View source code | Read notebook in online book format

A Quick Introduction to Numerical Data Manipulation with Python and NumPy

```
import datetime
print(f"Last updated: {datetime.datetime.now()}")
```

```
Last updated: 2024-09-05 13:15:36.894029
```

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.

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.

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 knolwedge 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. Avoid straight copying and pasting and instead, only use things that you could yourself reproduce with adequate effort.

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.

1. **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.

```
import numpy as np

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

```
2.1.1
```

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.

```
# 1-dimensonal 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]]])

a1.shape, a1.ndim, a1.dtype, a1.size, type(a1)
```

```
((3,), 1, dtype('int64'), 3, numpy.ndarray)
```

```
a2.shape, a2.ndim, a2.dtype, a2.size, type(a2)
```

```
((2, 3), 2, dtype('float64'), 6, numpy.ndarray)
```

```
a3.shape, a3.ndim, a3.dtype, a3.size, type(a3)
```

```
((2, 3, 3), 3, dtype('int64'), 18, numpy.ndarray)
```

```
a1
```

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

```
a2
```

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

```
a3
```

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

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

Anatomy of an array

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 examplify how NumPy is the backbone of many other libraries.

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

```
   a  b  c
0  5  8  0
1  3  3  2
```

```
2  1  6  7
3  7  3  9
4  6  6  7
```

```
a2
```

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

```
df2 = pd.DataFrame(a2)
df2
```

```
     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)

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

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

```
simple_array = np.array((1, 2, 3))
simple_array, simple_array.dtype
```

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

```
# Create an array of ones
ones = np.ones((10, 2))
ones
```

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

```
# The default datatype is 'float64'
ones.dtype
```

```
dtype('float64')
```

```
# You can change the datatype with .astype()
ones.astype(int)
```

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

```
# Create an array of zeros
zeros = np.zeros((5, 3, 3))
zeros
```

```
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.]]])
```

```
zeros.dtype
```

```
dtype('float64')
```

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

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

```python
# Random array
random_array = np.random.randint(10, size=(5, 3))
random_array
```

```
array([[8, 7, 6],
       [4, 2, 7],
       [6, 0, 6],
       [0, 8, 5],
       [6, 2, 9]])
```

```python
# Random array of floats (between 0 & 1)
np.random.random((5, 3))
```

```
array([[0.47811645, 0.49437395, 0.09426995],
       [0.80062461, 0.41609157, 0.45268566],
       [0.24531914, 0.56982162, 0.36856519],
       [0.32292926, 0.03760924, 0.13312765],
       [0.66844485, 0.88781517, 0.21807957]])
```

```python
np.random.random((5, 3))
```

```
array([[0.96868201, 0.87777028, 0.21900062],
       [0.88225041, 0.73815918, 0.83321165],
       [0.14038979, 0.79643185, 0.2741666 ],
       [0.48166491, 0.74364069, 0.75385132],
       [0.58920305, 0.43270563, 0.42922598]])
```

```python
# Random 5x3 array of floats (between 0 & 1), similar to above
np.random.rand(5, 3)
```

```
array([[0.90225603, 0.76253433, 0.84856067],
       [0.8961939 , 0.37019149, 0.00568981],
       [0.78797133, 0.07953581, 0.99870521],
       [0.07481087, 0.74846133, 0.0788899 ],
       [0.40156115, 0.80716411, 0.37204142]])
```

```python
np.random.rand(5, 3)
```

```
array([[0.80767414, 0.62863218, 0.32492877],
       [0.71402148, 0.06601142, 0.16626604],
       [0.81986587, 0.75875945, 0.73266779],
       [0.4233863 , 0.52077358, 0.21571921],
       [0.75862881, 0.65817717, 0.74667541]])
```

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.

```
# Set random seed to 0
np.random.seed(0)

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

```
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.

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

```
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.

```
# 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))
```

```
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.

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.

```
np.random.seed(0)
df = pd.DataFrame(np.random.randint(10, size=(5, 3)))
df
```

```
   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 unqiue values in a numpy array".

```
# 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.

```
a1
```

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

```
a2
```

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

```
a3
```

```
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.

```
a1[0]
```

```
np.int64(1)
```

```
a2[0]
```

```
array([1. , 2. , 3.3])
```

```
a3[0]
```

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

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

```
array([4. , 5. , 6.5])
```

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

```
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.

```
a4 = np.random.randint(10, size=(2, 3, 4, 5))
a4
```

```
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]]]])
```

```python
a4.shape
```

```
(2, 3, 4, 5)
```

```python
# Get only the first 4 numbers of each single vector
a4[:, :, :, :4]
```

```
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

```
a1
```

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

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

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

```
# Add two arrays
a1 + ones
```

```
array([2., 3., 4.])
```

```
# Subtract two arrays
a1 - ones
```

```
array([0., 1., 2.])
```

```
# Multiply two arrays
a1 * ones
```

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

```
# Multiply two arrays
a1 * a2
```

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

```
a1.shape, a2.shape
```

```
((3,), (2, 3))
```

```
# 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, 3) vs. (2, 3, 3)
----> 2 a2 * a3

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

```
a3
```

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

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

Broadcasting

- What is broadcasting?
  - 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.

```
a1
```

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

```
a1.shape
```

```
(3,)
```

```
a2.shape
```

```
(2, 3)
```

```
a2
```

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

a1 + a2

array([[2. , 4. , 6.3],
       [5. , 7. , 9.5]])

a2 + 2

array([[3. , 4. , 5.3],
       [6. , 7. , 8.5]])

# 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)

# Divide two arrays
a1 / ones

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

# Divide using floor division
a2 // a1

array([[1., 1., 1.],
       [4., 2., 2.]])

# Take an array to a power
a1 ** 2

array([1, 4, 9])

# You can also use np.square()
np.square(a1)

array([1, 4, 9])

# Modulus divide (what's the remainder)
a1 % 2
```

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

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

```
# Find the log of an array
np.log(a1)
```

```
array([0.        , 0.69314718, 1.09861229])
```

```
# Find the exponential of an array
np.exp(a1)
```

```
array([ 2.71828183,  7.3890561 , 20.08553692])
```

Aggregation

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

```
sum(a1)
```

```
np.int64(6)
```

```
np.sum(a1)
```

```
np.int64(6)
```

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

```
massive_array = np.random.random(100000)
massive_array.size, type(massive_array)
```

```
(100000, numpy.ndarray)
```

```
%timeit sum(massive_array) # Python sum()
%timeit np.sum(massive_array) # NumPy np.sum()
```

```
3.93 ms ± 145 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
20.5 µs ± 698 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.

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

```
(100000, list)
```

```
massive_list[:10]
```

```
[8, 9, 1, 0, 0, 6, 2, 8, 6, 3]
```

```
%timeit sum(massive_list)
%timeit np.sum(massive_list)
```

```
419 µs ± 6.74 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops
each)
2.72 ms ± 118 µs per loop (mean ± 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.

```
a2
```

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

```
# Find the mean
np.mean(a2)
```

```
np.float64(3.6333333333333333)
```

```
# Find the max
np.max(a2)
```

```
np.float64(6.5)
```

```
# Find the min
np.min(a2)
```

```
np.float64(1.0)
```

```
# Find the standard deviation
np.std(a2)
```

```
np.float64(1.8226964152656422)
```

```
# Find the variance
np.var(a2)
```

```
np.float64(3.3222222222222224)
```

```
# The standard deviation is the square root of the variance
np.sqrt(np.var(a2))
```

```
np.float64(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

```
# 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)
```

```
(np.float64(4296133.472222221), np.float64(8.0))
```

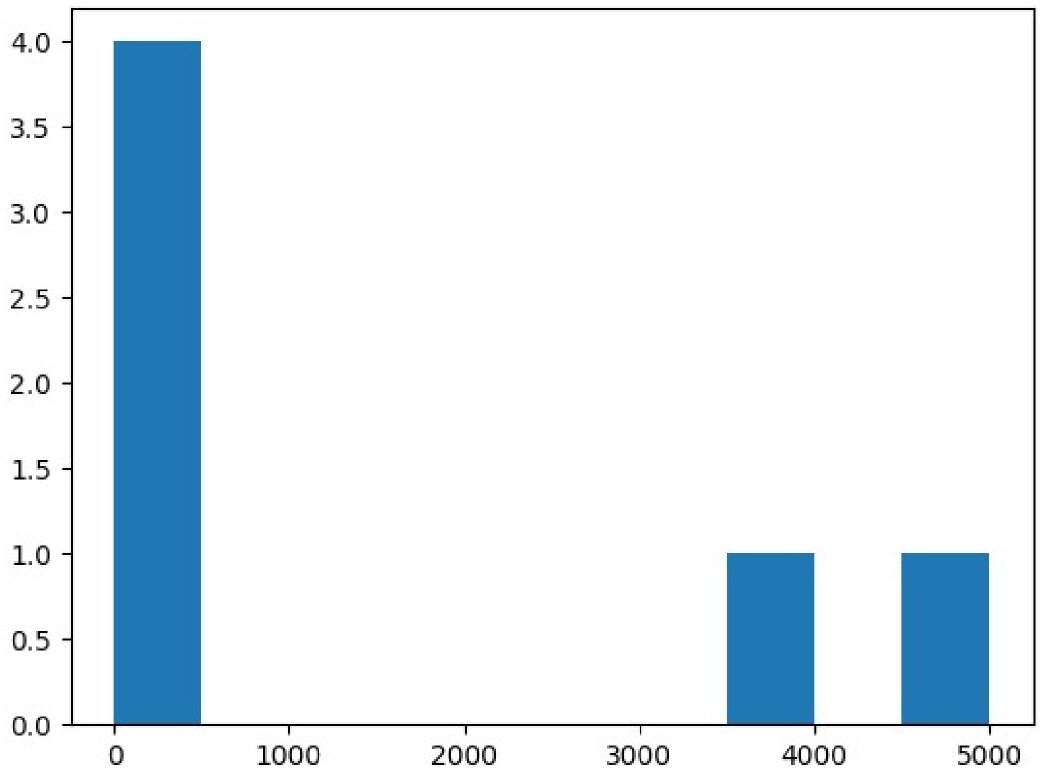
```
np.std(high_var_array), np.std(low_var_array)
```

```
(np.float64(2072.711623024829), np.float64(2.8284271247461903))
```

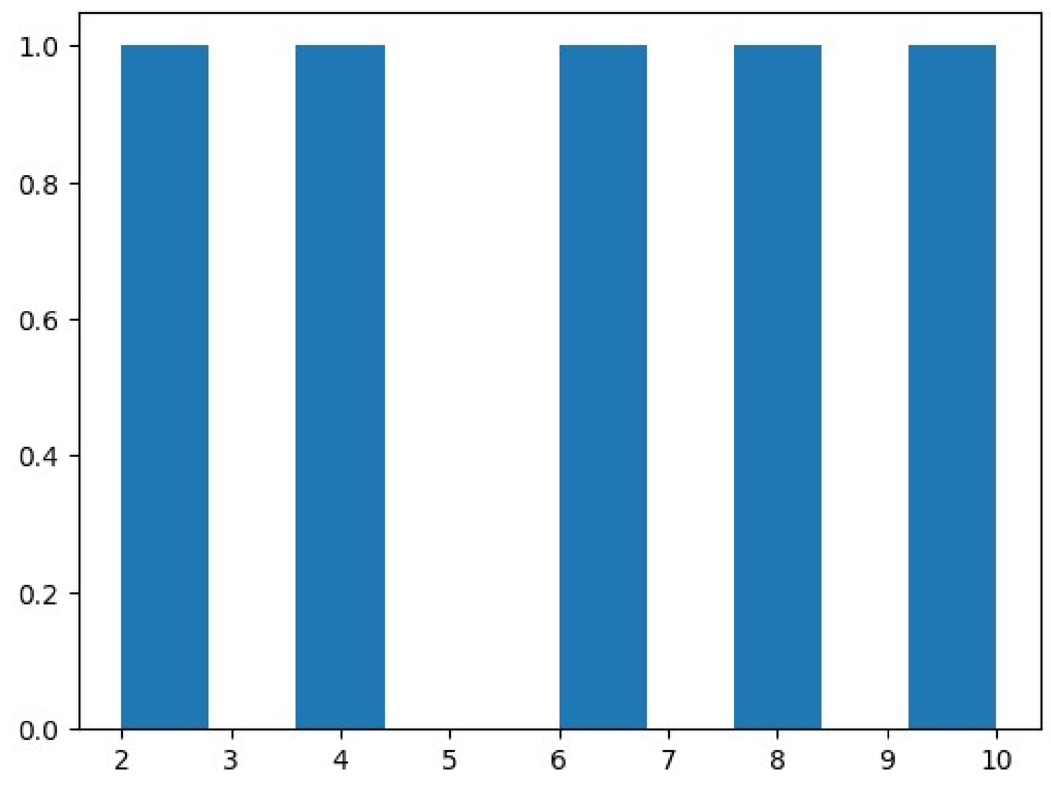
```
# The standard deviation is the square root of the variance
np.sqrt(np.var(high_var_array))
```

```
np.float64(2072.711623024829)
```

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



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



Reshaping

```
a2
```

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

```
a2.shape
```

```
(2, 3)
```

```
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)
```

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

```
a2.reshape(2, 3, 1) + a3
```

```
array([[[ 2. ,  3. ,  4. ],
        [ 6. ,  7. ,  8. ],
        [10.3, 11.3, 12.3]],

       [[14. , 15. , 16. ],
        [18. , 19. , 20. ],
        [22.5, 23.5, 24.5]]])
```

Transpose

A tranpose reverses the order of the axes.

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

```
a2.shape
```

```
(2, 3)
```

```
a2.T
```

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

```
a2.transpose()
```

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

```
a2.T.shape
```

```
(3, 2)
```

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

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

```
matrix = np.random.random(size=(5, 3, 3))
matrix
```

```
array([[[0.59816399, 0.17370251, 0.49752936],
        [0.51231935, 0.41529741, 0.44150892],
        [0.96844105, 0.23242417, 0.90336451]],

       [[0.35172075, 0.56481088, 0.57771134],
        [0.73115238, 0.88762934, 0.37368847],
        [0.35104994, 0.11873224, 0.72324236]],

       [[0.93202688, 0.09600718, 0.4330638 ],
        [0.71979707, 0.06689016, 0.20815443],
        [0.55415679, 0.08416165, 0.88953996]],

       [[0.00301345, 0.30163886, 0.12337636],
        [0.13435611, 0.51987339, 0.05418991],
        [0.11426417, 0.19005404, 0.61364183]],

       [[0.23385887, 0.13555752, 0.32546415],
        [0.81922614, 0.94551446, 0.12975713],
        [0.35431267, 0.37758386, 0.07987885]]])
```

```
matrix.shape
```

```
(5, 3, 3)
```

```
matrix.T
```

```
array([[[0.59816399, 0.35172075, 0.93202688, 0.00301345, 0.23385887],
        [0.51231935, 0.73115238, 0.71979707, 0.13435611, 0.81922614],
        [0.96844105, 0.35104994, 0.55415679, 0.11426417, 0.35431267]],

       [[0.17370251, 0.56481088, 0.09600718, 0.30163886, 0.13555752],
        [0.41529741, 0.88762934, 0.06689016, 0.51987339, 0.94551446],
        [0.23242417, 0.11873224, 0.08416165, 0.19005404, 0.37758386]],

       [[0.49752936, 0.57771134, 0.4330638 , 0.12337636, 0.32546415],
        [0.44150892, 0.37368847, 0.20815443, 0.05418991, 0.12975713],
        [0.90336451, 0.72324236, 0.88953996, 0.61364183,
0.07987885]]])
```

```
matrix.T.shape
```

```
(3, 3, 5)
```

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

```
True
```

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

```
array([[[ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True]],

       [[ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True]],

       [[ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True]]])
```

You can see more advanced forms of tranposing 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:
- `(3, 2) @ (3, 2)` won't work
- `(2, 3) @ (3, 2)` will work
- `(3, 2) @ (2, 3)` will work
1. 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 acheive the same result for 1-2 dimension arrays. However, their behaviour begins to differ at arrays with 3+ dimensions.

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

mat1.shape, mat2.shape
```

```
((3, 3), (3, 2))
```

```
mat1
```

```
array([[5, 0, 3],
       [3, 7, 9],
       [3, 5, 2]])
```

```
mat2
```

```
array([[4, 7],
       [6, 8],
       [8, 1]])
```

```
np.dot(mat1, mat2)
```

```
array([[ 44,  38],
       [126,  86],
       [ 58,  63]])
```

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

```
array([[ 44,  38],
       [126,  86],
       [ 58,  63]])
```

```python
np.random.seed(0)
mat3 = np.random.randint(10, size=(4,3))
mat4 = np.random.randint(10, size=(4,3))
mat3
```

```
array([[5, 0, 3],
       [3, 7, 9],
       [3, 5, 2],
       [4, 7, 6]])
```

```python
mat4
```

```
array([[8, 8, 1],
       [6, 7, 7],
       [8, 1, 5],
       [9, 8, 9]])
```

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

```
---------------------------------------------------------------------------
-----
ValueError                                Traceback (most recent call
last)
Cell In[105], line 2
      1 # This will fail as the inner dimensions of the matrices do
not match
----> 2 np.dot(mat3, mat4)

ValueError: shapes (4,3) and (4,3) not aligned: 3 (dim 1) != 4 (dim 0)
```

```python
mat3.T.shape
```

```
(3, 4)
```

```python
# Dot product
np.dot(mat3.T, mat4)
```

```
array([[118,  96,  77],
       [145, 110, 137],
       [148, 137, 130]])
```

```python
# Element-wise multiplication, also known as Hadamard product
mat3 * mat4
```

```
array([[40,  0,  3],
       [18, 49, 63],
       [24,  5, 10],
       [36, 56, 54]])
```

Dot product practical example, nut butter sales

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

```
array([[12, 15,  0],
       [ 3,  3,  7],
       [ 9, 19, 18],
       [ 4,  6, 12],
       [ 1,  6,  7]])
```

```
weekly_sales = pd.DataFrame(sales_amounts,
                            index=["Mon", "Tues", "Wed", "Thurs", 
"Fri"],
                            columns=["Almond butter", "Peanut butter", 
"Cashew butter"])
weekly_sales
```

```
       Almond butter  Peanut butter  Cashew butter
Mon               12             15              0
Tues               3              3              7
Wed                9             19             18
Thurs              4              6             12
Fri                1              6              7
```

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

```
array([10,  8, 12])
```

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

```
(1, 3)
```

```
weekly_sales.shape
```

```
(5, 3)
```

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

```
----------------------------------------------------------------------
-----
ValueError                                Traceback (most recent call 
last)
Cell In[114], line 2
```

```
      1 # Find the total amount of sales for a whole day
----> 2 total_sales = prices.dot(sales_amounts)
      3 total_sales

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

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

```
prices
```

```
array([10,  8, 12])
```

```
sales_amounts.T.shape
```

```
(3, 5)
```

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

```
array([240, 138, 458, 232, 142])
```

```
butter_prices.shape, weekly_sales.shape
```

```
((1, 3), (5, 3))
```

```
daily_sales = butter_prices.dot(weekly_sales.T)
daily_sales
```

```
       Mon  Tues  Wed  Thurs  Fri
Price  240   138  458    232  142
```

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

```
       Almond butter  Peanut butter  Cashew butter  Total
Mon               12             15              0    240
Tues               3              3              7    138
Wed                9             19             18    458
Thurs              4              6             12    232
Fri                1              6              7    142
```

Comparison operators

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

```
a1
```

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

```
a2

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

a1 > a2

array([[False, False, False],
       [False, False, False]])

a1 >= a2

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

a1 > 5

array([False, False, False])

a1 == a1

array([ True,  True,  True])

a1 == a2

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/indicies which gives the highest value(s) along an axis.
- `np.argmin()` - return the index/indices which gives the lowest value(s) along an axis.

```
random_array

array([[8, 7, 6],
       [4, 2, 7],
       [6, 0, 6],
       [0, 8, 5],
       [6, 2, 9]])

np.sort(random_array)

array([[6, 7, 8],
       [2, 4, 7],
       [0, 6, 6],
       [0, 5, 8],
       [2, 6, 9]])

np.argsort(random_array)
```

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

```
a1
```

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

```
# Return the indices that would sort an array
np.argsort(a1)
```

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

```
# No axis
np.argmin(a1)
```

```
np.int64(0)
```

```
random_array
```

```
array([[8, 7, 6],
       [4, 2, 7],
       [6, 0, 6],
       [0, 8, 5],
       [6, 2, 9]])
```

```
# Down the vertical
np.argmax(random_array, axis=1)
```

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

```
# Across the horizontal
np.argmin(random_array, axis=0)
```

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

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:

```
from matplotlib.image import imread

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

<class 'numpy.ndarray'>

panda.shape

(852, 1280, 3)

panda

array([[[14, 27, 17],
        [14, 27, 17],
        [12, 28, 17],
        ...,
        [42, 36, 24],
        [42, 35, 25],
        [41, 34, 24]],

       [[14, 27, 17],
        [14, 27, 17],
        [12, 28, 17],
        ...,
        [42, 36, 24],
        [42, 35, 25],
        [42, 35, 25]],

       [[13, 26, 16],
        [14, 27, 17],
        [12, 28, 17],
        ...,
        [42, 36, 24],
        [42, 35, 25],
        [42, 35, 25]],

       ...,

       [[47, 32, 27],
        [48, 33, 28],
        [48, 33, 26],
        ...,
        [ 6,  6,  8],
        [ 6,  6,  8],
        [ 6,  6,  8]],

       [[39, 24, 17],
        [40, 25, 18],
        [42, 27, 20],
        ...,
        [ 6,  6,  8],
        [ 6,  6,  8],
        [ 6,  6,  8]],

       [[32, 17, 10],
        [33, 18, 11],
        [36, 21, 14],
        ...,
        [ 6,  6,  8],
        [ 6,  6,  8],
        [ 6,  6,  8]]], dtype=uint8)
```

```
car = imread("../images/numpy-car-photo.png")
car.shape
```

```
(431, 575, 4)
```

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

```
(431, 575, 3)
```

```
dog = imread("../images/numpy-dog-photo.png")
dog.shape
```

```
(432, 575, 4)
```

```
dog
```

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