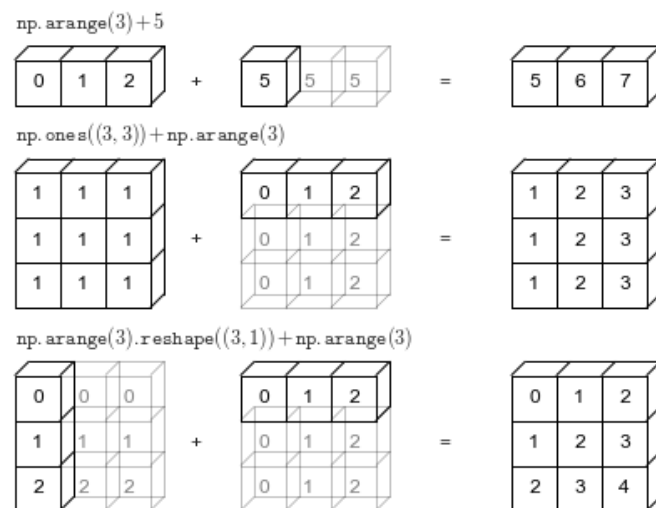
6.1 Array Broadcasting

Array Broadcasting Numpy arrays also support broadcasting, allowing arithmetic operations between two arrays with different numbers of dimensions but compatible shapes. Let's look at an example to see how it works.

`arr2 + arr4`

```
[[ 5  7  9 11]
 [ 9 11 13 15]
 [13  6  8 10]]
```

When the expression `arr2 + arr4` is evaluated, `arr4` (which has the shape (4,)) is replicated three times to match the shape (3, 4) of `arr2`. Numpy performs the replication without actually creating three copies of the smaller dimension array, thus improving performance and using lower memory.



figure(8): Arrays Broadcasting

So what exactly happened is we extended the shape of the smaller shape array, then we made a wise addition, the complete explanation of the process is illustrated in the below figure.

array 1 = $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \end{bmatrix}$

array 2 = $\begin{bmatrix} 4 & 5 & 6 & 7 \end{bmatrix}$

So When we say array 1 + array 2 means:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 \end{bmatrix} = \begin{bmatrix} 5 & 7 & 9 & 11 \\ 9 & 11 & 13 & 15 \\ 13 & 6 & 8 & 10 \end{bmatrix}$$

What we did is extending the shape of the smaller shape array (array 2), and make wise addition.

figure(9): Arrays Broadcasting Illustration

6.2 Array Comparison

Numpy arrays also support comparison operations like ==, !=, > etc. The result is an array of booleans.

```
arr1 = np.array([[1, 2, 3], [3, 4, 5]])
arr2 = np.array([[2, 2, 3], [1, 2, 5]])
```

```
print(arr1 == arr2)
print(arr1 != arr2)
print(arr1 >= arr2)
print(arr1 < arr2)
```

Output:

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

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

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

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

Array comparison is frequently used to count the number of equal elements in two arrays using the sum method. Remember that True evaluates to 1 and False evaluates to 0 when booleans are used in arithmetic operations. `(arr1 == arr2).sum()`

6.3 Array indexing and slicing

Numpy extends Python's list indexing notation using `[]` to multiple dimensions in an intuitive fashion. You can provide a comma-separated list of indices or ranges to select a specific element or a subarray (also called a slice) from a Numpy array.

```
arr3 = np.array([
    [[11, 12, 13, 14],
     [13, 14, 15, 19]],

    [[15, 16, 17, 21],
     [63, 92, 36, 18]],

    [[98, 32, 81, 23],
     [17, 18, 19.5, 43]]])
```

- Single Element:
`arr3[1, 1, 2]`
- Subarray using ranges:
`arr3[1:, 1, 3]`
- Mixing indices and ranges:
`arr3[1:, 1, :3]`
- Using fewer indices:
`arr3[1]`
`arr3[:2, 1]`

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

figure(10): Array indexing and slicing

7.0 More NumPy Functions

- Create array with range:
`#np.array(start, end, steps)`
`b = np.array(0, 11, 2) #b = [0, 2, 4, 6, 8, 10]`
- Create array of zeros:
`#np.zeros([dimensions, number of items])`
`a = np.zeros([2, 10])`
- Create array of ones:
`#np.full(shape, fill_value)`
`a = np.full([2, 2], 67)`
- Get square root:
`#np.sqrt(array)`
- Get min/max value:
`#np.min/max(array)`
- Get sin/cos/etc value:
`#np.sin(array)`
- Copy array into array and make it follow changes of the parent array:
`a = np.array([1,2,3])`
`b = a.view() #[1, 2, 3]`
`a[0] = 9 #b = 9`
- Iterate over an array
`for x in np.nditer(array):`
 `print(x)`
- sort:
`a = np.array([1,2,3,6,5,4,9,8,7])`
`a.sort # [1,2,3,4,5,6,7,8,9]`
`# same for string and boolean arrays`
- Searching by value to get index:
`a = np.array([1,2,3,6,5,4,9,8,7])`
`x = np.where(a==2)`
`# output: (array([1], dtype=int64),)`