

Data Science

- Lecture 4-

Prof. Woongsup Lee

Gyeongsang National University

넘파이

01 넘파이란?

02 넘파이 배열 객체 다루기

03 넘파이 배열 연산

04 비교 연산과 데이터 추출

학습목표

- 넘파이의 개념과 특징에 대해 알아보고, 모듈을 설치한다.
- 넘파이를 이용하여 넘파이 배열 객체를 다루는 다양한 방법에 대해 실습한다.
- 넘파이를 이용하여 넘파일 배열 연산을 수행한다.
- 넘파이를 이용하여 비교 연산과 데이터 추출 방법을 이해한다.

01 넘파이란?

1. 넘파이의 개념

- 파이썬의 고성능 과학 계산용 라이브러리
 - 벡터나 행렬 같은 선형대수의 표현법을 코드로 처리
 - 사실상의 표준 라이브러리
 - 다차원 리스트나 크기가 큰 데이터 처리에 유리

$$\begin{array}{ll} 2x_1 + 2x_2 + x_3 = 9 & \left[\begin{array}{cccc} 2 & 2 & 1 & 9 \end{array} \right] \\ 2x_1 - x_2 + 2x_3 = 6 & \left[\begin{array}{cccc} 2 & -1 & 2 & 6 \end{array} \right] \\ x_1 - x_2 + 2x_3 = 5 & \left[\begin{array}{cccc} 1 & -1 & 2 & 5 \end{array} \right] \end{array}$$

그림 3-1 선형대수의 예시

01 넘파이란?

2. 넘파이의 특징

- 속도가 빠르고 메모리 사용이 효율적
 - 데이터를 메모리에 할당하는 방식이 기존과 다름
- 반복문을 사용하지 않음
 - 연산할 때 병렬로 처리
 - 함수를 한 번에 많은 요소에 적용
- 다양한 선형대수 관련 함수 제공
- C, C++, 포트란 등 다른 언어와 통합 사용 가능

01 넘파이란?

3. numpy 모듈 설치

① conda 가상환경에서 numpy 모듈 설치

```
(base) C:\...> conda activate ml  
(ml) C:\...> conda install numpy
```

② 주피터 노트북 생성

[TIP] conda 가상환경에서 ‘jupyter notebook’을 입력하여 실행하면 된다.

③ numpy 모듈 호출

- import numpy as np

02 넘파이 배열 객체 다루기

1. 넘파이 배열과 텐서

- 넘파이 배열(ndarray) : 넘파이에서 텐서 데이터를 다루는 객체
- 텐서(tensor) : 선형대수의 데이터 배열
 - 랭크(rank)에 따라 이름이 다름

표 3-1 랭크(Rank)별 구분들

랭크(Rank)	이름	예
0	스칼라(scalar)	7
1	벡터(vector)	[10, 10]
2	행렬(matrix)	[[10, 10], [15, 15]]
3	3차원 텐서(3-order tensor)	[[[1, 5, 9], [2, 6, 10]], [[3, 7, 11], [4, 8, 12]]]
n	n차원 텐서(n-order tensor)	

02 넘파이 배열 객체 다루기

2. 배열의 메모리 구조

- 배열 생성
 - np.array 함수 사용하여 배열 생성

```
import numpy as np  
test_array = np.array([1, 4, 5, 8], float)
```

- 매개변수 1: 배열 정보
- 매개변수 2: 넘파이 배열로 표현하려는 데이터 타입

02 넘파이 배열 객체 다루기

- 파이썬 리스트와 넘파이 배열의 차이점

- 텐서 구조에 따라 배열 생성
 - 배열의 모든 구성 요소에 값이 존재해야 함

```
import numpy as np
test_list = [[1, 4, 5, 8], [1, 4, 5]]
np.array(test_list, float) # ValueError
```

- 동적 타이핑을 지원하지 않음
 - 하나의 데이터 타입만 사용
 - 데이터를 메모리에 연속적으로 나열
 - 각 값 메모리 크기가 동일
 - 검색이나 연산 속도가 리스트에 비해 훨씬 빠름

02 넘파이 배열 객체 다루기

In [1]:	import numpy as np test_array = np.array([1, 4, 5, 8], float) test_array
Out [1]:	array([1., 4., 5., 8.])
In [2]:	type(test_array[3])
Out [2]:	numpy.float64

02 넘파이 배열 객체 다루기

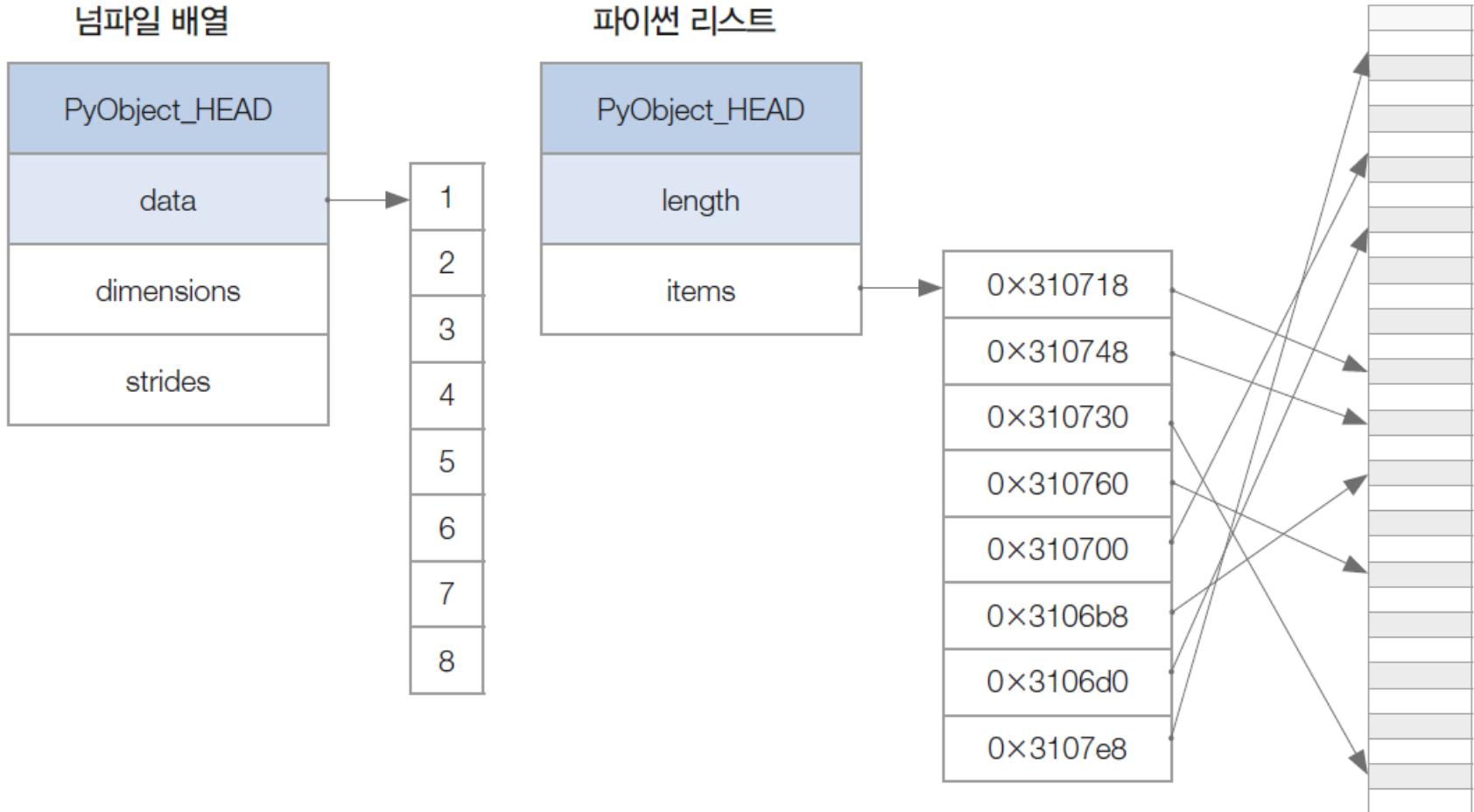


그림 3-2 넘파이 배열과 리스트의 차이

© Pythonic Perambulations

02 넘파이 배열 객체 다루기

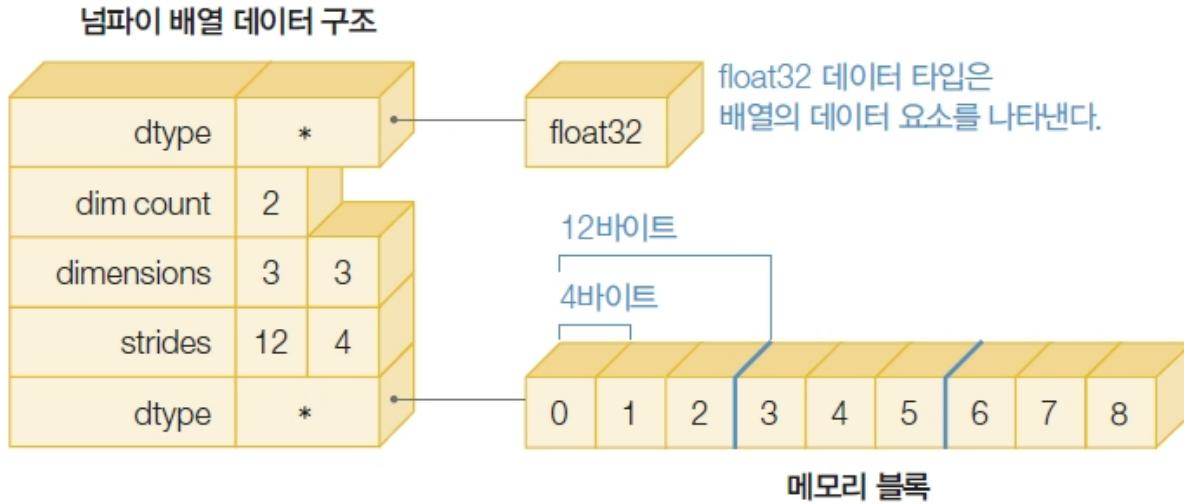


그림 3-3 넘파이 배열의 메모리 블록 배열

- float32 데이터 타입인 넘파이 배열의 각 값을 저장하는 메모리 블록은 4바이트씩 차지

02 넘파이 배열 객체 다루기

3. 배열의 생성

In [3]:	test_array = np.array([1, 4, 5, "8"], float) print(test_array)
Out [3]:	[1. 4. 5. 8.]

- 배열을 실수형으로 선언
- 배열을 출력해보면 값이 모두 실수형

In [4]:	print(type(test_array[3])) # 실수형(floating Type)으로 자동 형 변환 실시
Out [4]:	class 'numpy.float64'>

- 개별 값 데이터 타입도 <class 'numpy.float64'>로 출력
- float64는 64비트(bit), 즉 8바이트의 실수형 데이터

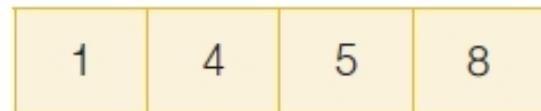
02 넘파이 배열 객체 다루기

In [5]:	print(test_array.dtype) # 배열 전체의 데이터 타입 반환
Out [5]:	float64
In [6]:	print(test_array.shape) # 배열의 구조(shape)를 반환함
Out [6]:	(4,)

- 데이터 특징을 출력하는 요소(property)는 `dtype`과 `shape`
- `dtype`은 넘파이 배열의 데이터 타입을 반환
- `shape`은 넘파이 배열에서 객체(object)의 차원(dimension)에 대한 구성 정보를 반환

02 넘파이 배열 객체 다루기

3.1 배열의 구조(shape)



(4,)

넘파이 배열(ndarray)의 구성

넘파이 배열의 구조(shape)
(타입은 튜플)

그림 3-4 랭크가 1일 때 배열의 구조

02 넘파이 배열 객체 다루기

3.1 배열의 구조(shape)

```
In [7]: matrix = [[1,2,5,8], [1,2,5,8], [1,2,5,8]]  
np.array(matrix, int).shape
```

```
Out [7]: (3, 4)
```

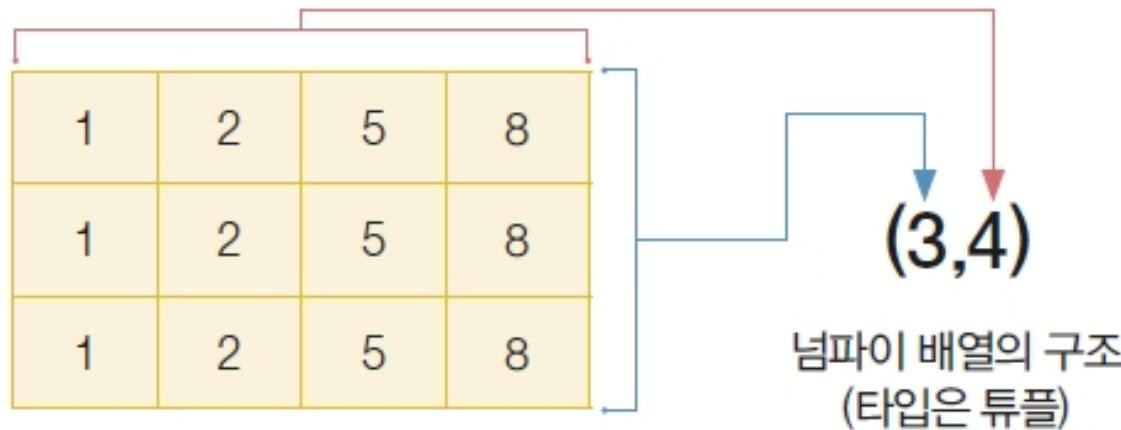


그림 3-5 랭크가 2일 때 배열의 구조

02 넘파이 배열 객체 다루기

3.1 배열의 구조(shape)

```
In [8]: tensor_rank3 = [  
    [[1, 2, 5, 8], [1, 2, 5, 8], [1, 2, 5, 8]],  
    [[1, 2, 5, 8], [1, 2, 5, 8], [1, 2, 5, 8]],  
    [[1, 2, 5, 8], [1, 2, 5, 8], [1, 2, 5, 8]],  
    [[1, 2, 5, 8], [1, 2, 5, 8], [1, 2, 5, 8]]  
]  
np.array(tensor_rank3, int).shape
```

Out [8]: (4, 3, 4)

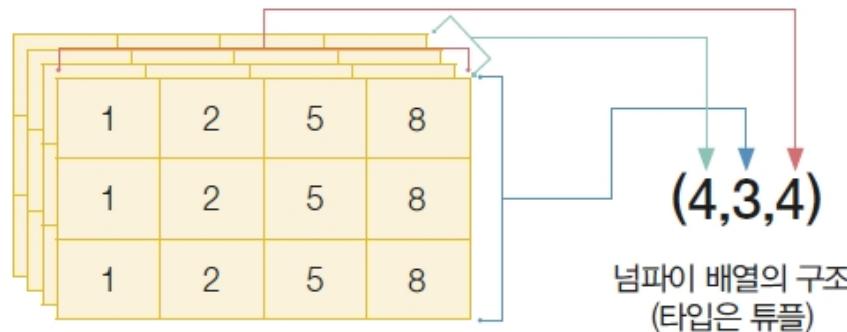


그림 3-6 랭크가 3일 때, 3차원 배열의 구조

02 넘파이 배열 객체 다루기

3.1 배열의 구조(shape)

In [9]:	tensor_rank3 = [[[1, 2, 5, 8], [1, 2, 5, 8], [1, 2, 5, 8]], [[1, 2, 5, 8], [1, 2, 5, 8], [1, 2, 5, 8]], [[1, 2, 5, 8], [1, 2, 5, 8], [1, 2, 5, 8]]] np.array(tensor_rank3, int).ndim
Out [9]:	3
In [10]:	np.array(tensor_rank3, int).size
Out [10]:	48

02 넘파이 배열 객체 다루기

3.2 dtype

- 매개변수 `dtype`으로 넘파이 배열의 데이터 타입 지정
 - 변수가 사용하는 메모리 크기가 정해짐

In [11]:	<code>np.array([[1, 2, 3.5], [4, 5, 6.5]], dtype=int)</code>
Out [11]:	<code>array([[1, 2, 3], [4, 5, 6]])</code>

- `dtype`을 실수형인 `float`으로 지정한다면 모든 데이터가 실수형으로 저장되는 것을 확인 가능

In [12]:	<code>np.array([[1, 2, 3.5], [4, 5, 6.5]], dtype=float)</code>
Out [12]:	<code>array([[1. , 2. , 3.5], [4. , 5. , 6.5]])</code>

02 넘파이 배열 객체 다루기

3.2 dtype

In [13]:	import sys np.array([[1, 2, 3.5], [4, 5, 6.5]], dtype=np.float64).itemsize
Out [13]:	8
In [14]:	np.array([[1, 2, 3.5], [4, 5, 6.5]], dtype=np.float32).itemsize
Out [14]:	4

- itemsize 요소(property)로 넘파이 배열에서 각 요소가 차지하는 바이트(byte) 확인
- np.float64로 dtype을 선언하면 64비트, 즉 8바이트 차지
- np.float32로 dtype을 선언하면 32비트, 즉 4바이트 차지

02 넘파이 배열 객체 다루기

4. 배열의 구조 다루기

- reshape 함수로 배열의 구조를 변경하고 랭크를 조절

In [15]:	x = np.array([[1, 2, 5, 8], [1, 2, 5, 8]]) x.shape
Out [15]:	(2,4)
In [16]:	x.reshape(-1,)
Out [16]:	array([1, 2, 5, 8, 1, 2, 5, 8])



그림 3-7 reshape 함수로 변경된 배열의 구조

02 넘파이 배열 객체 다루기

- 반드시 전체 요소의 개수는 통일

In [17]:	x = np.array(range(8)) x
Out [17]:	array([0, 1, 2, 3, 4, 5, 6, 7])
In [18]:	x.reshape(2,2)
Out [18]:	----- -- ValueError Traceback (most recent call last) <ipython-input-2-31f07738c8d3> in <module> ----> 1 x.reshape(2,2) ValueError: cannot reshape array of size 8 into shape (2,2)

02 넘파이 배열 객체 다루기

- 1을 사용하면 나머지 차원의 크기를 지정했을 때 전체 요소의 개수를 고려하여 마지막 차원이 자동으로 지정됨

In [19]:	x = np.array(range(8)).reshape(4,2) x
Out [19]:	array([[0, 1], [2, 3], [4, 5], [6, 7]])
In [20]:	x.reshape(2,-1)
Out [20]:	array([[[0, 1, 2, 3], [4, 5, 6, 7]]])

In [21]:	x.reshape(2,2,-1)
Out [21]:	array([[[0, 1], [2, 3]], [[4, 5], [6, 7]]])

02 넘파이 배열 객체 다루기

- `flatten` 함수는 데이터 그대로 1차원으로 변경
 - 데이터의 개수는 그대로 존재
 - 배열의 구조만 변한다

In [22]:	x = np.array(range(8)).reshape(2,2,2) x	<p>The diagram illustrates the flattening process. On the left, a 3D array of shape (2,2,2) is shown as a cube divided into 8 smaller cubes. The elements are labeled: top-front-left (0), top-front-right (1), top-back-left (4), top-back-right (5), bottom-front-left (2), bottom-front-right (3), bottom-back-left (6), and bottom-back-right (7). An arrow points from this 3D structure down to a 1D horizontal row on the right, which contains all 8 elements in sequence: 0, 1, 2, 3, 4, 5, 6, 7.</p> <p>(2,2,2)</p> <p>↓</p> <p>(8,)</p>
Out [22]:	array([[[0, 1], [2, 3]], [[4, 5], [6, 7]]])	
In [23]:	x.flatten()	
Out [23]:	array([0, 1, 2, 3, 4, 5, 6, 7])	

그림 3-8 `flatten` 함수로 변경된 배열의 구조

02 넘파이 배열 객체 다루기

5. 인덱싱과 슬라이싱

5.1 인덱싱

- 인덱싱(indexing) : 리스트에 있는 값의 상대적인 주소(offset)로 값에 접근
- 넘파이 배열의 인덱스 표현에는 ‘,’을 지원
 - ‘[행][열]’ 또는 ‘[행,열]’ 형태
- 3차원 텐서 이상은 shape에서 출력되는 랭크 순서대로 인덱싱에 접근

02 넘파이 배열 객체 다루기

In [24]:	x = np.array([[1, 2, 3], [4, 5, 6]], int) x
Out [24]:	array([[1, 2, 3], [4, 5, 6]])
In [25]:	x[0][0]
Out [25]:	1
In [26]:	x[0,2]
Out [26]:	3
In [27]:	x[0, 1] = 100 x
Out [27]:	array([[1, 100, 3], [4, 5, 6]])

02 넘파이 배열 객체 다루기

5.2 슬라이싱

- 슬라이싱(slicing) : 인덱스를 사용하여 리스트 일부를 잘라내어 반환
- 넘파이 배열은 행과 열을 나눠 슬라이싱할 수 있음

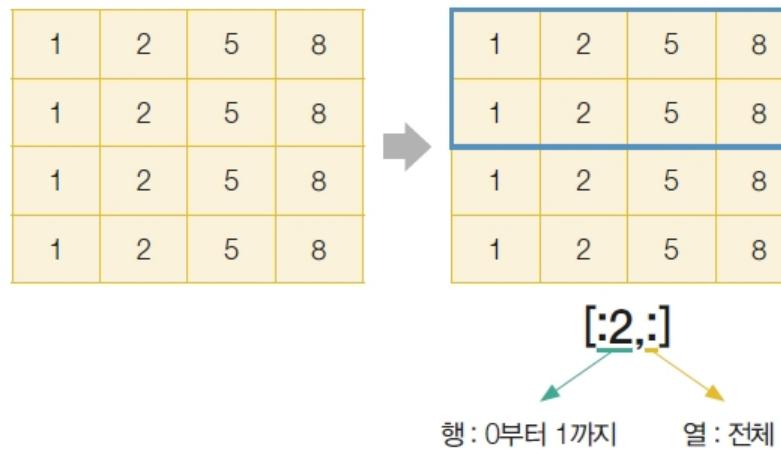


그림 3-9 슬라이싱

02 넘파이 배열 객체 다루기

In [28]:	x = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]], int) x[:,2:] # 전체 행의 2열 이상
Out [28]:	array([[3, 4, 5], [8, 9, 10]])

- x는 2행 5열인 행렬
- x[:,2:]는 행 부분은 행 전체, 열 부분은 인덱스가 2 이후의 값

In [29]:	x[1,1:3] # 1행의 1열 ~ 2열
Out [29]:	array([7, 8])

- x[1,1:3]은 행 부분은 첫 번째 행만을 의미
- 열 부분 1:3은 열이 1부터 2까지의 값을 추출

In [30]:	x[1:3] # 1행 ~ 2행의 전체
Out [30]:	array([[6, 7, 8, 9, 10]])

- 행렬 전체의 행의 개수가 2이기 때문에 이를 넘어가는 인덱스는 무시

02 넘파이 배열 객체 다루기

- 증가값(step) : 리스트에서 데이터의 요소를 호출할 때 데이터를 건너뛰면서 반환
 - '[시작 인덱스:마지막 인덱스:증가값]' 형태
 - 각 랭크에 있는 요소별로 모두 적용할 수 있음

In [31]:	x = np.array(range(15), int).reshape(3, -1) x
Out [31]:	array([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12, 13, 14]])

02 넘파이 배열 객체 다루기

In [32]:	x[:,::2]
Out [32]:	array([[0, 2, 4], [5, 7, 9], [10, 12, 14]])
In [33]:	x[::-2,::3]
Out [33]:	array([[0, 3], [10, 13]])

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

x[:,::2]

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

x[::-2,::3]

그림 3-10 증가값 활용시 행렬의 구조

02 넘파이 배열 객체 다루기

6. 배열 생성 함수

6.1 arange

- range 함수와 같이 차례대로 값을 생성
- '(시작 인덱스, 마지막 인덱스, 증가값)'으로 구성
- range 함수와 달리 증가값에 실수형이 입력되어도 값을 생성할 수 있음
- 소수점 값을 주기적으로 생성할 때 유용

02 넘파이 배열 객체 다루기

In [34]:	np.arange(10)
Out [34]:	array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In [35]:	np.arange(-5, 5)
Out [35]:	array([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4])
In [36]:	np.arange(0, 5, 0.5)
Out [36]:	array([0., 0.5, 1., 1.5, 2., 2.5, 3., 3.5, 4., 4.5])

- 증가값에 실수형이 입력되어도 값을 생성
- 소수점 값을 주기적으로 생성할 때 유용

02 넘파이 배열 객체 다루기

6.2 ones, zeros, empty

- **ones** 함수 : 1로만 구성된 넘파이 배열을 생성
 - 사전에 shape 값을 넣어서 원하는 크기의 넘파이 배열 생성
- **zeros** 함수 : 0으로만 구성된 넘파이 배열을 생성
- **empty** 함수 : 활용 가능한 메모리 공간 확보하여 반환
 - ones와 zeros는 먼저 shape의 크기만큼 메모리를 할당하고 그곳에 값을 채움
 - 해당 메모리 공간에 값이 남았을 경우 그 값을 함께 반환
 - empty는 메모리 초기화 않아 생성될 때마다 다른 값 반환
- 생성 시점에서 **dtype**을 지정해주면 해당 데이터 타입으로 배열 생성

02 넘파이 배열 객체 다루기

In [37]:	np.ones(shape=(5,2), dtype=np.int8)
Out [37]:	array([[1, 1], [1, 1], [1, 1], [1, 1], [1, 1]], dtype=int8)
In [38]:	np.zeros(shape=(2,2), dtype=np.float32)
Out [38]:	array([[0., 0.], [0., 0.]], dtype=float32)
In [39]:	np.empty(shape=(2,4), dtype=np.float32)
Out [39]:	array([[0.000e+00, 1.401e-45, 0.000e+00, 5.689e-43], [1.530e-42, 0.000e+00, 1.076e-42, 0.000e+00]], dtype=float32)

02 넘파이 배열 객체 다루기

6.3 ones_like, zeros_like, empty_like

- `ones_like` 함수 : 기존 넘파이 배열과 같은 크기로 만들어 내용을 1로 채움
- `zeros_like` 함수 : 기존 넘파이 배열과 같은 크기로 만들어 내용을 0으로 채움
- `empty_like` 함수 : 기존 넘파이 배열과 같은 크기로 만들어 빈 상태로 만듦

02 넘파이 배열 객체 다루기

6.3 ones_like, zeros_like, empty_like

In [40]:	x = np.arange(12).reshape(3,4) x
Out [40]:	array([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]])
In [41]:	np.ones_like(x)
Out [41]:	array([[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]])
In [42]:	np.zeros_like(x)
Out [42]:	array([[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]])

02 넘파이 배열 객체 다루기

6.4 identity, eye, diag

- **identity** 함수 : 단위행렬(i행렬)을 생성
 - 매개변수 n으로 $n \times n$ 단위행렬을 생성

In [43]:	<code>np.identity(n=3, dtype=int)</code>
Out [43]:	<code>array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])</code>
In [44]:	<code>np.identity(n=4, dtype=int)</code>
Out [44]:	<code>array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]])</code>

02 넘파이 배열 객체 다루기

6.4 identity, eye, diag

- `eye` 함수 : 시작점과 행렬 크기를 지정, 단위행렬 생성
 - N은 행의 개수, M은 열의 개수를 지정
 - k는 열의 값을 기준으로 시작 인덱스

In [45]:	<code>np.eye(N=3, M=5)</code>
Out [45]:	<code>array([[1., 0., 0., 0., 0.], [0., 1., 0., 0., 0.], [0., 0., 1., 0., 0.]])</code>
In [46]:	<code>np.eye(N=3, M=5, k=2)</code>
Out [46]:	<code>array([[0., 0., 1., 0., 0.], [0., 0., 0., 1., 0.], [0., 0., 0., 0., 1.]])</code>

02 넘파이 배열 객체 다루기

6.4 identity, eye, diag

- `diag` 함수 : 행렬의 대각성분 값을 추출

In [47]:	<code>matrix = np.arange(9).reshape(3,3)</code> <code>matrix</code>
Out [47]:	<code>array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])</code>
In [48]:	<code>np.diag(matrix)</code>
Out [48]:	<code>array([0, 4, 8])</code>
In [49]:	<code>np.diag(matrix, k=1)</code>
Out [49]:	<code>array([1, 5])</code>

0	1	2
3	4	5
6	7	8

(a) `np.diag(matrix)`

0	1	2
3	4	5
6	7	8

(b) `np.diag(matrix, k=1)`

그림 3-11 `diag` 함수 사용

02 넘파이 배열 객체 다루기

7. 통계 분석 함수

- uniform 함수 : 균등분포 함수
 - ‘np.random.uniform(시작값, 끝값, 데이터개수)’

In [50]:	np.random.uniform(0, 5, 10)
Out [50]:	array([3.87101195, 0.12263269, 0.80780157, 0.65361498, 0.55792293, 3.64577442, 0.93322468, 3.1913397, 1.82159678, 3.64401469])

02 넘파이 배열 객체 다루기

7. 통계 분석 함수

- normal 함수 : 정규분포 함수
 - ‘np.random.normal(평균값, 분산, 데이터개수)’

In [51]:	np.random.normal(0, 2, 10)
Out [51]:	array([4.92446265, -2.4753182 , -2.12734589, -2.75839296, -0.22365806, -0.93325909, 1.81593553, 1.74506567, 2.20788194, 1.42156357])

03 넘파이 배열 연산

1. 연산 함수

- 연산 함수(operation function) : 배열 내부 연산을 지원하는 함수
- 축(axis) : 배열의 랭크가 증가할 때마다 새로운 축이 추가되어 차원 증가
- sum 함수 : 각 요소의 합을 반환

In [1]:	import numpy as np test_array = np.arange(1, 11) test_array.sum()
Out [1]:	55

03 넘파이 배열 연산

- sum 함수를 랭크가 2 이상인 배열에 적용할 때 축으로 연산의 방향을 설정

In [2]:	test_array = np.arange(1,13).reshape(3,4) test_array	
Out [2]:	array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])	
In [3]:	test_array.sum(axis=0)	
Out [3]:	array([15, 18, 21, 24])	
In [4]:	test_array.sum(axis=1)	axis=0
Out [4]:	array([10, 26, 42])	axis=1

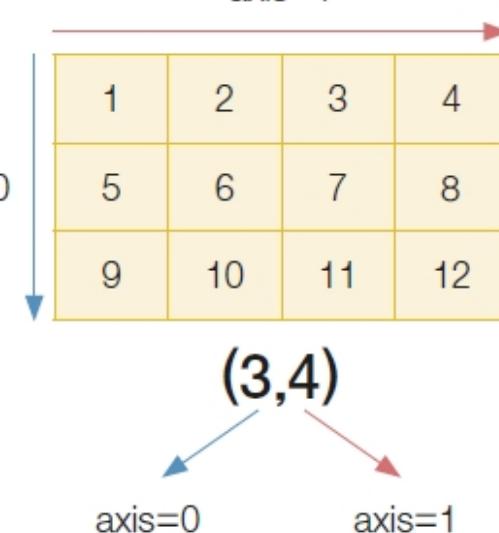

 $(3,4)$

그림 3-12 축에 따른 연산

03 넘파이 배열 연산

In [5]:	test_array = np.arange(1, 13).reshape(3, 4) third_order_tensor = np.array([test_array,test_array, test_array]) third_order_tensor
Out [5]:	array([[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]], [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]], [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]])

03 넘파이 배열 연산

In [6]:	third_order_tensor.sum(axis=0)
Out [6]:	array([[3, 6, 9, 12], [15, 18, 21, 24], [27, 30, 33, 36]])
In [7]:	third_order_tensor.sum(axis=1)
Out [7]:	array([[15, 18, 21, 24], [15, 18, 21, 24], [15, 18, 21, 24]])
In [8]:	third_order_tensor.sum(axis=2)
Out [8]:	array([[10, 26, 42], [10, 26, 42], [10, 26, 42]])

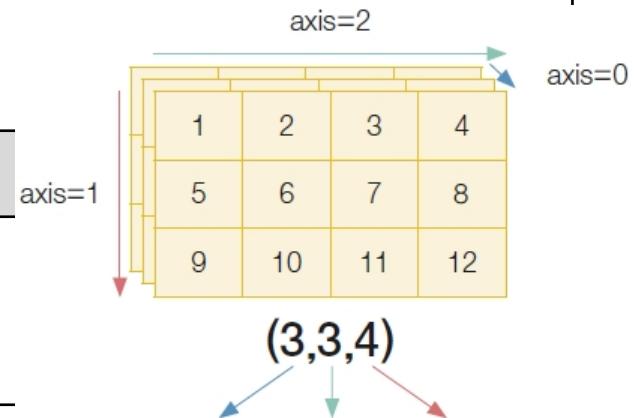


그림 3-13 3차원 텐서에서의 연산 =2

03 넘파이 배열 연산

In [9]:	test_array = np.arange(1, 13).reshape(3, 4) test_array
Out [9]:	array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
In [10]:	test_array.mean(axis=1) # axis=1 축을 기준으로 평균 연산
Out [10]:	array([2.5, 6.5, 10.5])
In [11]:	test_array.std() # 전체 값에 대한 표준편차 연산
Out [11]:	3.452052529534663

03 넘파이 배열 연산

In [12]:	test_array.std(axis=0) # axis=0 축을 기준으로 표준편차 연산
Out [12]:	array([3.26598632, 3.26598632, 3.26598632, 3.26598632])
In [13]:	np.sqrt(test_array) # 각 요소에 제곱근 연산 수행
Out [13]:	array([[1., 1.41421356, 1.73205081, 2.], [2.23606798, 2.44948974, 2.64575131, 2.82842712], [3., 3.16227766, 3.31662479, 3.46410162]])

03 넘파이 배열 연산

2. 연결 함수

- 연결 함수(concatenation functions) : 두 객체 간의 결합을 지원하는 함수
- vstack 함수 : 배열을 수직으로 붙여 하나의 행렬을 생성
- hstack 함수 : 배열을 수평으로 붙여 하나의 행렬을 생성

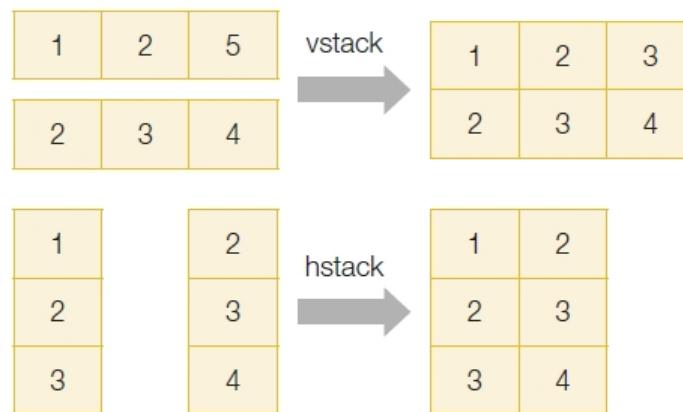


그림 3-14 연결을 통한 행렬의 결합

03 넘파이 배열 연산

- 넘파이는 열 벡터를 표현할 수 없어 2차원 행렬 형태로 표현

In [14]:	v1 = np.array([1, 2, 3]) v2 = np.array([4, 5, 6]) np.vstack((v1, v2))
Out [14]:	array([[1, 2, 3], [4, 5, 6]])
In [15]:	np.hstack((v1,v2))
Out [15]:	array([1, 2, 3, 4, 5, 6])

- 벡터 형태 그대로 hstack을 붙일 경우 그대로 벡터 형태의 배열 생성

03 넘파이 배열 연산

In [16]:	v1 = v1.reshape(-1, 1) v2 = v2.reshape(-1, 1) v1
Out [16]:	array([[1], [2], [3]])
In [17]:	V2
Out [17]:	array([[4], [5], [6]])
In [18]:	np.hstack((v1,v2))
Out [18]:	array([[1, 4], [2, 5], [3, 6]])

- 2차원 행렬 형태로 표현한 열 벡터를 hstack으로 연결

03 넘파이 배열 연산

- concatenate 함수 : 축을 고려하여 두 개의 배열을 결합
 - 스택(stack) 계열의 함수와 달리 생성될 배열과 소스가 되는 배열의 차원이 같아야 함
 - 두 벡터를 결합하고 싶다면, 해당 벡터를 일단 2차원 배열 꼴로 변환 후 행렬로 나타내야 함

In [19]:	v1 = np.array([[1, 2, 3]]) v2 = np.array([[4, 5, 6]]) np.concatenate((v1,v2), axis=0)
Out [19]:	array([[1, 2, 3], [4, 5, 6]])

- v1과 v2 모두 사실상 행렬이지만 벡터의 형태
- 매개변수 axis=0로 행을 기준으로 연결

03 넘파이 배열 연산

In [20]:	v1 = np.array([1, 2, 3, 4]).reshape(2,2) v2 = np.array([[5,6]]).T v1
Out [20]:	array([[1, 2], [3, 4]])
In [21]:	v2
Out [21]:	array([[5], [6]])
In [22]:	np.concatenate((v1,v2), axis=1)
Out [22]:	array([[1, 2, 5], [3, 4, 6]])

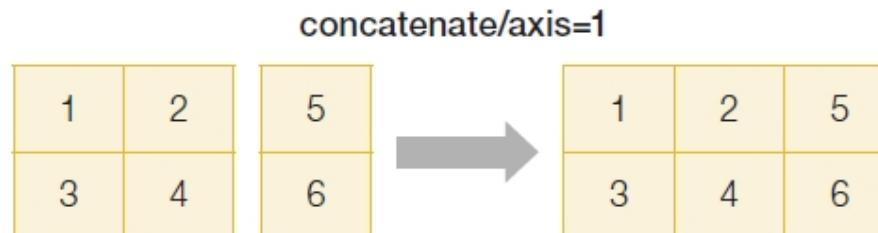


그림 3-15 배열의 연결

03 넘파이 배열 연산

3. 사칙연산 함수

- 넘파이는 파이썬과 동일하게 배열 간 사칙연산 지원
 - 행렬과 행렬, 벡터와 백터 간 사칙연산이 가능
- 같은 배열의 구조일 때 요소별 연산(element-wise operation)
 - 요소별 연산 : 두 배열의 구조가 동일할 경우 같은 인덱스 요소들끼리 연산

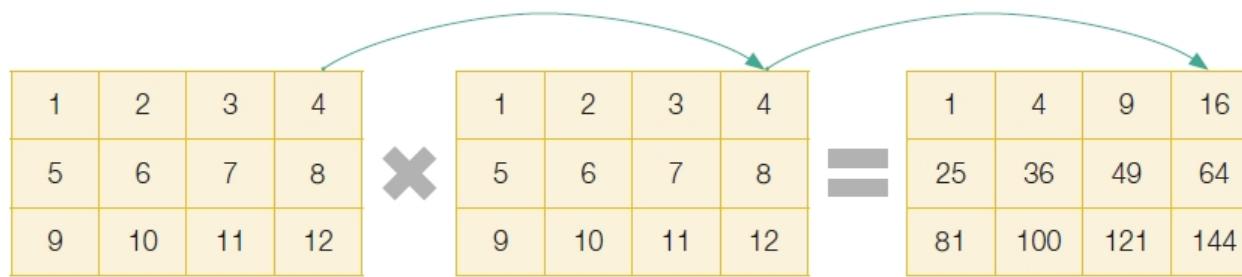


그림 3-16 배열 간 요소별 연산의 예시

03 넘파이 배열 연산

In [23]:	x = np.arange(1, 7).reshape(2,3) x
Out [23]:	array([[1, 2, 3], [4, 5, 6]])
In [24]:	x + x
Out [24]:	array([[2, 4, 6], [8, 10, 12]])
In [25]:	x - x
Out [25]:	array([[0, 0, 0], [0, 0, 0]])
In [26]:	x / x
Out [26]:	array([[1., 1., 1.], [1., 1., 1.]])
In [27]:	x ** x
Out [27]:	array([[1, 4, 27], [256, 3125, 46656]]), dtype=int32)

03 넘파이 배열 연산

- 배열 간의 곱셈에서는 요소별 연산과 벡터의 내적(dot product) 연산 가능
 - 벡터의 내적 : 두 배열 간의 곱셈
 - 두 개의 행렬에서 첫 번째 행렬의 열 크기와 두 번째 행렬의 행 크기가 동일해야 함
 - $m \times n$ 행렬과 $n \times l$ 행렬, 벡터의 내적 연산하면 $m \times l$ 의 행렬 생성

$$(m \times n) \cdot (n \times k) = (m \times k)$$

곱셈 연산이 정의됨

그림 3-17 벡터의 내적(dot product) 연산

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} (1 \cdot 5) + (2 \cdot 7) & (1 \cdot 6) + (2 \cdot 8) \\ (3 \cdot 5) + (4 \cdot 7) & (3 \cdot 6) + (4 \cdot 8) \end{bmatrix}$$
$$= \begin{bmatrix} 5 + 14 & 6 + 16 \\ 15 + 28 & 18 + 32 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

그림 3-18 벡터의 내적(dot product) 연산의 예시

03 넘파이 배열 연산

- dot 함수 : 벡터의 내적 연산

In [28]:	x_1 = np.arange(1, 7).reshape(2,3) x_2 = np.arange(1, 7).reshape(3,2) x_1
Out [28]:	array([[1, 2, 3], [4, 5, 6]])
In [29]:	x_2
Out [29]:	array([[1, 2], [3, 4], [5, 6]])
In [30]:	x_1.dot(x_2)
Out [30]:	array([[22, 28], [49, 64]])

- 2×3 행렬과 3×2 행렬의 연산 결과는 2×2 행렬

03 넘파이 배열 연산

- 브로드캐스팅 연산(broadcasting operations) :
하나의 행렬과 스칼라 값들 간의 연산이나 행렬과 벡터 간의 연산
 - 방송국의 전파가 퍼지듯 뒤에 있는 스칼라 값이 모든 요소에 퍼지듯이 연산

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + 3 = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

그림 3-19 브로드캐스팅 연산

03 넘파이 배열 연산

In [31]:	x = np.arange(1, 10).reshape(3,3) x
Out [31]:	array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
In [32]:	x + 10
Out [32]:	array([[11, 12, 13], [14, 15, 16], [17, 18, 19]])
In [33]:	x - 2
Out [33]:	array([[-1, 0, 1], [2, 3, 4], [5, 6, 7]])

03 넘파이 배열 연산

In [34]:	x // 3
Out [34]:	array([[0, 0, 1], [1, 1, 2], [2, 2, 3]], dtype=int32)
In [35]:	x ** 2
Out [35]:	array([[1, 4, 9], [16, 25, 36], [49, 64, 81]], dtype=int32)

03 넘파이 배열 연산

- 행렬과 스칼라 값 외에 행렬과 벡터, 벡터와 벡터 간에도 연산

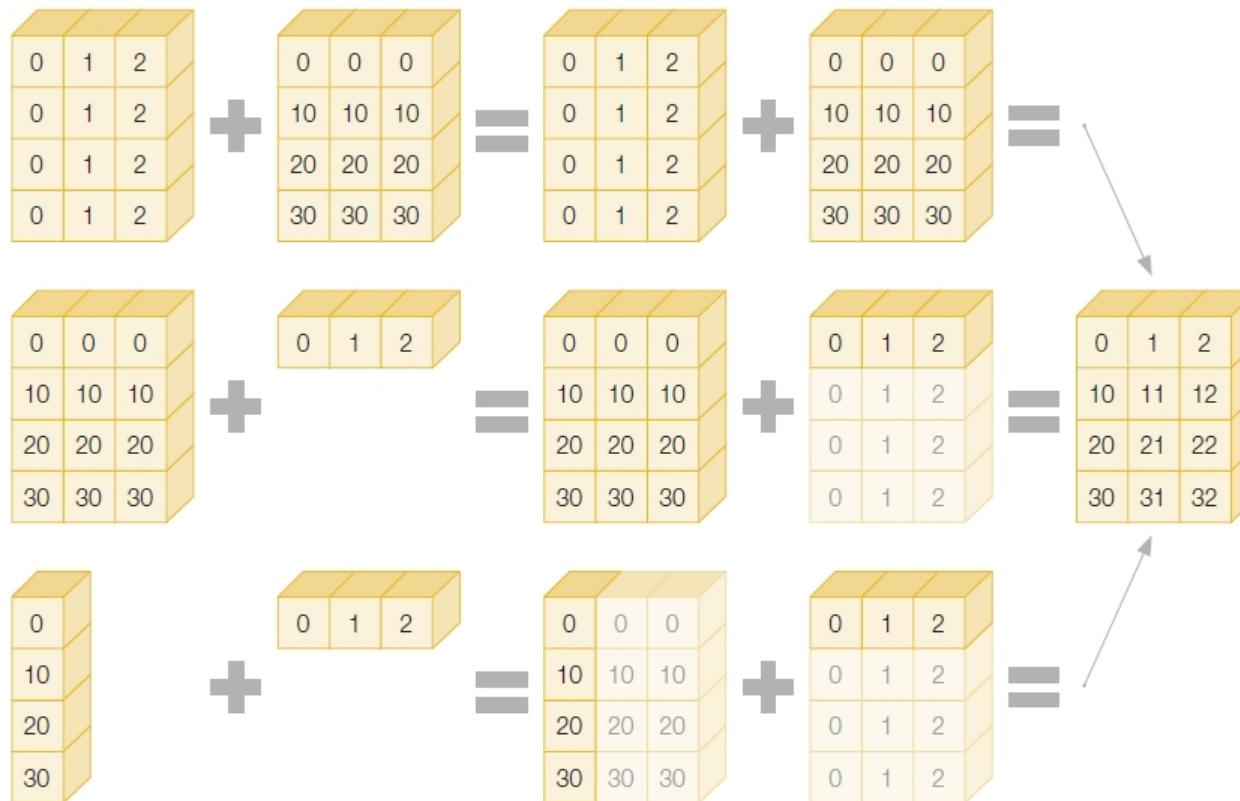


그림 3-20 확장된 브로드캐스팅 연산

03 넘파이 배열 연산

In [36]:	x = np.arange(1, 13).reshape(4,3) x
Out [36]:	array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
In [37]:	v = np.arange(10 , 40 , 10) v
Out [37]:	array([10, 20, 30])
In [38]:	v = np.arange(10 , 40 , 10) v
Out [38]:	array([10, 20, 30])
In [39]:	x + v
Out [39]:	array([[11, 22, 33], [14, 25, 36], [17, 28, 39], [20, 31, 42]])

03 넘파이 배열 연산

The diagram shows the addition of a 4x3 matrix and a 3-vector. The matrix has elements 1 through 12. The vector has elements 10, 20, and 30. The result is a 4x3 matrix where each row is the sum of the corresponding row of the matrix and the vector.

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

$$+ \quad =$$

10	20	30
11	22	33
14	25	36
17	28	39
20	31	42

그림 3-21 행렬과 벡터의 브로드캐스팅 연산

- 뒤에 있는 벡터가 앞에 있는 행렬과 크기를 맞추기 위해 4×3 의 행렬처럼 복제
- 그 다음 요소별 연산처럼 연산

03 넘파이 배열 연산

[하나 더 알기] 넘파이의 성능

- 넘파이의 텐서 연산의 장점 :
C와 유사한 형태로 메모리를 관리하면서 C와 같은 연산 속도로 계산할 수 있다
 - 메모리 구조상 요소들이 붙어있기 때문
 - 파이썬의 가장 큰 특징인 동적 타이핑을 포기했지만,
C로 구현되어 있어 배열 연산에 있어 매우 큰 성능적 우위 확보
 - 대용량 배열 연산에서 넘파이가 사실상 표준으로 사용됨
- 연결 연산처럼 여러 배열을 붙이는 연산에서는 일반적인 리스트에 비해 느림
 - 필요할 때마다 메모리 탐색 과정으로 새로운 공간을 잡아야 하기 때문

03 넘파이 배열 연산

[하나 더 알기] 넘파이의 성능

```
def scalar_vector_product(scalar, vector):
    result = []
    for value in vector:
        result.append(scalar * value)
    return result

iteration_max = 100000000

vector = list(range(iteration_max))
scalar = 2

# for문을 이용한 성능
%timeit scalar_vector_product(scalar, vector)
26.8 s ± 259 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
    평균 ± 표준편차
    7회 실행

# 리스트 컴프리헨션을 이용한 성능
%timeit [scalar * value for value in range(iteration_max)]
21.9 s ± 85.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

# 넘파이를 이용한 성능
%timeit np.arange(iteration_max) * scalar
561 ms ± 19.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

그림 3-22 넘파이의 성능 측정

04 비교 연산과 데이터 추출

1. 비교 연산

- 연산 결과는 항상 불린형(boolean type)을 가진 배열로 추출

1.1 브로드캐스팅 비교 연산

- 하나의 스칼라 값과 벡터 간의 비교 연산은 벡터 내 전체 요소에 적용

```
In [1]: import numpy as np  
x = np.array([4, 3, 2, 6, 8, 5])  
x > 3
```

```
Out [1]: array([ True, False, False, True, True, True])
```

04 비교 연산과 데이터 추출

1.2 요소별 비교 연산

- 두 개의 배열 간 배열의 구조(shape)가 동일한 경우
- 같은 위치에 있는 요소들끼리 비교 연산
- $[1 > 2, 3 > 1, 0 > 7]$ 과 같이 연산이 실시된 후 이를 반환

In [2]:	<pre>x = np.array([1, 3, 0]) y = np.array([2, 1, 7]) x > y</pre>
Out [2]:	<pre>array([False, True, False])</pre>

04 비교 연산과 데이터 추출

2. 비교 연산 함수

2.1 all과 any

- all 함수 : 배열 내부의 모든 값이 참일 때는 True,
하나라도 참이 아닐 경우에는 False를 반환
 - and 조건을 전체 요소에 적용
- any 함수 : 배열 내부의 값 중 하나라도 참일 때는 True,
모두 거짓일 경우 False를 반환
 - or 조건을 전체 요소에 적용

04 비교 연산과 데이터 추출

In [3]:	x = np.array([4, 6, 7, 3, 2]) (x > 3)
Out [3]:	array([True, True, True, False, False])

- x > 3 브로드캐스팅이 적용되어 불린형으로 이루어진 배열 반환

In [4]:	(x > 3).all()
Out [4]:	False
In [5]:	(x > 3).any()
Out [5]:	True

- all 함수를 적용하면 2개의 거짓이 있기 때문에 False를 반환
- any 함수를 적용하면 참이 있기 때문에 True를 반환

04 비교 연산과 데이터 추출

In [6]:	(x < 10).any()
Out [6]:	True
In [7]:	(x < 10).all()
Out [7]:	True
In [8]:	(x > 10).any()
Out [8]:	False

- $x > 10$ 의 경우 모든 값이 10을 넘지 못하므로 모두 거짓인데, 여기에 any 함수를 적용하면 False를 반환

04 비교 연산과 데이터 추출

2.2 인덱스 반환 함수

- where 함수 : 배열이 불린형으로 이루어졌을 때
참인 값들의 인덱스를 반환

In [9]:	x = np.array([4, 6, 7, 3, 2]) x > 5
Out [9]:	array([False, True, True, False, False])
In [10]:	np.where(x>5)
Out [10]:	(array([1, 2], dtype=int64),)

- x > 5를 만족하는 값은 6과 7
- 6과 7의 인덱스 값인 [1, 2]를 반환

04 비교 연산과 데이터 추출

- True/False 대신 참/거짓인 경우의 값을 지정할 수 있음

In [11]:	x = np.array([4, 6, 7, 3, 2]) np.where(x>5 , 10, 20)
Out [11]:	array([20, 10, 10, 20, 20])

- 참일 경우에 10을, 거짓일 경우에 20을 반환

04 비교 연산과 데이터 추출

2.3 정렬된 값의 인덱스를 반환해주는 함수

- `argsort` : 배열 내 값들을 작은 순서대로 인덱스를 반환
- `argmax` : 배열 내 값들 중 가장 큰 값의 인덱스를 반환
- `argmin` : 배열 내 값들 중 가장 작은 값의 인덱스를 반환

In [12]:	<code>x = np.array([4, 6, 7, 3, 2])</code> <code>np.argsort(x)</code>
Out [12]:	<code>array([4, 3, 0, 1, 2], dtype=int64)</code>
In [13]:	<code>np.argmax(x)</code>
Out [13]:	2
In [14]:	<code>np.argmin(x)</code>
Out [14]:	4

04 비교 연산과 데이터 추출

3. 인덱스를 활용한 데이터 추출

3.1 불린 인덱스

- 불린 인덱스(boolean index) : 배열에 있는 값들을 반환할 특정 조건을 불린형의 배열에 넣어서 추출
 - 인덱스에 들어가는 배열은 불린형이어야 함
 - 불린형 배열과 추출 대상이 되는 배열의 구조가 같아야 함

04 비교 연산과 데이터 추출

In [15]:	<pre>x = np.array([4, 6, 7, 3, 2]) x > 3</pre>
Out [15]:	array([True, True, True, False, False])
In [16]:	<pre>cond = x > 3 x[cond]</pre>
Out [16]:	array([4, 6, 7])
In [17]:	x.shape
Out [17]:	(5,)
In [18]:	cond.shape
Out [18]:	(5,)

04 비교 연산과 데이터 추출

3.2 팬시 인덱스

- 팬시 인덱스(fancy index) : 정수형 배열의 값을 사용하여 해당 정수의 인덱스에 위치한 값을 반환
 - 인덱스 항목에 넣을 배열은 정수로만 구성되어야 함
 - 정수 값의 범위는 대상이 되는 배열이 가지는 인덱스의 범위 내 대상이 되는 배열과 인덱스 배열의 구조(shape)가 같을 필요는 없음

In [19]:	x = np.array([4, 6, 7, 3, 2]) cond = np.array([1, 2, 0, 2, 2, 2], int) x[cond]
Out [19]:	array([6, 7, 4, 7, 7, 7])

04 비교 연산과 데이터 추출

In [20]:	x.take(cond)
Out [20]:	array([6, 7, 4, 7, 7, 7])
In [21]:	<pre>x = np.array([[1,4], [9,16]], int) a = np.array([0, 1, 1, 1, 0, 0], int) b = np.array([0, 0, 0, 1, 1, 1], int) x[a,b]</pre>
Out [21]:	array([1, 9, 9, 16, 4, 4])

	0	1
0	1	4
1	9	16

그림 3-23 팬시 인덱스