PYTHON NUMPY 선형대수

Moon Yong Joon

NUMPY 이해

Moon Yong Joon

ndarray와 matrix 구분

ndarray와 matrix 구분

다차원 배열을 생성하지만 matrix는 MATLAB 기능을 지원

구분	ndarray	matrix
차원	다차원 가능	2 차원
* 연산자	요소간 곱	행렬곱
numpy.multiply()	요소간 곱	요소간 곱
numpy.dot()	행렬곱	행렬곱

배열과 vector 구분 : ndarray

Array와 vector 구분 생성

```
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))
```

ndarray와 matrix 연산 비교

Matrix는 dot/* 처리가 동일, ndarry는 */multiply 가 동일

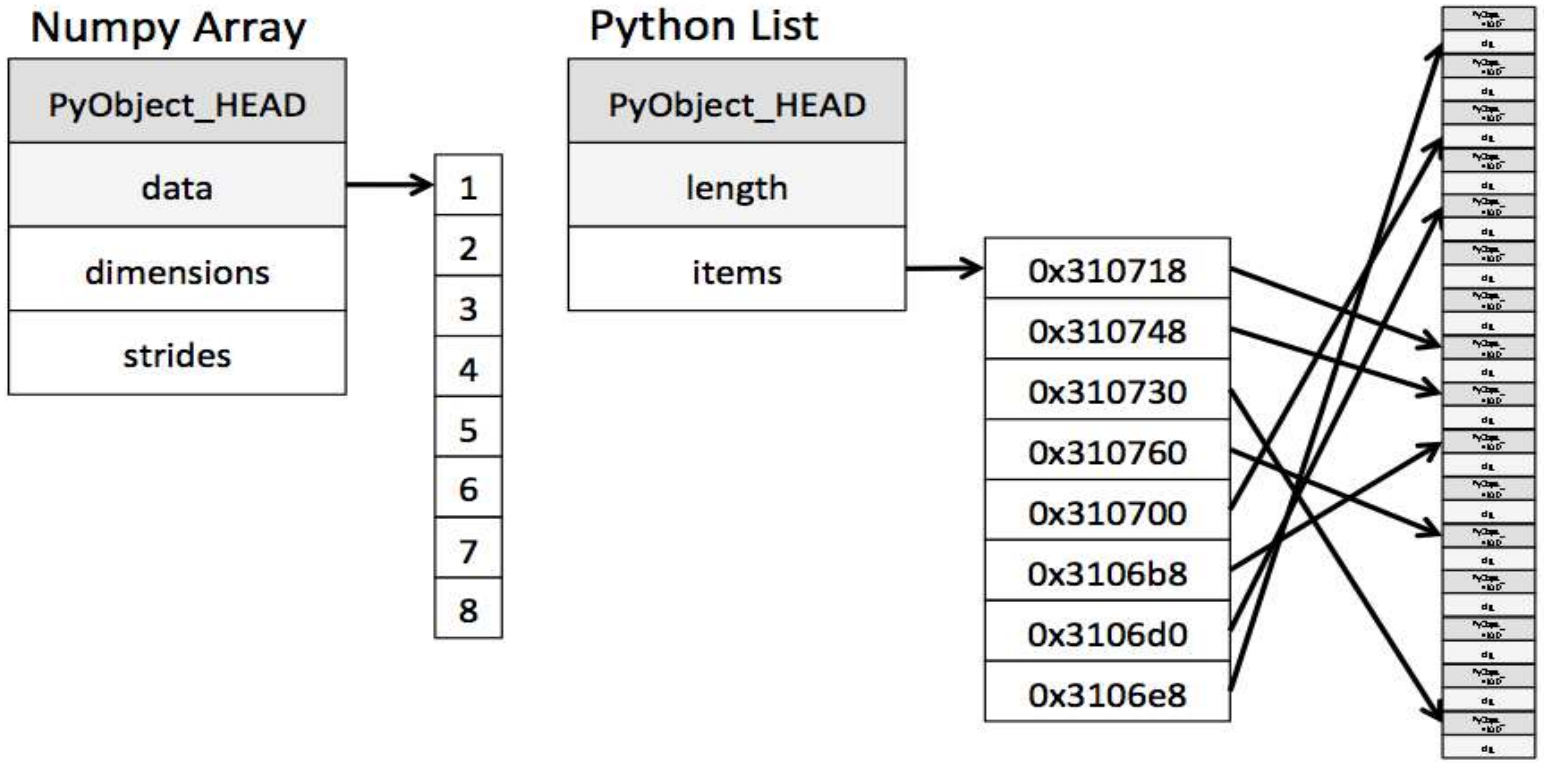
```
import numpy as np

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]
```

ndarray class

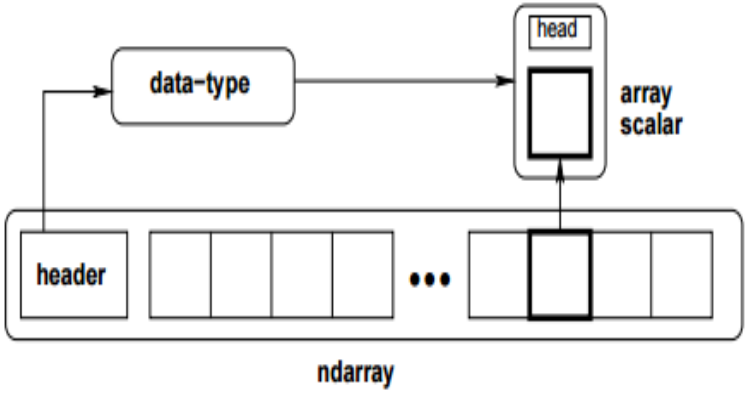
ndarray vs. list 구조

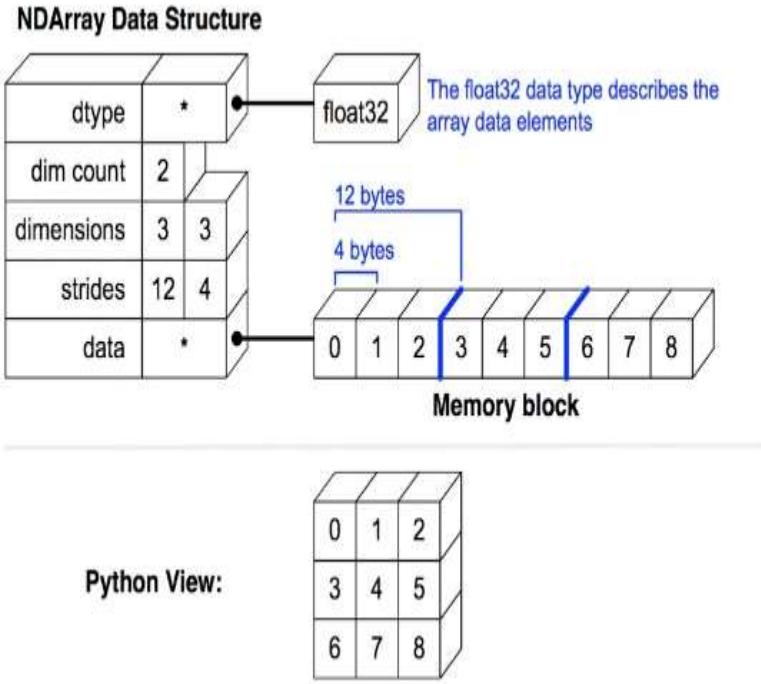
Ndarray와 list는 내부 구조부터 다르게 되어있어 ndarray가 더 처리가 빠르게 실행됨



ndarray 구조

Ndarray는 데이터를 관리하고 data-type은 실제 데이터들의 값을 관리하며, array scalar는 위치를 관리





데이터 타입 부여

ndarray는 각 원소별로 동일한 데이터 타입으로 처리

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']
```

0차원

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

Column: 열

Row : 행

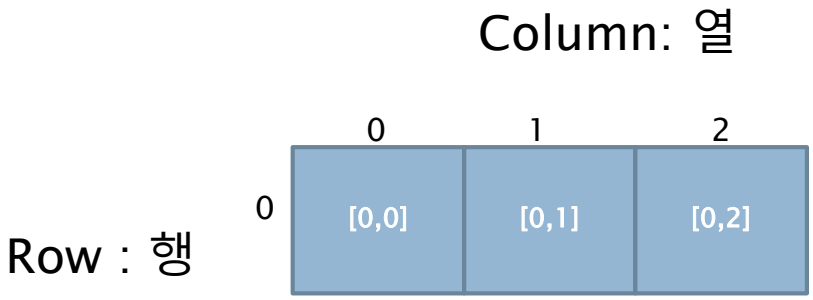

```
import numpy as np
a = np.array(10)
print(a)
print(a.ndim)
```

```
10
0
```

1차원

배열의 특징. 차원, 형태, 요소를 가지고 있음

생성시 데이터와 타입을 넣으면 ndim(차원)으로 확인




```
import numpy as np

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

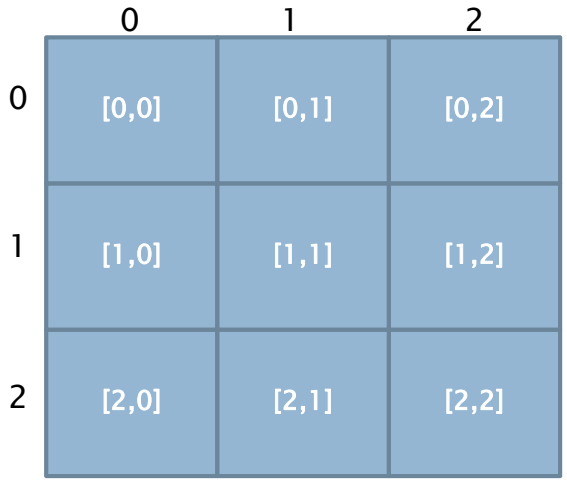
```
[1 2 3 4]
1
```

2차원 배열

3행, 3열의 배열을 기준으로 어떻게 내부를 행과 열로 처리하는 지를 이해

Column: 열

Row : 행



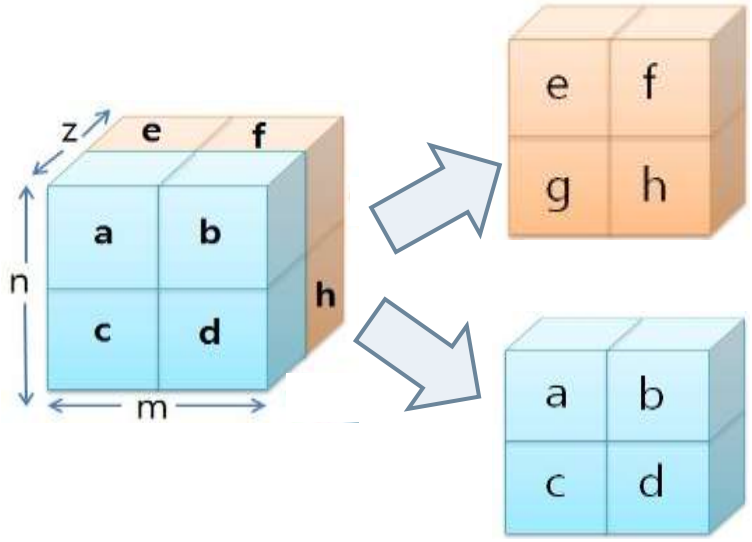

```
import numpy as np
a = np.array([[1,2],[3,4]])
print(a)
print(a.ndim)
```

```
[[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
```

할당은 참조만 전달

Ndarray 타입을 검색이나 슬라이싱은 참조만 할당하므로 변경을 반지하기 위해서는 새로운 ndarray로 만들어 사용. copy 메소드가 필요

```
import numpy as np

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

```
2
[99  2  3  4]
[99  2]
[1 2]
```

벡터화 연산 : for문 미사용

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

브로드캐스팅 처리

원소들끼리 계산을 하기 위해 동일한 모양으로 만들고 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]]
```

list와 ndarray 계산 성능

numpy.ndarray로 계산시 python list 타입에 비해 계산 속도가 빠름

```
import numpy as np
import time
size_of_vec = 10000000

def pure_python_version():
    t1 = time.time()
    X = range(size_of_vec)
    Y = range(size_of_vec)
    Z = []
    for i in range(len(X)):
        Z.append(X[i] + Y[i])
    print t1
    return time.time() - t1
def numpy_version():
    t2 = time.time()
    X = np.arange(size_of_vec)
    Y = np.arange(size_of_vec)
    Z = X + Y
    print t2
    return time.time() - t2

t1 = pure_python_version()
t2 = numpy_version()
print(t1, t2)
print("numpy is in this example " + str(t1/t2) + " faster!")
```

배열을 c언어처럼 관리하므로 별도의 index를 구성하지 않으므로 계산 속도 빠름

```
1470352261.39
1470352300.48
(27.127634048461914, 0.5557310581207275)
numpy is in this example 48.814320618 faster!
```

Ndarray 속성

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	ndarray 객체에 데이터는 itemsize 크기의 hex값으로 표현

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 할당할 경우 그 원천에 대한 것을 가지고 있음
ndarray.flat	ndarray 객체가 차원을 가질 경우 하나로 연계해서 index로 처리
ndarray.T	ndarray 객체에 대한 역핼력

내부 원소 접근 방식

__getitem__ 비교

numpy.ndarray 타입에서는 __getitem__에 논리 연산 등 다양한 처리를 허용

```
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.]
```

__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.]
```

2차원 : 행과 열 접근

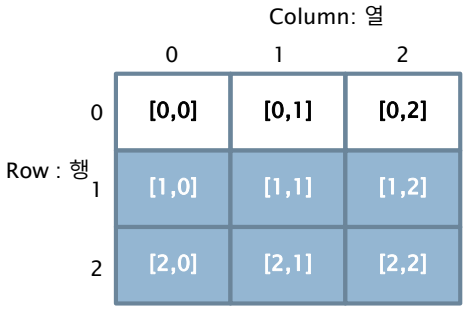
배열 접근하기 : 행과 열구분

배열명[행 범위, 열 범위] 행으로 접근, 열로 접근

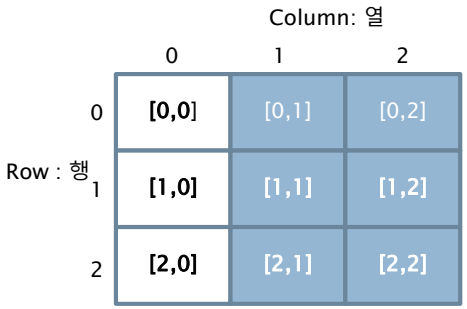
```
import numpy as np
l33 = [[1,2,3],[4,5,6],[7,8,9]]
np33 = np.array(l33,dtype=int)

print(np33.shape)
print(np33.ndim)
print np33

print(" first row",np33[0])

print("first column ", np33[:,0])
```

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

첫번째 행 접근



첫번째 열 접근

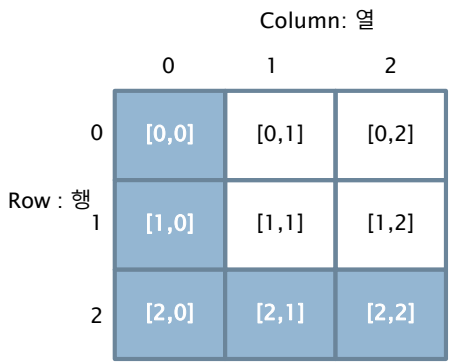


배열 접근하기 : 행렬로 구분

첫번째와 두번째 행과 두번째와 세번째 열로 접근

```
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]]
```

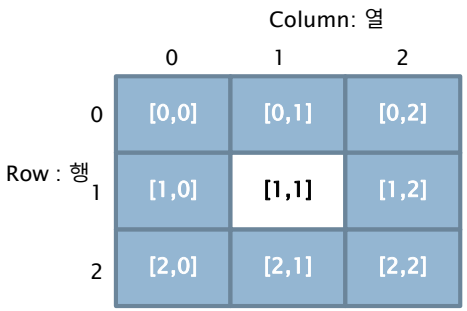


배열 접근하기 : 값

행과 열의 인덱스를 지정하면 실제 값에 접근해서 보여줌

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

  print(np33)
  print(np33[1,1])
```

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



배열 값 바꾸기 : Broadcasting

배열계산시 scalar 값과 계산시 크기가 작은 것을 동일한 크기로 계산되도록 확산이 발생

```
import numpy as np

l = [1,100,3,4,5,6,7]

npl = np.array(l,int)

npl[2:5] = 42
print npl
```

```
[  1 100  42  42  42   6   7]
```

npl[2:5]는 원소가 3개에 스칼라 값인 42를 할당했지만 [42,42,42]로 인식하여 처리

N차원 배열 처리

다차원 배열 : 열 조회/ 변경

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]]
```

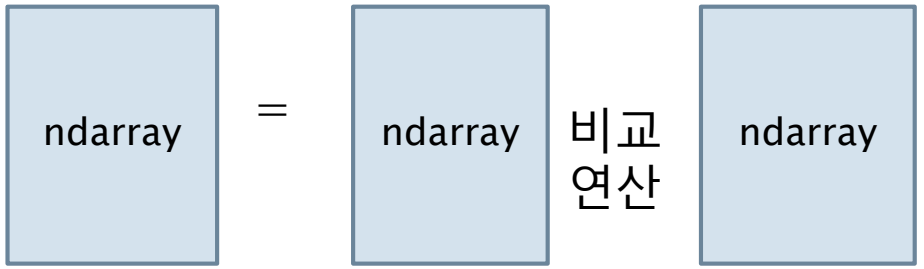
비교연산 처리

ndarray 와 비교연산 처리

ndarray와 ndarray간의 비교연산. Scala 값은 broadcasting하므로 ndarray 동일 모형의 동일값으로 인지해서 처리된 후 bool값을 가지는 ndarray가 생성됨

배열명[논리연산]

논리 연산 등 다양한 연산을 이용해서 배열 접근



1차원 배열 : 조회

[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.]
```

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

[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.        ]]
```

원소추출

행 검색

정수배열을 사용한 색인(양수,음수를 이용)이며 행에 대한 정보를 list로 제공해서 3번째와 1번째를 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]]
```

순서쌍 처리후 원소만 추출

두개의 배열을 주면 첫번째 배열은 행, 두번째 배열은 열로 순서쌍을 구성해서 값을 추출

```
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]
```

배열추출

행과 열로 구성된 배열 추출

첫번째 배열은 행을 처리 두번째 배열은 열을 처리해서 행과열로 구성된 배열 추출

```
import numpy as np

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

# arrary 형태로 출력
print(f[[1,2]][:, [0,2]])
```

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

np.ix_함수를 이용

```
import numpy as np

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

# arrary 형태로 출력
print(f[np.ix_([1,2],[0,2])])
```

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

표현식 사용

표현식 : 비교 연산

배열에 직적 비교연산 수식을 제공해서 원소 추출

```
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 나머지가 0이 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]
```

데이터 타입 정의 및 처리

numpy.dtype 생성 예시

Array 원소가 하나의 타입만 세팅할 수도 있고, 여러 개의 데이터 타입을 구성할 수 있음

Structured type : one field

```
dt = np.dtype([('f0', '<i8')])
data1 = np.zeros((3,2), dtype=dt)
print data1
print(data1.dtype)
```

```
[[(0,) (0,)]
 [(0,) (0,)]
 [(0,) (0,)]]
[('f0', '<i8')]
```

Structured type : two field

```
data = np.zeros((3,2), dtype="int, int")

print data
print(data.dtype)

dt = np.dtype([('f0', '<i8'), ('f1', '<i8')])
data1 = np.zeros((3,2), dtype=dt)
print data1
print(data1.dtype)
```

```
[[(0, 0) (0, 0)]
 [(0, 0) (0, 0)]
 [(0, 0) (0, 0)]]
[('f0', '<i8'), ('f1', '<i8')]
[[(0, 0) (0, 0)]
 [(0, 0) (0, 0)]
 [(0, 0) (0, 0)]]
[('f0', '<i8'), ('f1', '<i8')]
```

Byte 메모리 저장방식

(<: little-endian, >: big-endian, |: not-relevant),

<: little-endian

리틀 엔디안은 최하위 비트(LSB)부터 부호화되어 저장된다. 예를 들면, 숫자 12는 2진수로 나타내면 1100인데 리틀 엔디안은 0011로 각각 저장된다.

>: big-endian

이 방식은 데이터의 최상위 비트가 가장 높은 주소에 저장되므로 그냥 보기에는 역으로 보인다. 빅 엔디안은 최상위 비트(MSB)부터 부호화되어 저장되며 예를 들면, 숫자 12는 2진수로 나타내면 1100인데 빅 엔디안은 1100으로 저장된다.

|: not-relevant

문자를 저장할 때 사용 endian가 상관없이 처리

Little-endian/big-endian

itemsize 메소드는 저장되는 메모리의 크기이며 little-endian은 좌측, big-endian은 우측부터 저장

```
import numpy as np
ar = np.array([1,2,3],dtype='<i4')
print( ar.tostring())

l = ar.tostring()
print(ar.itemsize)

print(l[0:8])
```

```
b'\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
4
b'\x01\x00\x00\x00\x02\x00\x00\x00'
```

```
ar = np.array([1,2,3],dtype='>i4')
print(ar.tostring())

print(ar.dtype)
```

```
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03'
>i4
```

not-relevant

endian에 상관없이 문자를 저장할때 제일 왼쪽부터 저장됨

```
import numpy as np
sr = np.array(['a','b',"abc"],dtype='|S4')
print(sr.tostring())
print( sr.dtype)

print(sr.tostring()[0:10] )
```

```
b'a\x00\x00\x00b\x00\x00\x00abc\x00'
|S4
b'a\x00\x00\x00b\x00\x00\x00ab'
```

.

Ndarray 타입과 매칭 1

타입 매칭

Ndarray 타입	타입 코드	설명
int8, uint8	i1,u1	1byites 정수형
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 타입과 매칭2

타입 매칭

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

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

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)
```

선형대수 (벡터) 이해하기

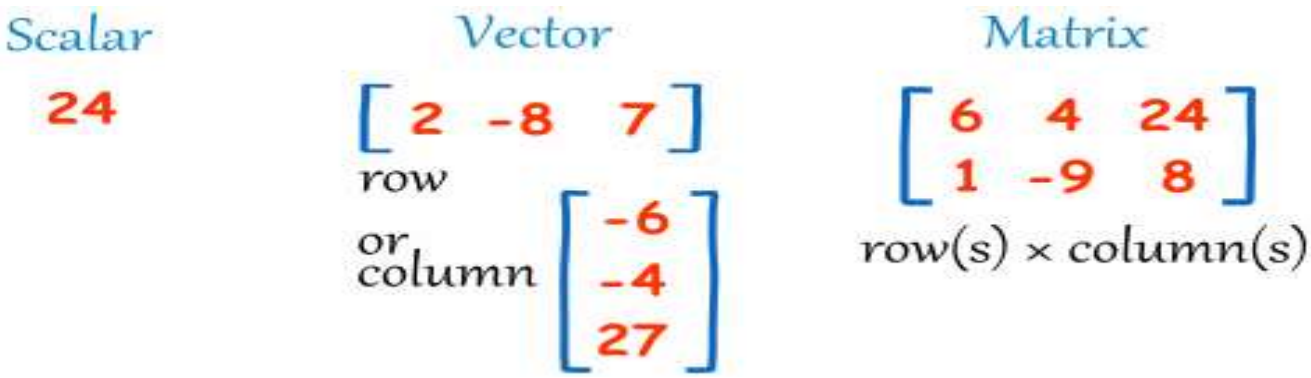
Moon Yong Joon

벡터란

스칼라/벡터/행렬

스칼라는 number, vector는 숫자들의 list(row or column), matrix는 숫자들의 array(rows, columns)

그리고 vector는 Matrix



배열과 vertor 구분

ndarray 는 벡터 1xN, Nx1, 그리고 N크기의 1차원 배열이 모두 각각 다르며, 벡터는 그 자체로 특정 좌표를 나타내기도 하지만 방향을 나타냄

scalar	배열	vector
양, 정적 위치	양, 정적 위치	변위, 속도, 힘(방향성)
1차원	N 차원	N 차원
단순 값	행,열 구분 없음	행벡터, 열벡터

스칼라/벡터/행렬 예시

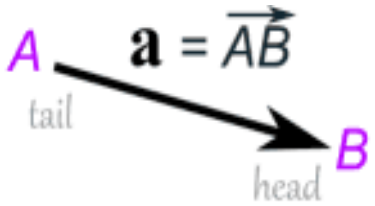
스칼라/벡터/행렬

```
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'>)
```

벡터란

크기(magnitude)와 방향(direction)을 표시

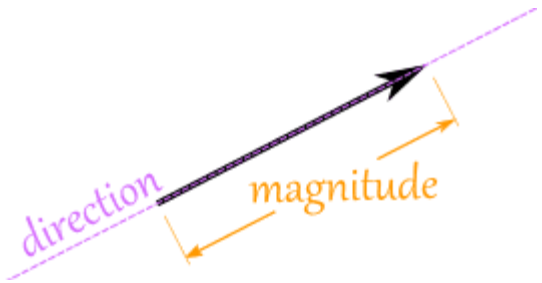


벡터는 tail부터 head까지의 유향선분으로 표시

벡터 크기

벡터 크기

벡터의 크기는 ||v|| = sqrt(v0^2 + v1^2 + v2^2... + vn^2) 로 표현



$$||u|| = \sqrt{u_x^2 + u_y^2}$$

벡터 **b** = (6,8) 의 크기

$|\mathbf{b}| = \sqrt{(6^2 + 8^2)} = \sqrt{(36+64)} = \sqrt{100} = 10$

Vector 크기 계산

벡터의 크기(Magnitude)는 원소들의 제곱을 더하고 이에 대한 제곱근의 값

벡터의 크기는 x축의 변위와 y축의 변위를 이용하여 피타고라스 정리

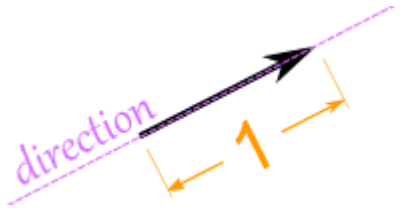
```
import math
import numpy as np

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

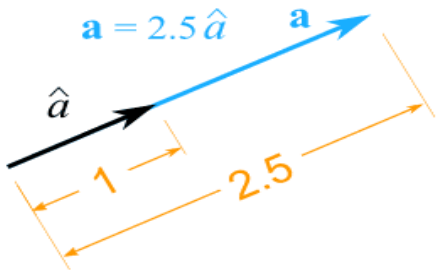
단위벡터

단위벡터(unit vector)는 크기가 1인 벡터



크기가 1인 벡터

표기법은 문자에 모자(hat)을 사용해서 표시



모든 벡터는 단위벡터에 대해 sclae 배수 만큼의 크기를 가진 벡터

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]
```

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

산술연산

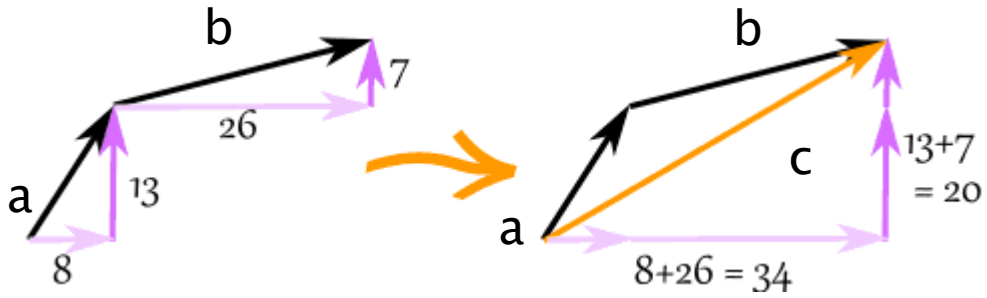
벡터: +

The vector (8,13) and the vector (26,7) add up to the vector (34,20)

Example: add the vectors a = (8,13) and b = (26,7)

c = a + b

c = (8,13) + (26,7) = (8+26,13+7) = (34,20)

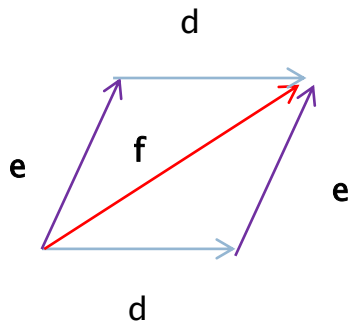


Vector 연산: +

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

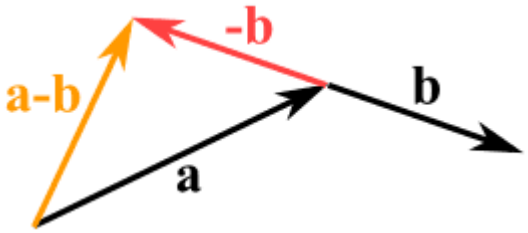
```
import math
import numpy as np

d = np.array([4,5])
e = np.array([3,8])

f = d + e
print(f)
```

```
[ 7 13]
```



벡터 : –

벡터의 방향성을 반대로 이동한 실제 벡터를 처리



Example: subtract **k** = (4,5) from **v** = (12,2)

a = **v** + −**k**

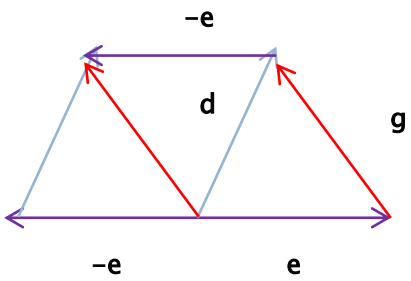
a = (12,2) + −(4,5) = (12,2) + (−4,−5) = (12−4,2−5) = (8,−3)

Vector 연산: -

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

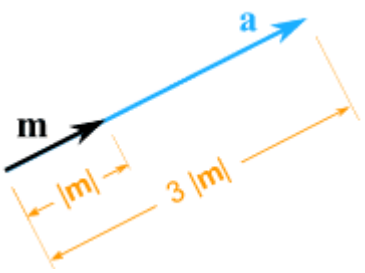
```
import math
import numpy as np

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

```
[-1  1]
```



벡터: 스칼라곱

벡터의 각 원소에 스칼라값만큼 곱하여 표시



벡터 m = [7,3]

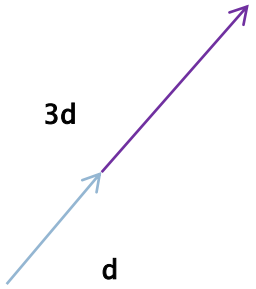
A = 3m= [21,9]

Vector 연산: 스칼라곱

스칼라 배수 만큼 벡터 내의 원소값이 커짐

```
import math
import numpy as np

d = np.array([1,2])
i = 3 * d
print(i)
```

```
[3 6]
```



스칼라와 연산

수치계산

Ndarray에 대한 수치 계산은 기본 벡터화해서 배열을 만들어서 계산함

+ : 배열간 덧셈
- : 배열간 뺄셈
* : 배열간 곱셈
/ : 배열간 나눗셈
** : 배열간 제곱
% : 배열간 나머지

```
import numpy as np

l = [1,2,3,4]
a = np.array(l,np.int)
print(a + 1)
print(a * 2)
print(a -3)
print(a /4)
```

```
[2 3 4 5]
[2 4 6 8]
[-2 -1  0  1]
[ 0.25  0.5   0.75  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]
```

선형대수
벡터 내적과 외적 이해하기

Moon Yong Joon

내적과 외적 비교

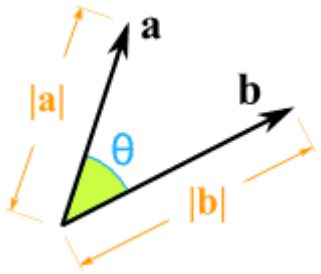
내적 vs 외적

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

스칼라곱

내적 산식

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



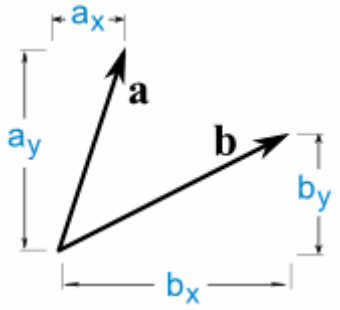
$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| \times |\mathbf{b}| \times \cos(\theta)$

Where:

$|\mathbf{a}|$: vector **a** 크기

$|\mathbf{b}|$: vector **b** 크기

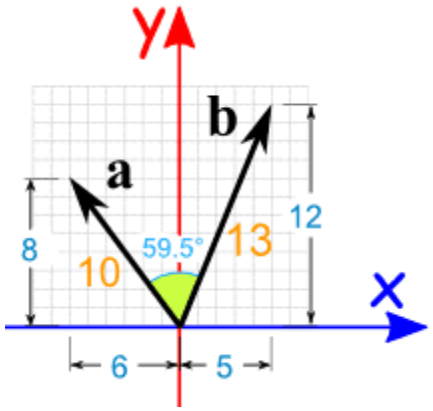
θ : **a** and **b** 사이의 각



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

내적 수학적 예시 : 2 차원

두벡터에 내적 연산에 대한 수학적 처리 예시

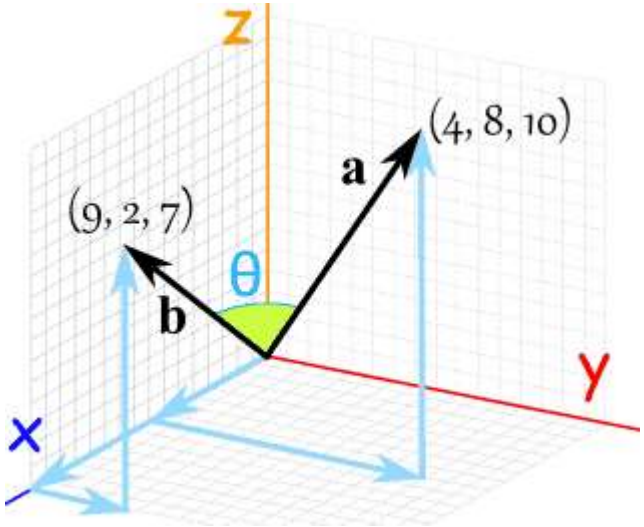


$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| \times |\mathbf{b}| \times \cos(\theta)$
$\mathbf{a} \cdot \mathbf{b} = 10 \times 13 \times \cos(59.5°)$
$\mathbf{a} \cdot \mathbf{b} = 10 \times 13 \times 0.5075...$
$\mathbf{a} \cdot \mathbf{b} = 65.98... = 66$ (rounded)

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

3차원 내적 예시 1

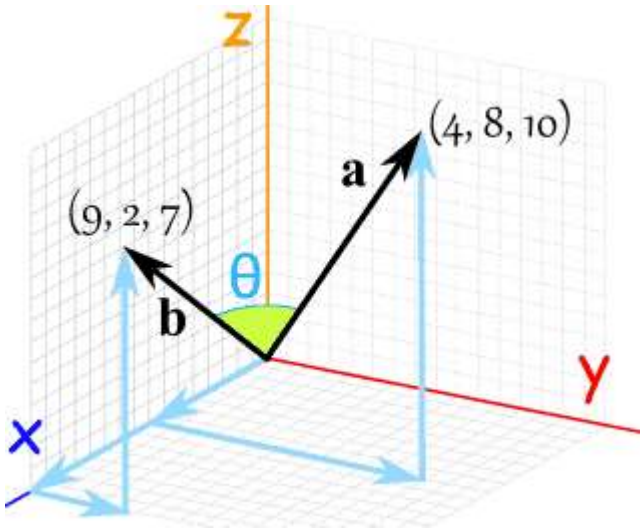
Dot 연산을 통한 계산



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

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 위치의 원소를 곱해서 합산함

두벡터의 곱셈은 단순히 원소를 곱해서 벡터를 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

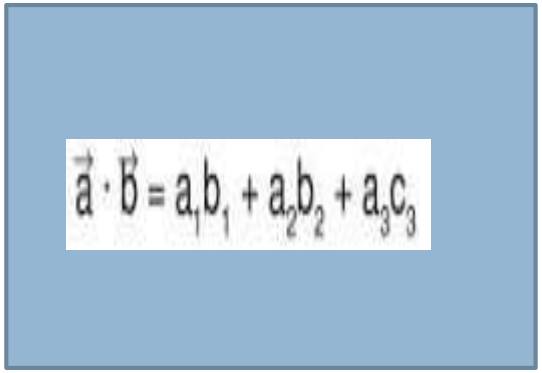
벡터(2차원)일 경우도 스칼라(dot)로 처리

```
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
```

dot 연산 : 1차원

1차원 배열간의 dot 연산은 각 원소의 곱이 합산으로 표시




```
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.dot(b))
```

```
60
```

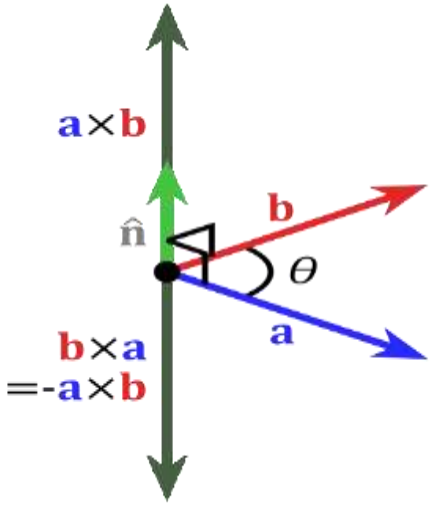
Vector product(외적)

vector 곱셈

벡터곱(vector곱, 영어: cross product) 또는 외적(外積)은 수학에서 3차원 공간의 벡터들간의 이항연산의 일종이다. 연산의 결과가 스칼라인 스칼라곱과는 달리 연산의 결과가 벡터

주요 산식 :
a*b = (a2b3−a3b2, a3b1−a1b3, a1b2−a2b1)

a = [0,0,1]
b = [0,1,0]
a*b = [0-1,0-0,0-0] = [-1,0,0]



외적

벡터 a 와 b 의 외적은 a × b 로 정의된다.

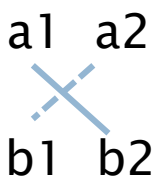
외적의 결과로 나온 벡터 c 는 벡터 a 와 b 의 수직인 벡터로 오른손 법칙의 방향

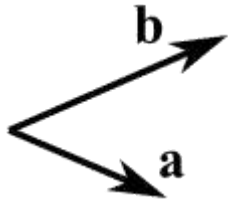
$$A = |\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}| \sin\theta.$$

외적 산식 : 2차원

벡터의 원소간의 cross 적을 처리

v = [a1,a2]
u = [b1,b2]





a1*b2 – a2*b1

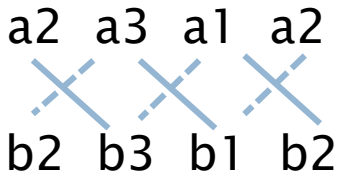
Example: The cross product of **a** = (2,3) and **b** = (5,6)
$c = a_1b_2 - a_2b_1 = 2\times6 - 3\times5 = -3$

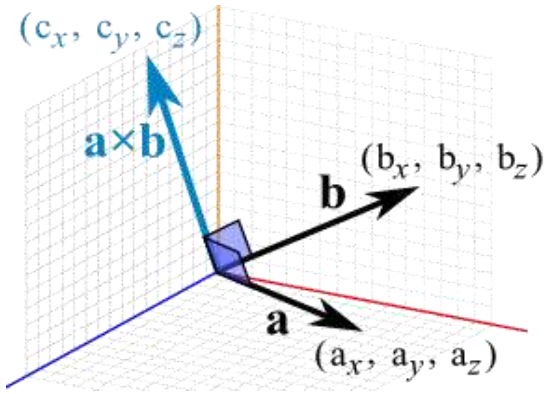
Answer: **a** × **b** = −3

외적 산식 : 3차원

벡터의 원소간의 cross 적을 처리

v = [a1,a2,a3]
u = [b1,b2,b3]





x 축 : a2*b3 – a3*b2

y 축 : a3*b1 – a1*b2

z 축 : a1*b2 – a2*b1

Example: The cross product of **a** = (2,3,4) and **b** = (5,6,7)
$c_x = a_y b_z - a_z b_y = 3\times7 - 4\times6 = -3$
$c_y = a_z b_x - a_x b_z = 4\times5 - 2\times7 = 6$
$c_z = a_x b_y - a_y b_x = 2\times6 - 3\times5 = -3$
Answer: **a** × **b** = (−3,6,−3)

cross 연산 : 1차원

벡터곱 연산을 np.cross 함수를 이용하여 처리

```
import numpy as np

x = np.array([0,0,1])
y = np.array([0,1,0])

print np.cross(x,y)
print np.cross(y,x)
```

```
[-1  0  0]
[1 0 0]
```

외적 산식예시

두벡터에 대한 외적(cross) 연산은 다른 위치의 원소를 곱해서 뺄셈

2차원 벡터는 스칼라 값으로 나옴 3차원 벡터이상 표시 됨

```
import math
import numpy as np

d = np.array([4,5])
e = np.array([3,8])

print(np.cross(d,e))

d = np.array([4,5,4])
e = np.array([3,8,2])
print(np.cross(d,e))
```

```
17
[-22   4  17]
```

Inner/outer 함수 이해하기

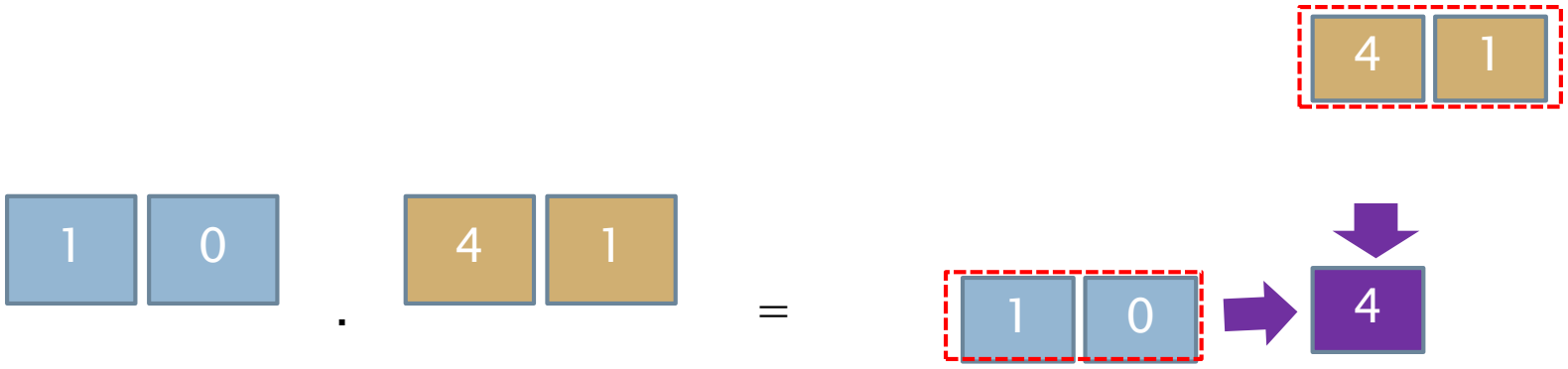
inner 계산 방식

A = [[a1,b1] B = [[a2,b2]]

numpy.inner(A,B)

array([[a1*a2 + b1*b2]])

➔[[1*4+0*1]]



Inner 예시

벡터의 내부 곱한 것을 더해서 값을 표현

```
import math
import numpy as np

a = np.array([[1,0]])
b = np.array([[4,1]])
print(a.shape, b.shape)
print(np.inner(a,b))

a1 = np.array([[1,0,3]])
b1 = np.array([[4,1,3]])
print(a1.shape, b1.shape)
print(np.inner(a1,b1))
```

```
((1, 2), (1, 2))
[[4]]
((1, 3), (1, 3))
[[13]]
```

dot/inner: 예시

벡터(2차원)일 경우 2차원으로 표시

```
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.dot(va,vc))
print(np.inner(va,vb))
```

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

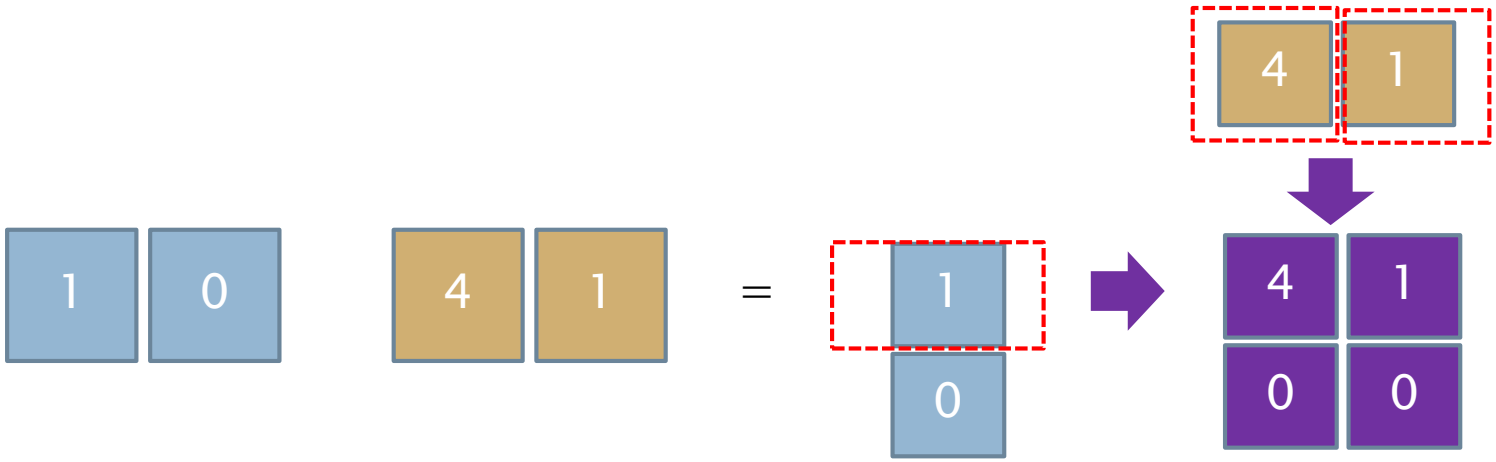
outer

A = [[a1,b1]] B = [[a2,b2]]

numpy.outer(A,B)

array([[a1*a2 , a1*b2][b1*a2, b1*b2]])

➔ [[1*4,1*1] [0*4+0*1]]



outer 예시

벡터의 내부 곱한 것을 더해서 값을 표현

첫번째 벡터의 전치와 두번째 벡터와의 Dot 연산과 같은 결과

```
import math
import numpy as np

a = np.array([[1,0]])
b = np.array([[4,1]])
print(a.shape, b.shape)
print(np.outer(a,b))
print(np.dot(a.T,b))
```

```
((1, 2), (1, 2))
[[4 1]
 [0 0]]
[[4 1]
 [0 0]]
```

PYTHON MATRIX CLASS로 VECTOR 이해하기

Moon Yong Joon

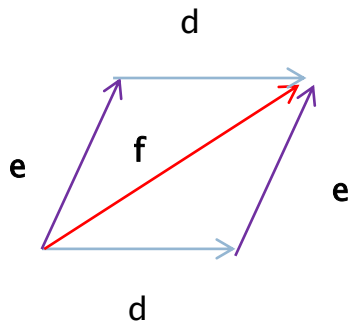
벡터 산술연산

Vector 연산: +

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

```
import math
import numpy as np

d = np.matrix([4,5])
e = np.matrix([3,8])

f = d + e
print(f)
```

```
[[ 7 13]]
```

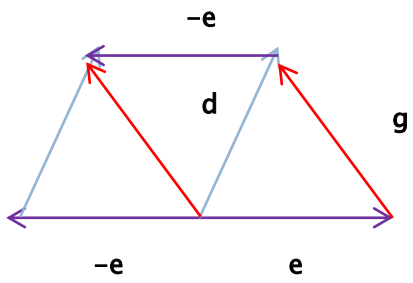


Vector 연산: -

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

```
import math
import numpy as np


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

```
[[-1  1]]
```

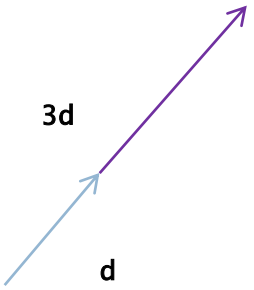


Vector 연산: 스칼라곱

벡터를 스칼라 곱 만큼 커짐

```
import math
import numpy as np

d = np.matrix([1,2])
i = 3 * d
print(i)
```

```
[[3 6]]
```



벡터 크기

Vector 크기 계산

벡터의 크기(Magnitude)는 원소들의 제곱을 더하고 이에 대한 제곱근의 값

벡터의 크기는 x축의 변위와 y축의 변위를 이용하여 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))

x = np.matrix([1,2])

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

```
2.2360679775
2.2360679775
```

$$||u|| = \sqrt{u_x^2 + u_y^2}$$

vector 내적

Vector 연산: 내적(dot)

두벡터에 대한 내적(dot) 연산은 같은 위치의 원소를 곱해서 합산함

```
import math
import numpy as np

d = np.matrix([4,5])

j= d*d.T
print(j)
print(np.dot(d,d.T))
```

```
[[41]]
[[41]]
```

vector 외적

Vector 연산: 외적(cross)

두벡터에 대한 외적(cross) 연산은 다른 위치의 원소를 곱해서 뺄셈

2차원 벡터는 array로 나옴 3차원이상으 matrix로 표시 됨

```
import math
import numpy as np

d = np.matrix([4,5])
e = np.matrix([3,8])

print(np.cross(d,e))

d = np.matrix([4,5,4])
e = np.matrix([3,8,2])
print(np.cross(d,e))
```

```
[17]
[[-22   4  17]]
```

선형대수 행렬 이해하기

Moon Yong Joon

행렬이란

행렬

매트릭스라고도 하는데 행렬의 가로 줄을 행, 세로 줄을 열로 표시함

$$\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} \Rightarrow$$ 3 x 2 (3 by 2) 행렬

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \Rightarrow$$ 2 x 3 (2 by 3) 행렬

Diagonal matrix

대각행렬

정사각행렬 A = (aij)(i, j = 1, 2, 3,…, n)의 원소 aij가 aij=0(i≠j)을 만족시키는 행렬
A의 주대각선 위에 있는 원소(대각선원소) aij(i = j) 외의 원소 aij(i≠j)가 모두 0인 행렬

$$AB = BA = \begin{bmatrix} a_1b_1 & 0 & \cdots & 0 \\ 0 & a_2b_2 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & a_nb_n \end{bmatrix}$$

(A,B는 각각 대각행렬)

Identity matrix

항등행렬

모든 행렬과 닷 연산시 자기 자신이 나오게 하는 단위행렬

$$I_1 = [1],\ I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},\ I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},\ \cdots,\ I_n = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

```
import numpy as np

a = np.array([[1,0],[0,1]])
b = np.array([[4,1],[3,2]])
print(np.dot(b,a))
print(np.dot(a,b))
```

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

Triangular matrix

삼각행렬

상삼각 행렬(Upper triangular matrix)과 하삼각 행렬(lower triangular matrix)을 총칭하여 일컫는 말.

Upper triangular matrix

$$\begin{bmatrix} 1 & 4 & 100 \\ 0 & 3 & 4 \\ 0 & 0 & 1 \end{bmatrix}$$

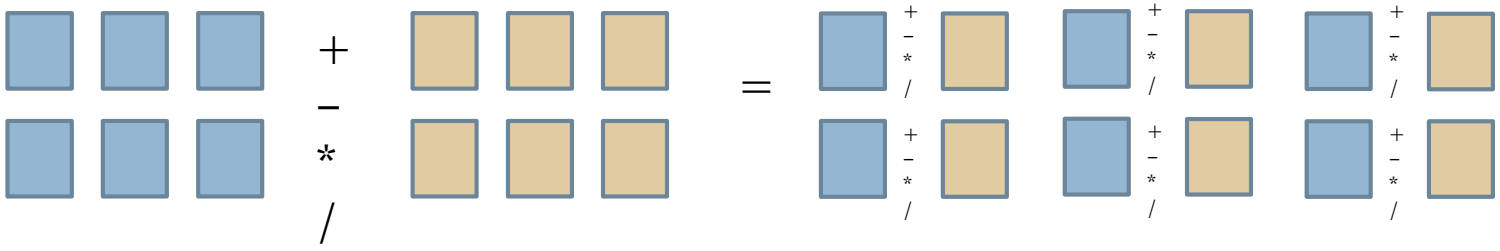
lower triangular matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 8 & 0 \\ 4 & 9 & 7 \end{bmatrix}$$

행렬 산술연산

행렬 산술연산

두 행렬의 원소별로 산술연산(+/-/*) 처리



1 2 3
4 5 6 + 1 2 3
4 5 6 = 1+1 2+2 3+3
4+4 5+5 6+6

= 2 4 6
8 10 12

행렬 산술연산 예시

행렬에 대한 산술연식은 각 원소별로 +/-/* 처리

```
:
import math
import numpy as np

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

```
[[ 6  8]
 [10 12]]
[[ 5 12]
 [21 32]]
[[ 5 12]
 [21 32]]
```

행렬의 전치(transpose)

행렬 전치

전치: 행렬의 행과 열을 서로 바꾸는 것.
수학책에서는 위첨자 T로 행렬 A의 전치를 나타낸다.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \qquad A^T = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}$$

행렬 전치 예시

python은 기본 속성에서 T 변수를 제공하고
numpy 모듈에서 transpose 함수 제공

```
import math
import numpy as np

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

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

dot 연산

dot vs inner 차이점(2차원이상)

Dot와 inner 함수는 계산시 축 기준이 차이가 있어 실제 계산된 값이 다름

dot	inner
행과 열로 계산	행과 행으로 계산
행벡터와 열벡터 간의 원소를 곱한후 덧셈	행벡터와 행벡터간의 원소를 곱한후에 덧셈
N*M 과 M*N 즉, 첫번째 열과 두번째 행이 동일	N*M과 N*M에 마지만 차원이 같은 경우
N*M . M*N 은 결과가 N*N	N*M과 N*M 은 결과가 N*N

dot 처리 기준 1*p, p*1

두 행렬 A와 B의 행렬곱셈은 행렬 A의 각 행과 행렬 B의 각 열끼리 곱해서 표시

$$\mathbf{A} = \begin{pmatrix} a_1 & a_2 & \Lambda & a_p \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_1 \\ b_2 \\ \mathrm{M} \\ b_p \end{pmatrix}$$

1행*p열

P행1열

$$\mathbf{AB} = \begin{pmatrix} a_1 b_1 & a_2 b_2 & \ldots & a_p b_p \end{pmatrix}$$

1행1열

dot 처리 기준

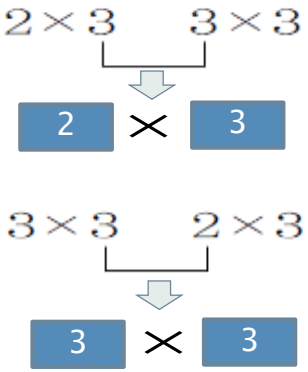
두 행렬 A와 B의 행렬곱셈은 행렬 A의 각 행과 행렬 B의 각 열끼리 곱해서 표시

$$\mathbf{AB} = \begin{pmatrix} a_{11} & a_{12} & \Lambda & a_{1j} & \Lambda & a_{1p} \\ a_{21} & a_{22} & \Lambda & a_{2j} & \Lambda & a_{2p} \\ \mathrm{M} & \mathrm{M} & & \mathrm{M} & & \mathrm{M} \\ a_{i1} & a_{i2} & \Lambda & a_{ij} & \Lambda & a_{ip} \\ \mathrm{M} & \mathrm{M} & & \mathrm{M} & & \mathrm{M} \\ a_{m1} & a_{m2} & \Lambda & a_{mj} & \Lambda & a_{mp} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & \Lambda & b_{1j} & \Lambda & b_{1n} \\ b_{21} & b_{22} & \Lambda & b_{2j} & \Lambda & b_{2n} \\ \mathrm{M} & \mathrm{M} & & \mathrm{M} & & \mathrm{M} \\ b_{i1} & b_{i2} & \Lambda & b_{ij} & \Lambda & b_{in} \\ \mathrm{M} & \mathrm{M} & & \mathrm{M} & & \mathrm{M} \\ b_{p1} & b_{p2} & \Lambda & b_{pj} & \Lambda & b_{pn} \end{pmatrix}$$

$$m \times p \qquad p \times n$$

$$\Downarrow$$

$$m \times n$$

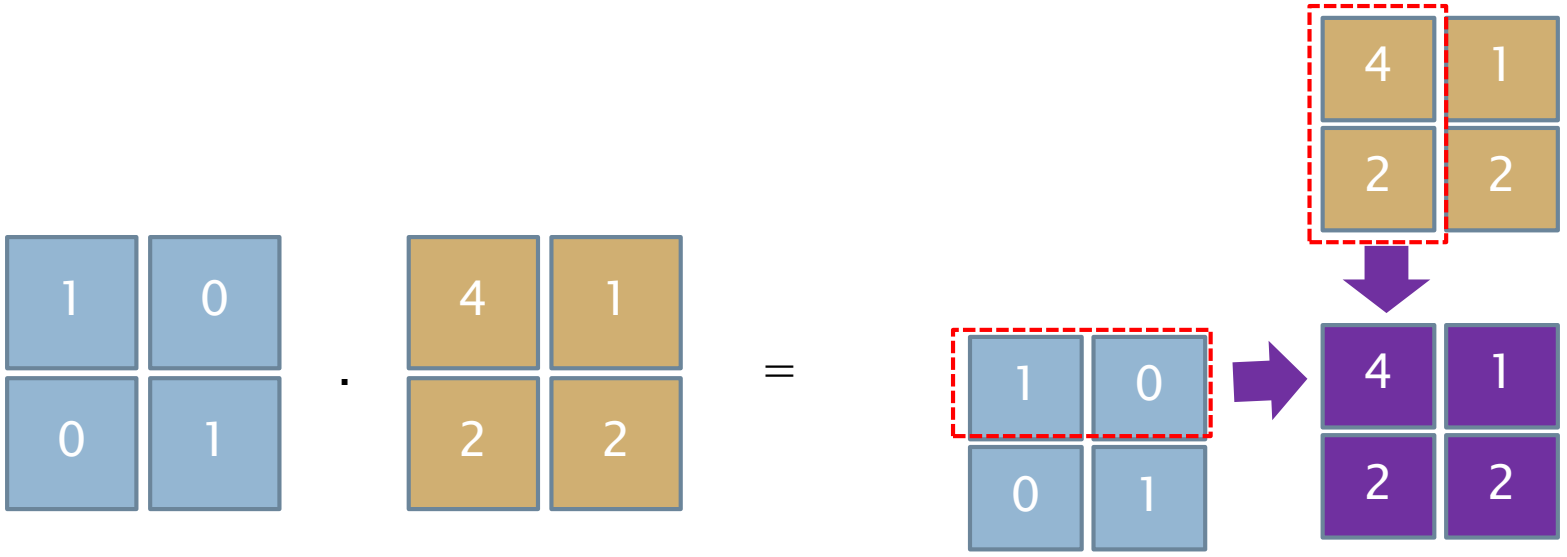


dot : 2차원

A = [[a1,b1],[c1,d1]] B = [[a2,b2],[c2,d2]]
numpy.dot(A,B)

array([[a1*a2 + b1*c2, a1*b2 + b1*d2], [c1*a2 + d1*c2, c1*b2 + d1*d2])

➔ [[1*4+ 0*2, 1*1+0*2],[0*4+1*2, 0*1+1*2]]



dot 예시

Numpy.dot 메소드 처리

```
import math
import numpy as np

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

```
[[19 22]
 [43 50]]
```

행렬식

행렬식(det)

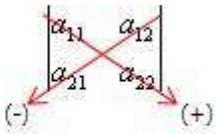
정방행렬에 하나의 수를 대응시킴으로써,

- 연립방정식의 해를 구하거나,
- 연립방정식 해의 존재성을 살피려고 할 때 쓰여짐

행렬식(det) : 2차원

행렬식 구하기

$$\det A = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}$$




```
import math
import numpy as np

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

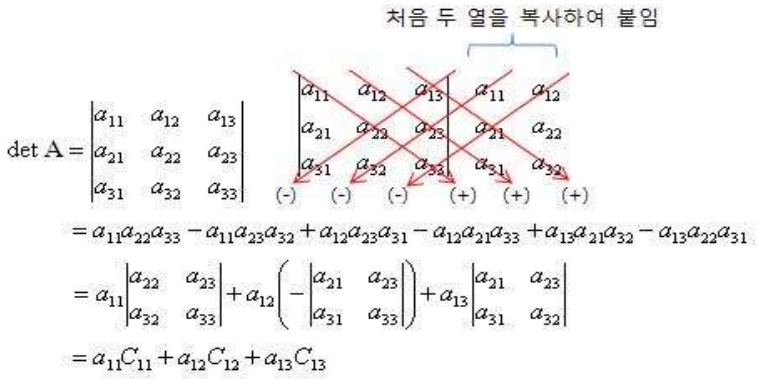
```
4.0
```

$$\det \begin{bmatrix} 3 & 1 \\ 2 & 2 \end{bmatrix} = 3*2 - 1*2$$

$$= 4$$

행렬식(det) : 3차원

행렬식을 계산시 앞에 두 열을 뒤에 복사 후 계산



$$\det A = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

$$= a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} + a_{12}a_{23}a_{31} - a_{12}a_{21}a_{33} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31}$$

$$= a_{11}\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} + a_{12}\left(-\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix}\right) + a_{13}\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{32} \end{vmatrix}$$

$$= a_{11}C_{11} + a_{12}C_{12} + a_{13}C_{13}$$

3 1 3 3 1
2 2 3 2 2
1 1 1 1 1

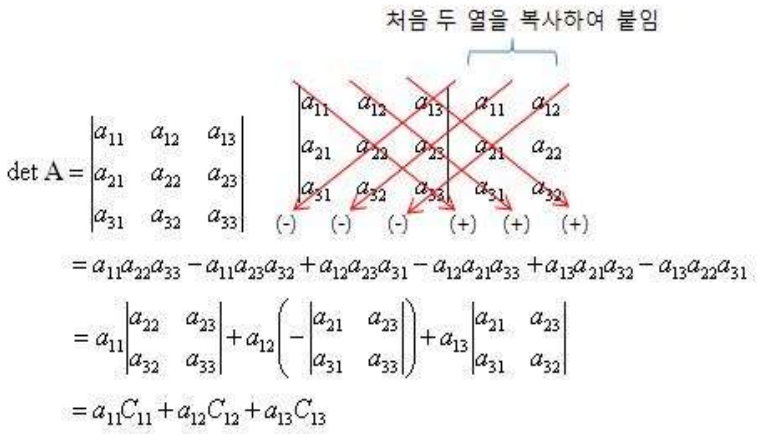
= 3*2*1 - 3*2*1 + 1*3*1 - 3*3*1 + 3*2*1 - 1*2*1
= 6 - 6 + 3 -9 + 6 -2
= 15 - 17
= -2

```
import math
import numpy as np

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

```
-2.0
```

행렬식(det) : 3차원

n



$$\det A = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

$$= a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} + a_{12}a_{23}a_{31} - a_{12}a_{21}a_{33} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31}$$

$$= a_{11}\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} + a_{12}\left(-\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix}\right) + a_{13}\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{32} \end{vmatrix}$$

$$= a_{11}C_{11} + a_{12}C_{12} + a_{13}C_{13}$$

$$det\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$= a_{11}det\begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix} - a_{12}det\begin{bmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{bmatrix} + a_{13}det\begin{bmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}$$

```
import math
import numpy as np

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

```
-2.0
```

minor determinant

소행렬식 2차원

i번째 행,j번째 열을 제거한 부분행렬의 행렬식 : Mij

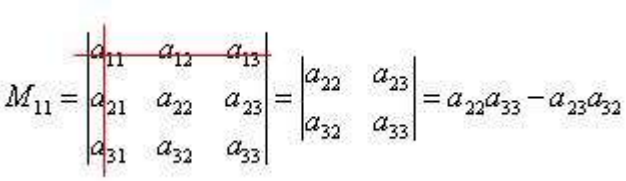
$$C_{ij} = (-1)^{i+j} M_{ij}$$

$$\begin{vmatrix} 2 & 1 \\ 1 & 2 \end{vmatrix}$$

M11	$\begin{vmatrix} 2 \end{vmatrix}$	2	부호 +
M12	$\begin{vmatrix} 1 \end{vmatrix}$	-1	부호 -
M21	$\begin{vmatrix} 1 \end{vmatrix}$	-1	부호 -
M22	$\begin{vmatrix} 2 \end{vmatrix}$	2	부호 +

소행렬식 3차원

i번째 행,j번째 열을 제거한 부분행렬의 행렬식 : Mij



$$C_{ij} = (-1)^{i+j} M_{ij}$$

3 1 3 2 2 3 1 1 1			
M11	2 3 1 1	2*1 −3*1	−1
M12	2 3 1 1	2*1 −3*1	−1
M13	2 2 1 1	2*1 −2*1	0

부호 +

부호 −

부호 +

= 3M11+(−1)* 1M12 + 3M13
= −3+1+0 = −2

소행렬식 예시

소행렬식을 구해서 행렬식 값 비교

```
import math
import numpy as np

a =  np.array([[3,1,3],[2,2,3],[1,1,1]])
print(np.linalg.det(a))
s,md = np.linalg.slogdet(a)
print(np.linalg.slogdet(a))
print(s * np.exp(md))
print(a[1:,1:])
print(np.linalg.det(a[1:,1:]))
m11 = np.linalg.det(a[1:,1:])
print(a[1:,(0,2)])
print(np.linalg.det(a[1:,(0,2)]))
m12 = np.linalg.det(a[1:,(0,2)])
print(a[1:,(0,1)])
print(np.linalg.det(a[1:,(0,1)]))
m13 = np.linalg.det(a[1:,(0,1)])
print(a[0,0]*m11-a[0,1]*m12+a[0,2] *m13)
```

```
-2.0
(-1.0, 0.6931471805599451)
-2.0
[[2 3]
 [1 1]]
-1.0
[[2 3]
 [1 1]]
-1.0
[[2 2]
 [1 1]]
0.0
-2.0
```

역행렬

여인수(cofactor)

소행렬식을 이용한 값을 여인수를 표시

$$C_{ij} = (-1)^{i+j} M_{ij}$$

$$\begin{vmatrix} 3 & 1 & 3 \\ 2 & 2 & 3 \\ 1 & 1 & 1 \end{vmatrix}$$

소행렬식				부호	결과값
m11	2 3 1 1	2-3	-1	+	-1
m12	2 3 1 1	2-3	-1	-	1
m13	2 2 1 1	2-2	0	+	0
m21	1 3 1 1	1-3	-2	-	2
m22	3 3 1 1	3-3	0	+	0
m23	3 1 1 1	3-1	2	-	-2
m31	1 3 2 3	3-6	-3	+	-3
m32	3 3 2 3	9-6	3	-	-3
m33	3 1 2 2	6-2	4	+	4

수반행렬(adj) 과 여인수행렬

소행렬식으로 계산된 원소 즉 여인수로 구성된 행렬의 전치행렬을 수반행렬이라 함

$$\text{adj}\,A = [C_{ij}]^T = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1n} \\ C_{21} & C_{22} & \cdots & C_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nn} \end{bmatrix}^T$$

$$= \begin{bmatrix} C_{11} & C_{21} & \cdots & C_{n1} \\ C_{12} & C_{22} & \cdots & C_{n2} \\ \vdots & \vdots & \vdots & \vdots \\ C_{1n} & C_{n2} & \cdots & C_{nn} \end{bmatrix}$$

$$\begin{bmatrix} -1 & 1 & 0 \\ 2 & 0 & -2 \\ -3 & -3 & 4 \end{bmatrix}^T$$

여인수행렬의 전치

$$\begin{bmatrix} -1 & 2 & -3 \\ 1 & 0 & -3 \\ 0 & -2 & 4 \end{bmatrix}$$

수반행렬

역행렬(inv) – 2차원

역행렬은 수반행렬에 행렬식으로 나눗값이 됨

$$A^{-1} = \frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \begin{bmatrix} \frac{d}{ad-bc} & -\frac{b}{ad-bc} \\ -\frac{c}{ad-bc} & \frac{a}{ad-bc} \end{bmatrix}$$

$$= \frac{\text{adj A}}{\det \text{A}}$$

$A_{-1} = 1/\det(A) * C_T$

$$\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}^{-1} \quad 1/3 * \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}$$

역행렬

```
[[ 0.66666667 -0.33333333]
 [-0.33333333  0.66666667]]
```

역행렬(inv) – 3차원

역행렬은 수반행렬에 행렬식으로 나눗값이 됨

$$A^{-1} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}^{-1} = \frac{1}{|A|} \begin{bmatrix} ei - fh & ch - bi & bf - ce \\ fg - di & ai - cg & cd - af \\ dh - eg & bg - ah & ae - bd \end{bmatrix}$$

$A_{-1} = 1 / \det(A) * C_T$

$$-0.5 * \begin{bmatrix} -1 & 2 & -3 \\ 1 & 0 & -3 \\ 0 & -2 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 1 & 3 \\ 2 & 2 & 3 \\ 1 & 1 & 1 \end{bmatrix}^{-1}$$

역행렬

```
[[ 0.5 -1.   1.5]
 [-0.5  0.   1.5]
 [ 0.   1.  -2. ]]
```

역행렬(inv) 예시

역행렬 계산

```
import math
import numpy as np

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

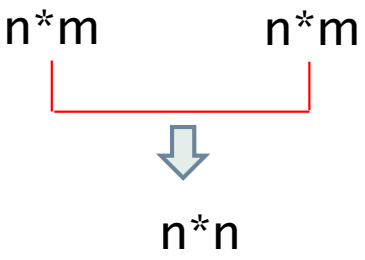
```
[[ 0.5 -1.   1.5]
 [-0.5  0.   1.5]
 [ 0.   1.  -2. ]]
```

Dot 연산

Dot 처리 기준

두 행렬 A와 B의 행렬곱셈은 행렬 A의 각 행과 행렬 B의 각 행끼리 곱한후 덧셈을 하여 표시

$$\mathbf{AB} = \begin{pmatrix} a_{11} & a_{12} & \Lambda & a_{1j} & \Lambda & a_{1p} \\ a_{21} & a_{22} & \Lambda & a_{2j} & \Lambda & a_{2p} \\ M & M & & M & & M \\ a_{i1} & a_{i2} & \Lambda & a_{ij} & \Lambda & a_{ip} \\ M & M & & M & & M \\ a_{m1} & a_{m2} & \Lambda & a_{mj} & \Lambda & a_{mp} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & \Lambda & b_{1j} & \Lambda & b_{1n} \\ b_{21} & b_{22} & \Lambda & b_{2j} & \Lambda & b_{2n} \\ M & M & & M & & M \\ b_{i1} & b_{i2} & \Lambda & b_{ij} & \Lambda & b_{in} \\ M & M & & M & & M \\ b_{p1} & b_{p2} & \Lambda & b_{pj} & \Lambda & b_{pn} \end{pmatrix}$$



두행렬의 마지막 차원이 값으면 처리가 가능하고 결과는 마지막 차원을 제외해서 구성

dot 행렬

n*m 행렬 일 경우 2차원으로 표시

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

a = np.array([[1, 0], [0, 1]])
b = np.array([[4, 1], [2, 2]])
print(a.ndim, a.shape)
print(b.ndim, b.shape)
print(np.dot(a, b))
print(a[0])
print(b[0])
print(np.dot(a[0], b[0]))
print(np.dot(a[1], b[1]))
```

```
(2, (2, 2))
(2, (2, 2))
[[4 1]
 [2 2]]
[1 0]
[4 1]
4
2
```

dot 예시 : 2차원

a(2,2) 행렬과 b(2,2)행렬의 마지막 차수가 같으므로 계산결과는 n*m, m*n = n*n

```
import math
import numpy as np

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

```
[[19 22]
 [43 50]]
```

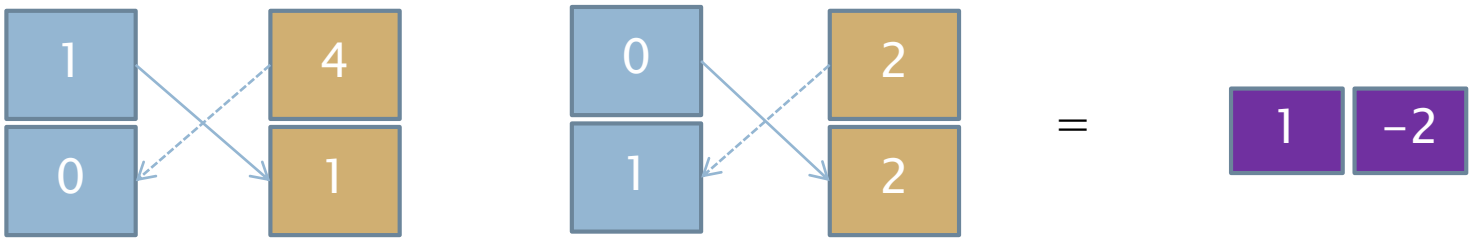
cross product

cross 계산 방식

A = [[a1,b1],[c1,d1]] B = [[a2,b2],[c2,d2]]
numpy.cross(A,B) = A.T * B

array([[a1*b2 – c1*a2 , b1*d2 – d1*c2])

➔[[1*1– 0*4,0*2–1*2]]



Cross 행렬

n*m 행렬 일 경우 2차원으로 표시

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

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

```
[ 1 -2]
```

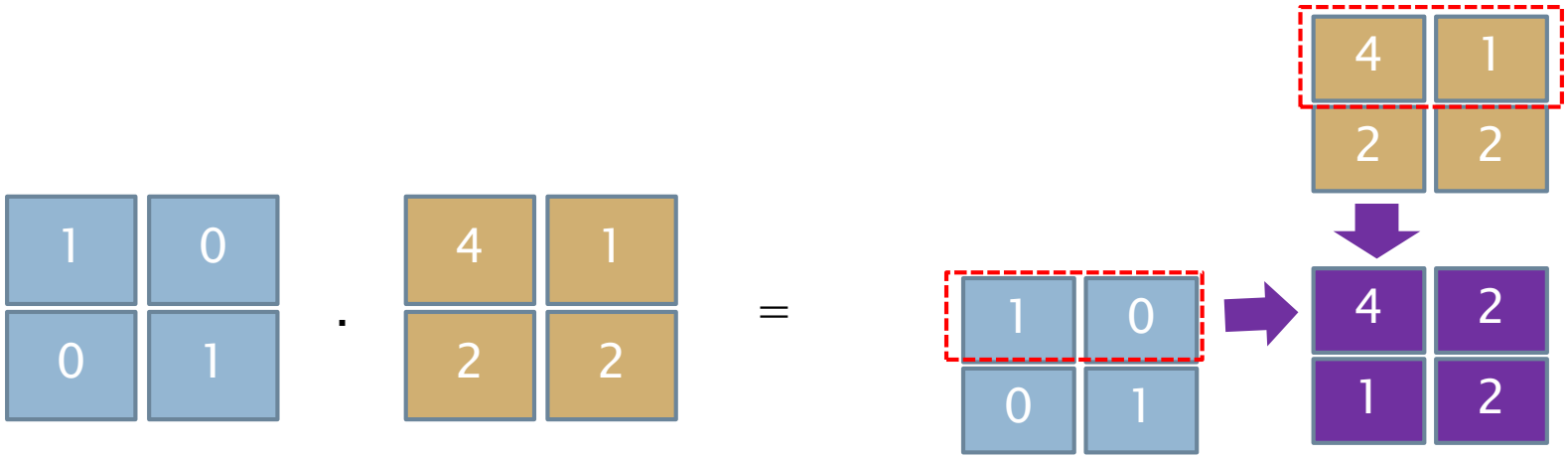
Inner 연산

inner 계산 방식

A = [[a1,b1],[c1,d1]] B = [[a2,b2],[c2,d2]]
numpy.inner(A,B)

array([[a1*a2 + b1*b2, a1*c2 + b1*d2], [c1*a2 + d1*b2, c1*c2 + d1*d2])

➔[[1*4+0*1,1*2+0*2],[0*4+1*1, 0*2+1*2]]



Inner 예시 : 2차원

a(2,2) 행렬과 b(2,2)행렬의 마지막 차수가 같으므로 계산결과는 out.shape = a.shape[:-1] + b.shape[:-1]

```
import math
import numpy as np

a = np.array([[1,0],[0,1]])
b = np.array([[4,1],[2,2]])
print(a.shape, b.shape)
print(np.inner(a,b))
```

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

Inner 예시 : 3차원

a(2,3,2) 행렬과 b(2,2)행렬의 마지막 차수가 같으므로 계산결과는 out.shape = a.shape[:-1] + b.shape[:-1]

```
import math
import numpy as np

a = np.array(np.arange(12).reshape(2,3,2))
b = np.array([[5,6],[7,8]])

print(a.shape)
print(np.inner(a,b))
```

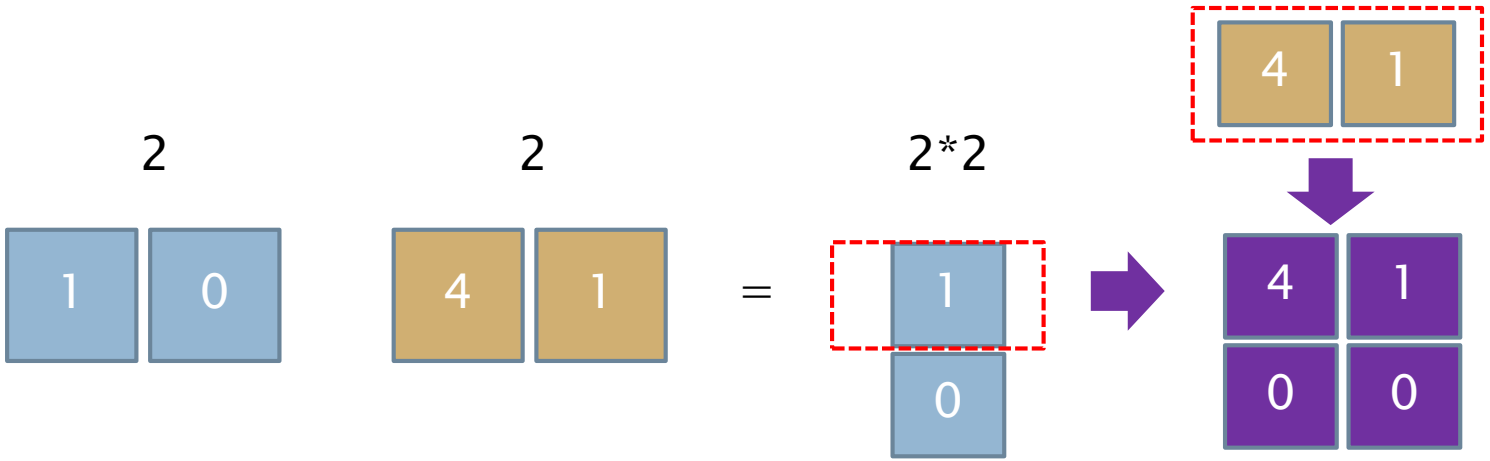
```
(2, 3, 2)
[[[  6   8]
  [ 28  38]
  [ 50  68]]

 [[ 72  98]
  [ 94 128]
  [116 158]]]
```

outer product

outer

두개의 벡터를 가지고 벡터 크기를 행과 열로 만드는 함수

1차원이 이상일 경우 1차원으로 만든 후에 행렬로 만듬



outer: 1

Out는 두개의 벡터에 대한 행렬로 구성
out[i, j] = a[i] * b[j]

```
import numpy as np

a = np.array([[1,2,3]])
b = np.array([[4,5]])

print(np.outer(a,b))

a = np.array([[1,0]])
b = np.array([[4,1]])

print(np.outer(a,b))
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
[[4 1]
 [0 0]]
```

outer: 2

첫번째 벡터가 행이되고 두번째 벡터가 열이 되어 5*5행렬을 만듬

```
import numpy as np

rl = np.outer(np.ones((5,)), np.linspace(-2, 2, 5))

print(np.ones(5,))
print(np.linspace(-2,2,5))

print(rl)
```

```
[ 1.  1.  1.  1.  1.]
[-2. -1.  0.  1.  2.]
[[-2. -1.  0.  1.  2.]
 [-2. -1.  0.  1.  2.]
 [-2. -1.  0.  1.  2.]
 [-2. -1.  0.  1.  2.]
 [-2. -1.  0.  1.  2.]]
```

outer: 3

벡터의 값이 문자일 경우 문자 배수만큼 처리

```
import numpy as np

x = np.array(['a', 'b', 'c'], dtype=object)
print(np.outer(x, [1, 2, 3]))

y = np.array([1,2,3], dtype=object)
print(np.outer(y, ['a', 'b', 'c']))
```

```
[['a' 'aa' 'aaa']
 ['b' 'bb' 'bbb']
 ['c' 'cc' 'ccc']]
[['a' 'b' 'c']
 ['aa' 'bb' 'cc']
 ['aaa' 'bbb' 'ccc']]
```

tensordot

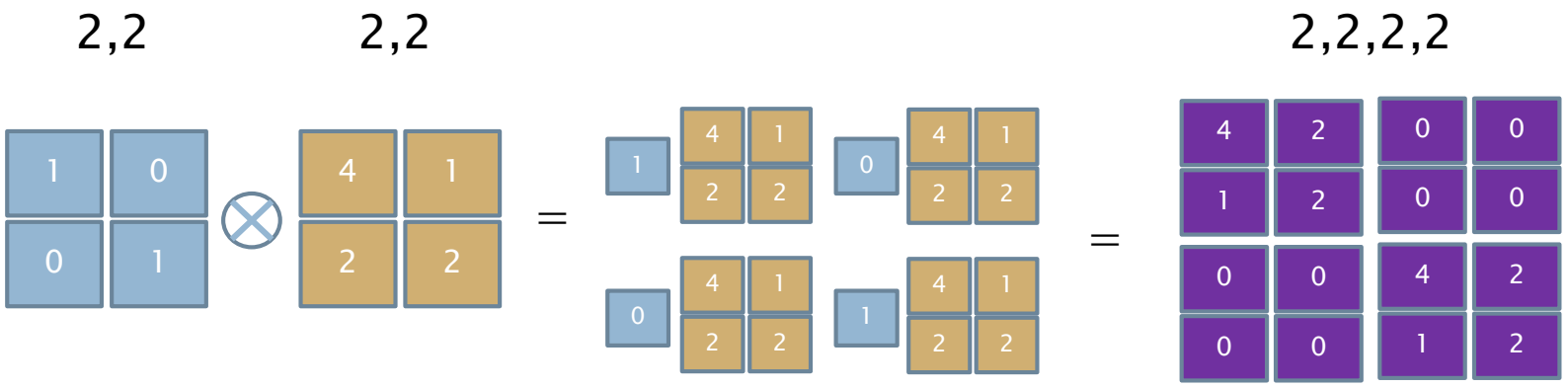
tensordot

Tensordot 함수에 axes를 0으로 줄 경우 tensor product을 연산

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \otimes \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} = \begin{bmatrix} a_{1,1}\begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} & a_{1,2}\begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} \\ a_{2,1}\begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} & a_{2,2}\begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} a_{1,1}b_{1,1} & a_{1,1}b_{1,2} & a_{1,2}b_{1,1} & a_{1,2}b_{1,2} \\ a_{1,1}b_{2,1} & a_{1,1}b_{2,2} & a_{1,2}b_{2,1} & a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} & a_{2,1}b_{1,2} & a_{2,2}b_{1,1} & a_{2,2}b_{1,2} \\ a_{2,1}b_{2,1} & a_{2,1}b_{2,2} & a_{2,2}b_{2,1} & a_{2,2}b_{2,2} \end{bmatrix}.$$

tensordot

Tensordot 함수에 axes를 0으로 줄 경우 tensor product을 연산



tensordot: 예시 1

2차원 행렬 2개가 만나 4차원 행렬 구성

```
import numpy as np

a = [[1, 0], [0, 1]]
b = [[4, 1], [2, 2]]
ts = np.tensordot(a,b,axes=0)
print(ts)
print(ts.ndim, ts.shape)

print(ts[0])
print(ts[0][0])
print(ts[0][0][0])
print(ts[0][0][0][0])
```

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

  [[0 0]
   [0 0]]]


 [[[0 0]
   [0 0]]

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

 [[0 0]
  [0 0]]]
[[4 1]
 [2 2]]
[4 1]
4
```

tensordot: 예시 2

axes = 0 tensor product, axes = 1 tensor dot product, axes = 2 tenser double contraction 즉 벡터연산

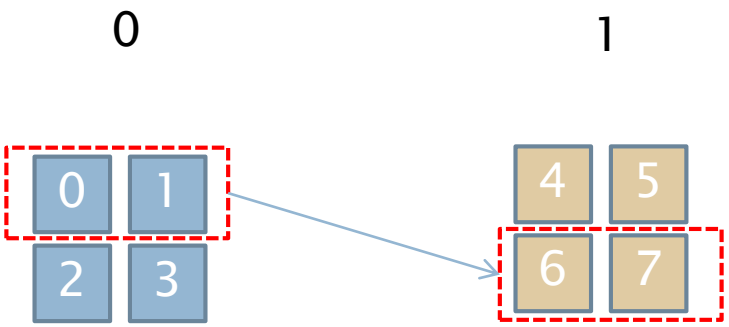
```
import numpy as np

a = [[1, 0], [0, 1]]
b = [[4, 1], [2, 2]]
ts = np.tensordot(a,b,axes=0)
print(ts)
print(ts.ndim, ts.shape)
print(np.tensordot(a,b,axes=1))
print(np.dot(a,b))
print(np.tensordot(a,b,axes=2))
print(np.dot([1,0,0,1],[4,1,2,2]))
```

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

  [[0 0]
   [0 0]]]


 [[[0 0]
   [0 0]]

  [[4 1]
   [2 2]]]]
(4, (2, 2, 2, 2))
[[4 1]
 [2 2]]
[[4 1]
 [2 2]]
6
6
```

대각행렬

Trace : 3차원 행렬

3차원(2,2,2) 대각행렬의 합은 첫번째 차원의 1과 두번째의 마지막을 합산해서 출력



trace

대각행렬의 합을 출력

```
import numpy as np

a = np.array([[1,0],[1,0]])
b = np.array([[4,1],[4,1]])
print(np.trace(a))
print(np.trace(b))

a = np.arange(8).reshape((2,2,2))
print(a)
print(np.trace(a))
```

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

 [[4 5]
  [6 7]]]
[6 8]
```

PYTHON MATRIX CLASS로 행렬 이해하기

Moon Yong Joon

행렬 이해하기

행렬

n개의 실수의 순서쌍에 성분별로 덧셈과 실수상수곱을 주면[2] 이는 "nn차원" 벡터공간이라 할 수 있고(, 벡터공간에서 벡터공간으로 가는 함수 중 덧셈과 상수배를 보존하는 함수를 선형사상을 행렬이라 함

행렬 생성

Numpy matrix 를 이용해서 행렬 생성

```
import numpy as np

d = np.array([4,5,4])
e = np.array([3,8,2])

m1 = np.matrix([d,e])
print(m1)
m2 = np.matrix([d,e])
m3 = m2.reshape(3,2)
print(m3)
```

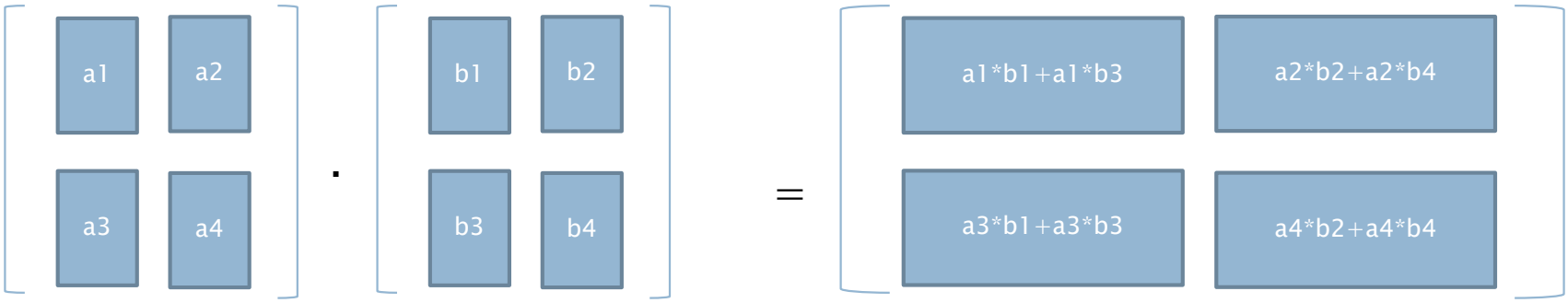
```
[[4 5 4]
 [3 8 2]]
[[4 5]
 [4 3]
 [8 2]]
```

행렬 연산하기

행렬 : 내적 dot(곱셈)

N*M 과 M* N인 행렬에 대한 dot 연산 처리 결과는 M*M으로 나옴

$$(x_{ij})(y_{ij})=(\Sigma_k x_{ik} y_{kj})$$



행렬 : dot

행렬에 대한 dot 연산 처리

```
import math
import numpy as np

a = np.array([[1, 4], [5, 6],[7,7]])
b = np.array([[4, 1], [2, 2]])

print(a)
print(b)
print(np.dot(a,b))
```

```
[[1 4]
 [5 6]
 [7 7]]
[[4 1]
 [2 2]]
[[12  9]
 [32 17]
 [42 21]]
```

행렬 : 외적cross

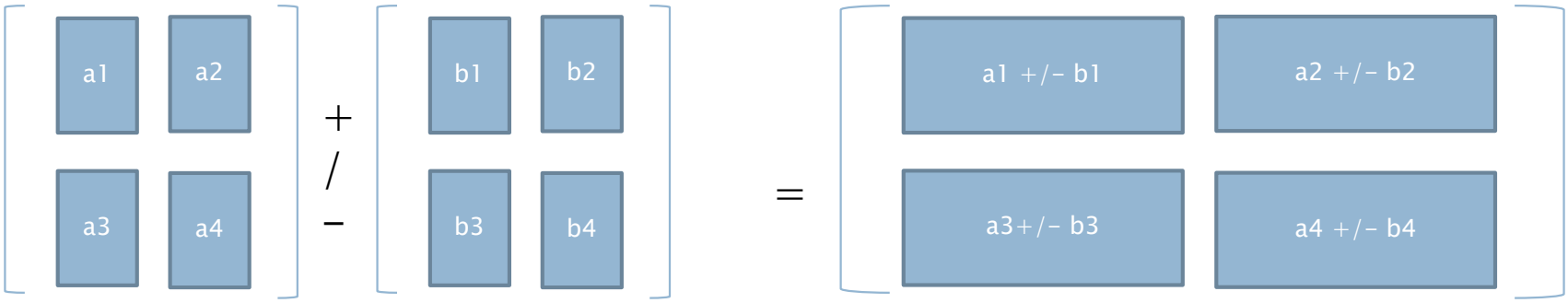
행렬에 대한 cross는 동등한 행렬일 경우 연산 처리

```
import math
import numpy as np

d = np.array([4,5,4])
e = np.array([3,8,2])

m1 = np.matrix([d,e])
print(m1)
f = np.array([2,9,4])
g = np.array([7,8,3])
m2 = np.matrix([f,g])

print(np.cross(m1,m2))
```

```
[[4 5 4]
 [3 8 2]]
[[-16  -8  26]
 [  8   5 -32]]
```

행렬 : +/-

N*M 과 N*M인 행렬에 대한 +/- 연산 처리 결과는 N*M으로 나옴



행렬 : +/-

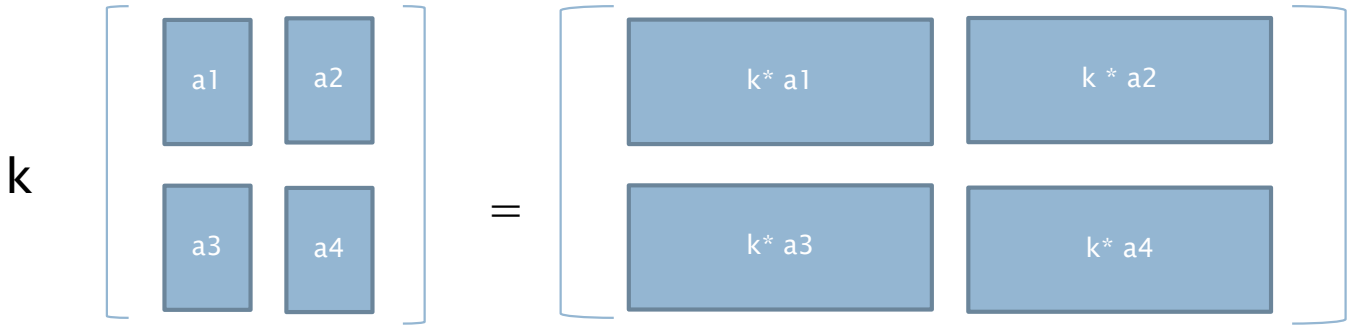
행렬에 대한 +/- 연산 처리

```
import math
import numpy as np

a = np.array([[1, 4], [5, 6]])
b = np.array([[4, 1], [2, 2]])
print(a + b)
print(a - b)
```

```
[[5 5]
 [7 8]]
[[-3  3]
 [ 3  4]]
```

행렬 : 상수 배

상수(k) 와 N*M 행렬에 대한 곱은 상수배만큼 증가함



행렬 : 상수 배

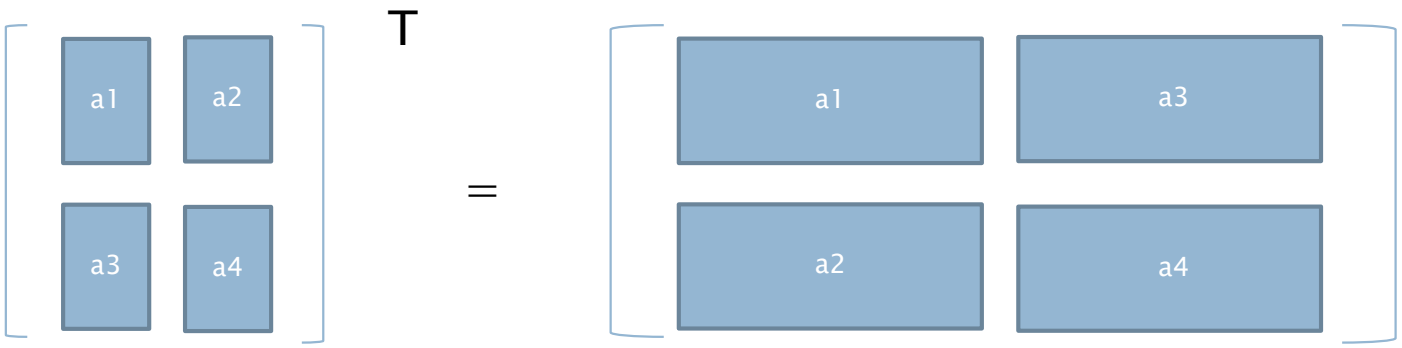
행렬에 대한 k 상수만큼 원소별로 곱하는 연산 처리

```
import math
import numpy as np

a = np.array([[1, 4], [5, 6]])
b = np.array([[4, 1], [2, 2]])


print(5*a)
```

```
[[ 5 20]
 [25 30]]
```

행렬 : 전치(transpose)

N*M 행렬을 M*N을 변환하는 처리



행렬 : 전치(transpose)

N*M 행렬을 M*N을 변환하는 방식은 T변수, transpose 메소드가 있음

```
import math
import numpy as np

a = np.array([[1, 4], [5, 6]])
b = np.array([[4, 1], [2, 2]])


print(a)
print(a.T)
print(a.transpose())
```

```
[[1 4]
 [5 6]]
[[1 5]
 [4 6]]
[[1 5]
 [4 6]]
```

matmul

Matrix 타입일 경우 곱셈은 dot 연산과 동일한 결과를 생성함

```
import numpy as np

a = [[1, 0], [0, 1]]
b = [[4, 1], [2, 2]]
print(np.matmul(a, b))
print(np.dot(a,b))
print(np.inner(a,b))
a = [[1, 0,3], [0, 1,3]]
b = [[4, 1], [2, 2],[3,3]]
print(np.matmul(a, b))
print(np.dot(a,b))
# print(np.inner(a,b)) 동일하지 않아서 오류처리
```

```
[[4 1]
 [2 2]]
[[4 1]
 [2 2]]
[[4 2]
 [1 2]]
[[13 10]
 [11 11]]
[[13 10]
 [11 11]]
```

Matmul: 차원계산

N*m, M*n 행렬에 따라 계산이 되지만 1차원인 경우는 행렬 계산을 처리

```
import numpy as np

a = [[1, 0], [0, 1]]
b = [1, 2]
print(np.matmul(a, b))
print(np.matmul(b, a))
a = [[1, 0], [0, 1]]
b = [[4, 1,1], [2, 2,3]]
print(np.matmul(a, b))
#print(np.matmul(b, a))   행렬에 대한 오류 발생
```

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

matrix_power

matrix_power는 정방행렬에 대해 dot 연산을 제곱승만큼 계산하는 것

$$A^3 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}^3$$

Step 1: Raised to the 2nd power

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}^2 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}^2 = \begin{pmatrix} 30 & 36 & 42 \\ 66 & 81 & 96 \\ 102 & 126 & 150 \end{pmatrix}$$

Step 2: Raised to the 3rd power
(Multiply the above matrix with the initial one)

$$\begin{pmatrix} 30 & 36 & 42 \\ 66 & 81 & 96 \\ 102 & 126 & 150 \end{pmatrix} * \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 468 & 576 & 684 \\ 1062 & 1305 & 1548 \\ 1656 & 2034 & 2412 \end{pmatrix}$$

matrix_power: 예시

반복적인 dot 연산을 처리

```
import numpy as np

from numpy import linalg as LA

i = np.array([[0, 1], [-1, 0]])
# matrix equiv. of the imaginary unit
print(LA.matrix_power(i, 3))
 # should = -i

i = np.array([[2, 3], [4, 5]])
print(LA.matrix_power(i, 3))
s = np.dot(i,i)
print(np.dot(s,i))
```

```
[[ 0 -1]
 [ 1  0]]
[[116 153]
 [204 269]]
[[116 153]
 [204 269]]
```

PYTHON NUMPY LINALG 함수 이해하기

Moon Yong Joon

Matrix and vector products

주요 함수

선형대수에 대한 함수들

함수	설명
dot(a, b[, out])	n차원 행렬 n*m m*l에 대한 production(결과는 n*l)
vdot(a, b)	Vector에 대한 prodution
inner(a, b)	N 차원 행렬에 대한 Inner product (행렬이 동일해야 함).
outer(a, b[, out])	2개 벡터에 대해 계산 후 행렬로 표시.
matmul(a, b[, out])	두 행렬에 대한 Matrix product (dot과 동일한 결과)
tensordot(a, b[, axes])	Compute tensor dot product along specified axes for arrays >= 1-D.
linalg.matrix_power(M, n)	Raise a square matrix to the (integer) power *n*.
cross(a, b, axisa=-1, axisb=-1, axisc=-1, axis=None)	행렬에 대한 외적을 구함
einsum(subscripts, *operands[, out, dtype, ...])	Evaluates the Einstein summation convention on the operands.
kron(a, b)	Kronecker product of two arrays.

Decompositions

주요 함수

선형대수에 대한 함수들

함수	설명
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

주요 함수

선형대수에 대한 함수들

함수	설명
linalg.eig(a)	Compute the eigenvalues and right eigenvectors of a square array.
linalg.eigh(a[, UPLO])	Return the eigenvalues and eigenvectors of a Hermitian or symmetric matrix.
linalg.eigvals(a)	Compute the eigenvalues of a general matrix.
linalg.eigvalsh(a[, UPLO])	Compute the eigenvalues of a Hermitian or real symmetric matrix.
linalg.eig(a)	Compute the eigenvalues and right eigenvectors of a square array.

Norms and other numbers

주요 함수

선형대수에 대한 함수들

함수	설명
linalg.norm(x[, ord, axis, keepdims])	Matrix or vector norm.
linalg.cond(x[, p])	Compute the condition number of a matrix.
linalg.det(a)	Compute the determinant of an array.
linalg.matrix_rank(M[, tol])	Return matrix rank of array using SVD method Rank of the array is the number of SVD singular values of the array that are greater than *tol*.
linalg.slogdet(a)	Compute the sign and (natural) logarithm of the determinant of an array.
trace(a[, offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.

Solving equations and inverting matrices

주요 함수

선형대수에 대한 함수들

함수	설명
linalg.solve(a, b)	Solve a linear matrix equation, or system of linear scalar equations.
linalg.tensorsolve(a, b[, axes])	Solve the tensor equation a x = b for x.
linalg.lstsq(a, b[, rcond])	Return the least-squares solution to a linear matrix equation.
linalg.inv(a)	Compute the (multiplicative) inverse of a matrix.
linalg.pinv(a[, rcond])	Compute the (Moore-Penrose) pseudo-inverse of a matrix.
linalg.tensorinv(a[, ind])	Compute the ‘inverse’ of an N-dimensional array.