Python Data

2019/2/12

1. 1.Numpy

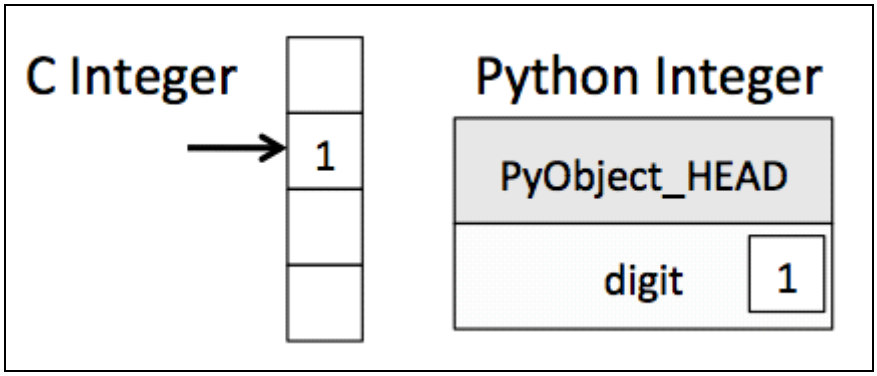
1.1. 파이썬 기본 데이터

1.1.1. 기본타입 구조

파이썬 데이터 타입의 구조와 이해
파이선의 기본타입은 C언어의 타입과는 다르다.
다음은 파이썬의 long타입을 표현하는 구조체 코드이다.
2번째 멤버인 ob_type은 현재 타입의 정보를 갖고 있다.
즉, 구조체 내의 부가적인 타입 정보에 따라 타입이 동적으로 결정 된다. *(참조 . C++ RTTI)*

```
struct _longobject
{
    long ob_refcnt;
    PytypeObject * ob_type;
    size_t ob_size;
    long ob_digit[1];
}
```

C언어의 변수는 실제 메모리공간에 대한 이름(label)이다.
반면, 파이썬의 기본 타입은 타입과 관련된 여러가지 정보를 포함하는 구조체의 형태이다.



https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/

1.1.2. 기본 타입

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

https://docs.scipy.org/doc/numpy-1.13.0/user/basics.types.html

```
import numpy as np
#명시적인 타입 생성
x= np.int32(1234)
print( x.dtype)
arr = np.arange(5, dtype = 'f')
print( arr)
print( np.int8(arr))
```

1.2. **Numpy** 시작

- Numerical Python
- 파이썬의 고성능 과학 계산용 패키지
- Matrix와 Vector와 같은 Array 연산의 사실상의 표준
- 일반 List에 비해 빠르고, 메모리 효율적
- 반복문 없이 데이터 배열에 대한 처리를 지원함
- 선형대수와 관련된 다양한 기능을 제공함
- C, C++, 포트란 등의 언어와 통합 가능

1.2.1. 파이썬 자료구조 리스트

리스트

- 가변적(동적) 표준 컨테이너
- 동적 타입 지원을 통한 서로 다른 데이터 요소 관리 가능

```
In [6]: L = list(range(10))
        print(L)
        type(L)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Out[6]: list
```

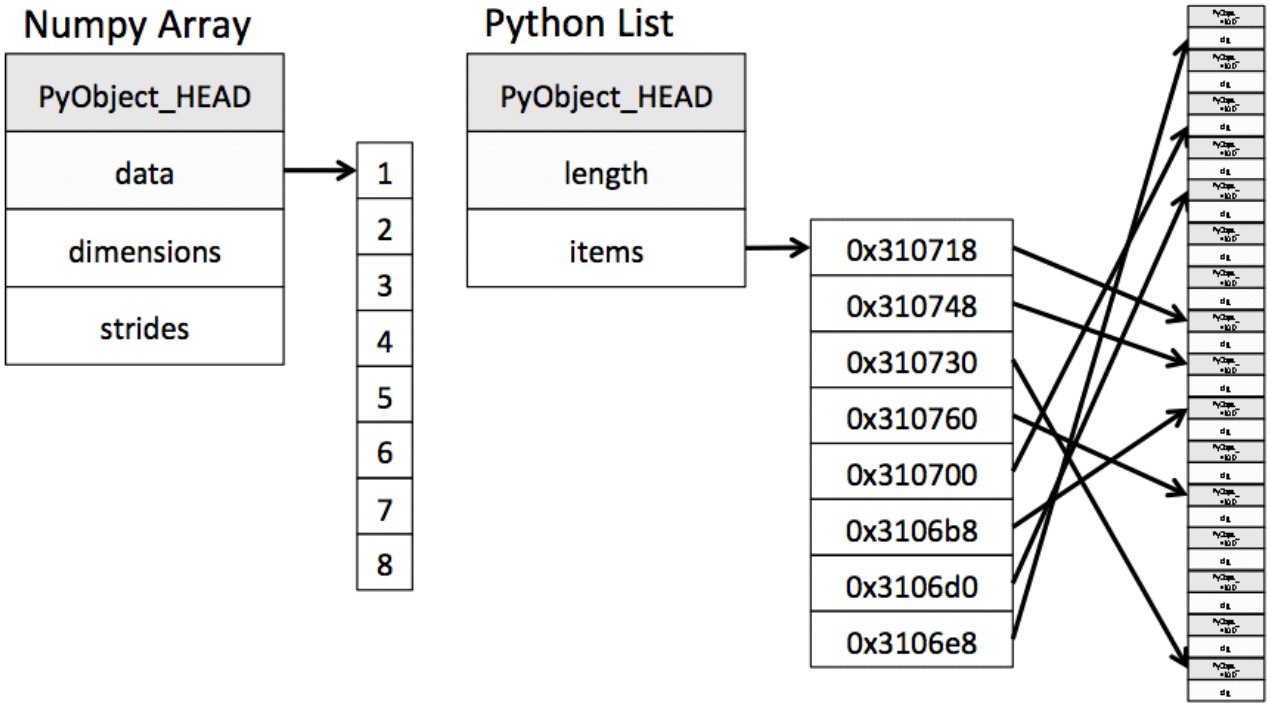
- 동적인 타입 변환

```
In [4]: L2 = [str(c) for c in L]
        L2
Out[4]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

- 서로 다른 타입을 하나의 리스트

```
In [6]: L3 = [True, 123, 'abc' , 3.5 ]
        [type(item) for item in L3]
Out[6]: [bool, int, str, float]
```

유연한 타입 허용은 부가적인 정보 (타입정보, 레퍼런스 카운팅, 기타 부가정보) 에 대한 비용
만약 다루어야하는 데이터가 위와 같은 다양한 타입이 아닌 동일 타입이라면…
고정 타입의 Numpy스타일의 배열을 사용 할것을 권장.



https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/

Numpy Array는 연접한 데이터 공간(배열)에 대한 단일 포인터를 통해 갖는다.
Python List는 객체를 차례로 가르키는 포인터 블록에 대한 포인터 (포인터의 포인터)

1.2.2. 파이썬 array

파이썬의 고정 타입 배열 (3.3 이후)

```
In [3]:
import array
L = list(range(10))
A = array.array('i', L) # 타입 지정 'i' int
A
```

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

파이썬의 array객체는 배열기반의 효율적인 데이터 저장소 제공
But , 효율적인 연산기능의 제공은 제한적…
Numpy ndarray 객체를 사용하자 !

1.2.3. Numpy list to array

파이썬 리스트를 통해 배열 생성
같은 타입만 가능
➔ Np.array([1,2,3])

```
In [3]:
import array
L = list(range(10))
A = array.array('i', L) # 타입 지정 'i' int
A

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

다른 타입이 들어올 경우 numpy 는 상위 타입으로 캐스팅 하여 저장

```
In [7]:
import numpy as np
a = np.array([1,2,3,4.0,'aaa'])
print(a)
type(a[0])

['1' '2' '3' '4.0' 'aaa']

Out[7]: numpy.str_
```

명시적으로 타입을 지정하여 배열 생성

```
In [10]:
#명시적인 타입 지정
import numpy as np
np.array([1,2,3,4.5,5.5] , dtype='float32' )

Out[10]: array([1. , 2. , 3. , 4.5, 5.5], dtype=float32)
```

다차원 1차원 리스트 여러 개로 다차원 배열을 생성

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

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

1.3.Ndarray

1.3.1. Ndarray Create

규모가 큰 배열의 경우 Numpy 함수들을 이용하여 처음부터 큰 배열 생성

- ➔ zeros() [0,0,0]
- ➔ ones() [1,1,1]
- ➔ full(…, 3.14) 3.14
- ➔ eye : 대각선이 1이고 나머지는 0인 2차원 배열 생성

```
In [24]:
import numpy as np
```

```
In [25]:
np.zeros( 10, dtype = int) # 0으로 채운 int 배열
```

```
Out[25]:
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
In [26]:
np.ones((3,5) , dtype = float) # 3 * 5 부동 소수점 배열
```

```
Out[26]:
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

```
In [27]:
np.full((3,5) ,  3.14) # 3.14로 채운 3 x 5 배열
```

```
Out[27]:
array([[3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14]])
```

```
In [28]:
np.arange(0,10,2) # 0~10 간격 : 2
```

```
Out[28]:
array([0, 2, 4, 6, 8])
```

```
In [29]:
np.linspace(0,1,5) # 0~1사이에 같은 간격의 5개의 값으로 채움
```

```
Out[29]:
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

```
In [30]:
np.eye(3)  #3x3 단위 행렬 만들기
```

```
Out[30]:
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Random 활용 생성

```
import numpy as np  # 모듈 생성
x = np.random.random( (3,3))
np.random.normal( 0,1,(3,3))  # 정규 분포  평균 : 0 표준편차 1
np.random.randint(1,45,( 5,6 )) # 5 x 6 1~45 사이 정수
```

empty – shape만 주어지고 비어있는 ndarray 생성 → 쓰레기값

```
1 np.empty( shape=(10))
```

```
array([6.23042070e-307, 4.67296746e-307, 1.69121096e-306, 2.33646845e-307,
       9.34577196e-307, 1.37962456e-306, 9.34608432e-307, 1.60219578e-306,
       1.33511290e-306, 1.78012156e-306])
```

ones_like

```
1
```

```
1 m = np.arange(30).reshape(5,6)
2 m
```

```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29]])
```

```
1 np.ones_like(m)
```

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

Identity

- 단위 행렬

```
1 np.identity(n=5)
```

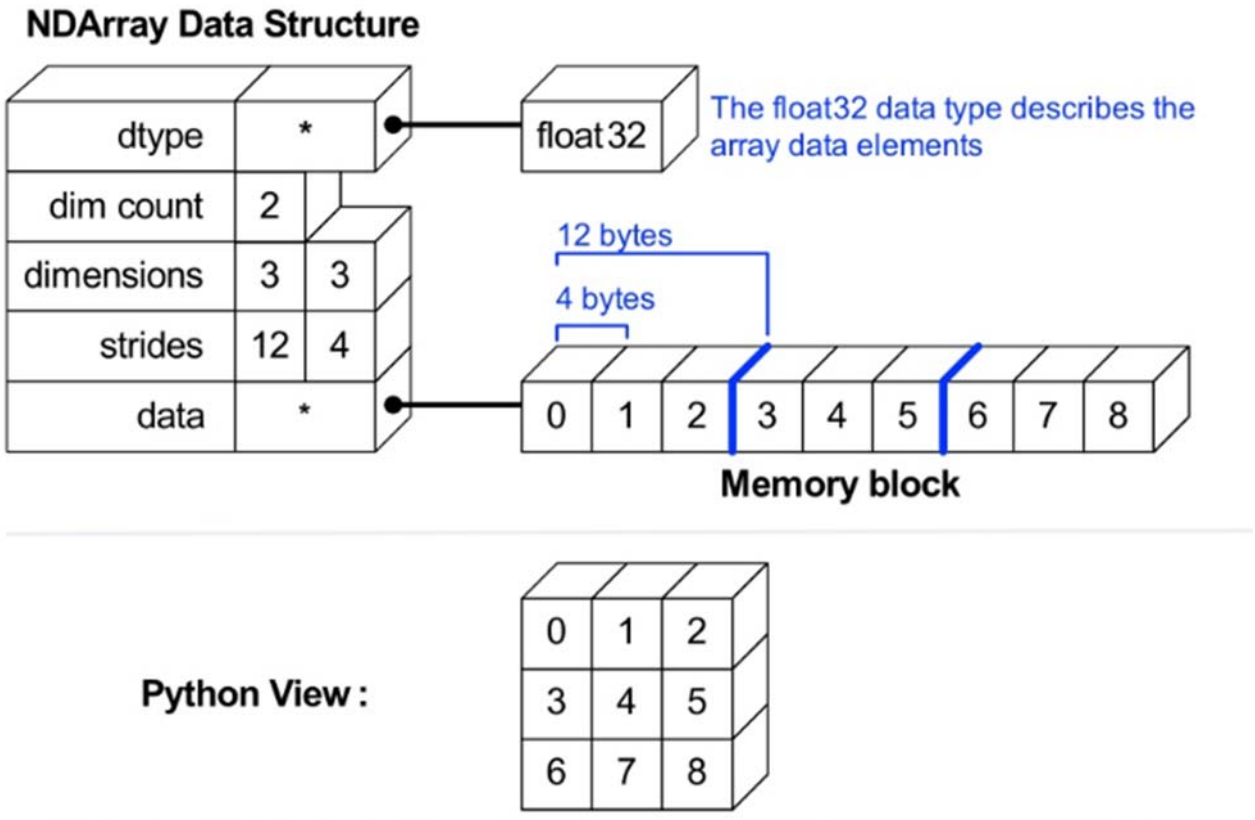
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

1.4.**Numpy** 배열 기초

1.4.1. **Array** 속성

Numpy 배열

- numpy 배열은 동일한 타입의 값을 갖는다.
- 배열의 차원을 rank라고 한다.
- Shape : 각차원의 크기를 튜플로 표시



https://www.slideshare.net/enthought/numpy-talk-at-siam

```
import numpy as np  # 모듈 생성


x1 = np.random.randint(10, size =3)
x2 = np.random.randint(10, size =(3,4))
x3 = np.random.randint(10, size =(3,4,5))
print( x3.ndim )  # 차원의 개수
print( x3.shape ) # 각 차원의 크기
print( x3.size )
print( x3.dtype)
```

```
print(x3.itemsize)
print(x3.nbytes)
```

```
3
(3, 4, 5)
60
int32
4
240
```

1.4.2. 배열 인덱싱

Numpy 인덱싱 : 배열의 성분을 추출

1 차원 : 열 index를 이용한 데이터 추출

```
x = [1,2,3,4,5]
x1 = np.array(x)
```

```
x1[0]  # 첫번째값
```

```
1
```

```
x1[0] = 123  # 값변경
x1
```

```
array([123,   2,   3,   4,   5])
```

```
x1[-1] , x1[-2]  # 마지막값
```

```
(5, 4)
```

```
x1[0:3] , x1[3:] , x1[:-1]
```

```
(array([123,   2,   3]), array([4, 5]), array([123,   2,   3,   4]))
```

```
x1[0:] = 0
x1
```

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

2차원 배열 인덱싱

2차원 array는 반드시 [,] 2개의 인자 필요

[행, 열]

```
x2 = [ [1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]  ]
a = np.array(x2)
a
```

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

```
a[:] #  배열 전체
```

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

```
#  행   열
a[ 0: , 0: ]  # 0에서부터 전체
```

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

```
a[ 0:1 , 0: ]   #  1행   0~ 전체
```

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

```
a[ 0:1, 0:1 ] , a[ 0:1, 0:2 ] , a[ 0:1, 0:3 ]
```

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

```
a[ 0:1, 0:1 ] , a[ 0:2, 0:1 ] , a[ 0:3, 0:1 ]
```

```
(array([[1]]), array([[1],
        [5]]), array([[1],
        [5],
        [9]]))
```

```
a[0::2, :: ]  #  0행부터 전체 2씩 증가 열은 전체
```

```
array([[ 1,  2,  3,  4],
       [ 9, 10, 11, 12]])
```

```
a[1::2, 0::2 ] = 0 # 1행부터 2씩 증가 열은 0부터 2씩 증가 0으로
a[:]
```

```
array([[ 1,  2,  3,  4],
       [ 0,  6,  0,  8],
       [ 9, 10, 11, 12],
       [ 0, 14,  0, 16]])
```

Boolean indexing

```
import numpy as np
data = np.array([1,2,3,4,5])
mask  =np.array([ True, False,False,True,True])

result = data[mask]
result
```

```
array([1, 4, 5])
```

Boolean indexing Mask 표현식 생성

Ex) 짝수값만 True 지정

```
src = [ [1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]  ]
arr = np.array(src)

mask = (arr %2 == 0 )
mask
```

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

```
arr[mask]
```

```
array([ 2,  4,  6,  8, 10, 12, 14, 16])
```

N = arr[arr % 2 == 0] ??

1.4.3. 배열 슬라이싱

: 콜론 기호를 이용 슬라이스 표기법으로 하위 배열에 접근 → 표준 리스트 구문따름
- List와 달리 행과 열 부분을 나눠서 slicing이 가능함
- Matrix의 부분 집합을 추출할 때 유용함

`X[start : stop : step ]`

1차원 슬라이싱

```
x = np.arange(10)
x
```

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

```
x[:5]  # 앞서부터 5개
```

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

```
x[5:]  # 인덱스 5부터 끝까지
```

```
array([5, 6, 7, 8, 9])
```

```
x[3:6] # 중간 부분 배열
```

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

```
x[0:9:2] # 0 ~ 9  2씩증가
```

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

```
x[::2] ,  x[1::2]
```

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

다차원

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

```
x2 = [ [1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]  ]
x2 = np.array(x2)
x2
```

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

```
x2[:2]  # 2개의 행
```

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

```
x2[:2 ,:2] # 행2 , 열2
```

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

```
x2[::2, ::2]
```

```
array([[ 1,  3],
       [ 9, 11]])
```

Numpy view

- 배열 스라이싱은 복사본 배열이 만들어지는 것이 아니라 뷰(view)를 반환한다.
- 큰 데이터 셋을 다를 때 데이터의 복사 없이 데이터의 일부에 접근 하고 처리 가능

```
x2
```

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

```
x2_view = x2[ 1::, ::2]
x2_view
```

```
array([[9, 1, 5],
       [7, 6, 7],
       [9, 9, 0]])
```

```
x2_view[0,0] = 9999;
x2
```

```
array([[   6,    9,    4,    1,    5],
       [9999,    7,    1,    3,    5],
       [   7,    3,    6,    6,    7],
       [   9,    1,    9,    6,    0]])
```

새로운 배열을 생성 copy()

```
x2_copy= x2[ 1::, ::2].copy()
x2_copy
```

```
array([[9999,    1,    5],
       [   7,    6,    7],
       [   9,    9,    0]])
```

```
x2_copy[0,0] = 1234;
x2_copy
```

```
array([[1234,    1,    5],
       [   7,    6,    7],
       [   9,    9,    0]])
```

1.4.4. Reshape

- Array shape 크기를 변경함

전체 사이즈만 같으면 다차원으로 자유롭게 변경 가능

```
matrics = np.arange(1, 9)
matrics
```

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

```
matrics.reshape((2,4))
```

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

```
matrics.reshape((2,2,2))
```

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

       [[5, 6],
        [7, 8]]])
```

```
matrics.reshape(-1,1)
```

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

newaxis

```
v = np.array([1,2,3])
v.reshape((1,3)) # reshapce를 이용한 행 벡터
```

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

```
v.reshape(3,1) # reshapce를 이용한 열 벡터
```

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

```
v[:, np.newaxis] # newaxis 이용한 열 벡터
```

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

flatten

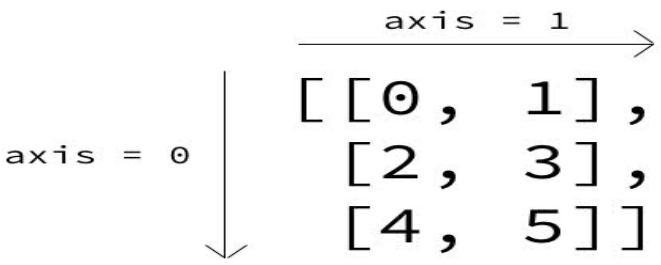
```
v.flatten() # 다차원 --> 1차원
```

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

1.5.배열 연산

1.5.1. axis

- 모든 operation function을 실행할 때, 기준이 되는 dimension 축
- numpy에서는 배열의 연산을 위한 여러 함수 제공
- 각배열의 요소의 합 : sum()
- 각배열의 요소의 곱 : prod()
- Axis 옵션을 통해 연산할 축(axis)을 옵션을 지정



https://deepage.net/features/numpy-axis.html

```
import numpy as np
x = np.arange(1,13).reshape(3,4)
x
```

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

```
x.sum( axis=0)
```

```
array([15, 18, 21, 24])
```

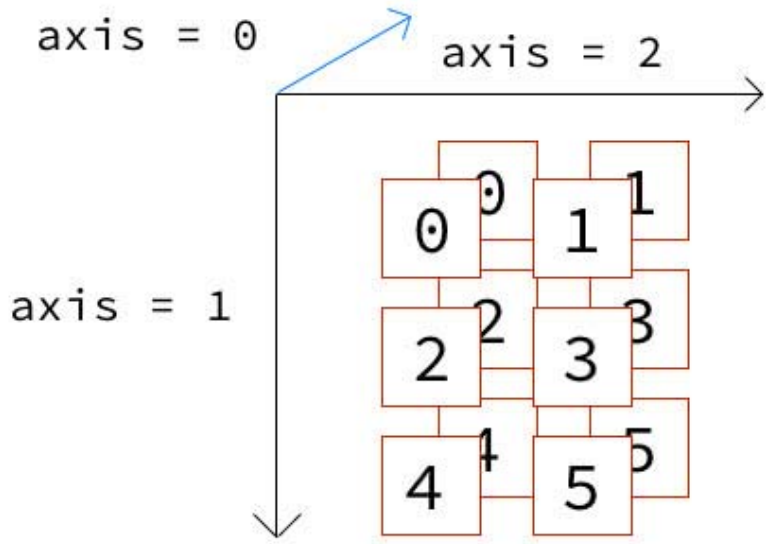
```
x.sum( axis=1)
```

```
array([10, 26, 42])
```

prod

```
p = np.array([ [1,2],[3,4]])
np.prod(p) , np.prod(p, axis=0), np.prod(p, axis=1)
```

```
(24, array([3, 8]), array([ 2, 12]))
```



https://deepage.net/features/ numpy-axis.html

```
import numpy as np
x = np.arange(1,13).reshape(2,3,2)
x
```

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

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

```
x.sum( axis=0)
```

```
array([[ 8, 10],
       [12, 14],
       [16, 18]])
```

```
x.sum( axis=1)
```

```
array([[ 9, 12],
       [27, 30]])
```

```
x.sum( axis=2)
```

```
array([[ 3,  7, 11],
       [15, 19, 23]])
```

1.5.2. 배열 연결 및 분할

Concatenate , vstack ., hstack 이용하여 배열 연결 및 결합

```
x = np.array([1,2,3])
y = np.array([3,2,1])
np.concatenate( [ x ,y ])  # concatenate
```

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

```
x = np.array([1,2,3])
y = np.array([3,2,1])
np.concatenate( [ x ,y ])  # concatenate
```

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

```
z = np.array([2,3,4])
np.concatenate( [x , y , z])
```

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

Concatenate + axis

Axis에 따라 결합의 결과가 달라진다.

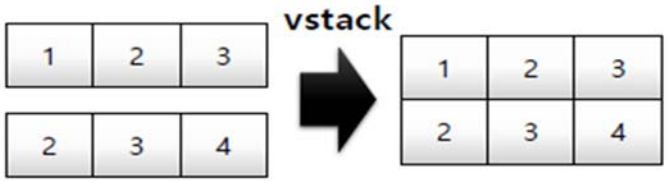
Axis를 기준으로 결합

```
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]])
```



Vstack , hstack

```
x = np.array([1,2,3])
y = np.array([3,2,1])
np.vstack((x, y))
```

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

```
x = np.array([1,2,3])
y = np.array([3,2,1])
np.hstack((x, y))
```

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

```
x = np.array( [ [1], [2] ])
y = np.array( [ [3], [4] ])
np.vstack((x, y))
```

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

np.split

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

```
x1, x2, x3 = np.split( x , [3,7]) #  [분할될 인덱스]
```

```
x1, x2, x3
```

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

np.vsplit np.hsplit

```
grid = np.arange(16).reshape(4,4)
grid
```

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

```
top , bottom = np.vsplit( grid , [2])
print(top)
print(bottom)
```

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

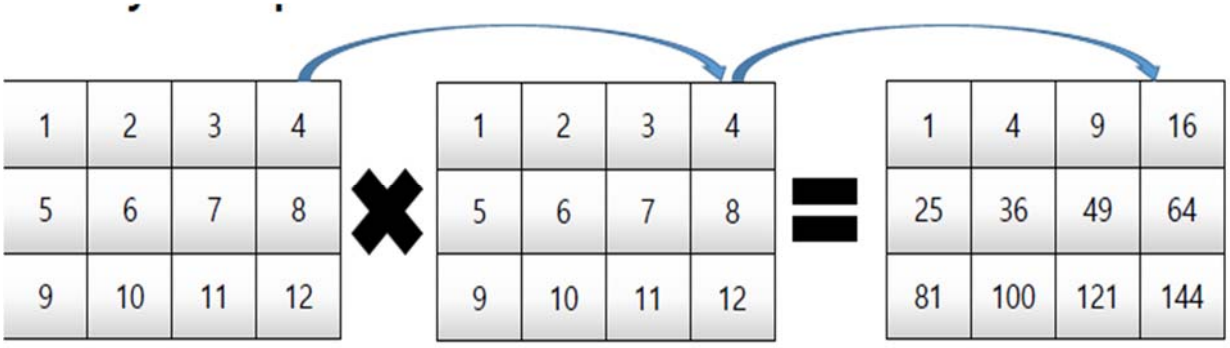
```
left, right = np.hsplit(grid, [2])
print(left)
print(right)
```

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

1.5.3. Numpy operator

Numpy는 array와 정수 array 와 array사이의 기본 연산을 지원
연산자 : + , - , * , /
대응함수 : add() , substrct(), multiply() , divide()

A + B A[0] + B[0] A[1] + B[1] …….



Array 기본 연산자 사용

```
1  x = np.arange(4)
2  print( 'x = ' , x )
3  print( 'x + 3 = ' , x + 3)
4  print( 'x - 3 = ' , x - 3)
5  print( 'x * 2 = ' , x * 3)
6  print( 'x / 3 = ' , x / 3)
7  print( 'x // 3 = ' , x // 3) # 나머지 버림
8  print( 'x ** 3 = ' , x ** 2)
9  print( 'x % 2 = ' , x % 2)
10
```

```
x =  [0 1 2 3]
x + 3 =  [3 4 5 6]
x - 3 =  [-3 -2 -1  0]
x * 2 =  [0 3 6 9]
x / 3 =  [0.         0.33333333 0.66666667 1.        ]
x // 3 =  [0 0 0 1]
x ** 3 =  [0 1 4 9]
x % 2 =  [0 1 0 1]
```

Array간의 기본 연산자 사용

```
x1 = np.arange(1,10).reshape((3,3))
x2 = np.arange(1,10).reshape((3,3))
x1 + x2
```

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

```
x1 * x2
```

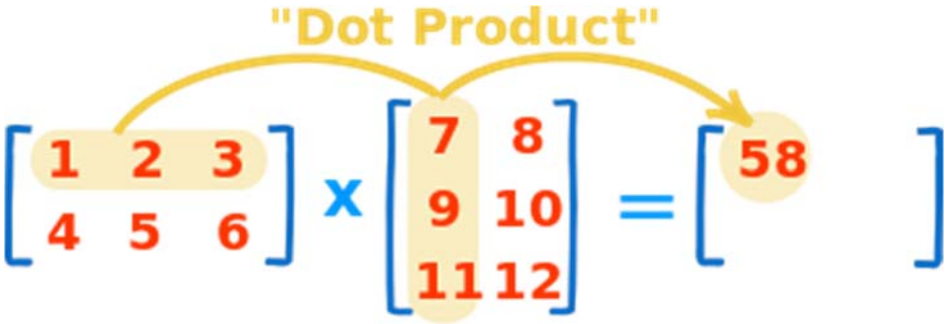
```
array([[ 1,  4,  9],
       [16, 25, 36],
       [49, 64, 81]])
```

Dot product

```
a = np.arange(1,7).reshape(2,3)
b = np.arange(7,13).reshape(3,2)
a.dot(b)
```

```
array([[ 58,  64],
       [139, 154]])
```

- Matrix 기본연산



1.5.1.Broadcasting

- shape가 다른 서로 다른 배열 사이의 연산

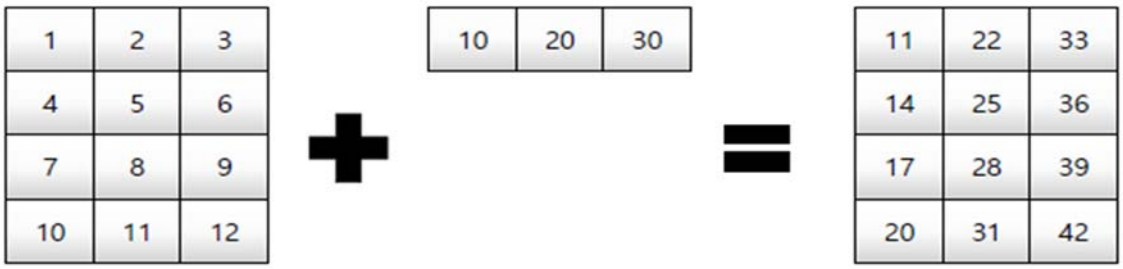
-

```
matrix = np.arange(1,13).reshape(3,4)
matrix
```

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

```
matrix + 100  # matrix + scalar
```

```
array([[101, 102, 103, 104],
       [105, 106, 107, 108],
       [109, 110, 111, 112]])
```

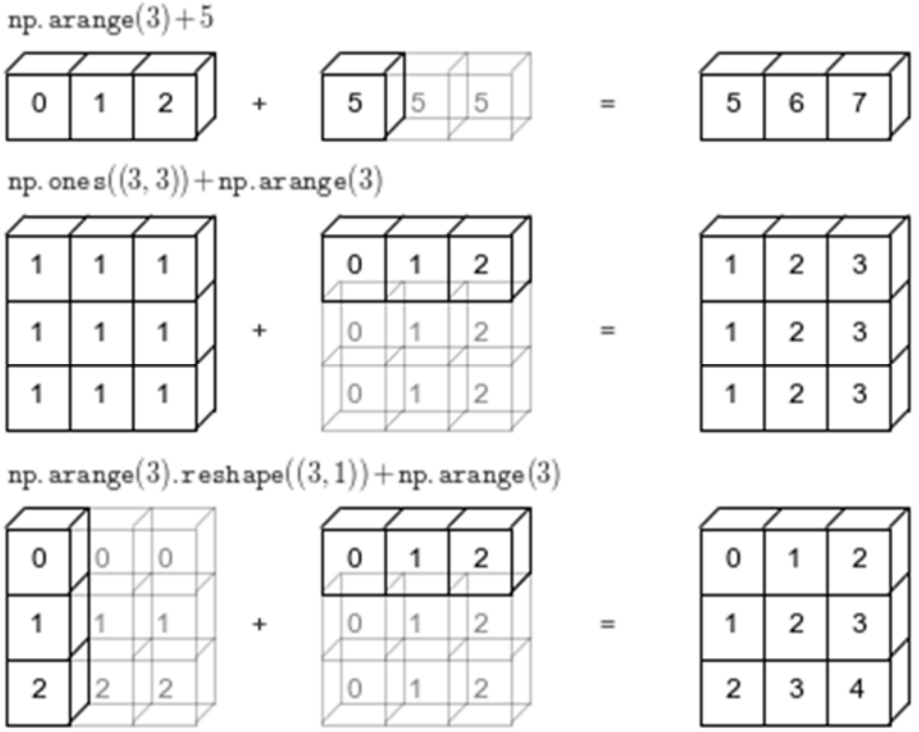


matrix = np.arange(1,13).reshape(3,4)
vector = np.array([10,10,10,10])
matrix + vector

```
array([[11, 12, 13, 14],
       [15, 16, 17, 18],
       [19, 20, 21, 22]])
```

브로드캐스팅 규칙

- 두배열의 차원 수가 다르면 더 작은 차원을 가진 배열의 앞을 1로 채운다
- 두배열의 형상이 어떤 차원에서도 일치하지 않는다면 해당 차원의 형상이 1인 배열이 늘어난다.
- 임의의 차원에서 크기가 일치하지 않고 1도 아니라면 오류가 발생




```
import numpy as np

a = np.array( [ [1,2] , [3,4] ] )
b = np.array([1,2])
c = np.array( [ [1],[2]] )
print( " [1,2] , [3,4]  * 2 \n" , a * 2)
print( " [1,2] , [3,4]  * [1,2] \n" , a * b)
print( " [1,2] , [3,4]  * [1],[2] \n" , a * c)
```

```
[1,2] , [3,4]  * 2
 [[2 4]
 [6 8]]
 [1,2] , [3,4]  * [1,2]
 [[1 4]
 [3 8]]
 [1,2] , [3,4]  * [1],[2]
 [[1 2]
 [6 8]]
```

1.5.2. UFunc

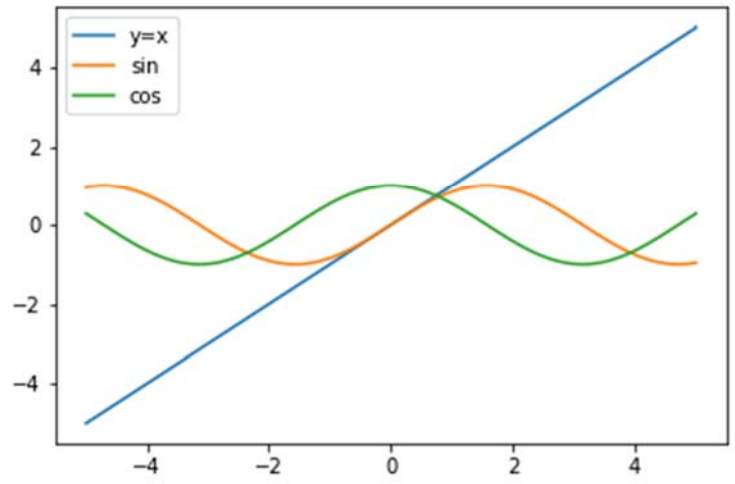
Numpy
유니버설 함수

- ufunc라고 불리는 유니버설 함수.
- ndarray 안에 있는 데이터 원소별로 연산을 수행
- 고속으로 수행할 수 있는 벡터화된 Wrapper 함수

```
x = np. linspace(-5, 5, 20)        # 선형공간 0~10에서 20개 데이터 고르게 추출

np.sqrt(x)                         # 제곱근 연산 √x
np.exp(x)                          # 지수함수 연산 e^x (e=2.718281…)
np.abs(x)                          # 절대값 연산
np.ceil(x)                         # 소수점 올림 연산
np.floor(x)                        # 소수점 내림 연산
np.round(x)                        # 소수점 반올림 연산
np.cos(x), np.sin(x)               # sin함수, cos함수 처리 (그 외 다양한 삼각함수들 존재)
np.power(x, n)                     # n 제곱 처리
np.log(x)                          # 자연로그 연산 (log_e x)
np.log10(x)                        # 로그10 연산  (log_10 x)
np.log2(x)                         # 로그2 연산   (log_2 x)
np.mod(x, n)                       # 각 요소별로 n으로 나눈 뒤의 나머지 추출
```

https://www.slideshare.net/TaeYoungLee1/1-115587182

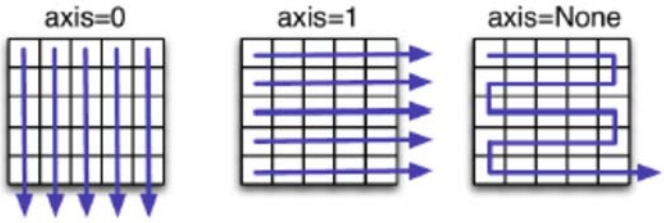
```
import matplotlib.pyplot as plt

x = np.linspace(-5,5,100)
y = x
y_sin = np.sin(x)
y_cos = np.cos(x)

fig= plt.figure()
ax = fig.add_subplot(111) # 1x1

ax.plot(x, y, label='y=x')
ax.plot(x, y_sin, label='sin')
ax.plot(x, y_cos, label='cos')

ax.legend()
plt.show()

```



https://www.slideshare.net/TaeYoungLee1/1-115587182?from_action=save

Numpy 기술 통계 함수

- 집계 합수(aggregate function)
- max (배열의 최대값) min (배열의 최소값) mean (배열의 평균 값)
- numpy의 모든 집계 함수는 AXIS 기준으로 계산이 이루어짐



함수	설명
sum	전체 성분의 합을 계산
mean	전체 성분의 평균을 계산
std, var	전체 성분의 표준편차, 분산을 계산
min, max	전체 성분의 최솟값, 최댓값을 계산
argmin, argmax	전체 성분의 최솟값, 최댓값이 위치한 인덱스를 반환
cumsum	맨 첫번째 성분부터 각 성분까지의 누적합을 계산 (0에서부터 계속 더해짐)
cumprod	맨 첫번째 성분부터 각 성분까지의 누적곱을 계산 (1에서부터 계속 곱해짐)

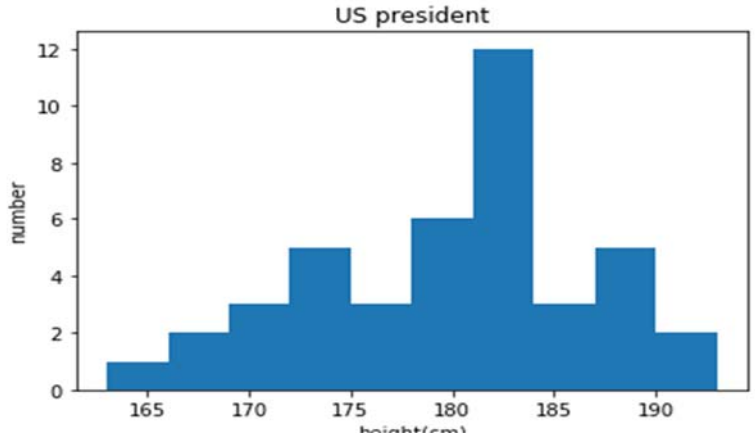
https://www.slideshare.net/TaeYoungLee1/1-115587182?from_action=save

```
1  import pandas as pd
2  data = pd.read_csv('data\president_heights.csv')
3  height = np.array( data['height(cm)'])
4  height
5  # 기본 기술 통계
6  print( height.mean() )
7  print( height.std() )
8  print( height.min() )
9  print( height.max() )
10 print( np.percentile(height,25) ) # 백분위수
11 print( np.percentile(height,75) ) # 중앙값
12 print( np.median(height) )
13
```

```
179.73809523809524
6.931843442745892
163
193
174.25
183.0
182.0
```

```
1  import matplotlib.pyplot as plt
2
3  plt.hist(height) # 히스토그램
4  plt.title("US president")
5  plt.xlabel('height(cm)')
6  plt.ylabel('number')
7
```

```
Text(0, 0.5, 'number')
```



1.5.3. comparisons

Array의 데이터 전부(and) 또는 일부(or)가 조건에 만족 여부 반환

```
import numpy as np
x = np.arange(10)
x
```

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

```
np.any(x < 5) , np.any(x < 0)   #  any : 하나라도 만족하면 true
```

```
(True, False)
```

```
np.all(x > 5) , np.all(x < 10)  # all : 모두 만족 true
```

```
(False, True)
```

비교 연산자 활용

```
x = np.random.randint(1,10,5) # 랜덤 5개
x
```

```
array([7, 5, 8, 3, 8])
```

```
x > 5  # >=
```

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

```
x < 5  # <=
```

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

```
x == 8
```

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

```
x != 8
```

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

배열 비교

```
a = np.random.randint(1,5,5)
b = np.random.randint(1,5,5)
a , b
```

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

```
a == b  #  각각 요소를 비교해서 boolean 타입 반환
```

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

```
(a > b).any() # 요소들중 하나라도 true이면  true
```

```
False
```

index search (where , argmax, argmin)

```
a = np.arange(10)
np.where( a > 3,   1  , 0 )  # where( 조건식, true,false)
```

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

```
np.where( a > 5)  # index return
```

```
(array([6, 7, 8, 9], dtype=int64),)
```

```
a = np.random.randint(1,10,12)
a
```

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

```
# array내 최대값 또는 최소값의 index를 반환함
np.argmax(a) ,  np.argmin(a) #
```

```
(3, 2)
```

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

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

```
np.argmax(a , axis=0) , np.argmax(a , axis=1)
```

```
(array([1, 2, 2, 0], dtype=int64), array([3, 0, 1], dtype=int64))
```

1.5.4. boolean & fancy index

1.5.4.1. boolean index

- numpy는 배열은 특정 조건에 따른 값을 배열 형태로 추출 할 수 있음
- Comparison operation 함수들도 모두 사용가능

```
import numpy as np
a = np.random.randint( 0,10, 15 ).reshape(3,5)
a
```

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

```
a > 5  # 5보다 조건에 해당하는 값들만 true
```

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

```
a[a>5]  # true값들만 추출
```

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

```
b = a < 5
a[b]  #  해당한는 값들만 추출
```

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

```
a = np.random.randint( 0,10, 10 )
a , np.count_nonzero( a > 5)   #  5보다 작은 값은 몇개 ?
```

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

```
np.sum( a > 5 ) # true : 1 false : 0  true 3개
```

```
3
```

1.5.4.2. fancy index

- numpy array 를 index value로 사용해서 값을 추출하는 방법

```
a = np.array([1,2,3,4,5,6,7,8,9,10]) # 원본 데이터
b = np.array([3,6,9])                # 인덱스 배열
a[b]
```

```
array([ 4,  7, 10])
```

```
a.take(b)      # 같은 결과
```

```
array([ 4,  7, 10])
```

```
x =  np.random.randint(100, size = 10)
x
```

```
array([71, 17, 63, 42, 93, 55, 24,  5, 51, 51])
```

```
[x[3], x[6], [9]] # 세개의 다른 요소 접근
```

```
[42, 24, [9]]
```

```
idx = [3,6,9]  # 인덱스 리스트
x[idx]
```

```
array([42, 24, 51])
```

```
idx = np.array([[3,7],
                [4,5]])
x[idx]       # 결과는 항상 인덱스 배열의 모양으로...
```

```
array([[42,  5],
       [93, 55]])
```

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

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

```
row = np.array([0,1,2])
col = np.array([0,1,3])
x[row,col]    # [0,2] [1,1] [2,3]
```

```
array([ 2,  5, 11])
```

```
x [ row[0] , col] # 0,2, 0,1, 0,3  브로드 캐스팅
```

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