

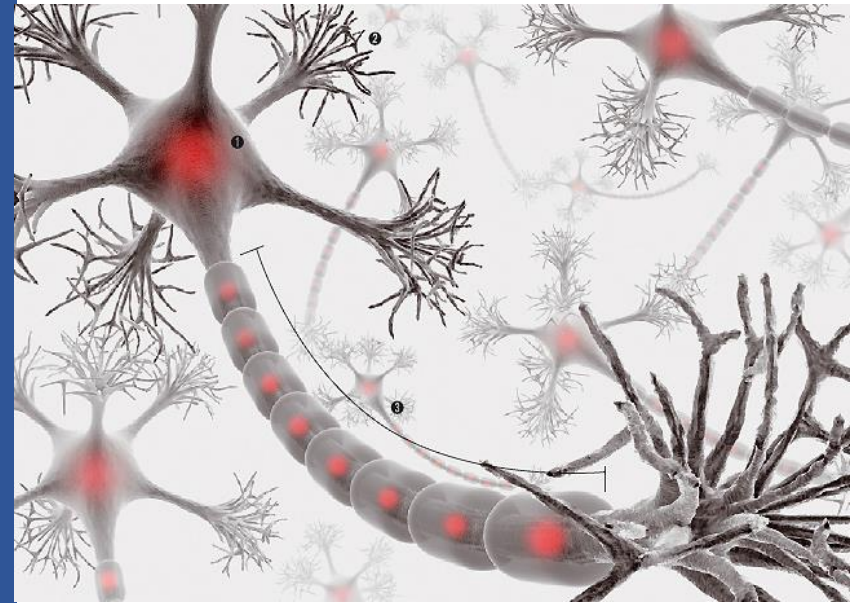
넘파이 (Numpy)

학습 목표

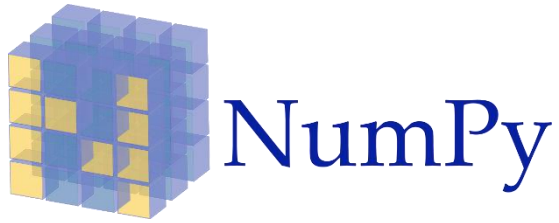
- 다차원 배열 및 행렬 연산을 지원하는 Numpy 라이브러리의 사용법을 이해한다.

주요 내용

- 넘파이 (Numpy)
- 배열 생성
- 배열 관리
- 배열 연산
- 선형대수 연산



넘파이 (Numpy)



NumPy는 행렬이나 대규모 다차원 배열을 쉽게 처리할 수 있도록 지원하는 파이썬의 라이브러리이다.

- 벡터 산술 연산을 위한 빠른 **다차원 배열**
- 데이터 배열 전체의 빠른 연산을 위한 **표준 수학 함수**
- **선형 대수**
- 정렬, 중복 제거, **집합 연산**
- **통계** 및 데이터 집계

다차원 배열

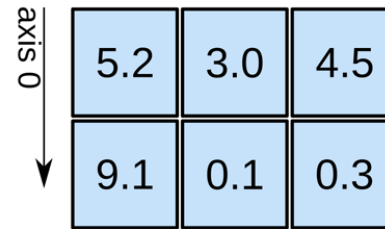
1D array



axis 0 →

shape: (4,)

2D array

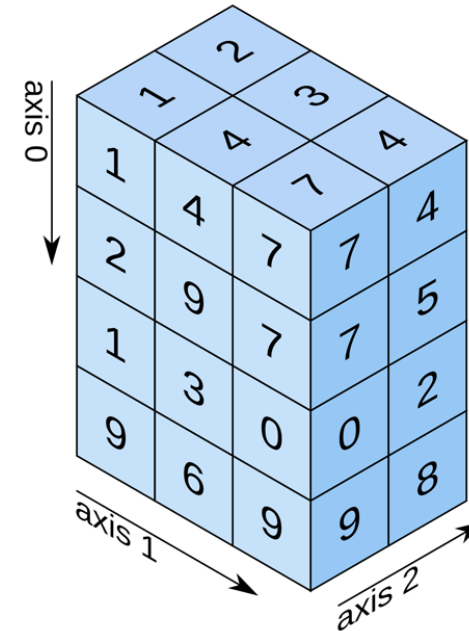


axis 0 ↓

axis 1 →

shape: (2, 3)

3D array



shape: (4, 3, 2)

<https://medium.com/datadriveninvestor/artificial-intelligence-series-part-2-numpy-walkthrough-64461f26af4f>

Numpy가 빠른 이유

Python

```
>>> def sum_trad():
>>>     start = time.time()
>>>     X = range(10000000)
>>>     Y = range(10000000)
>>>     Z = []
>>>     for i in range(len(X)):
>>>         Z.append(X[i] + Y[i])
>>>     return time.time() - start
```

```
>>> print 'time sum:',sum_trad(),' time sum NumPy:',sum_NumPy()
time sum: 2.1142539978 time sum NumPy: 0.0807049274445
```

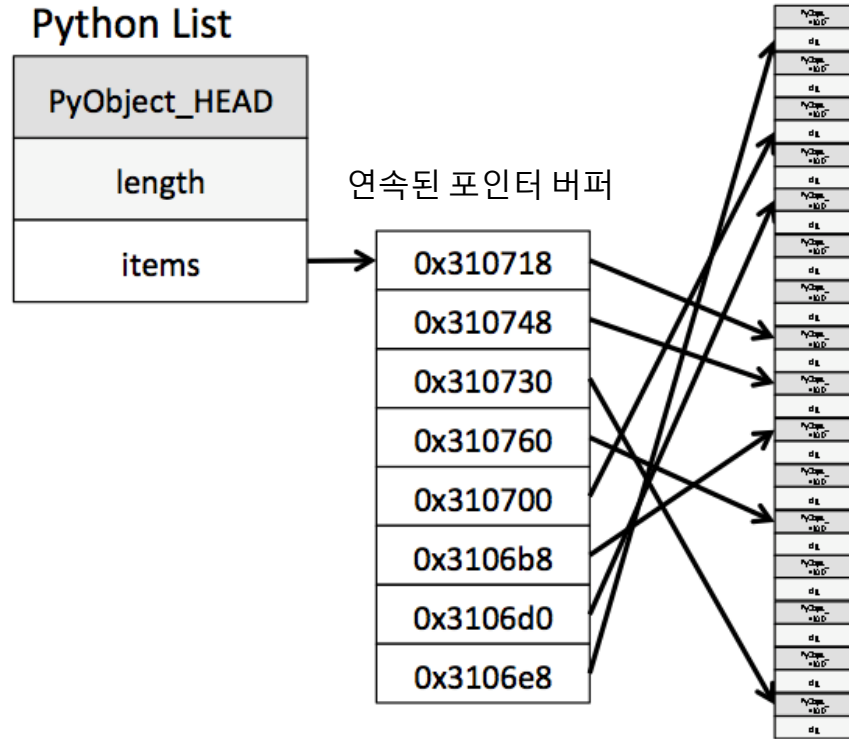
NumPy

```
>>> def sum_NumPy():
>>>     start = time.time()
>>>     X = np.arange(10000000)
>>>     Y = np.arange(10000000)
>>>     Z = X + Y
>>>     return time.time() - start
```

26배 이상의 성능 차이

Numpy가 빠른 이유

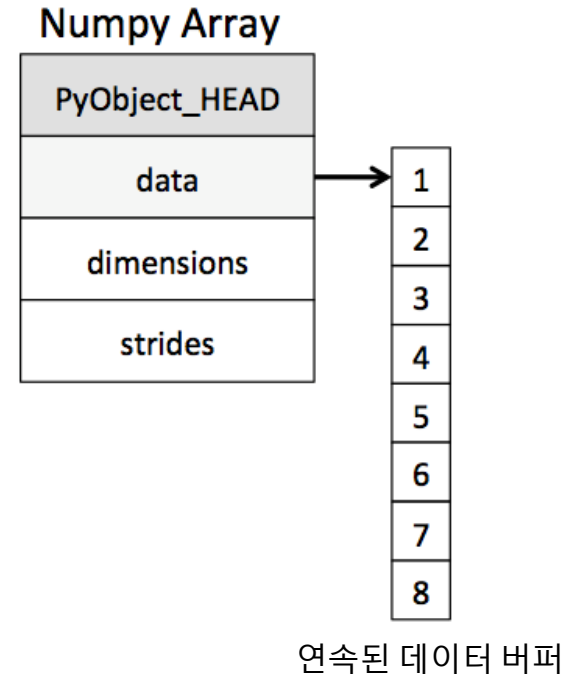
Python List



- 데이터에 접근하는데 여러 단계를 거침

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

Numpy Array



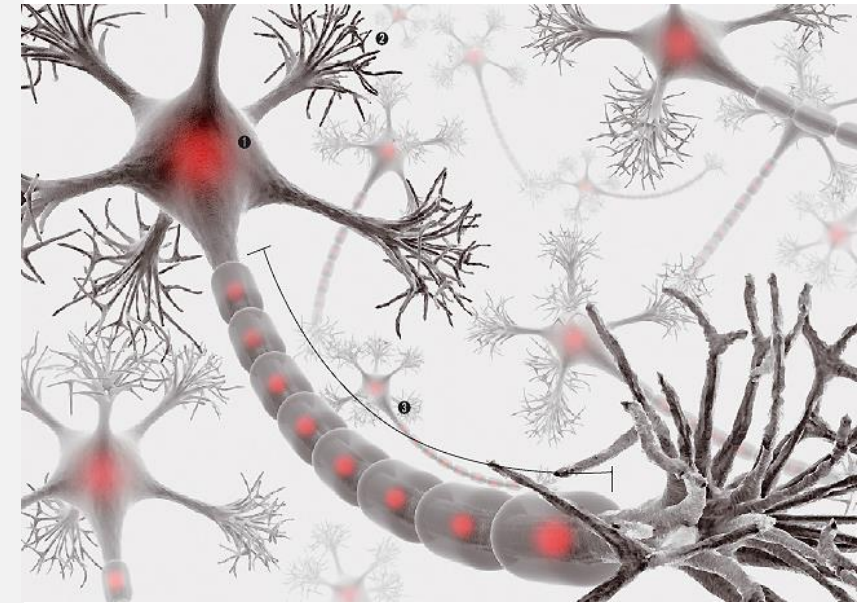
- C Array를 파이썬 객체화 함
- 연속된 데이터 버퍼를 관리
- 데이터를 접근하는데 한 단계만 거침

dtypes

	Data type	Description	Value/Range
Boolean	bool_		True or False
Integer	int_	Default, C long과 동일	int64 or int32.
	intc	C int와 동일	int32 or int64.
	intp	Indexing에 사용, C ssize_t와 동일	int32 or int64.
	int8	Byte.	-128 to 127
	int16		-32768 to 32767
	int32		-2147483648 to 2147483647
	int64		-9223372036854775808 to 9223372036854775807
Unsigned integer	uint8		0 to 255
	uint16		0 to 65535
	uint32		0 to 4294967295
	uint64		0 to 18446744073709551615
Float	float_	float64.	
	float16	Half precision	sign bit, 5 bits exponent, 10 bits mantissa
	float32	Single precision	sign bit, 8 bits exponent, 23 bits mantissa
	float64	Double precision	sign bit, 11 bits exponent, 52 bits mantissa
Complex	complex_	complex128	
	complex64		two 32-bit floats (real and imaginary components)
	complex128		two 64-bit floats (real and imaginary components)

1 배열 생성

1. 단일 값으로 초기화
2. 랜덤 초기화
3. 리스트에서 생성
4. 배열 복사
5. 단위 행렬, 대각 행렬
6. 영행렬, 1행렬

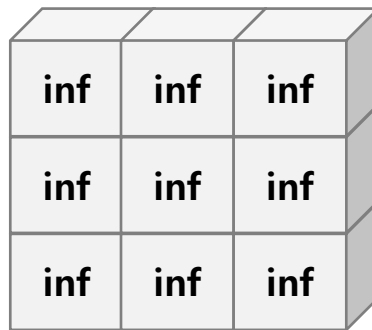


단일 값으로 초기화

단일 값으로 초기화된 배열 생성

```
>>> import numpy as np
>>> np.full((3, 3), np.inf)
array([[ inf,  inf,  inf],
       [ inf,  inf,  inf],
       [ inf,  inf,  inf]])
>>> np.full((3, 3), 10.1)
array([[ 10.1, 10.1, 10.1],
       [ 10.1, 10.1, 10.1],
       [ 10.1, 10.1, 10.1]])
```

`np.full((3, 3), np.inf)`



inf	inf	inf
inf	inf	inf
inf	inf	inf

배열을 단일 값으로 리셋

```
>>> arr = np.array([10, 20, 33], float)
>>> arr.fill(1)
>>> arr
array([ 1.,  1.,  1.])
```


랜덤 초기화

정수 순열로 초기화된 배열 생성

```
>>> np.random.permutation(3)
array([2, 0, 1])
```

정규분포로 초기화된 배열 생성

```
>>> np.random.normal(0,1,5)
array([-0.66494912, 0.7198794 , -0.29025382, 0.24577752, 0.23736908])
```

• np.random.normal(평균, 표준편차, 배열 모양)

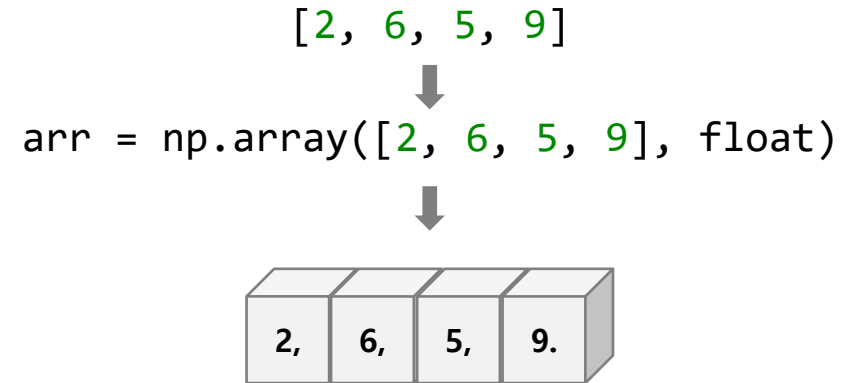
균등분포로 초기화된 배열 생성

```
>>> np.random.random(5)
array([ 0.48241564, 0.24382627, 0.25457204, 0.9775729, 0.61793725])
```

리스트에서 생성

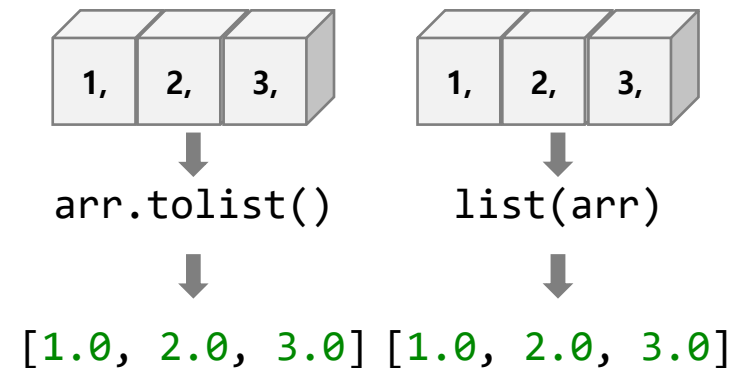
리스트에서 배열 생성

```
>>> arr = np.array([2, 6, 5, 9], float)
>>> arr
array([ 2.,  6.,  5.,  9.])
>>> type(arr)
<type 'numpy.ndarray'>
```



배열에서 리스트로 변환

```
>>> arr = np.array([1, 2, 3], float)
>>> arr.tolist()
[1.0, 2.0, 3.0]
>>> list(arr)
[1.0, 2.0, 3.0]
```

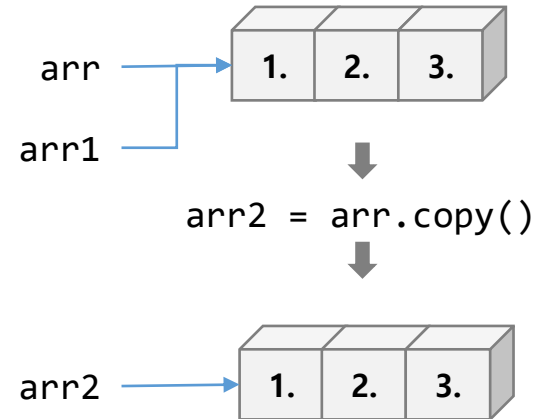


배열 복사



배열을 새 변수에 지정할 경우 메모리에 복사본(copy)을 만들지 않고 새 이름에 원래의 객체를 연결한다.

```
>>> arr = np.array([1, 2, 3], float)
>>> arr1 = arr          # 주소만 복사
>>> arr2 = arr.copy()   # 배열도 복사
>>> arr[0] = 0
>>> arr
array([0., 2., 3.])
>>> arr1
array([0., 2., 3.])
>>> arr2
array([1., 2., 3.])
```

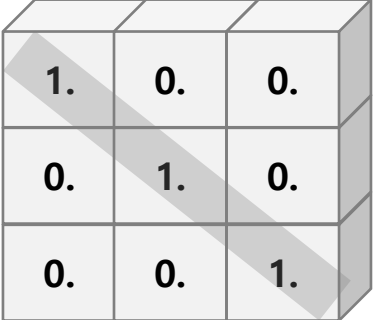


단위 행렬

단위 행렬 생성

```
>>> np.identity(3, dtype=int)
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
>>> np.identity(3) #default data type is float
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

np.identity(3)



A 3x3 grid representing the identity matrix. The diagonal elements are 1. and the off-diagonal elements are 0. A gray diagonal band highlights the path from the top-left to the bottom-right.

1.	0.	0.
0.	1.	0.
0.	0.	1.

대각행렬

대각 행렬 생성

```
>>> np.eye(3, k=1, dtype=float)
array([[ 0.,  1.,  0.],
       [ 0.,  0.,  1.],
       [ 0.,  0.,  0.]])
```

k는 대각 인덱스로
주 대각 main diagonal은 0,
상 대각 upper diagonal은 양수,
하 대각 lower diagonal은 음수다

```
np.eye(3, k=1, dtype=float)
```

0.	1.	0.
0.	0.	1.
0.	0.	0.

영행렬, 1행렬

영행렬 생성

```
>>> np.zeros(6, dtype=int)
array([0, 0, 0, 0, 0, 0])
```

np.zeros(6, dtype=int)

0	0	0	0	0	0
---	---	---	---	---	---

1로 채워진 행렬 생성

```
>>> np.ones((2,3), dtype=float)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

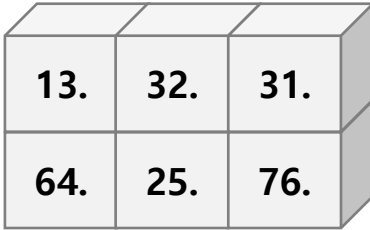
np.ones((2,3), dtype=float)

1.	1.	1.
1.	1.	1.

영행렬, 1행렬

주어진 배열과 동일한 차원의 영행렬, 1행렬 생성

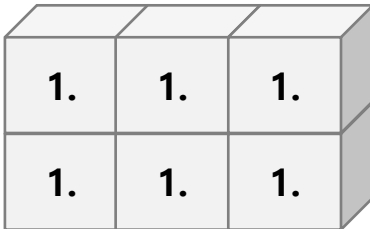
```
>>> arr = np.array([[13, 32, 31], [64, 25, 76]], float)
>>> np.zeros_like(arr)
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> np.ones_like(arr)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```



13.	32.	31.
64.	25.	76.



np.ones_like(arr)



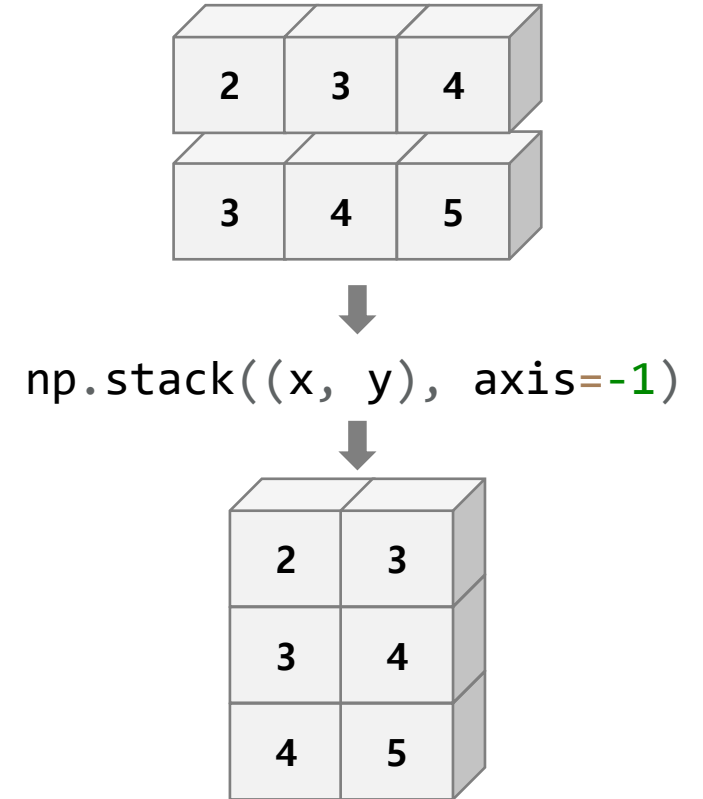
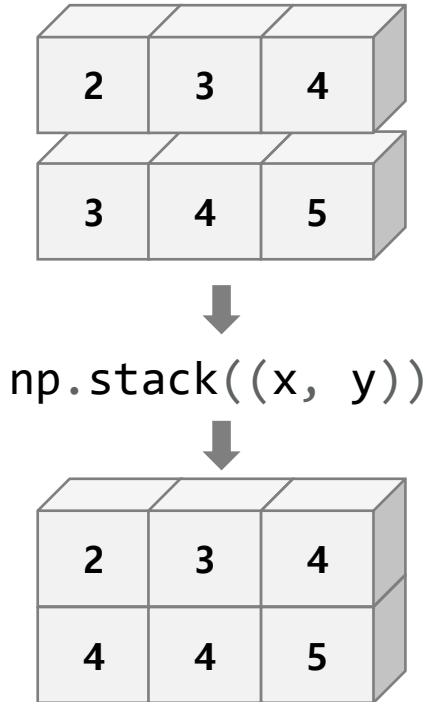
1.	1.	1.
1.	1.	1.

배열 쌓기

배열 쌓기

```
>>> x = np.array([2, 3, 4])
>>> y = np.array([3, 4, 5])
>>> np.stack((x, y))
array([[2, 3, 4],
       [3, 4, 5]])
```

```
>>> np.stack((x, y), axis=-1)
array([[2, 3],
       [3, 4],
       [4, 5]])
```



axis로 지정된 새로운 차원을 생성해서 배열들을 결합

- axis=-1 은 마지막 차원

2차원 배열 랜덤 초기화

균등분포로 초기화된 2차원 배열 생성

```
>>> np.random.rand(2,3)
array([[ 0.36152029,  0.10663414,  0.64622729],
       [ 0.49498724,  0.59443518,  0.31257493]])
```

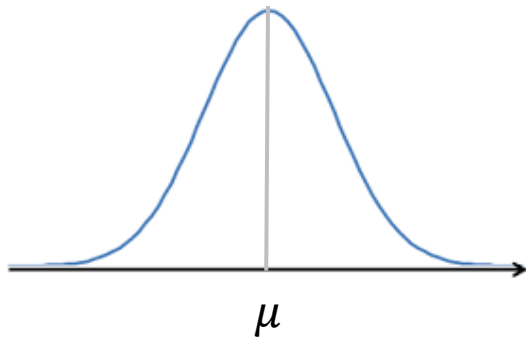
다변량 정규분포로 초기화된 2차원 배열 생성

```
>>> np.random.multivariate_normal([10, 0], [[3, 1], [1, 4]], size=[5,])
array([[ 11.8696466 , -0.99505689],
       [ 10.50905208,  1.47187705],
       [  9.55350138,  0.48654548],
       [ 10.35759256, -3.72591054],
       [ 11.31376171,  2.15576512]])
```

- 차원 : 2차원
- 샘플 수 : 5개
- [10,0]은 2차원 평균 벡터
- [[3, 1], [1, 4]]는 2차원 공분산 행렬

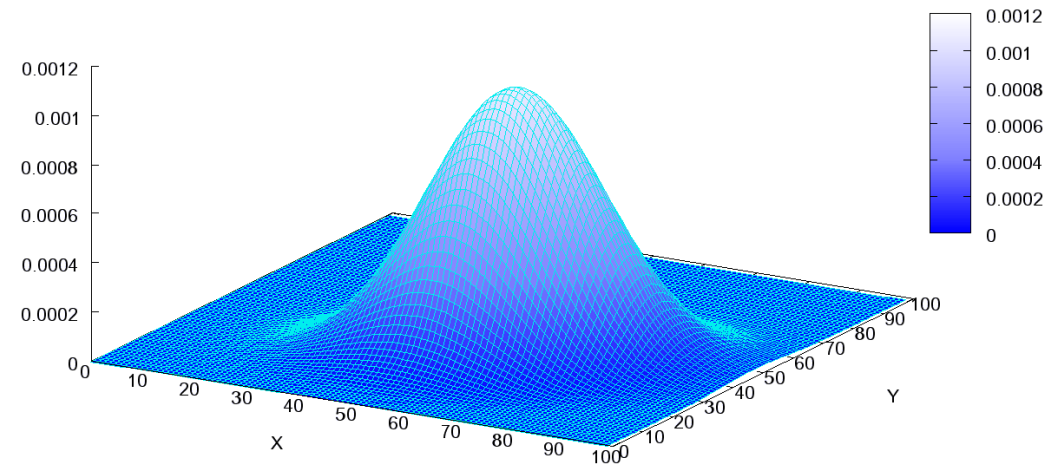
참고 다변량 가우시안 분포

Univariate Gaussian Distribution



$$N(x \mid \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Multivariate Gaussian Distribution



$$N(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sigma\sqrt{2\pi^k |\boldsymbol{\Sigma}|}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu})}$$

참고 공분산 행렬

n차원 변수 $x = (x_1, x_2, \dots, x_n)$ 의 공분산 행렬

$$\Sigma = \begin{bmatrix} \text{Cov}(x_1, x_1) & \cdots & \text{Cov}(x_1, x_n) \\ \vdots & \ddots & \vdots \\ \text{Cov}(x_n, x_1) & \cdots & \text{Cov}(x_n, x_n) \end{bmatrix}$$

$$= \mathbb{E} [(x - \mu) (x - \mu)^T]$$

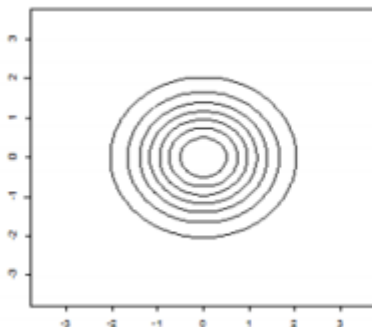
- 대칭 행렬
- 대각 방향으로서는 각 변수의 분산

μ : 평균

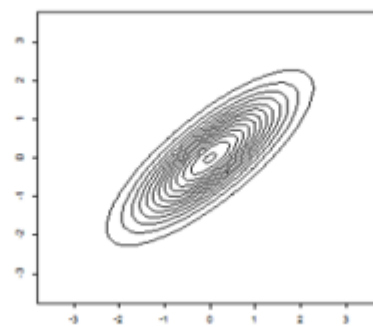
2차원 공분산 행렬 예제

$$\Sigma = \begin{bmatrix} \text{Cov}(x_1, x_1) & \text{Cov}(x_1, x_2) \\ \text{Cov}(x_1, x_2) & \text{Cov}(x_2, x_2) \end{bmatrix}$$

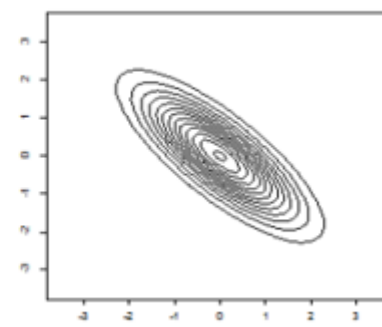
$$\mu = (0, 0)$$



$$\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$



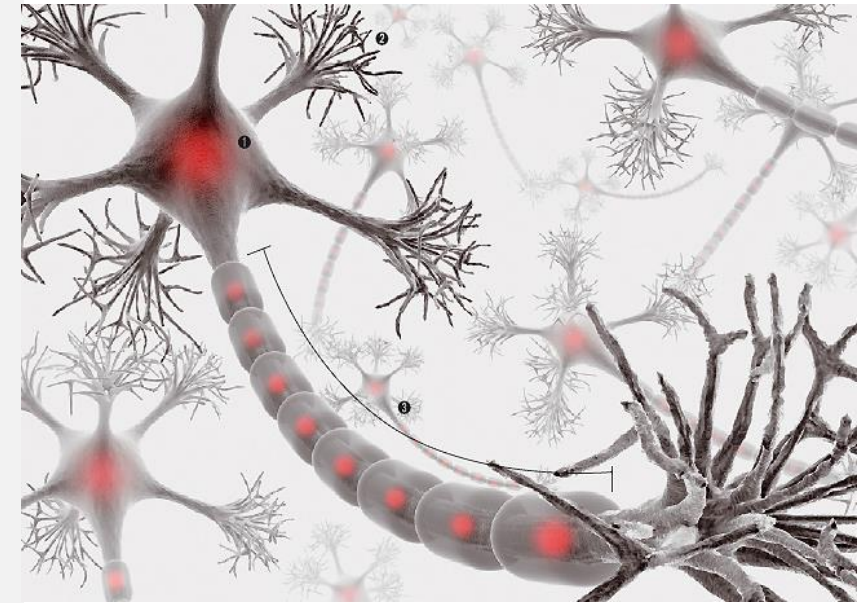
$$\Sigma = \begin{bmatrix} 1 & 0.8 \\ 0 & 1 \end{bmatrix}$$



$$\Sigma = \begin{bmatrix} 1 & -0.8 \\ 0 & 1 \end{bmatrix}$$

2 배열 관리

1. 배열 읽기/쓰기
2. 중복 제거
3. 정렬/섞기
4. 배열 슬라이싱
5. 1차원 배열로 펴기
6. 배열의 크기와 데이터 타입
7. 재배열
8. 전치 행렬
9. 차원 늘리기
10. 배열 결합
11. 문자열 변환

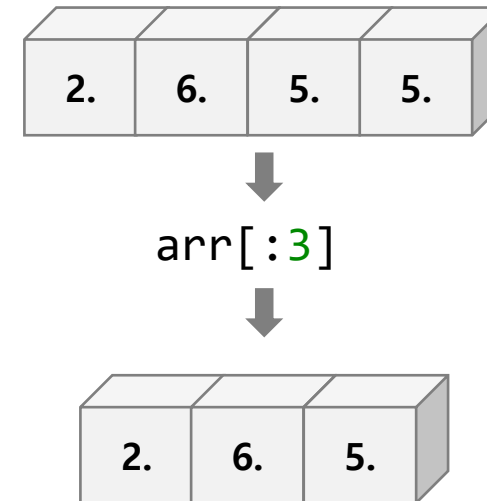


배열 읽기/쓰기

```
>>> arr = np.array([2., 6., 5., 5.])
>>> arr[:3]
array([ 2.,  6.,  5.])

>>> arr[3]
5.0

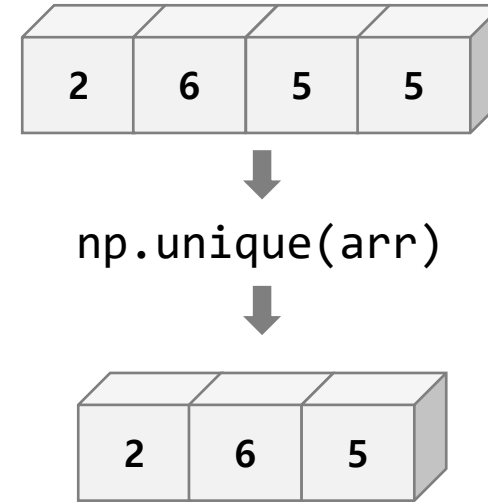
>>> arr[0] = 5.
>>> arr
array([ 5.,  6.,  5.,  5.])
```



중복 제거

중복 제거

```
>>> arr = np.array([2., 6., 5., 5.])  
>>> np.unique(arr)  
array([ 2.,  6.,  5.])
```



정렬/섞기

정렬

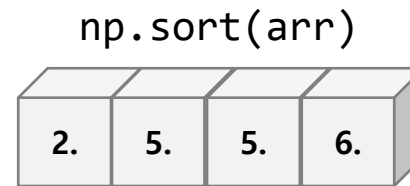
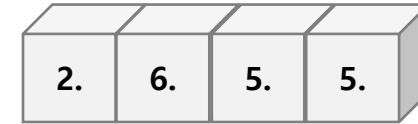
```
>>> arr = np.array([2., 6., 5., 5.])
>>> np.sort(arr)
array([ 2.,  5.,  5.,  6.])
```

정렬해서 인덱스 배열 생성

```
>>> np.argsort(arr)
array([0, 2, 3, 1])
```

랜덤하게 섞기

```
>>> np.random.shuffle(arr)
>>> arr
array([ 2.,  5.,  6.,  5.])
```



배열 비교

```
>>> np.array_equal(arr, np.array([1, 3, 2]))
False
```

배열 슬라이싱

슬라이싱

```
>>> matrix = np.array([[ 4., 5., 6.], [2., 3., 6.]], float)
```

```
>>> matrix
array([[ 4., 5., 6.],
       [ 2., 3., 6.]])
```

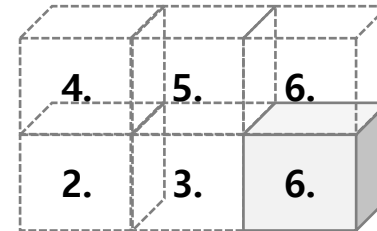
```
>>> arr[1:2,2:3]
array([[ 6.]])
```

```
>>> arr[1,:]
array([2, 3, 6])
```

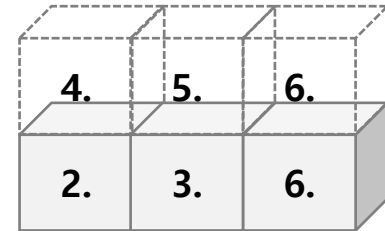
```
>>> arr[:,2]
array([ 6., 6.])
```

```
>>> arr[-1:,-2:]
array([[ 3., 6.]])
```

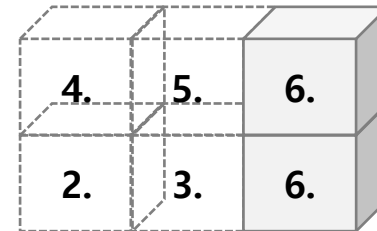
arr[1:2,2:3]



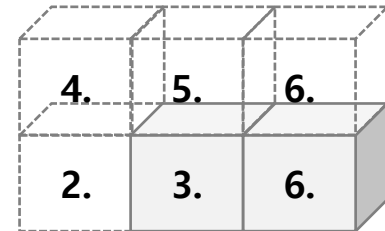
arr[1,:]



arr[:,2]



arr[-1:,-2:]

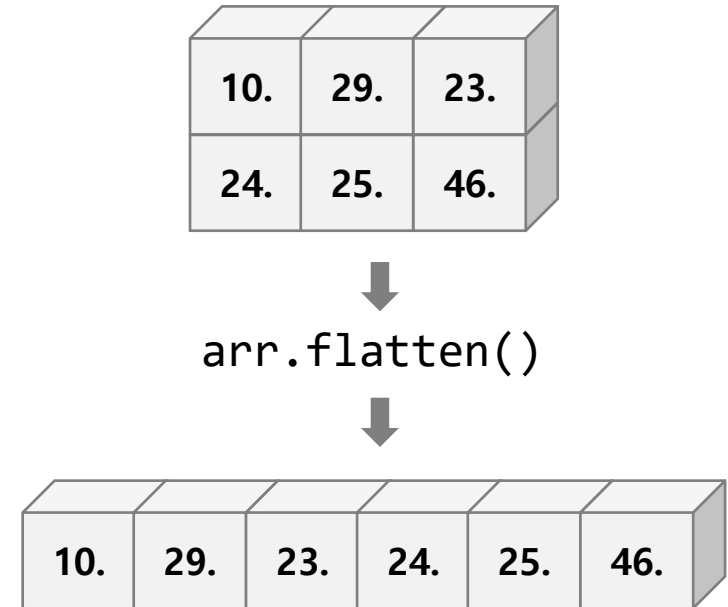


1차원 배열로 펴기

Flattening

```
>>> arr = np.array([[10, 29, 23], [24, 25, 46]], float)
>>> arr
array([[ 10.,  29.,  23.],
       [ 24.,  25.,  46.]])
>>> arr.flatten()
array([ 10.,  29.,  23.,  24.,  25.,  46.]])
```

다차원 배열에서 1차원 배열로 펴기



배열의 크기와 데이터 타입

배열의 모양 확인

```
>>> arr.shape  
(2, 3)
```

1차원의 길이

```
>>> arr = np.array([[ 4., 5., 6.], [ 2., 3., 6.]], float)  
>>> len(arr)  
2
```

데이터 타입 확인

```
>>> arr.dtype  
dtype('float64')
```

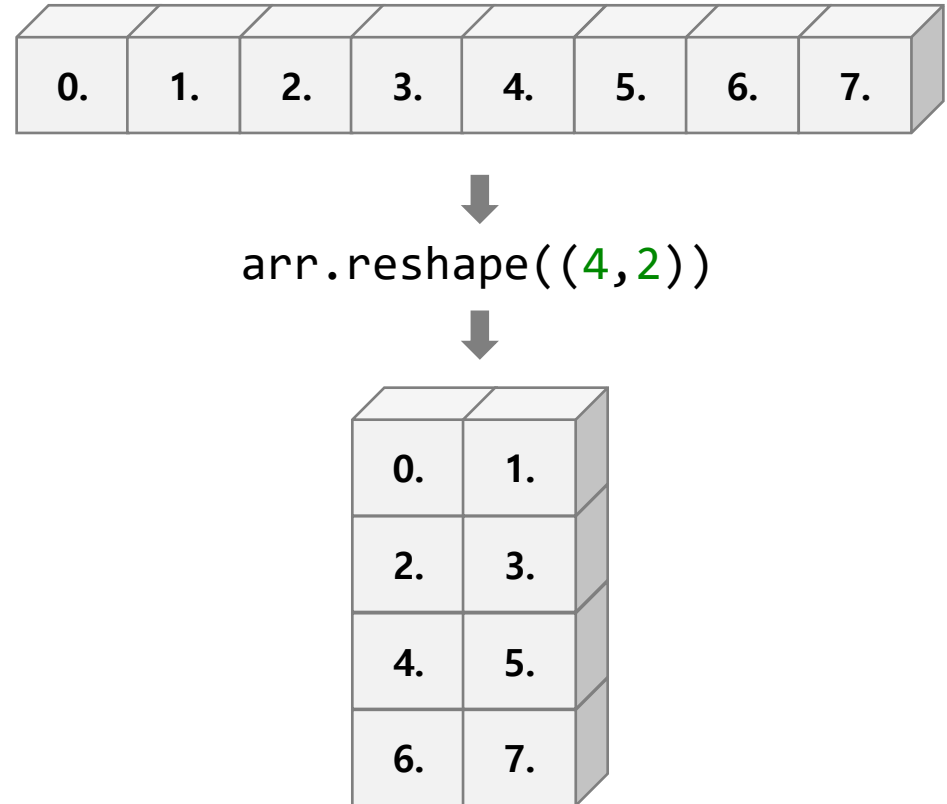
데이터 타입 변환

```
>>> int_arr = arr.astype(np.int32)  
>>> int_arr.dtype  
dtype('int32')
```

재배열

배열의 재배열

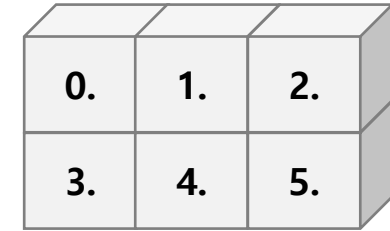
```
>>> arr = np.array(range(8), float)
>>> arr
array([ 0., 1., 2., 3., 4., 5., 6., 7.])
>>> arr = arr.reshape((4,2))
>>> arr
array([[ 0., 1.],
       [ 2., 3.],
       [ 4., 5.],
       [ 6., 7.]])
>>> arr.shape
(4, 2)
```



전치 행렬 (1/2)

전치 행렬 구하기

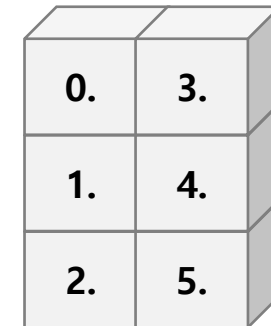
```
>>> arr = np.array(range(6), float).reshape((2, 3))
>>> arr
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
>>> arr.transpose()
array([[ 0.,  3.],
       [ 1.,  4.],
       [ 2.,  5.]])
```



0.	1.	2.
3.	4.	5.



arr.transpose()

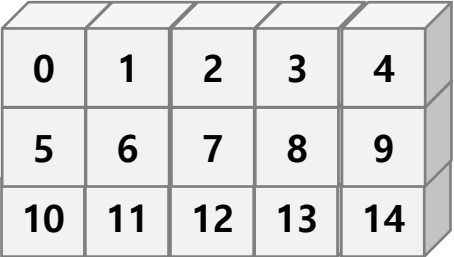


0.	3.
1.	4.
2.	5.

전치 행렬 (2/2)

행렬의 T 속성으로 전치하기

```
>>> matrix = np.arange(15).reshape((3, 5))
>>> matrix
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> matrix.T
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  6, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```




A 3x5 matrix visualization with rows and columns. The values are as follows:

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



matrix.T



A 5x3 matrix visualization with rows and columns. The values are as follows:

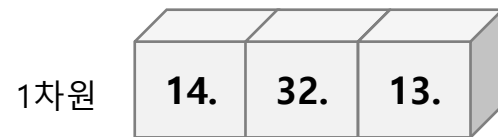
0	5	10
1	6	11
2	7	12
3	8	13
4	9	14

차원 늘리기

newaxis로 차원 늘리기

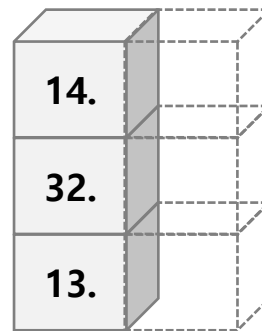
```
>>> arr = np.array([14, 32, 13], float)
>>> arr
array([ 14.,  32.,  13.])
>> arr[:,np.newaxis]
array([[ 14.],
       [ 32.],
       [ 13.]])
>>> arr[:,np.newaxis].shape
(3,1)

>>> arr[np.newaxis,:]
array([[ 14.,  32.,  13.]])
>>> arr[np.newaxis,:].shape
(1,3)
```



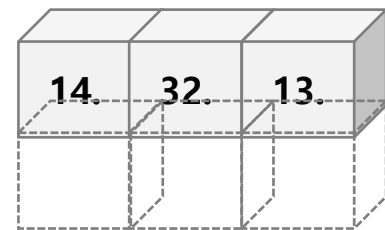
원래 데이터는 1차원에
두고 2차원을 추가

`arr[:,np.newaxis]`



원래 데이터는 2차원에
두고 1차원을 추가

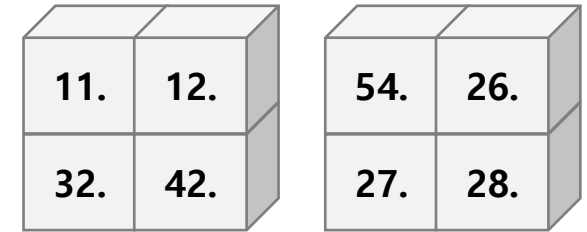
`arr[np.newaxis,:]`



newaxis로 길이가 1인 축을 추가

배열 결합

```
>>> arr1 = np.array([[11, 12], [32, 42]], float)
>>> arr2 = np.array([[54, 26], [27, 28]], float)
>>> np.concatenate((arr1, arr2))
array([[ 11.,  12.],
       [ 32.,  42.],
       [ 54.,  26.],
       [ 27.,  28.]])
>>> np.concatenate((arr1, arr2), axis=0)
array([[ 11.,  12.],
       [ 32.,  42.],
       [ 54.,  26.],
       [ 27.,  28.]])
>>> np.concatenate((arr1, arr2), axis=1)
array([[ 11.,  12.,  54.,  26.],
       [ 32.,  42.,  27.,  28.]])
```



np.concatenate((arr1, arr2))

1차원 방향으로 결합

2차원 방향으로 결합



문자열 변환

배열에서 문자열로 변환

```
>>> arr = np.array([10, 20, 30], float)
>>> str = arr.tostring()
>>> str
```

```
'\x00\x00\x00\x00\x00\x00$@\x00\x00\x00\x00\x00\x004@\x00\x00\x00\x00\x00\x00\x00>@'
```

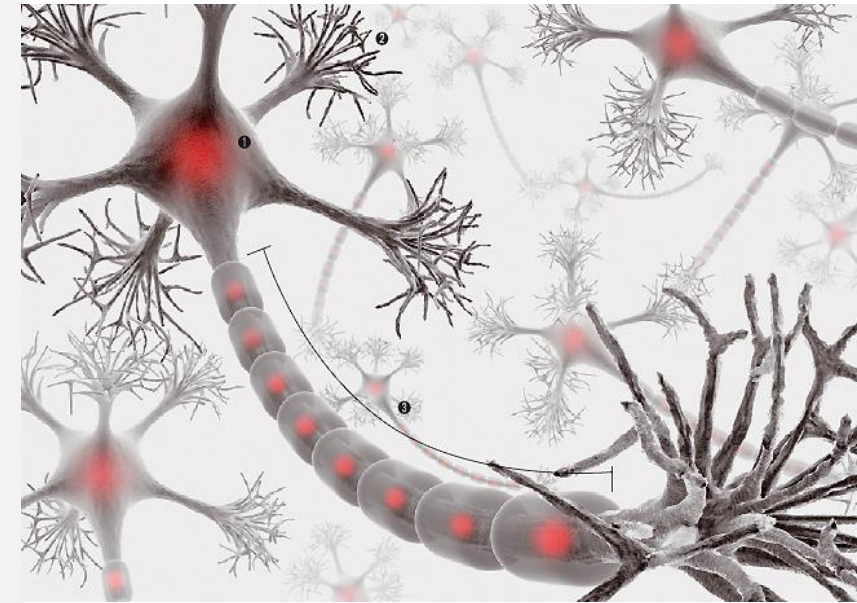
큰 데이터를 저장할 때는 문자열로 변환해서 바이너리 파일로 저장

문자열에서 배열로 변환

```
>>> np.fromstring(str)
array([ 10., 20., 30.])
```


3 배열 연산

1. 산술 연산
2. 브로드캐스팅
3. 배열 쿼리
4. 배열 수정 쿼리



산술연산

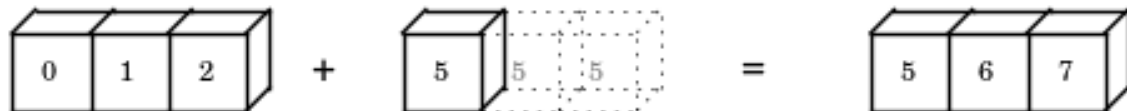
```
>>> arr1 = np.array([1,2,3], float)
>>> arr2 = np.array([1,2,3], float)
>>> arr1 + arr2
array([2., 4., 6.])
>>> arr1-arr2
array([0., 0., 0.])
>>> arr1 * arr2
array([1., 4., 9.])
>>> arr2 / arr1
array([1., 1., 1.])
>>> arr1 % arr2
array([0., 0., 0.])
>>> arr2**arr1
array([1., 4., 27.])
```

연산 시 배열의 크기를 자동으로 맞출 수 없을 때

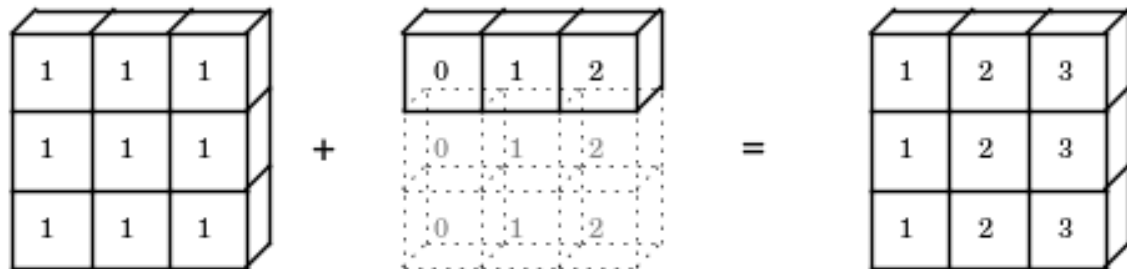
```
>>> arr1 = np.array([1,2,3], float)
>>> arr2 = np.array([1,2], float)
>>> arr1 + arr2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be
broadcast to a single shape
```

브로드캐스팅

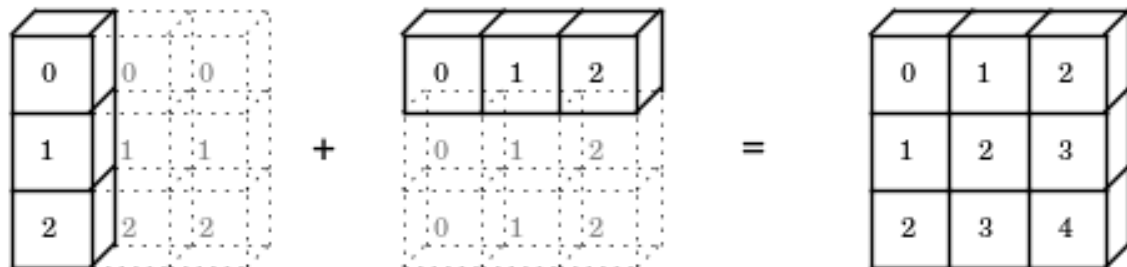
`np.arange(3) + 5`



`np.ones((3, 3)) + np.arange(3)`



`np.arange(3).reshape((3, 1)) + np.arange(3)`



브로드캐스팅 알고리즘

배열 A, B 연산 시

for 두 배열의 끝 쪽 차원에서 비교해서 앞쪽 방향으로 진행:

If A, B 중 차원이 존재하지 않는 배열이 있으면:

차원이 없는 배열에 차원 추가

If A, B가 차원의 크기가 다르면:

If A, B 중 차원의 크기 == 1:

크기가 1인 배열을 다른 배열의 크기로 확장

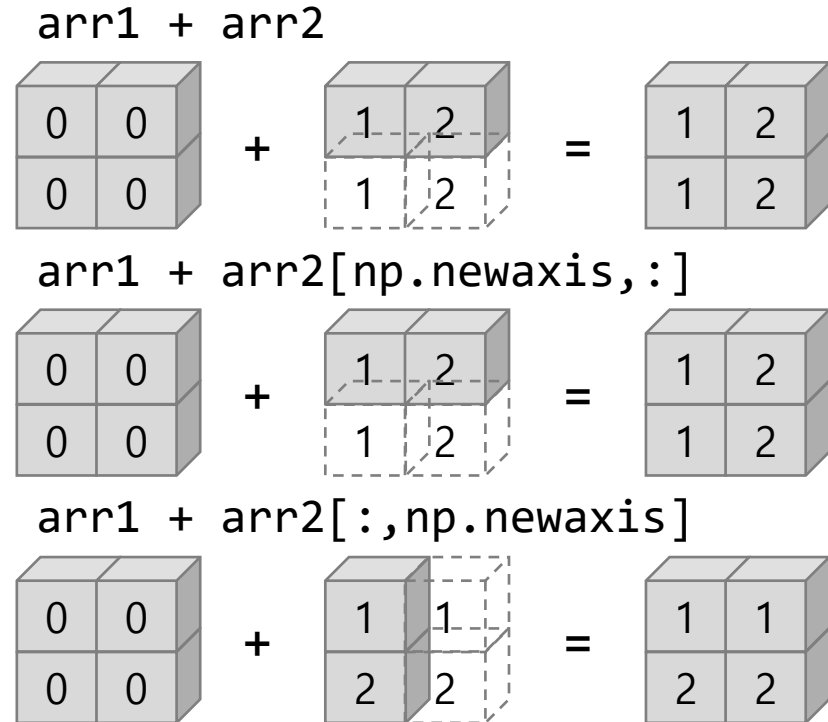
else

브로드캐스팅 에러

브로드캐스팅

배열의 브로드캐스팅 방식을 명시하고 싶다면 `newaxis` 상수를 이용해서 확장해야 할 축을 지정

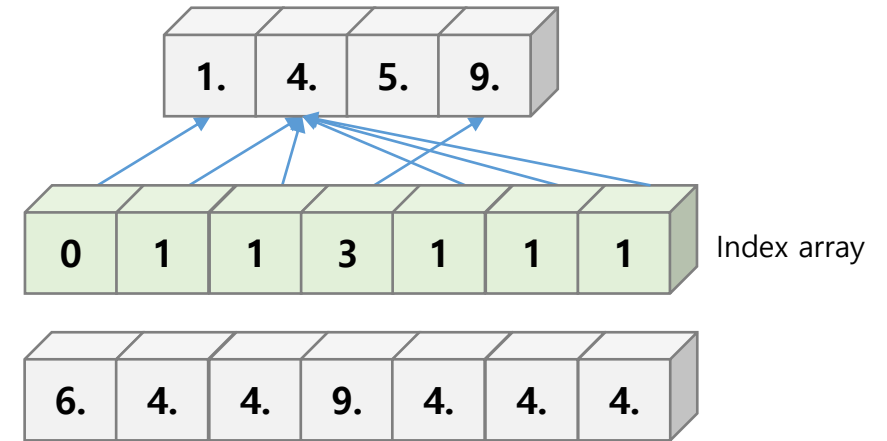
```
>>> arr1 = np.zeros((2,2), float)
>>> arr2 = np.array([1., 2.], float)
>>> arr1
array([[ 0.,  0.], [ 0.,  0.]])
>>> arr2
array([1., 2.])
>>> arr1 + arr2
array([[1., 2.], [1., 2.]])
>>> arr1 + arr2[np.newaxis,:]
array([[1., 2.], [1., 2.]])
>>> arr1 + arr2[:,np.newaxis]
array([[1., 1.], [2., 2.]])
```



배열 쿼리

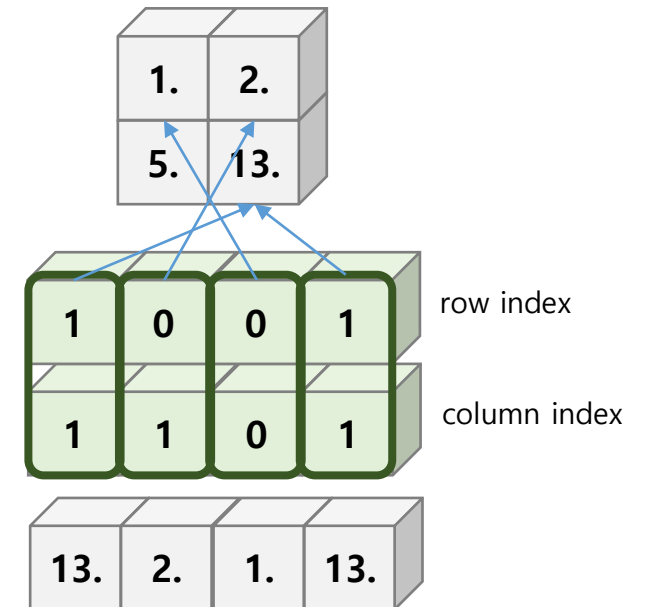
인덱스 배열로 쿼리

```
>>> arr1 = np.array([1, 4, 5, 9], float)
>>> arr2 = np.array([0, 1, 1, 3, 1, 1, 1], int)
>>> arr1[arr2]
array([ 1.,  4.,  4.,  9.,  4.,  4.,  4.])
>>> arr1[[0, 1, 1, 3, 1]] # 리스트로 쿼리
array([1., 4., 4., 9., 4.])
```



다차원 배열 인덱스 배열로 쿼리

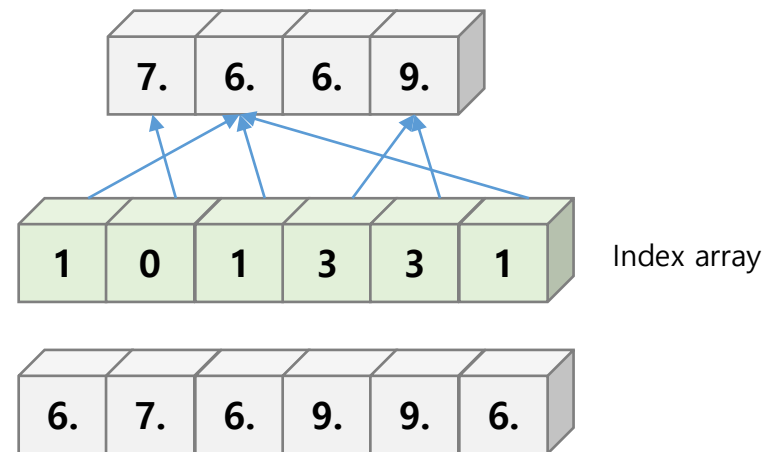
```
>>> arr1 = np.array([[1, 2], [5, 13]], float)
>>> arr2 = np.array([1, 0, 0, 1], int)
>>> arr3 = np.array([1, 1, 0, 1], int)
>>> arr1[arr2, arr3]
array([ 13.,  2.,  1., 13.])
```



배열 쿼리

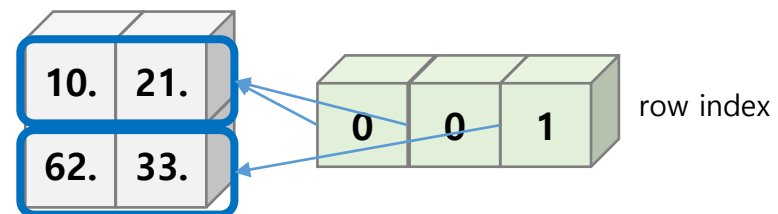
take 함수로 쿼리

```
>>> arr1 = np.array([7, 6, 6, 9], float)
>>> arr2 = np.array([1, 0, 1, 3, 3, 1], int)
>>> arr1.take(arr2)
array([ 6., 7., 6., 9., 9., 6.])
```



차원에 따라 부분집합 선택

```
>>> arr1 = np.array([[10, 21], [62, 33]], float)
>>> arr2 = np.array([0, 0, 1], int)
>>> arr1.take(arr2, axis=0)
array([[ 10.,  21.],
       [ 10.,  21.],
       [ 62.,  33.]])
>>> arr1.take(arr2, axis=1)
array([[ 10.,  10.,  21.],
       [ 62.,  62.,  33.]])
```



arr1.take(arr2, axis=0)

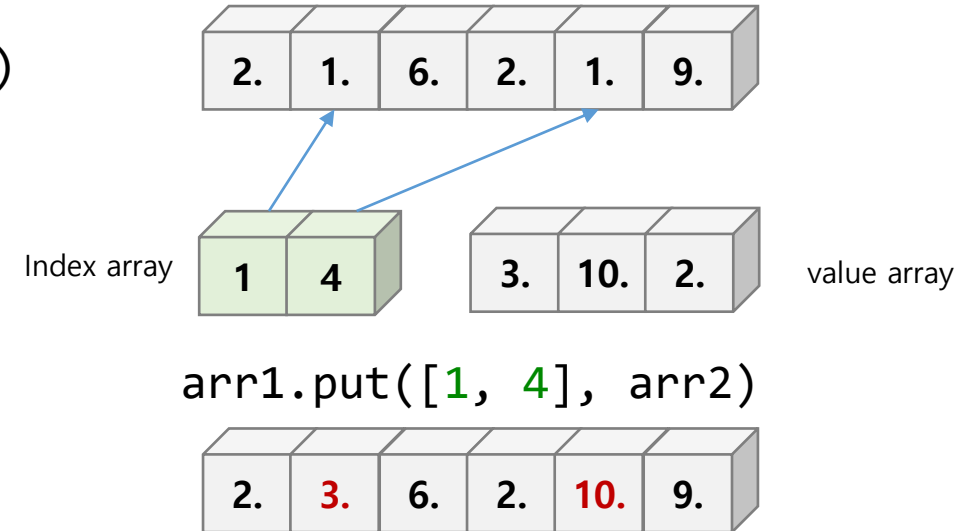


배열 수정 쿼리

인덱스 배열로 쿼리해서 수정

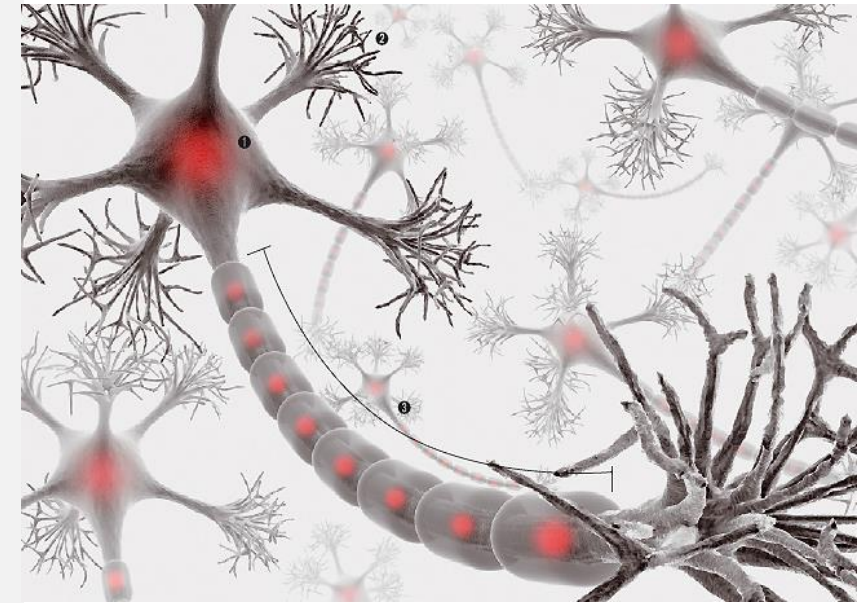
```
>>> arr1 = np.array([2, 1, 6, 2, 1, 9], float)
>>> arr2 = np.array([3, 10, 2], float)
>>> arr1.put([1, 4], arr2)
>>> arr1
array([ 2.,  3.,  6.,  2., 10.,  9.])
```

put 함수는 인덱스와 배열의 값을 입력
받아 해당 인덱스 위치에 값을 넣어줌



4 선형대수 연산

1. 행렬곱
2. 벡터의 외적, 내적, 벡터곱
3. 행렬식, 역행렬
4. 고윳값, 고유벡터
5. 통계 함수



행렬곱

```
>>> X = np.arange(15).reshape((3, 5))
>>> X
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> X.T
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
>>> np.dot(X.T, X)
array([[ 2.584 ,  1.8753,  0.8888],
       [ 1.8753,  6.6636,  0.3884],
       [ 0.8888,  0.3884,  3.9781]])
```

$$\begin{array}{c} \vec{a}_1 \rightarrow \\ \vec{a}_2 \rightarrow \end{array} \begin{array}{c} \vec{b}_1 \quad \vec{b}_2 \\ \downarrow \quad \downarrow \end{array} \begin{array}{c} A \\ B \end{array} \cdot \begin{array}{c} \begin{bmatrix} 3 & 3 \\ 5 & 2 \end{bmatrix} \\ C \end{array} = \begin{array}{c} \begin{bmatrix} \vec{a}_1 \cdot \vec{b}_1 & \vec{a}_1 \cdot \vec{b}_2 \\ \vec{a}_2 \cdot \vec{b}_1 & \vec{a}_2 \cdot \vec{b}_2 \end{bmatrix} \end{array}$$

벡터의 외적, 내적, 벡터곱

```
>>> arr1 = np.array([12, 43, 10], float)
>>> arr2 = np.array([21, 42, 14], float)
>>> np.outer(arr1, arr2)
array([[ 252.,  504.,  168.],
       [ 903., 1806.,  602.],
       [ 210.,  420.,  140.]])
>>> np.inner(arr1, arr2)
2198.0
>>> np.cross(arr1, arr2)
array([ 182.,  42., -399.])
```

참고 벡터의 외적, 내적, 벡터곱

Inner Product (내적)

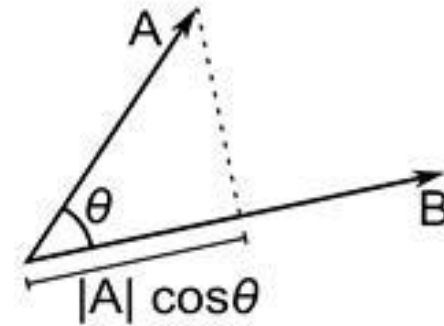
$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z = |\vec{a}| |\vec{b}| \cos(\theta)$$

Outer Product (외적)

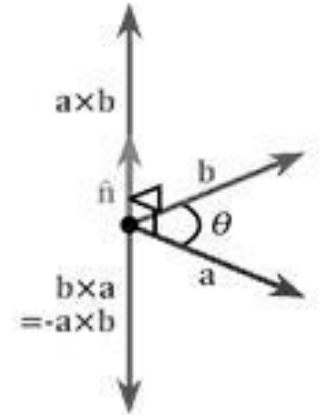
$$\mathbf{u} \otimes \mathbf{v} = \mathbf{u} \mathbf{v}^T = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} = \begin{bmatrix} u_1 v_1 & u_1 v_2 & u_1 v_3 \\ u_2 v_1 & u_2 v_2 & u_2 v_3 \\ u_3 v_1 & u_3 v_2 & u_3 v_3 \\ u_4 v_1 & u_4 v_2 & u_4 v_3 \end{bmatrix}.$$

Cross Product (벡터곱)

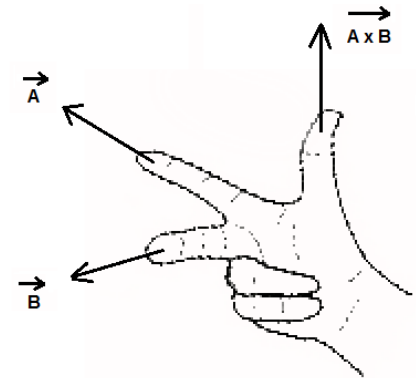
$$\vec{A} \times \vec{B} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = \begin{vmatrix} a_2 & a_3 \\ b_2 & b_3 \end{vmatrix} \mathbf{i} - \begin{vmatrix} a_1 & a_3 \\ b_1 & b_3 \end{vmatrix} \mathbf{j} + \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix} \mathbf{k}$$



a. Dot Product



b. Cross Product



행렬식, 역행렬

행렬식

```
>>> matrix = np.array([[74, 22, 10], [92, 31, 17], [21, 22, 12]], float)
>>> matrix
array([[ 74.,  22.,  10.],
       [ 92.,  31.,  17.],
       [ 21.,  22.,  12.]])
>>> np.linalg.det(matrix)
-2852.0000000000032
```

역행렬

```
>>> inv_matrix = np.linalg.inv(matrix)
>>> inv_matrix
array([[ 0.00070126,  0.01542777, -0.02244039],
       [ 0.26192146, -0.23772791,  0.11851332],
       [-0.48141655,  0.4088359 , -0.09467041]])
```

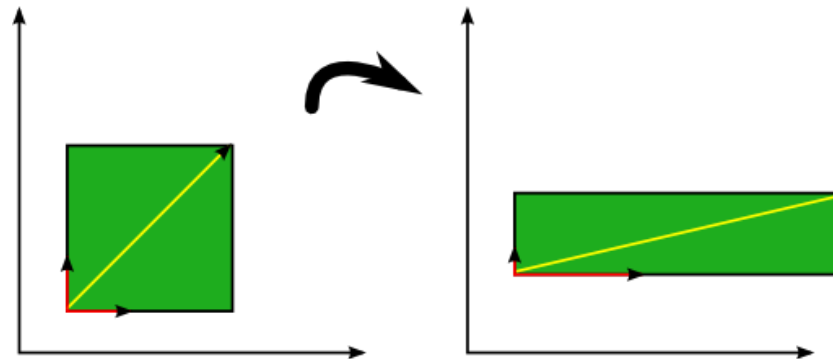
고윳값, 고유벡터

```
>>> vals, vecs = np.linalg.eig(matrix)
>>> vals
array([ 107.99587441, 11.33411853, -2.32999294])
>>> vecs
array([[ -0.57891525, -0.21517959,  0.06319955],
       [ -0.75804695,  0.17632618, -0.58635713],
       [ -0.30036971,  0.96052424,  0.80758352]])
```

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

Eigenvector of Matrix \mathbf{A}

Eigenvalue of Matrix \mathbf{A}



통계 함수

```
>>> arr = np.random.rand(8, 4)
>>> arr.mean()
0.45808075801881332
>>> np.mean(arr)
0.45808075801881332
>>> arr.sum()
14.658584256602026
```

메소드	설명
mean	평균 계산. 배열이 비어있을 경우 평균은 디폴트 NaN(Not a Number)으로 설정됨
std, var	std는 배열의 표준 편차 계산, var는 배열의 분산 계산 자유도(degree of freedom)를 옵션으로 줄 수 있음(디폴트는 배열의 길이)
min, max	최솟값과 최댓값 계산
argmin, argmax	최솟값과 최댓값을 갖는 요소의 인덱스 반환

Thank you!

