7. Numpy

Chapter Summary

- Introduction
- Data type, operator, comparison
- Reshape, flatten, T, concatenate, split, vsplit
- Append, insert, delete, unique
- Rounding decimal
- Random, randint, rand, uniform, ndenumerate
- Scalar, vector and matrix
- Broadcasting
- Universal Function
- Filter, Conditional Statement with np.where

What is NumPy

- NumPy : Numerical Python
- Open source project for computing in Python
- Widely used in python programming
- Extended to Pandas, SciPy, Matplotlib, scikit-learn, scikit-image and most other data science and scientific Python packages

Installation

- pip install numpy

Check version installed

- numpy.__version__

Create a simple 1-D array

In [3]:

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

Out[3]:

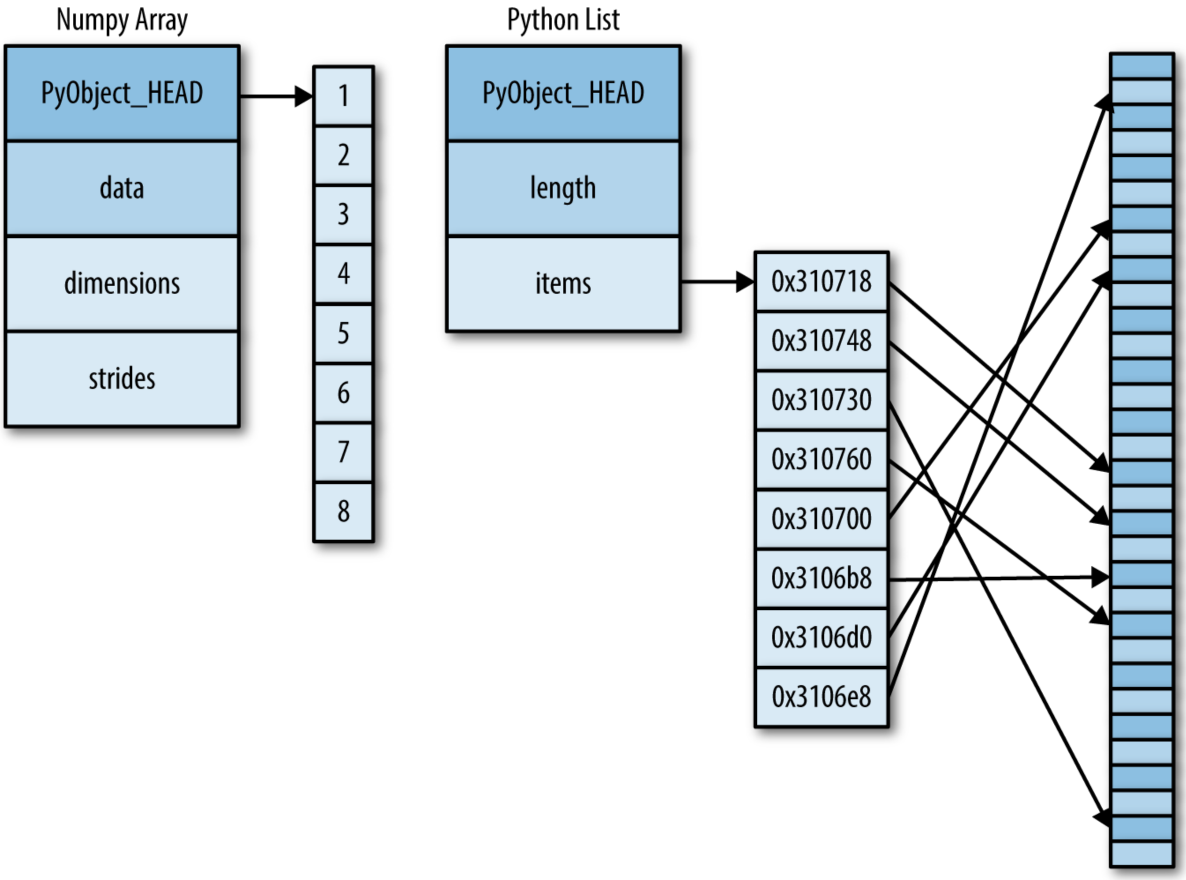
```
array([1, 2, 3])
```

In [4]:

```
type(arr)
```

Out[4]:

```
numpy.ndarray
```



Numpy array

In [4]:

```python
arr = np.array([1, 2, 3])
print(arr)
type(arr)
```

```
[1 2 3]
```

Out[4]:

```
numpy.ndarray
```

Python list

In [3]:

```python
python_list = [1,2,3]
print(python_list)
type(python_list)
```

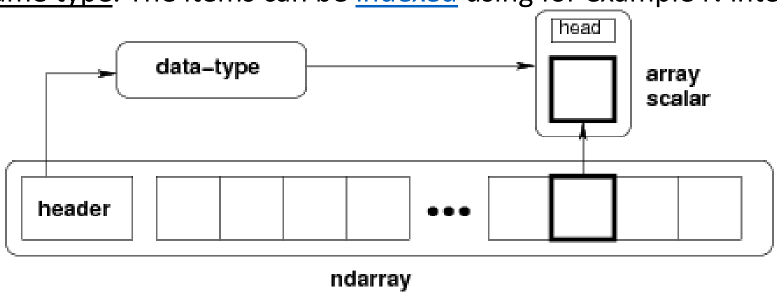
```
[1, 2, 3]
```

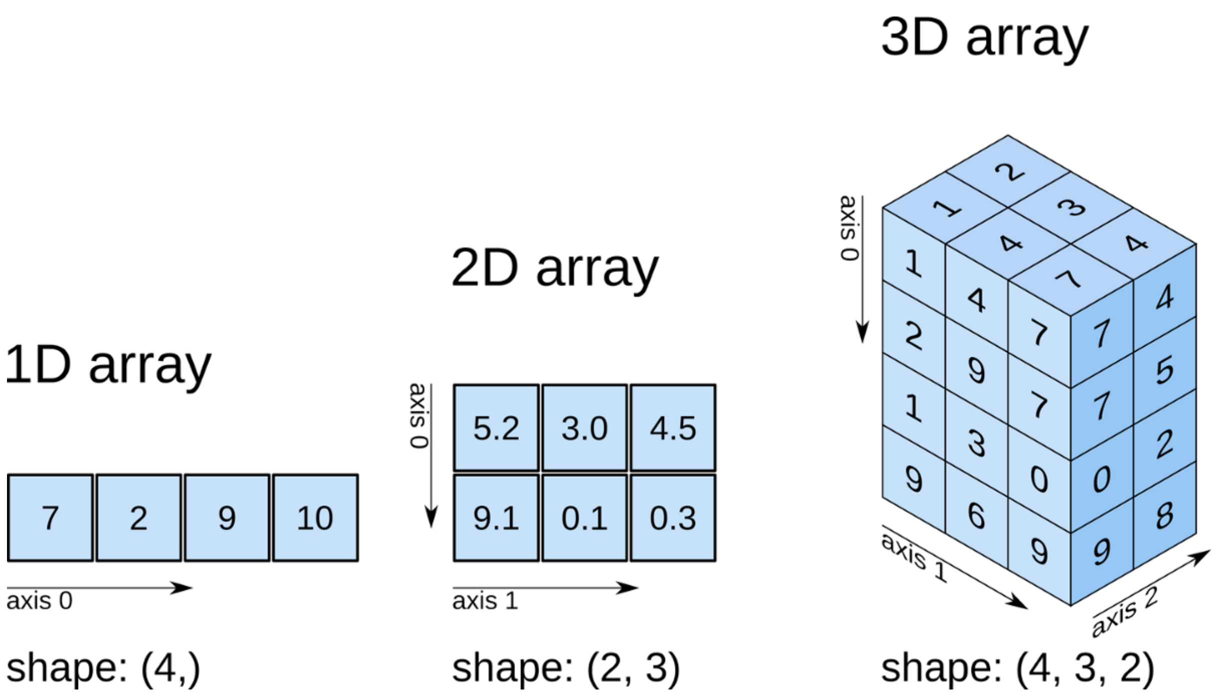
Out[3]:

```
list
```

Numpy Array Object

NumPy provides an N-dimensional array type, the ndarray, which describes a collection of "items" of the same type. The items can be indexed using for example N integers.





Create a 2-D array

In [5]:

```
arr_2D = np.array([[1,2],[3,4],[5,6],[7,8]])
print(arr_2D)
arr_2D.shape
```

```
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

Out[5]:

```
(4, 2)
```

Access the array value

In [6]:

```
arr_2D = np.array([[1,2],[3,4],[5,6],[7,8]])
print(arr_2D[1])     # [3 4]
print(arr_2D[3][0]) # 7
print(arr_2D[:2])
```

```
[3 4]
7
[[1 2]
 [3 4]]
```

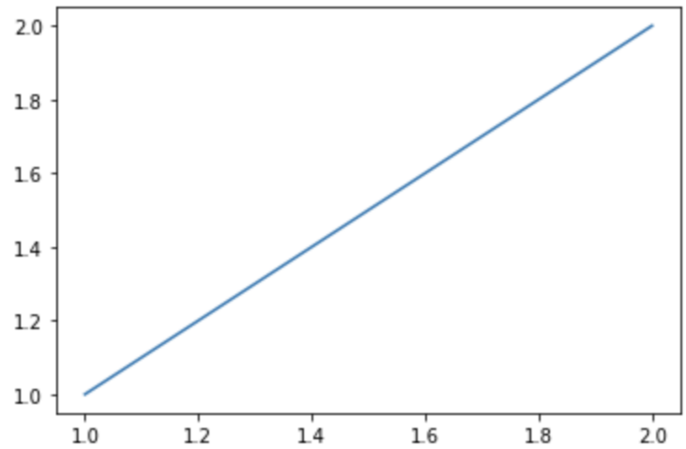
Slicing the array

In [8]:
```
arr_2D = np.array([[1,2],[3,4],[5,6],[7,8]])
print(arr_2D[:,0])
print(arr_2D[:,1])

import matplotlib.pyplot as plt
plt.plot(arr_2D[0][:], arr_2D[:][0])
```

```
[1 3 5 7]
[2 4 6 8]
```

Out[8]:
```
[<matplotlib.lines.Line2D at 0x116f92700>]
```



Reshape an existing array

In [8]:
```
data = np.array([1,2,3,4,5,6])
data.shape
```

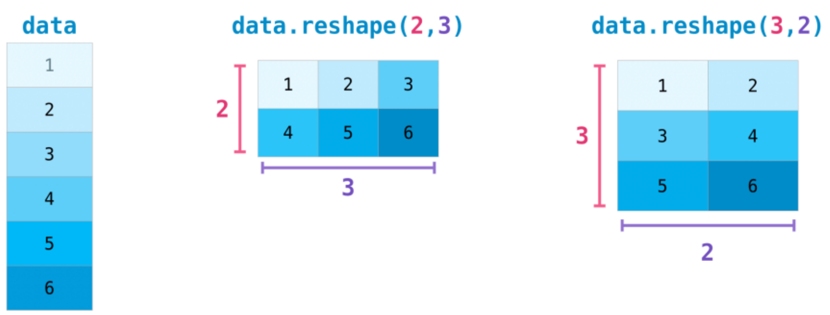
Out[8]:
```
(6,)
```

In [9]:
```
data.reshape(2,3)
```

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

In [10]:
```
data.reshape(3,2)
```

Out[10]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```



Create a 3D array

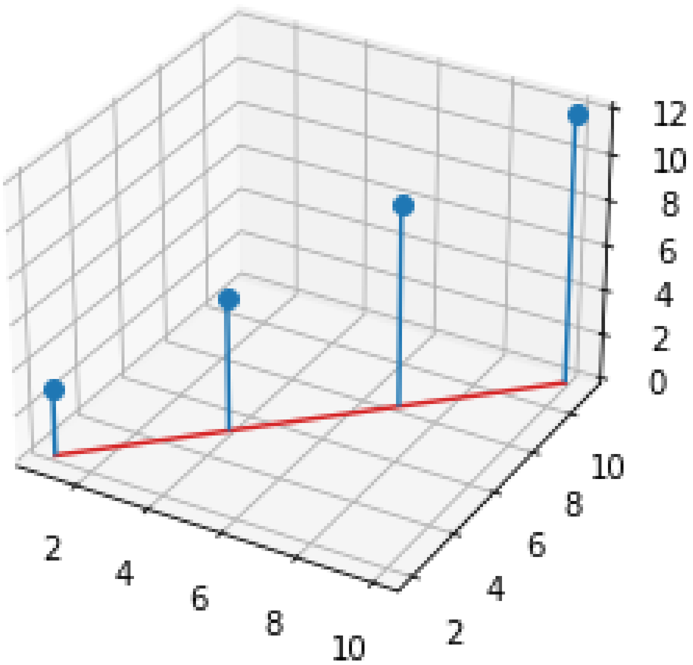
```
arr_3D = np.array([[[1,2,3],
                    [4,5,6]],
                   [[7,8,9],
                    [10,11,12]],
                   [[1,2,3],
                    [4,5,6]],
                  ])
arr_3D.shape
```

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

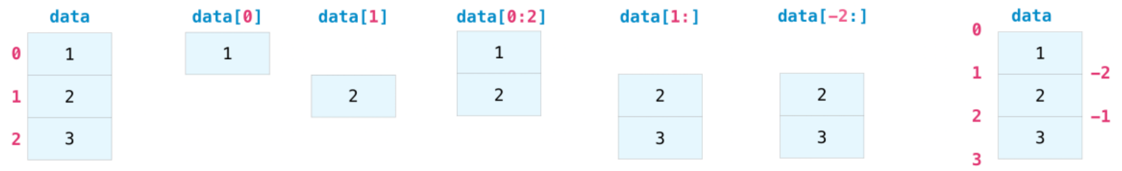
```
arr_3D = np.array([[1,2,3],
                   [4,5,6],
                   [7,8,9],
                   [10,11,12]])
```

```
X = arr_3D[:,0]  # 1,4,7,10
Y = arr_3D[:,1]  # 2,5,8,11
Z = arr_3D[:,2]  # 3,6,9,12

from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
fig, ax = plt.subplots(subplot_kw=dict(projection='3d'))
ax.stem(X, Y, Z)
plt.show()
```



1D Array Indexing and Slicing

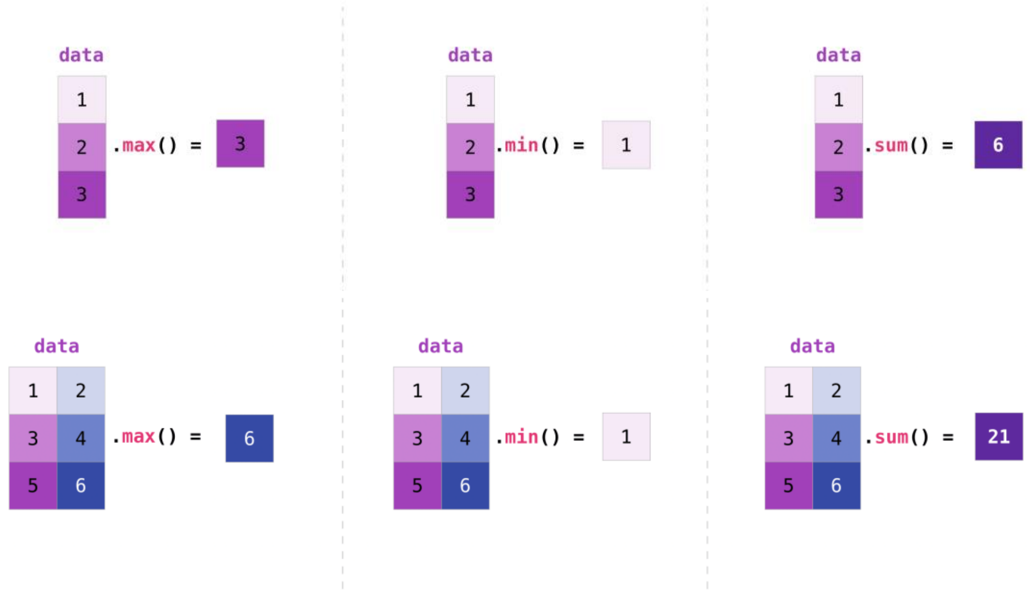


In [4]:

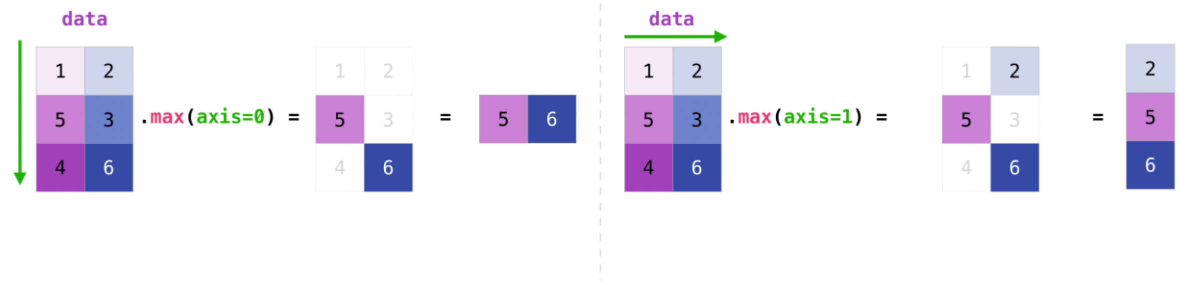
```
arr2 = np.array([1, 2, 3])
print(arr2[0])
print(arr2[1])
print(arr2[-1])
print(arr2[:])
print(arr2[1:])
print(arr2[:2])
```

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

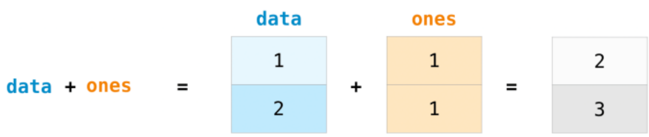
Array Operation – max, min, sum

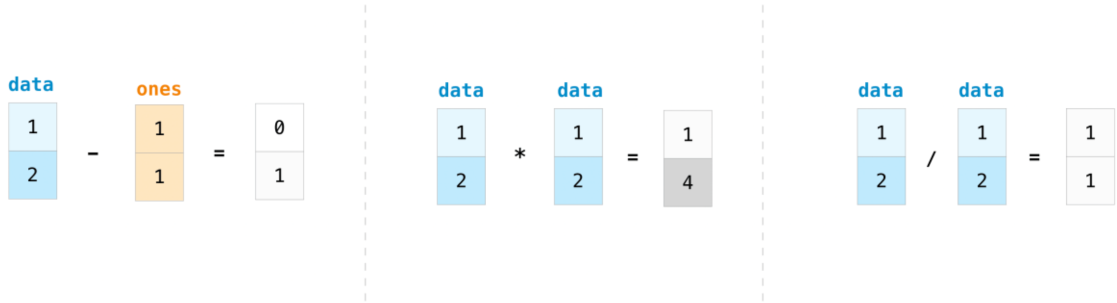


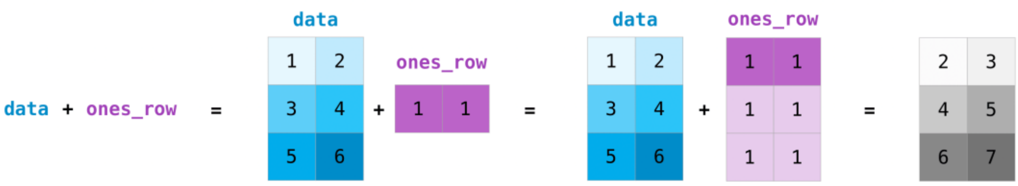
Array Operation – max, min, sum in axis



Array Operation







Standard Numpy Datatype

Data type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (–128 to 127)
int16	Integer (–32768 to 32767)
int32	Integer (–2147483648 to 2147483647)
int64	Integer (–9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64
float16	Half-precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single-precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double-precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128
complex64	Complex number, represented by two 32-bit floats
complex128	Complex number, represented by two 64-bit floats

Computation on NumPy arrays can be very fast, or it can be very slow. The key to making it fast is to use *vectorized* operations, generally implemented through Numpy's *universal functions* (ufuncs).

A universal function (or ufunc for short) is a function that operates on ndarrays in an element-by-element fashion, supporting array broadcasting, type casting, and several other standard features.

Common used functions

- np.linsspace
- np.arrange
- np.random.randint
- np.zeros
- np.ones

```
In [31]: # generate evenly spaced numbers over a specified interval
         np.linspace(0,2,9) # (smallest number, largest number, total number)

Out[31]: array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])

In [32]: # generate arrays with regularly incrementing values
         np.arange(10)

Out[32]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [33]: # np.arange(start, stop, interval)
         np.arange(9, 10, 0.1)

Out[33]: array([9. , 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9])

In [34]: # generate random integer (smallest integer, largest integer, total number)
         np.random.randint(1, 100, size=10)

Out[34]: array([87, 68, 82, 97, 87, 53, 47, 30, 34, 91])

In [35]: # create array of all zero
         np.zeros((2,3))

Out[35]: array([[0., 0., 0.],
                [0., 0., 0.]])

In [38]: # create array of all one
         np.ones((3,2))

Out[38]: array([[1., 1.],
                [1., 1.],
                [1., 1.]])
```

Arithmetic operators and functions

Operator	Equivalent ufunc	Description
+	np.add	Addition (e.g., 1 + 1 = 2)
-	np.subtract	Subtraction (e.g., 3 - 2 = 1)
-	np.negative	Unary negation (e.g., -2)
*	np.multiply	Multiplication (e.g., 2 * 3 = 6)
/	np.divide	Division (e.g., 3 / 2 = 1.5)
//	np.floor_divide	Floor division (e.g., 3 // 2 = 1)
**	np.power	Exponentiation (e.g., 2 ** 3 = 8)
%	np.mod	Modulus/remainder (e.g., 9 % 4 = 1)

Try the following functions in your iPython

- //
- Mod
- **
- np.sqrt()
- np.log10()
- np.log()
- np.exp()
- np.around()
- np.ceil()
- np.floor()
- np.rint()

```
In [37]: # floor divide
         7//2
Out[37]: 3

In [38]: # modulas, remainder
         np.mod(7,2)
Out[38]: 1

In [39]: # power
         2**3
Out[39]: 8

In [40]: # square root
         np.sqrt(9)
Out[40]: 3.0

In [41]: # 10 logarithm of the input
         np.log10(1000)
Out[41]: 3.0

In [42]: # natural logarithm, element-wise.
         np.log(2.718281828459045)
Out[42]: 1.0

In [43]: # exponential of all elements in the input
         np.exp(1)
Out[43]: 2.718281828459045

In [44]: np.around(3.14159265359, decimals=2, out=None)
Out[44]: 3.14

In [45]: # Return the ceiling of the input, element-wise
         np.ceil([3.1, 3.9, 3])
Out[45]: array([4., 4., 3.])

In [46]: # Return the floor of the input, element-wise
         np.floor([3.9, 3.2, 3])
Out[46]: array([3., 3., 3.])
```

Common Aggregate Functions

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute median of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

In [37]:
```
arr = np.arange(0,10,0.5)
arr
```
Out[37]:
```
array([0. , 0.5, 1. , 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])
```

In [38]:
```
arr.size
```
Out[38]:
```
20
```

In [39]:
```
arr.sum()
```
Out[39]:
```
95.0
```

In [40]:
```
arr.min()
```
Out[40]:
```
0.0
```

In [41]:
```
arr.max()
```
Out[41]:
```
9.5
```

In [42]:
```
arr.mean()
```
Out[42]:
```
4.75
```

In [43]:
```
np.median(arr)
```
Out[43]:
```
4.75
```

In [44]:
```
# Standard Deviation
np.std(arr)
```
Out[44]:
```
2.883140648667699
```

Data type structure in array

Character	Description	Example
'b'	Byte	np.dtype('b')
'i'	Signed integer	np.dtype('i4') == np.int32
'u'	Unsigned integer	np.dtype('u1') == np.uint8
'f'	Floating point	np.dtype('f8') == np.int64
'c'	Complex floating point	np.dtype('c16') == np.complex128
'S', 'a'	string	np.dtype('S5')
'U'	Unicode string	np.dtype('U') == np.str_
'V'	Raw data (void)	np.dtype('V') == np.void

```
# create an empty array with settings
client = np.array([('Alice','F',30,3500.5),
    ('Ben','M',35,2800.3),],dtype=[('name', 'U10'), ('gender', 'U1'), ('age', 'i4'), ('order_amount', 'f16')])
client
```

```
array([('Alice', 'F', 30, 3500.5), ('Ben', 'M', 35, 2800.3)],
      dtype=[('name', '<U10'), ('gender', '<U1'), ('age', '<i4'), ('order_amount', '<f16')])
```

```
client[0]
```

```
('Alice', 'F', 30, 3500.5)
```

```
type(client)
```

```
numpy.ndarray
```

```
new_client = client.copy()
new_client
```

```
array([('Alice', 'F', 30, 3500.5), ('Ben', 'M', 35, 2800.3)],
      dtype=[('name', '<U10'), ('gender', '<U1'), ('age', '<i4'), ('order_amount', '<f16')])
```

```
new_client[0]
```

```
('Alice', 'F', 30, 3500.5)
```

```
new_client[0][2]=18
```

```
new_client[0]
```

```
('Alice', 'F', 18, 3500.5)
```

Comparison Operator

Operator	Equivalent ufunc
==	np.equal
!=	np.not_equal
<	np.less
<=	np.less_equal
>	np.greater
>=	np.greater_equal

```
In [63]: arr = np.array([1, 2, 3, 4, 5])
         arr < 3

Out[63]: array([ True,  True, False, False, False])

In [64]: arr >= 3

Out[64]: array([False, False,  True,  True,  True])

In [65]: arr != 3

Out[65]: array([ True,  True, False,  True,  True])

In [66]: arr == 3

Out[66]: array([False, False,  True, False, False])

In [67]: len(arr)

Out[67]: 5

In [68]: np.count_nonzero(arr>=2)

Out[68]: 4
```

Array Manipulation

```
In [72]: arr = np.zeros([1,2])
         arr
```

```
Out[72]: array([[0., 0.]])
```

```
In [73]: # Append values to the end of an array
         # numpy.append(arr, values, axis=None)
         arr = np.append(arr, [[2,2]], axis=0)
         arr
```

```
Out[73]: array([[0., 0.],
                [2., 2.]])
```

```
In [74]: arr = arr.flatten()
         arr
```

```
Out[74]: array([0., 0., 2., 2.])
```

```
In [75]: arr = arr.reshape([2,2])
         arr
```

```
Out[75]: array([[0., 0.],
                [2., 2.]])
```

```
In [76]: # Insert values along the given axis before the given indices.
         # numpy.insert(arr, obj, values, axis=None) , axis=0 means insert by row
         arr = np.insert(arr, [1], [1,1], axis=0)
         arr
```

```
Out[76]: array([[0., 0.],
                [1., 1.],
                [2., 2.]])
```

```
In [77]: # axis=0 means insert by column
         arr = np.insert(arr, [1], [3], axis=1)
         arr
```

```
Out[77]: array([[0., 3., 0.],
                [1., 3., 1.],
                [2., 3., 2.]])
```

```
In [78]: # Join a sequence of arrays along an existing axis.
         # numpy.concatenate((a1, a2, ...), axis=0, out=None, dtype=None, casting="same_kind")
         arr2 = np.array([[3,3,3]])
         arr3 = np.concatenate((arr,arr2), axis=0)
         arr3
```

```
Out[78]: array([[0., 3., 0.],
                [1., 3., 1.],
                [2., 3., 2.],
                [3., 3., 3.]])
```

NumPy Array conversion

```
In [69]: # Python list to np.array
         arr = np.array([1,2,3,4,5])
         type(arr)
```

```
Out[69]: numpy.ndarray
```

```
In [70]: # np.array to Python list
         p_list = arr.tolist()
         type(p_list)
```

```
Out[70]: list
```

```
In [71]: # Tuple to np.array
         arr1 = np.array((1,2,3,4,5))
         type(arr1)
```

```
Out[71]: numpy.ndarray
```

```
In [72]: p_tuple = tuple(arr1)
         type(p_tuple)
```

```
Out[72]: tuple
```

Scalar, Vector and Matrix

Scalar is a magnitude or a numerical value.

- Example: mass, speed, distance, time, energy, density, volume, temperature, distance

Vector is a magnitude and a direction.

- Example: displacement, acceleration, force, momentum, weight, the velocity of light, a gravitational field, current

```python
import numpy as np
# Scalar
s = 1
```

```python
# Vector
v1 = np.array([1,4,5])  # horizontal
v2 = np.array([[1],
               [-5]])   # vetical vector
print("v1 shape: ", v1.shape)
print("v2 shape: ", v2.shape)
```

```
v1 shape:  (3,)
v2 shape:  (2, 1)
```

A matrix is a two-dimensional data structure where numbers are arranged into rows and columns. There is no data type for matrix in Python. It is defined by nested list or array.

```python
# Matrice
m = np.array([[1, 4, 5],
              [-5, 8, 9]])
m
```

```
array([[ 1,  4,  5],
       [-5,  8,  9]])
```

```python
m.shape
```

```
(2, 3)
```

np.transpose

np.T returns an array with axes transposed. Same as np.transpose.

```python
np.arange(1,7).reshape(3,2)
```

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

```python
np.arange(1,7).reshape(3,2).T
```

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

np.flatten( )

Return a copy of the array collapsed into one dimension.

```
np.arange(1,7).reshape(3,2)
```

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

```
np.arange(1,7).reshape(3,2).flatten()
```

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

np.concatenate numpy array

numpy.concatenate(*(a1, a2, ...)*, *axis=0*). Join a sequence of arrays along an existing axis.

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), axis=0)
```

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

```
np.concatenate((a, b.T), axis=1)
```

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

```
np.concatenate((a, b), axis=None)
```

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

np.split numpy array

np.split(*ary*, *indices_or_sections*, *axis=0*). Split an array into multiple sub-arrays as views into *array.*

```
x = np.arange(9)
x
```

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

```
arr1, arr2, arr3 = np.split(x, 3)
arr1
```

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

```
np.split(x, [0,6])
```

```
[array([], dtype=int64), array([0, 1, 2, 3, 4, 5]), array([6, 7, 8])]
```

```
np.split(x, [0,6])[1]
```

```
array([0, 1, 2, 3, 4, 5])
```

```
np.split(x, [0,6])[2]
```

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

np.append numpy array

np.append(*arr*, *values*, *axis=None*). Append values to the end of an array.

```
x = np.arange(1,7).reshape(2,3)
x
```

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

```
x = np.append(x, [[7,8,9]], axis=0)
x
```

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

```
np.append(x, [[3.5],[6.5],[9.5]], axis=1)
```

```
array([[1. , 2. , 3. , 3.5],
       [4. , 5. , 6. , 6.5],
       [7. , 8. , 9. , 9.5]])
```

np.insert numpy array

np.insert(*arr*, *obj*, *values*, *axis=None).* Insert values along the given axis before the given indices.

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

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

```
np.insert(a, [1], [[9],[9],[9]], axis=1)
```

```
array([[1, 9, 1],
       [2, 9, 2],
       [3, 9, 3]])
```

```
np.insert(a, 1, 5)
```

```
array([1, 5, 1, 2, 2, 3, 3])
```

Rounding decimal

There are primarily five ways of rounding off decimals in NumPy:

- truncation
- fix (same as truncation)
- rounding
- floor
- ceil

```
arr = np.array([-1.99, -1.13, 0.56, 1.28, 2.10, 2.65])
```

```
# truncate decimal
np.trunc(arr)
```

```
array([-1., -1.,  0.,  1.,  2.,  2.,  3.,  3.])
```

```
# Round to nearest integer towards zero.
np.fix(arr)
```

```
array([-1., -1.,  0.,  1.,  2.,  2.,  3.,  3.])
```

```
# round off to n decimal point
np.round(arr, 1)
```

```
array([-1.9, -1.1,  0.5,  1.2,  2.1,  2.6,  3.2,  3.8])
```

```
np.ceil(arr)
```

```
array([-1., -1.,  1.,  2.,  3.,  3.,  4.,  4.])
```

```
np.floor(arr)
```

```
array([-2., -2.,  0.,  1.,  2.,  2.,  3.,  3.])
```

Random Numbers

Random number does NOT mean a different number every time. Random means something that cannot be predicted logically. Random is crucial topic in data science and machine learning.

Application example:

- Cryptography
- Cryptocurrency wallets
- Simulations
- Machine learning
- Scientific studies

random.**randint**(*low*, *high=None*, *size=None*, *dtype=int*)

- Return random integers from *low* (inclusive) to *high* (exclusive).
- Return random integers from the "discrete uniform" distribution of the specified dtype in the "half-open" interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

```
from numpy import random
# Generate 10 ints between 0 and 99, inclusive
random.randint(100 , size=(10))
```

```
array([44,  2, 49, 22, 32, 39, 58, 65, 33, 72])
```

```
#Generate a 2 x 4 array of ints between 1 and 8, inclusive
np.random.randint(1,9, size=(2, 4))
```

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

random.**rand**(*d0*, *d1*, ..., *dn*)Random values in a given shape.

Create an array of the given shape and populate it with random samples from a uniform distribution over [0, 1].

```
random.rand(5)
```

```
array([0.8011264 , 0.25988628, 0.90796268, 0.7302317 , 0.26139519])
```

```
random.rand(4,5)
```

```
array([[0.3452997 , 0.84728647, 0.85974315, 0.70047975, 0.69632204],
       [0.07339232, 0.08965751, 0.93790906, 0.02398837, 0.56169819],
       [0.71365956, 0.62741451, 0.29302985, 0.65682291, 0.10798099],
       [0.31750426, 0.52681618, 0.37137185, 0.88634499, 0.13717501]])
```

random.choice

random.**choice**(*a*, *size=None*, *replace=True*, *p=None*). Generates a random sample from a given 1-D array

```
# choice of 1-29, pick 10 random, repeated number allowed
random.choice(np.arange(1,30), 10, replace=True)
```

```
array([19,  5, 20, 11,  1, 26, 10,  5,  1, 25])
```

```
random.choice(["A","B","C","D"], 2, replace=False)
```

```
array(['C', 'D'], dtype='<U1')
```

```
# pick 6 random mark 6 number
mark6 = random.choice(np.arange(1,50), 6, replace=False)
sorted(mark6)
```

```
[10, 11, 14, 18, 22, 44]
```

random.uniform

random.**uniform**(*low=0.0*, *high=1.0*, *size=None*). Draw samples from a uniform distribution. Samples are uniformly distributed over the half-open interval [low, high] (includes low, but excludes high).

```
# random number range -1 to +1 , size: 20
s = np.random.uniform(-1,1,20)
s
```

```
array([-0.68894514,  0.08310177, -0.85816642, -0.41840843,  0.3839412 ,
       -0.88159189, -0.03724022, -0.44483262,  0.20934449,  0.04664035,
       -0.3505514 ,  0.23609872, -0.98573827, -0.54794272, -0.442871  ,
        0.82176829,  0.02372383,  0.13043628,  0.38517956,  0.01710117])
```

ndenumerate numpy array

numpy.**ndenumerate**(*arr*). Multidimensional index iterator.

Return an iterator yielding pairs of array coordinates and values.

```
a = np.array([[1, 2], [3, 4]])
for index, x in np.ndenumerate(a):
    print(index, x)
```

```
(0, 0) 1
(0, 1) 2
(1, 0) 3
(1, 1) 4
```

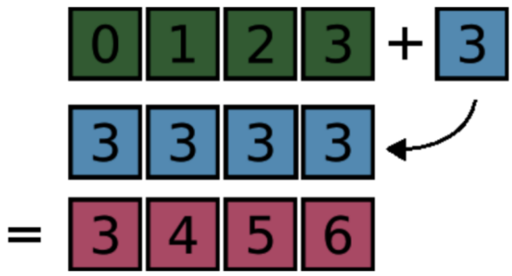
Broadcasting

The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations.

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimension and works its way left. Two dimensions are compatible when

1. they are equal, or
2. one of them is 1.

Broadcasting – 1D



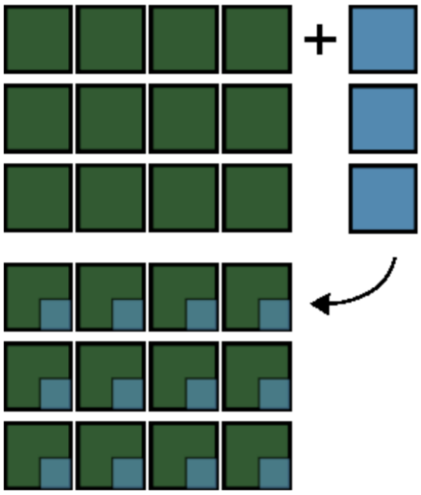

```
x = np.array([0,1,2,3])
x + np.array([3,3,3,3])
```

```
array([3, 4, 5, 6])
```

```
x = np.array([0,1,2,3])
x+3
```

```
array([3, 4, 5, 6])
```

Broadcasting – 2D



```
a = np.arange(12).reshape((3,4))
a
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

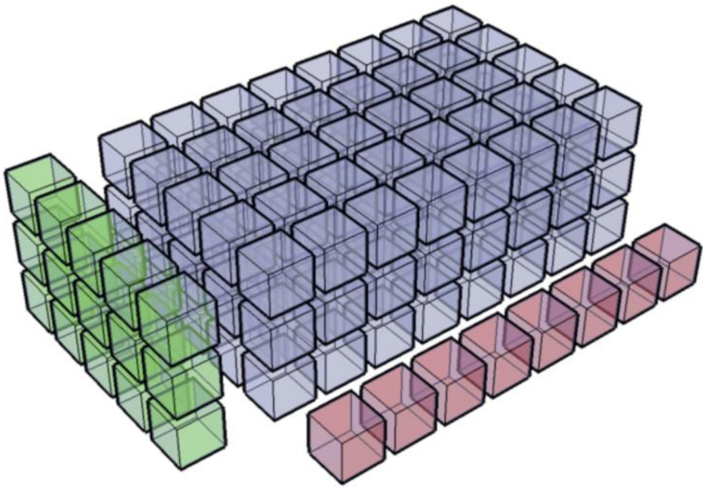
```
b = np.array([1 , 2 , 3])[:, None]
b
```

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

```
a + b
```

```
array([[ 1,  2,  3,  4],
       [ 6,  7,  8,  9],
       [11, 12, 13, 14]])
```

Broadcasting – 3D



```
x = np.zeros((3,5))
x
```

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

```
y = np.zeros(8)
y
```

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

The ellipsis ... means as many : as needed

```
x[...,None] + y     # shape(3,5,8)
```

```
array([[[0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.]],

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

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

Broadcasting Compare dimensions, starting from the last. Match when either dimension is one or None, or if dimensions are equal:

```
 1-D        2-D        3-D            ERROR

( ,)       (3, 4)     (3, 5, 1)      (3, 5, 2)
(3,)       (3, 1)     (      8)      (      8)
----       ------     -----------    ----------
(3,)       (3, 4)     (3, 5, 8)         XXX
```

Broadcasting is code-easy, but memory consumed. Use it wisely. Faster algorithm shall be np.add, np.subtract, np.multiply, np.divide, and other arithmetic.

Create your own ufunc

Create an ufunc to calculate area of rectangle within element-wise numpy array.

```
def area_rect(x,y):
    return x*y

# create an ufunc with 2 input 1 output
funcarea = np.frompyfunc(area_rect, 2, 1)

arr1 = [1,2,3,4]
arr2 = [2,3,4,5]
funcarea(arr1, arr2)
```

```
array([2, 6, 12, 20], dtype=object)
```

```
type(funcarea)
```

```
numpy.ufunc
```

Filter array with condition

```
arr = np.array(np.arange(1,10))
arr
```

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

```
# filter with even number condition
arr[arr % 2 == 0]
```

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

```
arr[arr > 5]
```

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

Almost every NumPy functions are element-wise. These functions help fasten data manipulation and performance.

NumPy functions are specifically designed for array and numbers processing.

Discover more on official website: numpy.org

Reference

Official Website:

https://numpy.org/doc/stable/index.html

Reference textbook:

Python Data Science Handbook, Jake VanderPlas, O'Reilly

Python for Data Analysis, 2nd edition, Wes McKinney, O'Reilly

More Exercise on:

https://www.machinelearningplus.com/