ML_4

NUMPY

Muhammad Naqeeb | Machine Learning | August 8, 2021

Table of Contents

Numpy 3

Import 5

DataTypes & Attributes 5

- ndarray 5
- Anatomy of a nampy array 6
- .shape 6
- .ndim 7
- .dtype 7
- .size 7
- .type 8
- Creating a DataFrame from a Numpy array 8

Creating Arrays 9

- .ones 9
- .zeros() 9
- .arange() 9
- .random.randint() 10
- random.random() 10
- random.rand() 11

NumPy Random Seed 11

Viewing Arrays and Matrices 12

- .[index] 12
- Slicing with array index 14

Manipulating & comparing arrays 14

- Arithmetic 14
- ValueError: operands could not be broadcast together with shapes (2,3) (5,4) / General Broadcasting Rules 16
- Aggregation 17
- Mean , min ,max 17
- standard deviation / np.std() 18
- Variance / np.var() 19
  - ***What does the variance and the standard deviation tell us?*** *19*

How to derive the variance from the standard deviation? 20
Mention the use of variance in statistics. 20
Demo of std and var 20
Reshape and Transpose 22
Reshape 22
Transpose 23
Dot product 24
Comparison operators 26
Sorting Arrays 27
.sort() 27
.argsort 28
.argmin 28
.argmax 29
Turn an image into a Numpy array 30

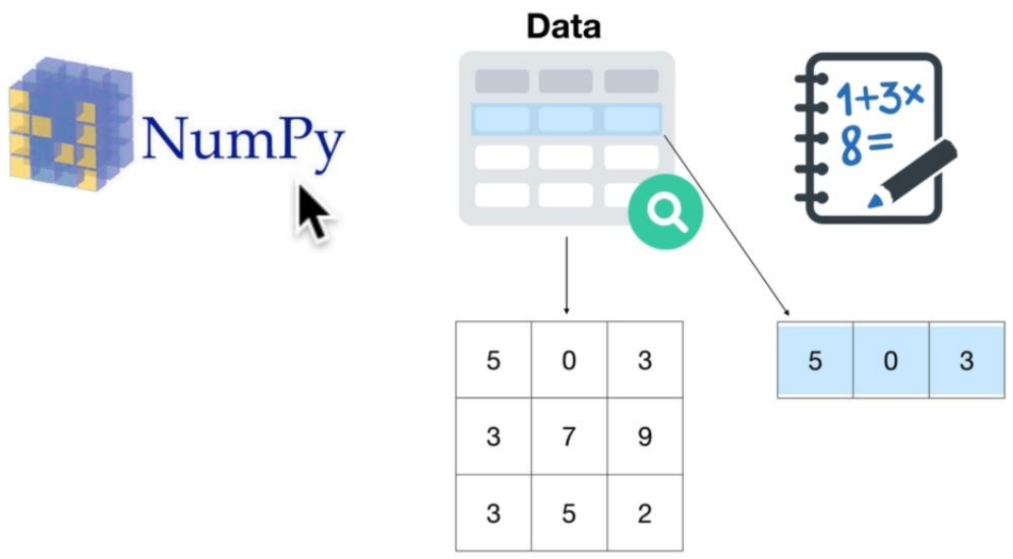
Numpy

Numpy, which is very similar to Python lists. It's one of the most used and popular libraries when it comes to data science and machine learning.

Numpy is written in C, a programming language that is really really fast.

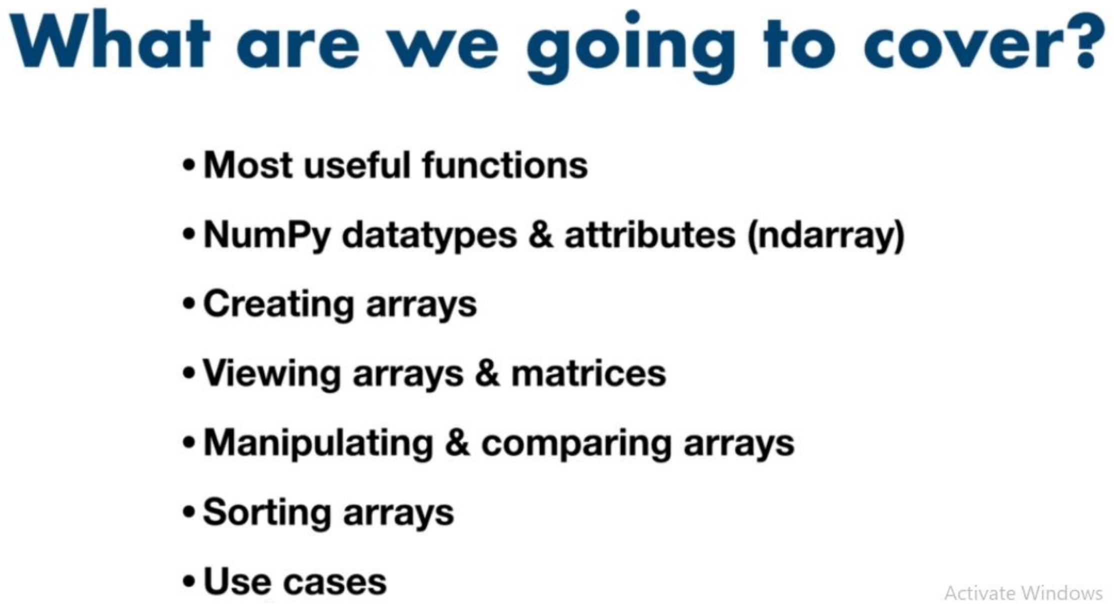
If we used Python lists it would actually be a lot slower to do some of the things that we're about to do so. Numpy operations on arrays and lists are a lot faster than performing the same operations in Python

What is NumPy?



Numpy is basically the backbone of all data science, machine learning and numerical computing in Python. And numpy sense for numerical python.

Numoy is going to form the foundation of turning your data into a series of numbers and then what a machine learning algorithm will do is work out the patterns in those numbers.



What are we going to cover?

- Most useful functions
- NumPy datatypes & attributes (ndarray)
- Creating arrays
- Viewing arrays & matrices
- Manipulating & comparing arrays
- Sorting arrays
- Use cases

Import

```
import numpy as np
```

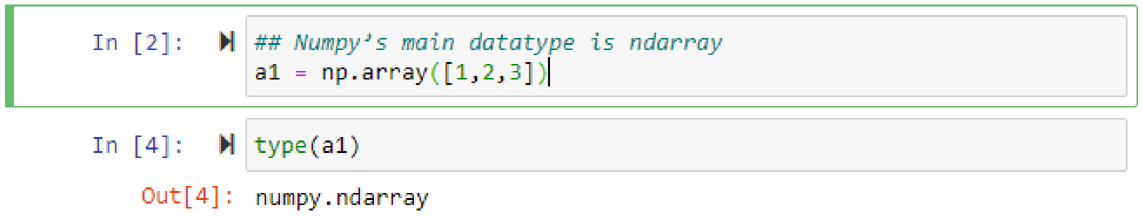
DataTypes & Attributes

ndarray

Numpy's main datatype is ndarray

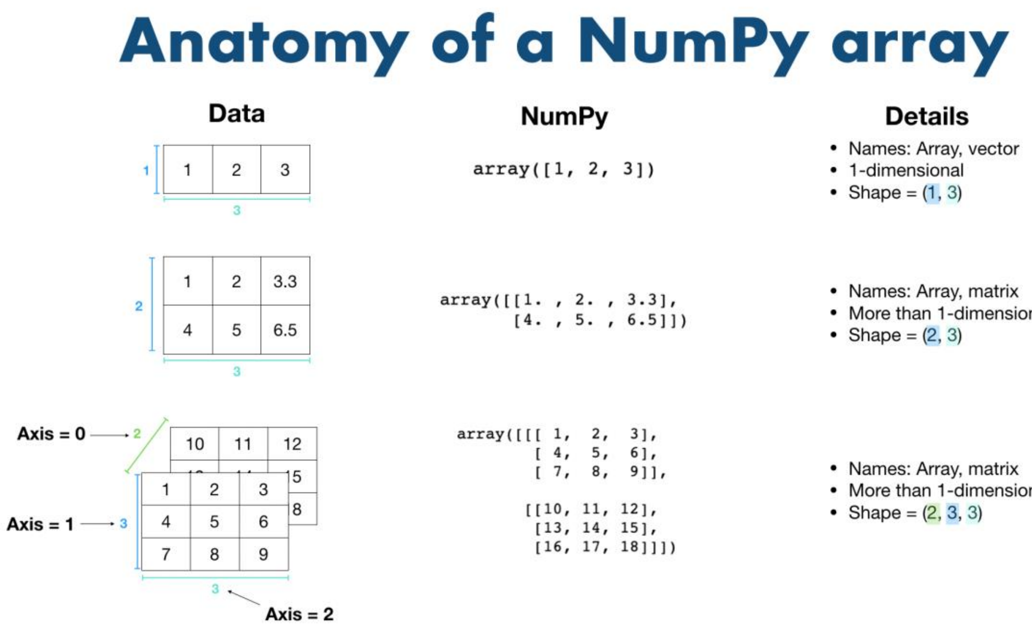
Numpy is likely an nd array and this stands for n dimensional array and an array can be almost any list of numbers you can imagine. An n dimensions means or the list of numbers could be almost any shape that you can imagine

```
a1 = np.array([1,2,3])
```



```
In [2]:  ## Numpy's main datatype is ndarray
         a1 = np.array([1,2,3])

In [4]:  type(a1)

Out[4]:  numpy.ndarray
```

Anatomy of a nampy array



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

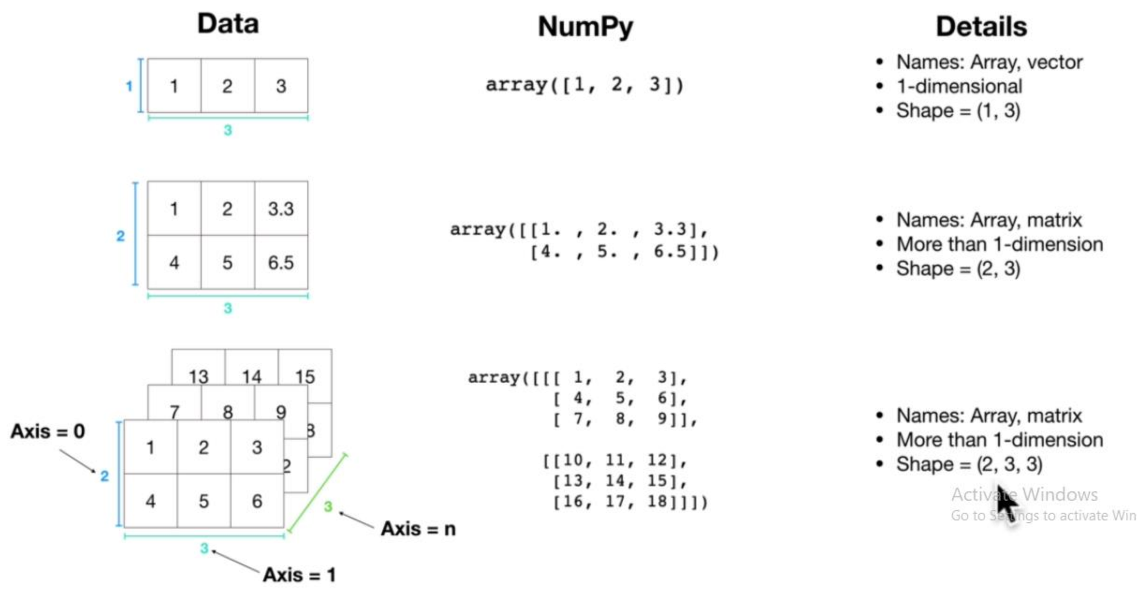
- Names: Array, vector
- 1-dimensional
- Shape = (1, 3)

```
array([[1. , 2. , 3.3],
       [4. , 5. , 6.5]])
```

- Names: Array, matrix
- More than 1-dimension
- Shape = (2, 3)

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

       [[10, 11, 12],
        [13, 14, 15],
        [16, 17, 18]]])
```

- Names: Array, matrix
- More than 1-dimension
- Shape = (2, 3, 3)



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

- Names: Array, vector
- 1-dimensional
- Shape = (1, 3)

```
array([[1. , 2. , 3.3],
       [4. , 5. , 6.5]])
```

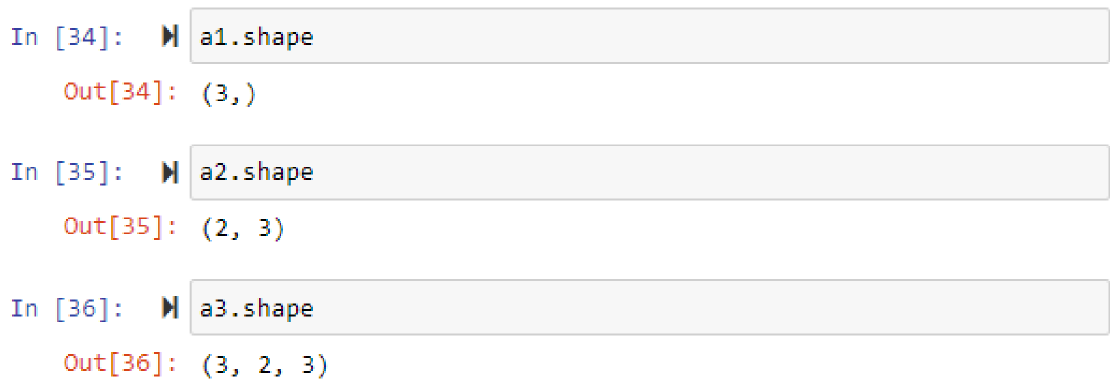
- Names: Array, matrix
- More than 1-dimension
- Shape = (2, 3)

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

       [[10, 11, 12],
        [13, 14, 15],
        [16, 17, 18]]])
```

- Names: Array, matrix
- More than 1-dimension
- Shape = (2, 3, 3)

.shape

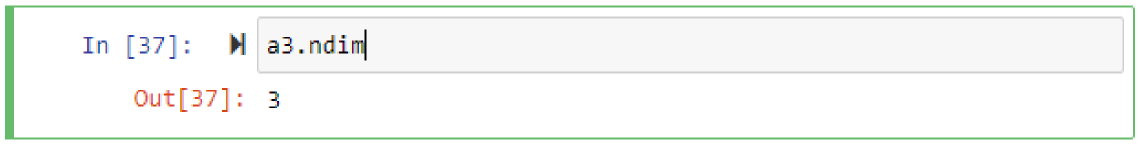
```
a1.shape
```



```
In [34]:  a1.shape
Out[34]: (3,)

In [35]:  a2.shape
Out[35]: (2, 3)

In [36]:  a3.shape
Out[36]: (3, 2, 3)
```

.ndim

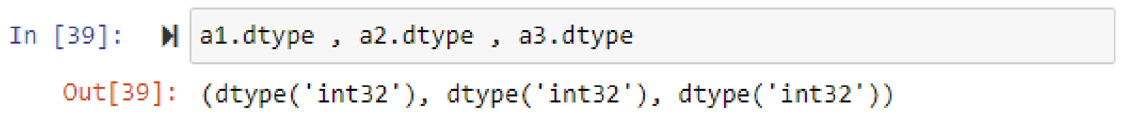
```
a3.ndim
```



```
In [37]:  a3.ndim
Out[37]: 3
```

.dtype

```
a1.dtype , a2.dtype , a3.dtype
```

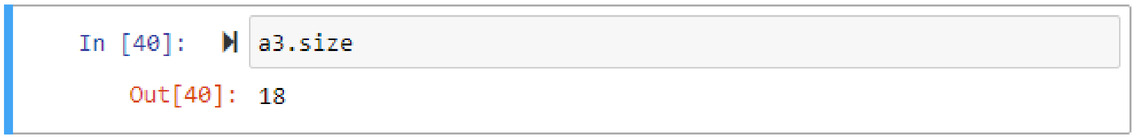


```
In [39]:  a1.dtype , a2.dtype , a3.dtype
Out[39]: (dtype('int32'), dtype('int32'), dtype('int32'))
```

.size

Size attribute tells us how many elements we've got total in our right now.

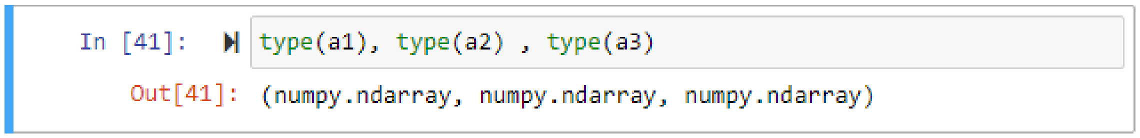
```
a3.size
```



```
In [40]:  a3.size
Out[40]:  18
```

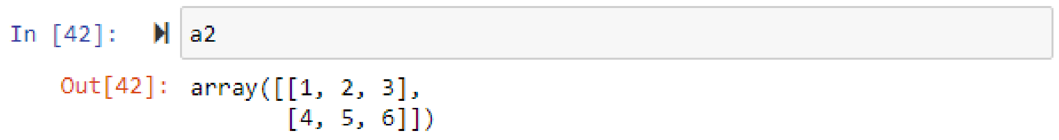
.type

```
type(a1), type(a2) , type(a3)
```

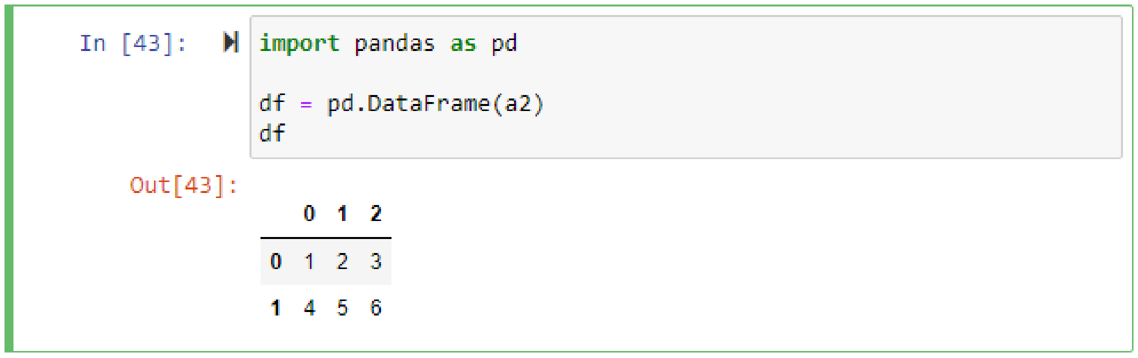


```
In [41]:  type(a1), type(a2) , type(a3)
Out[41]:  (numpy.ndarray, numpy.ndarray, numpy.ndarray)
```

Creating a DataFrame from a Numpy array



```
In [42]:  a2
Out[42]:  array([[1, 2, 3],
                 [4, 5, 6]])
```



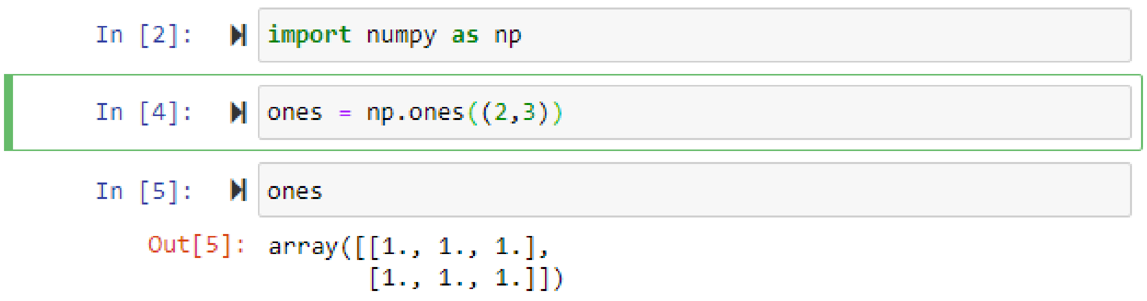
```
In [43]:  import pandas as pd

          df = pd.DataFrame(a2)
          df
Out[43]:
             0  1  2
          0  1  2  3
          1  4  5  6
```

Creating Arrays

.ones

Return a new array of given shape and type, filled with ones.

```
ones = np.ones((2,3))
```

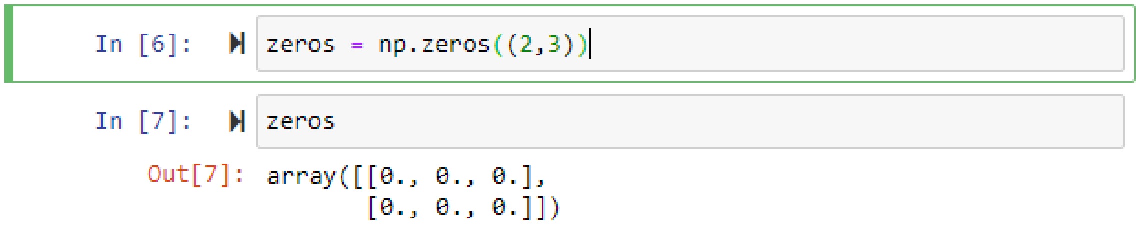


```
In [2]:  import numpy as np

In [4]:  ones = np.ones((2,3))

In [5]:  ones

Out[5]: array([[1., 1., 1.],
               [1., 1., 1.]])
```

.zeros()

Return a new array of given shape and type, filled with zero

```
zeros = np.zeros((2,3))
```



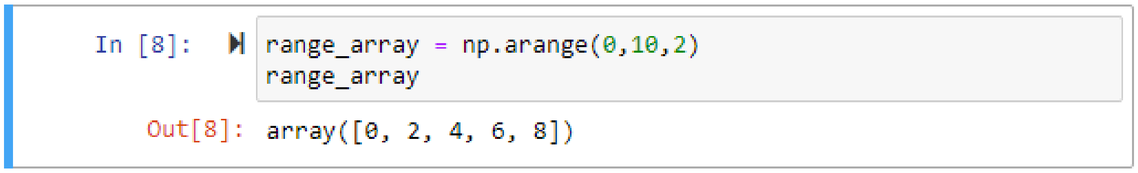
```
In [6]:  zeros = np.zeros((2,3))

In [7]:  zeros

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

.arange()

```
arange(start, stop, step, dtype=None, *, like=None)
```

Return evenly spaced values within a given interval.

```
range_array = np.arange(0,10,2)
```

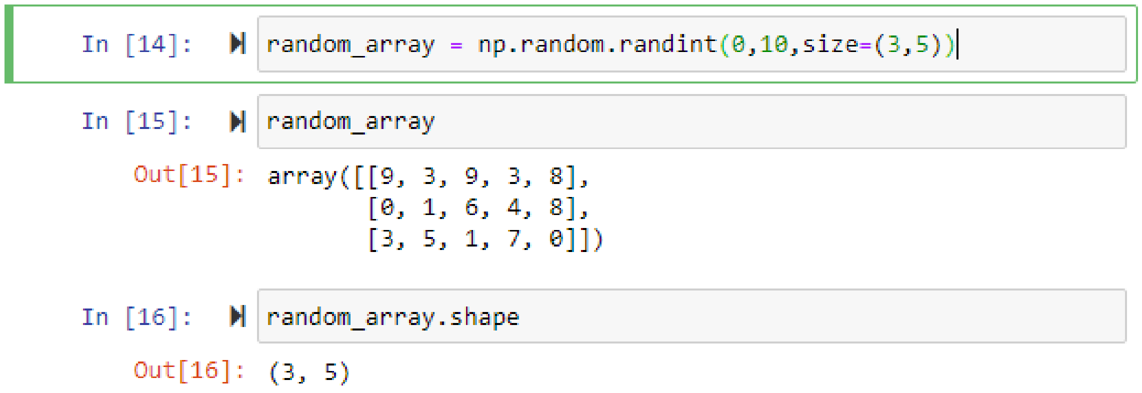


```
In [8]:  range_array = np.arange(0,10,2)
         range_array

Out[8]:  array([0, 2, 4, 6, 8])
```

.random.randint()

This will generate an array with random number

```
random_array = np.random.randint(0,10,size=(3,5))
```

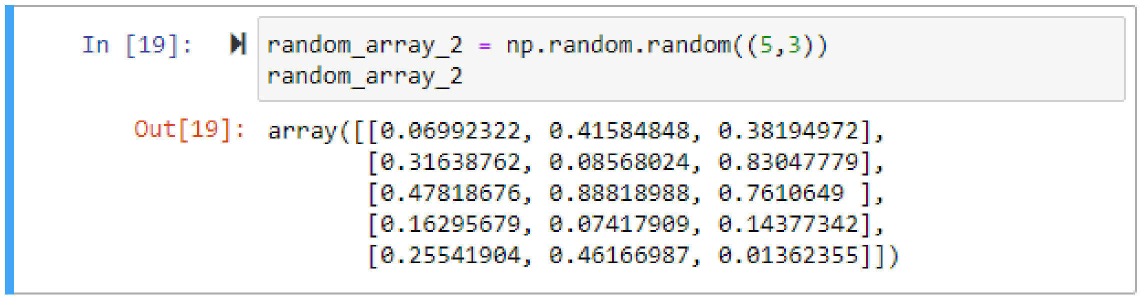


```
In [14]:  random_array = np.random.randint(0,10,size=(3,5))

In [15]:  random_array

Out[15]:  array([[9, 3, 9, 3, 8],
                 [0, 1, 6, 4, 8],
                 [3, 5, 1, 7, 0]])

In [16]:  random_array.shape

Out[16]:  (3, 5)
```

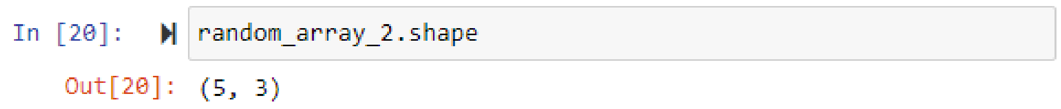
random.random()

Return random floats in the half-open interval [0.0, 1.0). and it take **size** as a parameter.

```
random_array_2 = np.random.random((5,3))
```



```
In [19]:  random_array_2 = np.random.random((5,3))
          random_array_2

Out[19]:  array([[0.06992322, 0.41584848, 0.38194972],
                 [0.31638762, 0.08568024, 0.83047779],
                 [0.47818676, 0.88818988, 0.7610649 ],
                 [0.16295679, 0.07417909, 0.14377342],
                 [0.25541904, 0.46166987, 0.01362355]])
```

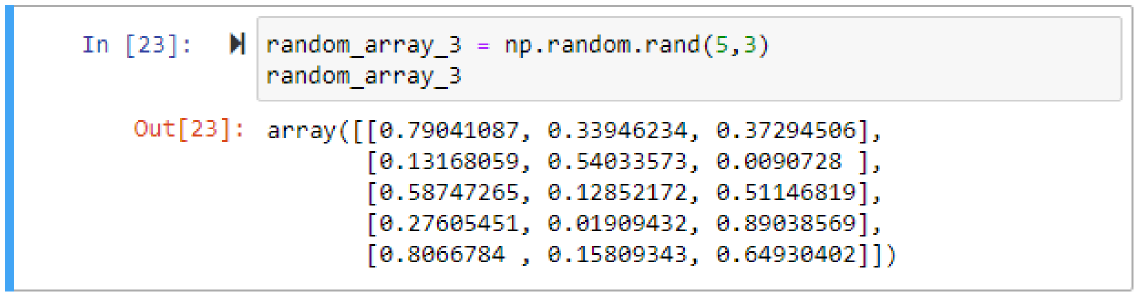


```
In [20]:  random_array_2.shape

Out[20]:  (5, 3)
```

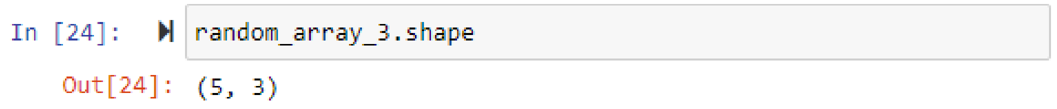
random.rand()

Random values in a given shape.

```
random_array_3 = np.random.rand(5,3)
```



```
In [23]:  random_array_3 = np.random.rand(5,3)
          random_array_3

Out[23]:  array([[0.79041087, 0.33946234, 0.37294506],
                 [0.13168059, 0.54033573, 0.0090728 ],
                 [0.58747265, 0.12852172, 0.51146819],
                 [0.27605451, 0.01909432, 0.89038569],
                 [0.8066784 , 0.15809343, 0.64930402]])
```



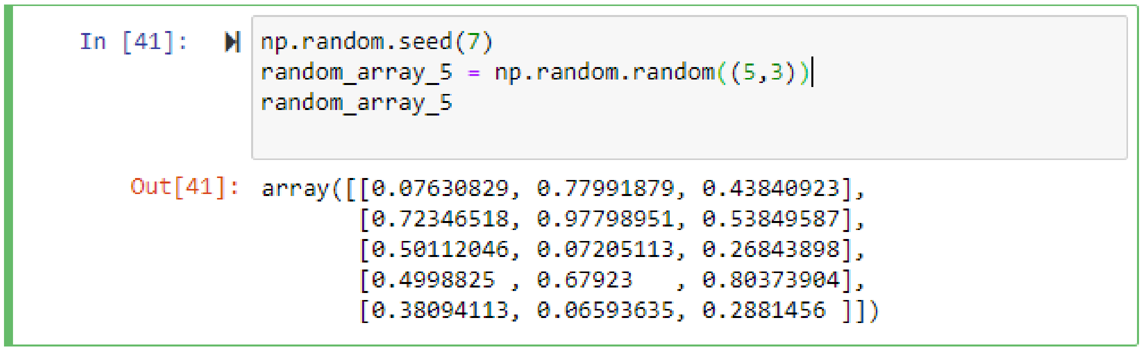
```
In [24]:  random_array_3.shape

Out[24]:  (5, 3)
```

NumPy Random Seed

- random() function is used to generate random numbers in Python. Not actually random, rather this is used to generate **pseudo-random numbers**. That implies that these randomly generated numbers can be determined.
- `random()` function generates numbers for some values. This value is also called *seed* value.
- If you use the same seed value twice you will get the same random number twice.

- The NumPy random seed function enables the coder to optimize codes very easily wherein random numbers can be used for testing the utility and efficiency.

```
np.random.seed(7)
random_array_5 = np.random.random((5,3))
```

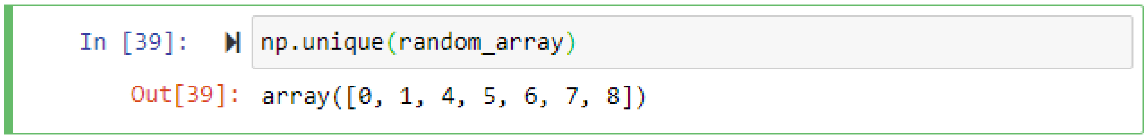


```
In [41]:  np.random.seed(7)
           random_array_5 = np.random.random((5,3))
           random_array_5

Out[41]:  array([[0.07630829, 0.77991879, 0.43840923],
                 [0.72346518, 0.97798951, 0.53849587],
                 [0.50112046, 0.07205113, 0.26843898],
                 [0.4998825 , 0.67923   , 0.80373904],
                 [0.38094113, 0.06593635, 0.2881456 ]])
```

Viewing Arrays and Matrices

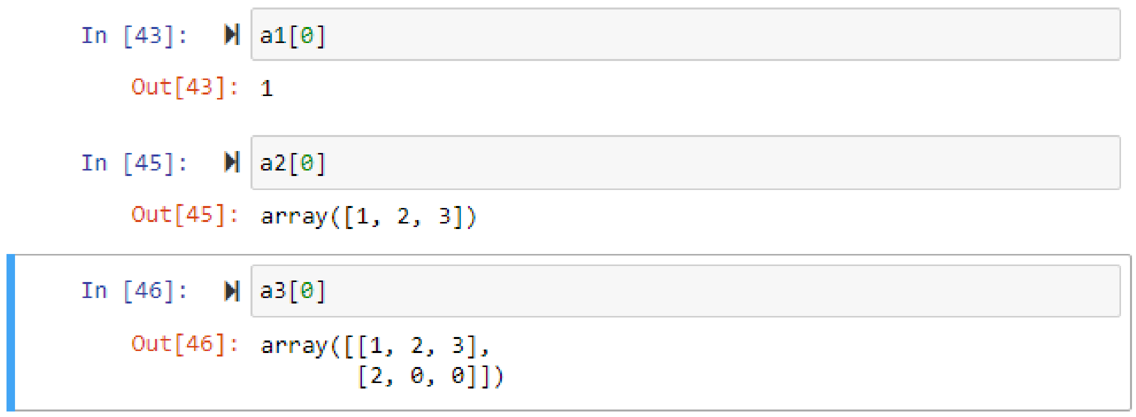
The numpy module of Python provides a function for finding unique elements in a numpy array. The numpy.unique() function finds the unique elements of an array and returns these unique elements as a sorted array.

```
np.unique(random_array)
```

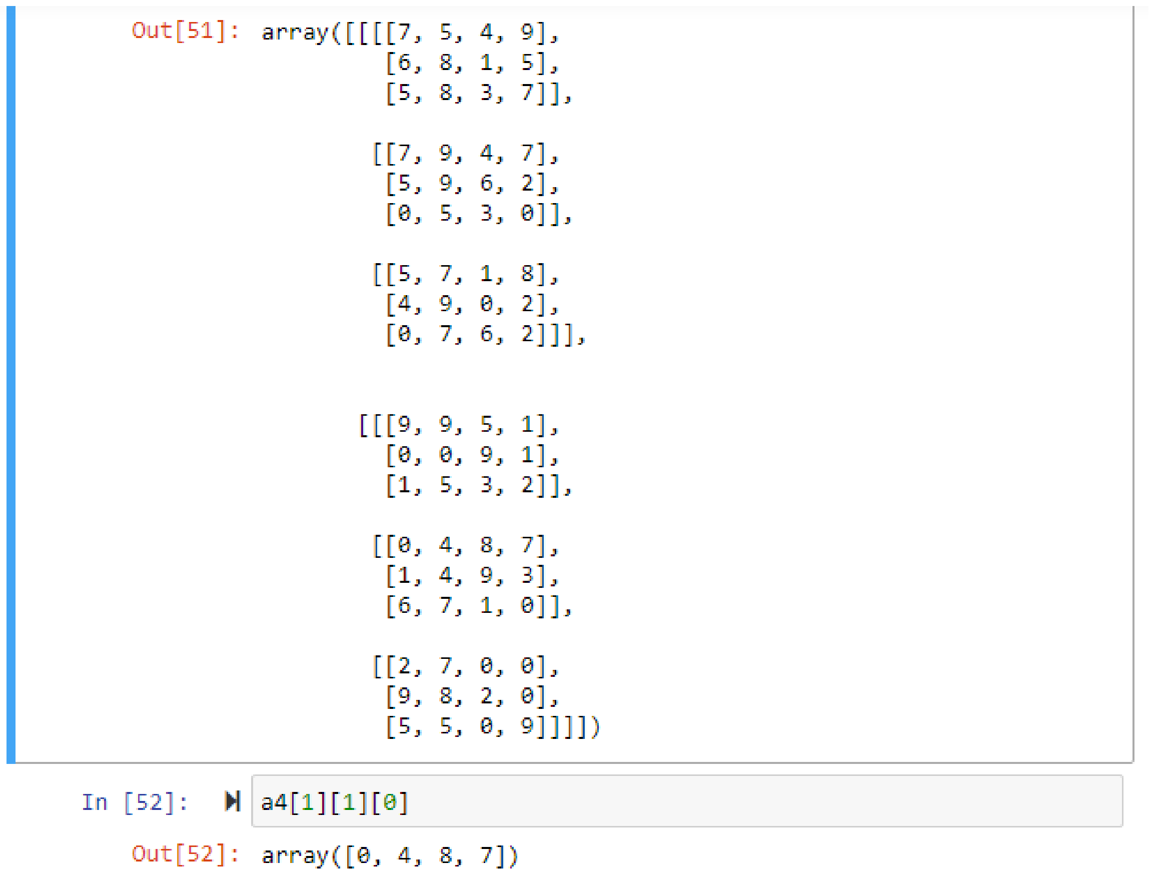


```
In [39]:  np.unique(random_array)

Out[39]:  array([0, 1, 4, 5, 6, 7, 8])
```

.[index]

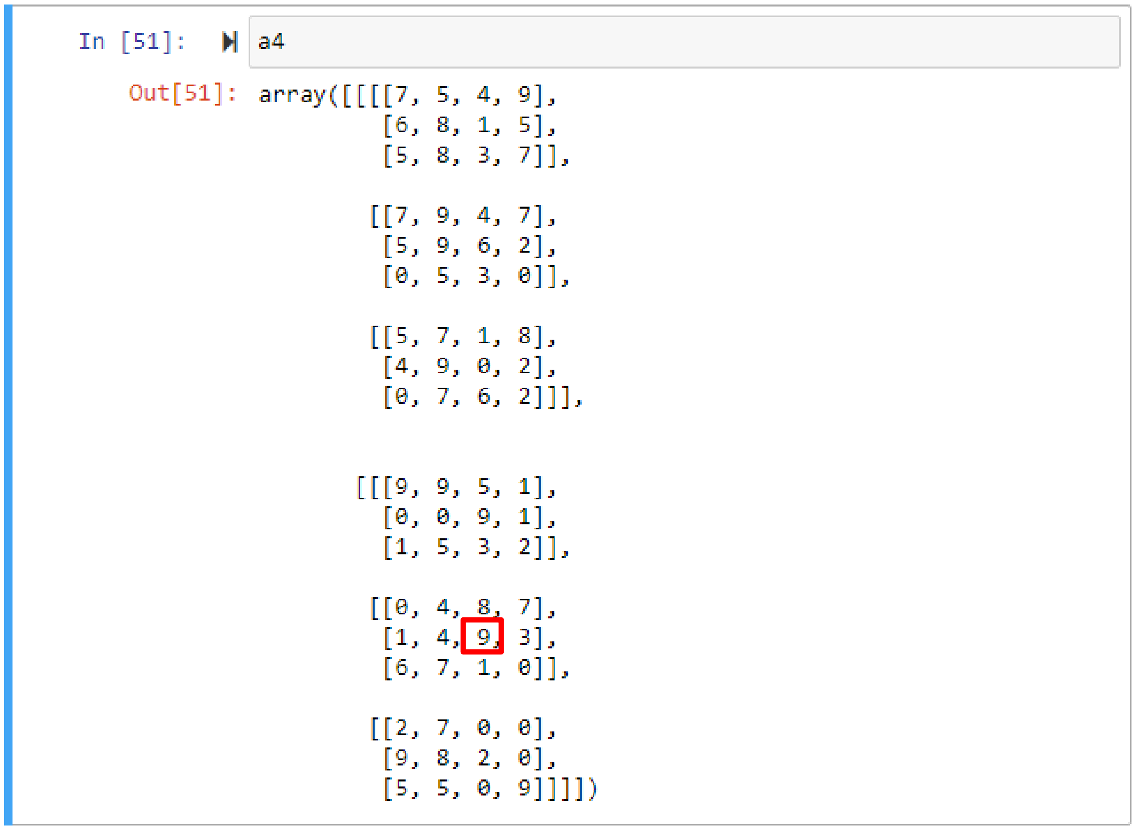


```
In [43]:  a1[0]

  Out[43]: 1

In [45]:  a2[0]

  Out[45]: array([1, 2, 3])

In [46]:  a3[0]

  Out[46]: array([[1, 2, 3],
                  [2, 0, 0]])
```

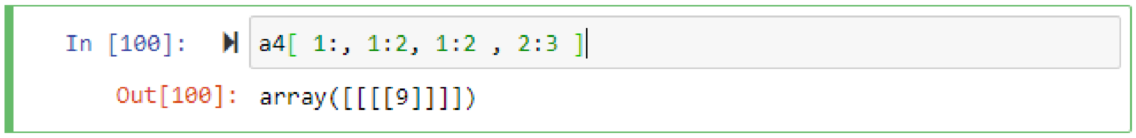


```
  Out[51]: array([[[[7, 5, 4, 9],
                     [6, 8, 1, 5],
                     [5, 8, 3, 7]],

                    [[7, 9, 4, 7],
                     [5, 9, 6, 2],
                     [0, 5, 3, 0]],

                    [[5, 7, 1, 8],
                     [4, 9, 0, 2],
                     [0, 7, 6, 2]]],


                   [[[9, 9, 5, 1],
                     [0, 0, 9, 1],
                     [1, 5, 3, 2]],

                    [[0, 4, 8, 7],
                     [1, 4, 9, 3],
                     [6, 7, 1, 0]],

                    [[2, 7, 0, 0],
                     [9, 8, 2, 0],
                     [5, 5, 0, 9]]]])

In [52]:  a4[1][1][0]

  Out[52]: array([0, 4, 8, 7])
```

Slicing with array index



```
In [51]:  a4

Out[51]: array([[[[7, 5, 4, 9],
                  [6, 8, 1, 5],
                  [5, 8, 3, 7]],

                 [[7, 9, 4, 7],
                  [5, 9, 6, 2],
                  [0, 5, 3, 0]],

                 [[5, 7, 1, 8],
                  [4, 9, 0, 2],
                  [0, 7, 6, 2]]],


                [[[9, 9, 5, 1],
                  [0, 0, 9, 1],
                  [1, 5, 3, 2]],

                 [[0, 4, 8, 7],
                  [1, 4, 9, 3],
                  [6, 7, 1, 0]],

                 [[2, 7, 0, 0],
                  [9, 8, 2, 0],
                  [5, 5, 0, 9]]]])
```



```
In [100]:  a4[ 1:, 1:2, 1:2 , 2:3 ]

Out[100]: array([[[[9]]]])
```

Manipulating & comparing arrays

Arithmetic

```
In [104]:  a1

Out[104]:  array([1, 2, 3])
```

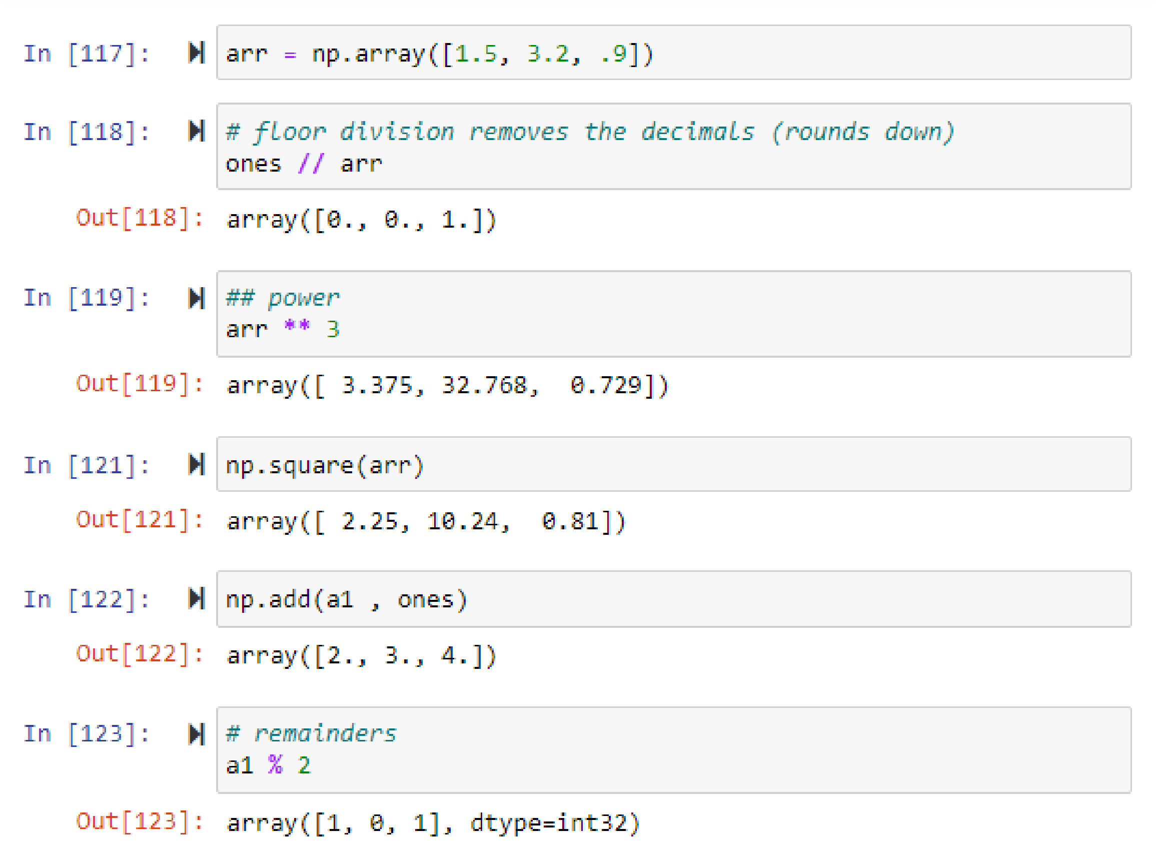
```
In [108]:  ones = np.ones(3)
           ones

Out[108]:  array([1., 1., 1.])
```

```
In [109]:  a1 + ones

Out[109]:  array([2., 3., 4.])
```

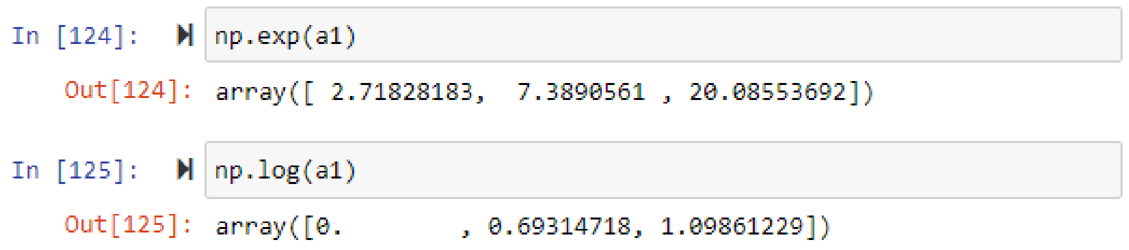
```
In [110]:  a1 - ones

Out[110]:  array([0., 1., 2.])
```

```
In [111]:   a1 * ones

Out[111]:  array([1., 2., 3.])
```

```
In [112]:  a1 / ones

Out[112]:  array([1., 2., 3.])
```



```
In [117]:  arr = np.array([1.5, 3.2, .9])

In [118]:  # floor division removes the decimals (rounds down)
           ones // arr

Out[118]:  array([0., 0., 1.])

In [119]:  ## power
           arr ** 3

Out[119]:  array([ 3.375, 32.768,  0.729])

In [121]:  np.square(arr)

Out[121]:  array([ 2.25, 10.24,  0.81])

In [122]:  np.add(a1 , ones)

Out[122]:  array([2., 3., 4.])

In [123]:  # remainders
           a1 % 2

Out[123]:  array([1, 0, 1], dtype=int32)
```



```
In [124]:  np.exp(a1)

Out[124]:  array([ 2.71828183,  7.3890561 , 20.08553692])

In [125]:  np.log(a1)

Out[125]:  array([0.        , 0.69314718, 1.09861229])
```

ValueError: operands could not be broadcast together with shapes (2,3) (5,4) / General Broadcasting Rules

When operating on two arrays, numpy compares their shapes-element wise.

It starts with the trailing dimensions and works that way forward. Two dimensions are compatible when :

- they are equal, or
- one of them is one.

Aggregation

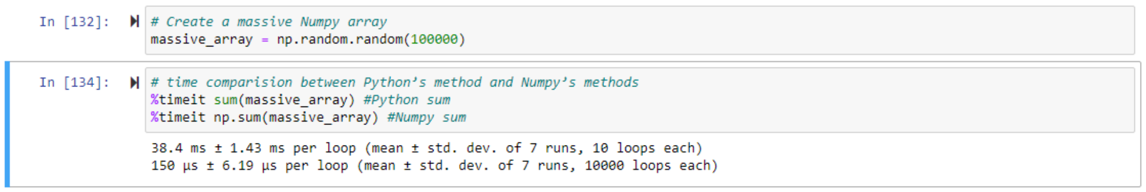
Performing the same operation on a number of things

So aggregation is performing the same operation on a number of things. And in our case the number of things is our numpy arrays.

Use Python's method (**sum()**) on python datatypes & use Numpy's methods on Numpy array (**np.sum()**)

```
# Create a massive Numpy array
massive_array = np.random.random(100000)
```

```
# time comparision between Python's method and Numpy's methods
%timeit sum(massive_array) #Python sum
%timeit np.sum(massive_array) #Numpy sum
```



```
In [132]: # Create a massive Numpy array
          massive_array = np.random.random(100000)

In [134]: # time comparision between Python's method and Numpy's methods
          %timeit sum(massive_array) #Python sum
          %timeit np.sum(massive_array) #Numpy sum

          38.4 ms ± 1.43 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
          150 µs ± 6.19 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

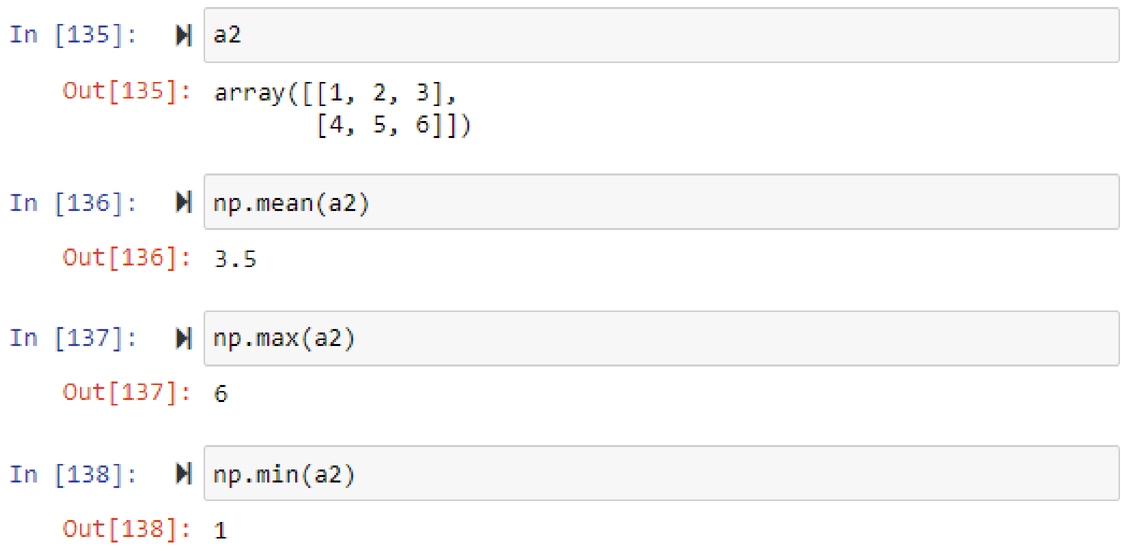
Mean , min ,max

Arithmetic mean is the sum of elements along an axis divided by the number of elements. The numpy.mean() function returns the arithmetic mean of elements in the array. If the axis is mentioned, it is calculated along it.

```
np.mean(a2)
```

```
np.max(a2)
```

```
np.min(a2)
```



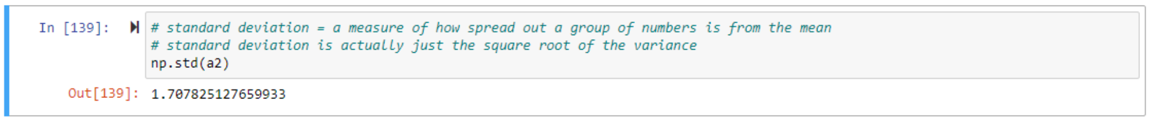
```
In [135]:  a2
Out[135]:  array([[1, 2, 3],
                  [4, 5, 6]])

In [136]:  np.mean(a2)
Out[136]:  3.5

In [137]:  np.max(a2)
Out[137]:  6

In [138]:  np.min(a2)
Out[138]:  1
```

standard deviation / np.std()

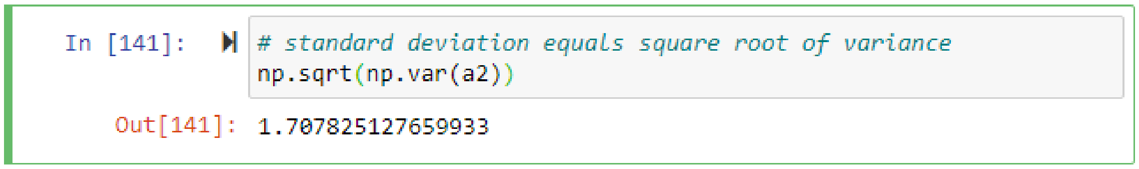
- A measure of how spread out a group of numbers is from the mean
- Standard deviation is actually just the square root of the variance
- **numpy.std(arr, axis = None) :** Compute the standard deviation of the given data (array elements) along the specified axis(if any)..
- **Standard Deviation** (**SD**) is measured as the spread of data distribution in the given data set.
- The Standard Deviation is a measure of how spread out numbers are.
- Its symbol is **σ** (the greek letter sigma)

$$\textit{Standard Deviation} = \sqrt{\text{mean}\,(\text{abs}(x - x.\text{mean}())^2)}$$

```
np.std(a2)
```



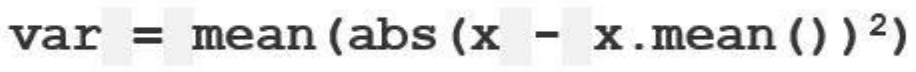
```
In [139]: # standard deviation = a measure of how spread out a group of numbers is from the mean
          # standard deviation is actually just the square root of the variance
          np.std(a2)

Out[139]: 1.707825127659933
```



```
In [141]: # standard deviation equals square root of variance
          np.sqrt(np.var(a2))

Out[141]: 1.707825127659933
```

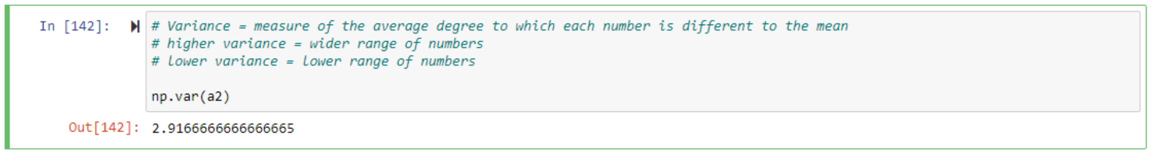
Variance / np.var()

- **Variance** = measure of the average degree to which each number is different to the mean
- **higher variance** = wider range of numbers
- **lower variance** = lower range of numbers
- **numpy.var(arr, axis = None)** : Compute the variance of the given data (array elements) along the specified axis(if any).



```
var = mean(abs(x - x.mean())²)
```

```
np.var(a2)
```



```
In [142]: # Variance = measure of the average degree to which each number is different to the mean
          # higher variance = wider range of numbers
          # lower variance = lower range of numbers

          np.var(a2)

Out[142]: 2.9166666666666665
```

What does the variance and the standard deviation tell us?

In probability theory and statistics, both the variance and standard deviation tell us how far the data values are spread out/dispersed from the mean of the given data set

How to derive the variance from the standard deviation?

The variance can be easily derived from the standard deviation by taking the square of the standard deviation.

Mention the use of variance in statistics.

In statistics, the variance is used to determine the measure of dispersion and the uncertainty in the given data set values.

Demo of std and var

```
# Demo of std and var
high_var_array = np.array([1, 100, 200, 300, 4000, 5000])
low_var_array = np.array([2, 4, 6, 8, 10])
```

```
np.var(high_var_array) , np.var(low_var_array)
```

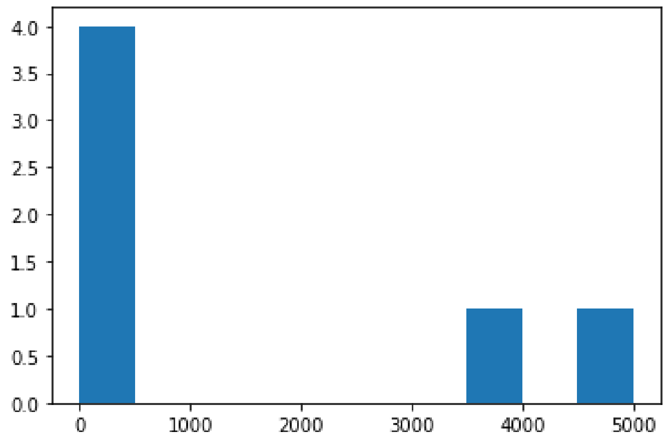
```
np.std(high_var_array) , np.std(low_var_array)
```

```
np.mean(high_var_array) , np.mean(low_var_array)
```

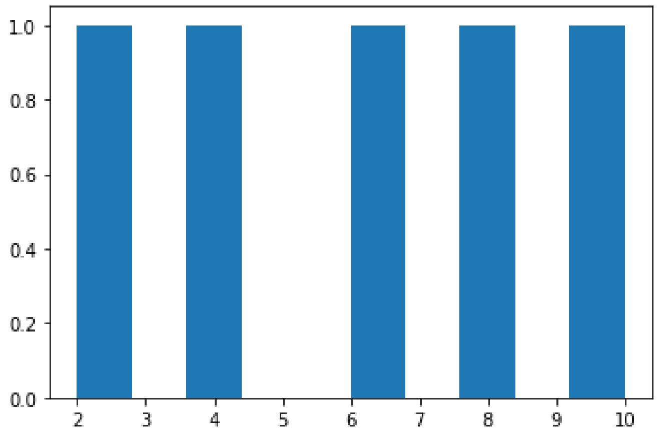
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.hist(high_var_array)
plt.show()
```

```
plt.hist(low_var_array)
plt.show()
```

```
In [143]:  # Demo of std and var
           high_var_array = np.array([1, 100, 200, 300, 4000, 5000])
           low_var_array = np.array([2, 4, 6, 8, 10])

In [144]:  np.var(high_var_array) , np.var(low_var_array)

Out[144]:  (4296133.472222221, 8.0)

In [145]:  np.std(high_var_array) , np.std(low_var_array)

Out[145]:  (2072.711623024829, 2.8284271247461903)

In [146]:  np.mean(high_var_array) , np.mean(low_var_array)

Out[146]:  (1600.1666666666667, 6.0)

In [148]:  %matplotlib inline
           import matplotlib.pyplot as plt
           plt.hist(high_var_array)
           plt.show()
```



```
In [149]:  plt.hist(low_var_array)
           plt.show()
```



Reshape and Transpose

Reshape

a.reshape(shape, order='C')

Returns an array containing the same data with a new shape.

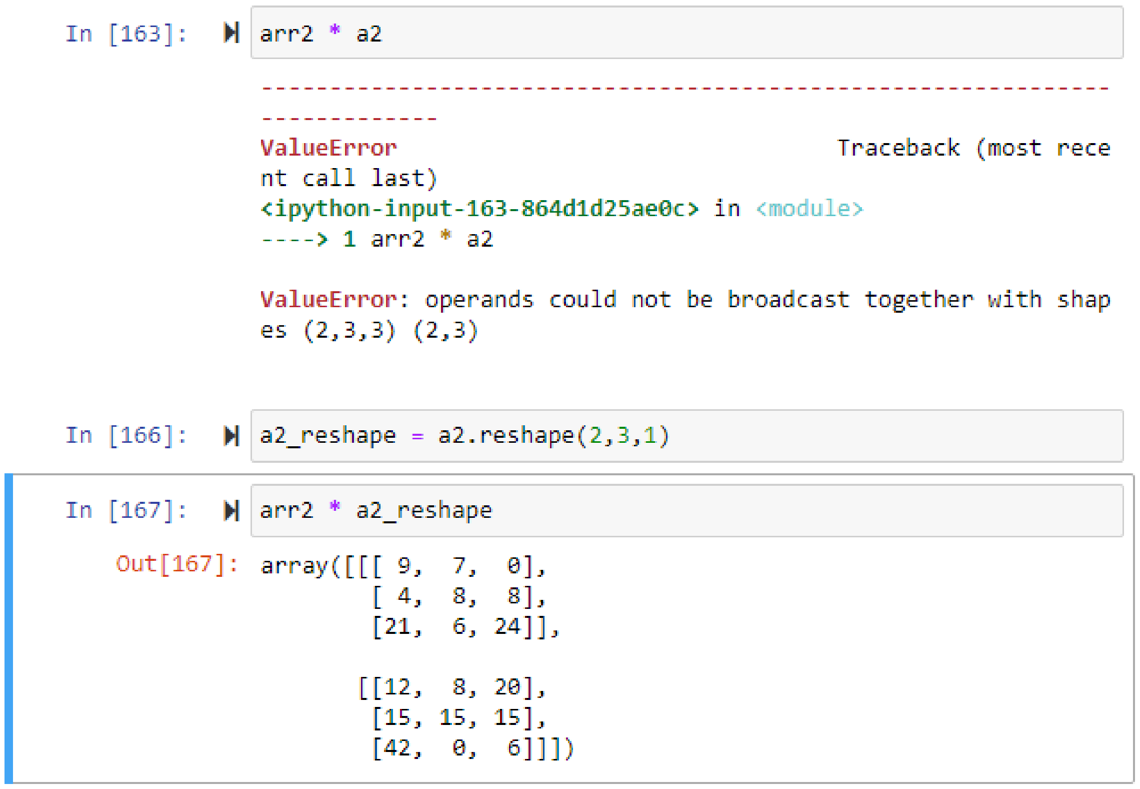
Solving this problem using **reshape.**

```
In [163]:  arr2 * a2

---------------------------------------------------------------
-------------
ValueError                                Traceback (most rece
nt call last)
<ipython-input-163-864d1d25ae0c> in <module>
----> 1 arr2 * a2

ValueError: operands could not be broadcast together with shap
es (2,3,3) (2,3)
```

We have to reshape a2 array from it shape (2,3), using rule of broadcasting

```
a2_reshape = a2.reshape(2,3,1)
```

```
arr2 * a2_reshape
```



```
In [163]:  arr2 * a2
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-163-864d1d25ae0c> in <module>
----> 1 arr2 * a2

ValueError: operands could not be broadcast together with shapes (2,3,3) (2,3)

In [166]:  a2_reshape = a2.reshape(2,3,1)

In [167]:  arr2 * a2_reshape

Out[167]: array([[[ 9,  7,  0],
                  [ 4,  8,  8],
                  [21,  6, 24]],

                 [[12,  8, 20],
                  [15, 15, 15],
                  [42,  0,  6]]])
```

So that is the benefit of reshape.

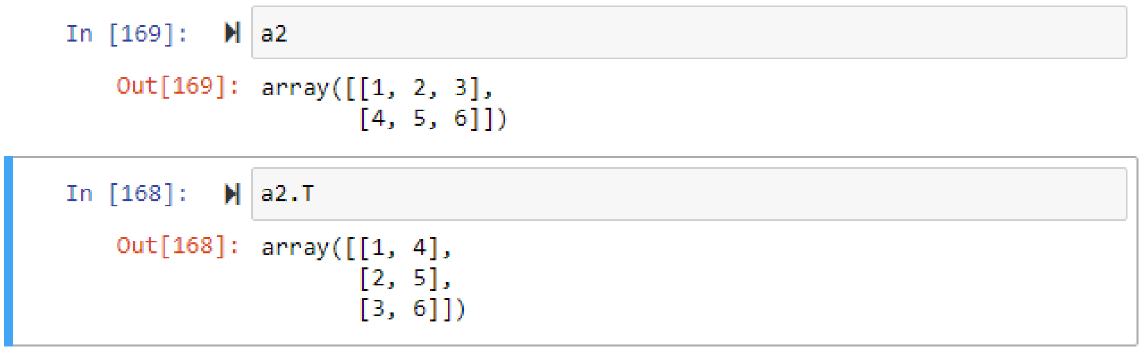
Now this is only a small example here. Before they were incompatible but because we reshaped it we reshaped **a2** to be within the broadcasting rules of numpy.

We could then use the multiplier function across these two arrays. that just because your numpy array comes in a certain shape like it begins like this doesn't mean it has to stay in that shape so this reshaped version of **a2** still contains the same information as the original **a2**. It's just in a different shape.

Transpose

- Transpose switches the axis
- In simple words it change rows into columns and columns into rows

```
a2.T
```



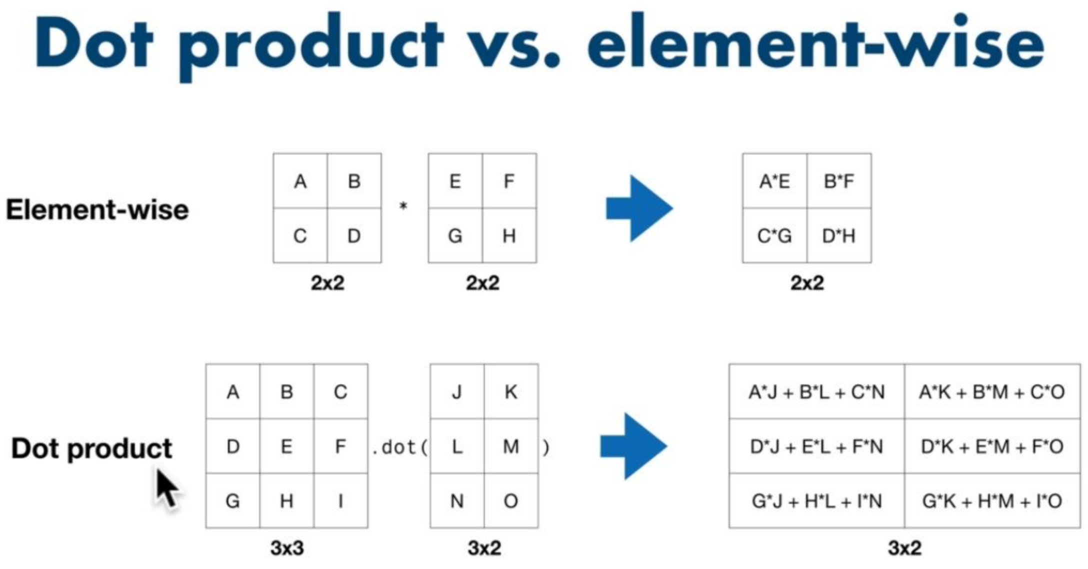
```
In [169]: a2
Out[169]: array([[1, 2, 3],
                 [4, 5, 6]])

In [168]: a2.T
Out[168]: array([[1, 4],
                 [2, 5],
                 [3, 6]])
```

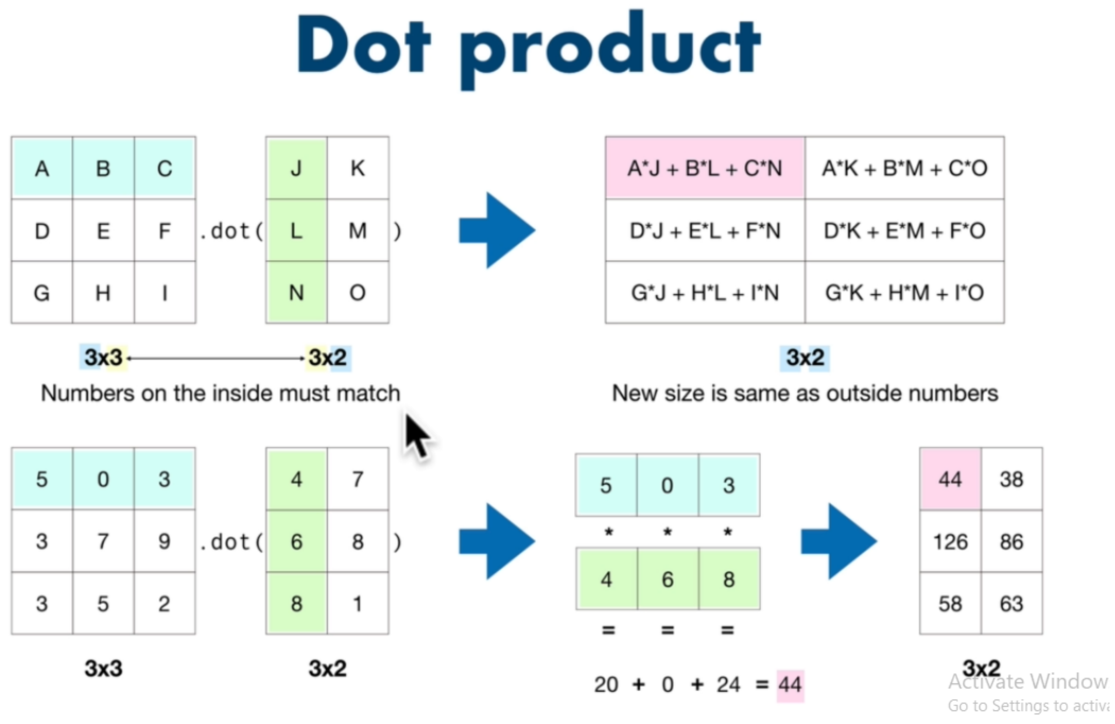
The important difference here between transpose and reshape is that **transpose** just flips the accesses around and in **reshape** you can create your own custom shapes.

Dot product

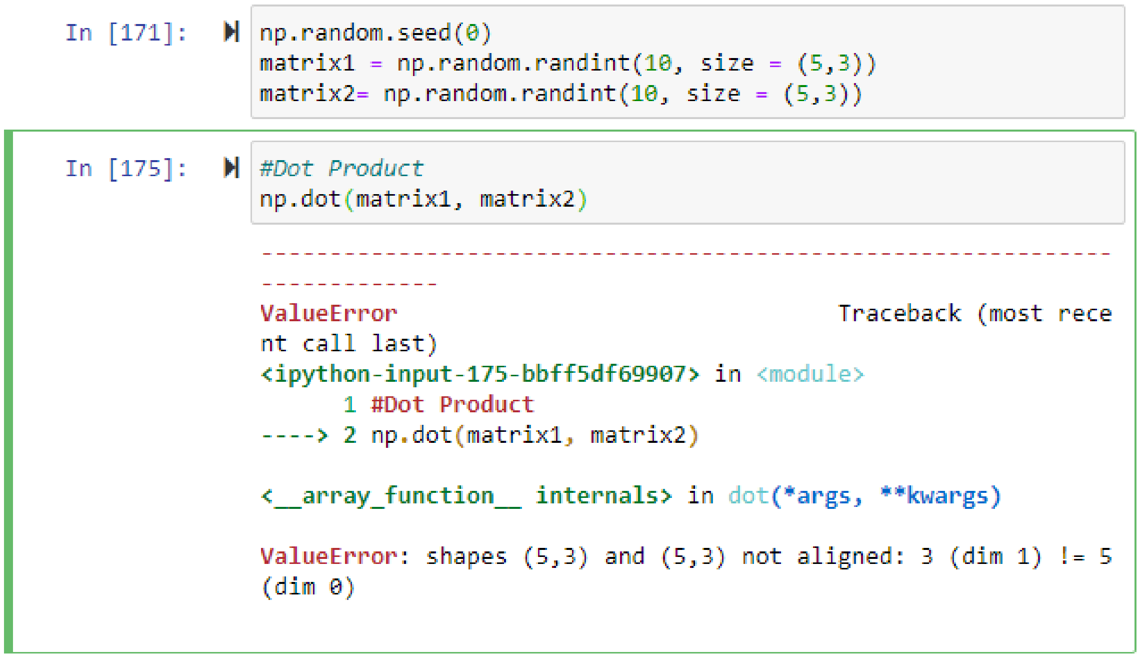
One of the main use cases for transposing a matrix is the dot product.

```
#Dot Product
np.dot(matrix1, matrix2)
```



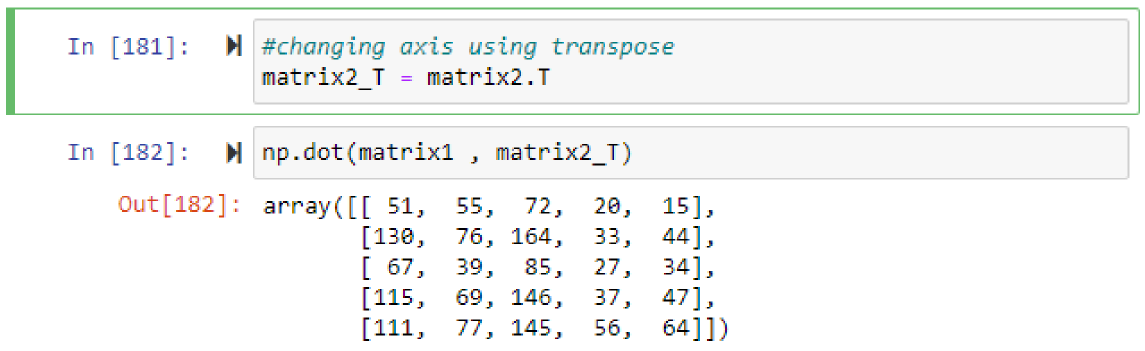

Dot product
A B C D E F G H I .dot(J K L M N O)
A*J + B*L + C*N | A*K + B*M + C*O
D*J + E*L + F*N | D*K + E*M + F*O
G*J + H*L + I*N | G*K + H*M + I*O
3x3 → 3x2
Numbers on the inside must match
3x2
New size is same as outside numbers
5 0 3 / 3 7 9 / 3 5 2 .dot(4 7 / 6 8 / 8 1)
3x3
3x2
5 0 3 * * * 4 6 8 = = =
20 + 0 + 24 = 44
44 38 / 126 86 / 58 63
3x2
Activate Window
Go to Settings to activa

Solving error using transpose and then using dot product.



```
In [171]:  np.random.seed(0)
           matrix1 = np.random.randint(10, size = (5,3))
           matrix2= np.random.randint(10, size = (5,3))
```

```
In [175]:  #Dot Product
           np.dot(matrix1, matrix2)

           ---------------------------------------------------------------
           -------------
           ValueError                                Traceback (most rece
           nt call last)
           <ipython-input-175-bbff5df69907> in <module>
                 1 #Dot Product
           ----> 2 np.dot(matrix1, matrix2)

           <__array_function__ internals> in dot(*args, **kwargs)

           ValueError: shapes (5,3) and (5,3) not aligned: 3 (dim 1) != 5
           (dim 0)
```



```
In [181]:  #changing axis using transpose
            matrix2_T = matrix2.T
```

```
In [182]:  np.dot(matrix1 , matrix2_T)

Out[182]: array([[ 51,  55,  72,  20,  15],
                 [130,  76, 164,  33,  44],
                 [ 67,  39,  85,  27,  34],
                 [115,  69, 146,  37,  47],
                 [111,  77, 145,  56,  64]])
```

Comparison operators

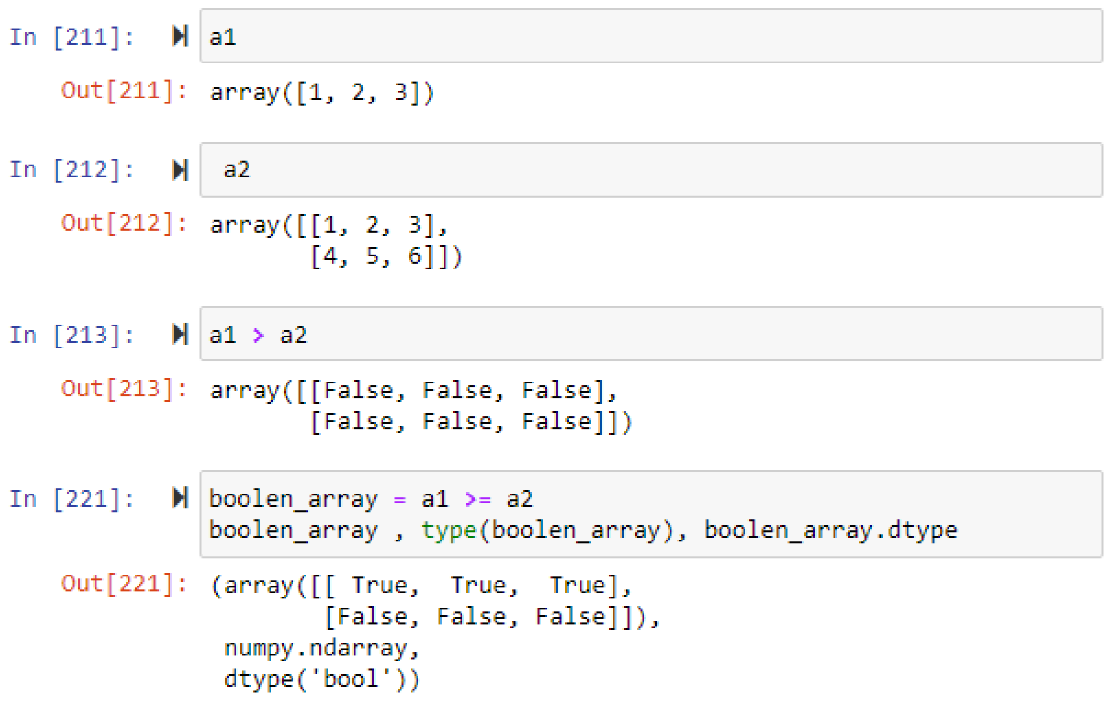
```
a1 > a2
```

```
boolen_array = a1 >= a2
boolen_array , type(boolen_array), boolen_array.dtype
```

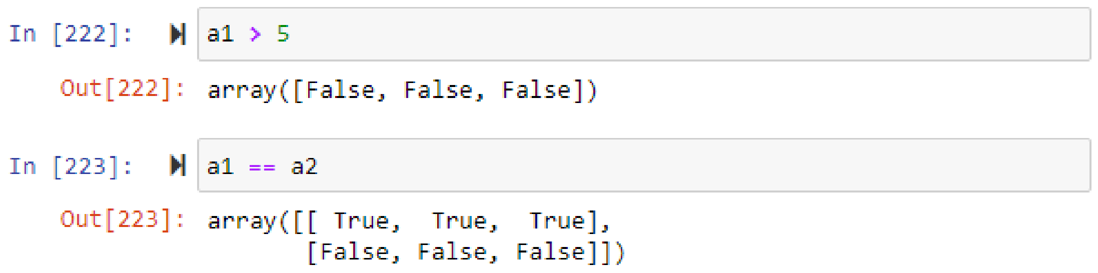
```
a1 > 5
```

```
a1 > 5
```

```
a1 == a2
```



```
In [211]:  a1

Out[211]: array([1, 2, 3])

In [212]:   a2

Out[212]: array([[1, 2, 3],
                 [4, 5, 6]])

In [213]:  a1 > a2

Out[213]: array([[False, False, False],
                 [False, False, False]])

In [221]:  boolen_array = a1 >= a2
           boolen_array , type(boolen_array), boolen_array.dtype

Out[221]: (array([[ True,  True,  True],
                  [False, False, False]]),
           numpy.ndarray,
           dtype('bool'))
```

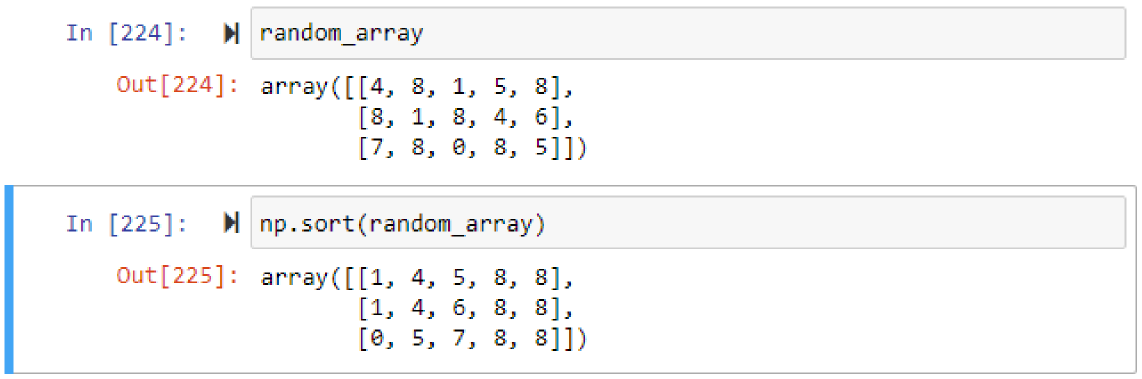


```
In [222]:  a1 > 5

Out[222]: array([False, False, False])

In [223]:  a1 == a2

Out[223]: array([[ True,  True,  True],
                 [False, False, False]])
```

Sorting Arrays

.sort()

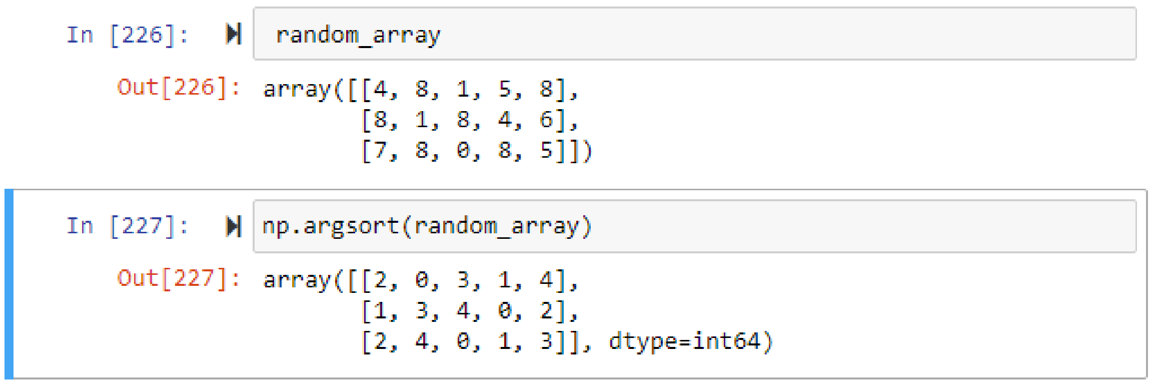
```
np.sort(random_array)
```



```
In [224]:  random_array

Out[224]: array([[4, 8, 1, 5, 8],
                 [8, 1, 8, 4, 6],
                 [7, 8, 0, 8, 5]])

In [225]:  np.sort(random_array)

Out[225]: array([[1, 4, 5, 8, 8],
                 [1, 4, 6, 8, 8],
                 [0, 5, 7, 8, 8]])
```

.argsort

- Returns the indices that would sort an array.
- Sort the array and return the index

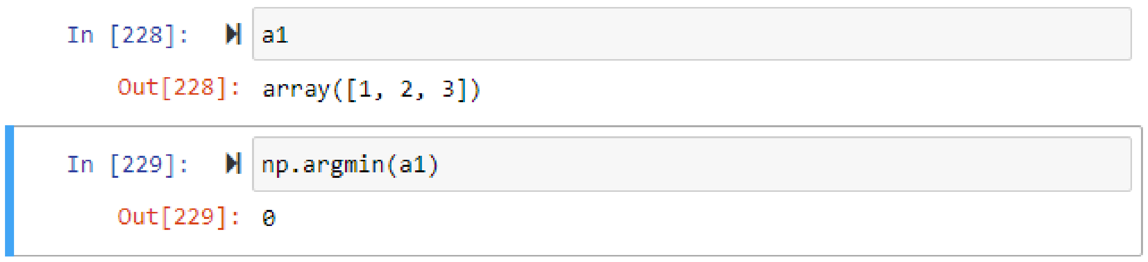


```
In [226]:  random_array

Out[226]: array([[4, 8, 1, 5, 8],
                 [8, 1, 8, 4, 6],
                 [7, 8, 0, 8, 5]])

In [227]:  np.argsort(random_array)

Out[227]: array([[2, 0, 3, 1, 4],
                 [1, 3, 4, 0, 2],
                 [2, 4, 0, 1, 3]], dtype=int64)
```

.argmin

- Returns the indices of the minimum values along an axis.
- Return index which have minimum value

```
np.argmin(a1)
```

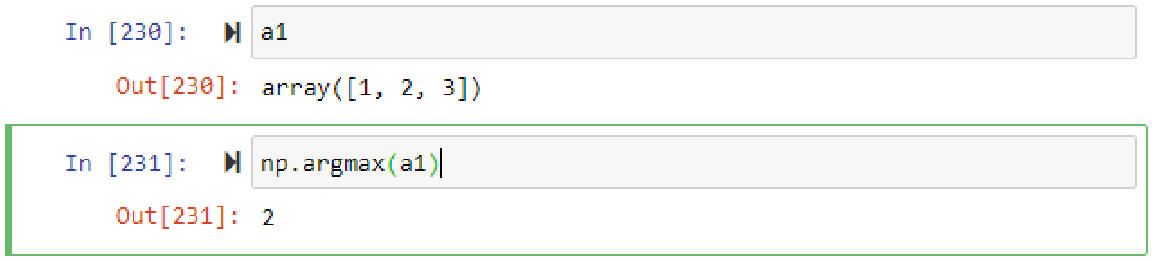


```
In [228]:  a1
Out[228]:  array([1, 2, 3])

In [229]:  np.argmin(a1)
Out[229]:  0
```

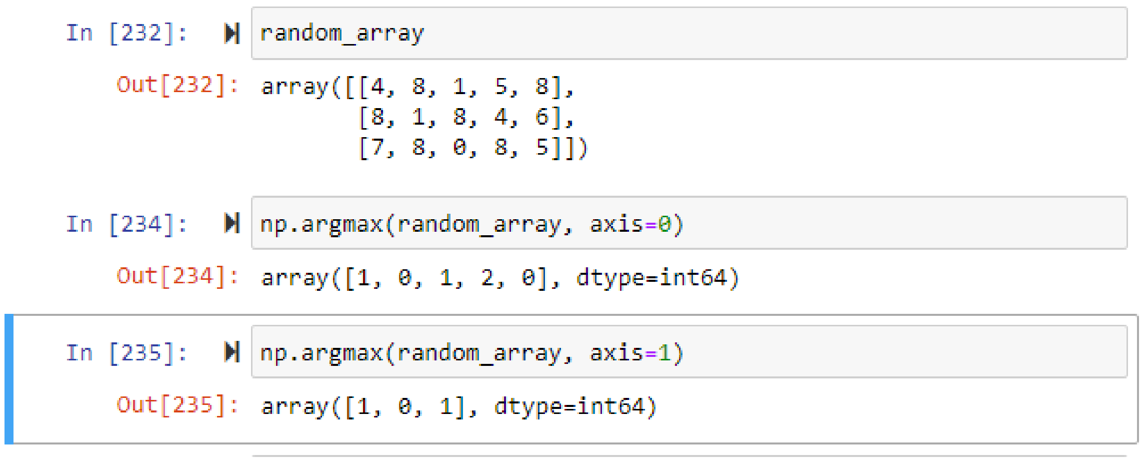
.argmax

Returns the indices of the maximum values along an axis.

```
np.argmax(a1)
```



```
In [230]:  a1
Out[230]:  array([1, 2, 3])

In [231]:  np.argmax(a1)
Out[231]:  2
```

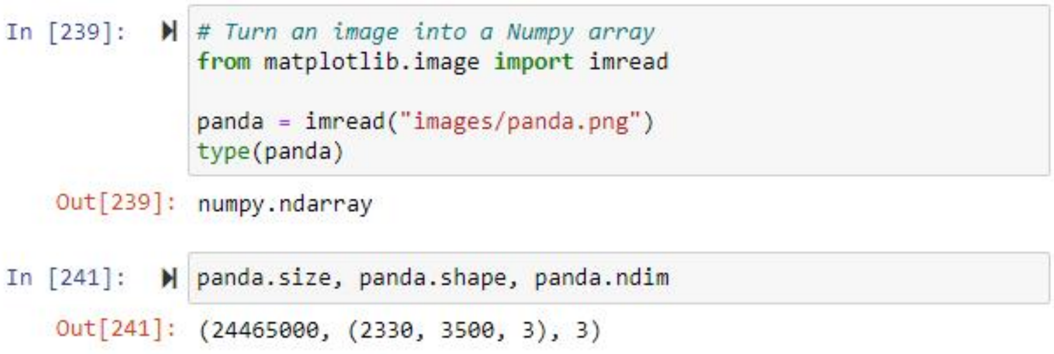


```
In [232]:  random_array
Out[232]:  array([[4, 8, 1, 5, 8],
                  [8, 1, 8, 4, 6],
                  [7, 8, 0, 8, 5]])

In [234]:  np.argmax(random_array, axis=0)
Out[234]:  array([1, 0, 1, 2, 0], dtype=int64)

In [235]:  np.argmax(random_array, axis=1)
Out[235]:  array([1, 0, 1], dtype=int64)
```

Turn an image into a Numpy array

```
# Turn an image into a Numpy array
from matplotlib.image import imread

panda = imread("images/panda.png")
type(panda)
```

```
panda.size, panda.shape, panda.ndim
```



```
In [239]:  # Turn an image into a Numpy array
           from matplotlib.image import imread

           panda = imread("images/panda.png")
           type(panda)
Out[239]:  numpy.ndarray

In [241]:  panda.size, panda.shape, panda.ndim
Out[241]:  (24465000, (2330, 3500, 3), 3)
```