서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

넘파이(NumPy)

서울대학교 성원용

자료출처
Introduction to Numpy by Bryan Van DeVen

Python Numpy 선형대수 Moon Yong Joon
https://www.slideshare.net/dahlmoon/numpy-20160519

1. 인공지능과 선형대수

2. NUMPY, NDARRAY

3. NUMPY ARRAY를 이용한 일반연산

4. MATRIX-VECTOR, MAT-MAT 연산

VECTOR (SCALAR에 대비)

- 여러 개의 요소로 구성이 되어 있다.
  - 일 예로, 성적을 [국어점수, 영어점수, 수학점수, 기타] 로 나타내면 1차원의 크기가 4인 vector
  - 평면의 위치 (x, y 좌표)를 나타내는 것이 2차원 벡터
  - 컴퓨터에서는 array (list와 비슷하지만 모든 요소의 data type이 같다)로 나타낸다.

- Vector를 확장한 것이 matrix 이다. 일 예로, 학생의 이름과 성적, 두개의 축으로 값을 나타내면 matrix이다. 이 때 학생의 숫자가 m 명, 과목 종류가 n 개이면, mxn matrix로 성적을 나타낸다.

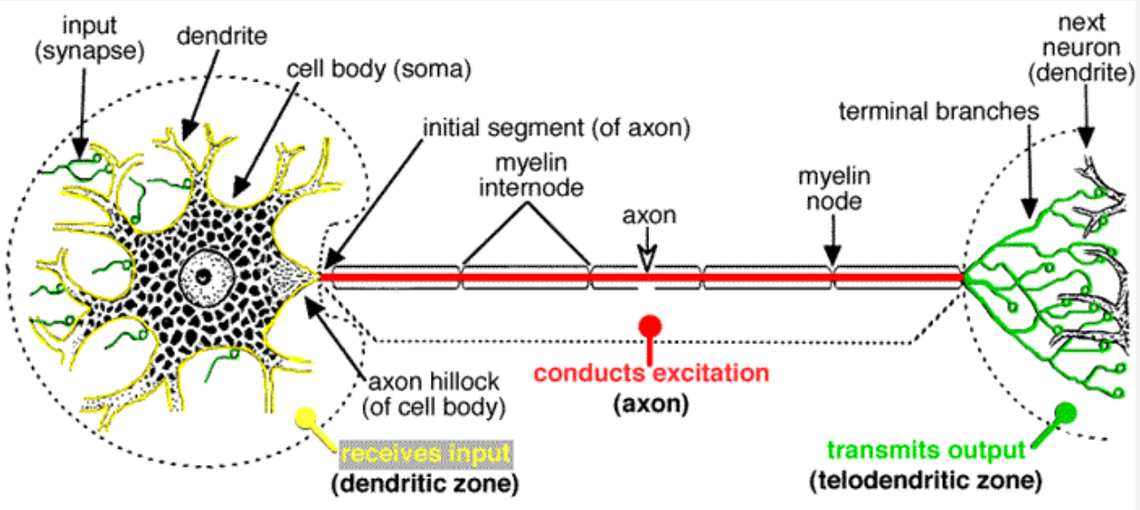
VECTOR ADDITION (SUBTRACTION), DOT PRODUCT

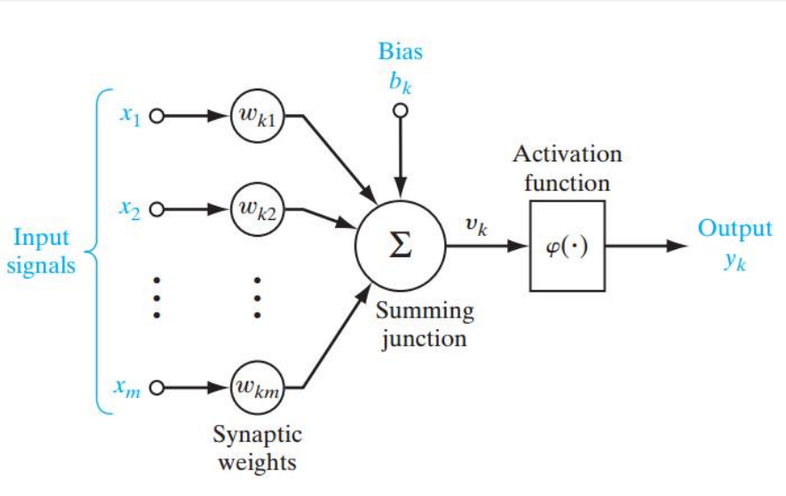
- Vector addition – 두개의 서로 차원이 같은 벡터의 요소 끼리의 합, 결과는 같은 차원의 벡터
- Vector dot product – 두개의 서로 차원이 같은 벡터의 요소 끼리의 곱을 다 더한 것 (결과는 scalar 값)

The dot product of two vectors **a** = [a_1, a_2, …, a_n] and **b** = [b_1, b_2, …, b_n] is defined as:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

벡터를 이용한 신경세포의 표현





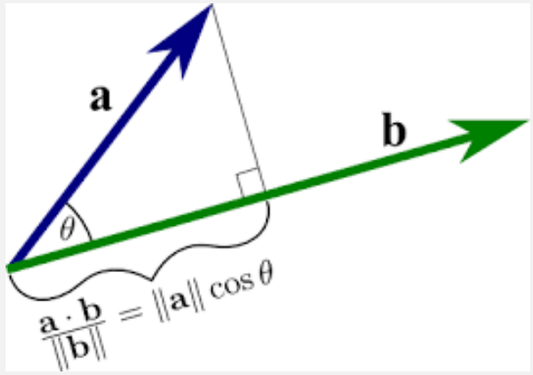
신경세포는 입력 벡터 (x1, x2, …, xm)과 weight (가중치) vector (wk1, wk2, …, wkm)의 dot-product 에 bias (scalar 값)를 더한 후, activation function 을 통한다. 대표적 activation 인 ReLU는 값이 +만 통과시키고, 마이너스이면 0으로 출력.

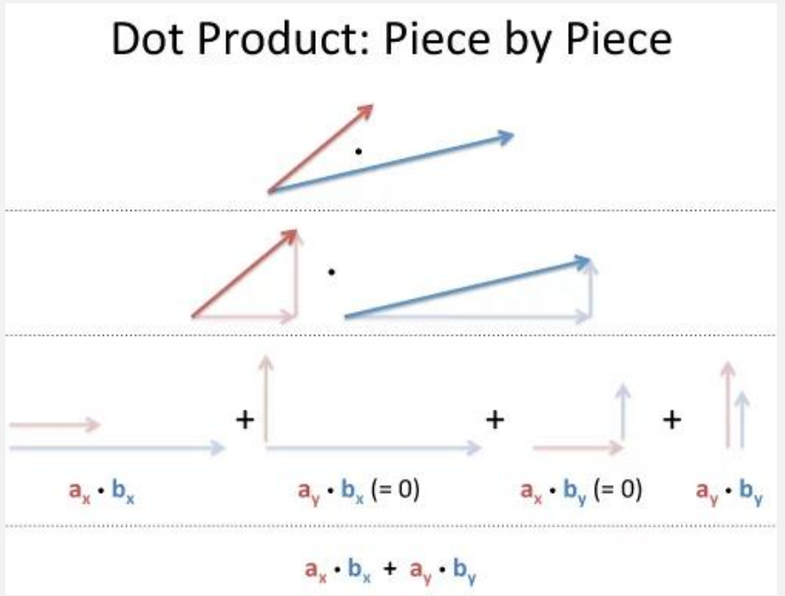
중요한 것 DOT PRODUCT

- Input [x1, x2, …, xn]
- Weight [w1, w2, …., wn]
- Activation 전 결과 v = x1*w1 + x2*w2 + … + xn*주
- 출력 y = ReLu or Sigmoid (v)

- Dot product 의 물리적 의미
  - 입력 x 와 w가 요소별로 같은 방향으로 값을 가지면 값이 커진다.
  - 예 x = [+1, +2, 0, -2], w = [+2, +2, -1, -2] -> +2+4+4=10
  - 예 x = [+1, -2, 2, -2], w = [+2, +2, -1, -2] -> +2-4-2+4=0

DOT PRODUCT 기하학적 의미

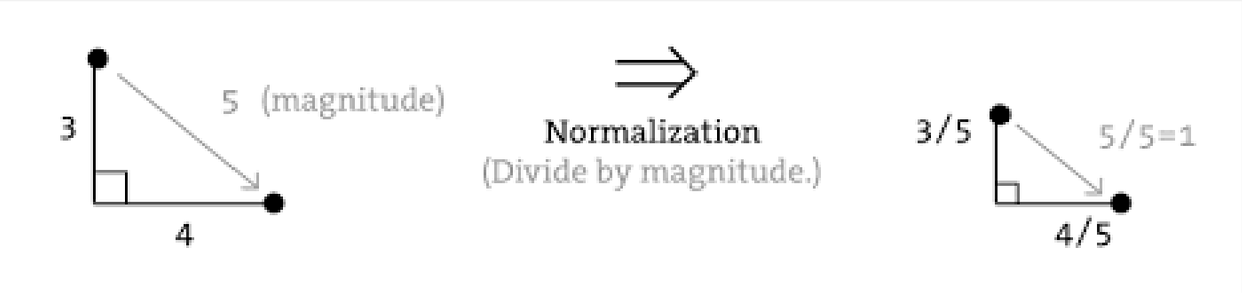




- 참고로 인간의 신경망에서 하나의 신경세포에 들어오는 입력의 개수는 보통 100개에서 10,000개 사이이다. (n= 100 ~ 10,000)

NORMALIZATION OF A VECTOR

- Vector 의 전체 크기가 너무 커지거나 작아지지 않도록 함. 이를 normalize (정규화) 라 함. Vector의 크기로 각 element의 값을 나누면 된다.
- Normalize 된 vector 의 크기는 1이고, 방향 정보만 남는다.



$$\hat{\mathbf{u}} = \frac{\mathbf{u}}{|\mathbf{u}|}$$

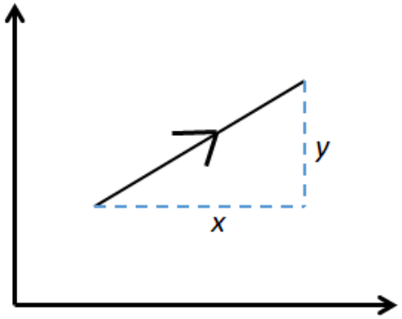
VECTOR MAGNITUDE

- Vector a 의 크기 – 벡터 a와 벡터 a 의 dot product 가 크기.
  - 2차원 벡터의 경우 피타고라스의 정리에 해당

The magnitude of vector $\vec{a}$ is written as $|\vec{a}|$.

The magnitude of vector $\overrightarrow{AB}$ is written as $|\overrightarrow{AB}|$.

If $\vec{a} = \begin{pmatrix} x \\ y \end{pmatrix}$ then the magnitude $|\vec{a}| = \sqrt{x^2 + y^2}$ (using Pythagoras theorem)



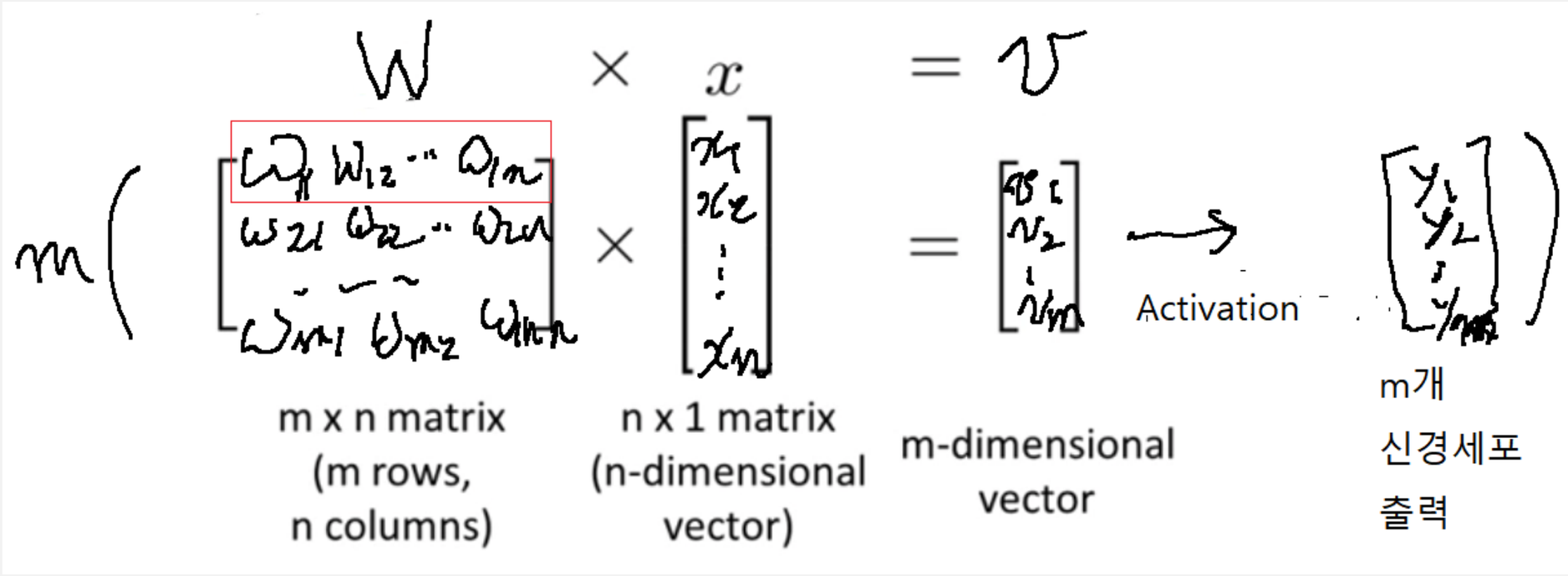
Magnitude= $\sqrt{x^2 + y^2}$ **(for 2D vectors)**

Magnitude= $\sqrt{x^2 + y^2 + z^2}$ **(for 3D vectors)**

DOT PRODUCT의 이용

- 인공신경세포 하나의 모델링
  - 인공신경세포가 많으면 matrix-vector inner product로 모델링 됨.
- Transformer model - Attention weight 값을 구할 때, 지금 어떤 출력(query)과 비슷한 과거의 입력(key)을 찾고자 할 때 이 둘의 dot product를 구한다.

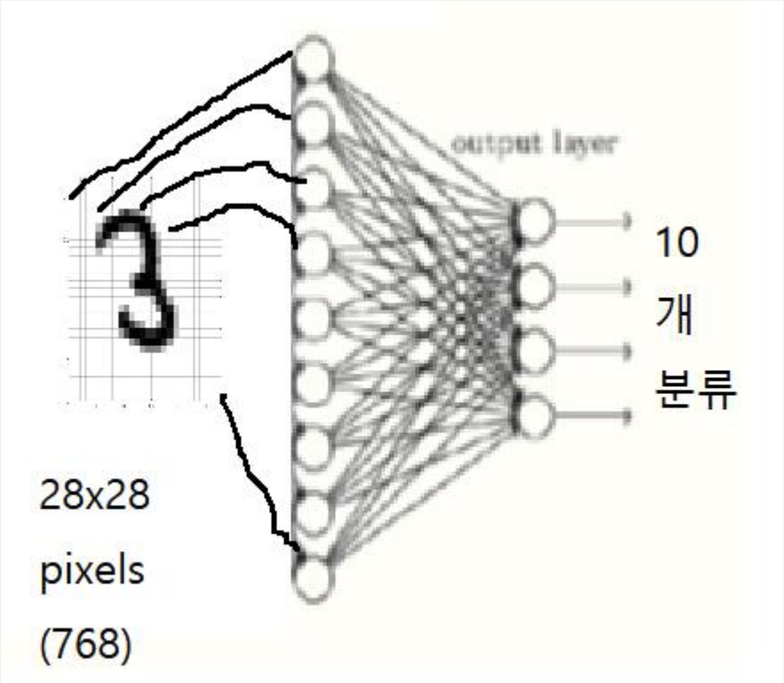
M 개의 신경세포 (MXN MATRIX와 NX1 입력 벡터의 INNER PRODUCT)의 모델링



- W의 row(수평방향벡터)와 x가 inner product 되어서 하나의 출력이 된다. 총 m개의 출력

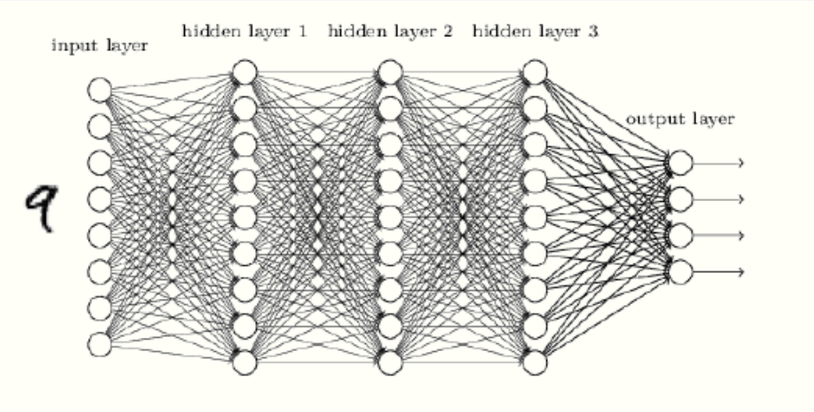
MNIST 분류 (얕은 신경망)

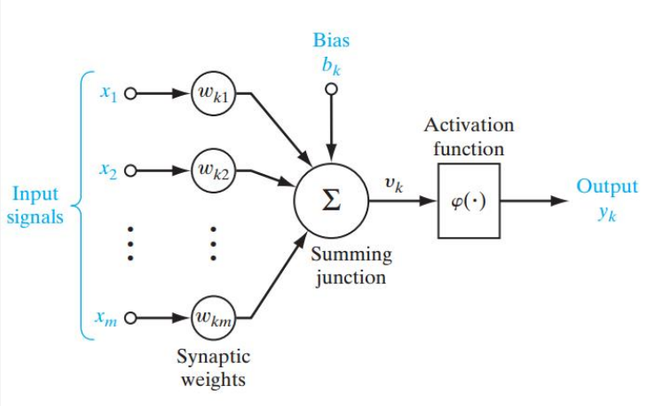
- 필기체 글자인식
- Input 28*28 pixels +1 bias -> 769
- 출력 10개 (앞의 식에서 m=10, n = 769에 해당)



MNIST CLASSIFICATION을 위한 FULLY-CONNECTED DEEP NEURAL NETWORK

- Assume four layers
- Input – 28x28 pixel, total 784 values (+1 for bias)
- 중간층 – (1024+1)*1024 matrix
- 맨 마지막 층 – (1024+1)*10개 matrix





총 파라미터의 숫자와 INFERENCE 를 위한 계산량

- 파라미터의 숫자 – 대충 4 million
- 하나의 파라미터로 하는 일 – 한번 곱셈, 덧셈
- 따라서 한번의 inference 를 위해서 하는 계산의 양 – 약 8 million (8백만번)
- 인간이 한다면 약 13년 걸림(with 10 sec for one operation, 8 hours a day, 200 days in a year)
- 1초에 10억번 계산하는 컴퓨터로 한다면, 약 1/100초
- 최근 GPU의 계산능력은 1초에 약 10조번 (1/1000,000 초 걸림)

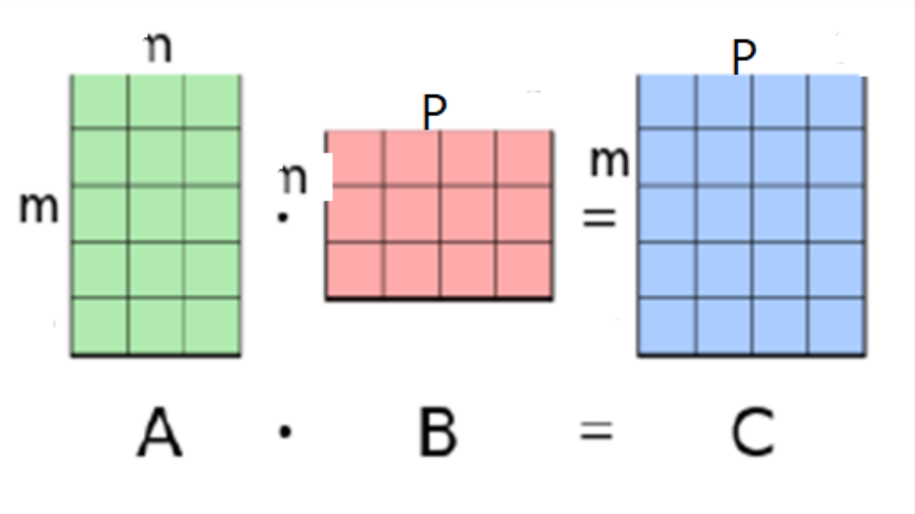
정리

- 신경세포는 입력 신호 (100~10,000차원의 벡터)와 그 것의 전달강도를 결정하는 weight 에 의해서 값을 결정한다. 입력과 weight 의 dot product 가 필요하다.
- Deep neural network 은 많은 숫자의 신경세포를 옆으로, 그리고 위로 쌓는다. 이러한 것을 matrix-vector 의 inner product 로 모델링 할 수 있다.

- Matrix-vector inner product 를 빨리 하는 방법이 필요하다.

MATRIX-MULTIPLICATION

- When the weight is represented as lxm (n-dim inputs are used by m neurons)
- The input (of n-dim) is applied with the batch size of p
- What is the advantage of increasing p (which is called batch processing)?
  - The efficiency increases because the weight matrix is re-used p-times.
- The matrix multiplication of A (mxn) * B(nxp) -> C (mxp)



If $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{B}$ is an $n \times p$ matrix,

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, p$.

1. 인공지능과 선형대수

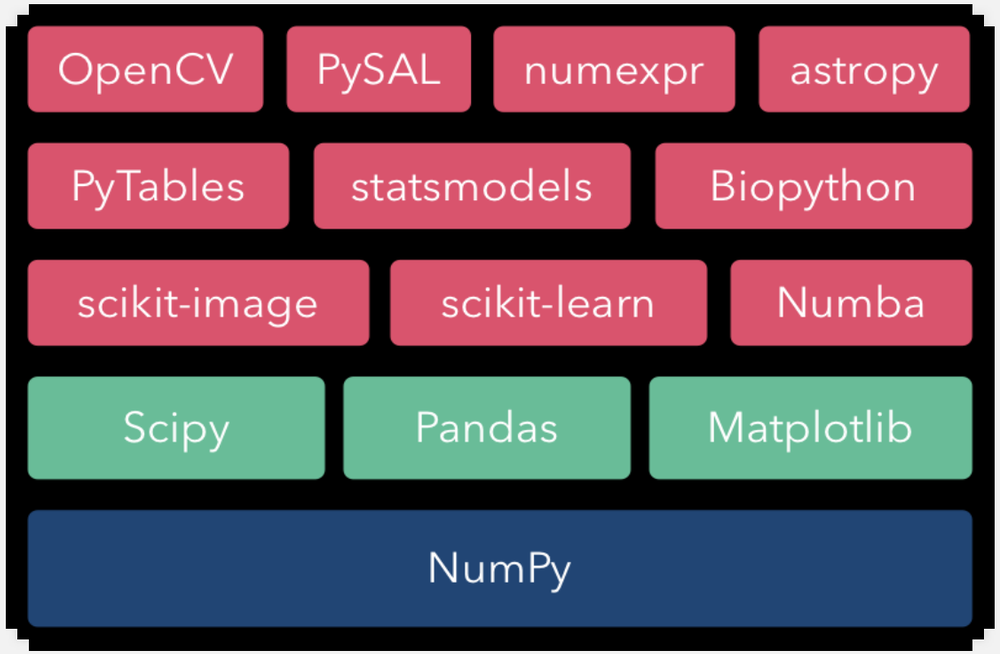
2. NUMPY, NDARRAY

3. NUMPY ARRAY를 이용한 일반연산

4. MATRIX-VECTOR, MAT-MAT 연산

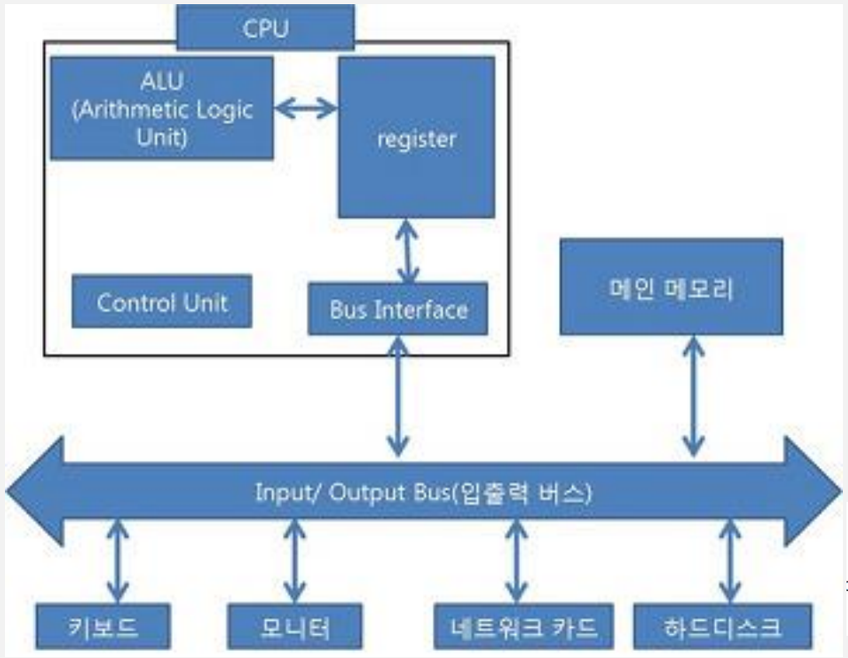
WHAT IS NUMPY? WHAT IS SCIPY?

- Python – 기본적으로 느리다 (interpretive, 자유로운 format
  - 계산이 많이 필요한 부분(loop) 을 벡터 명령어를 사용 가속
  - 이를 위한 pre-compiled library 가 필요
- Numpy – package for vector and matrix ma nipulation
- Scipy – package for scientific and technical computing

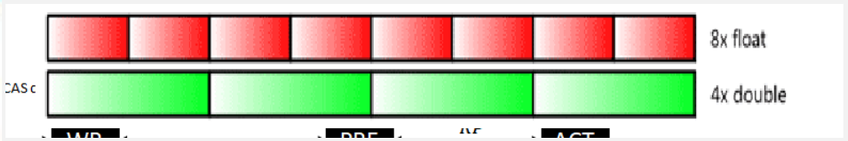


CPU 구조

- Conventional CPU
- 하나의 32bit or 64bit ALU가 한번에 하나의 연산



- CPU의 SIMD extension
  - 한번에 여러 개의 연산을 함 (2개 ~ 16개)
  - Intel의 경우 AVX명령어로 불리움.
  - AVX 128bit, AVX256bit
  - 이 때 적용되는 동작이 같아야 한다. (Single Instruction Multiple Data)
  - C/C++ compile에서 vector option 을 쓰면 금방 적용이 된다.



CPU 구조 (SIMD, SINGLE INSTRUCTION MULTIPLE DATA)

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

- 응용 C의 for loop

```
float a[256], b[256], c[256];

….

for (int i = 0; i <256; i++)
    c[i] = a[i] + b[i];
```

- 위의 loop를 256번이 아니고 8번 돌면서 끝낸다.
- 도중에 operation 의 종류를 바꿀 수 없다. 적용되는 data type 이 같아야 한다 (이 경우는 float). 따라서 scalar 연산에는 해당이 안되거나 실익이 거의 없다. 그런데 program에서 시간이 많이 걸리는 부분은 이렇게 loop가 도는 곳이다.

VECTOR 명령어 (SIMD 명령어)를 쓰기 위한 조건 및 장점

- Data 가 같은 type으로 정렬이 되어 있어야 한다
  - 비정열은 packing/unpacking 명령어로 정렬시킬 수 있지만 overhead 크다.
  - 적용되는 명령어는 벡터 데이터에 대해서 동일해야 한다 (+, *, > 등)
  - Run 하는 도중에 data type의 변경이 허용되지 않는다.

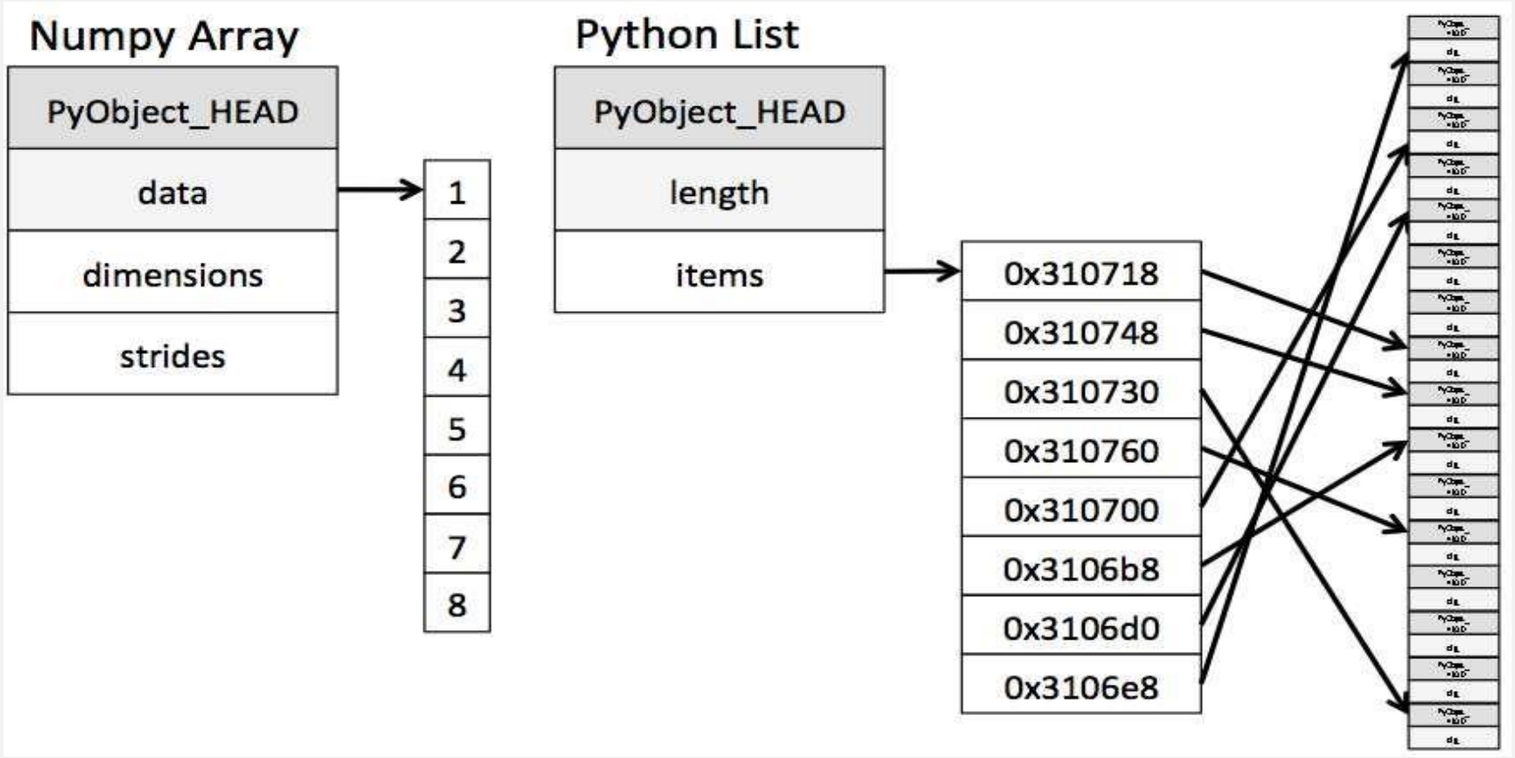
- Python의 list data-type은 이 기준에 맞지 않는다.
  - python_list_example = [“철수”, 10, -0.04, [1, 4]]

- NumPy에서는 ndarray type을 사용한다.
  - Multi-dimensional array (1차원, 2차원, 3차원, …)
  - 모든 element 가 같은 data type을 가진다.
  - C 언어의 배열과 가깝다.

NDARRAY VS. LIST 구조

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

Ndarray와 list는 내부 구조부터 다르게 되어있어
ndarray가 더 처리가 빠르게 실행됨



NDARRAY 타입과 매칭 1

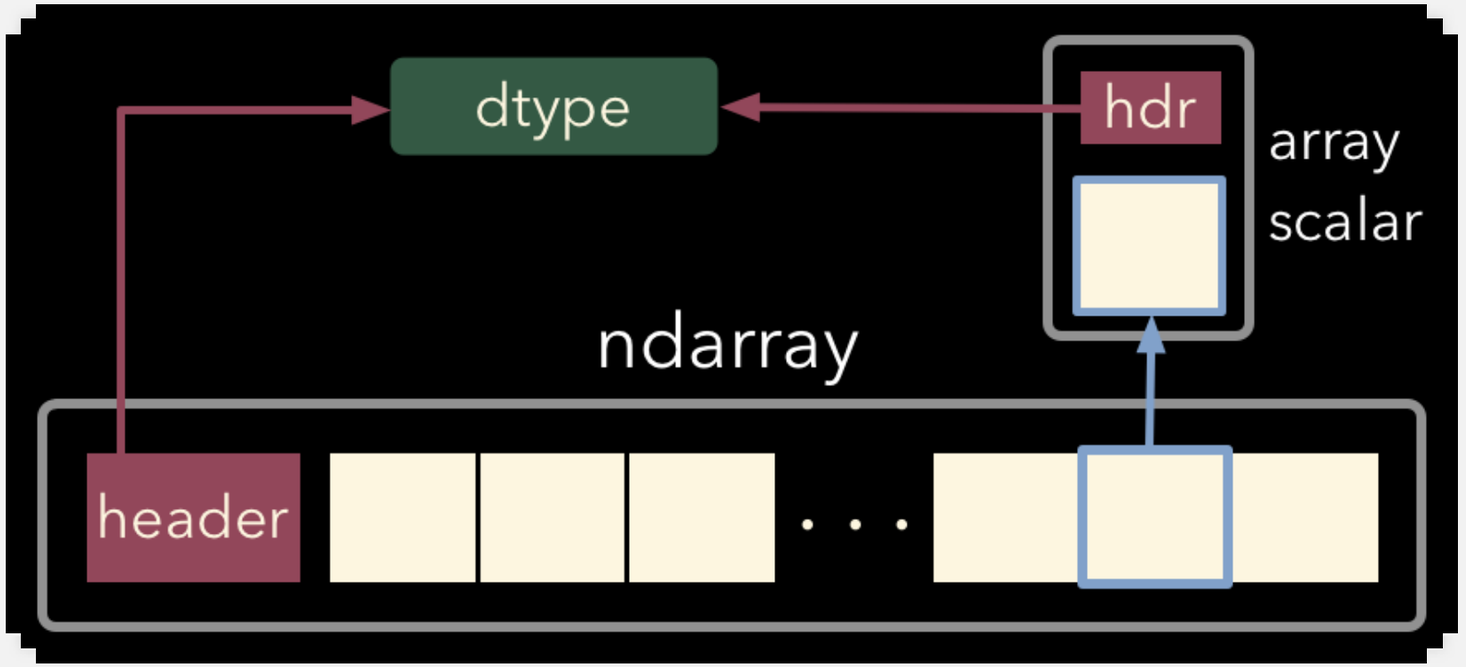
Ndarray 타입	타입 코드	설명
int8, uint8	i1,u1	1byites 정수형 int8(=128~127)
int16, uint16	i2,u2	2byites 정수형
int32, uint32	i4,u4	4byites 정수형
int64, uint64	i8,u8	8byites 정수형
float16	f2	반정밀도 부동소수점
float32	f4, f	단정밀도 부동소수점. C언어의 float형 호환
float64	f8, d	배정밀도 부동소수점, C언어의 double, Python의 float 객체
float128	f16, g	확장 정밀도 부동소수점
complex64	c8	32비트 부동소수점 2개를 가지는 복소수
complex128	c16	64비트 부동소수점 2개를 가지는 복소수

Ndarray 타입	타입 코드	설명
complex256	c32	128비트 부동소수점 2개를 가지는 복소수
bool	?	True, False 값을 저장하는 불리언 형
object	O	파이썬 객체형
string_	S	고정길이 문자열형-각 글자는 1바이트
unicode_	U	고정길이 유니코드

타입 매칭

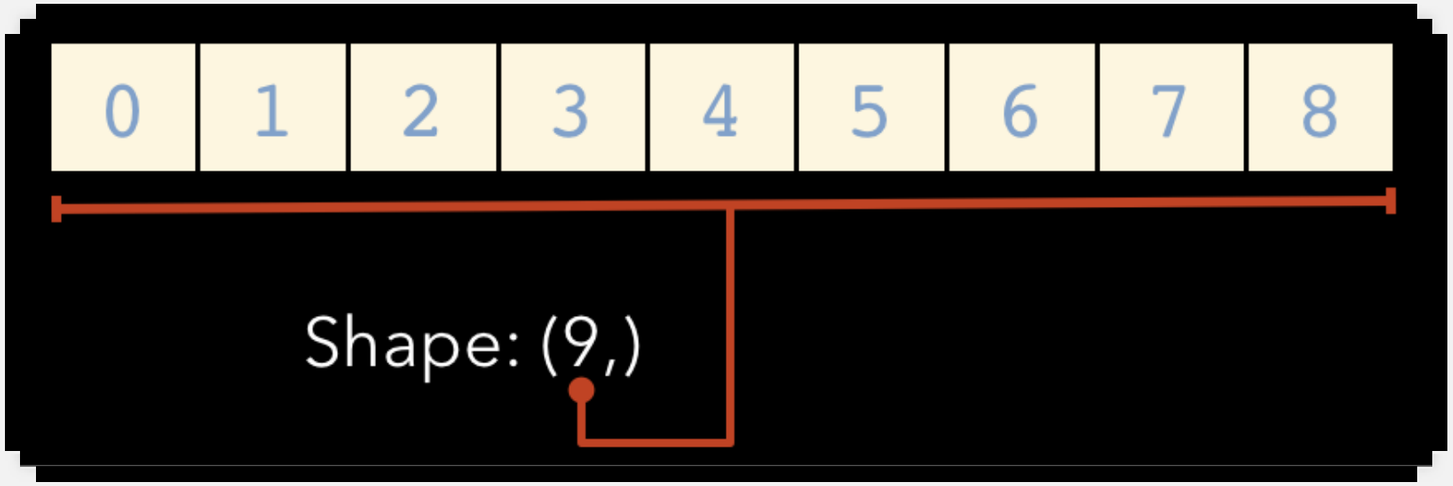
Python 과 달리 도중에 integer overflow가 날 수 있다. 왜냐하면 데이터의 크기를 run 하는 도중에 바꾸지 못한다.

ARRAY MEMORY LAYOUT



ARRAY SHAPE

One dimensional arrays have a 1-tuple for their shape



기억: Tuple, List

그냥 1-dim array이지 이 것이 9x1 또는 1x9를 의미하지 않는다.

NUMPY.ARRAY() 함수 – PYTHON LIST를 NDARRAY 로

- a_ndarray = numpy.array(list)

a = [1,2,3]

print(type(a)) -> <class 'list’>

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

print(type(a)) -> <class 'numpy.ndarray'>

isinstance(a, (np.ndarray)) -> True

데이터 타입 부여

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

ndarray는 각 원소별로 **동일한** 데이터 타입으로처리

list를 np.array(..) function을 이용하여 ndarray로 변환 가능

array([원소 , 원소 , 원소], dtype)

```
import numpy as np

l = [1,2,3,4]
a = np.array(l,np.int)
print(a)

a = np.array(l,np.float)
print(a)

a = np.array(l,np.str)
print(a)
```

```
[1 2 3 4]
[ 1.  2.  3.  4.]
['1' '2' '3' '4']
```

*참고
List 의 경우에는 element 사이의 간격에 쉼표(,)를 붙인다.

그런데 ndarray 의 경우에는 그냥 빈칸이다.

DTYPE을 따로 주지 않을 때 (INFERRED AUTOMATICALLY)

서울대학교 SEOUL NATIONAL UNIVERSITY

```
import numpy as np
a = np.array([1, 2, 3, 4])
print(type(a))
print(a.dtype)
print(a)

b = np.array([1, 2, 3.0, 4.0])
print(b.dtype)
print(b)

c= np.array([1234567890123, 0])
print(c.dtype)
print(c)

l = [12345678901231234567891011212333, 0]
print(type(l))
c= np.array(l)
print(c.dtype)
print(c)
```

```
<class 'numpy.ndarray'>
int32
[1 2 3 4]
float64
[1. 2. 3. 4.]
int64
[1234567890123                0]
<class 'list'>
object
[12345678901231234567891011212333 0]
```

```
import numpy as np
l = [12345678901234567890, 10]
print(type(l))
c = np.array(l)
print(c.dtype)
print(c)

<class 'list'>
float64
[1.23456789e+19
1.00000000e+01]
```

- Type() function 에 대한 결과

list vs numpy.ndarray

각 element의 type numpy.int32

- 1-D array의 shape (n,)

```
import numpy as np
a = [1, 2, 3, 4]
b = [1, 'abc', 2, 3]
print(type(b))
print(type(b[0]), type(b[1]))

c = np.array(a)
print('c array type=', type(c))
print(type(c[0]), type(c[1]))
print(c.ndim, c.shape)
print(c.dtype)
```

```
<class 'list'>
<class 'int'> <class 'str'>
c array type= <class 'numpy.ndarray'>
<class 'numpy.int32'> <class 'numpy.int32'>
1 (4,)
int32
d= [1. 2. 3. 4.]
```

0차원

numpy.array 생성시 단일값(scalar value)를 넣으면 arrary 타입이 아니 일반 타입을 만듬

Column: 열

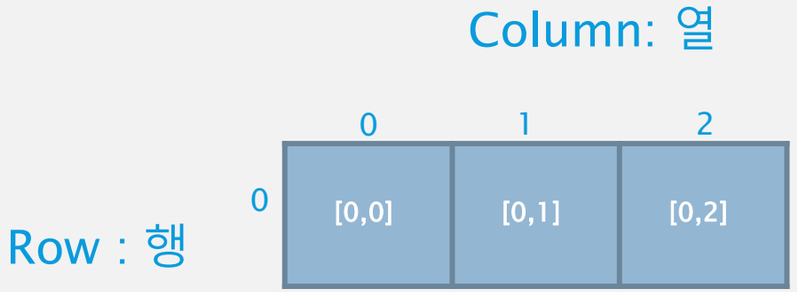
Row : 행


```
import numpy as np
a = np.array(10)
print(a)
print(a.ndim)
```

```
10
0
```

1차원

배열의 특징. 차원, 형태, 요소를 가지고 있음
생성시 데이터와 타입을 넣으면 ndim(차원)으로 확인




```python
import numpy as np

l = [1,2,3,4]
a = np.array(l)
print(a)
print(a.ndim)
```

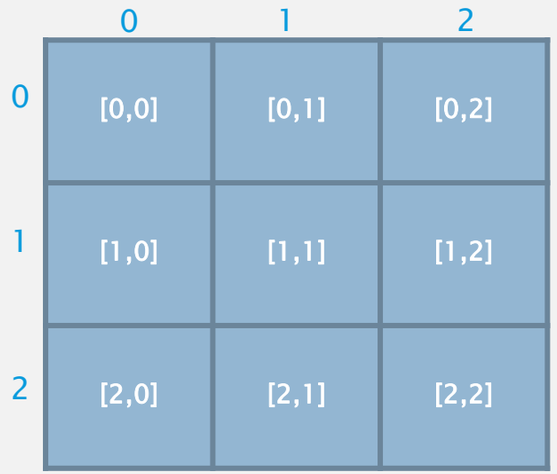
```
[1 2 3 4]
1
```

2차원 배열

– 3행, 3열의 배열을 기준으로 어떻게 내부를 행과 열로 처리하는 지를 이해,
- 맨 뒤의 index 가 열방향으로 움직인다.

Column: 열

Row : 행



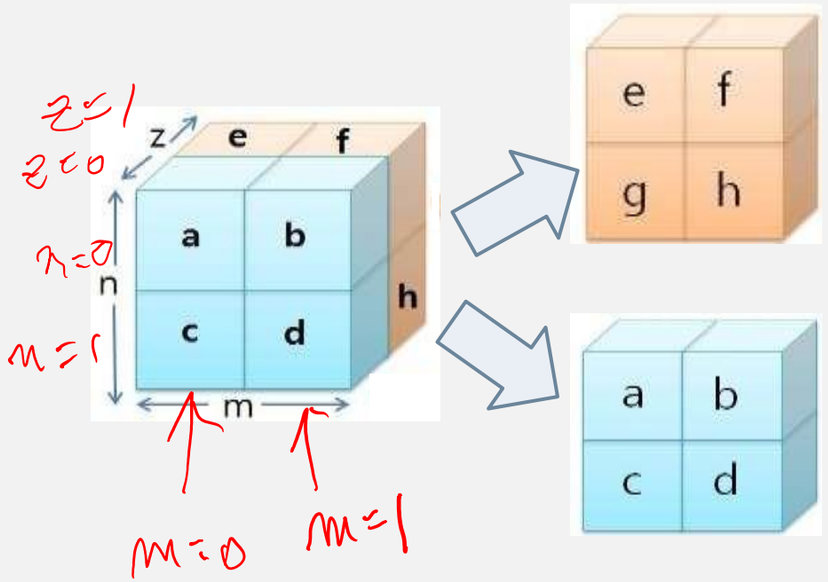

```
import numpy as np
a = np.array([[1,2],[3,4]])
print(a)
print(a.ndim)
```

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

Index 접근 표기법
배열명[행][열]
배열명[행, 열]
Slice 접근 표기법
배열명[슬라이스, 슬라이스]

3차원

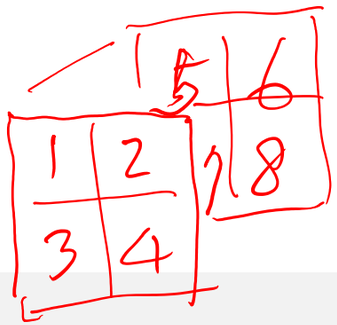
numpy.array 생성시 sequence 각 요소에 대해 접근변수와 타입을 정할 수 있음




```
import numpy as np
a = np.array([[[1,2],[3,4]], [[5,6],[7,8]]])
print(a)
print(a.ndim)
```

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

 [[5 6]
  [7 8]]]
3
```



index (z, n, m) → 원하는 값 (숫자)

In [37]:

```
aa1 = np.array([[1, 2, 3], [5, 6, 7]])
bb1 = np.array([[1, 2, 3], [4, 5, 6.12345678912345678]], float)
cc1 = np.array([[1, 2, 3], [4, 5, 6.12345678912345678]], dtype=np.float64)
dd1 = np.array([[1, 2, 3], [4, 5, 6.12345678912345678]], dtype=np.float32)
print(aa1)
print(bb1)
print(cc1)
print(dd1)
print(type(aa1))
print(aa1.ndim, aa1.shape)
dd=aa1.astype(float)
print('dd=', d)
```

```
[[1 2 3]
 [5 6 7]]
[[1.         2.         3.        ]
 [4.         5.         6.12345679]]
[[1.         2.         3.        ]
 [4.         5.         6.12345679]]
[[1.       2.       3.      ]
 [4.       5.       6.123457]]
<class 'numpy.ndarray'>
2 (2, 3)
dd= [[1. 2. 3.]
 [5. 6. 7.]]
```

In [44]:

```
a11 = np.array([[12345678910123456789012345678901, 2, 3], [5, 6, 7]])
b11 = np.array([[12345678910123456789012345678901, 2, 3], [5, 6, 7]], np.int64)
print(a11)
print(b11)
```

```
---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
<ipython-input-44-ddbb767bb739> in <module>
      1 a11 = np.array([[12345678910123456789012345678901, 2, 3], [5, 6, 7]])
----> 2 b11 = np.array([[12345678910123456789012345678901, 2, 3], [5, 6, 7]],
np.int64)
      3 print(a11)
      4 print(b11)

OverflowError: int too big to convert
```

```
b11=np.array([[12345678901234567890, 2, 3], [5, 6, 7], np.int64])
print(b11)
c11=np.array([[12345678901234567890, 2, 3], [5, 6, 7]])
print(c11)
```

```
[list([12345678901234567890, 2, 3]) list([5, 6, 7]) <class 'numpy.int64'>]
[[1.23456789e+19 2.00000000e+00 3.00000000e+00]
 [5.00000000e+00 6.00000000e+00 7.00000000e+00]]
```

할당은 참조만 전달 (ARRAY 내용을 카피하지 않는다)

Ndarray 타입을 검색이나 슬라이싱은 참조만 할당 하므로 변경을 방지하기 위해서는 새로운 ndarray 로 만들어 사용. copy 메소드가 필요

```
l = [1, 2, 3, 4]
a = np.array(l)
s = a[:2]
ss = a[:2].copy()
s[0] = 99
print("l=", l)
print("a=", a)
print("s=",s)
print("ss=",ss)
```

```
l= [1, 2, 3, 4]
a= [99  2  3  4]
s= [99  2]
ss= [1 2]
```

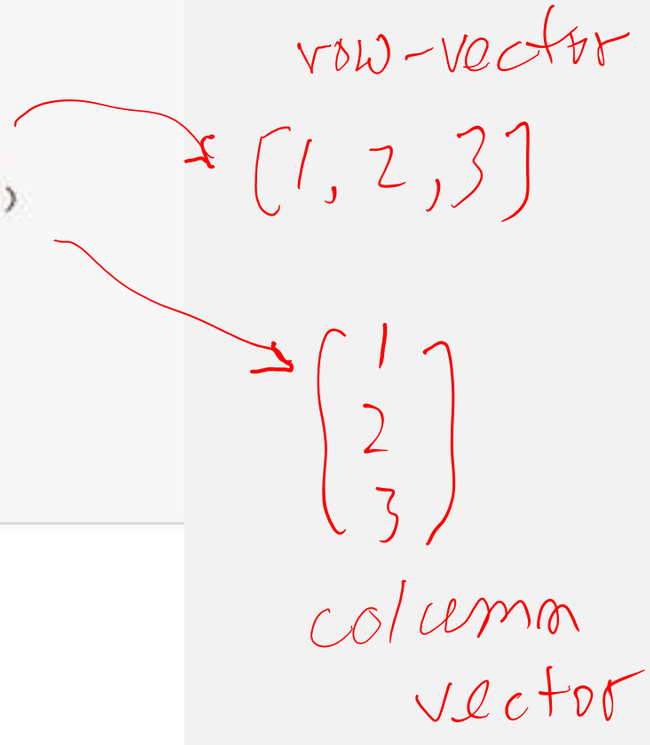
배열(1차원)과 다차원 구분 : NDARRAY (중요)


```
import numpy as np

a1 = np.array( [1, 2, 3] )
 #크기 (3,)인 1차원 배열
a2 = np.array( [ [1, 2, 3] ] )
 #크기 (1,3)인 2차원 배열 (행벡터)
a3 = np.array( [ [1], [2], [3] ] )
#크기 (3,1)인 2차원 배열 (열벡터)

print(a1)
print(a1.ndim, a1.shape)
print(a2)
print(a2.ndim, a2.shape)
print(a3)
print(a3.ndim, a3.shape)
```

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



ARRAY ELEMENT TYPE (DTYPE)

NumPy arrays comprise elements of a single data type

The type object is accessible through the `.dtype` attribute

Here are a few of the most important attributes of dtype objects

`dtype.byteorder` — big or little endian

`dtype.itemsize` — element size of this dtype

`dtype.name` — a name for this dtype object

`dtype.type` — type object used to create scalars

There are many others...

NDARRAY 생성 : 주요 변수 1

Ndarray 생성시 shape, dtype,strides이 인스턴스 속성이 생성됨

```
import numpy as np

l = [1,2,3,4]
a = np.array(l)
print(a)
print(type(a))
print(a.ndim)
print(a.shape)
print(a.size)
print(a.dtype)
print(a.itemsize)
print(a.data)
```

```
[1 2 3 4]
<class 'numpy.ndarray'>
1
(4,)
4
int32
4
<memory at 0x0000000005F95708>
```

변수	Description
ndarray.ndim	ndarray 객체에 대한 차원
ndarray.shape	ndarray 객체에 대한 다차원 모습
ndarray.size	ndarray 객체에 대한 원소의 갯수
ndarray.dtype	ndarray 객체에 대한 원소 타입
ndarray.itemsize	ndarray 객체에 대한 원소의 사이즈
ndarray.data	Python buffer object pointing to the start of the array's data.

shape

The shape tool gives a tuple of array dimensions and can be used to change the dimensions of an array.

(a). Using shape to get array dimensions

```
my__1D_array = numpy.array([1, 2, 3, 4, 5])
print my_1D_array.shape     #(5,) -> 1 row and 5 columns

my__2D_array = numpy.array([[1, 2],[3, 4],[6,5]])
print my_2D_array.shape     #(3, 2) -> 3 rows and 2 columns
```

(b). Using shape to change array dimensions

```
change_array = numpy.array([1,2,3,4,5,6])
change_array.shape = (3, 2)
print change_array

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

reshape

The reshape tool gives a new shape to an array without changing its data. It creates a new array and does not modify the original array itself.

```
import numpy

my_array = numpy.array([1,2,3,4,5,6])
print numpy.reshape(my_array,(3,2))

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

NDARRAY 생성 : 주요 변수 2

일차원과 다차원의 원소 개수를 len()함수로 처리시 다른 결과가 나옴

```
import numpy as np

l = [1,2,3,4]
a = np.array(l)
print(a)
print(type(a))
print(a.real)
print(a.imag)
print(a.strides)
print(a.base)
print(a.flat)
print(a.T)
```

```
[1 2 3 4]
<class 'numpy.ndarray'>
[1 2 3 4]
[0 0 0 0]
(4,)
None
<numpy.flatiter object at 0x000000000683DA00>
[1 2 3 4]
```

변수	Description
ndarray.real	ndarray 에 생성된 복소수에서 실수값
ndarray.imag	ndarray 에 생성된 복소수에서 허수값
ndarray.strides	ndarray 객체에 대한 원소의 크기
ndarray.base	ndarray 객체에 다른 곳에 할 당할 경우 그 원천에 대한 것을 가지고 있음
ndarray.flat	ndarray 객체가 차원을 가질 경우 하나로 연계해서 index로 처리
ndarray.T	ndarray 객체에 대한 transposed

ndarray.base
Base object if memory is from some other object.

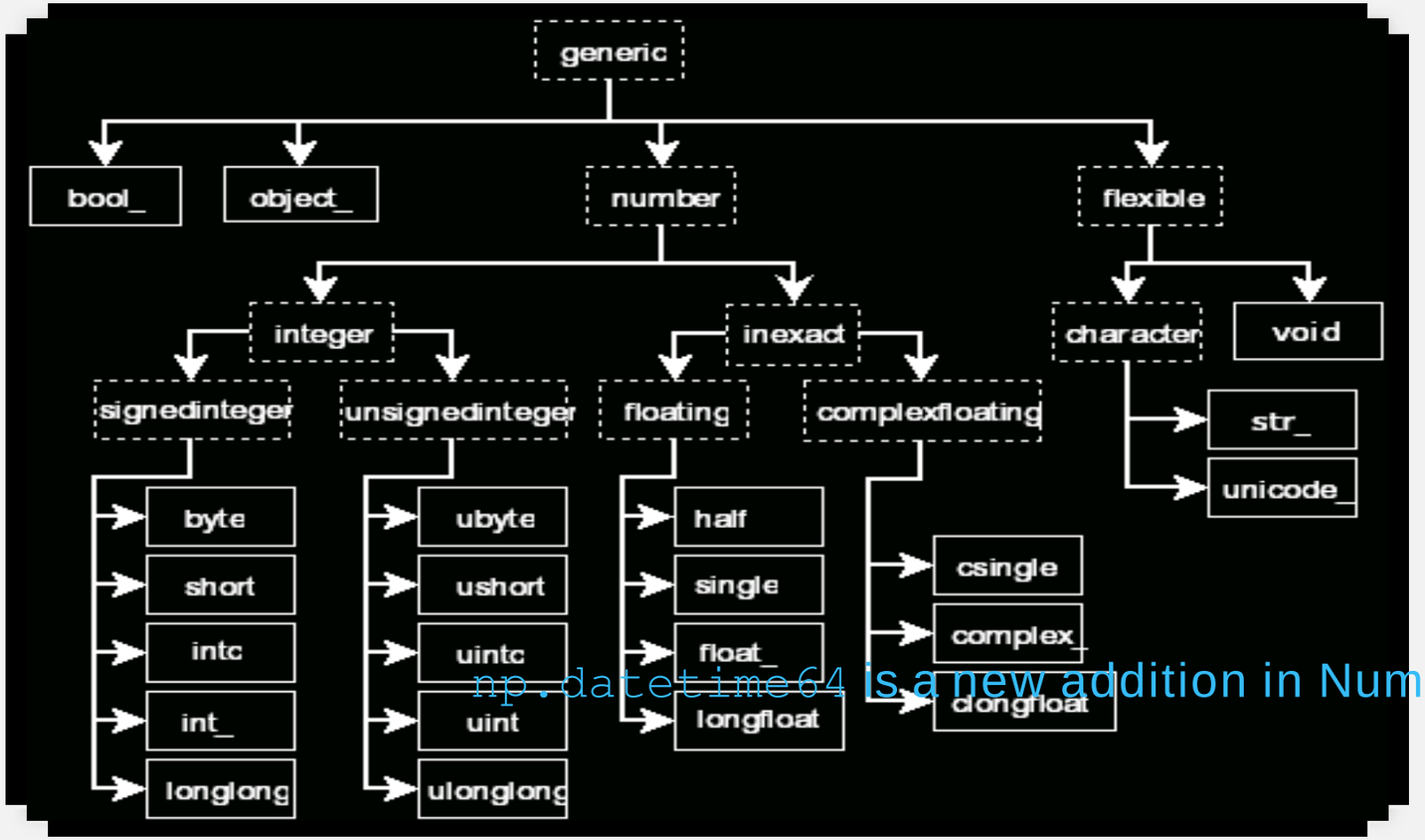
LEN, NP.SHAPE 의 차이

- len(array) 함수는 첫번째 index의 크기를 보낸다.
- numpy.shape(array) 함수는 전체 dimension을 알려준다.
- Object의 변수를 사용할 수도 있다. a.shape

```
a = np.array([[1., 2., 3.], [4., 5., 6.]])
print(len(a))
print(len(a.T))
print(np.shape(a))
print(np.shape(a.T))
print(a.shape)
print(a.T.shape)
```

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

NUMPY BUILTIN DTYPE HIERARCHY



`np.datetime64` is a new addition in NumPy 1.7

ARRAY CREATION

EXPLICITLY FROMA LIST OF VALUES

```
In [2]: np.array([1,2,3,4])
Out[2]: [1, 2, 3, 4]
```

As a range of values

```
In [3]: np.arange(10)
Out[3]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

By specifying the number of elements

```
In [4]: np.linspace(0, 1, 5)
Out[4]: [ 0.  , 0.25,    0.5 ,   0.75,   1.  ]
```

numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)[source]
Return evenly spaced numbers over a specified interval.

Returns num evenly spaced samples, calculated over the interval [start, stop].

ZERO-INITIALIZED

```
In [4]: np.zeros((2,2))
Out[4]:
[[ 0.,  0.],
[ 0.,  0.]]
```

One-initialized

```
In [5]: np.ones((1,5))
Out[5]: [[ 1.,  1.,  1.,  1.,  1.]]
```

Uninitialized

```
In [4]: np.empty((1,3))
Out[4]: [[ 2.12716633e-314,  2.12716633e-314,  2.15203762e-314]]
```

메모리에 값을 하나도 안 써 넣었기 때문에 약간 더 빠르게 생성될 수 있다. Random number 만드는 것과는 다르다.

CONSTANT DIAGONAL VALUE

```
In [6]: np.eye(3)
Out[6]:
array([[ 1.,  0.,  0.],
[ 0.,  1.,  0.],
[ 0.,  0.,  1.]])
```

```
aaa = np.eye(3)
print(aaa)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Multiple diagonal values

```
In [7]: np.diag([1,2,3,4])
Out[7]:
array([[1, 0, 0, 0],
[0, 2, 0, 0],
[0, 0, 3, 0],
[0, 0, 0, 4]])
```

(중요) RESHAPE – ARRAY의 차원을 바꾼다. 내용은 원래의 BASE에 의해서 정해진다.

서울대학교 SEOUL NATIONAL UNIVERSITY

In [37]:

```
import numpy as np
a = np.arange(16)
b = a.reshape(4, 4)
print(a, "₩n")
print(b, "₩n")
a[2] = 99
b[1, 1] = 77
print(b, "₩n")
print(a, "₩n")
print(b.base is a)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]

[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]

[[ 0  1 99  3]
 [ 4 77  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]

[ 0  1 99  3  4 77  6  7  8  9 10 11 12 13 14 15]

True
```

- Method를 이용 – good

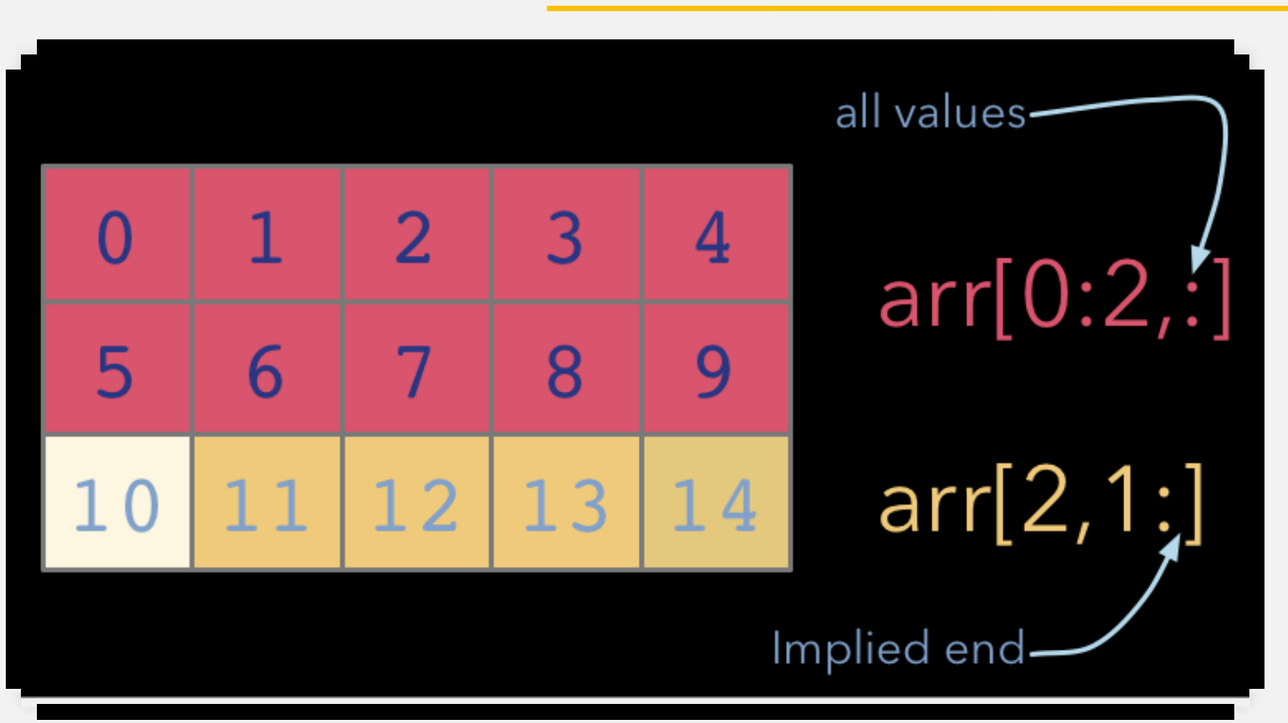
- np.arange 함수의 사용 방법은 사실 range 함수의 사용 방법과 동일합니다.
- np.arange(시작점(생략 시 0), 끝점(미포함), step size(생략 시 1)) 인자를 넣어주면 됩니다.
- range 함수에는 정수 단위만 지원하나, np.arange는 실수 단위도 표현 가능하다는 점입니다.
- 두번째로, range 메소드는 range iterator 자료형을 반환하고, np.arange 메소드는 numpy array 자료형을 반환합니다. 따라서, np.arange 메소드 결과는 넘파이에서 수행하는 연산 연계가 가능합니다.

ACCESS (RESHAPE과 동일한 효과, RESHAPE이 나은 방법)

```
import numpy as np
a = np.arange(16)
print(a)
print(a.shape)
b = a
b.shape = 2, -1  #2, 8 과 동일
print(b)
c = a
c.shape = -1, 2
print(c)
c.shape = 4, -1
print(c)
print(c.shape)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13
14 15]
(16,)
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]]
[[ 0  1]
 [ 2  3]
 [ 4  5]
 [ 6  7]
 [ 8  9]
 [10 11]
 [12 13]
 [14 15]]
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
(4, 4)
```

INDEXING AND SLICING



두번째 index
axis = 1

첫번째 index
axis = 0

참고 2-D list indexing
a[0][1] = 9

ndarray는 둘다 된다.

배열 접근하기 : 행과 열구분

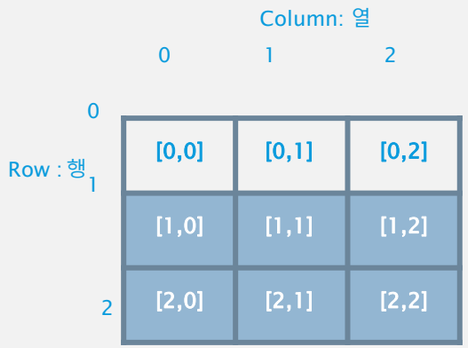
배열명[행 범위, 열 범위] 행으로 접근, 열로 접근

```
l33 = [[1,2,3], [4, 5,6], [7,8,9]]
a33 = np.array(l33)
print("first row = ", a33[0])
print("middle column = ", a33[:, 1])
print("first element = ", a33[0, 0])
```

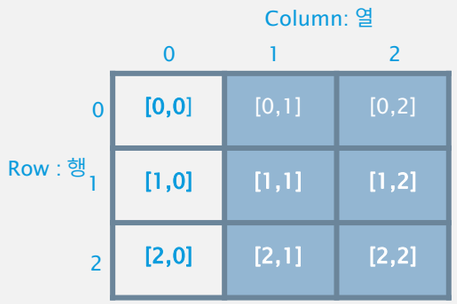
```
first row =  [1 2 3]
middle column =  [2 5 8]
first element =  1
```

#ndim becomes 1

첫번째 행 접근



첫번째 열 접근



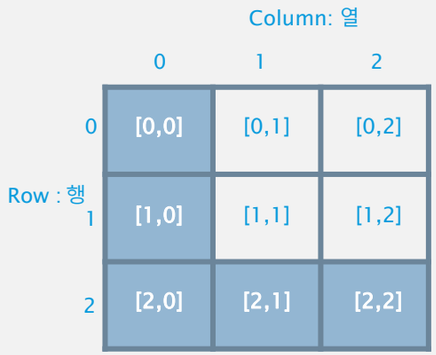
배열 접근하기 : 행렬로 구분

첫번째와 두번째 행과 두번째와 세번째 열로접근

```
import numpy as np
l33 = [[1,2,3],[4,5,6],[7,8,9]]
np33 = np.array(l33,int)
print np33

print np33[:2, 1:]
```

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



LIST INDEXING 과 NDARRAY INDEXING의 차이점

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

```
In [29]:
```

```
a=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
b = np.array(a)
print(b[0][0])
print(a[0][0])
print(b[0])
print(a[0])
print(b[0,2])
print(a[0,2])
```

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


---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-29-9667d58cd271> in <module>
      6 print(a[0])
      7 print(b[0,2])
----> 8 print(a[0,2])

TypeError: list indices must be integers or slices, not tuple
```

- List의 경우는 a[n, m]이 허용되지 않는다. a[n][m]

- ndarray의 경우는 a[n, m]이 허용이 되고, 더 효율적이다.

note that x[0,2] = x[0][2] though the second case is more inefficient as a new temporary array is created after the first index that is subsequently indexed by 2.

For simple indexing of a 2d array
both forms work:

```
In [28]: x = np.arange(6).reshape(2,3)
In [29]: x
Out[29]:
array([[0, 1, 2],
       [3, 4, 5]])
In [30]: x[1,2]
Out[30]: 5
In [31]: x[1][2]
Out[31]: 5
```

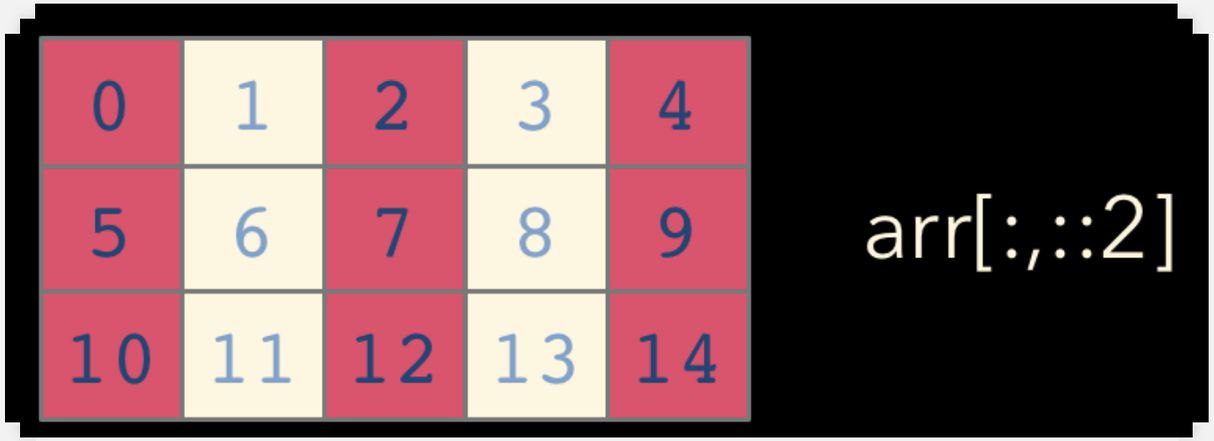
For np.matrix (which you probably shouldn't be using anyways) they aren't:

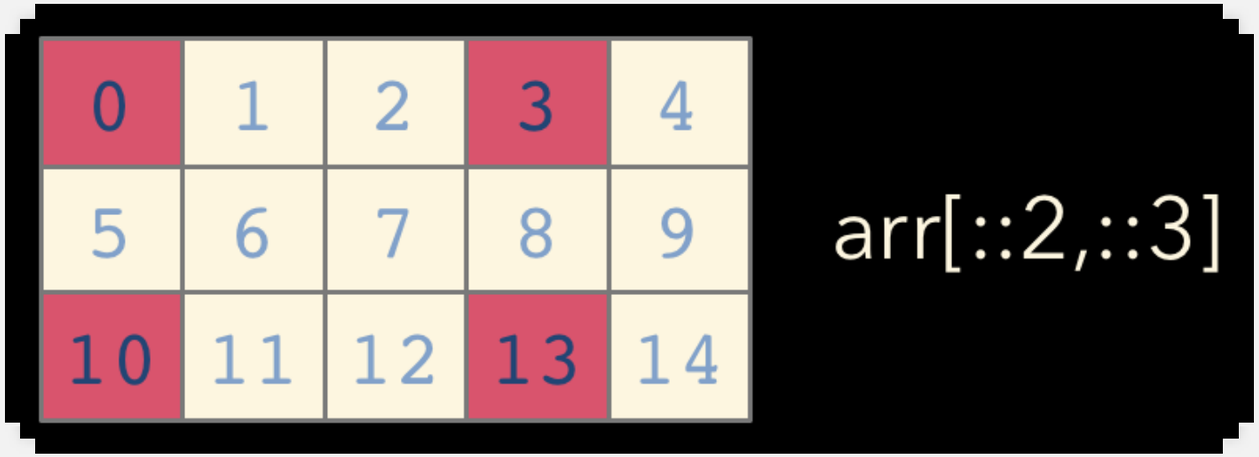
```
In [32]: X = np.matrix(x)
In [33]: X
Out[33]:
matrix([[0, 1, 2],
        [3, 4, 5]])
In [34]: X[1,2]
Out[34]: 5
In [35]: X[1][2]
...
IndexError: index 2 is out of bounds for axis 0 with size 1
```

The two forms are not syntactically the same. [1][2] first indexes with 1, and then indexes the result with 2. That's not the same as indexing once with both parameters.

```
In [36]: x[1]
Out[36]: array([3, 4, 5])      # (3,) shape
In [37]: X[1]
Out[37]: matrix([[3, 4, 5]])   # (1,3) shape
```

NUMPY ARRAY INDICES CAN ALSO TAKE AN OPTIONAL STRIDE





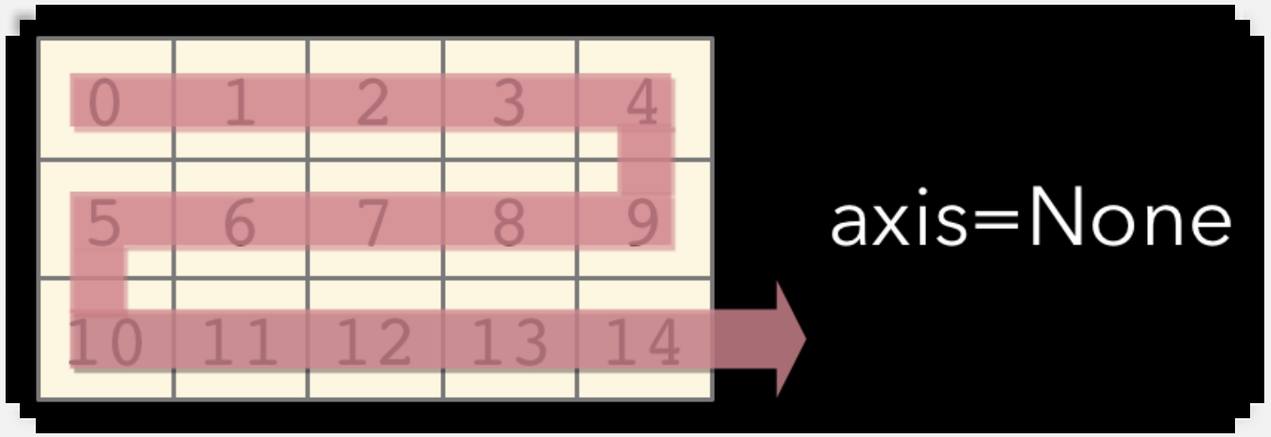
AXIS (중요)

Array method reductions take an optional `axis` parameter that specifies over which axes to reduce

`axis=None` reduces into a single scalar

Axis None is the default

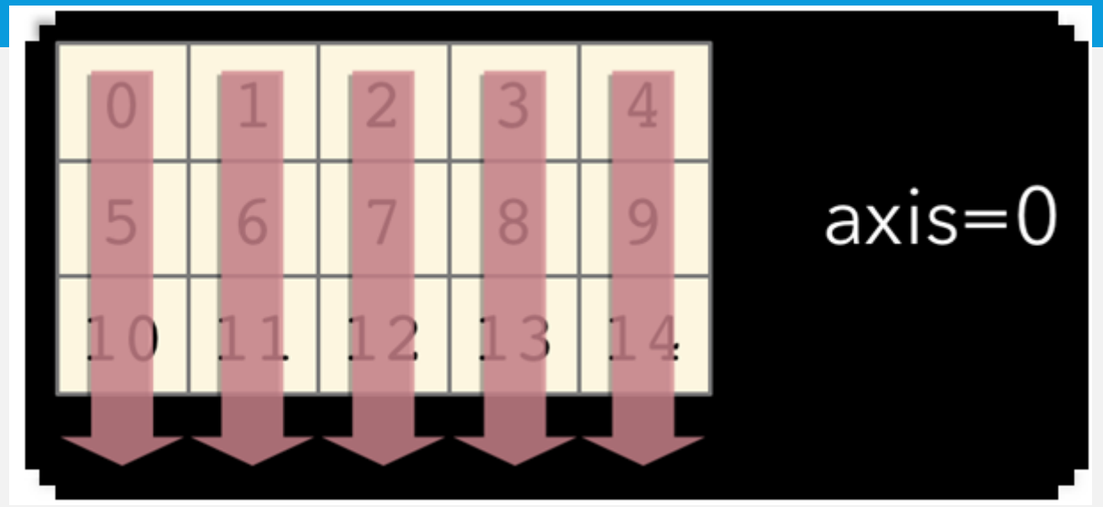
```
In [7]: a.sum()
Out[7]: 105
```



참고로 memory에 데이터가 저장이 되는 순서이다.

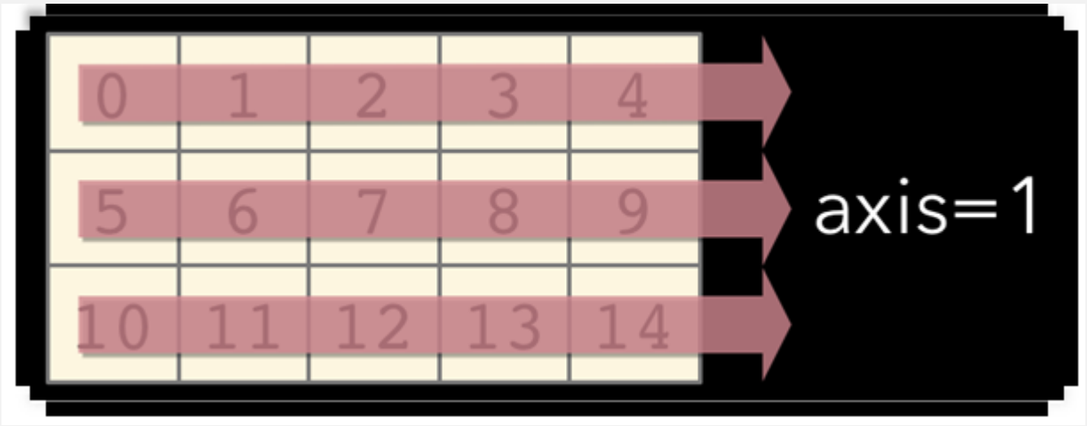
AXIS=0 REDUCES INTO THE ZEROTH DIMENSION

```
In [8]: a.sum(axis=0)
Out[8]: array([15, 18, 21, 24,
27])
```



axis=1 reduces into the first dimension

```
In [9]: a.sum(axis=1)
Out[9]: array([10, 35, 60])
```



```
a = np.arange(8)
b = a.reshape(2, 4)
print(b)
print(a.sum(), a.sum().shape)
print(b.sum(), b.sum().shape)
print(b.sum(axis=0), b.sum(axis=0).shape)
print(b.sum(axis=1), b.sum(axis=1).shape)
```

```
[[0 1 2 3]
 [4 5 6 7]]
28 ()
28 ()
[ 4  6  8 10] (4,)
[ 6 22] (2,)
```

NUMPY – ARRAY.SORT()

```
>>> f = array([3, 7, 4, 8, 2, 15])


>>> f
array([ 3,  7, 4, 8, 2, 15])


>>>   f.sort()


>>> f
array([ 2,  3, 4, 7, 8, 15])
```

- (1) 1차원 배열 정렬 : np.sort(x)

- (2) 1차원 배열 거꾸로 정렬 : np.sort(x)[::-1] , x[np.argsort(-x)]

- (3) 2차원 배열 열 축 기준으로 정렬 : np.sort(x, axis=1)

- (4) 2차원 배열 행 축 기준으로 정렬 : np.sort(x, axis=0)

- (5) 2차원 배열 행 축 기준으로 거꾸로 정렬 : np.sort(x, axis=0)[::-1]

.

출처: https://rfriend.tistory.com/357 [R, Python 분석과 프로그래밍의 친구 (by R Friend)]

```
a = np.array([1, 3, 5, 7, 6, 2])
b = a.reshape(2, 3)
print(b.sort(axis = 0))
print(b)
print(a)
```

```
None
[[1 3 2]
 [7 6 5]]
[1 3 2 7 6 5]
```

```
a = np.array([0, 2, 4, 3, 1, 5, 7, 9])
b = a.reshape(2, 4)


print(np.sort(a)[::-1]) #[9 7 5 4 3 2 1 0]
print(a)                #[0 2 4 3 1 5 7 9]
print(' b =', b)
print(np.sort(b, axis=0))
print(np.sort(b, axis=1)[::-1])
----..
 b = [[0 2 4 3]
 [1 5 7 9]]
[[0 2 4 3]
 [1 5 7 9]]
[[1 5 7 9]
 [0 2 3 4]]
```

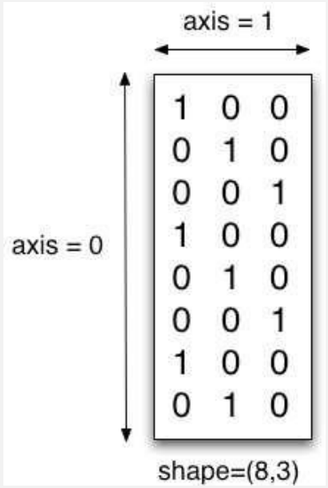
NUMPY – ARRAY.MAX()

```
>>> g = array([[1, 2], [3, 4]])

>>> g
array([[1, 2],
       [3, 4]])
>>>                  # axis=None 배열 전체에서 가장 큰 스칼라 값
g.max()
4


 >>>
 g.max(axis=0)
 array([3, 4])

 >>>
 g.max(1)
 array([2,
 4])
```



http://pages.physics.cornell.edu/~myers/teaching/ComputationalMethods/python/arrays.html

In [31]:

```
b_1d= np.array([1, 2, 3, 5, 6, 7])
b_2d = np.array([[1, 2, 3], [5, 6, 7]])
print('1d result =', b_1d.max(), b_1d.min(), b_1d.sum(), b_1d.mean())

print('2d result =', b_2d.max(), b_2d.min(), b_2d.sum(), b_2d.mean())
print('2d result in each row',b_2d.max(axis=0), b_2d.min(axis=0), b_2d.sum(axis=0), b_2d.mean(axis=(
print('2d result in each column',b_2d.max(axis=1), b_2d.min(axis=1), b_2d.sum(axis=1), b_2d.mean(axi
```

```
1d result = 7 1 24 4.0
2d result = 7 1 24 4.0
2d result in each row [5 6 7] [1 2 3] [ 6  8 10] [3. 4. 5.]
2d result in each column [3 7] [1 5] [ 6 18] [2. 6.]
```

NUMPY – ARRAY.TRANSPOSE()

```
>>> e = array([[1, 2], [3, 4]])


>>> e
array([[1, 2],
       [3, 4]])

>>> e.transpose()

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

좀 더 부가 설명을 하자면 **ndarray를 이용하여 벡터(vector)를 표현할 때는 2차 배열로 정의해야 한다**. 즉, 다음 세 가지는 모
세 번째 같이 생성해야 한다. (혼동하기 참 쉽다.)

```
a1 = np.array( [1, 2, 3] ) #크기 (3,)인 1차원 배열
a2 = np.array( [ [1, 2, 3] ] ) #크기 (1,3)인 2차원 배열 (행벡터)
a3 = np.array( [ [1], [2], [3] ] ) #크기 (3,1)인 2차원 배열 (열벡터)
```

여기서 **a1.T 는 동작하지 않는다**. 반면 a2.T 와 a3.T는 동작한다. 1차 배열은 행벡터나 열벡터 두 가지 모두로 취급되기도 한다

ndarray 객체를 사용하는데 있어서 장점과 단점은 다음과 같다.

- 장점
  - 1차 배열은 행벡터나 열벡터 둘 다로 취급할 수 있다. dot(A,v) 에서 v는 열벡터로 다루어지고 dot(v,A)에서는 행
    잡하게 수행할 필요가 없다.
  - 요소 간 곱셈이 쉽다 (예: A*B) 사실 모든 산술 연산 (+ - * / ** 등등)이 요소 간 연산이다.
  - ndarray 는 numpy 의 가장 기본 객체이므로 연산의 속도, 효율성 그리고 numpy를 이용하는 외부 라이브러리의 빈
  - 다차원 배열을 쉽게 구현한다.
  - tensor algebra 에 대한 장점이 있다.(?)
- 단점
  - 행렬간 곱에 obj.dot() 멤버함수를 사용해야 하므로 번잡할 수 있다. 세 행렬 A, B, C의 행렬곱은 dot(dot(A,B),C) 이

1. 인공지능과 선형대수

2. NUMPY, NDARRAY

3. NUMPY ARRAY를 이용한 일반연산

4. MATRIX-VECTOR, MAT-MAT 연산

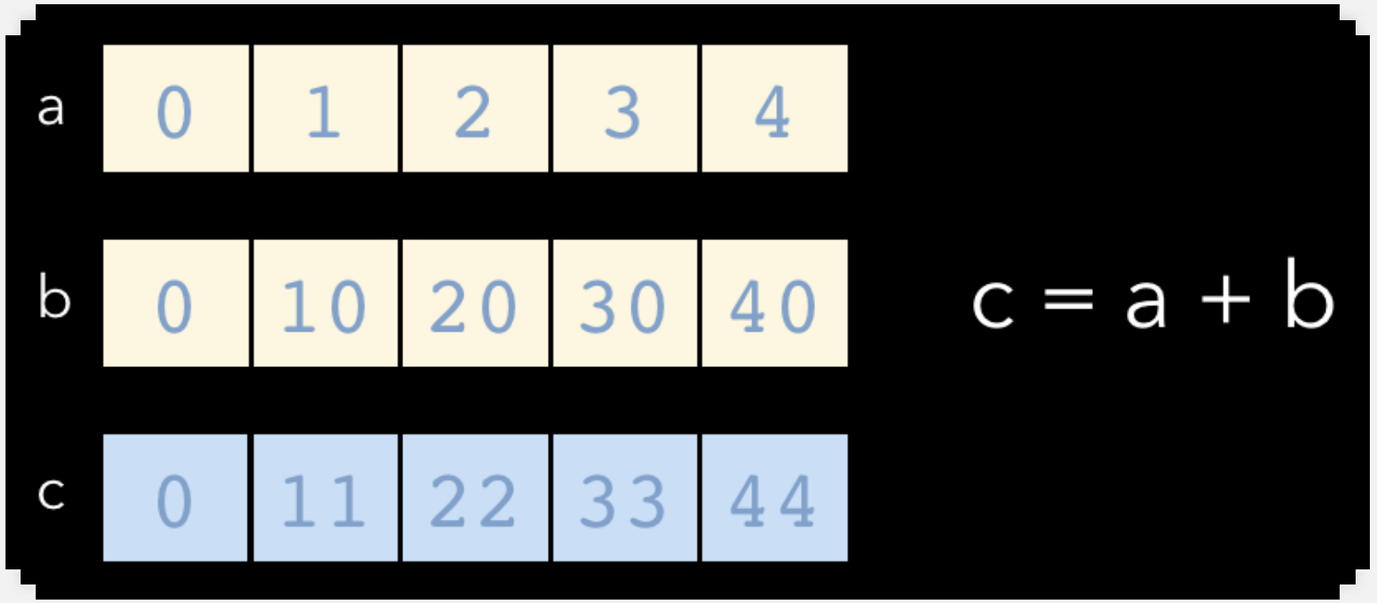
3부 NUMPY ARRAY를 이용한 연산

서울대학교 SEOUL NATIONAL UNIVERSITY

- Universal functions – 각 element-wise 하게 함수 적용
- Array methods – sum of array, transpose of array, ...
- Broadcasting - array 와 constant 숫자의 사칙연산, power, mod 등
- Array와 array 간의 연산 (둘의 shape이 같아야 함) -> element wise 연산 (+, -, *, /)

3.1 UNIVERSAL FUNCTIONS (UFUNCS)

NUMPY UFUNCS ARE FUNCTIONS THAT OPERATE ELEMENT-WIS
E



ufuncs dispatch to optimized C inner-loops based on array dtype

NUMPY HAS MANY BUILT-IN UFUNCS

- comparison: `<, <=, ==, !=, >=, >`
- arithmetic: `+, -, *, /, reciprocal, square`
- exponential: `exp, expm1, exp2, log, log10, log1p, log2, power, sqrt`
- trigonometric: `sin, cos, tan, acsin, arccos, atctan`
- hyperbolic: `sinh, cosh, tanh, acsinh, arccosh, atctanh`
- bitwise operations: `&, |, ~, ^, left_shift, right_shift`
- logical operations: `and, logical_xor, not, or`
- predicates: `isfinite, isinf, isnan, signbit`
- other: `abs, ceil, floor, mod, modf, round, sinc, sign, trunc`

3.2 ARRAY METHODS

서울대학교 SEOUL NATIONAL UNIVERSITY

- Predicates
  - `a.any(), a.all()`
- Reductions
  - `a.mean(), a.argmin(), a.argmax(), a.trace(), a.cumsum(), a.cumprod()`
- Manipulation
  - `a.argsort(), a.transpose(), a.reshape(...), a.ravel(), a.fill(...), a.clip(...)`
- Complex Numbers
  - `a.real, a.imag, a.conj()`

```
import numpy as np
a = np.array([0, 1, 2, 3])
print(a.all(), a.any())
b = np.array([-1, 1, 2, 3])
print(b.all(), b.any())
c = np.array([0,0])
print(c.all(), c.any())
```

```
False True
True True
False False
```

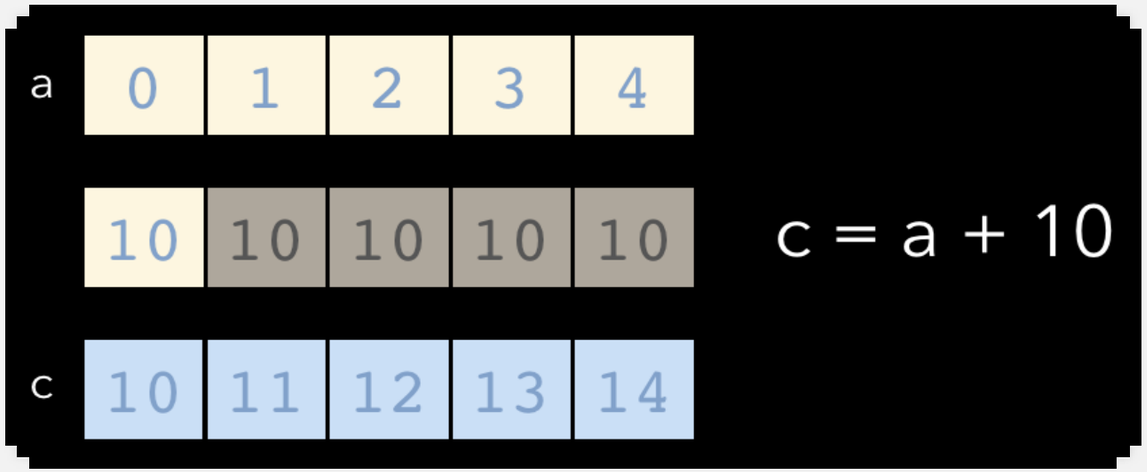
```
# NOTE : 절댓값 함수
x = np.array([-2, -1, 0, 1, 2])
y = np.array([3-4j, 4-3j, 2+0j, 0+1j])
print(abs(x))
print(np.abs(x))
print(np.absolute(x))
print(abs(y))
print(np.abs(y))
print(np.absolute(y))
Out [1] :
[2 1 0 1 2]
[2 1 0 1 2]
[2 1 0 1 2]
[5. 5. 2. 1.]
[5. 5. 2. 1.]
[5. 5. 2. 1.]
# NOTE : 삼각함수
theta = np.linspace(0, np.pi, 3) # NOTE : 0부터 pi까지 균등하게 3개의 원소로 구성된 배열을 만듦.
print("theta = ", theta)
print("sin(theta) = ", np.sin(theta))
print("cos(theta) = ", np.cos(theta))
```

일단, 정수 및 실수에 대해서는 절대값 정의대로 양수 혹은 0이면 그대로, 음수이면 -1을 곱하여 양수로 바뀐 값이 반환됩니다.
그러나, 복소수의 경우에는 절댓값 연산이 다소 다른 방식으로 수행되는데, 실수부와 허수부의 제곱합에 루트를 취한 크기가 반환됩니다.

```
# NOTE : 지수와 로그함수
x = [1, 2, 3]
print("x = ", x)
print("e^x = ", np.exp(x))
print("2^x = ", np.exp2(x))
print("3^x = ", np.power(3,x))
print('\n')
x = [1, 2, 4, 10]
print("x = ", x)
print("ln(x) = ", np.log(x))
print("log2(x) = ", np.log2(x))
```

3.3 BROADCASTING

DIMENSION 이 다를 때 (대체로 하나는 SCALAR 일 때 – 모든 ELEMENT로 전



In this case an array scalar is broadcast to an array with shape `(5,)`

```
a = np.arange(8)
b = a.reshape(2,4)
c = b/2
print(c)
------------
[[0.  0.5 1.  1.5]
 [2.  2.5 3.  3.5]]
```

BROADCASTING OF SCALAR (+, -, *, /)

```
a = np.arange(8)
b = a.reshape(2,4)
c = b/2
print(c)
------------
[[0.  0.5 1.  1.5]
 [2.  2.5 3.  3.5]]
```

```
print("x-5 = ", x - 5)
print("x*2 = ", x * 2)
print("x/2 = ", x / 2)
print("x//2 = ", x // 2) # NOTE : 바닥 나눗셈(나머지는 버림)
print("-x = ", -x)
print("x^2 = ", x**2)
print("x%2 = ", x % 2)
```

```
Out [1] :
x = [0 1 2 3]
x+5 = [5 6 7 8]
x-5 = [-5 -4 -3 -2]
x*2 = [0 2 4 6]
x/2 = [0. 0.5 1. 1.5]
x//2 = [0 0 1 1]
-x = [ 0 -1 -2 -3]
x^2 = [0 1 4 9]
x%2 = [0 1 0 1]
```

```
print("x*2 = ", np.multiply(x, 2))
print("x/2 = ", np.divide(x, 2))
print("x//2 = ", np.floor_divide(x, 2))
print("-x = ", -x)
print("x^2 = ", np.power(x, 2))
print("x%2 = ", np.mod(x, 2))
```

```
Out [1] :
x = [0 1 2 3]
x+5 = [5 6 7 8]
x-5 = [-5 -4 -3 -2]
x*2 = [0 2 4 6]
x/2 = [0. 0.5 1. 1.5]
x//2 = [0 0 1 1]
-x = [ 0 -1 -2 -3]
x^2 = [0 1 4 9]
x%2 = [0 1 0 1]
```

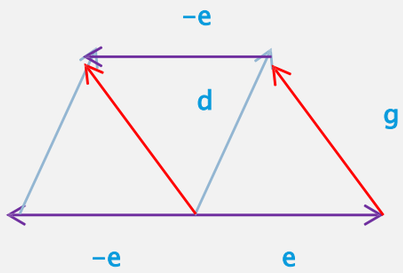
3.4 VECTOR 연산: ELEMENT-WISE OPERATION

두 벡터 반대 방향으로 평행 이동해 평행사변형을
만든 후 가운데 벡터가 실제 덧셈한 벡터를 표시

```
import math
import numpy as np

d = np.array([1,2])
e = np.array([2,1])
g = d - e
print(g)
```

```
[-1  1]
```



MULTIPLY 함수 :곱셈

multiply 함수는 1차원 ndarray에서는 *연산자와 같은 계산 결과가 나옴

```
import numpy as np

l = [1,2,3,4]
a = np.array(l,np.int)
l1 = a *2
b = np.array(l1, np.int)
print(a)
print(b)
print(np.multiply(a,b))
```

```
[1 2 3 4]
[2 4 6 8]
[ 2  8 18 32]
```

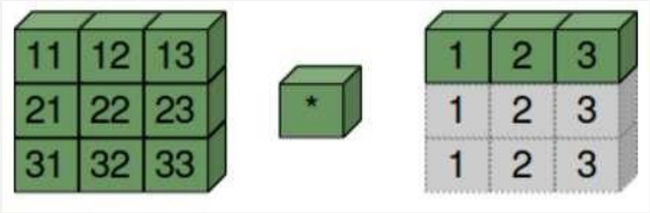
NP.DIVIDE(X, Y) ELEMENT-WISE DIVIDE AFTER BROADCASTING

```
>>> np.divide(2.0, 4.0)
0.5


>>> x1 = np.arange(9.0).reshape((3, 3))
>>> x2 = np.arange(3.0)
>>> np.divide(x1, x2)
array([[ NaN,  1. ,  1. ],
       [ Inf,  4. ,  2.5],
       [ Inf,  7. ,  4. ]])
```

브로드캐스팅 처리

원소들끼리 계산을 하기 위해 동일한 모양으로만들고 이를 원소별로 계산 처리




```
import numpy as np
A = np.array([ [11, 12, 13], [21, 22, 23], [31, 32, 33] ])
B = np.array([1, 2, 3])
print("Multiplication with broadcasting: ")
print(A * B)
print("... and now addition with broadcasting: ")
print(A + B)
```

```
Multiplication with broadcasting:
[[11 24 39]
 [21 44 69]
 [31 64 99]]
... and now addition with broadcasting:
[[12 14 16]
 [22 24 26]
 [32 34 36]]
```

BROADCASTING 응용 벡터화 연산: FOR문 사용할 필요 없다.

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

F (화씨) = C(섭씨) * 9 / 5 + 32 이 공식을 기준으로 연속적인 배열을 loop 문 없이 계산

```
cvalues = [25.3, 24.8, 26.9, 23.9]
#섭씨 ndarray 생성
C = np.array(cvalues)
print(C)
F = C * 9 / 5 + 32
print type(F), F

#기존방식, 리스트 컴프리헨션도 loop문 실행

F1 = [ x*9/5 + 32 for x in cvalues]
print type(F1), F1
```

```
[ 25.3  24.8  26.9  23.9]
<type 'numpy.ndarray'> [ 77.54  76.64  80.42  75.02]
<type 'list'> [77.54, 76.64, 80.42, 75.02]
```

ndarray 특징은 array 원소 만큼 자동으로 순환 계산해서 ndarray로 반환함

BROADCASTING RULES

In order for an operation to broadcast, the size of all the trailing dimensions for both arrays must either:

be **equal** OR be **one**

```
A.      (1d array):             3
B.(2d array):               2 x 3
Result (2d array):          2 x 3

A.      (2d array):         6 x 1
B.(3d array):           1 x 6 x 4
Result (3d array):      1 x 6 x 4

A.      (4d array):  3 x 1 x 6 x 1
B.(3d array):           2 x 1 x 4
Result (4d array):  3 x 2 x 6 x 4
```

SQUARE PEG IN A ROUND HOLE

If the dimensions do not match up, `np.newaxis` may be useful, shape을 돌려서 맞추어줌

```
In [16]: a = np.arange(6).reshape((2, 3))

In [17]: b =

np.array([10, 100])  In

[18]: a * b ----------------------------------------

ValueError                                Traceback (most recent cal
 last)  in ()
----> 1 a * b

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

(2)  In [19]: b[:,np.newaxis].shape
Out[19]: (2, 1)

In [20]: a
*b[:,np.newaxis]
Out[20]:
array([[   0,   10,   20],
[300, 400, 500]])
```

```
a = np.arange(6).reshape((2, 3))
b = np.array([10, 100]).reshape((2, 1))
print(b)
print(a*b)
-----------------
[[ 10]
 [100]]
[[  0  10  20]
 [300 400 500]]
```

[False True True] [2. 3.]

1차원 배열 : 조회

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

[f > 2.0] 조건의 원소가 True 인 것만을 조회

```
import numpy as np

f = np.array([1.0,2.0,3.0])
a = f > 2.0
print(a)

print(f[a])
```

```
[False False  True]
[ 3.]
```

옆의 프로그램에서 a = 1.9인 경우

[False True True]
[2. 3.]

각 VECTOR ELEMENT 에 조건부 처리 __GETITEM__

numpy.ndarray 타입에서는 __getitem__에 논리 연산 등 다양한 처리를 허용
- array 위치를 주면 해당값이 나온다. 또는 key를 vector로 주면 값이 나온다.

```
import numpy as np
l = range(1,10)
print type(l),l
nparr = np.arange(1,10,dtype=np.float_)
print type(nparr), nparr

print l.__getitem__(1)
arI = np.array([True,False,True,False,True,False,True,False,True,False])

print arI, arI.dtype
print nparr.__getitem__(1)
print nparr.__getitem__(arI)
```

```
<type 'list'> [1, 2, 3, 4, 5, 6, 7, 8, 9]
<type 'numpy.ndarray'> [ 1.  2.  3.  4.  5.  6.  7.  8.  9.]
2
[ True False  True False  True False  True False  True False] bool
2.0
[ 1.  3.  5.  7.  9.]
```

ndarray.__getitem__(self, key, /)
Return self[key].

__SETITEM__

Index와 slice 처리시 기존 리스트의 방식보다 더 다양한 처리를 위해__setitem__메소드를 override 함

```
import numpy as np
l = range(1,10)
print type(l),l
nparr = np.arange(1,10,dtype=np.float_)
print type(nparr), nparr

print l.__getitem__(1)

arI = np.array([True,False,True,False,True,False,True,False,True,False])
print arI, arI.dtype
print nparr.__getitem__(1)
print nparr.__getitem__(arI)

print l.__setitem__(1,100),l
print nparr.__setitem__(1,100), nparr
print nparr.__setitem__(arI, 99), nparr
```

```
<type 'list'> [1, 2, 3, 4, 5, 6, 7, 8, 9]
<type 'numpy.ndarray'> [ 1.  2.  3.  4.  5.  6.  7.  8.  9.]
2
[ True False  True False  True False  True False  True False] bool
2.0
[ 1.  3.  5.  7.  9.]
None [1, 100, 3, 4, 5, 6, 7, 8, 9]
None [   1.  100.    3.    4.    5.    6.    7.    8.    9.]
None [  99.  100.   99.    4.   99.    6.   99.    8.   99.]
```

ndarray.__setitem__(self, key, value, /)
Set self[key] to value.

다차원 배열 : 조회

[f > 0.5] 조건의 원소가 True 인 것만을 조회

```
import numpy as np

f = np.random.rand(3,4)
print(f)
a = f > 0.5
print(a)

print(f[a])
```

```
[[ 0.10333293  0.18157485  0.7166101   0.47587807]
 [ 0.173721    0.27780665  0.90892899  0.09924607]
 [ 0.0929639   0.51641182  0.29742881  0.5217259 ]]
[[False False  True False]
 [False False  True False]
 [False  True False  True]]
[ 0.7166101   0.90892899  0.51641182  0.5217259 ]
```

다차원 배열 : 변경

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

[data1 <0] = 99 실제 배열의 원소들 값이 0보다 작을 경우 99으로 전환

```
import numpy as np
data1 = np.random.randn(7,4)
print(data1)
data1[data1 < 0 ] = 99

print(data1)
```

```
[[-0.64163104 -0.23457396  0.55209566 -0.52003177]
 [ 1.03526692 -1.48943155  1.61109187  0.68909598]
 [-1.45307971  1.4563982  -0.73736959  0.18656807]
 [-0.2841279  -1.18045274  1.91581361  0.74631229]
 [ 0.2408152  -1.99151726  0.51560286  1.63974168]
 [ 0.17838888  1.10566446  2.02236861 -0.7942652 ]
 [-0.46511521  0.6138702  -1.55559268 -0.3082523 ]]
[[ 99.          99.           0.55209566  99.        ]
 [  1.03526692  99.           1.61109187   0.68909598]
 [ 99.           1.4563982   99.           0.18656807]
 [ 99.          99.           1.91581361   0.74631229]
 [  0.2408152   99.           0.51560286   1.63974168]
 [  0.17838888   1.10566446   2.02236861  99.        ]
 [ 99.           0.6138702   99.          99.        ]]
```

FANCY INDEXING

NumPy arrays may be used to index into other arrays

```
In [2]: a = np.arange(15).reshape((3,5))

In [3]: a
Out[3]:
array([[ 0,  1,  2,  3,  4],
[ 5,  6,  7,  8,  9],
[10, 11, 12, 13, 14]])
In [4]: i = np.array([[0,1], [1, 2]])

In [5]: j = np.array([[2, 1], [4, 4]])

In [6]: a[i,j]
Out[6]:
array([[ 2,  6],
[ 9, 14]])
```

In [54]:

```
aa = np.arange(15).reshape(3, 5)
i=np.array([[0, 1], [1, 2]])
j=np.array([[2, 1], [4, 4]])
print(aa, "₩n")
print(aa[i], "₩n")
print(aa[i].ndim, aa[i].shape)
print(aa[i, j])
print(aa[i, j].ndim, "₩n")
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]

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

 [[ 5  6  7  8  9]
  [10 11 12 13 14]]]

3 (2, 2, 5)
[[ 2  6]
 [ 9 14]]
2
```

BOOLEAN ARRAYS CAN ALSO BE USED AS INDICES INTO OTHER ARRAYS


```
In [2]: a = np.arange(15).reshape((3,5))

In [3]: a
Out[3]:
array([[ 0,  1,  2,  3,  4],
[ 5,  6,  7,  8,  9],
[10, 11, 12, 13, 14]])

In [4]: b = (a % 3 == 0)

In [5]: b
Out[5]:
array([[ True, False, False,  True, False],
[False,  True, False, False,  True],
[False, False,  True, False, False]], dtype=bool)

In [6]: a[b]
Out[6]: array([ 0,  3,  6,  9, 12])
```

행 검색

정수배열을 사용한 색인(양수,음수를 이용)이며 행에 대한 정보를 list로 제공해서 3번째와1번째를 출력

정방향

```
import numpy as np

f = np.arange(0,12).reshape(3,4)
print(f)

print(f[[2,0]])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[ 8  9 10 11]
 [ 0  1  2  3]]
```

역방향

```
import numpy as np

f = np.arange(0,12).reshape(3,4)
print(f)

print(f[[-1,-3]])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[ 8  9 10 11]
 [ 0  1  2  3]]
```

In [44]:

```
import numpy as np
a = np.arange(16)
b = a.reshape(4, 4)

print(b)
print(b[2, 0])
print(b[[2, 1, 0]])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
8
[[ 8  9 10 11]
 [ 4  5  6  7]
 [ 0  1  2  3]]
```

순서쌍 처리후 원소만 추출

두개의 배열을 주면 첫번째 배열은 행, 두번째 배열은 열로 순서쌍을 구성해서 값을추출

```
import numpy as np

f = np.arange(0,12).reshape(3,4)
print(f)

# (1,0),(2,2) 처리
print(f[[1,2],[0,2]])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[ 4 10]
```

```
import numpy as np
B = np.array([[142,56,189,65],
              [299,288,10,12],
              [55,142,17,18]])
#첫번째 행에 대한 위치조정 후 출력
e0 = np.array([0,0,0,0])
e1= np.array([0,3,2,1])

f = B[(e0,e1)]

print f
```

```
[142  65 189  56]
```

- NumPy arrays may be indexed with other arrays. The use of index arrays ranges from simple, straightforward cases to complex, hard-to-understand cases. For all cases of index arrays, what is returned is a copy of the original data, not a view as one gets for slices.

```
>>> x = np.arange(10,1,-1)
>>> x
array([10,  9,  8,  7,  6,  5,  4,  3,  2])

>>> x[np.array([3, 3, 1, 8])]
array([7, 7, 9, 2])

>>> x[np.array([3,3,-3,8])]
array([7, 7, 4, 2])

>>> x[np.array([[1,1],[2,3]])]
array([[9, 9],
       [8, 7]])
```

```
>>> y = np.arange(35).reshape(5,7)
>>> y[1:5:2,::3]
array([[ 7, 10, 13],
       [21, 24, 27]])

>>> y[np.array([0,2,4]), np.array([0,1,2])]
array([ 0, 15, 30])
```

https://numpy.org/doc/stable/user/basics.indexing.html

표현식 : 비교 연산

배열에 직접 비교연산 수식을 제공해서 원소추출

```
import numpy as np

f = np.arange(0,12).reshape(3,4)
print(f)

print(f[f>3])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[ 4  5  6  7  8  9 10 11]
```

표현식 : 산출 + 비교 연산

배열의 각 원소가 3으로 나누었을때 나머지가0이 아닌 경우 원소 추출

```
import numpy as np

f = np.arange(0,12).reshape(3,4)
print(f)

print(f[f%3 > 0])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[ 1  2  4  5  7  8 10 11]
```

메소드 사용

배열 내의 원소가 nonzero인 것을 식별하기 위해 nonzero메소드 사용

```
import numpy as np

f = np.arange(0,12).reshape(3,4)
print(f)

print(f[f.nonzero()])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[ 1  2  3  4  5  6  7  8  9 10 11]
```

하나의 필드명 정의

dtype 함수를 이용해서 필드명과 데이터 타입을 정의해서 직접 접근해서 출력하기

```
import numpy as np
# Structured type, one field name 'f1', containing int32
dt = np.dtype([('density', np.int32)])
print(dt)
x = np.array([(393,), (337,), (256,)],
             dtype=dt)

# fieldname으로 조회
print(x['density'])
print(x)
print("\nThe internal representation:")
print(repr(x))
```

```
[('density', '<i4')]
[393 337 256]
[(393,) (337,) (256,)]

The internal representation:
array([(393,), (337,), (256,)],
      dtype=[('density', '<i4')])
```

여러 필드명 정의 1

dtype 함수를 이용해서 여러 필드명과 데이터 타입을 정의해서 생성 및 출력하기

```
dt = np.dtype([('country', 'S20'), ('density', 'i4'), ('area', 'i4'), ('population', 'i4')])
x = np.array([('Netherlands', 393, 41526, 16928800),
('Belgium', 337, 30510, 11007020),
('United Kingdom', 256, 243610, 62262000),
('Germany', 233, 357021, 81799600),
('Liechtenstein', 205, 160, 32842),
('Italy', 192, 301230, 59715625),
('Switzerland', 177, 41290, 7301994),
('Luxembourg', 173, 2586, 512000),
('France', 111, 547030, 63601002),
('Austria', 97, 83858, 8169929),
('Greece', 81, 131940, 11606813),
('Ireland', 65, 70280, 4581269),
('Sweden', 20, 449964, 9515744),
('Finland', 16, 338424, 5410233),
('Norway', 13, 385252, 5033675)],
             dtype=dt)
print(x[:4])
```

```
[(b'Netherlands', 393,  41526, 16928800)
 (b'Belgium', 337,  30510, 11007020)
 (b'United Kingdom', 256, 243610, 62262000)
 (b'Germany', 233, 357021, 81799600)]
```

필드명: 칼럼 조회

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

dtype 함수를 이용해서 여러 필드명과 데이터 타입을 정의해서 생성 및 출력하기

```
print(x['density'])
print(x['country'])
print(x['area'][2:5])
```

```
[393 337 256 233 205 192 177 173 111  97  81  65  20  16  13]
[b'Netherlands' b'Belgium' b'United Kingdom' b'Germany' b'Liechtenstein'
 b'Italy' b'Switzerland' b'Luxembourg' b'France' b'Austria' b'Greece'
 b'Ireland' b'Sweden' b'Finland' b'Norway']
[243610 357021    160]
```

값 갱신

값을 row 단위 및 원소 하나를 갱신

```
import numpy as np

time_type = np.dtype( [('hour', int), ('min', int), ('sec', int)])
times = np.array([(11, 38, 5),
                  (14, 56, 0),
                  (3, 9, 1)], dtype=time_type)
print(times)
print(times[0])
# reset the first time record:
times[0] = (11, 42, 17)
print(times[0])

times['hour'][0] = 23
print(times[0])
```

```
[(11, 38, 5) (14, 56, 0) ( 3,  9, 1)]
(11, 38, 5)
(11, 42, 17)
(23, 42, 17)
```

다차원 데이터의 표현과 연산


```
import numpy as np

s = 10
v = [1,2,3]
m = np.array([[1,2,3],[4,5,6]])
print(s, type(s))
print(v, type(v))
print(m, type(m))
```

```
(10, <type 'int'>)
([1, 2, 3], <type 'list'>)
(array([[1, 2, 3],
        [4, 5, 6]]), <type 'numpy.ndarray'>)
```

1. 인공지능과 선형대수

2. NUMPY, NDARRAY

3. NUMPY ARRAY를 이용한 일반연산

4. MATRIX-VECTOR, MAT-MAT 연산

4부 MATRIX- VECTOR 연산

- 앞에서 본 것처럼 인공신경망 모델에 많이 사용됨
- 다음 세개의 함수 또는 method를 사용
  - np.matmul(a, b)
  - np.dot(a, b)
  - a.dot(b)
- 주의 a*b, 또는 np.multiply(a,b)는 element-wise 곱셈이지, matrix-matrix 곱셈이 아니다.

좀 더 부가 설명을 하자면 **ndarray를 이용하여 벡터(vector)를 표현할 때는 2차 배열로 정의해야 한다**. 즉, 다음 세 가지는 모
세 번째 같이 생성해야 한다. (혼동하기 참 쉽다.)

```
a1 = np.array( [1, 2, 3] ) #크기 (3,)인 1차원 배열
a2 = np.array( [ [1, 2, 3] ] ) #크기 (1,3)인 2차원 배열 (행벡터)
a3 = np.array( [ [1], [2], [3] ] ) #크기 (3,1)인 2차원 배열 (열벡터)
```

여기서 **a1.T 는 동작하지 않는다**. 반면 a2.T 와 a3.T는 동작한다. 1차 배열은 행벡터나 열벡터 두 가지 모두로 취급되기도 한다

ndarray 객체를 사용하는데 있어서 장점과 단점은 다음과 같다.

- 장점
  - 1차 배열은 행벡터나 열벡터 둘 다로 취급할 수 있다. dot(A,v) 에서 v는 열벡터로 다루어지고 dot(v,A)에서는 행
    잡하게 수행할 필요가 없다.
  - 요소 간 곱셈이 쉽다 (예: A*B) 사실 모든 산술 연산 (+ - * / ** 등등)이 요소 간 연산이다.
  - ndarray 는 numpy 의 가장 기본 객체이므로 연산의 속도, 효율성 그리고 numpy를 이용하는 외부 라이브러리의 빈
  - 다차원 배열을 쉽게 구현한다.
  - tensor algebra 에 대한 장점이 있다.(?)
- 단점
  - 행렬간 곱에 obj.dot() 멤버함수를 사용해야 하므로 번잡할 수 있다. 세 행렬 A, B, C의 행렬곱은 dot(dot(A,B),C) 이

1차원 NP.ARRAY() 연산

```
import numpy as np
a1 = np.array([1, 2, 3, 4])
print(a1, a1.T)
print(a1*a1, a1*a1.T)
print(a1.dot(a1), a1.dot(a1.T))
print(np.multiply(a1, a1))
```

```
[1 2 3 4] [1 2 3 4]
[ 1  4  9 16] [ 1  4  9 16]
30 30
[ 1  4  9 16]
```

1차원 array는 transpose 개념이 없다.

2차원 NDARRAY계산

- 순서는 우리가 보통 생각하는 것과 같음.

- np.matmul(a, b)
- np.dot(a, b)
- a.dot(b)

- a*b 는 element 끼리의 곱임.

```
In [28]:
```

```
a=np.array([[1, 0], [0, 2]])
b=np.array([[4, 1], [2,2]])
print(type(a))
print(np.matmul(a, b), "₩n")
print(np.dot(a, b), "₩n")
print(a.dot(b), "₩n")
print(b.dot(a), "₩n")
print(np.matmul(b,a), "₩n")
print(a*b)
```

```
<class 'numpy.ndarray'>
[[4 1]
 [4 4]]

[[4 1]
 [4 4]]

[[4 1]
 [4 4]]

[[4 2]
 [2 4]]

[[4 2]
 [2 4]]

[[4 0]
 [0 4]]
```

2차원 NDARRAY 연산 (중요)

```
import numpy as np

m1=np.array([[1, 2, 3, 4]])
print(type(m1))
print(m1)
print(m1.T)
print(m1*m1.T)  #element-wise mult
print(m1.dot(m1.T)) #(1,4)*(4,1)
print(m1.T.dot(m1))
print(np.multiply(m1, m1))
print(np.dot(m1, m1.T))
```

```
<class 'numpy.ndarray'>
[[1 2 3 4]]
[[1]
 [2]
 [3]
 [4]]
[[ 1  2  3  4]
 [ 2  4  6  8]
 [ 3  6  9 12]
 [ 4  8 12 16]]
[[30]]
[[ 1  2  3  4]
 [ 2  4  6  8]
 [ 3  6  9 12]
 [ 4  8 12 16]]
[[ 1  4  9 16]]
[[30]]
```

numpy.dot

numpy.**dot**(*a*, *b*, *out=None*)

Dot product of two arrays. Specifically,

- If both *a* and *b* are 1-D arrays, it is inner product of vectors (without complex conjugation).
- If both *a* and *b* are 2-D arrays, it is matrix multiplication, but using **matmul** (numpy.matmul.html#numpy.matmul) or `a @ b` is preferred.
- If either *a* or *b* is 0-D (scalar), it is equivalent to **multiply** (numpy.multiply.html#numpy.multiply) and using `numpy.multiply(a, b)` or `a * b` is preferred.
- If *a* is an N-D array and *b* is a 1-D array, it is a sum product over the last axis of *a* and *b*.
- If *a* is an N-D array and *b* is an M-D array (where `M>=2`), it is a sum product over the last axis of *a* and the second-to-last axis of *b*:

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

Parameters:

a : *array_like*

First argument.

b : *array_like*

Second argument.

out : *ndarray, optional*

Output argument. This must have the exact kind that would be

numpy.matmul

numpy.**matmul**(*x1, x2, /, out=None, *, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj]*) = *<ufunc 'matmul'>*

Matrix product of two arrays.

Parameters:

x1, x2 : *array_like*

Input arrays, scalars not allowed.

out : *ndarray, optional*

A location into which the result is stored. If provided, it must have a shape that matches the signature *(n,k),(k,m)->(n,m)*. If not provided or None, a freshly-allocated array is returned.

For 2-D arrays it is the matrix product:

```
>>> a = np.array([[1, 0],
...               [0, 1]])
>>> b = np.array([[4, 1],
...               [2, 2]])
>>> np.matmul(a, b)
array([[4, 1],
       [2, 2]])
```

```
>>> a = np.array([[1, 0],
...               [0, 1]])
>>> b = np.array([1, 2])
>>> np.matmul(a, b)
array([1, 2])
>>> np.matmul(b, a)
array([1, 2])
```

```python
import numpy as np
m1=np.array([[1, 2, 3, 4]])
print(type(m1), "₩n")
print(m1, m1.shape)
print(m1.T, "₩n", m1.T.shape, "₩n")
print("m1*m1T=", m1*m1.T, (m1*m1.T).shape,"₩n")
print("m1dotmiT=", m1.dot(m1.T), (m1.dot(m1.T)).shape, "₩n")
print("M1T dot m1 =", m1.T.dot(m1), "₩n")
print(np.multiply(m1, m1), "₩n")
print(np.dot(m1, m1.T))
```

```
<class 'numpy.ndarray'>

[[1 2 3 4]] (1, 4)
[[1]
 [2]
 [3]
 [4]]
 (4, 1)

m1*m1T= [[ 1  2  3  4]
 [ 2  4  6  8]
 [ 3  6  9 12]
 [ 4  8 12 16]] (4, 4)

m1dotmiT= [[30]] (1, 1)

M1T dot m1 = [[ 1  2  3  4]
 [ 2  4  6  8]
 [ 3  6  9 12]
 [ 4  8 12 16]]

[[ 1  4  9 16]]

[[30]]
```

```python
import numpy as np

A = np.array([[3, 6, 7], [5, -3, 0]])
B = np.array([[1, 1], [2, 1], [3, -3]])
C = a.dot(B)
print(C)

'''
Output:
[[ 36 -12]
 [ -1    2]]
'''
```

```
a=np.array([[1, 0], [0, 1], [3,4]])
b=np.array([[4, 1], [2,2]])
print(type(a))
print(np.matmul(a, b), "₩n")
print(np.dot(a, b), "₩n")
print(a.dot(b), "₩n")
```

```
<class 'numpy.ndarray'>
[[ 4  1]
 [ 2  2]
 [20 11]]

[[ 4  1]
 [ 2  2]
 [20 11]]

[[ 4  1]
 [ 2  2]
 [20 11]]
```

3-DIMENSIONAL ARRAY

```
In [54]:
d1 = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23])
d2 = d1.reshape((2, 3, 4))
e1 = np.array([0, 1, 2, 3, 4, 5, 6, 7]).reshape(4, 2)
print(d2, "₩n", "₩n", e1, "₩n")
print(np.dot(d2, e1), "₩n")
print(d2.dot(e1), "₩n")
print(np.multiply(d2, e1))
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]

 [[0 1]
 [2 3]
 [4 5]
 [6 7]]

[[[ 28  34]
  [ 76  98]
  [124 162]]

 [[172 226]
  [220 290]
  [268 354]]]
```

```
[[[ 28  34]
  [ 76  98]
  [124 162]]

 [[172 226]
  [220 290]
  [268 354]]]
```

```
---------------------------------------
-
ValueError
<ipython-input-54-549aadf275c4> in <
      5 print(np.dot(d2, e1), "₩n")
      6 print(d2.dot(e1), "₩n")
----> 7 print(np.multiply(d2, e1))

ValueError: operands could not be broac
```

MATRIX VECTOR MULTIPLICATION
C = A*B

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

In [39]:

```
aa=np.array([[0, 1], [3, 4]])
bb = np.array([1, 3])
cc = np.array([[1],[3]])
d = np.dot(aa,bb)
print(d, d.shape)
e = np.dot(aa,cc)
print(e, e.shape)
f = np.dot(bb, aa.T)
print(f)
print(bb.dot(aa.T))
print(np.matmul(aa, cc))
print(np.matmul(aa, bb))
```

```
[ 3 15] (2,)
[[ 3]
 [15]] (2, 1)
[ 3 15]
[ 3 15]
[[ 3]
 [15]]
[ 3 15]
```

NP.MATRIX() – MATLAB USER를 위한 2차원 NDARRAY

```
import numpy as np
m1=np.matrix([1, 2, 3, 4])
print(m1)
print(m1.T)
print(m1*m1.T)
print(m1.dot(m1.T))
print(m1.T.dot(m1))
print(np.multiply(m1, m1))
```

```
[[1 2 3 4]]
[[1]
 [2]
 [3]
 [4]]
[[30]]
[[30]]
[[ 1  2  3  4]
 [ 2  4  6  8]
 [ 3  6  9 12]
 [ 4  8 12 16]]
[[ 1  4  9 16]]
```

```
import numpy as np
m1=np.matrix([1, 2, 3, 4])
print(type(m1))
```

```
<class 'numpy.matrix'>
```

Matrix는 2-dim. 1차원 array를 주면 2차원으로 상향 조정됨. .

NDARRAY와 MATRIX (MATLAB사용자 대상) 차이

- 연산자 *, dot() 그리고 multiply()
- ndarray 는 '*'는 요소간 곱셈이다. 행렬곱을 할때는 obj.dot() 메쏘드를 사용해야 한다
- 반면 matrix는 '*'이 행렬곱이다. 그리고 numpy.multiply() 함수가 요소간 곱이다.
- 벡터와 1차 배열
- ndarray 는 벡터 1xN, Nx1, 그리고 N크기의 1차원 배열이 모두 각각 다르다. obj[:,1] 는 크기가 n인 1차 배열을 반환한다. 그리고 1차원 배열의 전치는 작동하지 않는다. (역자 주 : ndarray로 벡터를 표현할 때는 반드시 2차 배열을 이용해야 한다.)
- 반면 matrix 객체에서 1차 배열은 모두 1xn, 혹은 nx1 행렬(2차원 배열)로 상향 변환된다. (matrix 객체는 내부적으로 항상 2차원 배열이다.)
- ndarray는 고차원 배열이 가능하지만 matrix는 항상 2차원 배열 이다.

numpy.matrix

class numpy.**matrix**(*data*, *dtype=None*, *copy=True*) [source]
(https://github.com/numpy/numpy/blob/v1.18.4/numpy/__init__.py)

> Note:
> 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.

Returns a matrix from an array-like object, or from a string of data. A matrix is a specialized 2-D array that retains its 2-D nature through operations. It has certain special operators, such as `*` (matrix multiplication) and `**` (matrix power).

Parameters:

data : *array_like or string*

If **data** (numpy.matrix.data.html#numpy.matrix.data) is a string, it is interpreted as a matrix with commas or spaces separating columns, and semicolons separating rows.

dtype : *data-type*

Data-type of the output matrix.

copy : *bool*

If **data** (numpy.matrix.data.html#numpy.matrix.data) is already an **ndarray** (numpy.ndarray.html#numpy.ndarray), then this flag determines whether the data is copied (the default), or whether a view is constructed.

SINGLETON ARRAY 값을 SCALAR로

```
import numpy as np
matrix = np.array([[1]])
s = np.squeeze(matrix)
print(type(s))
print(s)
matrix = [[1]]
print(type(s))
print(s)
s = 1
print(type(s))
print(s)
--------------
<class 'numpy.ndarray'>
1
<class 'numpy.ndarray'>
1
<class 'int'>
1
```

```
import numpy as np

matrix = np.array([[[[7]]]])
print(matrix.item())
```

NDARRAY와 MATRIX 차이

- ndarray 를 사용하는 것이 더 효율적이라고 명시되어 있다. 개인적으로 matrix 객체는 MATLAB 사용자의 편의를 위해서 (억지로) 만들어진 것
- ndarray를 사용해야 하는 이유는 다음과 같이 요약할 수 있다.
- ndarray는 numpy에서 지원하는 표준형인 벡터/행렬/텐서 를 저장한다.
- 많은 numpy 함수가 matrix가 아니라 ndarray를 반환한다.
- 요소 간 연산과 선형대수 연산에 대해선 명확히 구분되어 있다.
- 표준 벡터나 열벡터/행벡터를 표현할 수 있다.
- ndarray를 사용할 경우의 한 가지 단점은 행렬의 곱셈을 수행할 때 ,dot() method를 사용해야한다는 점이다. 즉, ndarray 객체 A와 B를 행렬곱하려면 A.dot(B) 와 같이 수행해야 하며 A*B는 요소간 곱셈이 된다. 반면 matrix객체 A와 B는 단순히 A*B로 행렬곱이 수행된다.
-

출처: https://studymake.tistory.com/408 [스터디메이크]

NUMPY EINSUM (EINSTEIN SUMMATION)

- Matrix의 행, 열, .. 번호 등을 ijk 등 의 index로 나타내고, 이를 이용하여, 입력과 출력의 각 element 의 값을 나타낸다. 이 때 생략되는 index 에 대해서는 그 index가 변하면서 모두 곱하기한 후 더한다 (dot product). 출력을 표시하는 기호는 ->
- Result = einsum("dimension notation of A, dimension notation of B,...->Result Dimension", A, B, ...)
- Dot product of x, y vectors

```
dot = np.einsum('i,i->', x, y )
```

- Matrix-vector multiplication

$$b_i = \sum_j A_{ij} b_j$$

- b = np.einsum('ij,j->i', A, x) (입출력에 대해서 없어지는 인덱스에 대해서 dot product)

https://ita9naiwa.github.io/numeric%20calculation/2018/11/10/Einsum.html

NP.EINSUM (2/2)

- Transpose: R = np.einsum("ij->ji", A)
- diag = np.einsum('ii->i', A)
- trace =np.einsum('ii->', A)
- Matrix sum to scalar.
  R = np.einsum("ij->", A)
- row_sum = np.einsum("ij->i", A)
- col_sum =np.einsum("ij->j", A)
- dot = np.einsum('i,i->', x, y)
- outer = np.einsum('i,j->ij', x,y)

```
a = np.array([[1,2,3], [4,5,6]])
r = np.einsum("ij->", a)
print(r)
----------
21
```

Quadritc Form, or Matrix norm, or Distance with respect to Matrix(Mahalanobis distance)

$$r = x^T A x$$
$$r = a^T A b$$

- r = np.einsum('i,ij,j->', x, A, y)

- Matrix matrix multiplication:
  R = np.einsum('ik,kj->ij', A, b)

- Batch matrix multiplication:
  R = np.einsum('bik,bkj->bij',A, B)

```
import numpy as np
## Matrix-Matrix Multiplication
a = np.array([[1,2,3], [4,5,6]])
b = a.transpose()
r = np.einsum('ik,kj->ij', a, b)
print(r)
-----
[[14 32]
 [32 77]]
```

NUMPY LINEAR ALGEBRA (NP.LINALG)

Matrix and vector products

dot(a, b[, out]) : Dot product of two arrays.

linalg.multi_dot(arrays) : Compute the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order.

vdot(a, b) : Return the dot product of two vectors.

inner(a, b) : Inner product of two arrays.

outer(a, b[, out]) : Compute the outer product of two vectors.

matmul(x1, x2, /[, out, casting, order, …]) : Matrix product of two arrays.

tensordot(a, b[, axes]) : Compute tensor dot product along specified axes.

VECTOR 크기 계산

- 벡터의 크기(Magnitude)는 원소들의 제곱을 더 하고 이에 대한 제곱근의 값
- 벡터의 크기는 x축의 변위와 y축의 변위를 이용 하여 피타고라스 정리

```
import math
import numpy as np

x = np.array([1,2])
mag = lambda x: math.sqrt(sum(i**2 for i in x))
print(mag(x))

print(np.linalg.norm(x))
```

```
2.2360679775
2.2360679775
```

단위벡터 정규화

해당 벡터를 0 ~ 1의 값으로 정규화

```
import math
import numpy as np

def add(u, v):
    return [ u[i]+v[i] for i in range(len(u)) ]

def magnitude(v):
    return math.sqrt(sum(v[i]*v[i] for i in range(len(v))))

def normalize(v):
    vmag = magnitude(v)
    return [ v[i]/vmag  for i in range(len(v)) ]

l = [1, 1, 1]
v = [0, 0, 0]
h = normalize(add(l, v))
print(magnitude(add(l,v)))
print h
```

```
1.73205080757
[0.5773502691896258, 0.5773502691896258, 0.5773502691896258]
```

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{|\mathbf{v}|}$$

Decompositions

linalg.cholesky(a) : Cholesky decomposition.

linalg.qr(a[, mode]) : Compute the qr factorization of a matrix.

linalg.svd(a[, full_matrices, compute_uv, …]) : Singular Value Decomposition.

Matrix eigenvalues

linalg.eig(a) : Compute the eigenvalues and right eigenvectors of a square array.

linalg.eigh(a[, UPLO]) : Return the eigenvalues and eigenvectors of a complex Hermitian (conjugate symmetric) or a real symmetric matrix.

linalg.eigvals(a) : Compute the eigenvalues of a general matrix.

linalg.det(a) : Compute the determinant of an array.

linalg.matrix_rank(M[, tol, hermitian]) : Return matrix rank of array using SVD method

trace(a[, offset, axis1, axis2, dtype, out]) : Return the sum along diagonals of the array.

Solving equations and inverting matrices

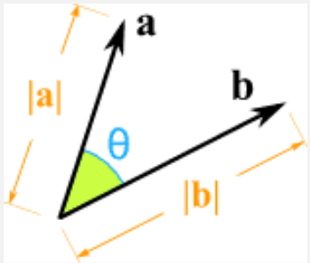
linalg.solve(a, b) : Solve a linear matrix equation, or system of linear scalar equations.

내적 vs 외적

구분	내적	외적
명칭	Inner product, dot product, scalar product	Outer product, vector product, cross product
표기	.(Dot)	X(cross)
대상 벡터	n 차원	3 차원
공식	$a_1 b_1 + a_2 b_2 + ... + a_n b_n$	$(a_2 b_3 - a_3 b_2, a_3 b_1 - a_1 b_3, a_1 b_2 - a_2 b_1)$
	|a||b| cos 각도	|a||b| sin각도 n
결과	scalar	vector

내적 산식

내적(Inner Product)산식은 두벡터의 크기에 cos각을 곱한 결과 또는 두벡터간의 원소들이 곱의 합산과 같은 결과



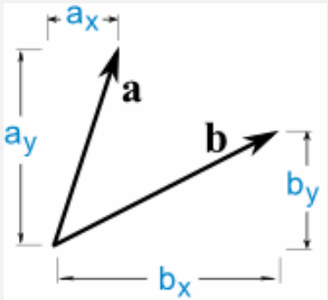
$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| \times |\mathbf{b}| \times \cos(\theta)$

Where:

$|\mathbf{a}|$: vector **a** 크기

$|\mathbf{b}|$: vector **b** 크기

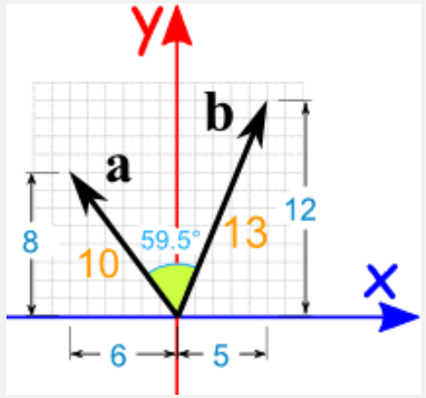
θ : **a** and **b** 사이의 각



$\mathbf{a} \cdot \mathbf{b} = a_x \times b_x + a_y \times b_y$

내적 수학적 예시 : 2차원

두벡터에 내적 연산에 대한 수학적 처리예시



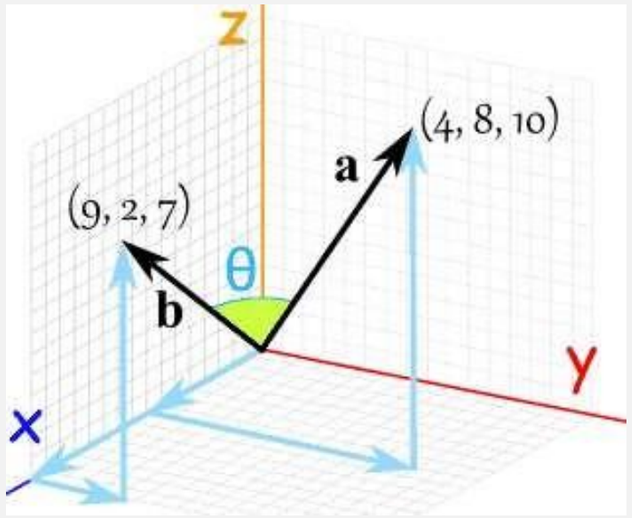
$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| \times |\mathbf{b}| \times \cos(\theta)$
$\mathbf{a} \cdot \mathbf{b} = 10 \times 13 \times \cos(59.5°)$
$\mathbf{a} \cdot \mathbf{b} = 10 \times 13 \times 0.5075...$
$\mathbf{a} \cdot \mathbf{b} = 65.98... = 66$ (rounded)

$\mathbf{a} \cdot \mathbf{b} = a_x \times b_x + a_y \times b_y$
$\mathbf{a} \cdot \mathbf{b} = -6 \times 5 + 8 \times 12$
$\mathbf{a} \cdot \mathbf{b} = -30 + 96$
$\mathbf{a} \cdot \mathbf{b} = 66$

3차원 내적 예시 1

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

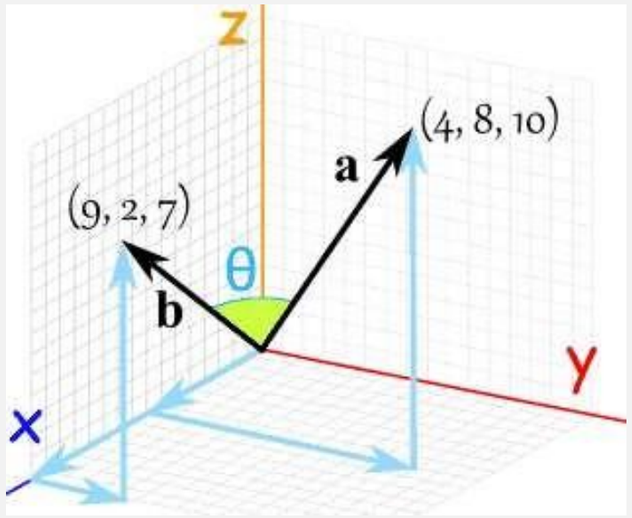
Dot 연산을 통한 계산



$$\mathbf{a} \cdot \mathbf{b} = a_x \times b_x + a_y \times b_y + a_z \times b_z$$
$$\mathbf{a} \cdot \mathbf{b} = 9 \times 4 + 2 \times 8 + 7 \times 10$$
$$\mathbf{a} \cdot \mathbf{b} = 36 + 16 + 70$$
$$\mathbf{a} \cdot \mathbf{b} = 122$$

3차원 내적 예시 2

두벡터 사이의 각 구하기



a벡터의 크기
$|\mathbf{a}| = \sqrt{(4^2 + 8^2 + 10^2)}$
$= \sqrt{(16 + 64 + 100)}$
$= \sqrt{180}$

b벡터의 크기
$|\mathbf{b}| = \sqrt{(9^2 + 2^2 + 7^2)}$
$= \sqrt{(81 + 4 + 49)}$
$= \sqrt{134}$

내적 구하기
$\mathbf{a} \cdot \mathbf{b} = 9*4 + 2*8 + 7*10 = 36+16+70 = 122$

각 구하기
$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| \times |\mathbf{b}| \times \cos(\theta)$ 산식에 대입

$122 = \sqrt{180} \times \sqrt{134} \times \cos(\theta)$
$\cos(\theta) = 122 / (\sqrt{180} \times \sqrt{134})$
$\cos(\theta) = 0.7855...$
$\theta = \cos^{-1}(0.7855...) = 38.2...°$

내적(DOT) 예시

- 두벡터에 대한 내적(dot) 연산은 같은 위치의원 소를 곱해서 합산함
- 두벡터의 곱셈은 단순히 원소를 곱해서벡터를 유지

```
import math
import numpy as np

d = np.array([4,5])
e = np.array([3,8])
j= d*e
print(j)
print(np.dot(d,e))
```

```
[12 40]
52
```

VDOT: VECTOR

벡터(2차원)일 경우도 스칼라(dot)로 처리

```
import numpy as np
import numpy.linalg as lin

va = np.array([[1, 0]])
vb = np.array([[4, 1]])
vc = np.array([[4],[1]])
print(np.vdot(va,vb))
print(np.vdot(va,vc))
```

```
4
4
```

WE USE NUMPY.LINALG.INV() FUNCTION TO CALCULATE THE INVERSE OF A MATRIX. THE INVERSE OF A MATRIX IS SUCH THAT IF IT IS MULTIPLIED BY THE ORIGINAL MATRIX, IT RESULTS IN IDENTITY MATRIX

```
x = np.array([[1,2],[3,4]])
y = np.linalg.inv(x)
print x
print y
print np.dot(x,y)
```

```
[[1 2]
 [3 4]]
[[-2.   1. ]
 [ 1.5 -0.5]]
[[  1.00000000e+00   1.11022302e-16]
 [  0.00000000e+00   1.00000000e+00]]
```

NUMPY PSEUDO-INVERSE

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

numpy.linalg.pinv

numpy.linalg.pinv(*a*, *rcond=1e-15*, *hermitian=False*)[source]

Compute the (Moore-Penrose) pseudo-inverse of a matrix.
Calculate the generalized inverse of a matrix using its singular-value decomposition (SVD) and including all *large* singular values.

The pseudo-inverse of a matrix A, denoted A^+, is defined as: “the matrix that ‘solves’ [the least-squares problem] Ax = b,” i.e., if \bar{x} is said solution, then A^+ is that matrix such that \bar{x} = A^+b.

It can be shown that if Q_1 \Sigma Q_2^T = A is the singular value decomposition of A, then A^+ = Q_2 \Sigma^+ Q_1^T, where Q_{1,2} are orthogonal matrices, \Sigma is a diagonal matrix consisting of A’s so-called singular values, (followed, typically, by zeros), and then \Sigma^+ is simply the diagonal matrix consisting of the reciprocals of A’s singular values (again, followed by zeros). [1]

numpy.linalg.svd

numpy.linalg.svd(*a*, *full_matrices=True*, *compute_uv=True*, *hermitian=False*)

Singular Value Decomposition.

When *a* is a 2D array, `u @ np.diag(s) @ vh = (u * s) @ vh`,

where *u* and *vh* are 2D unitary arrays and *s* is a 1D array of *a*'s singular values.

When *a* is higher-dimensional, SVD is applied in stacked mode as explained below.

```
a = np.random.randn(9, 6) + 1j*np.random.randn(9, 6)
```

Reconstruction based on reduced SVD, 2D case:

```
>>>
u, s, vh = np.linalg.svd(a, full_matrices=False)
u.shape, s.shape, vh.shape
((9, 6), (6,), (6, 6))
np.allclose(a, np.dot(u * s, vh))
True
smat = np.diag(s)
np.allclose(a, np.dot(u, np.dot(smat, vh)))
True
```

```
a = np.random.randn(9, 6)
B = np.linalg.pinv(a)
np.allclose(a, np.dot(a, np.dot(B, a)))
True
np.allclose(B, np.dot(B, np.dot(a, B)))
True
```

random.radn(m, n) --- 평균0, 표준편차 1인 Gaussian mxn array로

NUMPY FUNCTIONS

- Data I/O
  - `fromfile, genfromtxt, load, loadtxt, save, savetxt`
- Mesh Creation
  - `mgrid, meshgrid, ogrid`
- Manipulation
  - `einsum, hstack, take, vstack`

ARRAY SUBCLASSES

NUMPY.MA — MASKED ARRAYS

NUMPY.MATRIX — MATRIX OPERATORS

NUMPY.MEMMAP — MEMORY-MAPPED ARRAYS

NUMPY.RECARRAY — RECORD ARRAYS

OTHER SUBPACKAGES

- `numpy.fft` — Fast Fourier transforms
- `numpy.polynomial` — Efficient polynomials
- `numpy.linalg` — Linear algebra
  - `cholesky, det, eig, eigvals, inv, lstsq, norm, qr, svd`
- `numpy.math` — C standard library math functions
- `numpy.random` — Random number generation
  - `beta, gamma, geometric, hypergeometric, lognormal, normal, poisson, uniform, weibull`

NUMPY RANDOM MODULE

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

numpy package에는 random이라는 module이 있다.여기에 있는 random 함수를 쓰면 0에서 1 사

```
1 import numpy as np
2
3 numpy.random.random()
```

만약 여러개의 값을 한꺼번에 반환받고 싶으면, random 함수의 인자로 자기가 받고 싶은 자료의 크기를
들어 1x5의 random 함수의 결과를 받고 싶으면

```
1 numpy.random.random(5)
```

를 해주면 되고, 만약 2행 2열의 행렬의 형태로 받고 싶으면

```
1 numpy.random.random((2,2))
```

```
1 int_min = 0
2 int_max = 10
3
4 np.random.randint(int_min, int_max, (5,3))
```

첫번째 인자로 0, 두번째 인자로 1을 넣어주면 된다.

```
1 mean = 0
2 std = 1
3 np.random.normal(mean, std)
```

물론 이것도 앞에서 소개한 random과 같이 동시에 여러 개의 값을 arra
어주면 된다. 예를 들어 5x3 의 행렬로 반환받고 싶은 경우,

```
1 mean = 0
2 std = 1
3 np.random.normal(mean, std)
4
5 # 5x3
6 np.random.normal(mean, std, (5,3))
```

https://talkingaboutme.tistory.com/entry/Python-Numpy-Random-Module

FFT

```
import numpy as np
t = np.linspace(0,120,4000)
PI = np.pi
signal = 12*np.sin(3 * 2*PI*t)          # 3 Hz
signal += 6*np.sin(8 * 2*PI*t)          # 8 Hz
signal += 1.5*np.random.random(len(t)) # noise
FFT = abs(np.fft.fft(signal))
freqs = np.fft.fftfreq(signal.size, t[1]-t[0])
```

TAKING SYMBOLIC DERIVATIVES
X**5 -> 5*X**4

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

- pip install sympy

```
import sympy as sym
```

```
x = sp.Symbol('x')
```

```
sym.diff(x**5)
```

$$5x^4$$

```
sym.diff((x**2 + 1) * sym.cos(x))
```

$$2x\cos(x) - \left(x^2 + 1\right)\sin(x)$$

$$f(x, y) = x^2 y$$
$$\partial_x f(x^2 y) = 2xy$$
$$\partial_y f(x^2 y) = x^2$$

would I do this in Python? Good question. To start, you'll nee
ymbols. And in a traditional Python style, you can do this with

```
x, y = sym.symbols('x y')
```

sym.diff(x**2*y, x)
Sym.diff(x **2*y, y)

RESOURCES

- http://docs.scipy.org/doc/numpy/reference/
- http://docs.scipy.org/doc/numpy/user/index.html
- http://www.scipy.org/Tentative_NumPy_Tutorial
- http://www.scipy.org/Numpy_Example_List

These slides are currently available at https://github.com/ContinuumI

O/tutorials/blob/master/Intro*to*NumPy.pdf

Many thanks to

- Ben Zaitlin
- Stéfan van der Walt
- Amy Troschinetz
- Maggie Mari
- Travis Oliphant

Questions?

NDARRAY와 MATRIX 연산 비교

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

Matrix는 dot/* 처리가 동일, ndarry는 */multiply가 동일

```
import numpy as np

m1 = np.matrix([1,2,3,4])

print(m1*m1.T)
print(m1.dot(m1.T))
print(np.multiply(m1,m1))

a1 = np.array([1,2,3,4])

print(a1*a1.T)
print(a1.dot(a1.T))
print(np.multiply(a1,a1))
```

```
[[30]]
[[30]]
[[ 1  4  9 16]]
[ 1  4  9 16]
30
[ 1  4  9 16]
```

다차원 배열 : 열 조회/변경

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

7*4배열을 정의하고 첫번째 열의 값을 99으로 변경

배열명[행접근, 열접근]

Slicing도 행접근과 열접근으로 별도로 할 수 있음

배열명[행 슬라이싱, 열 슬라이싱]으로 배열을 접근 가능

```
import numpy as np
data1 = np.random.randn(7,4)
print(data1)
print("# second column ")
print(data1[:,1])

print("# updateing second column ")
data1[:,1] = 99
print(data1)
```

```
[[ 0.19447951 -1.41621328 -1.09835616 -0.12824362]
 [-1.28675483 -0.56357893  0.30512662  0.27225955]
 [-0.14058495  1.16316171 -1.11462148  0.15492861]
 [-0.4136672   0.19778725  1.77270056 -1.14325863]
 [-0.18705767  0.51032341 -2.28065836 -0.63879405]
 [-0.59003418 -0.15380713  0.66245183 -0.11166433]
 [ 1.66332656 -0.5733906   1.64218242 -0.13089124]]
# second column
[-1.41621328 -0.56357893  1.16316171  0.19778725  0.51032341 -0.15380713
 -0.5733906 ]
# updateing second column
[[  0.19447951  99.          -1.09835616  -0.12824362]
 [ -1.28675483  99.           0.30512662   0.27225955]
 [ -0.14058495  99.          -1.11462148   0.15492861]
 [ -0.4136672   99.           1.77270056  -1.14325863]
 [ -0.18705767  99.          -2.28065836  -0.63879405]
 [ -0.59003418  99.           0.66245183  -0.11166433]
 [  1.66332656  99.           1.64218242  -0.13089124]]
```

ARRAY VIEWS

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

Simple assigments do not make copies of arrays (same semantics as Python). Slicing operations do not make copies either; they return views on the original array.

```
In [2]: a = np.arange(10)

In [3]: b = a[3:7]

In [4]: b
Out[4]: array([3, 4, 5, 6])

In [5]: b[:] = 0

In [6]: a
Out[6]: array([0, 1, 3, 0, 0, 0, 0, 7, 8, 9])

In [7]: b.flags.owndata
Out[7]: False
```

```
a11 = np.arange(10)
c11 = a11
b11 = a11[3:7]

b11[:] = 0
print(a11)
print(b11)
print(c11)
```

```
[0 1 2 0 0 0 0 7 8 9]
[0 0 0 0]
[0 1 2 0 0 0 0 7 8 9]
```

Array views contain a pointer to the original data, but may have different shape or stride values. Views always have `flags.owndata` equal to `False`.