Introduction to Numpy for absolute beginners!

NumPy (Numerical Python) is an open source Python library that's used in almost every field of science and engineering. It's the universal standard for working with numerical data in Python, and it's at the core of the scientific Python.

The NumPy library provides ndarray, a **homogeneous** n-dimensional array object, with methods to **efficiently** operate on it. NumPy can be used to perform a wide variety of mathematical operations on arrays. It adds powerful data structures to Python that guarantee efficient calculations with arrays and matrices and it supplies an enormous library of high-level mathematical functions that operate on these arrays and matrices.

Installing NumPy

In [1]:
```
pip install numpy
```

```
Requirement already satisfied: numpy in c:\users\asus\anaconda3\lib\site-packages (1.21.5)
Note: you may need to restart the kernel to use updated packages.
```

You might want to consider using **Anaconda** . It's the easiest way to get started. The good thing about getting this distribution is the fact that you don't need to worry too much about separately installing NumPy or any of the major packages that you'll be using for your data analyses, like pandas, Scikit-Learn, etc.

How to import NumPy

To access NumPy and its functions import it in your Python code like this:

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

Central data structure of Numpy

An array is a central data structure of the NumPy library.

An array is a grid of values. The elements are all of the same type, referred to as the array **dtype**.

What’s the difference between a Python list and a NumPy array?

NumPy gives you an enormous range of fast and efficient ways of creating arrays and manipulating numerical data inside them.

- While a Python list can contain **different data types** within a single list, **all of the elements in a NumPy array should be homogeneous**.

The mathematical operations that are meant to be performed on arrays would be extremely inefficient if the arrays weren’t homogeneous.

NumPy arrays are **faster and more compact** than Python lists.

Example

One way we can initialize NumPy arrays is from Python lists, using nested lists for two- or higher-dimensional data.

```
In [11]: a = np.array([1, 2, 3, 4, 5, 6])
```

```
In [10]: b = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

We can access the elements in the array using square brackets. Remember that indexing in NumPy starts at 0.

```
In [7]: print(b[0])
```

```
[1 2 3 4]
```

A vector is an array with a single dimension (there’s no difference between row and column vectors), while a matrix refers to an array with two dimensions.

Attributes of an array

- **ndim** The number of dimensions (axes).
- **shape** The shape of the array is a tuple of integers giving the size of the array along each dimension.
- **size** The total number of elements of the array. This is the product of the elements of the array's shape.

In NumPy, dimensions are called axes. This means that if you have a 2D array, your array has 2 axes.

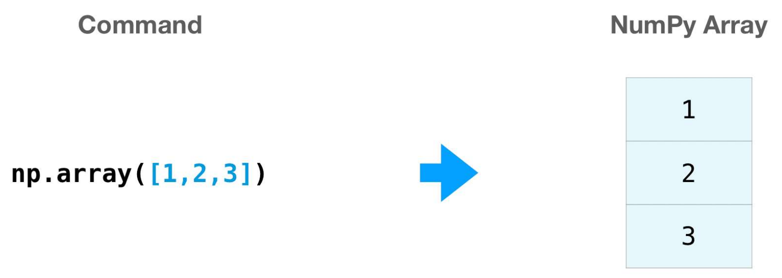
How to create a basic array

several ways, including:

- np.array()
- np.zeros()
- np.ones()
- np.empty()
- np.arange()
- np.linspace()

In [52]:
```
a = np.array([[1, 2, 3],[4,5,6]],dtype='float64')
print(a)
print(a.ndim)
print(a.shape)
print(a.size)
```

```
[[1. 2. 3.]
 [4. 5. 6.]]
2
(2, 3)
6
```



```
create an array filled with 0's:
```

In [15]:
```
np.zeros(2)
```

Out[15]:
```
array([0., 0.])
```

```
create an array filled with 1's:
```

In [19]:
```
np.ones((2,3))
```

Out[19]:
```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

creat an empty array:

Note: The function empty creates an array whose initial content is random and depends on the state of the memory. The reason to use empty over zeros (or something similar) is speed - just make sure to fill every element afterwards!

In [29]:
```
np.empty(2)
```

Out[29]:
```
array([1., 1.])
```

You can create an array with a range of elements:

```
In [22]: np.arange(10)
```

```
Out[22]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

You can create an array that contains a range of evenly spaced intervals:

```
In [24]: np.arange(1.5,10.5,0.5)
```

```
Out[24]: array([ 1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,  5.5,  6. ,  6.5,
                 7. ,  7.5,  8. ,  8.5,  9. ,  9.5, 10. ])
```

You can create an array with values that are spaced linearly in a specified interval:

```
In [26]: np.linspace(0,10,num=5)
```

```
Out[26]: array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

Specifying your data type:

You can explicitly specify which data type you want using the dtype keyword. common used dtypes:

- 'int'
- 'float'
- 'int16'
- 'int32'
- 'int64'
- 'float64'
- str

```
In [51]: np.ones(2, dtype='int')
```

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

adding, removing, sorting elements

sort:

In [68]:
```
a = np.array([[1,4],[3,1]])
print(a)
b = np.sort(a,axis=1)
print(b)
```

```
[[1 4]
 [3 1]]
[[1 4]
 [1 3]]
```

concat:

In [78]:
```
#1d vector concat
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])

np.concatenate((a,b))
```

Out[78]:
```
array([1, 2, 3, 4, 5, 6, 7, 8])
```

In [82]:
```
#matrix concat
x = np.array([[1, 2], [3, 4]])
y = np.array([[5, 6]])
np.concatenate((x,y), axis=0)
```

Out[82]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

or use np.vstack() and np.hstack()

remove

In order to remove elements from an array, it’s simple to use indexing to select the elements that you want to keep. or use np.delete()

In [92]:
```
x = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
np.delete(x,[0],axis=0)
```

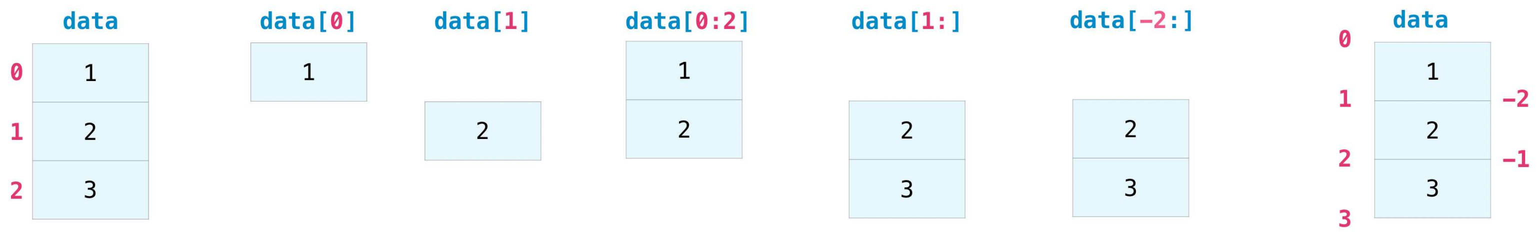
Out[92]:
```
array([[ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Indexing and slicing

```
You can index and slice NumPy arrays in the same ways you can slice Python lists.
```

In [98]:
```
data = np.array([1, 2, 3])
print(data[1])
print(data[0:2])
print(data[1:])
print(data[-2:])
```

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



In [116]:
```
data = np.array([[1, 2], [3, 4], [5, 6]])
print(data)
print(data[0, 1])
print(data[1:3])
print(data[0:2, 0])
```

```
[[1 2]
 [3 4]
 [5 6]]
2
[[3 4]
 [5 6]]
[1 3]
```

Boolean Masking/Indexing

If you want to select values from your array that fulfill certain conditions, it’s straightforward with NumPy.

For example, if you start with this array:

In [101]:
```
a = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
a
```

Out[101]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

You can easily print all of the values in the array that are less than 5:

In [102]:
```
print(a[a < 5])
```

```
[1 2 3 4]
```

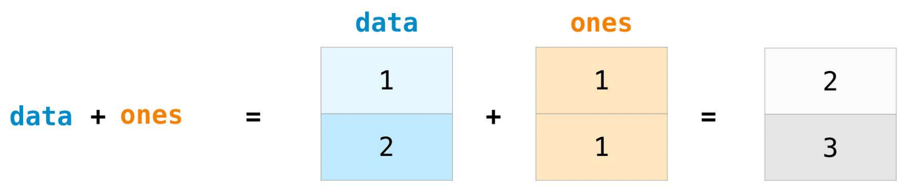
In [103]:
```
c = a[(a > 2) & (a < 11)]
print(c)
```

```
[ 3  4  5  6  7  8  9 10]
```

Basic array operations

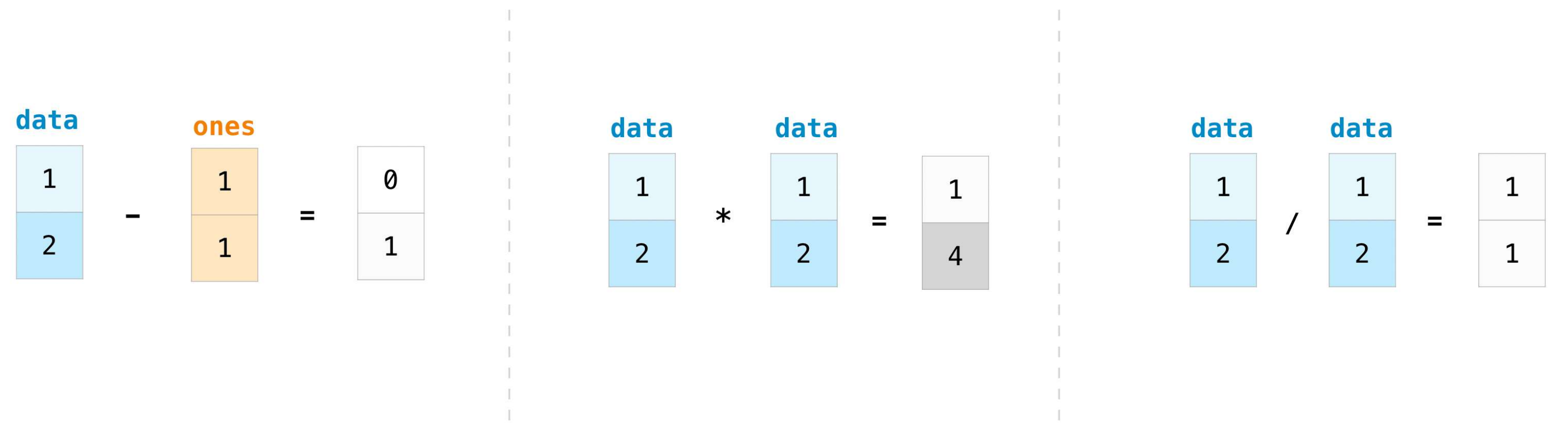
In [104]:
```
data = np.array([1, 2])
ones = np.ones(2, dtype=int)
data + ones
```

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



In [105]:
```
print(data - ones)
print(data * data)
print(data / data)
```

```
[0 1]
[1 4]
[1. 1.]
```



Can you reshape an array?

Using arr.reshape() will give a new shape to an array without changing the data. Just remember that when you use the reshape method, the array you want to produce needs to have the same number of elements as the original array. If you start with an array with 12 elements, you'll need to make sure that your new array also has a total of 12 elements.

In [93]:
```
a = np.arange(6)
print(a)
b = a.reshape(3, 2)
print(b)
```

```
[0 1 2 3 4 5]
[[0 1]
 [2 3]
 [4 5]]
```

How to convert a 1D array into a 2D array

Using np.newaxis will increase the dimensions of your array by one dimension when used once. This means that a 1D array will become a 2D array, a 2D array will become a 3D array, and so on.

For example, if you start with this array:

In [97]:
```python
a = np.array([1, 2, 3, 4, 5, 6])
print(a.shape)
b = a[:, np.newaxis]
print(b.shape)
c = a[np.newaxis,:]
print(c.shape)
```

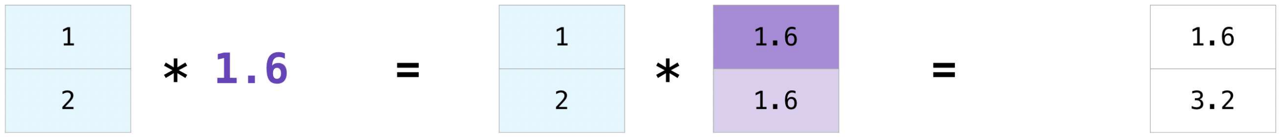
```
(6,)
(6, 1)
(1, 6)
```

Broadcasting

Broadcasting is a mechanism that allows NumPy to perform operations on arrays of different shapes. The dimensions of your array must be compatible, for example, when the dimensions of both arrays are equal or when one of them is 1. If the dimensions are not compatible, you will get a ValueError.

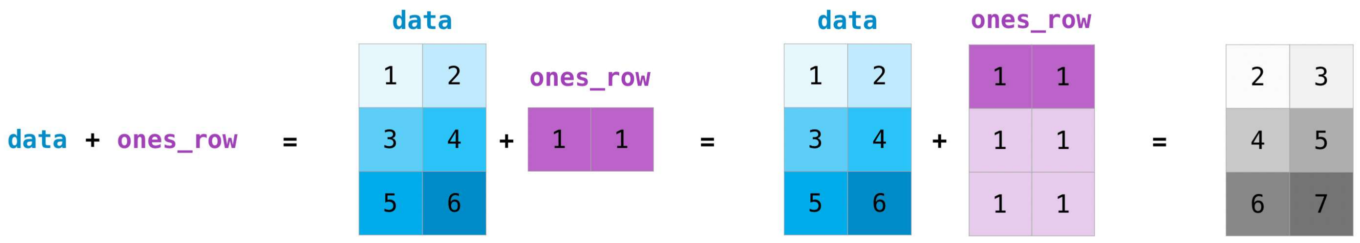
In [106]:
```python
data = np.array([1.0, 2.0])
data * 1.6
```

Out[106]:
```
array([1.6, 3.2])
```



In [121]:
```python
data = np.array([[1, 2], [3, 4], [5, 6]])
ones_row = np.array([[1, 1]])
data + ones_row
```

Out[121]:
```
array([[2, 3],
       [4, 5],
       [6, 7]])
```

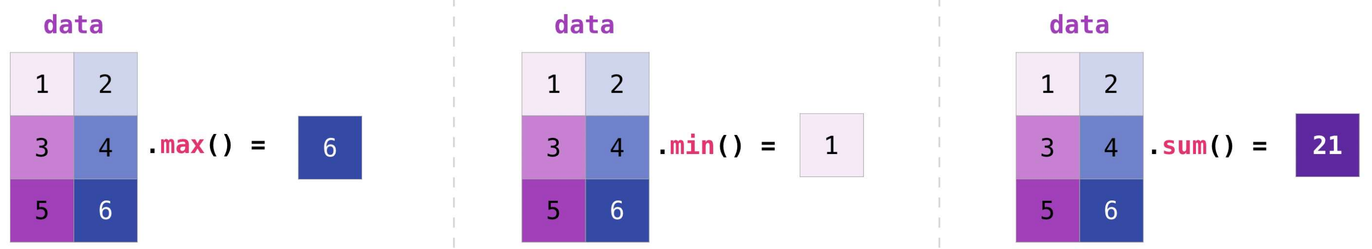


Useful operations

max, min, sum along axes

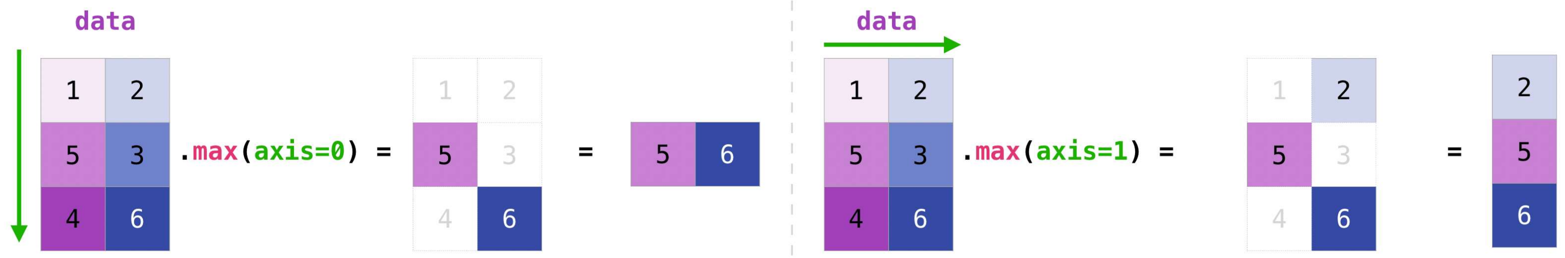
In [117]:
```python
data = np.array([[1, 2], [3, 4], [5, 6]])
data.max()
data.min()
data.sum()
```

Out[117]:
```
21
```



```
In [120]: print(data.max(axis=0))
          print(data.max(axis=1))
```

```
[5 6]
[2 4 6]
```



Transposing & reversing

You can transpose your array with arr.transpose() or arr.T. NumPy’s np.flip() function allows you to flip, or reverse, the contents of an array along an axis.

In [123]:
```
arr = np.arange(6).reshape((2, 3))
print(arr)
print(arr.transpose())
print(arr.T)
```

```
[[0 1 2]
 [3 4 5]]
[[0 3]
 [1 4]
 [2 5]]
[[0 3]
 [1 4]
 [2 5]]
```

In [126]:
```
arr_2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
np.flip(arr_2d, axis=0)
```

Out[126]:
```
array([[ 9, 10, 11, 12],
       [ 5,  6,  7,  8],
       [ 1,  2,  3,  4]])
```

An example: Working with mathematical formulas

The ease of implementing mathematical formulas that work on arrays is one of the things that make NumPy so widely used in the scientific Python community.

For example, this is the mean square error formula (a central formula used in supervised machine learning models that deal with regression):

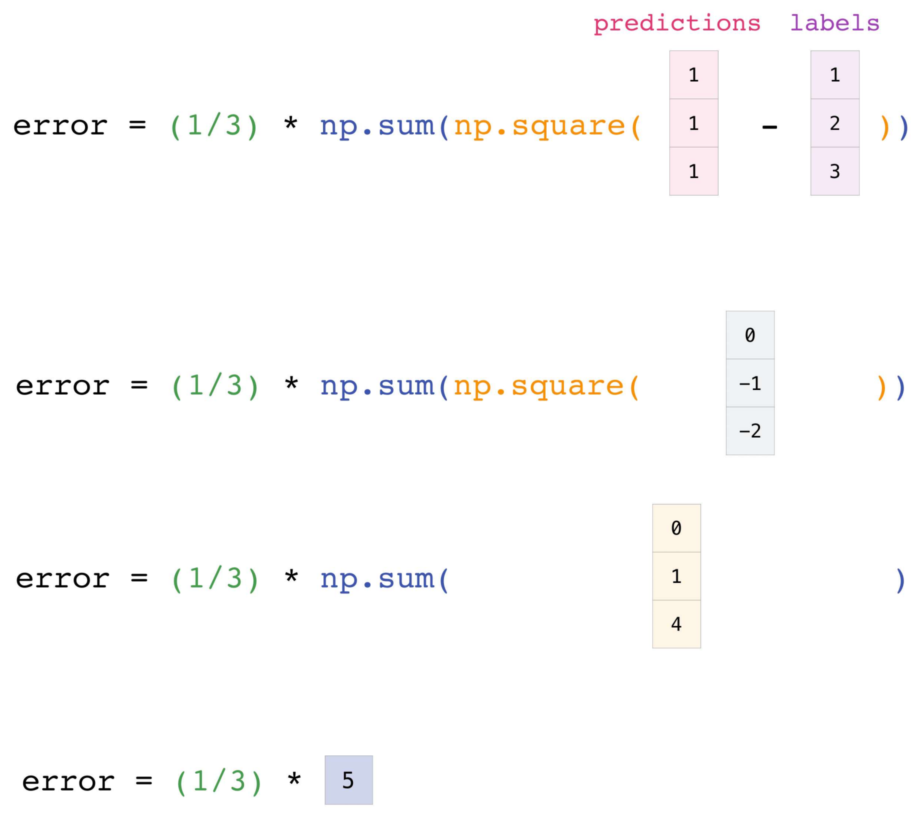
$$MeanSquareError = \frac{1}{n}\sum_{i=1}^{n}(Y\_prediction_i - Y_i)^2$$

Implementing this formula is simple and straightforward in NumPy:

```
error = (1/n) * np.sum(np.square(predictions - labels))
```

What makes this work so well is that predictions and labels can contain one or a thousand values. They only need to be the same size.

You can visualize it this way:


predictions
labels
error = (1/3) * np.sum(np.square(1 1 1 - 1 2 3))
error = (1/3) * np.sum(np.square(0 -1 -2))
error = (1/3) * np.sum(0 1 4)
error = (1/3) * 5