Popular Modules for Data Analytics

- (ch 30) Numpy
- (ch 31) Pandas
- (ch 32) Matplotlib, Seaborn
- (ch 33) Scipy
- (ch 34) Scikit learn

(Ch 30) Numpy: Table of Contents

Matrix Data in Python List [1/2]

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

```
data = [ [1, 2, 3, 4],

         [5, 6, 7, 8],

         [9, 10, 11, 12]

       ]
```

```
def sum_matrix(table):
    sum = 0
    for row in range(0,len(table)):
        for col in range(0,len(table[row])):
            sum = sum + table[row][col]
    return sum
```

Matrix Data in Python List [2/2]

```
data = [ [1, 2, 3, 4],
         [5, 6, 7, 8],
         [9, 10, 11, 12]
       ]
```

X_data Y_data

```
data = [ [1, 2, 3, 4],
         [5, 6, 7, 8],
         [9, 10, 11, 12]
       ]
```

Statistics module

```
>>>from statistics import *
>>>print(mean( [2, 6, 20] )
```

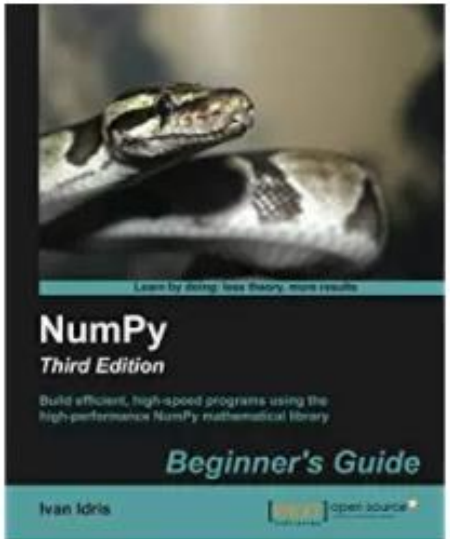
Sklearn module

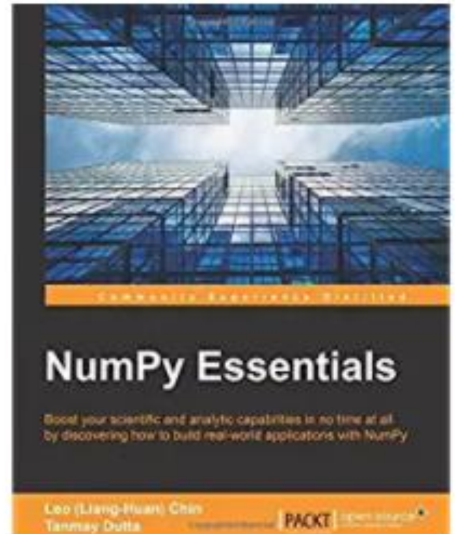
```
>>>from sklearn import linear_model
>>>
>>> regr = linear_model.LinearRegression( )
>>> regr.fit( X_data, Y_data)
>>> regr
```

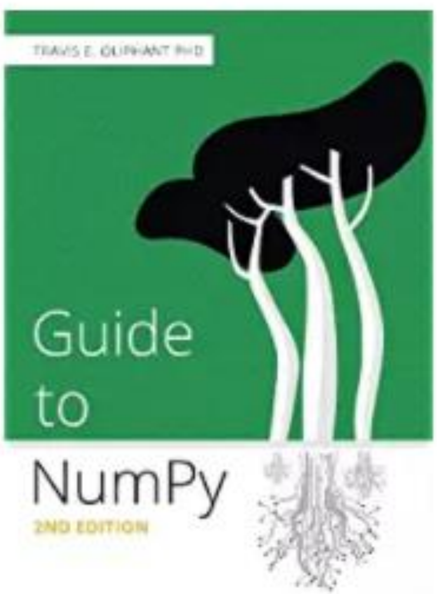
Plot module

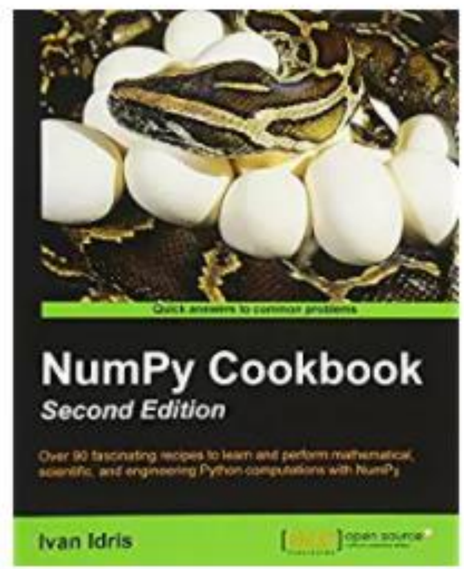
```
>>>from matplotlib.pyplot as plt
>>>
>>> plt.plot( X_data, Y_data )
```

Many Numpy Books









What is “numpy” Module?

- The “numpy” NumPy (Numeric Python) package provides basic routines for manipulating large arrays and matrices of numeric data

- The “scipy” SciPy (Scientific Python) package extends the functionality of NumPy with a substantial collection of useful algorithms
  - Minimization, Fourier Transformation, Regression, and Other Applied Math Techniques

- Numpy and SciPy are open source add-on modules (not Python Standard Library)

- More than functionalities of commercial packages like MatLab

- To catch up functionalities of R

- >>> import numpy as np
- Need to install “numpy.py” within Python directory

“numpy” Module : History

- Numeric (ancestor of NumPy)
  - was released in 1995, created by Jim Hugunin

- Numarray (second generation of Numeric)
  - Faster for large arrays
  - Slower than Numeric on small arrays

- Scipy module
  - Was released in 2001, created by Travis Oliphant et al.
  - Provides numerical operations on top of Numeric (later Numpy)
  - Provides scientific and technical operations
  - Stable version : 0.19.1 / 22 June 2017

- Numpy module
  - was released in 2005, created by Travis Oliphant
  - Incorporating features of Numeric with extensive modification
  - Numeric and Numarray are now deprecated
  - Stable version : 1.12.1 / 18 March 2017

numpy.ndarray Object in “numpy” Module

- Numpy object type : numpy.ndarray

- 1D numpy.ndarray object

  Example: array([3, 6]) array([3.5 , 6.4, 7.2])
- 2D numpy.ndarray object

  Example: array([[1, 0, 2], [3, 5, 2]]) # Shape : 2 X 3 Matrix
- 3D numpy.ndarray object

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

- Axes : axis의 복수 (= dimensions in numpy.ndarray object)
- **Numpy function 안에서 optional parameter 로**
  - **axis = 0 : 각 column에 대해서**
  - **axis = 1 : 각 row에 대해서**

Table of Contents

ndarray Creation : np.array( ) [1/4]

>>> import numpy as np

An array can be created from a list:

```
>>> a = np.array([1, 4, 5, 8], float)
>>> a
array([ 1.,  4.,  5.,  8.])
>>> type(a)
<type 'numpy.ndarray'>
```

다른것!

```
>>> import array
>>> a = array.array("i", [3,6,9])
>>> a
array('i', [3, 6, 9])
>>> b = array.array("u", "boy")
>>> b
array('u', 'boy')
>>>
```

Arrays can be multidimensional. Unlike lists, different axes are accessed using commas inside bracket notation. Here is an example with a two-dimensional array (e.g., a matrix):

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> a[0,0]
1.0
>>> a[0,1]
2.0
```

참고사항

np.array( ) function

numpy.array

`numpy.` `array` (*object, dtype=None, copy=True, order='K',*

Create an array.

Parameters: **object** : *array_like*

An array, any object exposing the
whose __array__ method returns ar
sequence.

dtype : *data-type, optional*

The desired data-type for the array
will be determined as the minimur
objects in the sequence. This argu
'upcast' the array. For downcasting

copy : *bool, optional*

If true (default), then the object is
only be made if __array__ returns a
sequence, or if a copy is needed to
requirements (`dtype`, *order*, etc.).

```
import numpy as np
a = np.array([1,2,3])
b = np.array({1,2,3})
c = np.array("snu")
```

```
In [7]: a
Out[7]: array([1, 2, 3])

In [8]: b
Out[8]: array({1, 2, 3}, dtype=object)

In [9]: c
Out[9]: array('snu', dtype='<U3')
```

```
In [10]: a[1]
Out[10]: 2

In [11]: b[1]
Traceback (most recent call last):
IndexError: too many indices for array

In [12]: c[1]
Traceback (most recent call last):
IndexError: too many indices for array
```

Python List에서 Numpy ndarray를
만드는것으로 관심을 한정합시다!

ndarray Creation: np.array( ) [2/4]

>>> import numpy as np

- np.array(list): from a pythin list of numbers as argument

```
>>> a = np.array(1,2,3,4)      # WRONG
>>> a = np.array([1,2,3,4])    # RIGHT
```

```
In [1]:

   x = array([1,2,3])

   type(x)
   numpy.ndarray

   x.dtype
   dtype('int32')
```

```
In [2]:

   x = np.array([1.0, 2.0, 3.0])
   x.dtype

 dtype('float64')

In [3]:

   x = np.array([1, 2, 3.0])
   x.dtype

 dtype('float64')
```

np.array( ) 에서 dtype parameter 값을 안주면
Integer는 “int32”, Float 는 “float64’, String은 “가장 긴 element”

ndarray Creation: np.array( ) [3/4]

In [4]:

```
x = np.array([1, 2, 3], dtype='f')
x.dtype
```

```
dtype('float32')
```

In [5]:

```
x[0] + x[1]
```

```
3.0
```

np.array( ) 에서 dtype parameter 값을 안주면
Integer는 “int32”, Float 는 “float64’, String은 “가장 긴 element”

dtype = ‘f’ ➔ ‘float32’
dtype = ‘f8’ ➔ ‘float64’

dtype = ‘i’ ➔ ‘int32’
dtype = ‘i8’ ➔ ‘in64’

dtype = ‘U’ ➔ “가장 긴 element 글자갯구 이내“
dtype = ‘U4’ ➔ ‘<U4’ // 4 글자 이내

dtype 접두사	설명	사용 예
b	불리언	b (참 혹은 거짓)
i	정수	i8 (64비트)
u	부호 없는 정수	u8 (64비트)
f	부동소수점	f8 (64비트)
c	복소 부동소수점	c16 (128비트)
O	객체	0 (객체에 대한 포인터)
S	바이트 문자열	S24 (24 글자)
U	유니코드 문자열	U24 (24 유니코드 글자)

ndarray Creation: np.array( ) [4/4]

```
In [36]: y = array(['abcd', 'efghkt'])

In [37]: y[0]
Out[37]: 'abcd'

In [38]: y[1]
Out[38]: 'efghkt'

In [39]: y.dtype
Out[39]: dtype('<U6')
```

6 글자 이내

String으로 array를 만들때에 dtype을 안주면 “가장 긴 element의 글자갯수 이내”로 dtype이 정해진다.

```
In [6]:
x = np.array([1, 2, 3], dtype='U')
x.dtype

dtype('<U1')

In [7]:
x[0] + x[1]

'12'
```

1 글자 이내

```
In [50]: y = array([1, 2, 'abcd', 'efghkt'])

In [51]: y
Out[51]: array(['1', '2', 'abcd', 'efghkt'], dtype='<U11')
```

11 글자 이내

Number와 String이 섞인 상태에서 array를 만들때에 dtype은 <U11 으로 정해진다

Special Attribute: np.inf (infinity) & np.nan (not a number)

In [8]:

```
np.array([0, 1, -1, 0]) / np.array([1, 0, 0, 0])
```

```
array([  0.,  inf, -inf,  nan])
```

In [9]:

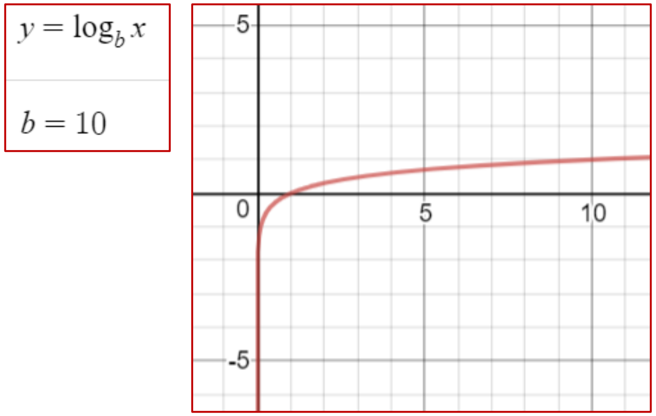
```
np.log(0)
```

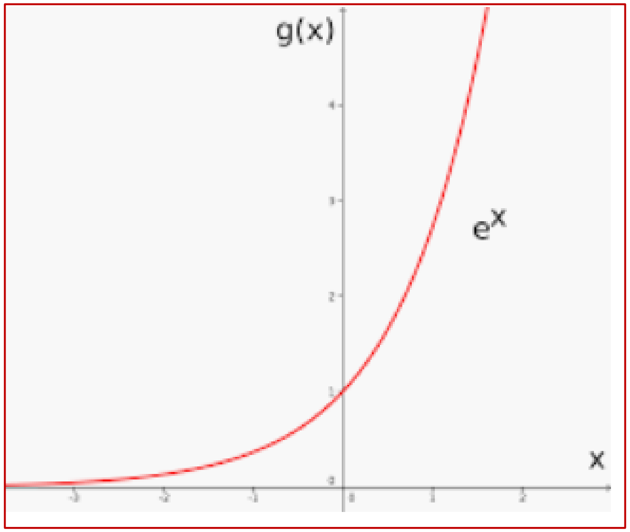
```
-inf
```

In [10]:

```
np.exp(-np.inf)
```

```
0.0
```





ndrray Creation : np.zeros( ) [1/2]

- np.zeros(array_size_tuple) :

  np.ndarray filled with 0

```
>>> a = np.zeros( (3,3) )
>>> a
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

In [11]:

```
a = np.zeros(5)
a
```

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

크기를 뜻하는 튜플을 입력하면 다차원 배열도 만들 수 있다.

In [12]:

```
b = np.zeros((2, 3))
b
```

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

`array` 명령과 마찬가지로 `dtype` 인수를 명시하면 해당 자료형 원소를 가진 배열을 만든다.

In [13]:

```
c = np.zeros((5, 2), dtype="i")
c
```

```
array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]], dtype=int32)
```

i8 ➔ 64bit 정수
i ➔ 32bit 정수

ndrray Creation : np.zeros( ) [2/2]

문자열 배열도 가능하지면 모든 원소의 문자열 크기가 같아야 한다. 만약 더 큰 크기의 문자열을 할당하면 잘릴 수 있다.

String으로 zero item를 만든다는것은 " " 을 만드는것으로 해석

U4 ➔ 4글자까지 들어가는 String

In [14]:

```
d = np.zeros(5, dtype="U4")
d
```

```
array(['', '', '', '', ''], dtype='<U4')
```

In [15]:

```
d[0] = "abc"
d[1] = "abcd"
d[2] = "ABCDE"
d
```

```
array(['abc', 'abcd', 'ABCD', '', ''], dtype='<U4')
```

ndrray Creation : np.ones( )   [1/2]

0이 아닌 1로 초기화된 배열을 생성하려면 `ones` 명령을 사용한다.

- np.ones(array_size_tuple) :   np.ndarray filled with 1

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

```
In [16]:

e = np.ones((2, 3, 4), dtype="i8")
e

array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]])
```

i8 ➔ 64bit 정수
i ➔ 32bit 정수

ndrray Creation : np.ones( )   [2/2]

In [12]:

```
b = np.zeros((2, 3))
b
```

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

만약 크기를 튜플로 명시하지 않고 다른 배열과 같은 크기의 배열을 생성하고 싶다면 `ones_like`, `zeros_like` 명령을 사용한다.

In [17]:

```
f = np.ones_like(b, dtype="f")
f
```

```
array([[1., 1., 1.],
       [1., 1., 1.]], dtype=float32)
```

ndrray Creation: np.full( ) & np.eye( )

>>> import numpy as np

- np.full(array_size_tuple , value) : np.ndarray filled with certain value

```
>>> a = np.full( (3,3), 7 )
>>> a
array([[7, 7, 7],
       [7, 7, 7],
       [7, 7, 7]])
```

- np.eye(array_size) : np.ndarray for Identity matrix (square matrix에만 해당)

```
>>> a = np.eye( 3 )
>>> a
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

ndarray Creation: empty( )

배열의 크기가 커지면 배열을 초기화하는데도 시간이 걸린다. 이 시간을 단축하려면 배열을 생성만 하고 특정한 값으로 초기화를 하지 않는 `empty` 명령을 사용할 수 있다. `empty` 명령으로 생성된 배열에는 기존에 메모리에 저장되어 있던 값이 있으므로 배열의 원소의 값을 미리 알 수 없다.

In [18]:

```python
g = np.empty((4, 3))
g
```

```
array([[6.94820328e-310, 4.67533915e-310, 5.28964691e+180],
       [6.01346953e-154, 4.81809028e+233, 7.86517465e+276],
       [6.01346953e-154, 2.58408173e+161, 2.46600381e-154],
       [2.47379808e-091, 4.47593816e-091, 6.01347002e-154]])
```

ndarray Creation: arange( )

- np.arange(start_number, end_number, interval) : from numbers in a range

`arange` 명령은 NumPy 버전의 `range` 명령이라고 볼 수 있다.

In [19]:

```
np.arange(10)  # 0 .. n-1
```

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

In [20]:

```
np.arange(3, 21, 2)  # 시작, 끝(포함하지 않음), 단계
```

```
array([ 3,  5,  7,  9, 11, 13, 15, 17, 19])
```

```
>>> np.arange( 0, 2, 0.3 )                 # it accepts float arguments
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

ndArray Creation: np.linspace( ) & np.logspace( )

`linspace` 명령이나 `logspace` 명령은 선형 구간 혹은 로그 구간을 지정한 구간의 수만큼 분할한다.

In [21]:

```
np.linspace(0, 100, 5)  # 시작, 끝(포함), 갯수
```

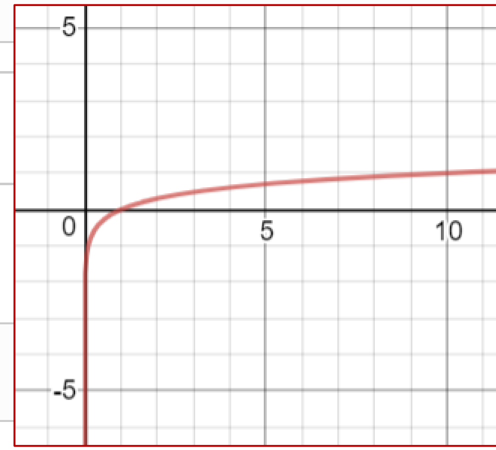
```
array([  0.,  25.,  50.,  75., 100.])
```

In [22]:

```
np.logspace(0.1, 1, 10)
```

$$y = \log_b x$$

$$b = 10$$




```
array([ 1.25892541,  1.58489319,  1.99526231,  2.51188643,  3.16227766,
        3.98107171,  5.01187234,  6.30957344,  7.94328235, 10.        ])
```

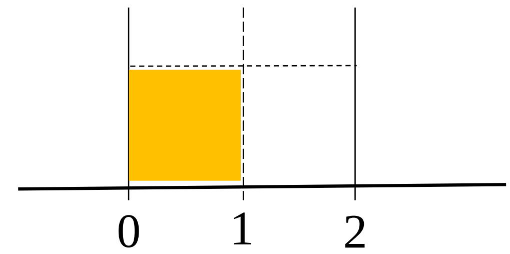
Y 값이 0.1 ~ 1 인 구간을 10개로
나눈 해당 X 값

np.random submodule: random( ), rand( ), uniform( ) [1/3]

>>> import numpy as np

- np.random.random(array_size_tuple) : np.ndarray filled with random values from a uniform distribution over [0, 1)

```
>>> a = np.random.random( (3,3) )
>>> a
array([[ 0.20194257,  0.63729801,  0.14885297],
       [ 0.93996771,  0.566249  ,  0.78957659],
       [ 0.72223359,  0.18619152,  0.90814515]])
```

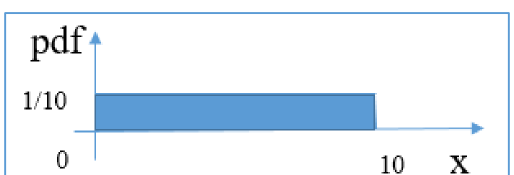


- np.random.rand(array_size) : same as the above, but parameter value is not tuple

```
>>> a = np.random.rand( 3,3 )
>>> a
array([[ 0.221797  ,  0.83165448,  0.06600647],
       [ 0.89332303,  0.30168798,  0.59380902],
       [ 0.81269937,  0.23821311,  0.4093413 ]])
```

- np.random.uniform (a, b, array_size) : random values from a uniform distribution over [a, b)

```
v = np.random.uniform(0., 10., 100)
```



v →

```
array([9.59907953, 0.02422541, 2.82054237, 8.84408103, 4.52971878,
       7.21823601, 4.30180311, 5.79395615, 5.56230367, 4.19906282,
       0.4331126 , 0.42914631, 6.52259286, 0.55867994, 8.94271047,
       … …)
```

np.random submodule: randint( ), random_integers() [2/3]

`randint`(low[, high, size, dtype])	Return random integers from *low* (inclusive) to *high* (exclusive).
`random_integers`(low[, high, size])	Random integers of type np.int between *low* and *high*, inclusive.

np.random. randint( )

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1],
       [3, 2, 2, 0]])
```

np.random. random_integers( )

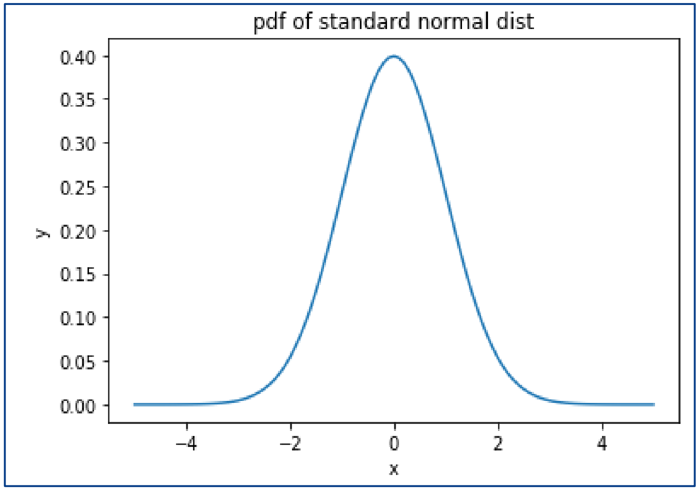
```
>>> np.random.random_integers(5)
4
>>> type(np.random.random_integers(5))
<type 'int'>
>>> np.random.random_integers(5, size=(3,2))
array([[5, 4],
       [3, 3],
       [4, 5]])
```

np.random submodule: randn( ), normal( ) [3/3]

np.random.randn( ) :

Return a sample (or samples) from the "standard normal" distribution

N(0,1)



For random samples from $N(\mu, \sigma^2)$, use: `sigma * np.random.randn(...) + mu`

```
>>> np.random.randn()
2.1923875335537315 #random
```

Single value 생성

Two-by-four array of samples from N(3, 6.25):

mu = 3
sigma = 2.5

2X4 matrix 생성

```
>>> 2.5 * np.random.randn(2, 4) + 3
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   #random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  #random
```

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

# 5x3
np.random.normal(mean, std, (5,3))
```

Table of Contents

Numpy ndarray Slicing

>>> import numpy as np

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a[1,:]
array([ 4.,  5.,  6.])
>>> a[:,2]
array([ 3.,  6.])
>>> a[-1:,-2:]
array([[ 5.,  6.]])
```

| 1, 2, 3 |
| 4, 5, 6 |

```
lst = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
arr = np.array(lst)

# 슬라이스
a = arr[0:2, 0:2]
print(a)

a = arr[1:, 1:]
print(a)
```

```
# 출력:
# [[1 2]
#  [4 5]]

# 출력:
# [[5 6]
#  [8 9]]
```

Matrix a 는 그대로 있다!

Numpy ndarray Transposing

2차원 배열의 전치(transpose) 연산은 행과 열을 바꾸는 작업이다.

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
>>> a.transpose()
array([[ 0.,  3.],
       [ 1.,  4.],
       [ 2.,  5.]])
```

3 X 2 Matrix로 transpose

```
In [23]:

A = np.array([[1, 2, 3], [4, 5, 6]])
A

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

In [24]:

A.T

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

attribute

Matrix a & A 는 그대로 있다!

Numpy ndarray Flattening

Flattening 2D np.ndarray into an 1D np.ndarray

```
In [61]: a = array( [[1, 2, 3], [4, 5, 6]])

In [62]: a
Out[62]:
array([[1, 2, 3],
       [4, 5, 6]])

In [63]: a.flatten( )
Out[63]: array([1, 2, 3, 4, 5, 6])

In [64]: a
Out[64]:
array([[1, 2, 3],
       [4, 5, 6]])

In [65]: a.ravel( )
Out[65]: array([1, 2, 3, 4, 5, 6])

In [66]: a
Out[66]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Matrix a 는 그대로 있다!

Numpy ndarray Concatenating

>>> import numpy as np

```
>>> a = np.array([1,2], float)
>>> b = np.array([3,4,5,6], float)
>>> c = np.array([7,8,9], float)
>>> np.concatenate((a, b, c))
array([1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

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

a 의 column 을 유지하면서 concatenation

a 의 row 을 유지하면서 concatenation

Numpy ndarray Reshaping [1/2]

일단 만들어진 배열의 내부 데이터는 보존한 채로 형태만 바꾸려면 `reshape` 명령이나 메서드를 사용한다. 예를 들어 12개의 원소를 가진 1차원 행렬은 3x4 형태의 2차원 행렬로 만들 수 있다.

In [25]:

```
a = np.arange(12)
a
```

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

In [26]:

```
b = a.reshape(3, 4)
b
```

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

```
In [66]: a
Out[66]:
array([[1, 2, 3],
       [4, 5, 6]])

In [67]: a.reshape(3,2)
Out[67]:
array([[1, 2],
       [3, 4],
       [5, 6]])

In [68]: a
Out[68]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Numpy ndarray Reshaping [2/2]

In [25]:

```
a = np.arange(12)
a
```

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

가장 내부에 있는
Matrix의 shape

In [27]:

```
a.reshape(3, -1)
```

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

In [28]:

```
a.reshape(2, 2, -1)
```

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

       [[ 6,  7,  8],
        [ 9, 10, 11]]])
```

In [29]:

```
a.reshape(2, -1, 2)
```

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

       [[ 6,  7],
        [ 8,  9],
        [10, 11]]])
```

-1을 쓰면, 3 & a의 shape에서
값을 추론이 되는값으로 대치

Reshaping ndarray with 'newaxis'

In [32]:

```
x = np.arange(5)
x
```

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

In [33]:

```
x.reshape(1, 5)
```

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

In [34]:

```
x.reshape(5, 1)
```

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

In [35]:

```
x[:, np.newaxis]
```

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

np.newaxis

새로운 axis를 추가

hstack( ) of ndarray

In [36]:

```
a1 = np.ones((2, 3))
a1
```

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

In [37]:

```
a2 = np.zeros((2, 2))
a2
```

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

In [38]:

```
np.hstack([a1, a2])
```

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

Horizontal Stacking

결과물의 dimension 그대로
결과물의 column 수는 증가
결과물의 row 수는 그대로

vstack( ) of ndarray

In [39]:

```
b1 = np.ones((2, 3))
b1
```

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

In [40]:

```
b2 = np.zeros((3, 3))
b2
```

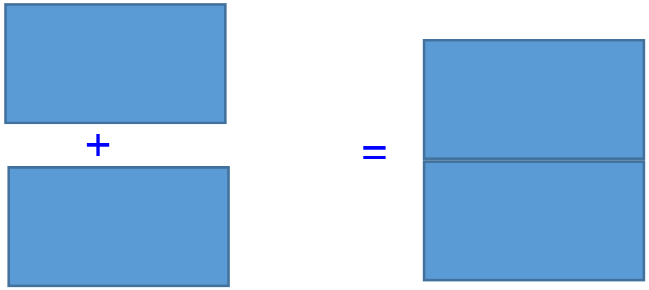
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

In [41]:

```
np.vstack([b1, b2])
```

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

Vertical Stacking



결과물의 dimension 그대로
결과물의 column 수는 그대로
결과물의 row 수는 증가

dstack( ) of ndarray

In [42]:

```
c1 = np.ones((3, 4))
c1
```

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

In [43]:

```
c2 = np.zeros((3, 4))
c2
```

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

In [44]:

```
np.dstack([c1, c2])
```

```
array([[[1., 0.],
        [1., 0.],
        [1., 0.],
        [1., 0.]],

       [[1., 0.],
        [1., 0.],
        [1., 0.],
        [1., 0.]],

       [[1., 0.],
        [1., 0.],
        [1., 0.],
        [1., 0.]]])
```

In [45]:

```
(np.dstack([c1, c2])).shape
```

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

Dimensional Stacking
Depth Stacking

결과물의 dimension이 증가
결과물의 column 수는 그대로
결과물의 row 수는 그대로

C1의 element와 C2의 corresponding element 를 list로 만들고 3 X 4를 유지

제3의 축 즉, 행이나 열이 아닌 깊이(depth) 방향으로 배열을 합친다. 가장 안쪽의 원소의 차원이 증가한다. 즉 가장 내부의 숫자 원소가 배열이 된다

stack( ) of ndarray [1/2]

In [42]:

```
c1 = np.ones((3, 4))
c1
```

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

In [43]:

```
c2 = np.zeros((3, 4))
c2
```

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

In [46]:

```
c = np.stack([c1, c2])
c
```

Vertical Stacking하고 Dimension을 증가시킨것

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

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

In [47]:

```
c.shape
```

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

결과물의 dimension이 확장

dstack의 기능을 확장한 것으로 dstack처럼 마지막 차원으로 연결하는 것이 아니라 사용자가 지정한 차원(축으로) 배열을 연결한다

stack( ) of ndarray [2/2]

In [42]:

```
c1 = np.ones((3, 4))
c1
```

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

In [43]:

```
c2 = np.zeros((3, 4))
c2
```

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

In [48]:

```
c = np.stack([c1, c2], axis=1)
c
```

```
array([[[1., 1., 1., 1.],
        [0., 0., 0., 0.]],

       [[1., 1., 1., 1.],
        [0., 0., 0., 0.]],

       [[1., 1., 1., 1.],
        [0., 0., 0., 0.]]])
```

In [49]:

```
c.shape
```

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

c1의 모든 row에 대해서 c2 row를 확장 시키고 Dimension을 증가시킨것

C1의 row와 C2의 corresponding row 를 2D list로 만들고, 만들어진 2D list 들을 list 로 구성

Special Indexer Method: r_ vs c_

r_ & c_ 특수 메서드를 **인덱서(indexer)**라고 한다
- 메서드임에도 불구하고 소괄호(parenthesis, ( ))를 사용하지 않고 인덱싱과 같이 대괄호(bracket, [ ])를 사용한다.

r_ 메서드는 hstack( ) 혹은 ravel( ) 과 유사하게 배열을 좌우로 연결한다.

```
In [50]:
np.r_[np.array([1, 2, 3]), np.array([4, 5, 6])]
```

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

c_ 메서드는 배열의 차원을 증가시킨 후 좌우로 연결한다.
만약 1차원 배열을 연결하면 2차원 배열이 된다

```
In [51]:
np.c_[np.array([1, 2, 3]), np.array([4, 5, 6])]
```

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

tile( ) of ndarray

In [52]:

```
a = np.array([[0, 1, 2], [3, 4, 5]])
np.tile(a, 2)
```

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

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

a 를 tile로 만들어서 2번반복

결과물의 dimension은 그대로

In [53]:

```
np.tile(a, (3, 2))
```

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

a 를 tile로 만들어서 3 X 2 방식으로 반복

np.meshgrid( ) [1/2]

In [54]:

```
x = np.arange(3)
x
```

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

In [55]:

```
y = np.arange(5)
y
```

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

In [56]:

```
X, Y = np.meshgrid(x, y)
```

In [57]:

```
X
```

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

In [58]:

```
Y
```

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

In [59]:

```
[list(zip(x, y)) for x, y in zip(X, Y)]
```

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

np.meshgrid( ) [2/2]

In [54]:

```
x = np.arange(3)
x
```

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

In [55]:

```
y = np.arange(5)
y
```

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

In [56]:

```
X, Y = np.meshgrid(x, y)
```

In [60]:

```
plt.title("np.meshgrid로 만든 Grid Points")
plt.scatter(X, Y, linewidths=10)
plt.show()
```

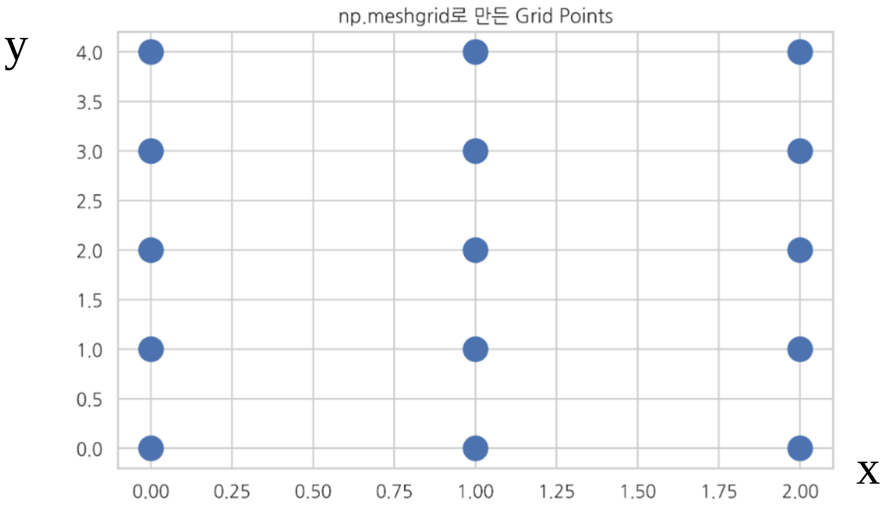


Table of Contents

Numpy ndarray Mathematics [1/5]

>>> import numpy as np

```
>>> a = np.array([1,2,3], float)
>>> b = np.array([5,2,6], float)
>>> a + b
array([6., 4., 9.])
>>> a – b
array([-4., 0., -3.])
>>> a * b
array([5., 4., 18.])
>>> b / a
array([5., 1., 2.])
>>> a % b
array([1., 0., 3.])
>>> b**a
array([5., 4., 216.])
```

a or b 에 constant를 넣어도 다 OK!

→ **If 2 operands are np.array objects in np.array mathematics, shape of both operands must be same!**
If one operand is constant, that is OK!

Numpy ndarray Mathematics [2/5]

>>> import numpy as np

```
>>> a = np.array( [1,2,3], float )
>>> b = np.array( [4,5,6], float )
>>> a + b
array([ 5.,  7.,  9.])
>>> b2 = np.array( [4,5,6,7], float )
```

```
>>> a+b2
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    a+b2
ValueError: operands could not be broadcast together with shapes (3,) (4,)
```

```
>>> b2 / a
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    b2 / a
ValueError: operands could not be broadcast together with shapes (4,) (3,)
>>> b2 ** a
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    b2 ** a
ValueError: operands could not be broadcast together with shapes (4,) (3,)
```

→ **If 2 operands are np.array objects in np.array mathemetics,**
shape of both operands must be same!
If one operand is constant, that is OK!

Numpy ndarray Mathematics [3/5]

>>> import numpy as np

- * operation in Python and Numpy
  - Python_List * number → Repetition of the whole list

```
>>> a = [1,2,3,4]
>>> a*2
[1, 2, 3, 4, 1, 2, 3, 4]
```

  - numpy.ndarray * number → multiply number to every element in numpy.ndarray object

```
import numpy as np

a = [1, 2, 3, 4]
npa = np.array( a )
```

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

In [4]: npa*2
Out[4]: array([2, 4, 6, 8])
```

Numpy ndarray Mathematics [4/5]

>>> import numpy as np

- \+ Operation in 1D Python lists and 1D np.ndarrays
  - List + List → Concatenating 2 lists into 1 list

```
>>> a = [1,2,3,4]
>>> b = [1,2,3,4]
>>> a+b
[1, 2, 3, 4, 1, 2, 3, 4]
```

  - np.ndarray + np.ndarray → Pairwide plus operation between 2 np.ndarray

```
>>> npa = np.array( a )
>>> npb = np.array( b )
>>> npa
array([1, 2, 3, 4])
>>> npb
array([1, 2, 3, 4])
>>> npa + npb
array([2, 4, 6, 8])
```

Numpy ndarray Mathematics [5/5]

>>> import numpy as np

- "+" Operation in 2D Python lists and 2D np.ndarrays
  - List + List → Concatenating 2D lists into 1 list

```
>>> a
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> b1
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> a + b1
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

  - numpy.array + numpy.array → Pairwise plus operation between 2 np.arrays

```
>>> anp= np.array( a )
>>> b1np = np.array( b1 )
>>> anp
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> b1np
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
>>> anp + b1np
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18]])
```

Numpy ndarray Iteration [1/2]

>>> import numpy as np

```
>>> a = np.array([1, 4, 5], int)
>>> for x in a:
...   print(x)
... <hit return>
1
4
5
```

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> for x in a:
...   print(x)
... <hit return>
[ 1.  2.]
[ 3.  4.]
[ 5.  6.]
```

Numpy ndarray Iteration [2/2]

>>> import numpy as np

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

       [[ 7,  8,  9],
        [10, 11, 12]],

       [[13, 14, 15],
        [16, 17, 18]]])
```

```
>>> for x in a :
        print (x)
        print ("-----")

[[1 2 3]
 [4 5 6]]
-----
[[ 7  8  9]
 [10 11 12]]
-----
[[13 14 15]
 [16 17 18]]
-----
```

→ 1 dimension iteration

```
>>> for x in a :
        for y in x :
            print ( y )
            print ("-----")

[1 2 3]
-----
[4 5 6]
-----
[7 8 9]
-----
[10 11 12]
-----
[13 14 15]
-----
[16 17 18]
-----
>>>
```

→ 2 dimension iteration

→ For each 'For', 1 Dimension Iteration is done

Table of Contents

Numpy ndarray Statistics [1/5]

>>> import numpy as np

np.array data type에 있는 통계함수 ➔ number or np.array를 return

```
>>> a = np.array([2, 4, 3], float)
>>> a.sum()
9.0
>>> a.prod()
24.0
```

product

```
>>> a = np.array([2, 1, 9], float)
>>> a.mean()
4.0
>>> a.var()
12.666666666666666
>>> a.std()
3.5590260840104371
```

variance

Standard deviation

```
>>> a = np.array([1, 4, 3, 8, 9, 2, 3], float)
>>> np.median(a)
3.0
```

Numpy ndarray Statistics [2/5]

>>> import numpy as np

```
>>> x = np.array( [ [1,2], [3,4] ] )
>>> print ( np.sum(x) )
10
>>> print ( np.sum(x, axis=0 ) )
[4 6]
>>> print ( np.sum(x, axis=1 ) )
[3 7]
```

x

$$\begin{vmatrix} 1, & 2 \\ 3, & 4 \end{vmatrix}$$

```
In [15]: x.sum()
Out[15]: 10

In [16]: x.sum(axis=0)
Out[16]: array([4, 6])

In [17]: x.sum(axis=1)
Out[17]: array([3, 7])
```

numpy module에 있는 통계함수
➔ number or list를 return

np.array data type에 있는 통계함수 ➔ number or np.array를 return

Optional parameter axis
axis = 0 : 각 column에 대한 합계를 계산
axis = 1 : 각 row에 대한 합계를 계산

Numpy ndarray Statistics [3/5]

>>> import numpy as np

- Numpy statistics is much easier than nested list statistics
  - sum( ) of 3D np.array data type

```
>>> npa
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> npa.sum()
45
```

  - nested_sum( ) for Python nested list

```
def nested_sum(L):
    total = 0
    for i in L:
        if isinstance(i, list):
            total += nested_sum(i)
        else:
            total += i
    return total
```

```
>>> a = [[1,2,3],[4,5,6],[7,8,9]]
>>> nested_sum( a )
45
```

Correlation Coefficient (상관계수), Covariance (공분산)

2개의 변수간의 분산 상황이 어느 정도 직선적인지를 나타내는 지표를 말한다. 시각적으로는 2 개의 변수의 조합을 2차원의 좌표축 상에 나타낸 경우 어느 정도 깨끗한 직선을 그리는가를 나타내고 있다고도 할 수 있다. 구체적으로 2개의 변수 x와 y가

$(x_1, \ldots, x_i, \ldots, x_n)(y_1, \ldots, y_i, \ldots, y_n)$

으로 주어졌다고 하자. 상관계수 r은 다음의 수식으로 나타낼 수 있다.

$$r = \frac{\Sigma(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\Sigma(x_i - \bar{x})^2}\sqrt{\Sigma(y_i - \bar{y})^2}}$$

두 확률변수 X, Y의 기댓값을 각각 $\mu_X = \mathrm{E}(X)$, $\mu_Y = \mathrm{E}(Y)$라고 하자. **공분산** $\mathrm{Cov}(X, Y)$는 다음과 같이 정의한다.

$$\mathrm{Cov}(X, Y) = \mathrm{E}[(X - \mu_X)(Y - \mu_Y)]$$

오른쪽 식을 기댓값의 성질을 이용하여 정리하면 공분산은 다음과 같이 구할 수 있다.

$$\mathrm{Cov}(X, Y) = \mathrm{E}(XY) - \mathrm{E}(X)\mathrm{E}(Y)$$

Covariance (공분산) Example

동아리 회원 10명의 키와 몸무게가 다음과 같다고 하자.

	수현	수지	효주	세영	진구	호준	서진	지우	재석	준하
키 (cm)	177	167	160	162	174	180	176	158	172	184
몸무게 (kg)	72	47	52	57	63	75	74	50	68	86

X를 회원의 키, Y를 몸무게라고 하면

$$\begin{aligned}\mu_X &= \frac{1}{10}(177+167+160+162+174+180+176+158+172+184) = 171 \\ \mu_Y &= \frac{1}{10}(72+47+52+57+63+75+74+50+68+86) = 64.4\end{aligned}$$

이다. $(X-\mu_X)$와 $(Y-\mu_Y)$를 구하여 곱을 구하면 다음 표와 같다.

	수현	수지	효주	세영	진구	호준	서진	지우	재석
$X-\mu_X$	6	-4	-11	-9	3	9	5	-13	1
$Y-\mu_Y$	7.6	-17.4	-12.4	-7.4	-1.4	10.6	9.6	-14.4	3.6
$(X-\mu_X)(Y-\mu_Y)$	45.6	69.6	136.4	66.6	-4.2	95.4	48	187.2	3.6

따라서 키와 몸무게의 공분산은 다음과 같다.

$$\begin{aligned}\mathrm{Cov}(X,Y) &= \mathrm{E}[(X-\mu_X)(Y-\mu_Y)] \\ &= \frac{1}{10}(45.6+69.6+136.4+66.6-4.2+95.4+48+187.2+3.6+280.8) \\ &= 92.9\,(\mathrm{cm}\cdot\mathrm{kg})\end{aligned}$$

Numpy ndarray Statistics [4/5]

>>> import numpy as np

The correlation coefficient for multiple variables observed at multiple instances can be found for arrays of the form [[x1, x2, …], [y1, y2, …], [z1, z2, …], …] where x, y, z are different observables and the numbers indicate the observation times:

```
>>> a = np.array([[1, 2, 1, 3], [5, 3, 1, 8]], float)
>>> c = np.corrcoef(a)
>>> c
array([[ 1.        ,  0.72870505],
       [ 0.72870505,  1.        ]])
```

Correlation Coefficient (상관계수)

Here the return array `c[i,j]` gives the correlation coefficient for the ith and jth observables. Similarly, the covariance for data can be found:

```
>>> np.cov(a)
array([[ 0.91666667,  2.08333333],
       [ 2.08333333,  8.91666667]])
```

Covariance (공분산)

Numpy ndarray Statistics [5/5]

>>> import numpy as np

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

       [[ 7,  8,  9],
        [10, 11, 12]],

       [[13, 14, 15],
        [16, 17, 18]]])
>>> np.corrcoef(a)
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    np.corrcoef(a)
  File "C:₩Python35₩lib₩site-packages₩numpy₩lib₩function_base.py", line 3154, in
 corrcoef
    c = cov(x, y, rowvar)
  File "C:₩Python35₩lib₩site-packages₩numpy₩lib₩function_base.py", line 3004, in
 cov
    raise ValueError("m has more than 2 dimensions")
ValueError: m has more than 2 dimensions
```

[[1,2,3], [4,5,6]]
[[7,8,9]. [10,11,12]]
[[13,14,15], [16,17,18]]

```
>>> a2 = np.array( [ [1,2,3,4,5], [6,7,8,9,10], [11,12,13,14,15] ] )
>>> np.corrcoef( a2 )
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

| 1, 2, 3, 4, 5 |
| 6, 7, 8, 9, 10 |
| 11, 12, 13, 14, 15 |

→ **Array should be only 2 dimension for np.corrcoef (arr) & np.cov(arr)**

Statistics in Various Python Places [1/6]

Python Standard Library: Statistics

- Averages and measures of central location

These functions calculate an average or typical value from a population or sample.

`mean()`	Arithmetic mean (“average”) of data.
`harmonic_mean()`	Harmonic mean of data.
`median()`	Median (middle value) of data.
`median_low()`	Low median of data.
`median_high()`	High median of data.
`median_grouped()`	Median, or 50th percentile, of grouped data.
`mode()`	Mode (most common value) of discrete data.

- Measures of spread

These functions calculate a measure of how much the population or sample tends to deviate from the typical or average values.

`pstdev()`	Population standard deviation of data.
`pvariance()`	Population variance of data.
`stdev()`	Sample standard deviation of data.
`variance()`	Sample variance of data.

Statistics in Various Python Places [2/6]

Statistics Functions in Numpy (A)

Order statistics¶

`amin`(a[, axis, out, keepdims])	Return the minimum of an array or minimum along an axis.
`amax`(a[, axis, out, keepdims])	Return the maximum of an array or maximum along an axis.
`nanmin`(a[, axis, out, keepdims])	Return minimum of an array or minimum along an axis, ignoring any NaNs.
`nanmax`(a[, axis, out, keepdims])	Return the maximum of an array or maximum along an axis, ignoring any NaNs.
`ptp`(a[, axis, out])	Range of values (maximum - minimum) along an axis.
`percentile`(a, q[, axis, out, ...])	Compute the qth percentile of the data along the specified axis.
`nanpercentile`(a, q[, axis, out, ...])	Compute the qth percentile of the data along the specified axis, while ignoring nan values.

Statistics in Various Python Places [3/6]

Statistics Functions in Numpy (B)

Averages and variances

`median`(a[, axis, out, overwrite_input, keepdims])	Compute the median along the specified axis.
`average`(a[, axis, weights, returned])	Compute the weighted average along the specified axis.
`mean`(a[, axis, dtype, out, keepdims])	Compute the arithmetic mean along the specified axis.
`std`(a[, axis, dtype, out, ddof, keepdims])	Compute the standard deviation along the specified axis.
`var`(a[, axis, dtype, out, ddof, keepdims])	Compute the variance along the specified axis.
`nanmedian`(a[, axis, out, overwrite_input, ...])	Compute the median along the specified axis, while ignoring NaNs.
`nanmean`(a[, axis, dtype, out, keepdims])	Compute the arithmetic mean along the specified axis, ignoring NaNs.
`nanstd`(a[, axis, dtype, out, ddof, keepdims])	Compute the standard deviation along the specified axis, while ignoring NaNs.
`nanvar`(a[, axis, dtype, out, ddof, keepdims])	Compute the variance along the specified axis, while ignoring NaNs.

Statistics in Various Python Places [4/6]

Statistics Functions in Numpy (C)

Correlating

`corrcoef`(x[, y, rowvar, bias, ddof])	Return Pearson product-moment correlation coefficients.
`correlate`(a, v[, mode])	Cross-correlation of two 1-dimensional sequences.
`cov`(m[, y, rowvar, bias, ddof, fweights, ...])	Estimate a covariance matrix, given data and weights.

Histograms

`histogram`(a[, bins, range, normed, weights, ...])	Compute the histogram of a set of data.
`histogram2d`(x, y[, bins, range, normed, weights])	Compute the bi-dimensional histogram of two data samples.
`histogramdd`(sample[, bins, range, normed, ...])	Compute the multidimensional histogram of some data.
`bincount`(x[, weights, minlength])	Count number of occurrences of each value in array of non-negative ints.
`digitize`(x, bins[, right])	Return the indices of the bins to which each value in input array belongs.

Statistics in Various Python Places [5/6]

Statistics in Pandas

Function	Description
count	Number of non-NA observations
sum	Sum of values
mean	Mean of values
mad	Mean absolute deviation
median	Arithmetic median of values
min	Minimum
max	Maximum
mode	Mode
abs	Absolute Value
prod	Product of values
std	Bessel-corrected sample standard deviation
var	Unbiased variance
sem	Standard error of the mean
skew	Sample skewness (3rd moment)
kurt	Sample kurtosis (4th moment)
quantile	Sample quantile (value at %)
cumsum	Cumulative sum
cumprod	Cumulative product
cummax	Cumulative maximum
cummin	Cumulative minimum

Statistics in Various Python Places [6/6]

Statistics in Scipy

scipy.stats submodule

- supports various distribution objects
- contains various statistical hypothesis test functions

- Statistical functions (`scipy.stats`)
  - Continuous distributions
  - Multivariate distributions
  - Discrete distributions
  - Statistical functions
  - Circular statistical functions
  - Contingency table functions
  - Plot-tests
  - Masked statistics functions
  - Univariate and multivariate kernel density estimation (`scipy.stats.kde`)

Table of Contents

Structure of Numpy Module & Submodules

- Core
  - Array Creation
  - Array Manipulation
  - Binary Operations
  - String Operation
  - Data Type Routines
  - ……
- Submodules
  - numpy.rec: Creating record arrays
  - numpy.char: Creating character arrays
  - numpy.ctypeslib: C-types Foreign Function Interface
  - numpy.dual: Optionally Scipy-accelrated routines
  - numpy.emath: Mathematical functions with automatic domain
  - numpy.fft: Discrete Fourier Transfom
  - numpy.linalg: Linear Algebra
  - numpy.matlib: Matrix Library
  - numpy.random: Random Sampling
  - numpy.testing: Test Support

Linear Algebra (numpy.linalg submodule) [1/3]

linalg. 가 prefix로 없는 function들은
개념적으로 Linear Algebra function 들이며,
위치는 numpy에 직접소속

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(a, b[, out])	Matrix product of two arrays.
tensordot(a, b[, axes])	Compute tensor dot product along specified axes for arrays >= 1-D.
einsum(subscripts, *operands[, out, dtype, ...])	Evaluates the Einstein summation convention on the operands.
einsum_path(subscripts, *operands[, optimize])	Evaluates the lowest cost contraction order for an einsum expression by considering the creation of intermediate arrays.
linalg.matrix_power(a, n)	Raise a square matrix to the (integer) power *n*.
kron(a, b)	Kronecker product of two arrays.

Linear Algebra (numpy.linalg submodule) [2/ 3]

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 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.

Linear Algebra (numpy.linalg submodule) [3/ 3]

Norms and other numbers

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, hermitian])	Return matrix rank of array using SVD method
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

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.

Product Operations in Vector and Matrix

- For Vector
  - Inner Product (벡터내적) :supported by np.dot() or np.inner( )
  - Outer Product (벡터외적) :supported by np.outer( )
  - Cross Product (벡터곱) :supported by np.cross( )

- For Matrix
  - Matrix Multiplication (행렬곱) :supported by np.dot( ) or np.matmult( )
  - Inner Product (행렬내적) :supported by np.inner( )

Vector Inner Product

벡터내적

$$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

The **inner product** of two vectors in matrix form

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &= \mathbf{a}^{\mathrm{T}} \mathbf{b} \\ &= \begin{pmatrix} a_1 & a_2 & \cdots & a_n \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \\ &= a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \\ &= \sum_{i=1}^{n} a_i b_i , \end{aligned}$$

where $\mathbf{a}^{\mathrm{T}}$ denotes the transpose of $\mathbf{a}$.

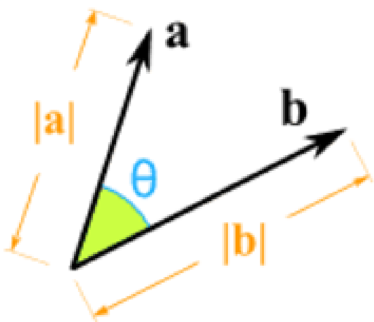
$$(a,b,c) \bullet \begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix} = a + 4b + 7c$$

symbol이 없어도 OK

Vector 2개로 숫자 1개 생성

Application of Vector Inner Product

2개의 Vector a, b의 사이에 있는 각도를 계산

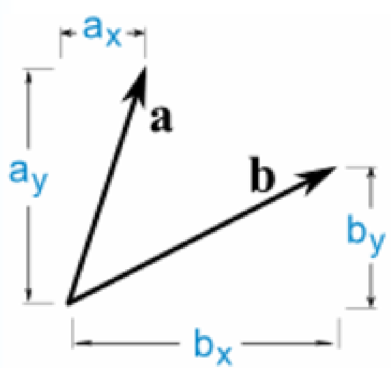


$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| \times |\mathbf{b}| \times \cos(\theta)$$

Where:
$|\mathbf{a}|$ is the magnitude (length) of vector **a**
$|\mathbf{b}|$ is the magnitude (length) of vector **b**
θ is the angle between **a** and **b**

$$\cos\theta = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \, \|\mathbf{B}\|}$$

OR we can calculate it this way:



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

So we multiply the x's, multiply the y's, then add.

numpy.dot( ) and numpy inner( ) for 1D ndarrays

- If a and b are 1D arrays, it is inner product of vectors (without complex conjugation)

```
>>> a = np.array([1, 2, 3], float)
>>> b = np.array([0, 1, 1], float)
>>> np.dot(a, b)
5.0
```

- Ordinary inner product of vectors for 1D ndarrays (without complex conjugation)

```
np.inner(a, b) = sum(a[:] * b[:])
```

```
>>> a = np.array([1, 2])
>>> b = np.array([0, 3])
>>> np.inner(a, b)
6
```

- a or b may be scalars

```
np.inner(a, b) = a * b
```

```
>>> a = np.array([1, 2])
>>> np.inner(a, 3)
array([3, 6])
```

Vector Outer Product

벡터외적

$$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

The **outer product** (also known as the **dyadic product** or **tensor product**) of two vectors in matrix form

$$\mathbf{a} \otimes \mathbf{b} = \mathbf{a}\mathbf{b}^{\mathrm{T}} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \begin{pmatrix} b_1 & b_2 & \cdots & b_n \end{pmatrix} = \begin{pmatrix} a_1 b_1 & a_1 b_2 & \cdots & a_1 b_n \\ a_2 b_1 & a_2 b_2 & \cdots & a_2 b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_n b_1 & a_n b_2 & \cdots & a_n b_n \end{pmatrix}.$$

Vector 2개로 Matrix 생성

$$\begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix} \otimes \begin{pmatrix} a & d \end{pmatrix} = \begin{pmatrix} 1a & 1d \\ 4a & 4d \\ 7a & 7d \end{pmatrix}$$

Applications of Vector Outer Product

Matrix A 와 B를 multiply할때
Vector Outer Product를 사용가능

$$\mathbf{AB} = \begin{pmatrix} \bar{\mathbf{a}}_1 & \bar{\mathbf{a}}_2 & \cdots & \bar{\mathbf{a}}_m \end{pmatrix} \begin{pmatrix} \bar{\mathbf{b}}_1 \\ \bar{\mathbf{b}}_2 \\ \vdots \\ \bar{\mathbf{b}}_m \end{pmatrix}$$
$$= \bar{\mathbf{a}}_1 \otimes \bar{\mathbf{b}}_1 + \bar{\mathbf{a}}_2 \otimes \bar{\mathbf{b}}_2 + \cdots + \bar{\mathbf{a}}_m \otimes \bar{\mathbf{b}}_m$$
$$= \underline{\sum_{i=1}^{m} \bar{\mathbf{a}}_i \otimes \bar{\mathbf{b}}_i}$$

where this time

$$\bar{\mathbf{a}}_i = \begin{pmatrix} A_{1i} \\ A_{2i} \\ \vdots \\ A_{ni} \end{pmatrix}, \quad \bar{\mathbf{b}}_i = \begin{pmatrix} B_{i1} & B_{i2} & \cdots & B_{ip} \end{pmatrix}.$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} a & d \\ b & e \\ c & f \end{pmatrix} = \underline{\begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix} \otimes \begin{pmatrix} a & d \end{pmatrix}} + \begin{pmatrix} 2 \\ 5 \\ 8 \end{pmatrix} \otimes \begin{pmatrix} b & e \end{pmatrix} + \begin{pmatrix} 3 \\ 6 \\ 9 \end{pmatrix} \otimes \begin{pmatrix} c & f \end{pmatrix}$$
$$= \begin{pmatrix} 1a & 1d \\ 4a & 4d \\ 7a & 7d \end{pmatrix} + \begin{pmatrix} 2b & 2e \\ 5b & 5e \\ 8b & 8e \end{pmatrix} + \begin{pmatrix} 3c & 3f \\ 6c & 6f \\ 9c & 9f \end{pmatrix}$$
$$= \begin{pmatrix} 1a + 2b + 3c & 1d + 2e + 3f \\ 4a + 5b + 6c & 4d + 5e + 6f \\ 7a + 8b + 9c & 7d + 8e + 9f \end{pmatrix}.$$

- Digital Image Processing
  - Deep Neural Net의 CNN 처리
- Covariance Matrix 구할때도 사용

Vector Cross Product

벡터곱

The cross product is defined by the formula[3][4]

$$\underline{\mathbf{a} \times \mathbf{b}} = \|\mathbf{a}\| \, \|\mathbf{b}\| \sin(\theta) \, \mathbf{n}$$

where θ is the angle between **a** and **b** in the plane containing them (

Vector 2개로 vector 1개 생성

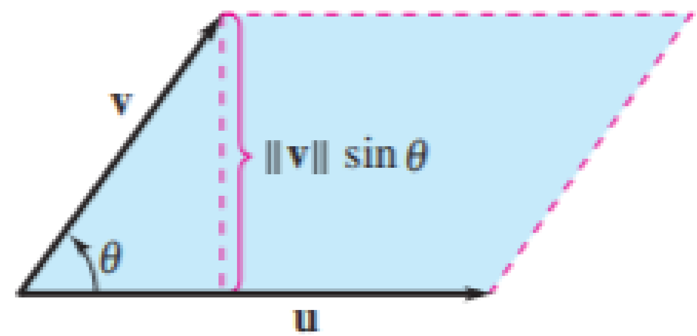
$$\mathbf{u} = u_1\mathbf{i} + u_2\mathbf{j} + u_3\mathbf{k}$$
$$\mathbf{v} = v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}$$

The cross product can also be expressed as the formal[note 1]

$$\mathbf{u} \times \mathbf{v} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \end{vmatrix}$$

$$\mathbf{u} \times \mathbf{v} = \begin{vmatrix} u_2 & u_3 \\ v_2 & v_3 \end{vmatrix} \mathbf{i} - \begin{vmatrix} u_1 & u_3 \\ v_1 & v_3 \end{vmatrix} \mathbf{j} + \begin{vmatrix} u_1 & u_2 \\ v_1 & v_2 \end{vmatrix} \mathbf{k}$$

2개의 Vector a, b로 만들어지는 Area를 계산



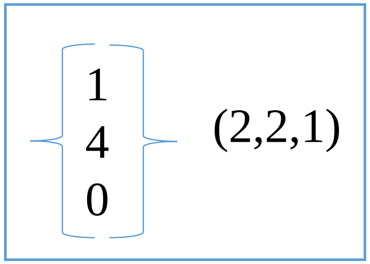
$$\text{Area} = \overbrace{\|\mathbf{u}\|}^{\text{Base}} \, \overbrace{\|\mathbf{v}\| \sin\theta}^{\text{Height}} = \|\mathbf{u} \times \mathbf{v}\|$$

Outer( ) and Cross( ) for 1D ndarrays

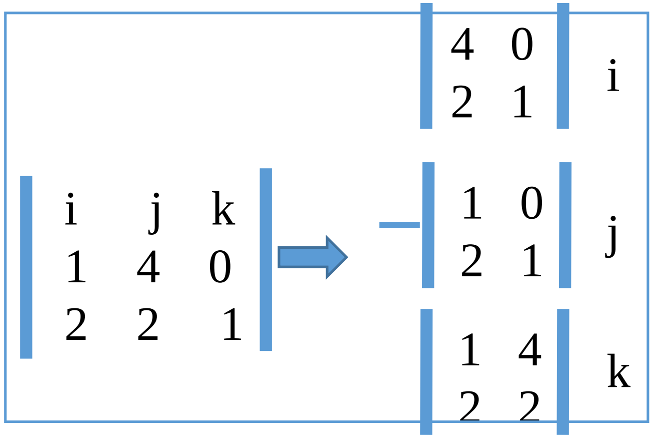
>>> import numpy as np

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

```
>>> np.outer(a, b)
array([[ 2.,   2.,   1.],
       [ 8.,   8.,   4.],
       [ 0.,   0.,   0.]])
```




```
>>> np.cross(a, b)
array([ 4., -1., -6.])
```



Matrix Multiplication (= Matrix Product)

행렬곱

$$\mathbf{A} = \begin{pmatrix} a & b & c \\ x & y & z \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} \alpha & \rho \\ \beta & \sigma \\ \gamma & \tau \end{pmatrix},$$

their matrix products are:

Matrix 2개로 matrix 1개 생성

$$\mathbf{AB} = \begin{pmatrix} a & b & c \\ x & y & z \end{pmatrix} \begin{pmatrix} \alpha & \rho \\ \beta & \sigma \\ \gamma & \tau \end{pmatrix} = \begin{pmatrix} a\alpha + b\beta + c\gamma & a\rho + b\sigma + c\tau \\ x\alpha + y\beta + z\gamma & x\rho + y\sigma + z\tau \end{pmatrix};$$

and

$$\mathbf{BA} = \begin{pmatrix} \alpha & \rho \\ \beta & \sigma \\ \gamma & \tau \end{pmatrix} \begin{pmatrix} a & b & c \\ x & y & z \end{pmatrix} = \begin{pmatrix} \alpha a + \rho x & \alpha b + \rho y & \alpha c + \rho z \\ \beta a + \sigma x & \beta b + \sigma y & \beta c + \sigma z \\ \gamma a + \tau x & \gamma b + \tau y & \gamma c + \tau z \end{pmatrix}.$$

- 행렬사이에 symbol이 없으면 matrix multiplication
- ● 이 있어도 matrix multiplication
- X 로 matrix multiplication을 표현하지 않는다

Application of Matrix Multiplication

폐품수집품의 총가격

Recyclables Collected (lb)			
Item	Week 1	Week 2	Week 3
Glass	29	25	15
Cans	9	10	7
Newspaper	162	125	205

Price Per Pound ($)			
Week	Glass	Cans	Newspaper
1	0.02	0.60	0.02
2	0.02	0.55	0.01
3	0.01	0.42	0.02

$$\begin{pmatrix} 29 & 25 & 15 \\ 9 & 10 & 7 \\ 162 & 125 & 205 \end{pmatrix} \begin{pmatrix} 0.02 & 0.60 & 0.02 \\ 0.02 & 0.55 & 0.01 \\ 0.01 & 0.42 & 0.02 \end{pmatrix} \Rightarrow \begin{pmatrix} 1.23, & 37.45, & 1.13 \\ 0.45, & 13.84, & 0.42 \\ 7.79, & 252.05, & 8.59 \end{pmatrix} \Rightarrow$$

322.94 (Sum of all elements of matrix)

numpy.dot( ) for 2D ndarrays

행렬곱
Matrix Multiplication

- The `dot( )` function also generalizes to matrix multiplication
- If both `a` and `b` are 2D arrays, it is matrix multiplication, but using `matmul( )` or `a @ b` is preferred. (notations for convenience)

```
>>> a = np.array([[0, 1], [2, 3]], float)
>>> b = np.array([2, 3], float)
>>> c = np.array([[1, 1], [4, 0]], float)
```

```
>>> np.dot(b, a)
array([ 6., 11.])
>>> np.dot(a, b)
array([ 3., 13.])
>>> np.dot(a, c)
array([[ 4.,  0.],
       [14.,  2.]])
>>> np.dot(c, a)
array([[2., 4.],
       [0., 4.]])
```

$b \bullet a$

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

$a \bullet b$

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

$a \bullet c$

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

$\mathrm{c} \bullet \mathrm{a}$

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

```
>>> np.dot(c, a)
array([[2., 4.],
       [0., 4.]])
>>> np.matmul(c, a)
array([[2., 4.],
       [0., 4.]])
>>> c @ a
array([[2., 4.],
       [0., 4.]])
```

@ operator calls __matmul__

Matrix Inner Product

행렬내적

- The **Frobenius inner product** is a binary operation that takes two matrices and returns a number
- It is denoted as $\langle \mathbf{A}, \mathbf{B} \rangle_{\mathrm{F}}$
- Ferdinand Georg Frobenius (1849–1917), german mathematician
- The operation is a component-wise inner product of two matrices
- The two matrices must have the same dimension (same number of rows and columns)

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

$$\Downarrow$$

$$\mathrm{A} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix}, \qquad \mathrm{B} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix}$$

$$\langle \mathbf{A}, \mathbf{B} \rangle_{\mathrm{F}} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n = \sum_{i=1}^{n} a_i b_i,$$

For two real-valued matrices, if

$$\mathbf{A} = \begin{pmatrix} 2 & 0 & 6 \\ 1 & -1 & 2 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} 8 & -3 & 2 \\ 4 & 1 & -5 \end{pmatrix}$$

then

$$\begin{aligned} \langle \mathbf{A}, \mathbf{B} \rangle_{\mathrm{F}} &= 2 \cdot 8 + 0 \cdot (-3) + 6 \cdot 2 + 1 \cdot 4 + (-1) \cdot 1 + 2 \cdot (-5) \\ &= 16 + 12 + 4 - 1 - 10 \\ &= 21 \end{aligned}$$

np.inner( ) for 2D ndarrays

행렬내적

- More generally, if ndim(a) = r > 0 and ndim(b) = s > 0,

$$\texttt{np.inner(a, b)}[\underline{i_0, \ldots, i_{r-1}}, \underline{j_0, \ldots, j_{s-1}}]$$
$$= \texttt{sum}(a[\underline{i_0, \ldots, i_{r-1}}, :] * b[\underline{j_0, \ldots, j_{s-1}}, :])$$

- If a and b are 2D ndarrays

```
>>> a = np.array([[0, 1], [2, 3]], float)
>>> b = np.array([[1, 1], [4, 0]], float)
>>>
>>> np.inner(a, b)
array([[1., 0.],
       [5., 8.]])
>>>
>>> np.dot(a, b)
array([[ 4.,  0.],
       [14.,  2.]])
```

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 4 & 0 \end{bmatrix} = \begin{bmatrix} 0\times1+1\times1 & 0\times4+1\times0 \\ 2\times1+3\times1 & 2\times4+3\times0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 4 & 0 \end{bmatrix} = \begin{bmatrix} 0\times1+1\times4 & 0\times1+1\times0 \\ 2\times1+3\times4 & 2\times1+3\times0 \end{bmatrix}$$

numpy.dot( ) in Heterogenious ndarrays

- 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`:

```
        a          dot          b

[[[0., 1.],
  [2., 3.]],               [[1., 1.],
                   .         [4., 0.]]
 [[1., 1.],
  [4., 0.]]]
```

a sum product over last axis of `a` and 1st axis of `b`

```
     [
       [[0., 1.],     .   [[1., 1.],
        [2., 3.]]           [4., 0.]]
=    ,
       [[1., 1.],     .   [[1., 1.],
        [4., 0.]]           [4., 0.]]
                                       ]

=   [[[ 4.,  0.],
      [14.,  2.]],

     [[ 5.,  1.],
      [ 4.,  4.]]]
```

np.inner( ) vs np.dot( ) in Heterogeneous ndarrays

- If `a` is 3D ndarray and `b` is 2D ndarray

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

```
>>> np.inner(a, b)
array([[[ 1.,  0.],
        [ 5.,  8.]],

       [[ 2.,  4.],
        [ 4., 16.]]])
>>>
>>> np.dot(a, b)
array([[[ 4.,  0.],
        [14.,  2.]],

       [[ 5.,  1.],
        [ 4.,  4.]]])
```

```
[
  [[0., 1.],     [[1., 1.],
   [2., 3.]]  •   [4., 0.]]
,                                 =    [[[ 1.,  0.],
  [[1., 1.],     [[1., 1.],              [ 5.,  8.]],
   [4., 0.]]  •   [4., 0.]]
                              ]         [[ 2.,  4.],
                                         [ 4., 16.]]]
```

```
[
  [[0., 1.],     [[1., 1.],
   [2., 3.]]  •   [4., 0.]]
,                                 =    [[[ 4.,  0.],
  [[1., 1.],     [[1., 1.],              [14.,  2.]],
   [4., 0.]]  •   [4., 0.]]
                              ]         [[ 5.,  1.],
                                         [ 4.,  4.]]]
```

Wait! Matrix Data Type in Numpy

- Numpy matrices are strictly 2-dimensional, while numpy arrays (ndarrays) are N-dimensional (np.mat( ) 가 return 하고, print하면 2D List 내부에 comma가 없다)
- Matrix objects are a subclass of ndarray, so they inherit all the attributes and methods of ndarrays
- The main advantage of numpy matrices is that they provide a convenient notation for matrix multiplication: if a and b are matrices, then a*b is their matrix product

Matrix Data Type

```
import numpy as np

a=np.mat('4 3; 2 1')
b=np.mat('1 2; 3 4')
print(a)
# [[4 3]
#  [2 1]]
print(b)
# [[1 2]
#  [3 4]]
print(a*b)
# [[13 20]
#  [ 5  8]]
```

```
>> a
    matrix([[4, 3],
            [2, 1]])
```

Ndarray Data Type

```
c=np.array([[4, 3], [2, 1]])
d=np.array([[1, 2], [3, 4]])
print(c*d)
# [[4 6]
#  [6 4]]
```

```
print(np.dot(c,d))
# [[13 20]
#  [ 5  8]]
```

혹은 아래 방식도 OK!
np.matmul(c,d)
c @ b

Numpy Code using Linalg Functions Example

Nearest Neighbor Search - Iterative Python algorithm and vectorized NumPy version

```
>>> # # # Pure iterative Python # # #
>>> points = [[9,2,8],[4,7,2],[3,4,4],[5,6,9],[5,0,7],[8,2,7],[0,3,2],[7,3,0],[6,1,1],[2,9,6]]
>>> qPoint = [4,5,3]
>>> minIdx = -1
>>> minDist = -1
>>> for idx, point in enumerate(points):  # iterate over all points
        dist = sum([(dp-dq)**2 for dp,dq in zip(point,qPoint)])**0.5  # compute the euclidean distance for each point to q
        if dist < minDist or minDist < 0:  # if necessary, update minimum distance and index of the corresponding point
            minDist = dist
            minIdx = idx

>>> print('Nearest point to q: ', points[minIdx] )
Nearest point to q:  [3, 4, 4]
```

Five lines

np.linalg.norm( ) ➔ vector norm 만들기
np.argmin( ) ➔ index of smallest one

```
>>> # # # Equivalent NumPy vectorization # # #
>>> import numpy as np
>>> points = np.array([[9,2,8],[4,7,2],[3,4,4],[5,6,9],[5,0,7],[8,2,7],[0,3,2],[7,3,0],[6,1,1],[2,9,6]])
>>> qPoint = np.array([4,5,3])
>>> minIdx = np.argmin(np.linalg.norm(points-qPoint,axis=1))  # compute all euclidean distances at once and return the index of the smallest one
>>> print('Nearest point to q: ', points[minIdx])
Nearest point to q:  [3 4 4]
```

Only a single line!

Table of Contents

Numpy File IO: Numpy NDArray, Text File, CSV File [1/2]

- savetxt( ) function saves numpy ndarray to a csv file

```python
import numpy as np

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

np.savetxt(r"C:\Users\hjk\Desktop\foo.csv", a, delimiter= ",")

np.savetxt(r"C:\Users\hjk\Desktop\foo.txt", a, delimiter= ",")
```

foo.csv

	A	B	C	D
1	1.00E+00	2.00E+00	3.00E+00	
2	4.00E+00	5.00E+00	6.00E+00	
3				

foo.txt

foo - 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
1.000000000000000000e+00,2.000000000000000000e+00,3.000000000000000000e+00
4.000000000000000000e+00,5.000000000000000000e+00,6.000000000000000000e+00
```

Numpy File IO: Numpy NDArray, Text File, CSV File [2/2]

- loadtxt( ) function은 csv file or text file을 numpy ndarray 로 읽어드린다

```
In [15]: b = np.loadtxt(fname= r"C:\Users\hjk\Desktop\foo.txt", delimiter=',')

In [16]: b
Out[16]:
array([[1., 2., 3.],
       [4., 5., 6.]])

In [17]: b = np.loadtxt(fname= r"C:\Users\hjk\Desktop\foo.csv", delimiter=',')

In [18]: b
Out[18]:
array([[1., 2., 3.],
       [4., 5., 6.]])
```

Table of Contents

Numpy Function List [1/13]

Array creation

`empty`(shape[, dtype, order])
`empty_like`(a[, dtype, order, subok])
`eye`(N[, M, k, dtype])
`identity`(n[, dtype])
`ones`(shape[, dtype, order])
`ones_like`(a[, dtype, order, subok])
`zeros`(shape[, dtype, order])
`zeros_like`(a[, dtype, order, subok])
`full`(shape, fill_value[, dtype, order])
`full_like`(a, fill_value[, dtype, order, subok])

`array`(object[, dtype, copy, order, subok, ndmin])
`asarray`(a[, dtype, order])
`asanyarray`(a[, dtype, order])
`ascontiguousarray`(a[, dtype])
`asmatrix`(data[, dtype])
`copy`(a[, order])
`frombuffer`(buffer[, dtype, count, offset])
`fromfile`(file[, dtype, count, sep])
`fromfunction`(function, shape, **kwargs)
`fromiter`(iterable, dtype[, count])
`fromstring`(string[, dtype, count, sep])
`loadtxt`(fname[, dtype, comments, delimiter, ...])

Numpy Function List [2/13]

Numerical ranges

arange([start,] stop[, step,][, dtype])
linspace(start, stop[, num, endpoint, ...])
logspace(start, stop[, num, endpoint, base, ...])
geomspace(start, stop[, num, endpoint, dtype])
meshgrid(*xi, **kwargs)
mgrid
ogrid

Building matrices

diag(v[, k])
diagflat(v[, k])
tri(N[, M, k, dtype])
tril(m[, k])
triu(m[, k])
vander(x[, N, increasing])
mat(data[, dtype])
bmat(obj[, ldict, gdict])

Changing array shape

reshape(a, newshape[, order])
ravel(a[, order])
ndarray.flat
ndarray.flatten([order])

Transpose operation

moveaxis(a, source, destination)
rollaxis(a, axis[, start])
swapaxes(a, axis1, axis2)
ndarray.T
transpose(a[, axes])

Numpy Function List [3/13]

Changing number of dimension

atleast_1d(*arys)
atleast_2d(*arys)
atleast_3d(*arys)
broadcast
broadcast_to(array, shape[, subok])
broadcast_arrays(*args, **kwargs)
expand_dims(a, axis)
squeeze(a[, axis])

Concatenating arrays

concatenate((a1, a2, ...)[, axis])
stack(arrays[, axis])
column_stack(tup)
dstack(tup)
hstack(tup)
vstack(tup)
block(arrays)

Changing kind of array

asarray(a[, dtype, order])
asanyarray(a[, dtype, order])
asmatrix(data[, dtype])
asfarray(a[, dtype])
asfortranarray(a[, dtype])
ascontiguousarray(a[, dtype])
asarray_chkfinite(a[, dtype, order])
asscalar(a)
require(a[, dtype, requirements])

Splitting arrays

split(ary, indices_or_sections[, axis])
array_split(ary, indices_or_sections[, axis])
dsplit(ary, indices_or_sections)
hsplit(ary, indices_or_sections)
vsplit(ary, indices_or_sections)

Numpy Function List [4/13]

Tile arrays

tile(A, reps)
repeat(a, repeats[, axis])

Rearranging elements

flip(m, axis)
fliplr(m)
flipud(m)
reshape(a, newshape[, order])
roll(a, shift[, axis])

Adding and removing elements

delete(arr, obj[, axis])
insert(arr, obj, values[, axis])
append(arr, values[, axis])
resize(a, new_shape)
trim_zeros(filt[, trim])
unique(ar[, return_index, return_inverse, ..

Bit operation

bitwise_and(x1, x2, /[, out, where, ...])
bitwise_or(x1, x2, /[, out, where, casting, ...])
bitwise_xor(x1, x2, /[, out, where, ...])
invert(x, /[, out, where, casting, order, ...])
left_shift(x1, x2, /[, out, where, casting, ...])
right_shift(x1, x2, /[, out, where, ...])

Numpy Function List [5/13]

Financial function

`fv`(rate, nper, pmt, pv[, when])
`pv`(rate, nper, pmt[, fv, when])
`npv`(rate, values)
`pmt`(rate, nper, pv[, fv, when])
`ppmt`(rate, per, nper, pv[, fv, when])
`ipmt`(rate, per, nper, pv[, fv, when])
`irr`(values)
`mirr`(values, finance_rate, reinvest_rate)
`nper`(rate, pmt, pv[, fv, when])
`rate`(nper, pmt, pv, fv[, when, guess, tol, ...])

Functional programming

`apply_along_axis`(func1d, axis, arr, *args, ...)
`apply_over_axes`(func, a, axes)
`vectorize`(pyfunc[, otypes, doc, excluded, ...])
`frompyfunc`(func, nin, nout)
`piecewise`(x, condlist, funclist, *args, **kw)

Generating index arrays

`nonzero`(a)
`where`(condition, [x, y])
`indices`(dimensions[, dtype])
`ix_`(*args)
`ogrid`
`ravel_multi_index`(multi_index, dims[, mode, ...])
`unravel_index`(indices, dims[, order])
`diag_indices`(n[, ndim])
`diag_indices_from`(arr)
`mask_indices`(n, mask_func[, k])
`tril_indices`(n[, k, m])
`tril_indices_from`(arr[, k])
`triu_indices`(n[, k, m])
`triu_indices_from`(arr[, k])

Numpy Function List [6/13]

Indexing-like operations

`take`(a, indices[, axis, out, mode])
`choose`(a, choices[, out, mode])
`compress`(condition, a[, axis, out])
`diag`(v[, k])
`diagonal`(a[, offset, axis1, axis2])
`select`(condlist, choicelist[, default])

Inserting data into array

`place`(arr, mask, vals)
`put`(a, ind, v[, mode])
`putmask`(a, mask, values)
`fill_diagonal`(a, val[, wrap])

Save and Load

`load`(file[, mmap_mode, allow_pickle, ...])
`save`(file, arr[, allow_pickle, fix_imports])
`savez`(file, *args, **kwds)
`savez_compressed`(file, *args, **kwds)
`loadtxt`(fname[, dtype, comments, delimiter, ...])
`savetxt`(fname, X[, fmt, delimiter, newline, ...])
`genfromtxt`(fname[, dtype, comments, ...])
`fromregex`(file, regexp, dtype)
`fromstring`(string[, dtype, count, sep])
`ndarray.tofile`(fid[, sep, format])
`ndarray.tolist`()

Raw binary files

`fromfile`(file[, dtype, count, sep])
`ndarray.tofile`(fid[, sep, format])

Numpy Function List [7/13]

Array contents checking

`isfinite`(x, /[, out, where, casting, order, ...])
`isinf`(x, /[, out, where, casting, order, ...])
`isnan`(x, /[, out, where, casting, order, ...])
`isneginf`(x[, out])
`isposinf`(x[, out])

Array type testing

`iscomplex`(x)
`iscomplexobj`(x)
`isfortran`(a)
`isreal`(x)
`isrealobj`(x)
`isscalar`(num)

Logical operations

`logical_and`(x1, x2, /[, out, where, ...])
`logical_or`(x1, x2, /[, out, where, casting, ...])
`logical_not`(x, /[, out, where, casting, ...])
`logical_xor`(x1, x2, /[, out, where, ...])

Comparison

`allclose`(a, b[, rtol, atol, equal_nan])
`isclose`(a, b[, rtol, atol, equal_nan])
`array_equal`(a1, a2)
`array_equiv`(a1, a2)

`greater`(x1, x2, /[, out, where, casting, ...])
`greater_equal`(x1, x2, /[, out, where, ...])
`less`(x1, x2, /[, out, where, casting, ...])
`less_equal`(x1, x2, /[, out, where, casting, ...])
`equal`(x1, x2, /[, out, where, casting, ...])
`not_equal`(x1, x2, /[, out, where, casting, ...])

Numpy Function List [8/13]

Mathematical - Trigonometric

`sin`(x, /[, out, where, casting, order, ...])
`cos`(x, /[, out, where, casting, order, ...])
`tan`(x, /[, out, where, casting, order, ...])
`arcsin`(x, /[, out, where, casting, order, ...])
`arccos`(x, /[, out, where, casting, order, ...])
`arctan`(x, /[, out, where, casting, order, ...])
`hypot`(x1, x2, /[, out, where, casting, ...])
`arctan2`(x1, x2, /[, out, where, casting, ...])

`degrees`(x, /[, out, where, casting, order, ...])
`radians`(x, /[, out, where, casting, order, ...])
`unwrap`(p[, discont, axis])

`deg2rad`(x, /[, out, where, casting, order, ...])
`rad2deg`(x, /[, out, where, casting, order, ...])

`sinh`(x, /[, out, where, casting, order, ...])
`cosh`(x, /[, out, where, casting, order, ...])
`tanh`(x, /[, out, where, casting, order, ...])
`arcsinh`(x, /[, out, where, casting, order, ...])
`arccosh`(x, /[, out, where, casting, order, ...])
`arctanh`(x, /[, out, where, casting, order, ...])

Mathematical - Rounding

`around`(a[, decimals, out])
`round_`(a[, decimals, out])
`rint`(x, /[, out, where, casting, order, ...])
`fix`(x[, out])
`floor`(x, /[, out, where, casting, order, ...])
`ceil`(x, /[, out, where, casting, order, ...])
`trunc`(x, /[, out, where, casting, order, ...])

Numpy Function List [9/13]

Sums, Products, Differences

prod(a[, axis, dtype, out, keepdims])
sum(a[, axis, dtype, out, keepdims])
nanprod(a[, axis, dtype, out, keepdims])
nansum(a[, axis, dtype, out, keepdims])
cumprod(a[, axis, dtype, out])
cumsum(a[, axis, dtype, out])
nancumprod(a[, axis, dtype, out])
nancumsum(a[, axis, dtype, out])
diff(a[, n, axis])
ediff1d(ary[, to_end, to_begin])
gradient(f, *varargs, **kwargs)
cross(a, b[, axisa, axisb, axisc, axis])
trapz(y[, x, dx, axis])

Exponents and Logarithms

exp(x, /[, out, where, casting, order, ...])
expm1(x, /[, out, where, casting, order, ...])
exp2(x, /[, out, where, casting, order, ...])
log(x, /[, out, where, casting, order, ...])
log10(x, /[, out, where, casting, order, ...])
log2(x, /[, out, where, casting, order, ...])
log1p(x, /[, out, where, casting, order, ...])
logaddexp(x1, x2, /[, out, where, casting, ...])
logaddexp2(x1, x2, /[, out, where, casting, ...])

Handling Complex numbers

angle(z[, deg])
real(val)
imag(val)
conj(x, /[, out, where, casting, order, ...])

Numpy Function List [10/13]

Arithmetic operations

add(x1, x2, /[, out, where, casting, order, ...])
reciprocal(x, /[, out, where, casting, ...])
negative(x, /[, out, where, casting, order, ...])
multiply(x1, x2, /[, out, where, casting, ...])
divide(x1, x2, /[, out, where, casting, ...])
power(x1, x2, /[, out, where, casting, ...])
subtract(x1, x2, /[, out, where, casting, ...])
true_divide(x1, x2, /[, out, where, ...])
floor_divide(x1, x2, /[, out, where, ...])
float_power(x1, x2, /[, out, where, ...])
fmod(x1, x2, /[, out, where, casting, ...])
mod(x1, x2, /[, out, where, casting, order, ...])
modf(x[, out1, out2], / [[, out, where, ...])
remainder(x1, x2, /[, out, where, casting, ...])
divmod(x1, x2[, out1, out2], / [[, out, ...])
dot(a, b[, out])

Mathematical - Miscellaneous

convolve(a, v[, mode])
clip(a, a_min, a_max[, out])
sqrt(x, /[, out, where, casting, order, ...])
cbrt(x, /[, out, where, casting, order, ...])
square(x, /[, out, where, casting, order, ...])
absolute(x, /[, out, where, casting, order, ...])
fabs(x, /[, out, where, casting, order, ...])
sign(x, /[, out, where, casting, order, ...])
heaviside(x1, x2, /[, out, where, casting, ...])
maximum(x1, x2, /[, out, where, casting, ...])
minimum(x1, x2, /[, out, where, casting, ...])
fmax(x1, x2, /[, out, where, casting, ...])
fmin(x1, x2, /[, out, where, casting, ...])
nan_to_num(x[, copy])
real_if_close(a[, tol])
interp(x, xp, fp[, left, right, period])

Numpy Function List [11/13]

Linear Algebra (numpy.linalg)

Matrix and vector products¶

dot(a, b[, out])
linalg.multi_dot(arrays)

vdot(a, b)
inner(a, b)
outer(a, b[, out])
matmul(a, b[, out])
tensordot(a, b[, axes])

Norms and other numbers

linalg.norm(x[, ord, axis, keepdims])
linalg.cond(x[, p])
linalg.det(a)
linalg.matrix_rank(M[, tol, hermitian])
linalg.slogdet(a)

trace(a[, offset, axis1, axis2, dtype, out])

Decompositions¶

linalg.cholesky(a)
linalg.qr(a[, mode])
linalg.svd(a[, full_matrices, compute_uv])

Matrix Eigenvalues

linalg.eig(a)
linalg.eigh(a[, UPLO])

linalg.eigvals(a)
linalg.eigvalsh(a[, UPLO])

Solving equations

linalg.solve(a, b)
linalg.tensorsolve(a, b[, axes])
linalg.lstsq(a, b[, rcond])
linalg.inv(a)
linalg.pinv(a[, rcond])
linalg.tensorinv(a[, ind])

Numpy Function List [12/13]

Random sampling (numpy.random)

rand(d0, d1, ..., dn)
randn(d0, d1, ..., dn)
randint(low[, high, size, dtype])
random_integers(low[, high, size])
random_sample([size])
random([size])
ranf([size])
sample([size])
choice(a[, size, replace, p])
bytes(length)

Permutation

shuffle(x)
permutation(x)

Set Boolean operations

in1d(ar1, ar2[, assume_unique, invert])
intersect1d(ar1, ar2[, assume_unique])
isin(element, test_elements[, ...])
setdiff1d(ar1, ar2[, assume_unique])
setxor1d(ar1, ar2[, assume_unique])
union1d(ar1, ar2)

Sorting

sort(a[, axis, kind, order])
lexsort(keys[, axis])
argsort(a[, axis, kind, order])
ndarray.sort([axis, kind, order])
msort(a)
sort_complex(a)
partition(a, kth[, axis, kind, order])
argpartition(a, kth[, axis, kind, order])

Numpy Function List [13/13]

Searching

`argmax`(a[, axis, out])
`nanargmax`(a[, axis])
`argmin`(a[, axis, out])
`nanargmin`(a[, axis])
`argwhere`(a)
`nonzero`(a)
`flatnonzero`(a)
`where`(condition, [x, y])
`searchsorted`(a, v[, side, sorter])
`extract`(condition, arr)

Counting

`count_nonzero`(a[, axis])

Statistics

`median`(a[, axis, out, overwrite_input, keepdims])
`average`(a[, axis, weights, returned])
`mean`(a[, axis, dtype, out, keepdims])
`std`(a[, axis, dtype, out, ddof, keepdims])
`var`(a[, axis, dtype, out, ddof, keepdims])
`nanmedian`(a[, axis, out, overwrite_input, ...])
`nanmean`(a[, axis, dtype, out, keepdims])
`nanstd`(a[, axis, dtype, out, ddof, keepdims])
`nanvar`(a[, axis, dtype, out, ddof, keepdims])

`corrcoef`(x[, y, rowvar, bias, ddof])
`correlate`(a, v[, mode])
`cov`(m[, y, rowvar, bias, ddof, fweights, ...])