Tensor in Machine Learning

speech data

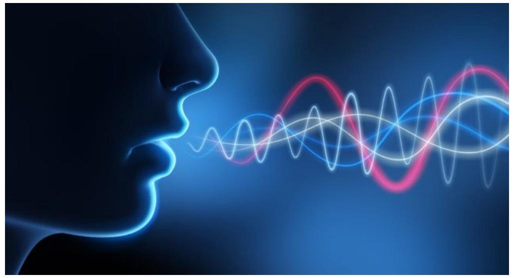
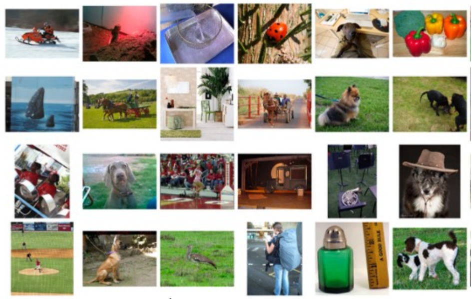


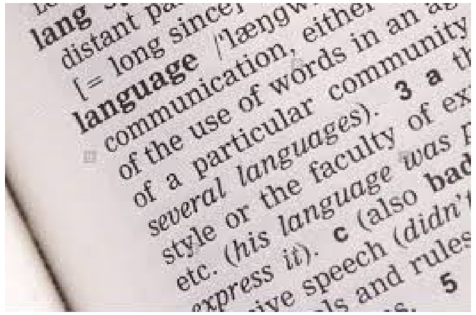
Image from www.computerweekly.com

image data



imagenet

text data



www.alamy.com

Load Data → Data in tensors → Algorithms (tensors) →

Speech Recognition
Image Classification
Language Translation

Data is represented by tensors

Algorithms operate on tensors

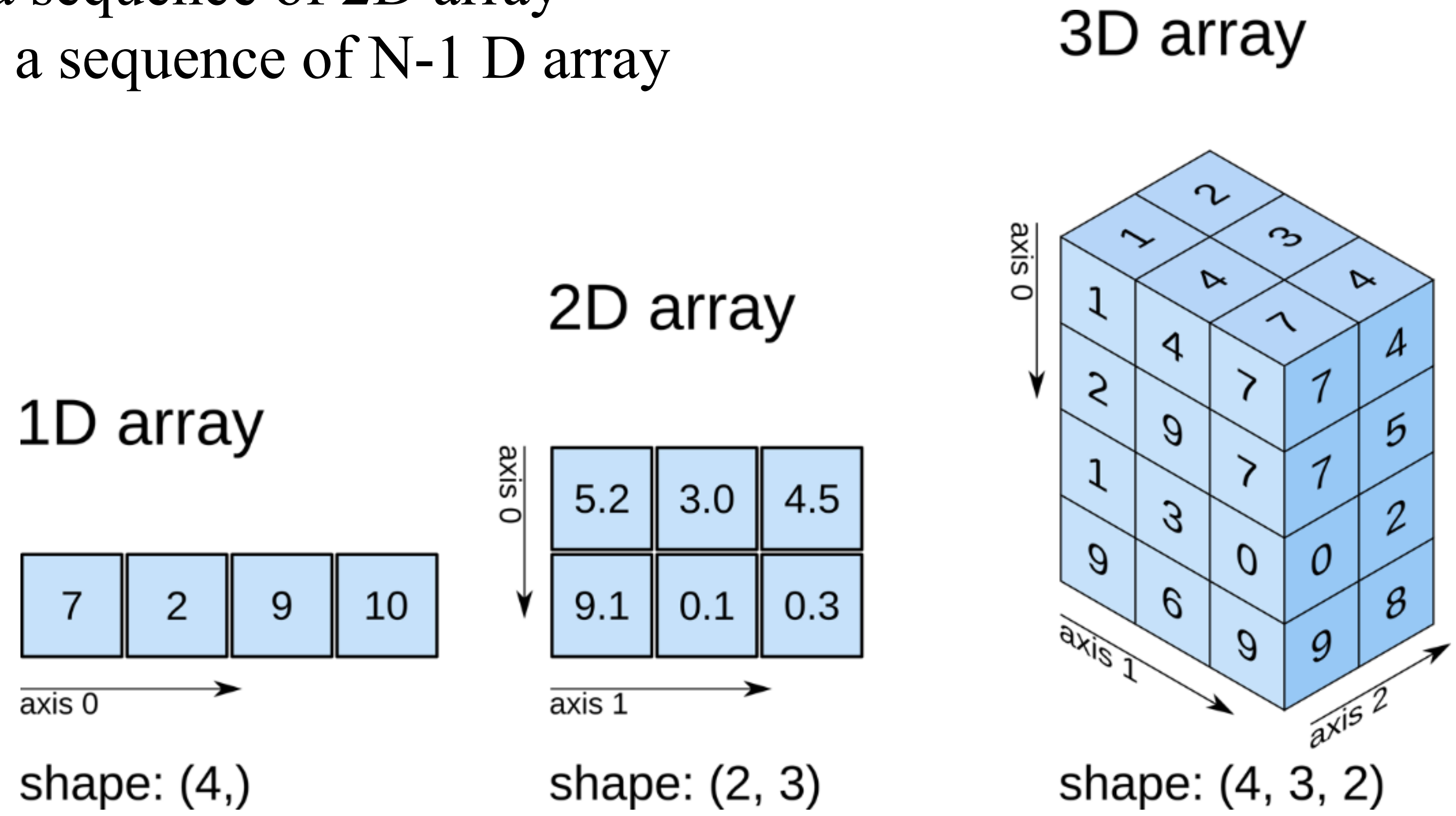
Numpy Array is Tensor

Tensor in Machine Learning

- Tensors are generalizations of scalars (that have no indexes), vectors (that have exactly one index), and matrices (that have exactly two indexes) to an arbitrary number of indices.
- a **rank-n** tensor has **n indexes**
- a rank-0 tensor is a scalar
- a rank-1 tensor is a vector using one index to locate an element
- a rank-2 tensor is a matrix using two indexes to locate an element
- a rank-3 tensor can represent a 3D volume or a sequence of rank-2 tensors. It uses three indexes to locate an element.

In Numpy, Tensor is represented by Array

- Rank-0 tensor: 0D array
- Rank-1 tensor: 1D array is a sequence of objects (e.g. numbers: int64, float64)
- Rank-2 tensor: 2D array is a sequence of 1D array
- Rank-3 tensor: 3D array is a sequence of 2D array
- Rank-N tensor: ND array is a sequence of N-1 D array



https://www.safaribooksonline.com/library/view/elegant-scipy/9781491922927/ch01.html

1D (dimensional) Numpy Array

```
import numpy as np
```

use the function **np.array** to create an array from a list

```
A = np.array([1, 2, 3], dtype='int32')
```

A is a 1D array with three elements
the data type (dtype) of each element is int32 (integer with 32 bits)

use the property **shape** to get the 'shape' of the array

```
A.shape
```

A.shape is (3,) which is a tuple with only one element

Create Special Numpy Arrays

```
import numpy as np

A0 = np.zeros(10, dtype='float32')

A1 = np.ones(10, dtype='float32')

A2 = np.empty(10, dtype='float32')

A3 = np.random.rand(10)
```

A3.dtype is 'float64'

Create Special Numpy Arrays

```
import numpy as np
```

```
A0 = np.arange(0, 10, 2, dtype='int32')
```

A0 is array([0, 2, 4, 6, 8], dtype=int32)

```
A1 = np.linspace (0, 1, 5)
```

A 1 is array([0. , 0.25, 0.5 , 0.75, 1.])

A1.dtype is 'float64'

Numpy Array Attributes

Attribute	Description
shape	A tuple that contains the number of elements (i.e., the length) for each dimension (axis) of the array
size	The total number of elements in the array
ndim	Number of dimensions (axes)
dtype	The data type of the elements in the array
nbytes	Number of bytes used to store the data

Code

```
A = np.array([[1, 2, 3], [4, 5, 6]], dtype='float64')
print('A is', A)
print('shape is', A.shape)
print('size is', A.size)
print('ndim is', A.ndim)
print('nbytes is', A.nbytes)
print('dtype is', A.dtype)
```

Output

```
A is [[1. 2. 3.]
      [4. 5. 6.]]
shape is (2, 3)
size is 6
ndim is 2
nbytes is 48
dtype is float64
```

1D 'array': [1., 2., 3.]
1D 'array': [4., 5., 6.]
A is a sequence of the two 1D arrays.
Thus, A is a 2D array

Demo:
1D_Numpy_array_basics.ipynb
Numpy_array.ipynb

Vectorized Operations on Numpy Array

```
import numpy as np
A = np.array([1, 2, 3], dtype='int32')
B = np.array([4, 5, 6], dtype='int32')
```

Get a new array C such that C[n]=A[n]+B[n] for n=1,2,3

Use a for loop

```
C = np.zeros(A.shape, dtype='int32')
for n in range(0, C.shape[0]):
    C[n] = A[n] + B[n]
```

Use the Vectorized Operation +

```
C = A + B
```

The operation + is performed between two vectors A and B

Vectorized Operations on Numpy Array

```
import numpy as np
A = np.array([1, 2, 3], dtype='int32')
B = np.array([4, 5, 6], dtype='int32')
```

Get a new array C such that C[n]=A[n] / B[n] for n=1,2,3

Use a for loop

```
C = np.zeros(A.shape, dtype='int32')
for n in range(0, C.shape[0]):
    C[n] = A[n] / B[n]
```

Use the Vectorized Operation /

```
C = A / B
```

The operation / is performed between two vectors A and B

Vectorized Operations on Numpy Array

```
import numpy as np
A = np.array([1, 2, 3], dtype='int32')
B = np.array([4, 5, 6], dtype='int32')
```

Vectorized Operations

```
C1 = A + B
C2 = A - B
C3 = A * B
C4 = A / B
C5 = (A**2 + B**2)/(A+B)
C6 = np.sqrt(A)
C7 = A + 1
C8 = A * 100
```

Vectorized Operations are Much Faster Than Loops

```
import numpy as np
import timeit
A=np.random.rand(10000000)
```

Measure duration of a for loop

```
t1=timeit.default_timer()
for k in range(0, A.shape[0]):
    A[k] *= 3
t2=timeit.default_timer()
duration=t2-t1
print('duration:', duration, '(seconds)')
```

Measure duration of a vectorized operation

```
t1=timeit.default_timer()
A *= 3
t2=timeit.default_timer()
duration=t2-t1
print('duration:', duration, '(seconds)')
```

much faster…

Table 2-2. Arithmetic operators implemented in NumPy

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`)

- the table is from Python Data Science Handbook

Numerical Data Types in Numpy

https://docs.scipy.org/doc/numpy-1.15.0/user/basics.types.html

dtype	Description
int8, int16, int32, int64	Integer
uint8, uint16, uint32, uint64	Unsigned (non-negative) integer
bool	Boolean (True or False)
float16, float32, float64	Floating-point numbers
complex64, complex128	Complex-valued floating-point numbers

```
import numpy as np
A1 = np.array([1, 2, 3], dtype='int64')
A2 = np.array([1, 2, 3], dtype='uint64')
A3 = np.array([True, False, True], dtype='bool')
A4 = np.array([1.0, 2.0, 3.0], dtype='float64')
A5 = np.array([1+1j, 2+2j, 3+3j], dtype='complex64')
```

use a byte to represent a nonnegative integer (uint8)

computer memory stores data in binary format, i.e., a sequence of 0s and 1s

Integer (uint8)		1 byte = 8 bits							
0	→	0	0	0	0	0	0	0	0
1	→	0	0	0	0	0	0	0	1
2	→	0	0	0	0	0	0	1	0
255	→	1	1	1	1	1	1	1	1

1 byte can represent an unsigned integer in the range of 0 to 255

use a byte to represent an integer (int8)

computer memory stores data in binary format, i.e., a sequence of 0s and 1s

Number (int8)		1 byte = 8 bits							
0	→	0 (+)	0	0	0	0	0	0	0
1	→	0 (+)	0	0	0	0	0	0	1
127	→	0 (+)	0	1	1	1	1	1	1
-128	→	1 (-)	0	0	0	0	0	0	0
-1	→	1 (-)	1	1	1	1	1	1	1

1 byte can represent an integer in the range of -128 to 127

integer overflow - too large

dtype	bytes	range
int8	1	-128 to 127
int16	2	-32768 to 32767
int32	4	-2147483648 to 2147483647
int64	8	-9223372036854775808 to 9223372036854775807

https://docs.scipy.org/doc/numpy-1.15.0/user/basics.types.html

```
a = np.array([1, 2, 100], dtype='int8')
b = a*a
print(b)
```

~~[1 4 10000]~~

Here is the output:

```
[ 1  4 16]
```

integer **overflow**

There is no warning message from numpy !

Solutions: 1) rescale your data; **2)** use int32/int64 for numerical computation whenever possible

integer underflow (wrap): too small

dtype	bytes	range
uint8	1	0 to 255
uint16	2	0 to 65535
uint32	4	0 to 4294967295
uint64	8	0 to 18446744073709551615

https://docs.scipy.org/doc/numpy-1.14.0/user/basics.types.html

```
a = np.array([1, 2, 3], dtype = 'uint64')
b = a - 3
print(b)
```

~~[-2, -1, 0]~~

Here is the output:

```
[18446744073709551614
 18446744073709551615
 0]
```

There is no warning message from numpy !

Solution: Avoid using uint8/16/32/64 for numerical computation whenever possible

Range of Float (floating-point number)

float numbers can only represent a finite number of real numbers in a limited range

dtype	bytes	min	max
float16	2	-6.55040e+04	6.55040e+04
float32	4	-3.4028235e+38	3.4028235e+38
float64	8	-1.7976931348623157e+308	1.7976931348623157e+308

A float number (e.g. float32) is a binary number (0s and 1s) to represent a real number (e.g, 1.2)
The format of a float number is defined by IEEE Standard for Floating-Point Arithmetic (IEEE 754)

np.spacing(x)

Return the distance between a float x and the nearest float number

```
x=np.float32(1)
np.spacing(x)
```

```
1.1920929e-07
```

https://docs.scipy.org/doc/numpy/reference/generated/numpy.spacing.html

np.spacing(x)

Return the distance between a float x and the nearest adjacent float number (x + spacing(x) based on the document)

Spacing is not a constant

```python
x=np.float32(1)
np.spacing(x)
```

1.1920929e-07

```python
x=np.float32(100)
np.spacing(x)
```

7.6293945e-06

```python
x=np.float32(1e+5)
np.spacing(x)
```

0.0078125

```python
x=np.float32(1e+10)
np.spacing(x)
```

1024.0

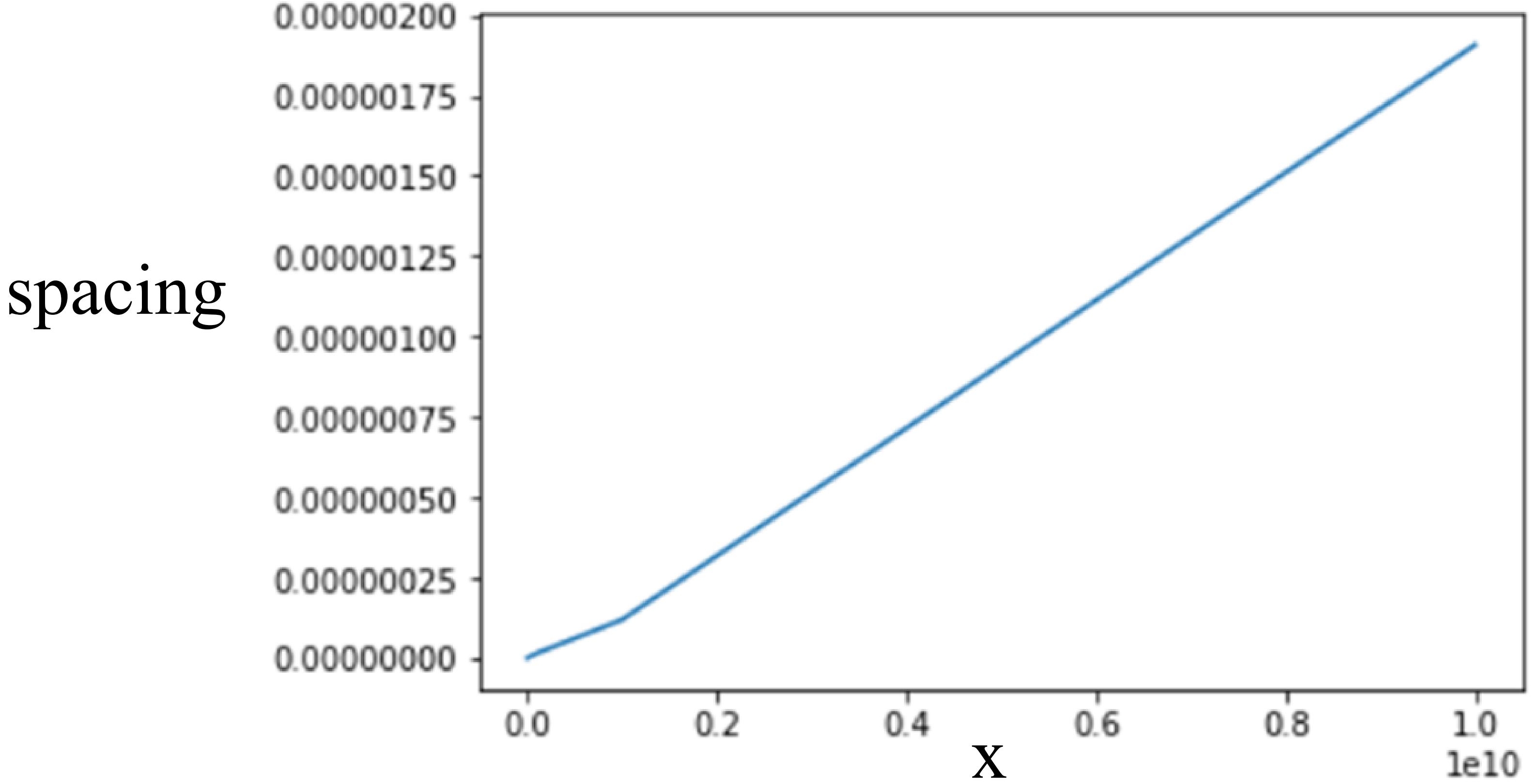
```python
x=np.float32(1e+30)
np.spacing(x)
```

7.5557864e+22

Spacing increases as x becomes larger

```
dList=[]
xList=[1, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6, 1e7, 1e8, 1e9, 1e10]
for x in xList:
    dList.append(np.spacing(x))
plt.plot(xList, dList)
```

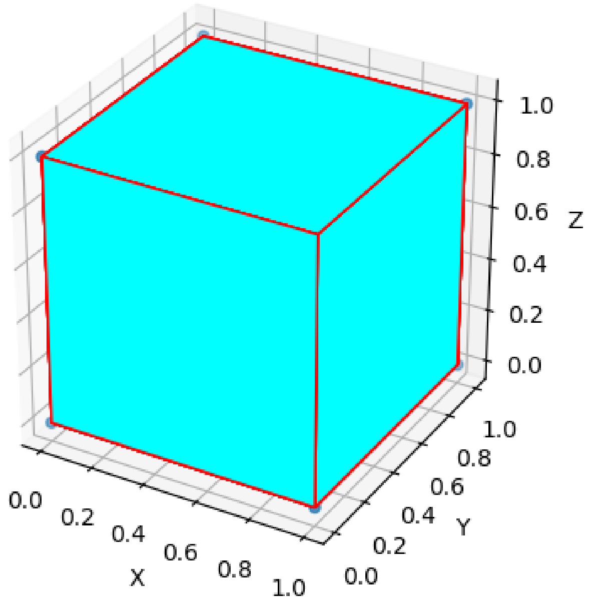
```
[<matplotlib.lines.Line2D at 0x217e8883dd8>]
```



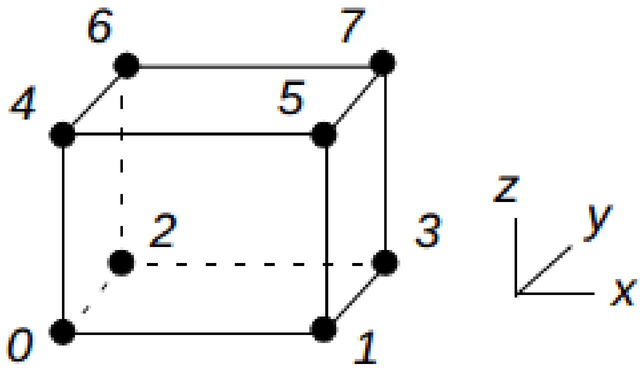
Demo:
Numpy_array.ipynb

An Example of 2D Numpy Array

a unit cube in 3D space



the cube is defined by 8 points
the order of points:



Point ID	x	y	z
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	0
4	0	0	1
5	1	0	1
6	0	1	1
7	1	1	1

```
import numpy as np
```

```
cube = np.array([[0, 0, 0],
                 [1, 0, 0],
                 [0, 1, 0],
                 [1, 1, 0],
                 [0, 0, 1],
                 [1, 0, 1],
                 [0, 1, 1],
                 [1, 1, 1]])
```

- An object is created in computer memory.
- The type of the object is numpy ndarray
- The object is named cube.

Dimensions/Axes of a 2D Numpy Array

create an array A

```
A = np.array([[0, 0, 0],
              [1, 0, 0],
              [0, 1, 0],
              [1, 1, 0],
              [0, 0, 1],
              [1, 0, 1],
              [0, 1, 1],
              [1, 1, 1]])
```

0	**0**	**0**
1	**0**	**0**
0	**1**	**0**
1	**1**	**0**
0	**0**	**1**
1	**0**	**1**
0	**1**	**1**
1	**1**	**1**

Dimensions/Axes of a 2D Numpy Array

A[i, j]
i is along axis_0
j is along axis_1

A[i, j]	A[:, 0]	A[:, 1]	A[:, 2]
A[0, :]	**0**	**0**	**0**
A[1, :]	**1**	**0**	**0**
A[2, :]	**0**	**1**	**0**
A[3, :]	**1**	**1**	**0**
A[4, :]	**0**	**0**	**1**
A[5, :]	**1**	**0**	**1**
A[6, :]	**0**	**1**	**1**
A[7, :]	**1**	**1**	**1**

axis=1

axis=0

A[7, 2]

A.shape is (8, 3)

A[i, j] is the same as A[i][j]

Dimensions/Axes of a 2D Numpy Array

A[i, j]
i is along axis_0
j is along axis_1

A[i, j]	A[:, 0]	A[:, 1]	A[:, 2]
A[0, :]	**0**	**0**	**0**
A[1, :]	**1**	**0**	**0**
A[2, :]	**0**	**1**	**0**
A[3, :]	**1**	**1**	**0**
A[4, :]	**0**	**0**	**1**
A[5, :]	**1**	**0**	**1**
A[6, :]	**0**	**1**	**1**
A[7, :]	**1**	**1**	**1**

axis=1

axis=0

A[7, 2]

A.shape is (8, 3)

average of the eight rows
(A[0,:] + A[1, :] +…+ A[7, :])/8

mean_ax0 = A.mean(axis=0)

mean_ax0 is array([0.5, 0.5, 0.5])

average of the three columns
(A[:, 0] + A[:, 1] + A[:, 1])/3

mean_ax1 = A.mean(axis=1)

array([0., 0.3, 0.3, 0.6, 0.3, 0.6, 0.6, 1.])

Dimensions/Axes of a N-D Numpy Array, a rank-N tensor

$$A[i_0\,,\,i_1\,,\,i_2\,,\,i_3\,,\,i_4\,,\,\ldots\,,\,i_{N\text{-}1}]$$

index on axis 0

index on axis 3

index on axis N-1

Dimensions/Axes of a 3D Numpy Array

```
A = np.array([ [ [1, 2], [1, 2], [1, 2] ],
               [ [3, 4], [3, 4], [3, 4] ],
               [ [5, 6], [5, 6], [5, 6] ],
               [ [7, 8], [7, 8], [7, 8] ] ], dtype='int64')
```

A.shape is (4, 3, 2)

A[0,:,:] is array([[1, 2], [1, 2], [1, 2]], dtype=int64)

A[1,:,:] is array([[3, 4], [3, 4], [3, 4]], dtype=int64)

A[1,0,:] is array([3, 4], dtype=int64)

A[1,0,1] is 4

Reshape an array

```
import numpy as np
A = np.array([ [0, 1,  2, 3],
               [4, 5, 6, 7] ])
```

A.shape is (2,4)

```
B1 = A.reshape(1,8)
```

B1 is array([[0, 1, 2, 3, 4, 5, 6, 7]])

```
B2 = A.reshape(8)
```

B2 is array([0, 1, 2, 3, 4, 5, 6, 7])

A, B1 and B2 share the same data in computer memory in most cases

Repeat elements of an array

```
import numpy as np
A = np.array([[0, 1,  2, 3]])
```

```
B1=A.repeat(2, axis=0)
```

B1 is array([[0, 1, 2, 3],
[0, 1, 2, 3]])

```
B2=A.repeat(2, axis=1)
```

B2 is array([[0, 0, 1, 1, 2, 2, 3, 3]])

Swap axes/dimensions

```
A = np.array([[0, 1,  2, 3],
              [4, 5, 6, 7] ])
B=A.transpose(1,0)
B
```

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

```
A = np.array([[0, 1,  2, 3],
              [4, 5, 6, 7] ])
B=A.transpose(0,1)
B
```

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

```python
A = np.array([[[0, 1], [2, 3]],
              [[4, 5], [6, 7]],
              [[8, 9], [10, 11]]])
A
```

```
array([[[ 0,  1],
        [ 2,  3]],

       [[ 4,  5],
        [ 6,  7]],

       [[ 8,  9],
        [10, 11]]])
```

```python
A.shape
```

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

```python
B=A.transpose(0,2,1)
B
```

```
array([[[ 0,  2],
        [ 1,  3]],

       [[ 4,  6],
        [ 5,  7]],

       [[ 8, 10],
        [ 9, 11]]])
```

```python
A = np.array([[[0, 1], [2, 3]],
              [[4, 5], [6, 7]],
              [[8, 9], [10, 11]]])
A
```

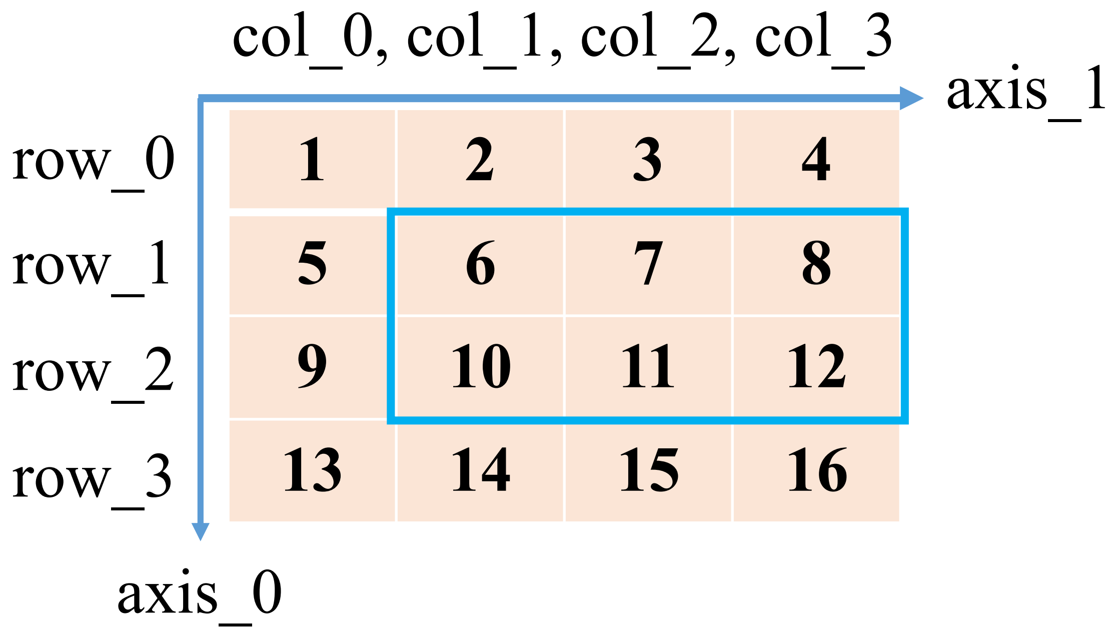
```python
B=A.transpose(2,1,0) # A[i,j,k] = B[k,j,i]
B
```

```
array([[[ 0,  4,  8],
        [ 2,  6, 10]],

       [[ 1,  5,  9],
        [ 3,  7, 11]]])
```

Demo:
Numpy_array.ipynb

Slicing a Numpy Array

```
A = np.array([[1,    2,  3,   4],
              [5,    6,  7,   8],
              [9,   10, 11, 12],
              [13, 14, 15, 16]])
B = A[1:3,1:4]
print("B is", B)
```



Output

```
B is [[ 6   7  8]
      [10 11 12]]
```

A[1:3,1:4]

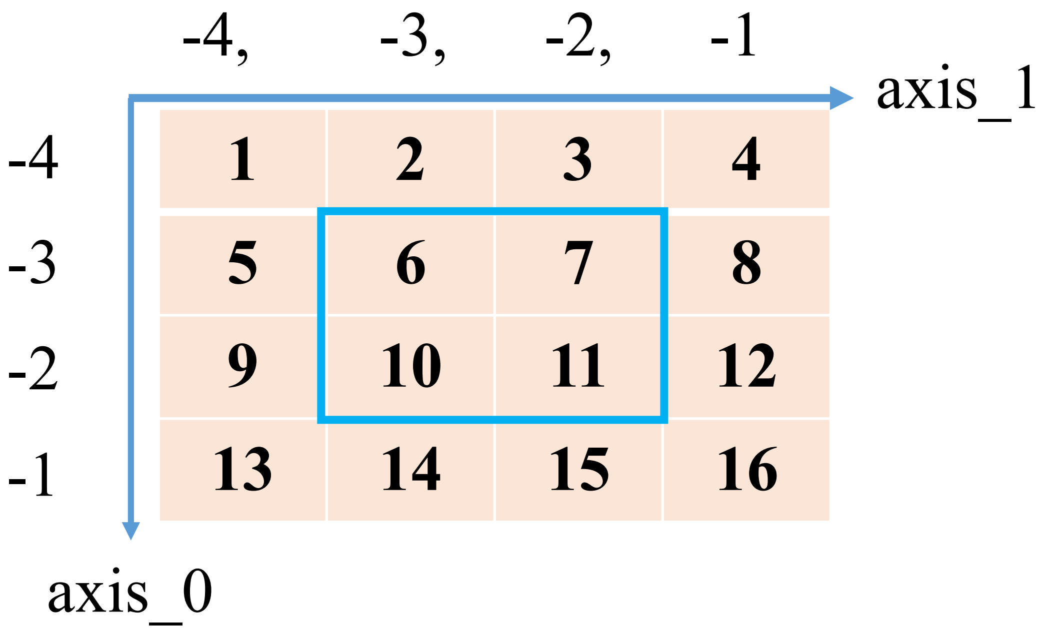
1:3: row_1, row_2 (row_3 is not included)
1:4: col_1, col_2, col_3 (no col_4)

Slicing a Numpy Array

```
A = np.array([[1,   2,  3,   4],
              [5,   6,  7,   8],
              [9,  10, 11, 12],
              [13, 14, 15, 16]])
B = A[-3:-1,-3:-1]
print("B is", B)
```

Output

```
B is [[ 6   7]
      [10 11]]
```



A[-3:-1, -3:-1] using negative index

After Slicing, The Two Arrays Share Data

```
import numpy as np
A = np.array([0, 1, 2, 3, 4, 5], dtype='int64')
```

```
B = A[0:1]
```

B has only one element equal to A[0]

Modify the element of B

```
B[0] = 100
```

What is A now ?

A is array([100,   1,   2,   3,   4,   5], dtype=int64)

Get a Sub-array by using an IndexList

```
import numpy as np
A = np.array([0, 1, 2, 3, 4, 5], dtype='int64')
```

```
IndexList=[0]
B = A[IndexList]
```

B is array([0], dtype=int64)

Modify the element of B

```
B[0] = 100
```

What is A now ?   A is array([0, 1, 2, 3, 4, 5], dtype='int64')

After Slicing, The Two Arrays Share Data

```
import numpy as np
A = np.array([ [0, 1, 2],
               [3, 4, 5]], dtype='int64')
```

```
B = A[0:1,0:1]
```

B has only one element equal to A[0,0]

Modify the element of B

```
B[0] = 100
```

What is A now ?

```
A is array([[100,   1,   2],
            [  3,   4,   5]], dtype=int64)
```

Get a Sub-array by using Index Lists

```
import numpy as np
A = np.array([ [0, 1, 2],
               [3, 4, 5]], dtype='int64')
```

```
RowIndexList = [0, 1]
ColIndexList  = [1, 2]
B = A[RowIndexList, ColIndexList]
```

B is array([1, 5], dtype=int64)

Modify the elements of B

```
B[0] = 100; B[1] = 200;
```

What is A now ?   A is array([[0, 1, 2],
                  [3, 4, 5]], dtype='int64')

Exercise

```
import numpy as np
A = np.array([0, 1, 2, 3, 4, 5], dtype='int64')
```

```
B = A[1:5]
B[0]=100
```

What is A now ?

```
IndexList=[1,2,3,4]
B = A[IndexList]
B[0]=100
```

What is A now ?

Array Concatenation

concatenate more than two arrays into one array

```
import numpy as np
x1 = np.array([ [1, 2],
                [3, 4] ])
x2 = np.array([ [5, 6],
                [7, 8] ])
x3 = np.array([ [9, 10],
                [11, 12] ])
```

```
y0 = np.concatenate([x1, x2, x3], axis=0)
```

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

```
y1 = np.concatenate([x1, x2, x3], axis=1)
```

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

Exercise

```
import numpy as np
X1 = np.array([ [1, 2,  3, 4],
                [3, 4, 5, 6] ])
X2 = np.array([ [5, 6, 7, 8],
                [7, 8, 9, 10],
                [8, 9, 10, 11] ])
```

concatenate the two arrays

Table 2-3. Aggregation functions available in NumPy

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

- the table is from Python Data Science Handbook

```python
import numpy as np
```

axis 1

```python
A = np.array([[0, 1, 2],
              [3, 4, 5]], dtype='int64')
```

axis 0

```python
np.any(A>10, axis=1)
```

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

```python
np.any(A>0, axis=0)
```

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

```python
import numpy as np
```

```python
A = np.array([[1, 2],
              [3, 4]], dtype='int64')
```

```python
np.percentile(A, 0) # min
```

```
1.0
```

```python
np.percentile(A, 50) # median
```

```
2.5
```

```python
np.percentile(A, 90) # close to max
```

```
3.7
```

```python
np.percentile(A, 100) # maximum
```

```
4.0
```

axis 1

```python
A = np.array([[1, 2],
              [3, 4]],
```

axis 0

```python
np.percentile(A, 100) # maximum
```

```
4.0
```

```python
np.percentile(A, 0, axis=1) # min of each row
```

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

```python
np.percentile(A, 0, axis=0) # min of each col
```

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

```python
np.percentile(A, 50, axis=0) # median of each col
```

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

```python
np.percentile(A, 100, axis=0) # max of each col
```

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

Aggregation Functions

```
import numpy as np
A = np.array([ [0, 1, 2],
               [3, 4, 5]], dtype='int64')
```

axis 1

axis 0

```
s1 = A.sum()
```

s1=15, sum of all elements in A

```
s2 = A.sum(axis=0)
```

s2 is array([3, 5, 7], dtype=int64)

```
s3 = A.sum(axis=1)
```

s3 is array([3, 12], dtype=int64)

```
import numpy as np
A = np.array([ [0, 1, 2],
               [3, 4, 5]], dtype='int64')
```

axis 1

axis 0

```
s2 = A.sum(axis=0, keepdims=True)
```

s2 is array([[3, 5, 7]], dtype=int64)
s2.shape is (1, 3)

```
s3 = A.sum(axis=1 , keepdims=True)
```

s3 is array([[3],
              [12]], dtype=int64)
s3.shape is (2, 1)

```
import numpy as np
A = np.array([ [0, 1, 2],
               [3, 4, 5]], dtype='int64')
```

axis 1

axis 0

```
s1 = A.sum()
```

```
s1 = np.sum(A)
```

```
s2 = A.sum(axis=0)
```

```
s2 = np.sum(A, axis=0)
```

```
s3 = A.sum(axis=1)
```

```
s3 = np.sum(A, axis=1)
```

```python
import numpy as np
A = np.array([ [0, 1, 2],
               [3, 4, 5]], dtype='int64')
```

axis 1

axis 0

```python
s1 = A.mean()
```

```python
s1 = np.mean(A)
```

```python
s2 = A.mean(axis=0)
```

```python
s2 = np.mean(A, axis=0)
```

```python
s3 = A.mean(axis=1)
```

```python
s3 = np.mean(A, axis=1)
```

$$A[i_0\ , i_1\ , i_2\ , i_3\ , i_4\ , \dots , i_{N-1}]$$

index on axis 0

index on axis 3

index on axis N-1

what is A.sum(axis=m) ?

axis 0 axis 1 axis 2

```python
A=np.random.rand(4,3,2)
A.shape
```

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

```python
A0=A[:,:,0]
A0.shape
```

```
(4, 3)
```

```python
A1=A[:,:,1]
A1.shape
```

```
(4, 3)
```

```python
A0+A1
```

```
array([[0.93529969, 1.24377814, 0.96407248],
       [1.11518658, 1.62142335, 1.48923818],
       [0.91457861, 0.63107406, 0.84041144],
       [0.30051152, 1.2741225 , 0.86650336]])
```

```python
B=A.sum(axis=2)
B
```

```
array([[0.93529969, 1.24377814, 0.96407248],
       [1.11518658, 1.62142335, 1.48923818],
       [0.91457861, 0.63107406, 0.84041144],
       [0.30051152, 1.2741225 , 0.86650336]])
```

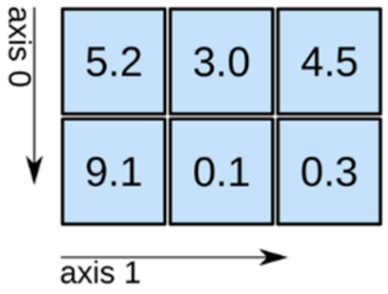
```
A = np.random.rand(n0, n1, n2, n3)

Then
A.sum(axis=2) is the same as

B = np.zeros((n0, n1, n3))

for m in range(0, n2):
    B+=A[:, :, m, :]
```

A[:, :, m, :] is the same as A[:, :, m]

Demo:
Numpy_array.ipynb

Basic Math Operations (+ - * ** /)

2D array



shape: (2, 3)

```
import numpy as np
A = np.array([ [0, 1, 2],
               [3, 4, 5]], dtype='int64')
B = np.array([ [1, 1, 1],
               [2, 2, 2]], dtype='int64')
```

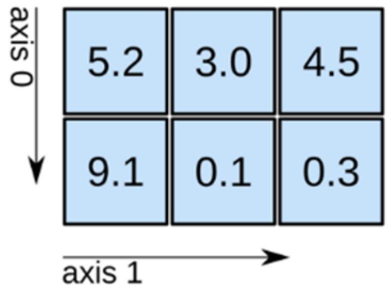
```
C = A + B
```

C[m, n] is equal to A[m, n] + B[m, n]

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

Basic Math Operations (+ - * ** /)

2D array



shape: (2, 3)

```
import numpy as np
A = np.array([ [0, 1, 2],
               [3, 4, 5]], dtype='int64')
B = np.array([ [1, 1, 1],
               [2, 2, 2]], dtype='int64')
```

```
C = A / B
```

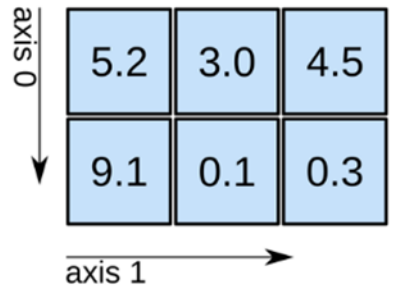
C[m, n] is equal to A[m, n] / B[m, n]

```
array([[0. , 1. , 2. ],
       [1.5, 2. , 2.5]])
```

dtype is float64

Basic Math Operations (+ - * ** /)

2D array



shape: (2, 3)

```
import numpy as np
A = np.array([ [0, 1, 2],
               [3, 4, 5]], dtype='int64')
```

power operation

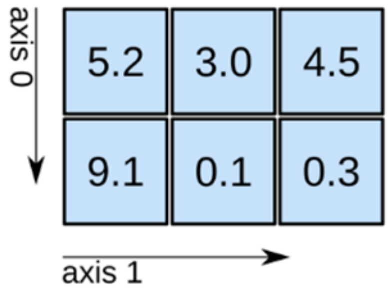
```
B = A ** 2
```

B[m, n] is equal to A[m, n] * A[m, n]

```
array([[ 0,  1,  4],
       [ 9, 16, 25]], dtype=int64)
```

Basic Math Operations (+ - * ** /)

2D array



shape: (2, 3)

```
import numpy as np
A = np.array([ [0, 1, 2],
               [3, 4, 5]], dtype='int64')
```

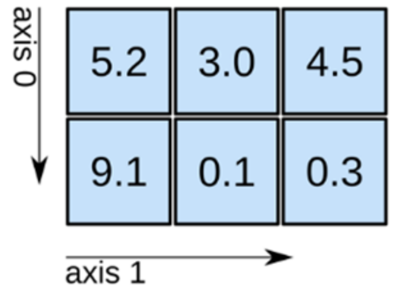
```
B = A * 2
```

B[m, n] is equal to A[m, n] * 2 for each m and n

```
array([[ 0,  2,  4],
       [ 6,  8, 10]],  dtype=int64)
```

Basic Math Operations (+ - * ** /)

2D array



shape: (2, 3)

```
import numpy as np
A = np.array([ [0, 1, 2],
               [3, 4, 5]], dtype='int64')
B = np.array([ [1, 1, 1],
               [2, 2, 2]], dtype='int64')
```

```
C = A * B
```

C[m, n] is equal to A[m, n] * B[m, n]

```
array([[ 0,  1,  2],
       [ 6,  8, 10]],  dtype=int64)
```

Element-wise multiplication

NOT matrix multiplication

Two Ways to do Matrix Multiplication

- (1) convert numpy array to numpy matrix

```
import numpy as np
A = np.array([ [0, 1],
               [2, 3]], dtype='int64')
A = np.matrix(A)
A = A*A  # matrix multiplication
```

https://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html

"It is no longer recommended to use this class, even for linear algebra. Instead use regular arrays. The class may be removed in the future."

- (2) use linear algebra functions

https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

Linear Algebra Function - **matmul**

https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

```
import numpy as np
A = np.array([ [0, 1, 2],
               [3, 4, 5]], dtype='int64')
B = np.array([ [1, 1, 1],
               [2, 2, 2]], dtype='int64')
```

matrix multiplication

```
C = np.matmul(A, B.T)
```

C is array([[3, 6],
            [12, 24]], dtype=int64)

Linear Algebra Function - **dot**

https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

```
import numpy as np
A = np.array([ [0, 1, 2],
               [3, 4, 5]], dtype='int64')
B = np.array([ [1, 1, 1],
               [2, 2, 2]], dtype='int64')
```

matrix multiplication

```
C = np.dot(A, B.T)
```

```
C is array([[ 3,  6],
             [12, 24]], dtype=int64)
```

Linear Algebra Function - **linalg.inv**

https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

```
import numpy as np
A = np.array([ [1, 1, 2],
               [3, 4, 5],
               [6, 7, 8]], dtype='float64')
```

inverse of matrix A

```
B = np.linalg.inv(A)
```

```
B is array([[ 1.        , -2.        ,  1.        ],
            [-2.        ,  1.33333333, -0.33333333],
            [ 1.        ,  0.33333333, -0.33333333]])
```

Demo:
Numpy_array.ipynb

Linear Algebra Functions in Numpy

- Matrix and vector products
- Decompositions (e.g. Singular Value Decomposition)
- Matrix eigenvalues and eigenvectors
- Norm, Rank, Determinant
- Solving equations and inverting matrices

Read the document:
https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

Array Broadcasting in Numpy

A = np.array([0,1,2,3]) B = 1

0	1	2	3	+	1

C = A + B

a vector + a scalar : it is not linear algebra !

Here is **rule** for the operation **broadcasting**

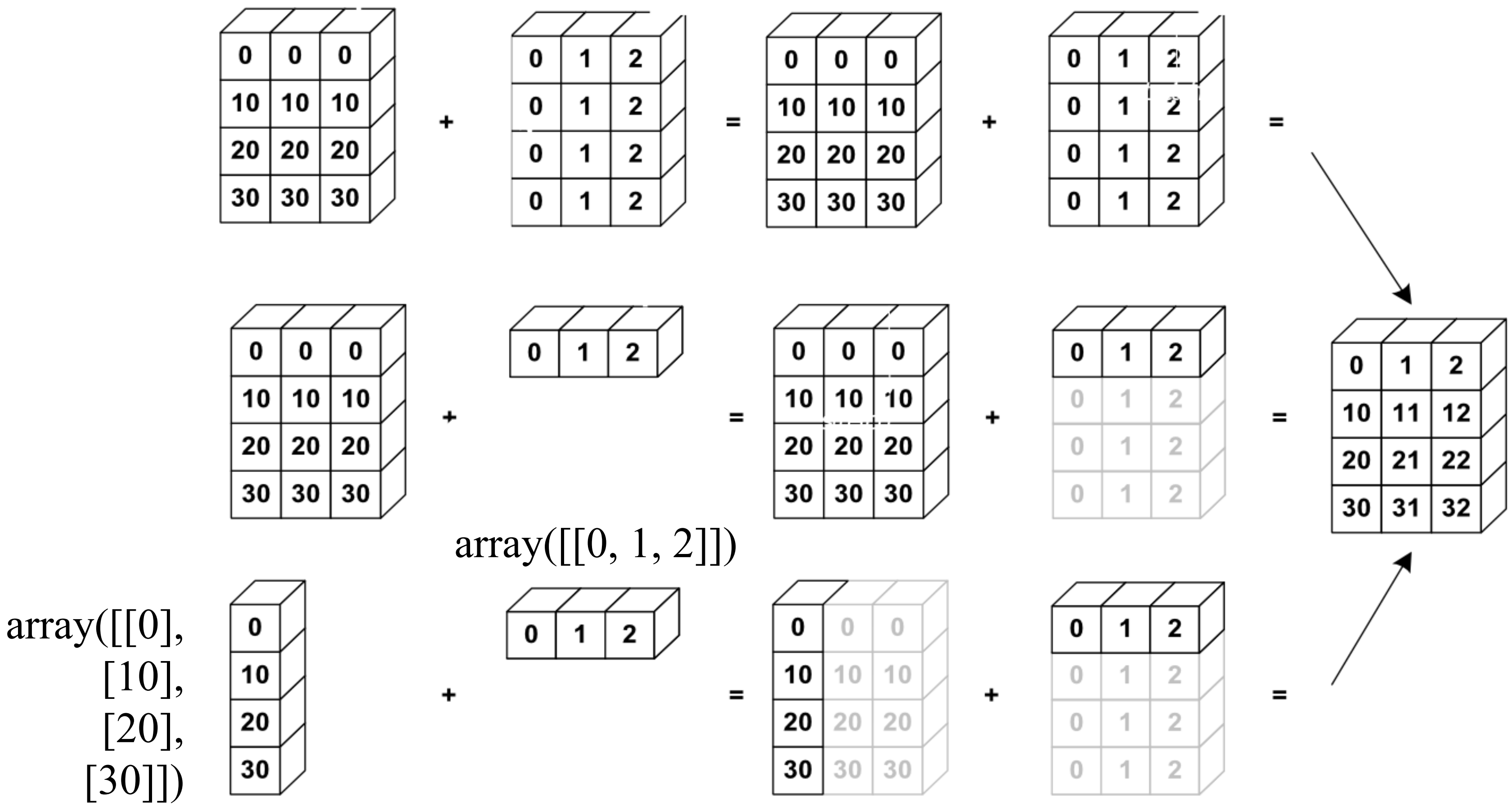
0	1	2	3	+	1	1	1	1

Now, A and B have the number of elements

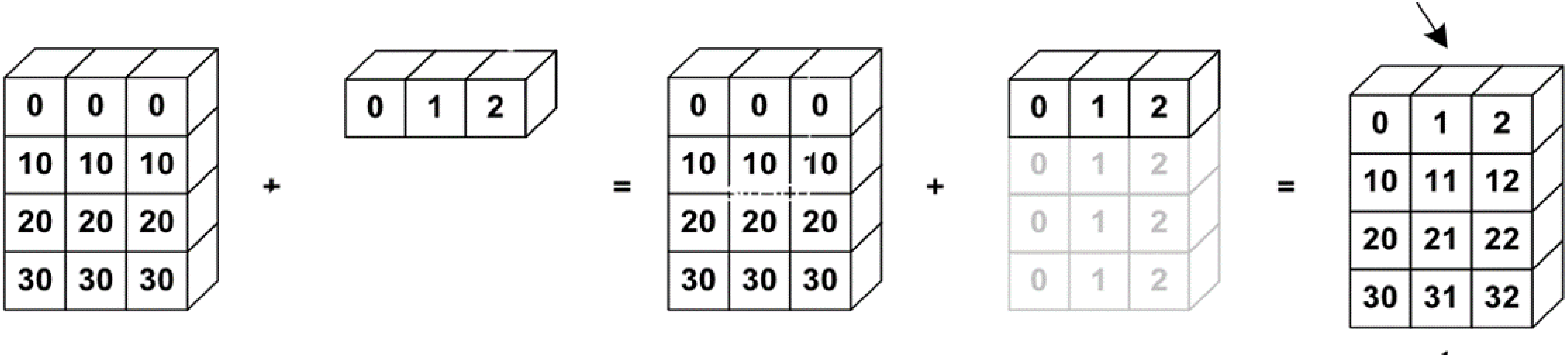
1	2	3	4

C is array([0,1,2,3])

Array Broadcasting in Numpy



https://mathematica.stackexchange.com/questions/99171/how-to-implement-the-general-array-broadcasting-method-from-numpy

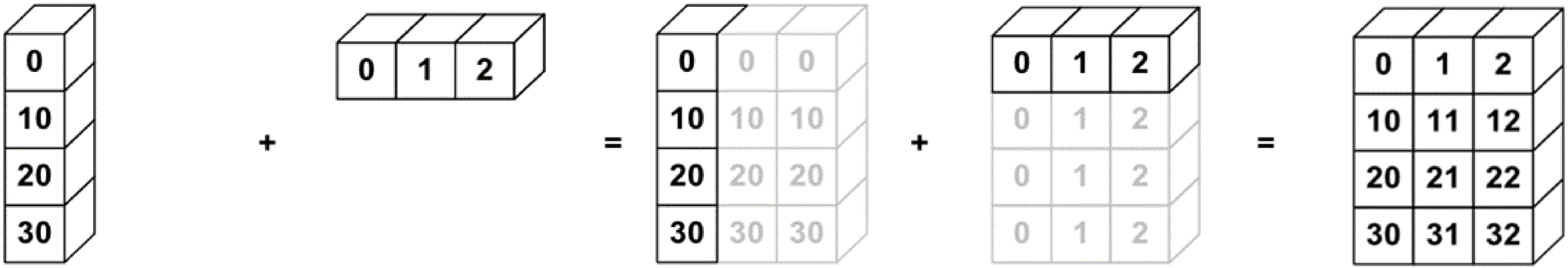



```
import numpy as np
A = np.array([ [0,   0,   0],
               [10, 10, 10],
               [30, 30, 30]])

B = np.array([[0, 1, 2]])

C = A + B
```

A.shape is (4, 3)

B.shape is (1, 3)

C is array([[0, 1, 2],
            [10, 11, 12],
            [20, 21, 22],
            [30, 31, 32]])



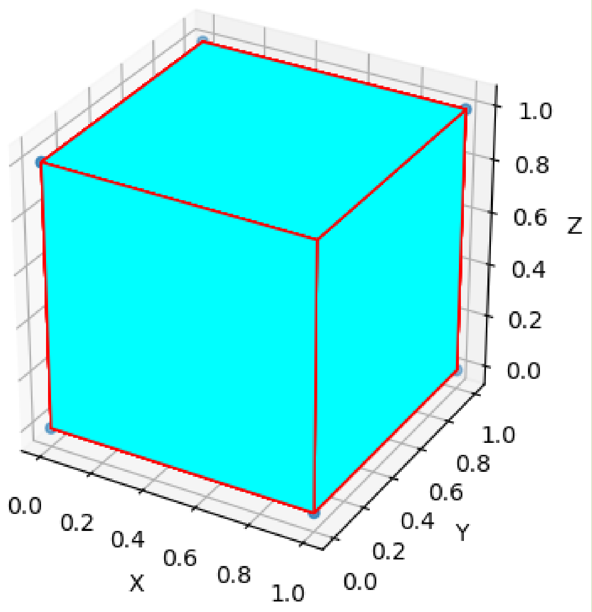

```
import numpy as np
A = np.array([ [0],
               [10],
               [20],
               [30]])

B = np.array([[0, 1, 2]])

C = A + B
```

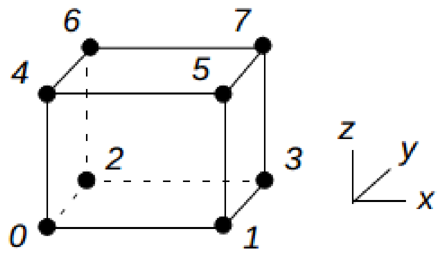
A.shape is (4, 1)

B.shape is (1, 3)

```
C is array([[ 0,  1,  2],
            [10, 11, 12],
            [20, 21, 22],
            [30, 31, 32]])
```

Example: calculate the distances between points in array A and a single point B in 3D space






```
import numpy as np
```

```
A = np.array([[0, 0, 0],
              [1, 0, 0],
              [0, 1, 0],
              [1, 1, 0],
              [0, 0, 1],
              [1, 0, 1],
              [0, 1, 1],
              [1, 1, 1]],
              dtype='float64')
```

```
B = np.array([2, 2, 2],
              dtype='float64')
```

calculate the distances in a for loop

```
dist = np.zeros(8, dtype='float64')
for n in range(0, 8):
    d = A[n,:] - B # displacement
    dist[n] = np.sqrt(np.sum(d**2))
```

for loop is slow !!!

too many lines of code - not readable

use array broadcasting

```
D = A - B
dist = np.sqrt(np.sum(D**2, axis=1))
```

Demo:
Numpy_array.ipynb