

[Open in Colab](#)[View source code](#) | [Read notebook in online book format](#)

A Quick Introduction to Numerical Data Manipulation with Python and NumPy

```
In [1]: import datetime
print(f"Last updated: {datetime.datetime.now()}")
```



Last updated: 2024-04-26 12:33:29.134290

What is NumPy?

NumPy stands for numerical Python. It's the backbone of all kinds of scientific and numerical computing in Python.

And since machine learning is all about turning data into numbers and then figuring out the patterns, NumPy often comes into play.

Table of contents

What is NumPy?

Why NumPy?

What does this notebook cover?

Where can I get help?

0. Importing NumPy

1. DataTypes and attributes

Anatomy of an array

pandas DataFrame out of
NumPy arrays

2. Creating arrays

What unique values are in the
array a3?

3. Viewing arrays and matrices
(indexing)

4. Manipulating and comparing
arrays

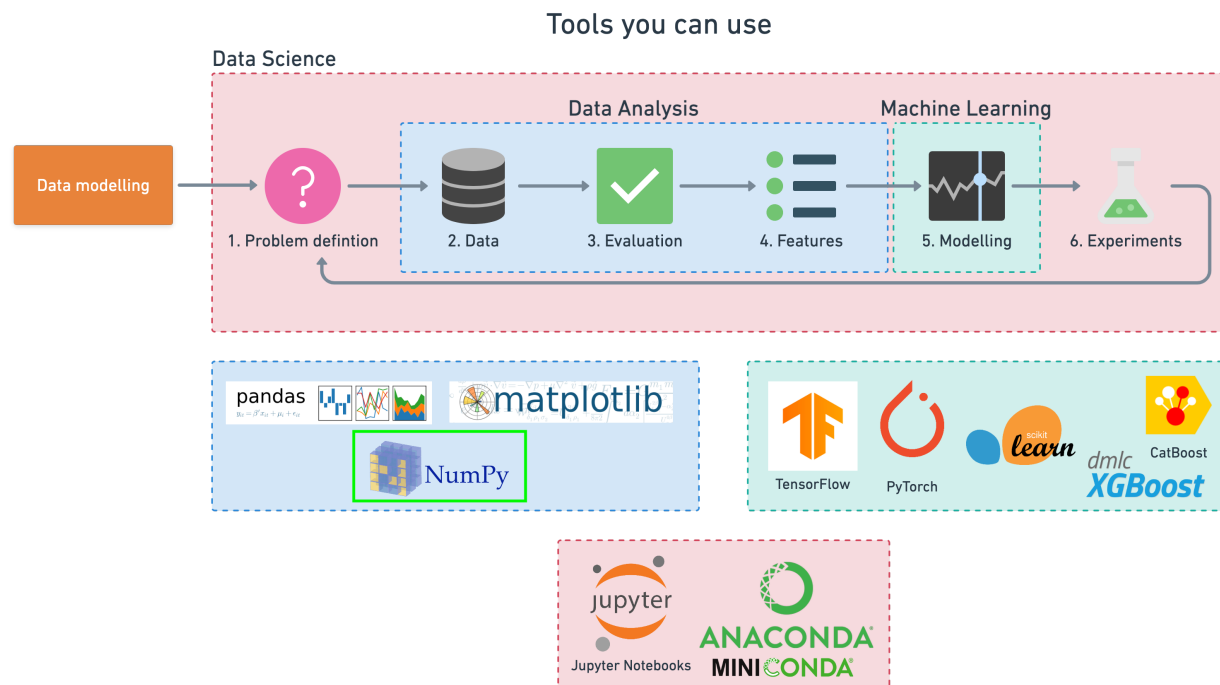
Arithmetic

Broadcasting

Aggregation

Reshaping

Transpose



Dot product

Dot product practical example,
nut butter sales

Why NumPy?

You can do numerical calculations using pure Python. In the beginning, you might think Python is fast but once your data gets large, you'll start to notice slow downs.

One of the main reasons you use NumPy is because it's fast. Behind the scenes, the code has been optimized to run using C. Which is another programming language, which can do things much faster than Python.

[PDFmyURL](#) converts web pages and even full websites to PDF easily and quickly.

The benefit of this being behind the scenes is you don't need to know any C to take advantage of it. You can write your numerical computations in Python using NumPy and get the added speed benefits.

If your curious as to what causes this speed benefit, it's a process called vectorization. [Vectorization](#) aims to do calculations by avoiding loops as loops can create potential bottlenecks.

NumPy achieves vectorization through a process called [broadcasting](#).

What does this notebook cover?

The NumPy library is very capable. However, learning everything off by heart isn't necessary. Instead, this notebook focuses on the main concepts of NumPy and the `ndarray` datatype.

You can think of the `ndarray` datatype as a very flexible array of numbers.

More specifically, we'll look at:

- NumPy datatypes & attributes
- Creating arrays
- Viewing arrays & matrices (indexing)
- Manipulating & comparing arrays
- Sorting arrays
- Use cases (examples of turning things into numbers)

After going through it, you'll have the base knowledge of NumPy you need to keep moving forward.

Where can I get help?

If you get stuck or think of something you'd like to do which this notebook doesn't cover, don't fear!

The recommended steps you take are:

1. **Try it** - Since NumPy is very friendly, your first step should be to use what you know and try figure out the answer to your own question (getting it wrong is part of the process). If in doubt, run your code.
2. **Search for it** - If trying it on your own doesn't work, since someone else has probably tried to do something similar, try searching for your problem in the following places (either via a search engine or direct):
 - [NumPy documentation](#) - The ground truth for everything NumPy, this resource covers all of the NumPy functionality.
 - [Stack Overflow](#) - This is the developers Q&A hub, it's full of questions and answers of different problems across a wide range of software development topics and chances are, there's one related to your problem.
 - [ChatGPT](#) - ChatGPT is very good at explaining code, however, it can make mistakes. Best to verify the code it writes first before using it. Try asking "Can you explain the following code for me? {your code here}" and then continue with follow up questions from there.

An example of searching for a NumPy function might be:

```
"how to find unique elements in a numpy array"
```

Searching this on Google leads to the NumPy documentation for the `np.unique()` function:

<https://numpy.org/doc/stable/reference/generated/numpy.unique.html>

The next steps here are to read through the documentation, check the examples and see if they line up to the problem you're trying to solve.

If they do, **rewrite the code** to suit your needs, run it, and see what the outcomes are.

3. **Ask for help** - If you've been through the above 2 steps and you're still stuck, you might want to ask your question on [Stack Overflow](#). Be as specific as possible and provide details on what you've tried.

Remember, you don't have to learn all of the functions off by heart to begin with.

What's most important is continually asking yourself, "what am I trying to do with the data?".

Start by answering that question and then practicing finding the code which does it.

Let's get started.

0. Importing NumPy

To get started using NumPy, the first step is to import it.

The most common way (and method you should use) is to import NumPy as the abbreviation `np`.

If you see the letters `np` used anywhere in machine learning or data science, it's probably referring to the NumPy library.

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

        # Check the version
        print(np.__version__)
```

1.25.2

1. DataTypes and attributes

Note: Important to remember the main type in NumPy is `ndarray`, even seemingly different kinds of arrays are still `ndarray`'s. This means an operation you do on one array, will work on another.

```
In [3]: # 1-dimensional array, also referred to as a vector
        a1 = np.array([1, 2, 3])

        # 2-dimensional array, also referred to as matrix
        a2 = np.array([[1, 2.0, 3.3],
                       [4, 5, 6.5]])

        # 3-dimensional array, also referred to as a matrix
        a3 = np.array([[[1, 2, 3],
                       [4, 5, 6],
                       [7, 8, 9]],
                       [[10, 11, 12],
                       [13, 14, 15],
                       [16, 17, 18]]])
```

```
In [4]: a1.shape, a1.ndim, a1.dtype, a1.size, type(a1)
```

```
Out[4]: ((3,), 1, dtype('int64'), 3, numpy.ndarray)
```

```
In [5]: a2.shape, a2.ndim, a2.dtype, a2.size, type(a2)
```



```
Out[5]: ((2, 3), 2, dtype('float64'), 6, numpy.ndarray)
```

```
In [6]: a3.shape, a3.ndim, a3.dtype, a3.size, type(a3)
```



```
Out[6]: ((2, 3, 3), 3, dtype('int64'), 18, numpy.ndarray)
```

```
In [7]: a1
```



```
Out[7]: array([1, 2, 3])
```

```
In [8]: a2
```



```
Out[8]: array([[1. , 2. , 3.3],
               [4. , 5. , 6.5]])
```

```
In [9]: a3
```


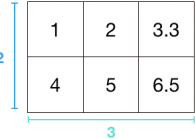
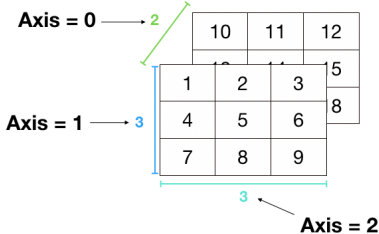


```
Out[9]: array([[[ 1,  2,  3],
                 [ 4,  5,  6],
                 [ 7,  8,  9]],

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

Anatomy of an array

Anatomy of a NumPy array

Data	NumPy	Details
	<pre>array([1, 2, 3])</pre>	<ul style="list-style-type: none">Names: Array, vector1-dimensionalShape = (1, 3)
	<pre>array([[1. , 2. , 3.3], [4. , 5. , 6.5]])</pre>	<ul style="list-style-type: none">Names: Array, matrixMore than 1-dimensionShape = (2, 3)
	<pre>array([[[1, 2, 3], [4, 5, 6], [7, 8, 9]], [[10, 11, 12], [13, 14, 15], [16, 17, 18]])</pre>	<ul style="list-style-type: none">Names: Array, matrixMore than 1-dimensionShape = (2, 3, 3)

Key terms:

- **Array** - A list of numbers, can be multi-dimensional.
- **Scalar** - A single number (e.g. 7).
- **Vector** - A list of numbers with 1-dimension (e.g. `np.array([1, 2, 3])`).
- **Matrix** - A (usually) multi-dimensional list of numbers (e.g. `np.array([[1, 2, 3], [4, 5, 6]])`).

pandas DataFrame out of NumPy arrays

This is to exemplify how NumPy is the backbone of many other libraries.

```
In [10]: import pandas as pd
df = pd.DataFrame(np.random.randint(10, size=(5, 3)),
                  columns=['a', 'b', 'c'])
df
```

Out[10]:

	a	b	c
0	2	3	6
1	1	5	6
2	7	0	2
3	2	1	3
4	8	0	7

```
In [11]: a2
```

Out[11]: array([[1. , 2. , 3.3],
 [4. , 5. , 6.5]])

```
In [12]: df2 = pd.DataFrame(a2)
df2
```

Out[12]:

	0	1	2
0	1.0	2.0	3.3
1	4.0	5.0	6.5

2. Creating arrays

- `np.array()`
- `np.ones()`
- `np.zeros()`
- `np.random.rand(5, 3)`
- `np.random.randint(10, size=5)`
- `np.random.seed()` - pseudo random numbers
- Searching the documentation example (finding `np.unique()` and using it)

```
In [13]: # Create a simple array  
simple_array = np.array([1, 2, 3])  
simple_array
```

```
Out[13]: array([1, 2, 3])
```

```
In [14]: simple_array = np.array((1, 2, 3))  
simple_array, simple_array.dtype
```

```
Out[14]: (array([1, 2, 3]), dtype('int64'))
```

```
In [15]: # Create an array of ones  
ones = np.ones((10, 2))  
ones
```

```
Out[15]: array([[1., 1.],
               [1., 1.],
               [1., 1.],
               [1., 1.],
               [1., 1.],
               [1., 1.],
               [1., 1.],
               [1., 1.],
               [1., 1.]])
```

```
In [16]: # The default datatype is 'float64'
ones.dtype
```

```
Out[16]: dtype('float64')
```

```
In [17]: # You can change the datatype with .astype()
ones.astype(int)
```

```
Out[17]: array([[1, 1],
               [1, 1],
               [1, 1],
               [1, 1],
               [1, 1],
               [1, 1],
               [1, 1],
               [1, 1],
               [1, 1],
               [1, 1]])
```

```
In [18]: # Create an array of zeros
zeros = np.zeros((5, 3, 3))
zeros
```

```
Out[18]: 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.]])
```

```
In [19]: zeros.dtype
```



```
Out[19]: dtype('float64')
```

```
In [20]: # Create an array within a range of values
range_array = np.arange(0, 10, 2)
range_array
```



```
Out[20]: array([0, 2, 4, 6, 8])
```

```
In [21]: # Random array
random_array = np.random.randint(10, size=(5, 3))
random_array
```



```
Out[21]: array([[1, 7, 2],
               [7, 0, 2],
               [8, 8, 8],
               [2, 5, 2],
               [4, 8, 6]])
```

```
In [22]: # Random array of floats (between 0 & 1)
np.random.random((5, 3))
```

```
Out[22]: array([[0.09607892, 0.034903 , 0.47743753],
               [0.51703027, 0.90409121, 0.54436342],
               [0.8095754 , 0.60294712, 0.71141937],
               [0.50802295, 0.57255717, 0.99090604],
               [0.66225284, 0.87588103, 0.25643785]])
```

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

```
Out[23]: array([[0.42800066, 0.76816054, 0.14858447],
               [0.48390262, 0.3708042 , 0.231316 ],
               [0.29166801, 0.64327528, 0.18039386],
               [0.89010443, 0.51218751, 0.31543512],
               [0.38781697, 0.25729731, 0.66219967]])
```

```
In [24]: # Random 5x3 array of floats (between 0 & 1), similar to above
np.random.rand(5, 3)
```

```
Out[24]: array([[0.28373526, 0.10074198, 0.24643463],
               [0.8268303 , 0.48672847, 0.57633359],
               [0.77867161, 0.38490598, 0.53343872],
               [0.67396616, 0.15888354, 0.47710898],
               [0.92319417, 0.19133444, 0.51837588]])
```

```
In [25]: np.random.rand(5, 3)
```

```
Out[25]: array([[0.73585424, 0.83359732, 0.93900774],
               [0.27563836, 0.55971665, 0.26819222],
               [0.29253202, 0.64152402, 0.90479721],
               [0.6585366 , 0.36165565, 0.37515932],
               [0.82890572, 0.54502359, 0.48398256]])
```

NumPy uses pseudo-random numbers, which means, the numbers look random but aren't really, they're predetermined.

For consistency, you might want to keep the random numbers you generate similar throughout experiments.

To do this, you can use `np.random.seed()`.

What this does is it tells NumPy, "Hey, I want you to create random numbers but keep them aligned with the seed."

Let's see it.

```
In [26]: # Set random seed to 0
         np.random.seed(0)

         # Make 'random' numbers
         np.random.randint(10, size=(5, 3))
```

```
Out[26]: array([[5, 0, 3],
               [3, 7, 9],
               [3, 5, 2],
               [4, 7, 6],
               [8, 8, 1]])
```

With `np.random.seed()` set, every time you run the cell above, the same random numbers will be generated.

What if `np.random.seed()` wasn't set?

Every time you run the cell below, a new set of numbers will appear.

```
In [27]: # Make more random numbers  
np.random.randint(10, size=(5, 3))
```

```
Out[27]: array([[6, 7, 7],  
               [8, 1, 5],  
               [9, 8, 9],  
               [4, 3, 0],  
               [3, 5, 0]])
```

Let's see it in action again, we'll stay consistent and set the random seed to 0.

```
In [28]: # Set random seed to same number as above  
np.random.seed(0)  
  
# The same random numbers come out  
np.random.randint(10, size=(5, 3))
```

```
Out[28]: array([[5, 0, 3],  
               [3, 7, 9],  
               [3, 5, 2],  
               [4, 7, 6],  
               [8, 8, 1]])
```

Because `np.random.seed()` is set to 0, the random numbers are the same as the cell with `np.random.seed()` set to 0 as well.

Setting `np.random.seed()` is not 100% necessary but it's helpful to keep numbers the same throughout your experiments.

For example, say you wanted to split your data randomly into training and test sets.

[PDFmyURL](#) converts web pages and even full websites to PDF easily and quickly.

Every time you randomly split, you might get different rows in each set.

If you shared your work with someone else, they'd get different rows in each set too.

Setting `np.random.seed()` ensures there's still randomness, it just makes the randomness repeatable. Hence the 'pseudo-random' numbers.

```
In [29]: np.random.seed(0)
df = pd.DataFrame(np.random.randint(10, size=(5, 3)))
df
```

Out [29]:

	0	1	2
0	5	0	3
1	3	7	9
2	3	5	2
3	4	7	6
4	8	8	1

What unique values are in the array a3?

Now you've seen a few different ways to create arrays, as an exercise, try find out what NumPy function you could use to find the unique values are within the `a3` array.

You might want to search some like, "how to find the unique values in a numpy array".

```
In [30]: # Your code here
```


3. Viewing arrays and matrices (indexing)

Remember, because arrays and matrices are both `ndarray`'s, they can be viewed in similar ways.

Let's check out our 3 arrays again.

```
In [31]: a1
```

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

```
In [32]: a2
```

```
Out[32]: array([[1. , 2. , 3.3],
               [4. , 5. , 6.5]])
```

```
In [33]: a3
```

```
Out[33]: array([[ 1,  2,  3],
               [ 4,  5,  6],
               [ 7,  8,  9]],

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

Array shapes are always listed in the format `(row, column, n, n, n...)` where `n` is optional extra dimensions.

```
In [34]: a1[0]
```

```
Out[34]: 1
```

```
In [35]: a2[0]
```

```
Out[35]: array([1. , 2. , 3.3])
```

```
In [36]: a3[0]
```

```
Out[36]: array([[1, 2, 3],  
               [4, 5, 6],  
               [7, 8, 9]])
```

```
In [37]: # Get 2nd row (index 1) of a2  
a2[1]
```

```
Out[37]: array([4. , 5. , 6.5])
```

```
In [38]: # Get the first 2 values of the first 2 rows of both arrays  
a3[:2, :2, :2]
```

```
Out[38]: array([[[ 1,  2],  
                [ 4,  5]],  
               [[10, 11],  
                [13, 14]])
```

This takes a bit of practice, especially when the dimensions get higher. Usually, it takes me a little trial and error of trying to get certain values, viewing the output in the notebook and trying again.

NumPy arrays get printed from outside to inside. This means the number at the end of the shape comes first, and the number at the start of the shape comes last.

```
In [39]: a4 = np.random.randint(10, size=(2, 3, 4, 5))  
a4
```



```
Out[39]: array([[[[6, 7, 7, 8, 1],  
                [5, 9, 8, 9, 4],  
                [3, 0, 3, 5, 0],  
                [2, 3, 8, 1, 3]],  
                [[3, 3, 7, 0, 1],  
                [9, 9, 0, 4, 7],  
                [3, 2, 7, 2, 0],  
                [0, 4, 5, 5, 6]],  
                [[8, 4, 1, 4, 9],  
                [8, 1, 1, 7, 9],  
                [9, 3, 6, 7, 2],  
                [0, 3, 5, 9, 4]]],  
                [[[4, 6, 4, 4, 3],  
                [4, 4, 8, 4, 3],  
                [7, 5, 5, 0, 1],  
                [5, 9, 3, 0, 5]],  
                [[0, 1, 2, 4, 2],  
                [0, 3, 2, 0, 7],  
                [5, 9, 0, 2, 7],  
                [2, 9, 2, 3, 3]],  
                [[2, 3, 4, 1, 2],  
                [9, 1, 4, 6, 8],  
                [2, 3, 0, 0, 6],  
                [0, 6, 3, 3, 8]]]])
```

```
In [40]: a4.shape
```



Out[40]: (2, 3, 4, 5)

In [41]: *# Get only the first 4 numbers of each single vector*
a4[:, :, :, :4]



Out[41]: array([[[[6, 7, 7, 8],
[5, 9, 8, 9],
[3, 0, 3, 5],
[2, 3, 8, 1]],

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

[[8, 4, 1, 4],
[8, 1, 1, 7],
[9, 3, 6, 7],
[0, 3, 5, 9]]],

[[[4, 6, 4, 4],
[4, 4, 8, 4],
[7, 5, 5, 0],
[5, 9, 3, 0]],

[[0, 1, 2, 4],
[0, 3, 2, 0],
[5, 9, 0, 2],
[2, 9, 2, 3]],

[[2, 3, 4, 1],
[9, 1, 4, 6],
[2, 3, 0, 0],
[0, 6, 3, 3]]]])

`a4`'s shape is (2, 3, 4, 5), this means it gets displayed like so:

- Inner most array = size 5
- Next array = size 4
- Next array = size 3
- Outer most array = size 2

4. Manipulating and comparing arrays

- Arithmetic
 - `+`, `-`, `*`, `/`, `//`, `**`, `%`
 - `np.exp()`
 - `np.log()`
 - **Dot product** - `np.dot()`
 - Broadcasting
- Aggregation
 - `np.sum()` - faster than Python's `.sum()` for NumPy arrays
 - `np.mean()`
 - `np.std()`
 - `np.var()`

- `np.min()`
- `np.max()`
- `np.argmin()` - find index of minimum value
- `np.argmax()` - find index of maximum value
- These work on all `ndarray`'s
 - `a4.min(axis=0)` – you can use axis as well
- Reshaping
 - `np.reshape()`
- Transposing
 - `a3.T`
- Comparison operators
 - `>`
 - `<`
 - `<=`
 - `>=`
 - `x != 3`
 - `x == 3`
 - `np.sum(x > 3)`

Arithmetic

In [42]: `a1`



Out[42]: `array([1, 2, 3])`

In [43]: `ones = np.ones(3)`
`ones`



Out[43]: `array([1., 1., 1.])`

In [44]: `# Add two arrays`
`a1 + ones`



Out[44]: `array([2., 3., 4.])`

In [45]: `# Subtract two arrays`
`a1 - ones`



Out[45]: `array([0., 1., 2.])`

In [46]: `# Multiply two arrays`
`a1 * ones`



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

In [47]: `# Multiply two arrays`
`a1 * a2`



```
Out[47]: array([[ 1. ,  4. ,  9.9],
               [ 4. , 10. , 19.5]])
```

```
In [48]: a1.shape, a2.shape
```

```
Out[48]: ((3,), (2, 3))
```

```
In [49]: # This will error as the arrays have a different number of dimensions (2, 3) vs. (2, 3, 3)
a2 * a3
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[49], line 2
      1 # This will error as the arrays have a different number of dimensions (2,
      2 3) vs. (2, 3, 3)
----> 2 a2 * a3

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

```
In [50]: a3
```

```
Out[50]: array([[[ 1,  2,  3],
                 [ 4,  5,  6],
                 [ 7,  8,  9]],

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

Broadcasting

- What is broadcasting?

[PDFmyURL](#) converts web pages and even full websites to PDF easily and quickly.

- Broadcasting is a feature of NumPy which performs an operation across multiple dimensions of data without replicating the data. This saves time and space. For example, if you have a 3x3 array (A) and want to add a 1x3 array (B), NumPy will add the row of (B) to every row of (A).
- Rules of Broadcasting
 - a. If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.
 - b. If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
 - c. If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

The broadcasting rule: In order to broadcast, the size of the trailing axes for both arrays in an operation must be either the same size or one of them must be one.

```
In [51]: a1
```

```
Out[51]: array([1, 2, 3])
```

```
In [52]: a1.shape
```

```
Out[52]: (3,)
```

```
In [53]: a2.shape
```

```
Out[53]: (2, 3)
```

```
In [54]: a2
```

```
Out[54]: array([[1. , 2. , 3.3],
               [4. , 5. , 6.5]])
```

```
In [55]: a1 + a2
```

```
Out[55]: array([[2. , 4. , 6.3],
               [5. , 7. , 9.5]])
```

```
In [56]: a2 + 2
```

```
Out[56]: array([[3. , 4. , 5.3],
               [6. , 7. , 8.5]])
```

```
In [57]: # Raises an error because there's a shape mismatch (2, 3) vs. (2, 3, 3)
a2 + a3
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[57], line 2
      1 # Raises an error because there's a shape mismatch (2, 3) vs. (2, 3, 3)
----> 2 a2 + a3

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

```
In [58]: # Divide two arrays
a1 / ones
```

```
Out[58]: array([1., 2., 3.])
```

```
In [59]: # Divide using floor division
a2 // a1
```

```
Out[59]: array([[1., 1., 1.],
               [4., 2., 2.]])
```

```
In [60]: # Take an array to a power  
a1 ** 2
```

```
Out[60]: array([1, 4, 9])
```

```
In [61]: # You can also use np.square()  
np.square(a1)
```

```
Out[61]: array([1, 4, 9])
```

```
In [62]: # Modulus divide (what's the remainder)  
a1 % 2
```

```
Out[62]: array([1, 0, 1])
```

You can also find the log or exponential of an array using `np.log()` and `np.exp()`.

```
In [63]: # Find the log of an array  
np.log(a1)
```

```
Out[63]: array([0.          , 0.69314718, 1.09861229])
```

```
In [64]: # Find the exponential of an array  
np.exp(a1)
```

```
Out[64]: array([ 2.71828183,  7.3890561 , 20.08553692])
```

Aggregation

Aggregation - bringing things together, doing a similar thing on a number of things.

[PDFmyURL](#) converts web pages and even full websites to PDF easily and quickly.

```
In [65]: sum(a1)
```

```
Out[65]: 6
```

```
In [66]: np.sum(a1)
```

```
Out[66]: 6
```

Tip: Use NumPy's `np.sum()` on NumPy arrays and Python's `sum()` on Python `list` s.

```
In [67]: massive_array = np.random.random(100000)
         massive_array.size, type(massive_array)
```

```
Out[67]: (100000, numpy.ndarray)
```

```
In [68]: %timeit sum(massive_array) # Python sum()
         %timeit np.sum(massive_array) # NumPy np.sum()
```

```
4.38 ms ± 119 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
20.3 µs ± 110 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Notice `np.sum()` is faster on the Numpy array (`numpy.ndarray`) than Python's `sum()`.

Now let's try it out on a Python list.

```
In [69]: import random
         massive_list = [random.randint(0, 10) for i in range(100000)]
         len(massive_list), type(massive_list)
```

```
Out[69]: (100000, list)
```

```
In [70]: massive_list[:10]
```

```
Out[70]: [0, 4, 5, 9, 7, 0, 1, 7, 8, 1]
```

```
In [71]: %timeit sum(massive_list)
         %timeit np.sum(massive_list)
```

598 μ s \pm 959 ns per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
2.72 ms \pm 10.6 μ s per loop (mean \pm std. dev. of 7 runs, 100 loops each)

NumPy's `np.sum()` is still fast but Python's `sum()` is faster on Python `list`s.

```
In [72]: a2
```

```
Out[72]: array([[1. , 2. , 3.3],
               [4. , 5. , 6.5]])
```

```
In [73]: # Find the mean
         np.mean(a2)
```

```
Out[73]: 3.6333333333333333
```

```
In [74]: # Find the max
         np.max(a2)
```

```
Out[74]: 6.5
```

```
In [75]: # Find the min
         np.min(a2)
```

```
Out[75]: 1.0
```

```
In [76]: # Find the standard deviation
         np.std(a2)
```

```
Out[76]: 1.8226964152656422
```

```
In [77]: # Find the variance  
np.var(a2)
```

```
Out[77]: 3.3222222222222224
```

```
In [78]: # The standard deviation is the square root of the variance  
np.sqrt(np.var(a2))
```

```
Out[78]: 1.8226964152656422
```

What's mean?

Mean is the same as average. You can find the average of a set of numbers by adding them up and dividing them by how many there are.

What's standard deviation?

[Standard deviation](#) is a measure of how spread out numbers are.

What's variance?

The [variance](#) is the averaged squared differences of the mean.

To work it out, you:

1. Work out the mean
2. For each number, subtract the mean and square the result
3. Find the average of the squared differences

```
In [79]: # Demo of variance  
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)
```

```
Out[79]: (4296133.472222221, 8.0)
```

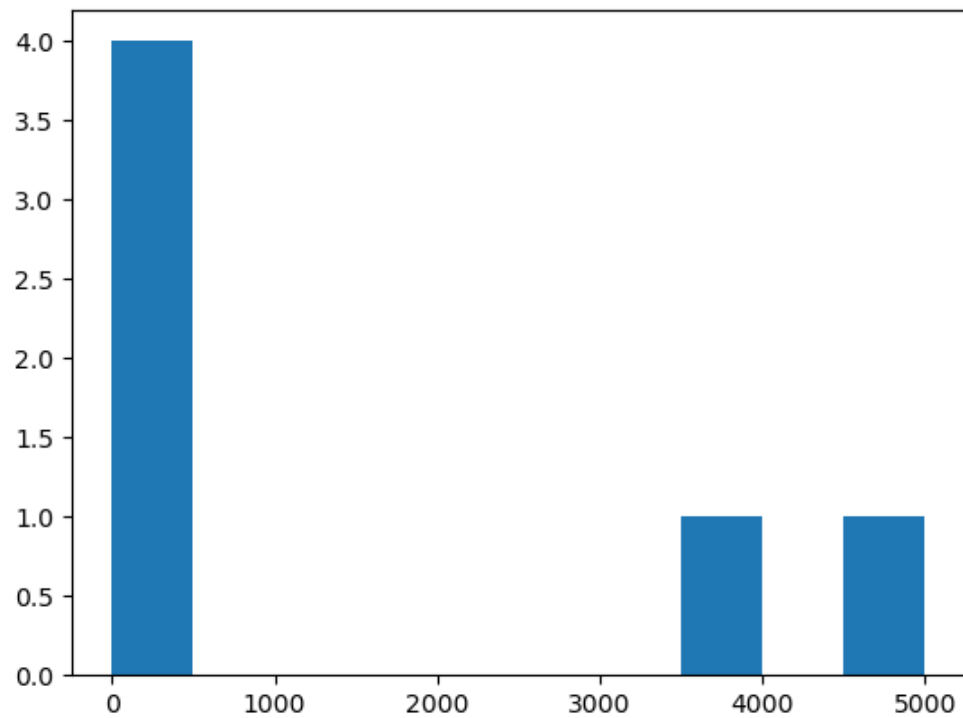
```
In [80]: np.std(high_var_array), np.std(low_var_array)
```

```
Out[80]: (2072.711623024829, 2.8284271247461903)
```

```
In [81]: # The standard deviation is the square root of the variance  
np.sqrt(np.var(high_var_array))
```

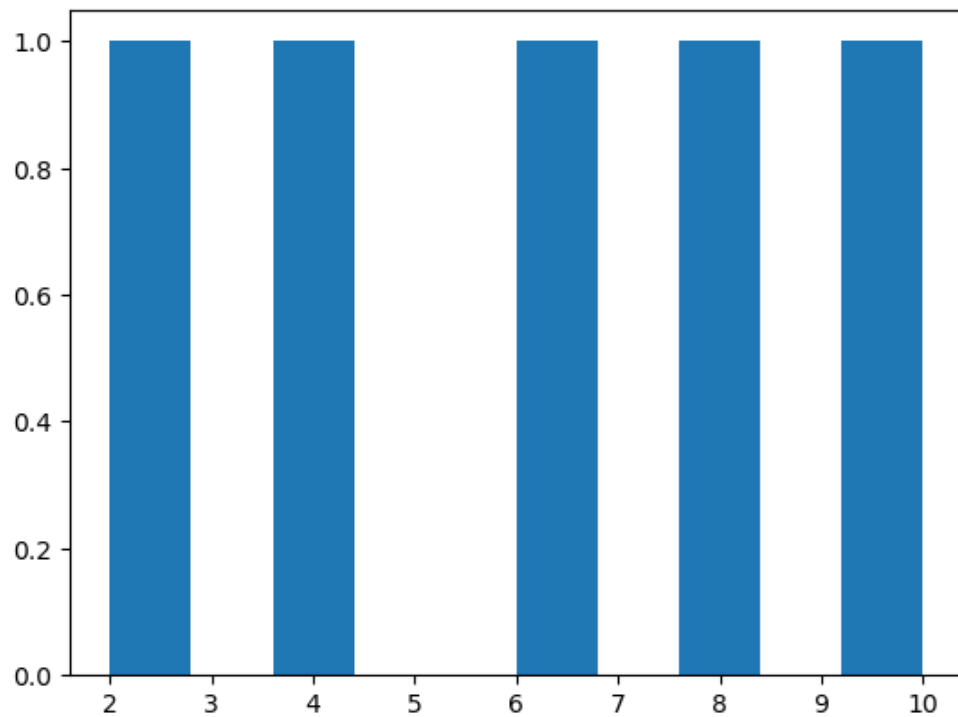
```
Out[81]: 2072.711623024829
```

```
In [82]: %matplotlib inline  
import matplotlib.pyplot as plt  
plt.hist(high_var_array)  
plt.show()
```



```
In [83]: plt.hist(low_var_array)
plt.show()
```





Reshaping

```
In [84]: a2
```



```
Out[84]: array([[1. , 2. , 3.3],  
               [4. , 5. , 6.5]])
```

```
In [85]: a2.shape
```



```
Out[85]: (2, 3)
```

```
In [86]: a2 + a3
```

```
-----  
ValueError                                Traceback (most recent call last)  
Cell In[86], line 1  
----> 1 a2 + a3  
  
ValueError: operands could not be broadcast together with shapes (2,3) (2,3,3)
```

```
In [ ]: a2.reshape(2, 3, 1)
```

```
In [ ]: a2.reshape(2, 3, 1) + a3
```

Transpose

A tranpose reverses the order of the axes.

For example, an array with shape `(2, 3)` becomes `(3, 2)`.

```
In [ ]: a2.shape
```

```
In [ ]: a2.T
```

```
In [ ]: a2.transpose()
```

```
In [ ]: a2.T.shape
```

For larger arrays, the default value of a transpose is to swap the first and last axes.

For example, (5, 3, 3) -> (3, 3, 5).

```
In [ ]: matrix = np.random.random(size=(5, 3, 3))  
matrix
```

```
In [ ]: matrix.shape
```

```
In [ ]: matrix.T
```

```
In [ ]: matrix.T.shape
```

```
In [ ]: # Check to see if the reverse shape is same as transpose shape  
matrix.T.shape == matrix.shape[::-1]
```

```
In [ ]: # Check to see if the first and last axes are swapped  
matrix.T == matrix.swapaxes(0, -1) # swap first (0) and last (-1) axes
```

You can see more advanced forms of transposing in the NumPy documentation under [numpy.transpose](#).

Dot product

The main two rules for dot product to remember are:

1. The **inner dimensions** must match:

[PDFmyURL](#) converts web pages and even full websites to PDF easily and quickly.

- `(3, 2) @ (3, 2)` won't work
- `(2, 3) @ (3, 2)` will work
- `(3, 2) @ (2, 3)` will work

2. The resulting matrix has the shape of the **outer dimensions**:

- `(2, 3) @ (3, 2) -> (2, 2)`
- `(3, 2) @ (2, 3) -> (3, 3)`

Note: In NumPy, `np.dot()` and `@` can be used to achieve the same result for 1-2 dimension arrays. However, their behaviour begins to differ at arrays with 3+ dimensions.

```
In [ ]: np.random.seed(0)
        mat1 = np.random.randint(10, size=(3, 3))
        mat2 = np.random.randint(10, size=(3, 2))

        mat1.shape, mat2.shape
```

```
In [ ]: mat1
```

```
In [ ]: mat2
```

```
In [ ]: np.dot(mat1, mat2)
```

```
In [ ]: # Can also achieve np.dot() with "@"
        # (however, they may behave differently at 3D+ arrays)
        mat1 @ mat2
```

```
In [ ]: np.random.seed(0)
mat3 = np.random.randint(10, size=(4,3))
mat4 = np.random.randint(10, size=(4,3))
mat3
```

```
In [ ]: mat4
```

```
In [ ]: # This will fail as the inner dimensions of the matrices do not match
np.dot(mat3, mat4)
```

```
In [ ]: mat3.T.shape
```

```
In [ ]: # Dot product
np.dot(mat3.T, mat4)
```

```
In [ ]: # Element-wise multiplication, also known as Hadamard product
mat3 * mat4
```

Dot product practical example, nut butter sales

```
In [ ]: np.random.seed(0)
sales_amounts = np.random.randint(20, size=(5, 3))
sales_amounts
```

```
In [ ]: weekly_sales = pd.DataFrame(sales_amounts,
                                     index=["Mon", "Tues", "Wed", "Thurs", "Fri"],
```

```
columns=["Almond butter", "Peanut butter", "Cashew bu  
weekly_sales
```

```
In [87]: prices = np.array([10, 8, 12])  
prices
```

```
Out[87]: array([10,  8, 12])
```

```
In [88]: butter_prices = pd.DataFrame(prices.reshape(1, 3),  
index=["Price"],  
columns=["Almond butter", "Peanut butter", "Cashew b  
butter_prices.shape
```

```
Out[88]: (1, 3)
```

```
In [89]: weekly_sales.shape
```

```
-----  
NameError                                Traceback (most recent call last)  
Cell In[89], line 1  
----> 1 weekly_sales.shape  
  
NameError: name 'weekly_sales' is not defined
```

```
In [90]: # Find the total amount of sales for a whole day  
total_sales = prices.dot(sales_amounts)  
total_sales
```

```
-----  
NameError                                Traceback (most recent call last)  
Cell In[90], line 2  
      1 # Find the total amount of sales for a whole day  
----> 2 total_sales = prices.dot(sales_amounts)  
      3 total_sales
```

```
NameError: name 'sales_amounts' is not defined
```

The shapes aren't aligned, we need the middle two numbers to be the same.

```
In [91]: prices
```

```
Out[91]: array([10,  8, 12])
```

```
In [92]: sales_amounts.T.shape
```

```
-----
NameError                                Traceback (most recent call last)
Cell In[92], line 1
----> 1 sales_amounts.T.shape

NameError: name 'sales_amounts' is not defined
```

```
In [93]: # To make the middle numbers the same, we can transpose
        total_sales = prices.dot(sales_amounts.T)
        total_sales
```

```
-----
NameError                                Traceback (most recent call last)
Cell In[93], line 2
      1 # To make the middle numbers the same, we can transpose
----> 2 total_sales = prices.dot(sales_amounts.T)
      3 total_sales

NameError: name 'sales_amounts' is not defined
```

```
In [94]: butter_prices.shape, weekly_sales.shape
```

```
-----
NameError                                Traceback (most recent call last)
```

```
Cell In[94], line 1
----> 1 butter_prices.shape, weekly_sales.shape
```

NameError: name 'weekly_sales' is not defined

```
In [95]: daily_sales = butter_prices.dot(weekly_sales.T)
        daily_sales
```

NameError Traceback (most recent call last)

```
Cell In[95], line 1
----> 1 daily_sales = butter_prices.dot(weekly_sales.T)
      2 daily_sales
```

NameError: name 'weekly_sales' is not defined

```
In [96]: # Need to transpose again
        weekly_sales["Total"] = daily_sales.T
        weekly_sales
```

NameError Traceback (most recent call last)

```
Cell In[96], line 2
      1 # Need to transpose again
----> 2 weekly_sales["Total"] = daily_sales.T
      3 weekly_sales
```

NameError: name 'daily_sales' is not defined

Comparison operators

Finding out if one array is larger, smaller or equal to another.


```
In [97]: a1
```

```
Out[97]: array([1, 2, 3])
```

```
In [98]: a2
```

```
Out[98]: array([[1. , 2. , 3.3],  
               [4. , 5. , 6.5]])
```

```
In [99]: a1 > a2
```

```
Out[99]: array([[False, False, False],  
               [False, False, False]])
```

```
In [100]: a1 >= a2
```

```
Out[100]: array([[ True,  True, False],  
               [False, False, False]])
```

```
In [101]: a1 > 5
```

```
Out[101]: array([False, False, False])
```

```
In [102]: a1 == a1
```

```
Out[102]: array([ True,  True,  True])
```

```
In [103]: a1 == a2
```

```
Out[103]: array([[ True,  True, False],  
               [False, False, False]])
```

5. Sorting arrays

- `np.sort()` - sort values in a specified dimension of an array.
- `np.argsort()` - return the indices to sort the array on a given axis.
- `np.argmax()` - return the index/indices which gives the highest value(s) along an axis.
- `np.argmin()` - return the index/indices which gives the lowest value(s) along an axis.

```
In [104]: random_array
```



```
Out[104]: array([[1, 7, 2],  
                [7, 0, 2],  
                [8, 8, 8],  
                [2, 5, 2],  
                [4, 8, 6]])
```

```
In [105]: np.sort(random_array)
```



```
Out[105]: array([[1, 2, 7],  
                [0, 2, 7],  
                [8, 8, 8],  
                [2, 2, 5],  
                [4, 6, 8]])
```

```
In [106]: np.argsort(random_array)
```



```
Out[106]: array([[0, 2, 1],  
                [1, 2, 0],  
                [0, 1, 2],
```

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

```
In [107]: a1
```

```
Out[107]: array([1, 2, 3])
```

```
In [108]: # Return the indices that would sort an array  
np.argsort(a1)
```

```
Out[108]: array([0, 1, 2])
```

```
In [109]: # No axis  
np.argmin(a1)
```

```
Out[109]: 0
```

```
In [110]: random_array
```

```
Out[110]: array([[1, 7, 2],  
                [7, 0, 2],  
                [8, 8, 8],  
                [2, 5, 2],  
                [4, 8, 6]])
```

```
In [111]: # Down the vertical  
np.argmax(random_array, axis=1)
```

```
Out[111]: array([1, 0, 0, 1, 1])
```

```
In [112]: # Across the horizontal  
np.argmin(random_array, axis=0)
```

```
Out[112]: array([0, 1, 0])
```

6. Use case

Turning an image into a NumPy array.

Why?

Because computers can use the numbers in the NumPy array to find patterns in the image and in turn use those patterns to figure out what's in the image.

This is what happens in modern computer vision algorithms.

Let's start with this beautiful image of a panda:



```
In [ ]: from matplotlib.image import imread

panda = imread('../images/numpy-panda.jpeg')
print(type(panda))

<class 'numpy.ndarray'>
```

```
In [114]: panda.shape
```

```
Out[114]: (2330, 3500, 3)
```

```
In [115]: panda
```

```
Out[115]: array([[0.05490196, 0.10588235, 0.06666667],
                 [0.05490196, 0.10588235, 0.06666667],
                 [0.05490196, 0.10588235, 0.06666667],
                 ...,
                 [0.16470589, 0.12941177, 0.09411765],
                 [0.16470589, 0.12941177, 0.09411765],
                 [0.16470589, 0.12941177, 0.09411765]],

                [[0.05490196, 0.10588235, 0.06666667],
                 [0.05490196, 0.10588235, 0.06666667],
                 [0.05490196, 0.10588235, 0.06666667],
                 ...,
                 [0.16470589, 0.12941177, 0.09411765],
                 [0.16470589, 0.12941177, 0.09411765],
                 [0.16470589, 0.12941177, 0.09411765]],

                [[0.05490196, 0.10588235, 0.06666667],
                 [0.05490196, 0.10588235, 0.06666667],
                 [0.05490196, 0.10588235, 0.06666667],
                 ...,
                 [0.16470589, 0.12941177, 0.09411765],
                 [0.16470589, 0.12941177, 0.09411765],
                 [0.16470589, 0.12941177, 0.09411765]])
```

```

[0.16470589, 0.12941177, 0.09411765]],

...,

[[0.13333334, 0.07450981, 0.05490196],
 [0.12156863, 0.0627451 , 0.04313726],
 [0.10980392, 0.05098039, 0.03137255],
 ...,
 [0.02745098, 0.02745098, 0.03529412],
 [0.02745098, 0.02745098, 0.03529412],
 [0.02745098, 0.02745098, 0.03529412]],

[[0.13333334, 0.07450981, 0.05490196],
 [0.12156863, 0.0627451 , 0.04313726],
 [0.12156863, 0.0627451 , 0.04313726],
 ...,
 [0.02352941, 0.02352941, 0.03137255],
 [0.02352941, 0.02352941, 0.03137255],
 [0.02352941, 0.02352941, 0.03137255]],

[[0.13333334, 0.07450981, 0.05490196],
 [0.12156863, 0.0627451 , 0.04313726],
 [0.12156863, 0.0627451 , 0.04313726],
 ...,
 [0.02352941, 0.02352941, 0.03137255],
 [0.02352941, 0.02352941, 0.03137255],
 [0.02352941, 0.02352941, 0.03137255]]], dtype=float32)

```



```
In [ ]: car = imread("../images/numpy-car-photo.png")  
car.shape
```



```
Out[ ]: (431, 575, 4)
```

```
In [117]: car[:, :, :3].shape
```



```
Out[117]: (431, 575, 3)
```

Made with Material for MkDocs Insiders



Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js



```
In [118]: dog = imread("../images/numpy-dog-photo.png")
          dog.shape
```



```
Out[118]: (432, 575, 4)
```

```
In [119]: dog
```



```
Out[119]: array([[0.70980394, 0.80784315, 0.88235295, 1.         ],
                  [0.72156864, 0.8117647 , 0.8862745 , 1.         ],
                  [0.7411765 , 0.8156863 , 0.8862745 , 1.         ]])
```

[PDFmyURL](#) converts web pages and even full websites to PDF easily and quickly.


```

...,
[0.49803922, 0.6862745 , 0.8392157 , 1.      ],
[0.49411765, 0.68235296, 0.8392157 , 1.      ],
[0.49411765, 0.68235296, 0.8352941 , 1.      ]],

[[0.69411767, 0.8039216 , 0.8862745 , 1.      ],
[0.7019608 , 0.8039216 , 0.88235295, 1.      ],
[0.7058824 , 0.80784315, 0.88235295, 1.      ],

...,
[0.5019608 , 0.6862745 , 0.84705883, 1.      ],
[0.49411765, 0.68235296, 0.84313726, 1.      ],
[0.49411765, 0.68235296, 0.8392157 , 1.      ]],

[[0.6901961 , 0.8      , 0.88235295, 1.      ],
[0.69803923, 0.8039216 , 0.88235295, 1.      ],
[0.7058824 , 0.80784315, 0.88235295, 1.      ],

...,
[0.5019608 , 0.6862745 , 0.84705883, 1.      ],
[0.49803922, 0.6862745 , 0.84313726, 1.      ],
[0.49803922, 0.6862745 , 0.84313726, 1.      ]],

...,

[[0.9098039 , 0.81960785, 0.654902  , 1.      ],
[0.8352941 , 0.7490196 , 0.6509804 , 1.      ],
[0.72156864, 0.6313726 , 0.5372549 , 1.      ],

...,
[0.01568628, 0.07058824, 0.02352941, 1.      ],
[0.03921569, 0.09411765, 0.03529412, 1.      ],
[0.03921569, 0.09019608, 0.05490196, 1.      ]],

[[0.9137255 , 0.83137256, 0.6784314 , 1.      ],
[0.8117647 , 0.7294118 , 0.627451  , 1.      ],
[0.65882355, 0.5686275 , 0.47843137, 1.      ],

...,
[0.00392157, 0.05490196, 0.03529412, 1.      ],
[0.03137255, 0.09019608, 0.05490196, 1.      ],

```

```
[0.04705882, 0.10588235, 0.06666667, 1.      ]],  
[[0.9137255 , 0.83137256, 0.68235296, 1.      ],  
 [0.76862746, 0.68235296, 0.5882353 , 1.      ],  
 [0.59607846, 0.5058824 , 0.44313726, 1.      ],  
 ...,  
 [0.03921569, 0.10196079, 0.07058824, 1.      ],  
 [0.02745098, 0.08235294, 0.05882353, 1.      ],  
 [0.05098039, 0.11372549, 0.07058824, 1.      ]]], dtype=float32)
```