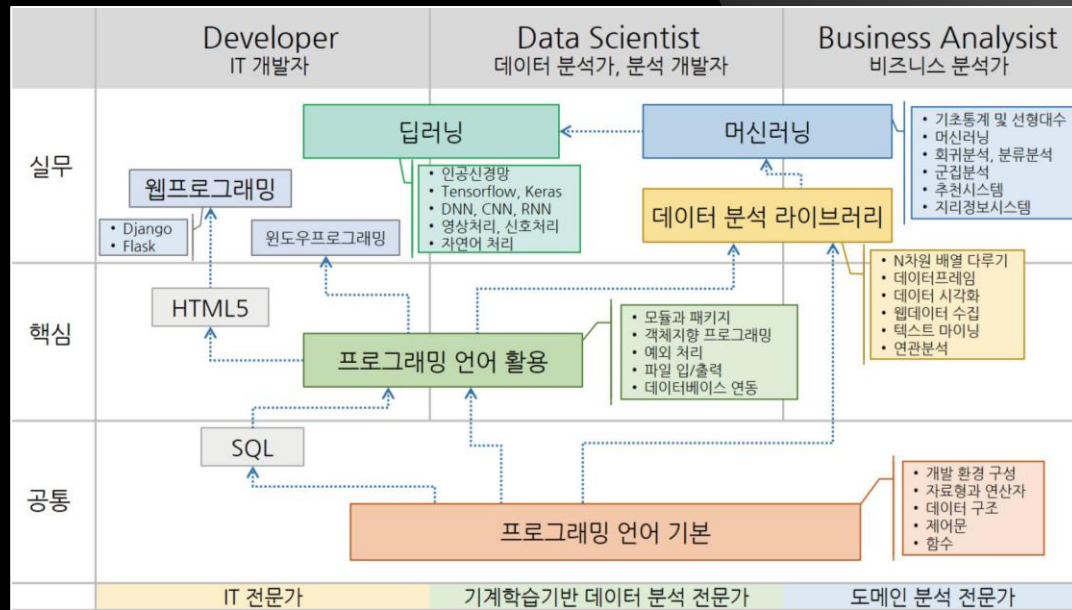


# 파이썬 데이터 전처리 및 탐색 라이브러리



A4 Screen

이 슬라이드에서 사용한 서체 :

- Open Sans(<https://ko.cooltext.com/Download-Font-Open+Sans>)
- KoPubWorld돋움체(<http://www.kopus.org/biz/electronic/font.aspx>)

- #1 N차원 배열 다루기
- #2 데이터프레임과 시리즈
- #3 데이터 시각화
- #4 웹 데이터 수집
- #5 텍스트 마이닝
- #6 연관분석





# 1장. N차원 배열 다루기 - Numpy

넘파이 패키지를 이용해서 N차원 배열을 만들고 데이터를 다루는 방법에 대해 설명합니다.

- 1절. 넘파이 패키지
- 2절. 넘파이 배열
- 3절. 배열 합치기/분할하기
- 4절. 복사와 뷰
- 5절. 고급 인덱싱
- 6절. 선형대수학
- 7절. 유용한 정보 및 팁



# THE PYTHON - 파이썬 프로그래밍

3부. 데이터 분석 라이브러리 활용

허진경 지음

BOOKK 

## 1장. N차원 배열 다루기 - Numpy



### 1절. 넘파이 패키지

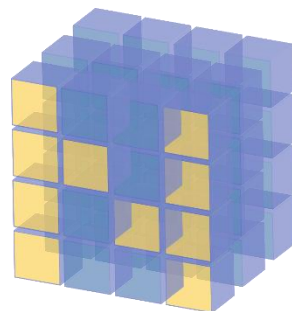
넘파이 : 넘파이(Numpy)는 파이썬을 사용한 **과학 컴퓨팅**의 기본 패키지

동종 모음 : 넘파이의 주요 객체는 **동종(homogeneous)**의 다차원 배열

넘파이 기능 : N 차원 배열 객체 생성 및 관리  
선형 대수학(Linear algebra), 푸리에 변환(Fourier transform) 기능  
난수(Random number) 생성 기능

축(Axis) : 넘파이의 차원들은 **축(axis)**으로 불림

공식 사이트 : <https://www.numpy.org/>



NumPy

용도	넘파이 주요 함수
배열 만들기	arange, array, copy, empty, empty_like, eye, fromfile, fromfunction, identity, linspace, logspace, mgrid, ogrid, ones, ones_like, r, zeros, zeros_like
모양 바꾸기	ndarray.astype, atleast_1d, atleast_2d, atleast_3d, mat
배열 조작하기	array_split, column_stack, concatenate, diagonal, dsplit, dstack, hsplit, hstack, ndarray.item, newaxis, ravel, repeat, reshape, resize, squeeze, swapaxes, take, transpose, vsplit, vstack
찾기	all, any, nonzero, where
정렬하기	argmax, argmin, argsort, max, min, ptp, searchsorted, sort
배열 운영하기	choose, compress, cumprod, cumsum, inner, ndarray.fill, imag, prod, put, putmask, real, sum
기초 통계	cov, mean, std, var
선형 대수	cross, dot, outer, linalg.svd, vdot

## 01

## ndarray 속성

```
1 A = np.arange(15).reshape(3,5)
```

```
2 A
```

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

`ndarray.ndim` : 배열의 축(Axis) 수, 차원 `A.ndim → 2`

`ndarray.shape` : 각 차원의 배열 크기를 나타내는 정수 타입의 튜플.  
`shape`는 (n, m) 형태. 행렬은 n개의 행과 m개의 열.  
`shape` 튜플의 길이는 축의 수(ndim).

`A.shape → (3, 5)`

`darray.size` : 배열의 요소의 총수. `shape`의 각 요소의 곱과 동일. `A.size → 15`

`ndarray.dtype` : 배열 내의 요소의 타입. 파이썬의 자료형 또는 넘파이의 자료형(`numpy.int32`, `numpy.int16`, `numpy.float64` 등)을 이용해 지정함.  
**형변환의 개념이 아님.** 지정한 타입의 크기만큼 잘라서 해당 타입으로 인식 함.  
**형변환은 `astype(t)` 함수를 이용**

`A.dtype → dtype('int32')`

`ndarray.itemsize` : 배열의 각 요소의 바이트 단위의 사이즈. Ex) `float64` 유형의 요소 배열에는 `itemsize 8(=64/8)`, `complex32` 유형에는 `itemsize 4(=32/8)`가 있음.  
 이것은 `ndarray.dtype.itemsize`과 같음.

`A.itemsize → 4`

```
1 A = np.arange(15).reshape(3,5)
```

```
2 A
```

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

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

```
2 A
```

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

```
1 A.dtype
```

```
dtype('int32')
```

```
1 A.dtype = np.int64
```

```
2 A
```

```
array([[ 4294967296, 12884901890],
       [21474836484, 30064771078],
       [38654705672, 47244640266]], dtype=int64)
```



- dtype 속성으로 타입을 지정하면 기존의 데이터들이 새로 바뀌는 타입의 크기만큼 읽혀짐
- 예에서 처음의 배열은 shape가 (3, 4) 이고 아이템들의 타입이 int32였다면 이후 dtype을 int64로 지정하면 정수 두 개 값이 하나의 정수로 읽혀 짐
- 이 과정에서 배열의 처음 0과 1은 1과 0값으로 순서가 바뀌어 결합되고 이것이 64비트로 표현되어 정수를 계산하므로 1(00000000 00000000 00000000 00000001)과 0이 결합되어 새로운 2진수 00000000 00000000 00000000 00000000 00000001 00000000 00000000 00000000 00000000 이 만들어 지는데 이것을 10진수로 변환하면 4294967296가 됨
- 마찬가지로 int64 타입으로 변환된 두 번째 데이터는 2(00000000 00000000 00000000 00000010)와 3(00000000 00000000 00000000 00000011)이 순서가 바뀌어 결합되어 00000000 00000000 00000000 00000000 00000011 00000000 00000000 00000000 00000010이 만들어지고 이것을 10진수로 변환해서 12884901890가 됨





# THE PYTHON - 파이썬 프로그래밍

3부. 데이터 분석 라이브러리 활용

허진경 지음

BOOKK 

## 1장. N차원 배열 다루기 - Numpy



### 2절. 넘파이 배열



## 넘파이 배열 만드는 방법

- ▶ `array()` 함수를 이용한 다른 파이썬 구조(예: 리스트, 튜플)로부터의 변환
- ▶ 넘파이 배열을 생성하는 함수 이용(예: `arange`, `ones`, `zeros`, 등.)
- ▶ 표준 형식 또는 사용자 정의 형식으로 디스크에서 배열 읽기
- ▶ 문자열이나 버퍼를 사용하여 원시 바이트에서 배열 만들기
- ▶ 특수 라이브러리 함수(예: `random`) 이용



참고 : <https://docs.scipy.org/doc/numpy/index.html>

`array()` 함수를 사용하여 리스트 또는 튜플로부터 넘파이 배열을 생성

```
numpy.array(object, dtype=None, copy=True)
```

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

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

```
1 A.dtype
```

```
dtype('int32')
```

결과 배열의 유형은  
리스트 또는 튜플의  
요소 유형에서 추론됨

```
1 B = np.array([1.2, 3.5, 5.1])
2 B.dtype
```

```
dtype('float64')
```

```
1 C = np.array([[1,2], [3,4]], dtype=complex)
2 C
```

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

```
1 D = np.array(C, copy=False)
```

```
1 id(C), id(D)
```

```
(135970128, 135970128)
```

copy=False이고 dtype을  
지정하지 않으면 새로운 배열의  
복사본을 생성하지 않음

초기 특정 값으로 채워진 배열을 만드는 몇 가지 기능을 제공

✓ `empty()` : 초기 내용이 임의이고 메모리의 상태에 따라 달라지는 배열

✓ `zeros()` : 0으로 채워진 배열

✓ `ones()` : 1로 구성된 배열

✓ 배열의 dtype은 float64

✓ 원하면 dtype 속성으로 타입을 지정 가능

```
1 np.zeros( (3,4) )
```

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

```
1 np.ones( (2,3,4), dtype=np.int16 )
```

```
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]]], dtype=int16)
```

```
1 np.empty( (2,3) )
```

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

`numpy.arange([from, ]to, [by, ]dtype=None)`



from부터 to까지(포함 안 함) by씩 건너뛴 값 목록을 생성

```
1 np.arange( 10, 30, 5 )
```

```
array([10, 15, 20, 25])
```

```
1 np.arange( 0, 2, 0.3 )
```

```
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
```



많은 수의 점 데이터를  
만들 때 유용

`numpy.linspace(begin, end, num=50)`



begin부터 end까지(포함) num개 목록을 생성

```
1 np.linspace( 0, 2, 9 )
```

```
array([0. , 0.25, 0.5 , 0.75, 1. , 1.25, 1.5 , 1.75, 2. ])
```

```
1 x = np.linspace( 0, 2*np.pi, 100 )
```

```
2 f = np.sin(x)
```

배열은 각 축을 따라 요소 수만큼 주어진 차원을 가짐

```
1 A = np.arange(12).reshape(3,4)
2 A
```

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

```
1 A.shape
```

```
(3, 4)
```

```
1 A.ravel()
```

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

```
1 A.reshape(6,2)
```

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

```
1 A.T
```

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

```
1 A.resize(2,6)
```

```
1 A
```

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

```
1 A.shape = (3,4)
```

```
1 A
```

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



- `ravel()` : 차원이 풀린 배열을 반환
- `reshape(6, 2)` : shape가 수정된 배열을 반환
- `T` : 전치행렬(transposed) 반환
- `resize()` : 현재 배열 객체를 수정
- `shape` : shape를 튜플형식으로 지정

```
1 A.reshape(4,-1)
```

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

reshape 작업에서 크기가 -1로 주어지면 해당 차원의 크기는 자동으로 계산

```
1 A.resize(4,-1)
```

```
ValueError
cent call last)
```

```
on-input-198-1c8130cdf291> in <module>
1 A.resize(4,-1)
```

```
error: negative dimensions not allowed
```

resize에서는 음수 사용 못함

Traceback (most re

## 배열을 인쇄 할 때 레이아웃

- ✓ 마지막 축은 왼쪽에서 오른쪽으로 인쇄
- ✓ 나머지는 위에서 아래로 인쇄
- ✓ 각 슬라이스는 빈 줄로 구분

## 1차원 배열은 행, 2차원은 행렬, 3차원은 행렬 목록으로 인쇄

## reshape() 함수는 배열의 모양(shape)을 변경

```
1 A = np.arange(6)
2 A

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

```
1 B = np.arange(12).reshape(4,3)
2 B

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

```
1 C = np.arange(24).reshape(2,3,4)
2 C

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]])
```



- `numpy.set_printoptions(threshold=None)`
- 넘파이가 전체 배열을 인쇄하도록 하려면 `set_printoptions()` 함수로 인쇄 옵션 변경  
`np.set_printoptions(threshold=1000)`  
`print(np.arange(1000).reshape(10,100))`



(2, 3, 4)는 (2, 3)모양  
배열이 4개가 있는 것이  
아니고 (3, 4)모양  
배열이 2개 있음을 의미

## 배열 연산 시 산술 연산자는 요소별로 적용

```
1 A = np.array([20,30,40,50])
2 B = np.arange(4)
3 B
```

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

```
1 A-B
```

```
array([20, 29, 38, 47])
```

```
1 B**2
```

```
array([0, 1, 4, 9], dtype=int32)
```

```
1 10*np.sin(A)
```

```
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
```

```
1 A<35
```

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

## 행렬의 곱은 dot() 또는 @ 연산자(3.5 이상)

```
1 A = np.array( [[1,1],
2                [0,1]] )
3 B = np.array( [[2,0],
4                [3,4]] )
```

```
1 A*B
```

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

\* 연산자는 요소별로 계산

```
1 A@B
```

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

행렬의 곱

@ 연산자는 파이썬 3.5  
버전부터 사용 가능

```
1 A.dot(B)
```

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



`+=` 와 `*=` 등 복합대입연산자들은 새 배열을 생성하지 않고 기존 배열을 수정하기 위해 사용

```
1 A = np.ones((2,3), dtype=int)
2 B = np.random.random((2,3))
```

```
1 A *= 3
2 A
```

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

```
1 B += 3
2 B
```

```
array([[3.49401053, 3.91777491, 3.5583599 ],
       [3.83327187, 3.64503634, 3.39717144]])
```

```
1 A += B
```

```
UFuncTypeError
cent call last)
<ipython-input-222-004333d39f34> in <module>
----> 1 A += B
```

```
UFuncTypeError: Cannot cast ufunc 'add' output from dtype
('float64') to dtype('int32') with casting rule 'same_kind'
```



넘파이 배열의 타입은 자동으로 하향 형 변환 (down casting) 되지 않기 때문에 a의 타입이 a+b의 결과를 저장할 수 없는 타입이라면 에러가 발생

Traceback (most re

배열의 합 계산과 같은 단항 연산은 **ndarray 클래스의 메서드로 구현되어 있음**

```
1 A = np.random.random((2,3))
```

```
2 A
```

```
array([[0.48978832, 0.05429558, 0.58571083],  
       [0.82169344, 0.64340243, 0.18859382]])
```

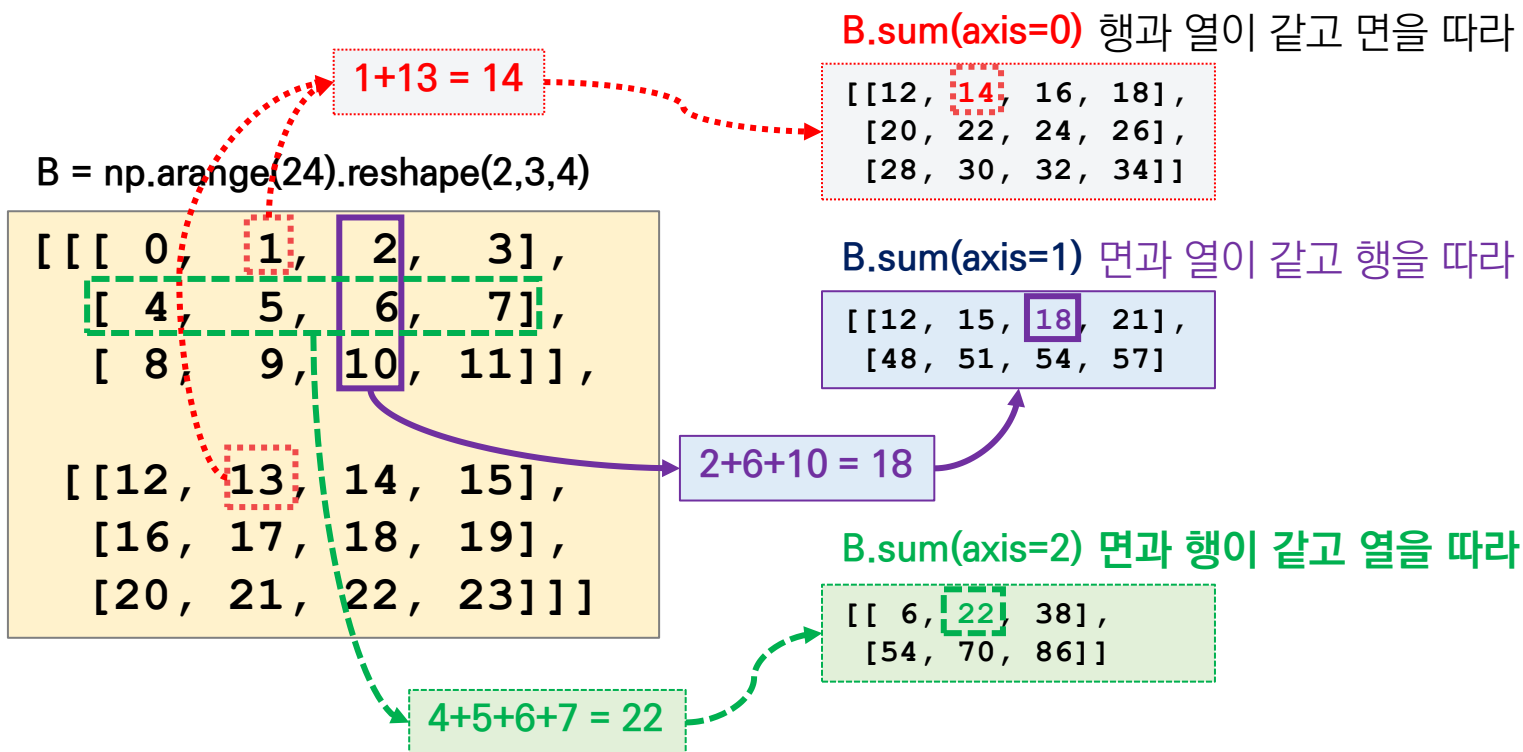
```
1 A.sum(), A.min(), A.max()
```

```
(2.7834844216425125, 0.05429558317460692, 0.8216934430028698)
```



sum()은 배열의 모든 요소의 합,  
min()은 배열의 요소들 중에서 가장 작은 값,  
max()은 가장 큰 값을 반환함

axis 매개 변수를 지정하면 배열의 지정된 **축을 따라 작업을 적용** 함



수학 함수, 삼각 함수, 비트 함수, 비교 함수, 부동 함수 등이 있음

배열의 각 요소마다 적용되어 배열을 출력함

```
1 B = np.arange(3)
```

```
1 np.exp(B)
```

```
array([1.          , 2.71828183, 7.3890561 ])
```

```
1 np.sqrt(B)
```

```
array([0.          , 1.          , 1.41421356])
```

```
1 C = np.array([2., -1., 4.])
```

```
2 np.add(B, C)
```

```
array([2., 0., 6.])
```

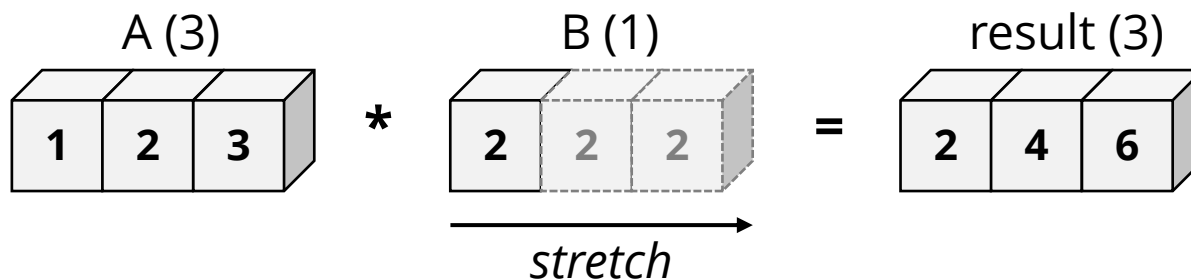


- 범용함수들은 함수의 마지막 인수로 결과를 저장할 변수를 지정할 수 있음
  - `function_name(x1, x2[, output])`
- 출력 인수의 지정은 많은 양의 데이터를 연산해야 할 경우 메모리를 절약할 수 있음
- 다음 수식들은 모두 동일함(A,B,C 변수 선언됨)
  - `G = A * B + C`
  - `T1 = A * B; G = T1 + C; del T1`
  - `G = A * B; add(G, C, G)`
  - `G = A * B; G += C`

참고 : <https://docs.scipy.org/doc/numpy/reference/ufuncs.html>

가장 간단한 예제는 배열과 스칼라 값이 연산에서 결합될 때 발생

연산 시 메모리 사용량을 줄임



```
1 A = np.array([1,2,3])
2 B = np.array([2,2,2])
```

```
1 A * B
```

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

```
1 np.multiply(A, B)
```

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

```
1 A = np.array([1,2,3])
2 B = 2
```

```
1 A * B
```

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

```
1 np.multiply(A, B)
```

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

브로드 캐스팅 예

‘연산의 두 배열에 대한 후미 축의 크기가 같거나 둘 중 하나가 1이어야 한다.’

```
1 A = np.array([[ 0,  0,  0],
2               [10, 10, 10],
3               [20, 20, 20],
4               [30, 30, 30]])
5 B = np.array([1, 2, 3])
```

```
1 A + B
```

```
array([[ 1,  2,  3],
       [11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```

A (4x3)

0	0	0
10	10	10
20	20	20
30	30	30

B (3)

0	1	2
0	1	2
0	1	2
0	1	2

+

stretch

=

result (4x3)

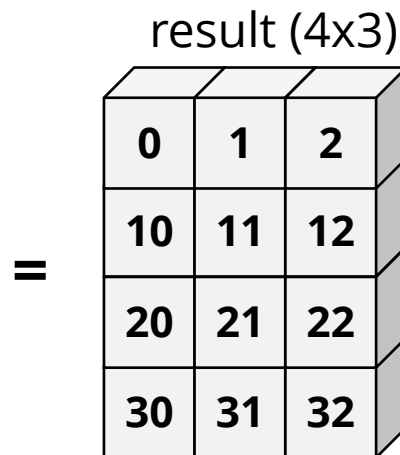
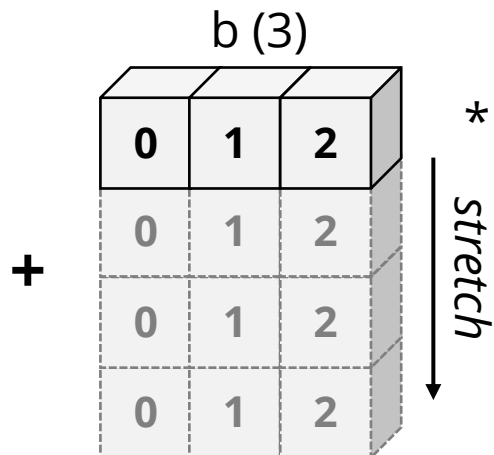
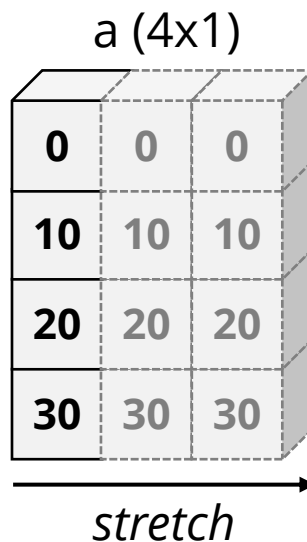
0	1	2
10	11	12
20	21	22
30	31	32

두 배열의 바깥 쪽(또는 다른 모든 바깥 쪽) 작업을 수행하는 편리한 방법을 제공함

```
1 A = np.array([0.0, 10.0, 20.0, 30.0])
2 B = np.array([1.0, 2.0, 3.0])
```

```
1 A[:, np.newaxis] + B
```

```
array([[ 1.,  2.,  3.],
       [11., 12., 13.],
       [21., 22., 23.],
       [31., 32., 33.]])
```





# THE PYTHON - 파이썬 프로그래밍

3부. 데이터 분석 라이브러리 활용

허진경 지음

BOOKK 

## 1장. N차원 배열 다루기 - Numpy



### 3절. 배열 합치기/분할하기



맨 처음 요소의 인덱스는 0이며 이후로 1씩 증가하도록 양수로 지정하는 것이 일반적임

맨 뒤의 항목부터 음수를 이용해 지정할 수 있음

```
1 A = np.arange(10)**3
```

```
2 A
```

```
array([ 0,  1,  8, 27, 64, 125, 216, 343, 512, 729], dtype=int32)
```

0	1	8	27	64	128	216	343	512	729
---	---	---	----	----	-----	-----	-----	-----	-----

데이터

0    1    2    3    4    5    6    7    8    9

양수 인덱스

-10   -9   -8   -7   -6   -5   -4   -3   -2   -1

음수 인덱스

`np_array_obj[index]`

```
1 A = np.arange(10)**3
```

```
2 A
```

```
array([ 0,  1,  8, 27, 64, 125, 216, 343, 512, 729], dtype=int32)
```

```
1 A[2]
```

양수 인덱스로 인덱싱

```
8
```

```
1 A[-2]
```

음수 인덱스로 인덱싱

```
512
```

```
1 A[10]
```

인덱스 범위를 벗어나면 오류 발생

```
---
IndexError
```

Traceback (most recent call last)

```
<ipython-input-240-1d70858cd699> in <module>
```

```
----> 1 A[10]
```

```
IndexError: index 10 is out of bounds for axis 0 with size 10
```

`np_array_obj[ from : to ]` : from부터 to까지 (to 포함 안함)

`np_array_obj[ from : to : by ]` : from부터 to까지 by 마다 (to 포함 안함)

```
1 A[2:5]
```

```
array([ 8, 27, 64], dtype=int32)
```

```
1 A[0:9:2]
```

```
array([ 0,  8, 64, 216, 512], dtype=int32)
```

```
1 A[::2]
```

```
array([ 0,  8, 64, 216, 512], dtype=int32)
```

from과 to를 생략하면 자동 지정됨

```
1 A[::-1]
```

```
array([729, 512, 343, 216, 125, 64, 27,  8,  1,  0], dtype=int32)
```

```
1 A[6:2] = 1000
```

```
2 A
```

```
array([1000,  1, 1000,  27, 1000, 125, 216, 343, 512, 729],
      dtype=int32)
```

할당문을 가지면 값을 변경할 수 있음

2차원 배열 인덱싱 : `np_array_obj[0축인덱스, 1축인덱스]`

3차원 배열 인덱싱 : `np_array_obj[0축인덱스, 1축인덱스, 2축인덱스]`

```
B = np.arange(20).reshape(5,4)
B
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

```
B[2,3]
```



2행, 3열 데이터

11

```
B[-3, -1]
```



-3행, -1열 데이터

11

```
C = np.arange(24).reshape(2,3,4)
C
```

```
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]]])
```

```
C[0,1,2]
```

6

```
C[1,2,3]
```

23

3차원 배열 인덱싱

[0,1,2] 인덱스

```
[[[ 0,  1,  2,  3],
   [ 4,  5,  6,  7],
   [ 8,  9, 10, 11]],
 [[12, 13, 14, 15],
  [16, 17, 18, 19],
  [20, 21, 22, 23]]]
```

[1,2,3] 인덱스

[1,2,3]은 깊이 1이며, 2행, 3열인 데이터

`np_array_obj[ from:to ]` 형식과 `np_array_obj[ from:to:by ]` 형식 사용

차원이 여러 개 인 경우 콤마로 구분해서 각 차원 별로 from, to 인덱스를 지정

1 `B[0:5, 1]`

`array([ 1, 5, 9, 13, 17])`

💡 0행부터 5행까지  
1열 데이터

1 `B[:, 1]`

`array([ 1, 5, 9, 13, 17])`

💡 모든 행, 1열  
데이터

1 `B[1:3, :]`

`array([[ 4, 5, 6, 7],  
[ 8, 9, 10, 11]])`

💡 1행부터 3행까지  
모든 열 데이터

[	0,	1,	2,	3],	
[	4,	5,	6,	7],	
[	8,	9,	10,	11],	
[	12,	13,	14,	15],	
[	16,	17,	18,	19]]	

→ `[0:5, 1]`

축의 수보다 더 적은 수의 인덱스가 제공되면 누락 된 인덱스는 모든 항목을 선택함

넘파이에서 도트를 사용하여 B[i,...] 형식으로 작성

B[-1]

```
array([16, 17, 18, 19])
```

C[0]

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

C[0,0]

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

C[0,...]

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

C[:, :, 0]

```
array([[ 0,  4,  8],
       [12, 16, 20]])
```

C[... , 0]

```
array([[ 0,  4,  8],
       [12, 16, 20]])
```



...은 나머지 축들의  
모든 항목을 선택



3차원 배열의 경우  
[:, :, i]과 [... , i]은 같음

다차원 배열의 반복문에 의한 처리는 첫 번째 축을 기준으로 수행됨

```
for row in B:  
    print(row)
```

```
[0 1 2 3]  
[4 5 6 7]  
[ 8  9 10 11]  
[12 13 14 15]  
[16 17 18 19]
```

```
for element in B.flat:  
    print(element, end=" ")
```



배열의 flat 속성을 사용하면  
하나의 반복문으로 모든 배열  
요소에 연산을 수행할 수 있음

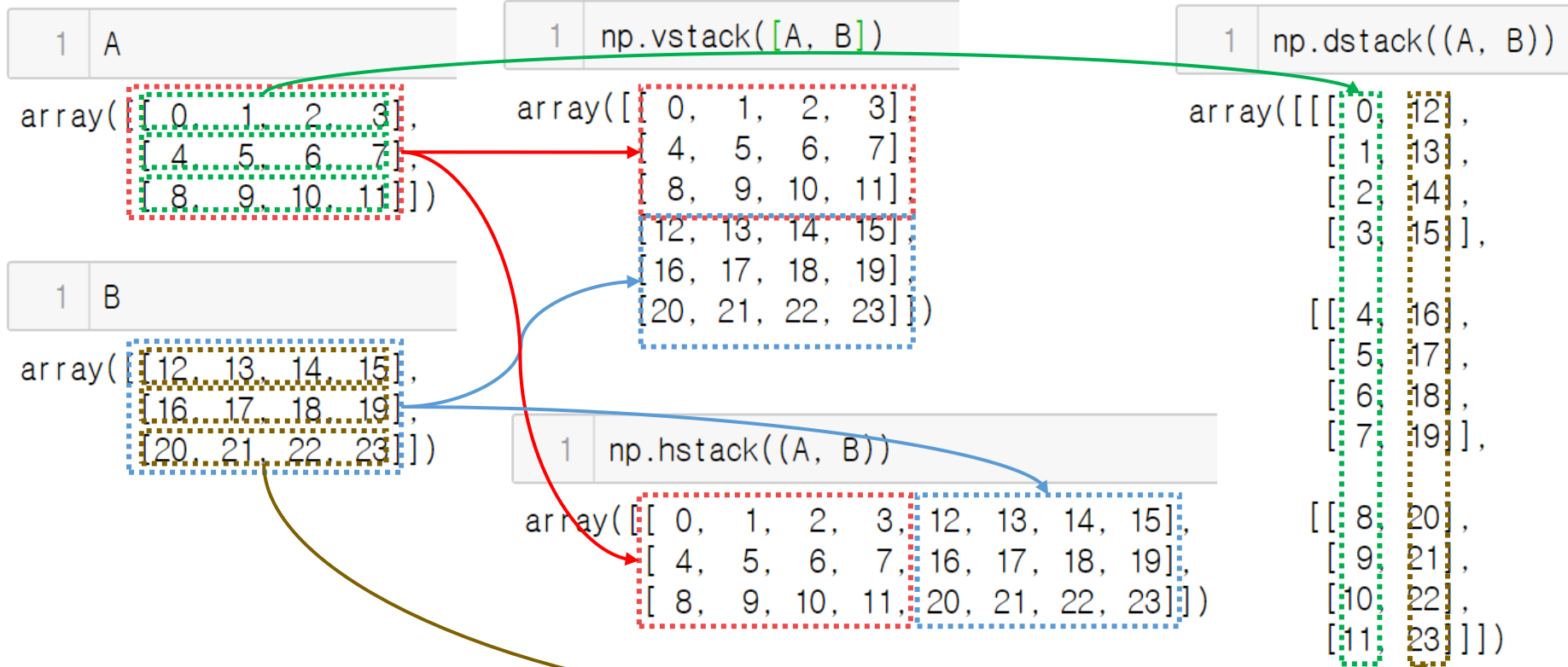
```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

# 03 쌓기 – hstack(), vstack(), dstack()

**hstack()** : 배열을 옆에 추가하는 방식으로 쌓아 합침

**vstack()** : 배열을 아래에 추가하는 방식으로 쌓아 합침

**dstack()** : 3번째 축(depth)을 따라 쌓아 합침





column\_stack() 함수는 1차원 배열을 열 단위로 배열하여 2차원 배열을 만듦

```
1 A = np.array((1,2,3,4))
2 B = np.array((5,6,7,8))
3 C = np.array((9,10,11,12))
```

```
1 np.column_stack((A,B,C))
```

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

column\_stack() 함수는  
1차원 배열들을 입력받아  
2차원 배열을 만듦

```
1 np.hstack((A,B))
```

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

hstack() 함수는 1차원  
배열을 입력받으면 그  
결과는 1차원



- hstack()을 이용해 1차원 배열을 열 단위로 쌓으려면 newaxis를 이용해서 1차원 배열이 2차원 구조가 되도록 해야 함
- newaxis 속성은 2차원 컬럼 벡터를 갖도록 함

```
1 A[:, np.newaxis]
```

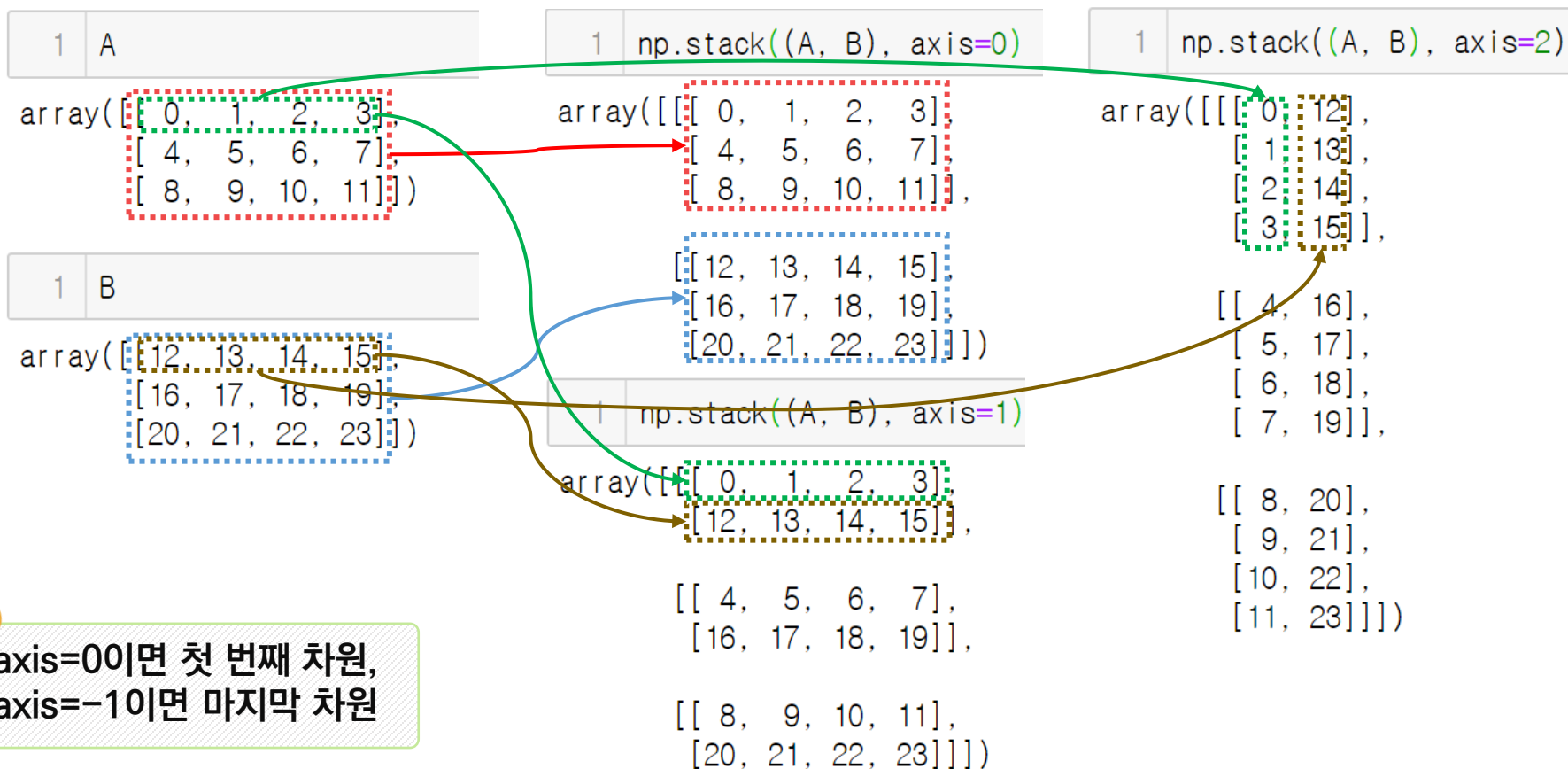
```
array([[1],
       [2],
       [3],
       [4]])
```

```
1 np.hstack((A[:, np.newaxis],
2             B[:, np.newaxis]))
```

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

stack() 함수는 축 속성 axis의 값에 따라 배열을 합침

axis 매개 변수는 결과의 차원에서 새 축의 인덱스를 지정



r\_[] 객체와 c\_[] 객체는 한 개의 축을 따라 데이터를 나열해 배열을 만들 때 유용

```
1 A = np.array((1,2,3,4))
2 B = np.array((5,6,7,8))
3 C = np.array((9,10,11,12))
```

```
1 np.r_[A,B,C]
```

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

```
1 np.c_[A,B,C]
```

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



c\_() 함수는 열 단위로 데이터를 쌓아 준다.

# 03 나누기 – vsplit(), hsplit(), dsplit()

**vsplit()** : 수직 축을 따라 나눔. 배열을 위/아래로 나눔

**hsplit()** : 수평 축을 따라 나눔. 배열을 좌/우로 나눔

**dsplit()** : depth를 따라 나눔. dsplit()은 3차원 이상 배열에서만 동작

1	A	원본 데이터
		<pre>array([[ 0,  1,  2,  3],         [ 4,  5,  6,  7],         [ 8,  9, 10, 11]])</pre>

```
1 np.vsplit(A, 3)
```

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

```
1 np.vsplit(A, 3)[0]
```

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

```
1 np.hsplit(A, 2)
```

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

vsplit(), hsplit(), dsplit()과 비슷하지만 split()은 axis 파라미터를 가질 수 있음

axis=0이면 vsplit(), axis=1인 경우 hsplit() 그리고 axis=2는 dsplit()과 동일하게 동작

```
1 A
```

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

```
1 np.split(A, 3, axis=0)
```

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

```
1 np.split(A, 2, axis=1)
```

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



vsplit(arr, n)은  
split(arr, n, axis=0)과  
같은 방식으로 동작

# 03 나누기 - axis=1

hsplit() 함수는 split() 함수의 axis 매개 변수가 1일 때와 같음

배열의 차원에 상관없이 항상 두 번째 축을 이용해 분할

```
1 C = np.arange(24).reshape(2,3,4)
```

```
1 C
```

```
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]]
```

```
1 c_hsplit = np.hsplit(C, 3)
2 c_hsplit
```

```
[array([[[ 0,  1,  2,  3]],
       [[12, 13, 14, 15]]]),
```

```
array([[[ 4,  5,  6,  7]],
       [[16, 17, 18, 19]]]),
```

```
array([[[ 8,  9, 10, 11]],
       [[20, 21, 22, 23]])])
```



hsplit(arr, n)은  
split(arr, n, axis=1)과  
같은 방식으로 동작

# 03 나누기 – 인덱스 목록으로 나누기

분할하기 위한 인수를 튜플 형식으로 지정하면 해당 인덱스를 기준으로 나눔

```
1 A = np.arange(24).reshape(3,8)
2 A
```

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]])

index                      2                      5                      6

```
1 a_hsplit = np.hsplit(A, (2,5,6))
```

첫 인덱스부터 튜플로 지정한  
각각의 인덱스까지 각각 부분  
집합을 생성함

(2,5,6)의 경우 (0~2, 2~5,  
5~6, 6~마지막)에 해당하는  
부분집합이 생성됨

```
1 a_hsplit[0]
```

array([[ 0, 1],  
 [ 8, 9],  
 [16, 17]])

처음부터 2열까지

```
1 a_hsplit[1]
```

array([[ 2, 3, 4],  
 [10, 11, 12],  
 [18, 19, 20]])

2열부터 5열까지

```
1 a_hsplit[2]
```

array([[ 5],  
 [13],  
 [21]])

5열부터 6열까지

```
1 a_hsplit[3]
```

array([[ 6, 7],  
 [14, 15],  
 [22, 23]])

6열부터 끝까지

split()과 차이점은 **indices\_or\_sections**를 **균등 분할하지 않아도 사용할 수 있다는 것**

**n개로 나눠야 하는 길이 1 배열은  $(1//n)+1$  크기  $1\%n$ 개 배열과 나머지  $(1//n)$  배열 반환**

```
1 A = np.arange(10)
2 A
```

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

```
1 np.split(A, 4)
```

```
TypeError
~\Anaconda3\lib\site-packages\
_sections, axis)
```

```
777 try:
--> 778     len(indices_or_sections)
779 except TypeError:
```

ValueError: array split does not result in an equal division

10개 요소를 갖는 배열은  
4개의 부분 배열로  
균등하게 나뉘지지 않음

```
1 np.array_split(A, 4)
```

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

```
1 np.array_split(A, 3)
```

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

```
1 np.array_split(A, 6)
```

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

- **//**는 몫, **%**는 나머지
- 길이 10인 배열을 4개로 나누면  $10//4+1=3$  크기 배열 10%4=2개 배열과  $10//4=2$  크기 배열들 반환
- 길이 10인 배열을 3개로 나누면  $10//3+1=4$  크기 배열 10%3=1개 배열과  $10//3=3$  크기 배열들 반환
- 길이 10인 배열을 6개로 나누면  $10//6+1=2$  크기 배열 10%6=4개 배열과  $10//6=1$  크기 배열들 반환





# THE PYTHON - 파이썬 프로그래밍

3부. 데이터 분석 라이브러리 활용

허진경 지음

BOOKK 

## 1장. N차원 배열 다루기 - Numpy



### 4절. 복사와 뷰

단순한 할당은 배열 객체나 데이터의 사본을 만들지 않음

```
1 A = np.arange(12)
```

```
1 B = A
```

```
1 A
```

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

```
1 B
```

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

```
1 B.shape = (3,4)
```

```
1 A
```

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

 b의 shape를 바꾸면 a의 shape도 바뀐다

```
1 A.shape
```

```
(3, 4)
```

첫 행부터 매 2번째 행의 값을 0으로 바꿈

```
1 B[:,2] = 0
```

```
1 B
```

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

```
1 A
```

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

 b의 데이터를 바꾸면 a의 데이터도 바뀐다

뷰는 동일한 데이터를 공유 할 수 있는 다른 객체

view()는 동일한 데이터를 보는 새로운 배열 객체를 생성

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

```
1 A
```

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

```
1 C = A.view()
```

```
1 C is A
```

False

```
1 C.flags.owndata
```

False

```
1 C.shape = (2,6)
```

```
1 A.shape, C.shape
```

```
((3, 4), (2, 6))
```

```
1 C
```

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

```
1 A
```

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

다른 모양 같은 데이터

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

```
1 A
```

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

```
1 S = A[:, 1:3]
```

```
1 S
```

```
array([[ 1,  2],
       [ 5,  6],
       [ 9, 10]])
```

```
1 S[:, 1] = 100
```

```
1 A
```

```
array([[ 0,  1, 100,  3],
       [ 4,  5, 100,  7],
       [ 8,  9, 100, 11]])
```

- 슬라이싱 하면 뷰가 반환
- s[:, :] 형식으로 배열을 자르면 뷰가 반환. 배열을 자르고 할당할 경우 원본 배열의 값이 바뀜

# 04 복사와 뷰 – 깊은 복사 카피(copy)

`copy()` 함수는 배열 및 해당 데이터의 전체 복사본을 생성

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

```
1 A
```

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

```
1 D = A.copy()
```

```
1 D is A
```

```
False
```



`copy()`에 의한  
복사는 새로운  
객체가 되므로 다른  
데이터와 다른  
모양을 가짐

```
1 D[0, :] = [10, 20, 30, 40]
```

```
1 D
```

```
array([[10, 20, 30, 40],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
1 A
```

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



# THE PYTHON - 파이썬 프로그래밍

3부. 데이터 분석 라이브러리 활용

허진경 지음

BOOKK 

## 1장. N차원 배열 다루기 - Numpy



### 5절. 고급 인덱싱

배열의 인덱싱을 단일 숫자가 아닌 넘파이 배열을 이용

```
1 import numpy as np
```

```
1 A = np.arange(12)**2
```

```
1 idx = np.array([1,1,3,8,5])
```

```
1 A[idx]
```

```
array([ 1,  1,  9, 64, 25], dtype=int32)
```

```
1 jdx = np.array([[3,4], [9,7]])
2 A[jdx]
```

```
array([[ 9, 16],
       [81, 49]], dtype=int32)
```

이미지 저장을  
팔레트의 인덱스  
배열을 이용해  
저장하는 예

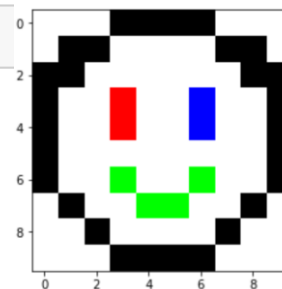
```
1 palette = np.array( [ [0,0,0],           # black
2                       [255,0,0],         # red
3                       [0,255,0],         # green
4                       [0,0,255],         # blue
5                       [255,255,255] ] )   # white
```

```
1 image_index = np.array([[4,4,4,0,0,0,0,4,4,4],
2                          [4,0,0,4,4,4,4,0,0,4],
3                          [0,0,4,4,4,4,4,4,0,0],
4                          [0,4,4,1,4,4,3,4,4,0],
5                          [0,4,4,1,4,4,3,4,4,0],
6                          [0,4,4,4,4,4,4,4,4,0],
7                          [0,4,4,2,4,4,2,4,4,0],
8                          [4,0,4,4,2,2,4,4,0,4],
9                          [4,4,0,4,4,4,4,0,4,4],
10                         [4,4,4,0,0,0,0,4,4,4]])
```

```
1 image_data = palette[image_index]
```

```
1 from matplotlib import pyplot as plt
2 %matplotlib inline
3 plt.imshow(image_data, interpolation='nearest')
4 plt.show()
```

인덱스 배열이  
다차원 인 경우,  
인덱스의 단일  
배열은 첫 번째  
차원을 참조



다차원 배열로 인덱스 정의. 각 차원에 대한 인덱스 배열은 동일한 모양이어야 함

```
1 import numpy as np
2 A = np.arange(12).reshape(3,4)
3 A
```

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

0행  
1행  
2행

```
1 ind_i = np.array([[0,1],
2                   [1,2]])
```

```
1 A[ind_i]
```

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

```
1 A[ind_i, :]
```

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

0열 1열 2열 3열

```
1 ind_j = np.array([[2,1],
2                   [3,3]])
```

```
1 A[ind_i, ind_j]
```

```
array([[ 2,  5],
       [ 7, 11]])
```

## 배열을 대상으로 인덱싱을 사용해서 값 변경 가능

```
1 A = np.arange(5)
```

```
1 A
```

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

```
1 A[[1,3,4]] = 0
```

```
1 A
```

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

```
1 A[1,3,4] = 10
```

```
IndexError
e cent call last)
<ipython-input-7-d8ee560b88db> in <module>
--> 1 A[1,3,4] = 10
```

**IndexError**: too many indices for array

배열 인덱스를 이용한 값 변경

이것은 축을 이용한 것임

Traceback

```
1 A = np.arange(5)
```

```
1 A[[0,0,2]] = [10,20,30]
```

```
1 A
```

```
array([20, 1, 30, 3, 4])
```

```
1 A = np.arange(5)
```

```
1 A[[0,0,2]] += 1
```

```
1 A
```

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

동일 인덱스를 포함할 경우  
가장 마지막 값이 할당

`+=` 연산자는 예상대로  
동작하지 않을 수 있음



ix\_() 함수는 N개의 1차원 시퀀스 입력받아 추출해서 각각 N차원인 N개의 출력을 반환

결과의 모양(shape)은 1차원을 제외한 모든 차원이 1

```
1 A = np.array([2,3,4,5])
2 B = np.array([8,5,4])
```

```
1 np.ix_(A, B)
```

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

```
1 A = np.array([2,3,4,5])
2 B = np.array([8,5,4])
3 C = np.array([5,4,6,8,3])
```

```
1 np.ix_(A, B, C)
```

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

A, B, C 배열의 모든 쌍에 대해서  $A + B * C$ 를 계산할 때 ix\_() 함수를 사용

```
1 AX, BX, CX = np.ix_(A, B, C)
```


```
1 AX + BX*CX
```

```
array([[[42, 34, 50, 66, 26],  
        [27, 22, 32, 42, 17],  
        [22, 18, 26, 34, 14]],  
  
       [[43, 35, 51, 67, 27],  
        [28, 23, 33, 43, 18],  
        [23, 19, 27, 35, 15]],  
  
       [[44, 36, 52, 68, 28],  
        [29, 24, 34, 44, 19],  
        [24, 20, 28, 36, 16]],  
  
       [[45, 37, 53, 69, 29],  
        [30, 25, 35, 45, 20],  
        [25, 21, 29, 37, 17]]])
```

**리듀스 함수 :** 매개변수로 함수와 값들(iterable)을 입력받아 각 요소들에 함수를 적용하고 하나의 결과 값을 반환하는 함수

```
1 A = np.array([2,3,4,5])
2 B = np.array([8,5,4])
```

```
1 def reduce_func(*arrs, func=np.add):
2     aix = np.ix_(*arrs)
3     result = aix[0]
4     for item in aix[1:]:
5         result = func(result, item)
6     return result
```

 `ix_()` 함수를 응용해서 1차원 배열과 함수(`np.add` 또는 `np.multiply`)를 입력받아 배열의 모든 가능한 조합의 연산을 구한 후 하나의 값을 반환하는 함수를 만들 수 있음

```
1 reduce_func(A, B, func=np.add)
```

```
array([[10,  7,  6],
       [11,  8,  7],
       [12,  9,  8],
       [13, 10,  9]])
```

```
1 reduce_func(A, B, func=np.divide)
```

```
array([[0.25 , 0.4  , 0.5  ],
       [0.375, 0.6  , 0.75 ],
       [0.5  , 0.8  , 1.   ],
       [0.625, 1.   , 1.25 ]])
```

1차원 shape=(4,)

0	1	2	3
---	---	---	---

2차원 shape=(3,4)

11	12	13	14
21	22	23	24
31	32	33	34

3차원 shape=(2,3,4)

11	12	13	14
21	22	23	24
31	32	33	34

4차원 shape=(3,2,3,4)

11	12	13	14
21	22	23	24
31	32	33	34

11	12	13	14
21	22	23	24
31	32	33	34

11	12	13	14
21	22	23	24
31	32	33	34

깊이가 2이고 3행 4열짜리 3차원 배열이 3개

5차원 shape=(2,3,2,3,4)

11	12	13	14
21	22	23	24
31	32	33	34

11	12	13	14
21	22	23	24
31	32	33	34

11	12	13	14
21	22	23	24
31	32	33	34

11	12	13	14
21	22	23	24
31	32	33	34

11	12	13	14
21	22	23	24
31	32	33	34

11	12	13	14
21	22	23	24
31	32	33	34

깊이가 2이고 3행 4열짜리 3차원 배열이 2행 3열

6차원 shape=(2,2,3,2,3,4)

11	12	13	14
21	22	23	24
31	32	33	34

11	12	13	14
21	22	23	24
31	32	33	34

11	12	13	14
21	22	23	24
31	32	33	34

11	12	13	14
21	22	23	24
31	32	33	34

11	12	13	14
21	22	23	24
31	32	33	34

11	12	13	14
21	22	23	24
31	32	33	34



# THE PYTHON - 파이썬 프로그래밍

3부. 데이터 분석 라이브러리 활용

허진경 지음

BOOKK 

## 1장. N차원 배열 다루기 - Numpy



### 6절. 선형대수학

행렬 곱 : @ 또는 `dot()`

전치 행렬 : `A.T`

역행렬(Inverse of a matrix) : `np.linalg.inv(A)`

단위행렬(Identity matrix) : `np.eye(n)`

대각행렬(Diagonal matrix) : `np.diag(A)`

대각합(Trace) : `np.trace(A)`

행렬식(Matrix Determinant) : `np.linalg.det(A)`

연립방정식 해 풀기(Solve a linear matrix equation) : `np.linalg.solve(A, B)`

고유값(Eigenvalue), 고유벡터 (Eigenvector) : `w, v = np.linalg.eig(A)`

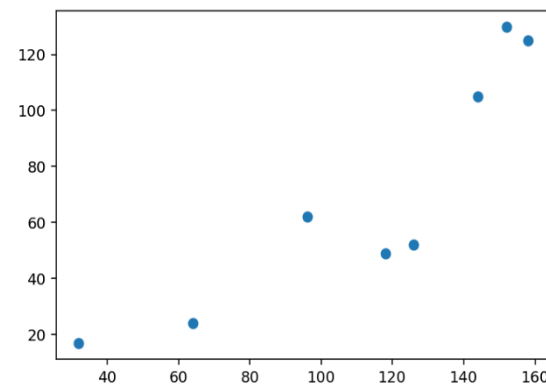
특잇값 분해(Singular Value Decomposition) : `u, s, vT = np.linalg.svd(A)`

모델 추정 문제를 행렬식 형태로 표현한 후에 선형대수학을 적용

✓ 예) 직선  $f(x) = ax + b$  추정 문제는 결국 다음 식들을 만족하는  $a, b$ 를 찾는 것.

$$\begin{aligned} ax_1 + b &= y_1 \\ ax_2 + b &= y_2 \\ &\vdots \\ ax_n + b &= y_n \end{aligned} \quad (1)$$

그래프의 점들을 가장 잘 표현할 수 있는 직선 식을 구해야 할 경우



$X = [32, 64, 96, 118, 126, 144, 152, 158]$

$Y = [17, 24, 62, 49, 52, 105, 130, 125]$

✓ 행렬식 표현

$$\begin{pmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} \quad (2)$$

$$AX = B \quad (3)$$

✓ A가 정방행렬이 아니면 역행렬은 존재하지 않지만 의사역행렬(pseudo inverse)을 이용

$$\begin{aligned} X &= \text{pinv}(A)B \\ &= (A^T A)^{-1} A^T B \end{aligned} \quad (4)$$



# THE PYTHON - 파이썬 프로그래밍

3부. 데이터 분석 라이브러리 활용

허진경 지음

BOOKK 

## 1장. N차원 배열 다루기 - Numpy



### 7절. 유용한 정보 및 팁



배열의 모양을 변경할 때 자동으로 추론 될 크기 중 하나를 생략 가능

```
1 a = np.arange(30)
```

```
1 a.shape = 2, -1, 3
2 a.shape
```

(2, 5, 3)

```
1 a
```

```
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 a.shape = 2, 3, -1
2 a.shape
```

(2, 3, 5)

```
1 a
```

```
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]]])
```



shape의 값 중 하나가  
-1이면 크기는 자동  
지정됨

histogram() 함수는 한 쌍의 벡터, 즉 배열의 막대그래프와 빈 벡터를 반환

```
1 import numpy as np
2 import matplotlib.pyplot as plt
```

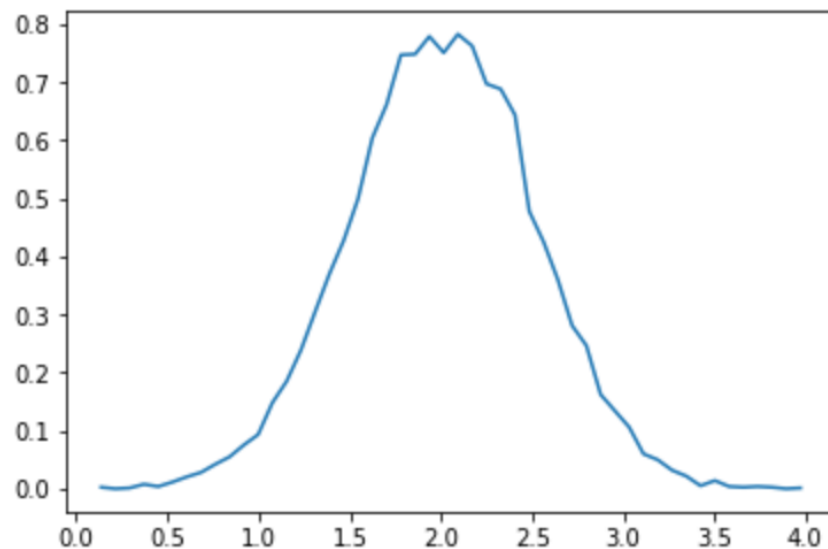
```
1 mu, sigma = 2, 0.5
2 v = np.random.normal(mu, sigma, 10000)
```

```
1 (n, bins) = np.histogram(v, bins=50, density=True)
```

```
1 n, bins
```

```
(array([0.00255555, 0.00127778, 0.00766666, 0.00383333,
        0.01149999, 0.02044442, 0.02811108, 0.04216662, 0.05494439,
        0.07538881, 0.09327768, 0.14822206, 0.18527758, 0.23894419,
        0.30538856, 0.37055516, 0.4280551 , 0.49961057, 0.60438824,
        0.66188818, 0.7474992 , 0.74877697, 0.77944361, 0.75133253,
        0.78327694, 0.76283251, 0.69766592, 0.68872148, 0.64399931,
        0.47788838, 0.42549954, 0.36033295, 0.28111081, 0.24661085,
        0.1622776 , 0.13416652, 0.10605544, 0.06005549, 0.04983328,
        0.03194441, 0.0217222 , 0.00511111, 0.01405554, 0.00383333,
        0.00255555, 0.00383333, 0.00255555, 0.00127778]),
 array([0.09751299, 0.17577394, 0.25403489, 0.33229585, 0.4105568 ,
```

```
1 plt.plot(.5*(bins[1:]+bins[:-1]), n)
2 plt.show()
```



- matplotlib의 hist() 함수는 히스토그램 그래프를 그림
- numpy.histogram() 함수는 데이터 만 생성함