Table of contents

Introduction to `Numpy`

Bois! We are going Deep DEep DEEp DEEEEEP inside `ML`. And we will face some roadblocks. First one is `numpy`.

This is a entry point and the first `boss` fight of this series. Because almost everything after this point is written in `numpy` or uses `numpy` functions. So, understanding and having a good knowledge of `numpy` is very important. So, without further ado, let's defeat `numpy`.

- LinkedIn
- YouTube
- gtihub
- Gmail
- discord

Installation

You need `conda` installed in you system. If you don't have it, you can install it from here.

If you have `conda` installed, Just run the following command in your terminal:

```
conda install numpy
```
And

If you don't have `conda` installed, you can still intall `numpy` on base `python` by running the following command:

```
pip install numpy
```

> I heavily recommend you to use `conda` for installing `numpy` because it will install all the dependencies of `numpy` and you will have less errors as our python package ecosystem grows.

Intro

`Numpy` is faster `python`.

That's it end the article here. If you know `python` you already know 80% of `numpy`.

Numpy specializes in `numerical computing`, specifically `arrays`.

> If you take python and optimise it so hard that it becomes faster than C++, you can call it `numpy`.

We can do everything we can do in `python` with `numpy` but it's waaaay faster than `python`.

But the main use case of `numpy` is `array/vector/matrix` operations. And that's what we will be doing in this article.

Let's take an array.

In [99]:
```
arr = [1,2,3,4,5]
arr
```

Out[99]:
```
[1, 2, 3, 4, 5]
```

Here you can see a very simple python list. Which is pretty powerful on its own. But with numpy it becomes a superpower.

We can transform any array into a numpy array by using the `array()` function from numpy. We have to import `numpy` first because it's a external library.

In [100…
```
import numpy as np

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

Out[100…
```
array([1, 2, 3, 4, 5])
```

> In python you can rename a library when imported by using the `as` keyword. For example, if you import `numpy` as `np` you can use it as `np.array()` instead of `numpy.array()`.

Here you can see that by using the `np.array()` we are transforming our simple python list into a numpy array.

We can do the same thing with a 2D list as well.

In [101…
```
arr_2D = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
narr_2D = np.array(arr_2D)
narr_2D
```

Out[101…
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

A numpy array gives us a lot of methods and attributes to work with the data.

> A vector is a one-dimensional array which is a traditional array of numbers.
> A matrix is a two-dimensional array.

Numpy Methods

There are a lot of methods that we can use to make and work with numpy arrays.

First we can talk about the `arange` method.

Numpy Arange

This is equivalent to the python `range` function. Works exactly the same way.

In the range function, we can specify the start, stop, and step values.

And it generates a sequence of numbers from start to stop by step.

This is exactly what the `arange` method does. It also takes the start, stop, and step values and generates a sequence of numbers from start to stop by step.

In [102…
```
np.arange(10)
```

Out[102…
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

> Just like for the range function, the start value in inclusive and the end value is exclusive. It's the same for `arange`. By default the starting index is set to 0 and the steps are 1.

In [103…
```
np.arange(35, 45, 2)
```

Out[103…
```
array([35, 37, 39, 41, 43])
```

Some Quick Numpy Array generators

If you want a array of all zeros, you can use the `zeros` function:

In [104…
```
np.zeros(5)
```

Out[104… `array([0., 0., 0., 0., 0.])`

This will generate a numpy array of 5 zeros. We can also make a `matrix(2D)` array by specifying the number of `rows and columns` in a `tuple`.

In [105…
```
np.zeros((3,4))
```

Out[105…
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

And it'll generate a `3x4` matrix. The cool part is we can make 3D or more than 4D array of all zeros with this.

In [106…
```
np.zeros((3,4,5))
```

Out[106…
```
array([[[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]],

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

       [[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]]])
```

In [107…
```
np.zeros((2,3,4,2))
```

Out[107…
```
array([[[[0., 0.],
         [0., 0.],
         [0., 0.],
         [0., 0.]],

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

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


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

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

        [[0., 0.],
         [0., 0.],
         [0., 0.],
         [0., 0.]]]])
```

Numpy has another function that can generate arrays filled with only `one's` . This function is called `ones` .

In [108…
```
print('Array of 4 ones')
a = np.ones(4)
print(a)

print('2D array of all ones')
b = np.ones((2,2))
print(b)

print('3D array of all ones')
c = np.ones((2,2,2))
print(c)
```

```
Array of 4 ones
[1. 1. 1. 1.]
2D array of all ones
[[1. 1.]
 [1. 1.]]
3D array of all ones
[[[1. 1.]
  [1. 1.]]

 [[1. 1.]
  [1. 1.]]]
```

This method works exactly same the `zeros` method, but it generates array of any dimentions with all elements as `Ones` .

Now, another very useful method I want to talk about is `linspace` method. This method is used to generate array with evenly spaced elements.

Let me break it down with an example.

In [109…
```
np.linspace(1, 10, 5)
```

Out[109…
```
array([ 1.  ,  3.25,  5.5 ,  7.75, 10.  ])
```

`linspace` function takes 3 arguments:

- `start` : The starting value of the range
- `stop` : The ending value of the range
- `num` : The number of values in the range

This might look like the `arange` function, but it's not the same. Linspace generates a range of numbers with `evenly spaced values` between `start` and `stop` with `num` number of values.

Let's say I want 5 numbers between 0 and 5. `Linspace` will return a list of 5 numbers between 0 and 5 with all the numbers evenly spaced.

In [110…
```
np.linspace(1,5,5)
```

Out[110…
```
array([1., 2., 3., 4., 5.])
```

As you can see we have an array `[1,2,3,4,5]` . You can put any number as arguments and it will generate the array.

> Let's say we want to create an array of 30 numbers from 45 to 100. All the numbers should be equally spaced.

In [111…
```
np.linspace(start=45, stop=100, num=30)
```

Out[111…
```
array([ 45.        ,  46.89655172,  48.79310345,  50.68965517,
        52.5862069 ,  54.48275862,  56.37931034,  58.27586207,
        60.17241379,  62.06896552,  63.96551724,  65.86206897,
        67.75862069,  69.65517241,  71.55172414,  73.44827586,
        75.34482759,  77.24137931,  79.13793103,  81.03448276,
        82.93103448,  84.82758621,  86.72413793,  88.62068966,
        90.51724138,  92.4137931 ,  94.31034483,  96.20689655,
        98.10344828, 100.        ])
```

I hope, you understood the `linspace` function.

Now, before going to other things, , one thing in this linspace section, got my mind. It the `dots` after each number in this example.

In [112…
```
np.linspace(1,5,5)
```

Out[112…
```
array([1., 2., 3., 4., 5.])
```

We are generating 5 numbers from a uniform distribution between 1 to 5 and we can see that the numbers are `1,2,3,4,5` but still there is the dot after each number(`1.` , `2.` , `3.` , `4.` , `5.`). Why is that?

We should check the type of the return value of `numpy.linspace(1,5,5)` .

In [113…
```
arr = np.linspace(1,5,5)
arr.dtype
```

Out[113…
```
dtype('float64')
```

> We can use the `arr.dtype` attribute from any numpy array variable to get the data type of the array.

As you can see even though we have a numpy array of all whole numbers, the data type is still `float64` .

So, by default, `linspace` function returns a `float64` array.

Now let's talk about `indentity` matrix.

If you don't know what an identity matrix is, it's a square matrix with ones on the main diagonal and zeros elsewhere.

> It is also used to represent `1` in matrix.

Simply, `1` can be written as

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

It can be a square matrix of any size. `1x1` , `2x2` , `3x3` , `n x n` etc.

All of this is just a matrix representation of `1` .

And in numpy, we can get an identity matrix of any size by using `np.eye` function.

In [114…
```
np.eye(3)
```

Out[114…
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

> When you multiply a matrix by a Identity matrix, you get the same matrix back.

Random

One of the most useful modules in `numpy` is `random`. It provides a number of functions to generate random numbers it different distributions.

> We will visualize what different distributions look like in the visualizations library articles.

This `random` module is not far off from the `random` module in Python's standard library. The main difference is that `numpy` has a number of functions to generate random numbers in different distributions.

So, what about we generate a array of random numbers?

In [115…
```
np.random.rand()
```

Out[115…
```
0.104472293407741
```

> `rand()` mathod will return a random number between 0 and 1.

We can get an array of random numbers between 0 and 1 by using the `rand()` method too.

In [116…
```
np.random.rand(3)
```

Out[116…
```
array([0.43158752, 0.93239361, 0.44953181])
```

If you want a 2D array, you can just pass the number of rows and columns.

In [117…
```
np.random.rand(3,4)
```

Out[117…
```
array([[0.24906832, 0.10305916, 0.00878262, 0.10214106],
       [0.57093834, 0.87229698, 0.8436116 , 0.98163268],
       [0.39778114, 0.13329881, 0.16237252, 0.34686357]])
```

We can make as many dimentions we want. Just pass the size of the dimentions and this method will create the array with the given dimentions.

In [118…
```
np.random.rand(2,3,2,4)
```

Out[118…
```
array([[[[0.09585684, 0.29087593, 0.24061427, 0.73769875],
         [0.02423596, 0.43138607, 0.6835523 , 0.24803958]],

        [[0.11992288, 0.63738875, 0.26562533, 0.05615219],
         [0.361197  , 0.02373971, 0.42246836, 0.01021005]],

        [[0.08753078, 0.34904328, 0.47139676, 0.21621414],
         [0.57750292, 0.15275787, 0.3866508 , 0.13562721]]],


       [[[0.55345656, 0.10652228, 0.80768043, 0.83750555],
         [0.66459144, 0.56850705, 0.91782622, 0.70125719]],

        [[0.79756121, 0.46633319, 0.38049552, 0.68322635],
         [0.75353685, 0.09774307, 0.46149039, 0.66267493]],

        [[0.1947155 , 0.7963879 , 0.22959122, 0.29666245],
         [0.47503294, 0.36008213, 0.45462091, 0.68463677]]]])
```

Now, the values generated from the `rand` function is from the `uniform` distribution. We can generate random numbers from `gaussian` distribution or `normal` distribution using the `randn` function.

In [119…
```
np.random.randn(3,4)
```

Out[119…
```
array([[ 5.15903818e-02, -3.47550608e-01,  3.63186890e-01,
         1.06059104e-01],
       [ 1.41121504e+00,  5.28145615e-01,  8.51892804e-02,
        -5.18088223e-01],
       [ 3.61883780e-01,  1.09034214e+00,  3.33586450e-01,
        -8.58124283e-04]])
```

This works as same as the `rand` function. But the number generated is from `normal distribution`. That's why you can see negetive numbers as well.

These are all numbers between -1 and 1.

But what if you need interger numbers? You can use `np.random.randint(low, high, size)`

In [120…
```
np.random.randint(1, 100, size=5)
```

Out[120…
```
array([48, 66,  6, 81, 20])
```

`randint` method will generate rnaodm integers between min and max values and if you want an array or a matrix or highers dimension you can set the size or dimension size as a tuple.

In [121…
```
np.random.randint(20,50, size=(2,3,6))
```

Out[121…
```
array([[[24, 40, 43, 47, 46, 49],
        [24, 29, 41, 48, 32, 38],
        [41, 25, 41, 48, 39, 49]],

       [[36, 28, 26, 41, 34, 42],
        [46, 33, 47, 24, 41, 46],
        [45, 21, 40, 39, 20, 32]]])
```

Pretty easy right? 😃

This module is very useful when you want a dummy data to work with. It's a very simple module that you can use to create a dummy dataset very fast.

Now, I hope you got the gist of what numpy is and how it works. Now time for some useful functions that can help you undersand the `array` better.

Some Useful Functions

With numpy, it is very easy to create a multidimensional array. But sometime seeing a multidimensional array can be a bit confusing and hard to wrap your head around. But it helps if we can know the shape of a array.

So, let's experiment.

In [122…
```
rarr = np.random.randint(1,50,size=(10,5))
rarr
```

Out[122…
```
array([[23, 48, 23, 39, 17],
       [27, 41, 27, 35, 47],
       [48, 38,  3,  6, 29],
       [41, 47, 43, 27, 45],
       [24, 21, 37, 16, 19],
       [44, 28,  4, 13, 14],
       [12, 43, 49, 43,  7],
       [44, 11, 41, 18, 15],
       [39, 11,  6, 35, 10],
       [27, 17, 37, 19,  5]])
```

Here I created a matrix of size `10x5` , for me it is pretty easy to understand, but if you are a only seeing this array for the first time you might get confused. So, what we can do is, see the shape of the array.

In [123…
```
rarr.shape
```

Out[123…
```
(10, 5)
```

`shape` is a numpy array attribute that returns the shape of the array. I created the `10x5` array and with simple command you can also see that it is a 2D matrix that has 10 rows and 5 columns.

Now, what if you want to change the size of the rows or columns? You can do that by using the `reshape` method. Let's see how it works.

In [124…
```
rarr.reshape(5,10)
```

Out[124…
```
array([[23, 48, 23, 39, 17, 27, 41, 27, 35, 47],
       [48, 38,  3,  6, 29, 41, 47, 43, 27, 45],
       [24, 21, 37, 16, 19, 44, 28,  4, 13, 14],
       [12, 43, 49, 43,  7, 44, 11, 41, 18, 15],
       [39, 11,  6, 35, 10, 27, 17, 37, 19,  5]])
```

Here you can see I interchange the rows and columns of a matrix. And it is seamlessly done by numpy.

> reshape() method doesn't reshape the matrix, it just returns a new matrix with the same data but with a different shape. So, to permanently change the shape of a matrix, you can store the reshaped matrix in a new variable or in the same variable.

Now, can we change the whole array into any shape?

In [125…
```
rarr.reshape(2,5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[125], line 1
----> 1 rarr.reshape(2,5)

ValueError: cannot reshape array of size 50 into shape (2,5)
```

And it's showing an error. This error is because we are trying to change the share of the array in a wrong shape. We need to change the shape of the array in a way that it doesn't doesn't change the elements of the array.

The `rarr` array has a shape of `(10, 5)`. So, it has 50 elements. But, we want to change the shape of the array to `(2,5)`. Which means the matrix can have only 10 elements. Which is not possible. So, we should be careful when reshaping the array.

> One clever way to reshape is to multiply the sizes of different dimensions and check if it's equal to the total number of elements in the array.

So, that means the number of elements should not change so. If I want to reshape the array to `(5,2,5)` then it should work right?

In [126…
```
rarr.reshape(5,2,5)
```

Out[126…
```
array([[[23, 48, 23, 39, 17],
        [27, 41, 27, 35, 47]],

       [[48, 38,  3,  6, 29],
        [41, 47, 43, 27, 45]],

       [[24, 21, 37, 16, 19],
        [44, 28,  4, 13, 14]],

       [[12, 43, 49, 43,  7],
        [44, 11, 41, 18, 15]],

       [[39, 11,  6, 35, 10],
        [27, 17, 37, 19,  5]]])
```

Yes! It works. Because in the end `5x2x5=50` and we are not changing the element size.

With that out of the way, now let's think about some more thing we should be able to do.

Ow! I got one.

How do we find the maximum or the minimum value in a numpy array?

In [127…
```
arr = np.random.randint(1,100,size=30)
```

For finding the maximum value:

In [128…
```
arr.max()
```

Out[128…
```
np.int64(97)
```

For finding the minimum value:

In [129…
```
arr.min()
```

Out[129…
```
np.int64(7)
```

Noice! But what if I tell you to find the index of the maximum value or the minimum value of a list? `Python` doesn't provide a direct way to do that but `Numpy` does.

To get the index of the maximum value of a numpy array:

In [130…
```
arr.argmax()
```

Out[130…
```
np.int64(5)
```

To find the index of the minimum value:

In [131…
```
arr.argmin()
```

Out[131…
```
np.int64(27)
```

And I think that covers the elementary stuff. And we can now go on to the more complecated stuff.

Indexing

One of the best things about numpy is that it allows you high level indexing. So you can do things like,

- `Selecting`.
- `Slicing`.
- `n-D matrix slicing`.
- `Conditional indexing`.

And many more.

Also, you can do indexing just like you would in python.

In [132…
```
arr = np.arange(11)
```

```
arr
```

Out[132…
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [133…
```
arr[5]
```

Out[133…
```
np.int64(5)
```

As you can see selecting any index is the same as we do in python lists.

And indexing like [-1] is also present in numpy arrays.

In [134…
```
arr[-1]
```

Out[134…
```
np.int64(10)
```

It's all the same as the base python. But numpy excels in handling slicing. For a normal vector, it's the same as python.

We can use colon `[:]` to select all the elements of an array.

> `[start:end:step]` Start is set to 0 by default. End is set to the length of the array by default. Step is set to 1 by default.

So, Let's say I want values from index 0 to 4, I can do this:

In [135…
```
arr[0:5]
```

Out[135…
```
array([0, 1, 2, 3, 4])
```

The end value is exclusive, so `5th` index is excluded and we get a list of values from `0` to `4`.

> Just remember to put (index you want + 1). If you want from `0` to `4` then you have to put `[0:5]` or if you want from `3` to `7` then you have to put `[3:8]`.

As the default start index is set to `0` if you want a slice from beginning to any index you can keep the start index empty and just put the end index.

In [136…
```
arr[:3]
```

Out[136…
```
array([0, 1, 2])
```

It's the same as

In [137…
```
arr[0:3:1] # start from 0, step by 1, stop at 3
```

Out[137…
```
array([0, 1, 2])
```

If you want a slice from a specific index to the end of the array, you can keep the end index empty too. Because the end index is set to the length of the array by default.

In [138…
```
arr[5:]
```

Out[138… array([5, 6, 7, 8, 9, 10])

You can also experiment with the step size too

In [139…
```
arr[:8] # first 8 elements
```

Out[139… array([0, 1, 2, 3, 4, 5, 6, 7])

In [140…
```
arr[:8:2] # start from index 0 and take every 2nd element to the 7th eleme
```

Out[140… array([0, 2, 4, 6])

Looks good. But one thing that might just through you off, is when you want to store a slice in a different variable.

In [141…
```
arr_2 = arr[1:7]
arr_2
```

Out[141… array([1, 2, 3, 4, 5, 6])

Easy and looks just like python right?

Now, we can do indexing in this new sliced array.

In [142…
```
arr_2[2:]
```

Out[142… array([3, 4, 5, 6])

Looks great!

Nothing out of the ordinary here. Let's try to change a value in the array.

In [143…
```
arr_2[2] = 99
arr_2
```

Out[143… array([1, 2, 99, 4, 5, 6])

We can change values of a numpy array exactly we do in a python list.

BUUUUUUT

If I look at the main array, Some interesting will happen.

In [144…
```
arr
```

Out[144… array([0, 1, 2, 99, 4, 5, 6, 7, 8, 9, 10])

WHAT!

The Main array changed! Why?! HOW?!

I changed the value of the sliced array but why would the main array change?

This is because the numpy doesn't copy the array when slicing. It just broadcasts the sliced array to any other variable.

So, when we `slice` the array, it is `not` taking any `new space` in the memory. It is just that the sliced that is `directly broadcasted`. And if we `store` that slice in a `new variable`, it'll look like and feel like a new array but still it is a `part` of the main array which we are just using as a new variable.

So, if we `change` an array that is `sliced` from another numpy array, it will change the main array.

Let's make a new array named `main_arr`:

In [145…
```
main_arr = np.arange(20)
main_arr
```

Out[145…
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

Now get a slice of the array,

In [146…
```
sliced_arr = main_arr[3:13]
sliced_arr
```

Out[146…
```
array([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

Change the whole slice to 12

In [147…
```
sliced_arr[:] = 12 # changing the values of the whole array
sliced_arr
```

Out[147…
```
array([12, 12, 12, 12, 12, 12, 12, 12, 12, 12])
```

Now, let's take a look at the main array

In [148…
```
main_arr
```

Out[148…
```
array([ 0,  1,  2, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 13, 14, 15, 16,
       17, 18, 19])
```

And you can see the that the sliced part of the main array has also changed to 12.

> That's why be careful when you are slicing an array. Because you might unintentionally change the original array.

And if you want a copy of the slice, you can use the copy() method.

This method will return a copy of the slice instead of braodcasting the slice from the original array.

In [149…
```
main_arr = np.arange(10)
print(f"main array is {main_arr}")

copied_slice = main_arr[4:9].copy()
print(f"copied slice is {copied_slice}")
```

```
#changing the whole array to 99
copied_slice[:] = 99
print(f'changed the array to {copied_slice}')
print(f'main array is {main_arr}')
```

```
main array is [0 1 2 3 4 5 6 7 8 9]
copied slice is [4 5 6 7 8]
changed the array to [99 99 99 99 99]
main array is [0 1 2 3 4 5 6 7 8 9]
```

This way your original array will be unaffected.

Now, for the fun part.

Matrix Indexing

`Numpy` specializes in matrix handling.

In [150…
```
matrix = np.array([[1, 2, 3, 4], [10, 20, 30, 40], [100, 200, 300, 400], [
matrix
```

Out[150…
```
array([[   1,    2,    3,    4],
       [  10,   20,   30,   40],
       [ 100,  200,  300,  400],
       [1000, 2000, 3000, 4000]])
```

In numpy, there's two ways you can index a matrix:

The python way(double square brackets)

In [151…
```
matrix[0][1]
```

Out[151…
```
np.int64(2)
```

Or,

The Numpy way (single bracket with comma)

In [152…
```
matrix[0,1]
```

Out[152…
```
np.int64(2)
```

There are both equivalent ways to get elements from a matrix. I would prefer to use the second way, because we are using `numpy` and we should stick to it's syntax to avoid confusions.

Now, let's talk about matrix slicing. The Broadcasting from the original array is still valid. But we get some very useful features for matrix slicing in `numpy`.

Normally if we have a 2D list in python. If we want a cirtain part of that list we have to do it logically.

For example, let's say we have a list like this:

In [153…
```
matrix_python = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
matrix_python
```

Out[153…
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

I want the `[[5,6], [8,9]]` slice.

To get that first we need to slice by rows.

In [154…
```
slice = matrix_python[1:]
slice
```

Out[154…
```
[[4, 5, 6], [7, 8, 9]]
```

And now, the issue arrives that we cannot slice by columns in python. So, we need to use a for loop to get the rows in the slice and then get the slice of each row and add them to a new array.

> I know I wrote some shiz you guys didn't understand, me too.

In [155…
```
slice = [arr[1:] for arr in slice]
slice
```

Out[155…
```
[[5, 6], [8, 9]]
```

And we have the slice we need. This way too much work. Numpy has a great way to do this.

We have this matrix.

In [156…
```
matrix
```

Out[156…
```
array([[   1,    2,    3,    4],
       [  10,   20,   30,   40],
       [ 100,  200,  300,  400],
       [1000, 2000, 3000, 4000]])
```

And I want

$$\begin{bmatrix} 20 & 30 \\ 200 & 300 \end{bmatrix}$$

We can do this simply slicing the array because numpy has column slicing too.

In [157…
```
matrix_slice = matrix[1:3, 1:3]
matrix_slice
```

Out[157…
```
array([[ 20,  30],
       [200, 300]])
```

> We use the comma separated way to slice in numpy for matrices.

If we only want to slice by row.

We can do that normal way.

In [158…
```
matrix[1:]
```

Out[158…
```
array([[  10,   20,   30,   40],
       [ 100,  200,  300,  400],
       [1000, 2000, 3000, 4000]])
```

The best part is if you want `slice by column`, you can use a `comma` to separate the `row` and `column` index.

In [159…
```
matrix[:,1:3]
```

Out[159…
```
array([[   2,    3],
       [  20,   30],
       [ 200,  300],
       [2000, 3000]])
```

Here I'm slicing only the column from 1 to 2.

> The end value will be excluded.

So, by `[:, 1:3]`, I saying give me all the rows but the columns from 1 to 2.

> You cannot do direct column slicing like `df[, 1:3]`. It'll show an error.

This can be a tough concept to get the hang of. So, I'll give you a task.

> Make a random array of size 20. Reshape it into a 4x5 matrix. Then, trying indexing the that matrix and then slicing different parts of it.

Try it yourself and remember the end value is excluded.

> I was traumatized by this when I was learning.

There's more type of selections you can do.

Conditional Selection

Row and Column indexing is one those numpy concepts that is very important to understand but the use-cases are limited.

`COnditional selection` on the other hand, is a very a very powerful concept and it's use is very frequent in data analysis.

Because it let's you select the data based on conditions.

Let's see an example.

In [160…
```
arr = np.random.randint(1,101, size=16)
arr
```

Out[160…
```
array([77, 61,  8, 67, 63, 18, 81, 20, 49, 99, 20, 61, 53, 50, 14, 10])
```

Here, I have a completely random array of numbers. And I don't know what's inside this array. Now, for some reason, I need to an array of all the numbers that are greater than 25.

Now, if you are somewhat experienced in programming, you might be thinking that we can use a for loop to do this. But, numpy has it's own way.

First let's see what happens if we directly compare the array with a number.

In [161…
```
arr > 25
```

Out[161…
```
array([ True,  True, False,  True,  True, False,  True, False,  True,
        True, False,  True,  True,  True, False, False])
```

Owh! that's shocking instead of returning a single boolean value, when we compare with a numpy array it returns a `boolean array` of the `same` size.

Now what are these boolean values? They represent if the corresponding element of the array meets the condition or not.

As we can see, first value of that array is `True`, that means the first element should be `greater` than `25`.

Let's have a look at the first element of the array.

In [162…
```
arr[0]
```

Out[162…
```
np.int64(77)
```

And yes! It's `77` it is most definitely greater than `25`.

Now, the fun part is we can use this boolean array to get a subset of all the values that are greater than `25`.

In [163…
```
boolean_arr = arr > 25
boolean_arr
```

Out[163…
```
array([ True,  True, False,  True,  True, False,  True, False,  True,
        True, False,  True,  True,  True, False, False])
```

In [164…
```
arr[boolean_arr]
```

Out[164…
```
array([77, 61, 67, 63, 81, 49, 99, 61, 53, 50])
```

And look at that, we got a list of all the numbers that are greater than 25.

This is called `conditional indexing/selection`.

We can directly put the condition inside brackets to get the list of numbers that are greater than 25.

In [165…
```
arr[arr > 25]
```

Out[165…
```
array([77, 61, 67, 63, 81, 49, 99, 61, 53, 50])
```

> I know it's kinda weird to see the variable named inside the brackets, but it's just to tell which elements of the array are satisfying the condition.

all the elements that has a corresponding value of `True` is then returned as a new numpy array.

And when I say `new`, I mean completely new array.

> Check if the array you are getting from conditional selection can change the original array by changing the value of a single element.

We can use every conditional operators and it'll work the same. So, no need of any for loops and no need of a empty array to store the result.

> Remember, the more you experiment with the numpy, the more you'll understand it better.

Numpy Operations

The last topic we'll cover is the `numpy operations`.

We know that we have operators in python that can be used to perform mathematical operations on numbers. But what about numpy?

As numpy specializes in `arrays`, numpy array operations are different from the normal operations.

Let's see an example.

In [166…
```
arr = np.arange(11)
arr
```

Out[166…
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [167…
```
arr + arr
```

Out[167…
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20])
```

As you can we `arr+arr` is adding all the elements of the array.

This is because of the `broadcasting` rule of numpy.

When you are `adding` two numpy arrays, the arrays broadcast together and the result is an array of all the elements of the two arrays added together.

We will have the same effect if we `subtract` or `multiply` or `divide` two numpy arrays.

But one other interesting thing is that if we `add` a `scalar` to a numpy array, the scalar is added to all the elements of the array.

> 0 dimensional values are called scalars. A single integer is a scalar. A single float is a scalar.

In [168…
```
arr + 100
```

Out[168…
```
array([100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110])
```

And as you can see, the output is an numpy array with `100` added to each element of the array.

> This exactly like conditional selection we talked about earlier.

We can we can subtract a single integer and it'll be subtracted from all the elements of the array. This is also because of the `broadcasting` property of numpy.

In [169…
```
arr-1
```

Out[169…
```
array([-1,  0,  1,  2,  3,  4,  5,  6,  7,  8,  9])
```

Multiplying by a scalar:

In [170…
```
arr*2
```

Out[170…
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20])
```

Division:

In [171…
```
arr/2
```

Out[171…
```
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ])
```

Now, One extra thing I want to ask you about is that, what happens if we `divide` a number by `zero`?

We should get an error right?

In [172…
```
25/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[172], line 1
----> 1 25/0

ZeroDivisionError: division by zero
```

Normally, python shows the error but in numpy it's a little different.

In [173…
```
arr/0
```

```
/tmp/ipykernel_187240/4291252909.py:1: RuntimeWarning: divide by zero encou
ntered in divide
  arr/0
/tmp/ipykernel_187240/4291252909.py:1: RuntimeWarning: invalid value encoun
tered in divide
  arr/0
```

Out[173… `array([nan, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf])`

It still returns an array but all the values are set to infinity.

And give you a little warning. That's it.

And if you look closely, as the first element of the array is `zero`, when we are dividing it zero, the value it returns is `Nan`. This is because zero divided by zero is undefined.

> So, any number divided by zero is `infinity` in numpy.

> Zero divided by zero is `Nan` in numpy.

Now, let's talk about universal functions.

Universal Functions

`Universal functions` are functions that can be applied to any type of data in numpy.

one example would be `np.max()` which can be used to find the maximum value in a numpy array.

In [174…
```
np.max(arr)
```

Out[174… `np.int64(10)`

A numpy array already has a `max()` method.

> We talked about the `max()` method at the basics of numpy.

In [175…
```
arr.max()
```

Out[175… `np.int64(10)`

Although these two do the same thing, but some arrays might not have a `max` mathod. If this happens we can use `numpy.max` instead.

Here's some usefull `universal functions` that numpy provides.

In [176…
```
#Examples of universal functions in numpy
print('Maximum value in array is: ')
np.max(arr)#Maximum value in array
```

```
Maximum value in array is:
```

Out[176… `np.int64(10)`

In [177…
```
print('Minimum value in array is: ')
np.min(arr)#Minimum value in array
```

```
Minimum value in array is:
```

Out[177… `np.int64(0)`

In [178…
```
print('Index of maximum value in array is: ')
np.argmax(arr)#Index of maximum value in array
```

```
Index of maximum value in array is:
```

Out[178… np.int64(10)

In [179…
```
print('Index of minimum value in array is: ')
np.argmin(arr)#Index of minimum value in array
```

```
Index of minimum value in array is:
```

Out[179… np.int64(0)

In [180…
```
print('Shape of array is: ')
np.shape(arr)#Shape of array
```

```
Shape of array is:
```

Out[180… (11,)

In [181…
```
print('Sum of all elements in array is: ')
np.sum(arr) #Sum of all elements in array
```

```
Sum of all elements in array is:
```

Out[181… np.int64(55)

In [182…
```
print('Mean of all elements in array is: ')
np.mean(arr)#Mean of all elements in array
```

```
Mean of all elements in array is:
```

Out[182… np.float64(5.0)

In [183…
```
print('Standard deviation of all elements in array is: ')
np.std(arr)#Standard deviation of all elements in array
```

```
Standard deviation of all elements in array is:
```

Out[183… np.float64(3.1622776601683795)

In [184…
```
print('Exponential of all elements in array is: ')
np.exp(arr)#Exponential of all elements in array
```

```
Exponential of all elements in array is:
```

Out[184…
```
array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
       5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
       2.98095799e+03, 8.10308393e+03, 2.20264658e+04])
```

In [185…
```
print('Square of all elements in array is: ')
np.square(arr)#Square of all elements in array
```

```
Square of all elements in array is:
```

Out[185…
```
array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100])
```

In the numpy official universal functions documentation page, you can find a lot more functions on different mathematical operations.

Universal functions are very useful in machine learning and data science.

And that's it for this article.

Last Words

`Numpy` is the most `important` library in the world of `Machine Learning` and `Data Science`. If you have a keen interest in `Machine Learning` and `Data Science`, then you must have a very good understanding of `Numpy` arrays and multidimensional arrays.

Everything after this will be connected to `Numpy` one way or another.

And I hope I've helped you in some way to understand `Numpy` concepts. And we will learn more about `Numpy` in the upcoming articles.

I hope you liked this article and if you did, please leave a `like` and `share` this article with other `machine learning` enthusiasts. I have discord server where I talk about `weird tech` stuff. You can also join there to discuss `machine learning` and your weird ideas. (Link is given at the start of the article)

Happy Coding!!