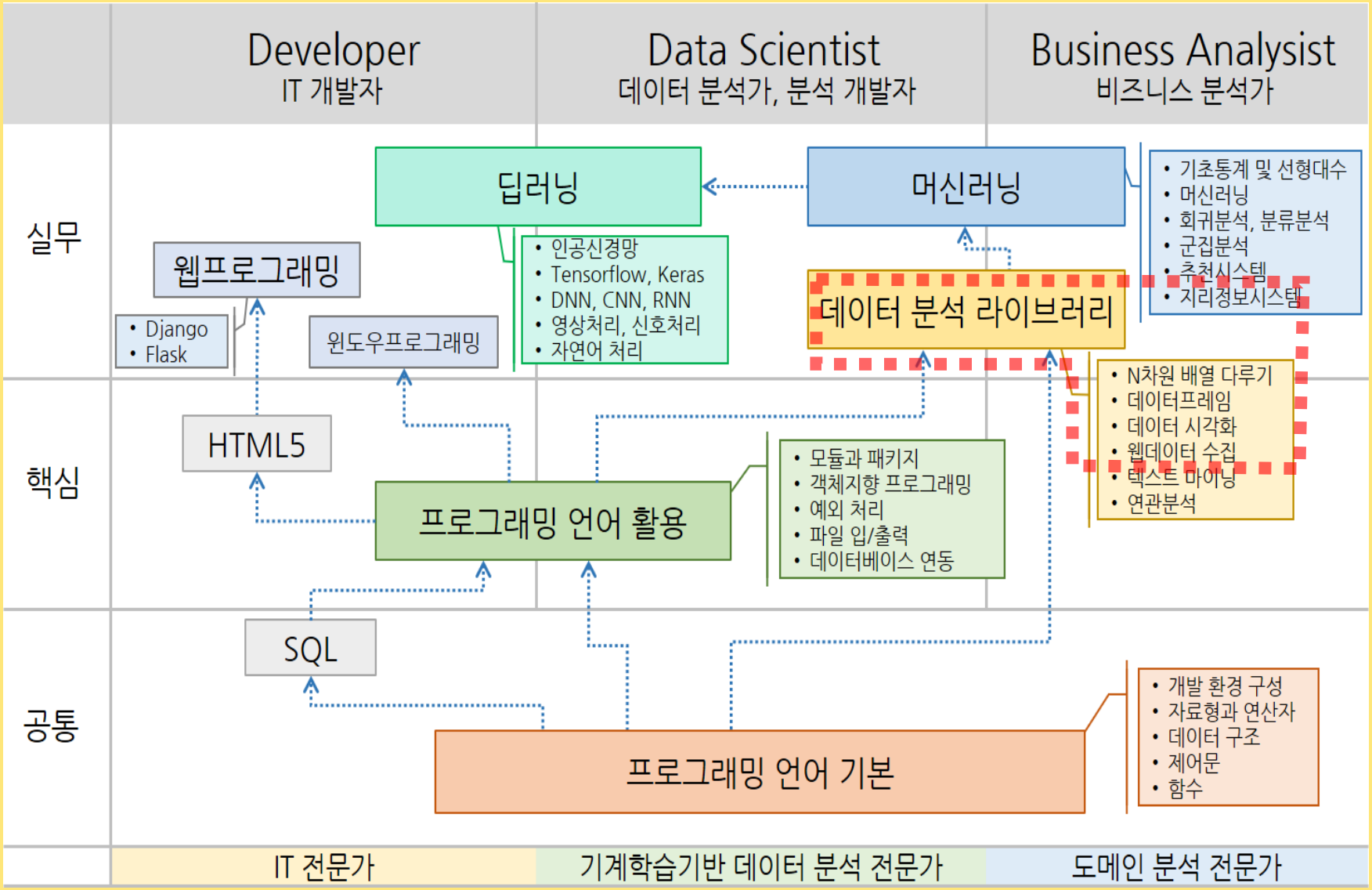
파이썬 학습 로드맵


Developer
IT 개발자
Data Scientist
데이터 분석가, 분석 개발자
Business Analysist
비즈니스 분석가
실무
핵심
공통
딥러닝
• 인공신경망
• Tensorflow, Keras
• DNN, CNN, RNN
• 영상처리, 신호처리
• 자연어 처리
머신러닝
• 기초통계 및 선형대수
• 머신러닝
• 회귀분석, 분류분석
• 군집분석
• 추천시스템
• 지리정보시스템
웹프로그래밍
• Django
• Flask
윈도우프로그래밍
데이터 분석 라이브러리
• N차원 배열 다루기
• 데이터프레임
• 데이터 시각화
• 웹데이터 수집
• 텍스트 마이닝
• 연관분석
HTML5
프로그래밍 언어 활용
• 모듈과 패키지
• 객체지향 프로그래밍
• 예외 처리
• 파일 입/출력
• 데이터베이스 연동
SQL
프로그래밍 언어 기본
• 개발 환경 구성
• 자료형과 연산자
• 데이터 구조
• 제어문
• 함수
IT 전문가
기계학습기반 데이터 분석 전문가
도메인 분석 전문가

11장. N차원 배열 다루기

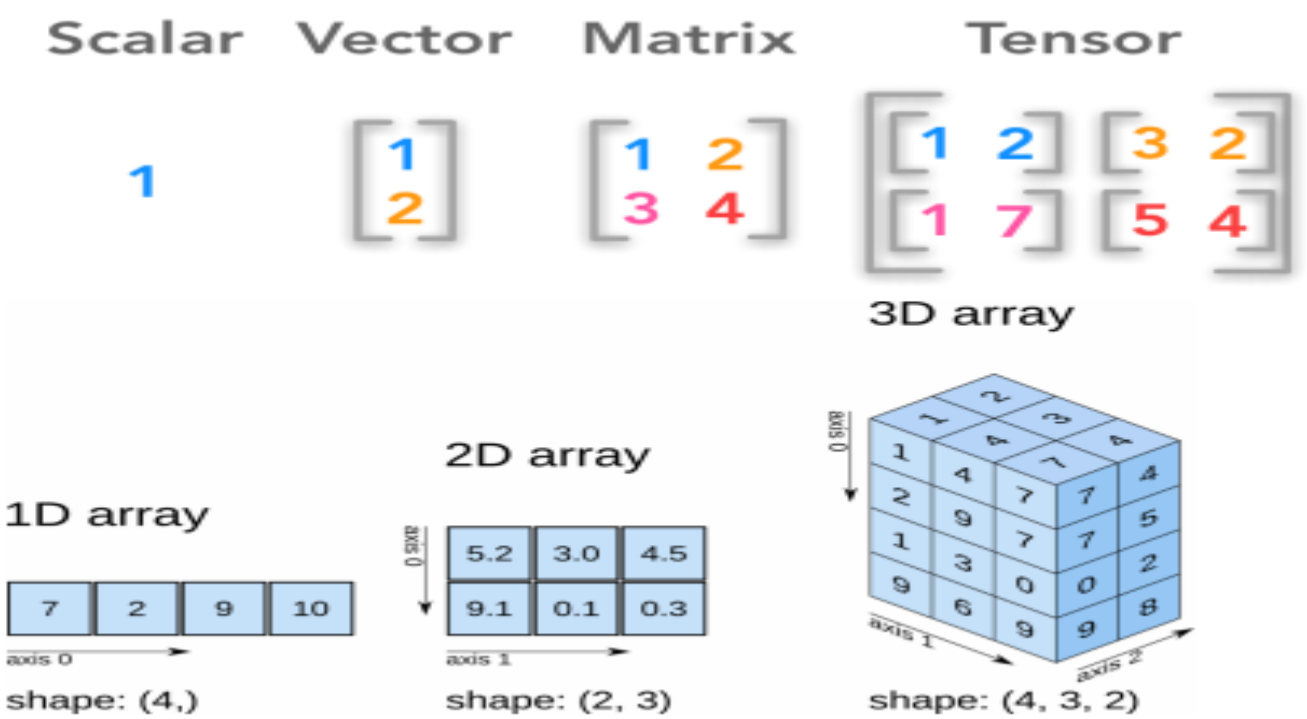
- 1. 넘파이 패키지
- 2. 넘파이 배열
- 3. 배열 합치기/분할하기
- 4. 복사와 뷰
- 5. 고급 인덱싱
- 6. 선형 대수학
- 7. 유용한 정보 및 팁

12장. 데이터프레임과 시리즈

13장. 데이터 시각화

14장. 웹 데이터 수집

데이터 구조



https://art28.github.io/blog/linear-algebra-1

https://velog.io/@mingki/Numpy-넘파이

1.1. 넘파이 소개

1절. 넘파이 패키지

- 넘파이(NumPy - http://www.numpy.org)
- 파이썬을 사용한 과학 컴퓨팅의 기본 패키지
- 넘파이 기능
  - N 차원 배열 객체C/C++, 포트란(Fortran) 코드 통합 도구
  - 선형 대수학(Linear algebra)
  - 푸리에 변환(Fourier transform)
  - 난수(Random number) 기능
- 넘파이의 주요 객체는 동일(homogeneous)의 다차원 배열
  - 모든 데이터가 같은 타입인 테이블
- 넘파이의 차원들은 **축(axis)**으로 불림

넘파이 주요 함수

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

1.2. ndarray 속성

- ndarray.ndim : 배열의 축수(차원)
- ndarray.shape : 각 차원의 배열 크기를 나타내는 정수 타입의 튜플. shape는 (n, m) 형태. 행렬은 n개의 행과 m개의 열. shape 튜플의 길이는 축의 수(ndim).
- ndarray.size : 배열의 요소의 총수. shape의 각 요소의 곱과 동일.
- ndarray.dtype : 배열 내의 요소의 타입. 파이썬의 기본 타입 또는 넘파이의 타입(numpy.int32, numpy.int16, numpy.float64 등)을 이용해 지정
  - *형변환의 개념이 아님. 지정한 타입의 크기만큼 잘라서 해당 타입으로 인식*
  - *형변환은 astype(t)메서드를 이용*
- ndarray.itemsize : 배열의 각 요소의 바이트 단위의 사이즈. 예를 들어, float64 유형의 요소 배열에는 itemsize 8(=64/8)이 있고, complex32 유형에는 itemsize 4(=32/8)가 있음. 이것은 ndarray.dtype.itemsize과 같음.

1.2. ndarray 속성

1절. 넘파이 패키지

- a.shape # (3, 5)
- a.ndim # 2
- a.dtype.name # 'int32'
- a.itemsize # 4
- a.size # 15
- type(a) # <type 'numpy.ndarray'>

```
1  import numpy as np
2  a = np.arange(15).reshape(3, 5)
3  a
```

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

시스템의인터프리터에 따라 int64, 8로 출력될 수 있음

1.3. dtype의 이해

1절. 넘파이 패키지

```
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]])
```

```
1  a.dtype
```

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

```
1  a.dtype = np.int64
```

```
1  a
```

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

- dtype 속성으로 타입을 지정하면 기존의 데이터들이 새로 바뀌는 타입의 크기만큼 배열에서 취해져 새로운 타입으로 만들어 짐
- 예에서 처음의 배열은 shape가 (3,4) 이고 아이템들의 타입이 int32였다면 이후 dtype을 int64로 지정하면 정수 두 개가 하나의 정수로 만들어 짐
- 이 과정에서 처음의 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 00000001 00000000 00000000 00000000 00000000 이 만들어 지는데 이것을 10진수로 변환하면 42949672960가 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 00000011 00000000 00000000 00000000 00000010이 만들어지고 이것을 10진수로 변환해서 12884901890가 됨

2절. 넘파이 배열

- 넘파이에서 배열을 만드는 방법
  ① array 함수를 이용한 다른 파이썬 구조(예: 리스트, 튜플)로부터의 변환
  ② 넘파이 배열을 생성하는 함수 이용(예: arange, ones, zeros, 등.)
  ③ 표준 형식 또는 사용자 정의 형식으로 디스크에서 배열 읽기
  ④ 특수 라이브러리 함수 (예: random) 이용

  - 이 절에서는 array, ones, zeros등 함수들을 이용해서 넘파이 배열을 만드는 방법 제시(arrange, array, zeros, ones, empty, linspace, normal, random)

- 매뉴얼 페이지
  - https://docs.scipy.org/doc/numpy/index.html

2.1. 넘파이 배열 만들기

2절. 넘파이 배열

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

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

```
1 import numpy as np
2 a = np.array([2,3,4])
3 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 print(id(c), id(d))
```

```
3009112305952 3009112305952
```

copy=False이고 dtype을 지정하지 않으면 새로운 배열의 복사본을 생성하지 않음

2.2. 기본값이 있는 배열 만들기

2절. 넘파이 배열

- 초기 특정 값으로 채워진 배열을 만드는 몇 가지 기능을 제공
  - zeros() 함수는 0으로 채워진 배열
  - ones() 함수는 1로 구성된 배열
  - empty() 함수는 초기 내용이 임의이고 메모리의 상태에 따라 달라지는 배열

- 배열의 dtype은 float64
  - 원한다면 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.]])
```

2.3. 연속된 값을 갖는 배열 만들기

2절. 넘파이 배열

- numpy.arange([start,]stop, [step,]dtype=None)
  - start부터 stop까지(포함 안 함) step씩 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])
```

- numpy.linspace(start, stop, num=50)
  - start부터 stop까지(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.  ])
```

많은 수의 점들을 생성할 때 유용

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

2.4. 배열의 차원 변경하기

2절. 넘파이 배열

- 배열은 각 축을 따라 요소 수만큼 주어진 차원을 가짐

```
1  a = np.floor(10*np.random.random((3,4)))
2  a
```

```
array([[2., 2., 6., 5.],
       [5., 5., 5., 9.],
       [7., 6., 5., 2.]])
```

```
1  a.shape
```

```
(3, 4)
```

- a.ravel() # 차원이 풀린 배열을 반환
- a.reshape(6,2) # shape가 수정된 배열을 반환
- a.T # 전치행렬(transposed) 반환

2.4. 배열의 차원 변경하기

2절. 넘파이 배열

- reshape() 함수는 차원이 수정 된 배열을 반환하지만 resize() 함수는 배열 자체를 수정

```
1  a = np.floor(10*np.random.random((3,4)))
2  a
```

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

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

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

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

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

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

resize에서는 음수 사용 못함

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-291-1bde992df2d8> in <module>
----> 1 a.resize(4,-1)

ValueError: negative dimensions not allowed
```

2.5. 배열 인쇄

2절. 넘파이 배열

- 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  print(a)
```

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

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

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

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

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

- (2, 3, 4)는 (2, 3)모양 배열이 4개가 있다는 것이 아님
- 이것은 (3, 4)모양 배열이 2개 있음을 의미

2.5. 배열 인쇄

2절. 넘파이 배열

- numpy.set_printoptions(*threshold=None*)
  - 넘파이가 전체 배열을 인쇄하도록 하려면 set_printoptions() 함수를 사용하여 인쇄 옵션을 변경

```
1  np.set_printoptions(threshold=1000)
2  print(np.arange(1000).reshape(10,100))
```

```
[[  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  30  31  32  33  34  35
   36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
   54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
   72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
   90  91  92  93  94  95  96  97  98  99]
 [100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117
  118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135
  136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153
  154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171
  172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189
```

1) 요소별로 연산

- 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])
```

2) 행렬의 곱

- 행렬의 곱 계산 : dot() 함수 또는 @ 연산자(3.5 이상)

$$A=\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix},\ B=\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}\text{일때,}$$

$$AB=\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}=\begin{pmatrix} a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22} \\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22} \end{pmatrix}$$

```
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 계산됨

```
1 A @ B
```

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

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

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

3) 복합대입연산자의 사용

- += 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 += a
2  b
```

```
array([[3.80389461, 3.85489825, 3.84410726],
       [3.4576111 , 3.10522101, 3.54854784]])
```

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

```
1  a += b
```

```
---------------------------------------------------------------------
-----------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-9-294cacd62d6f> in <module>()
----> 1 a += b

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

4) 배열 요소의 집계

- 배열의 모든 요소의 합계 계산과 같은 많은 단항 연산은 ndarray 클래스의 메서드로 구현되어 있음

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

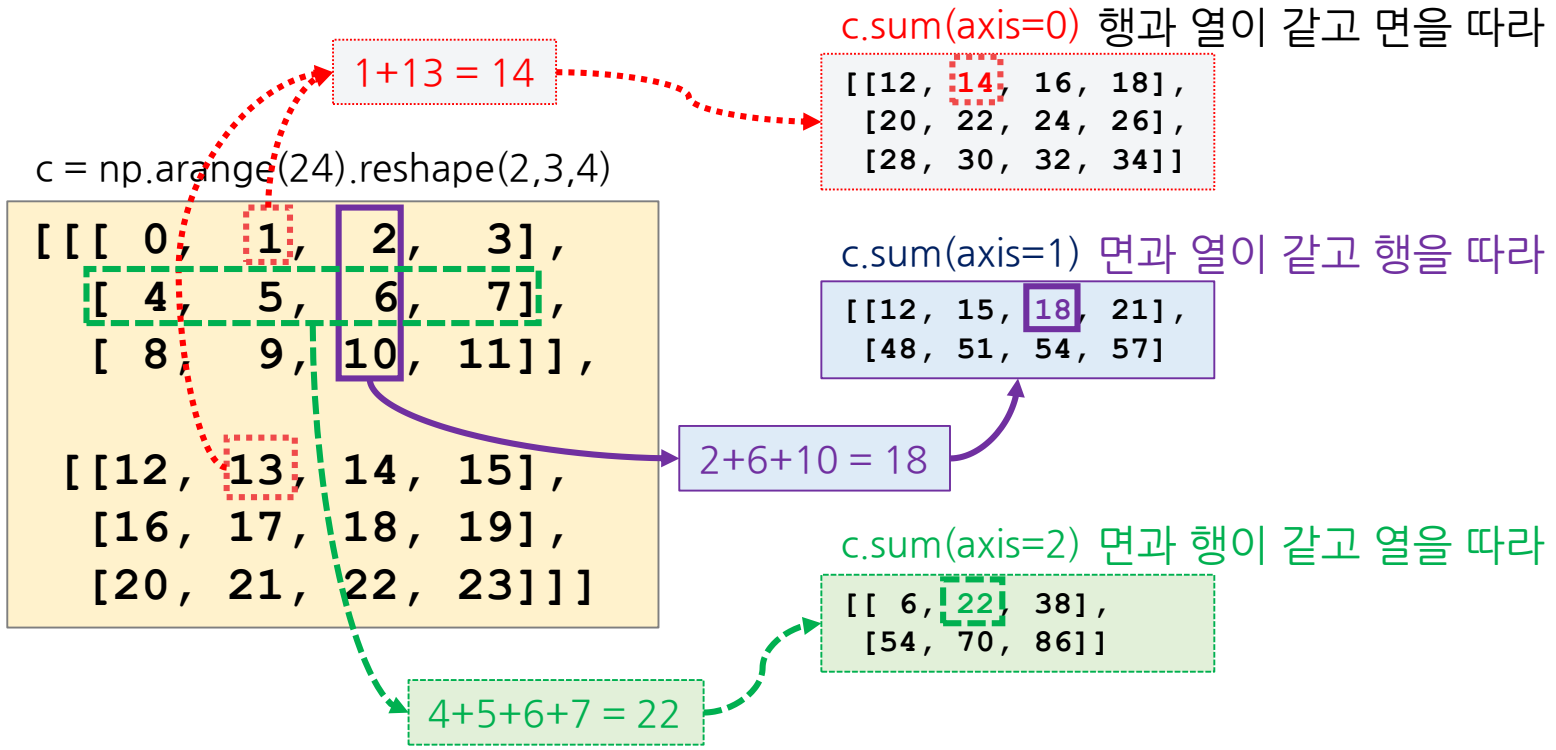
```
array([[0.38706696, 0.23036274, 0.93402891],
       [0.73949307, 0.2374663 , 0.45035011]])
```

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

```
(2.978768097316757, 0.2303627374044147, 0.9340289127216925)
```

5) 축을 지정한 집계

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



1) 범용 함수

- 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.])
```

2) 출력 인수의 사용

- 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
  - G = A * B + C
  - T1 = A * B; G = T1 + C; del T1
  - G = A + B; add(G, C, G)
  - G = A * B; G += C

3) 출력 인수의 사용과 메모리 사용량 및 실행시간 비교

- memory_profiler 패키지를 이용한 메모리 사용량 측정
  - %memit *expression*
- 셀 실행시간 측정
  - 셀의 맨 위에 %%time 입력

❖ 실행시간과 메모리 사용량의 정확한 비교를 하려면 각각 다른 커널에서 실행시켜야 함

```
1  %load_ext memory_profiler
2  import numpy as np
3  A = np.random.randn(10000000)
4  B = np.random.randn(10000000)
5  C = np.random.randn(10000000)
```

```
1  %%time
2  %memit G = A * B + C
3  print(G)
```

```
peak memory: 499.22 MiB, increment: 213.30 MiB
[ 0.52116869  1.26445922  2.50095377 ...  2.2466111  -0.59820757
  2.25796686]
Wall time: 1.36 s
```

```
1  %%time
2  %memit G = A * B; np.add(G, C, G)
3  print(G)
```

```
peak memory: 438.93 MiB, increment: 73.38 MiB
[ 0.52116869  1.26445922  2.50095377 ...  2.2466111  -0.59820757
  2.25796686]
Wall time: 1.41 s
```

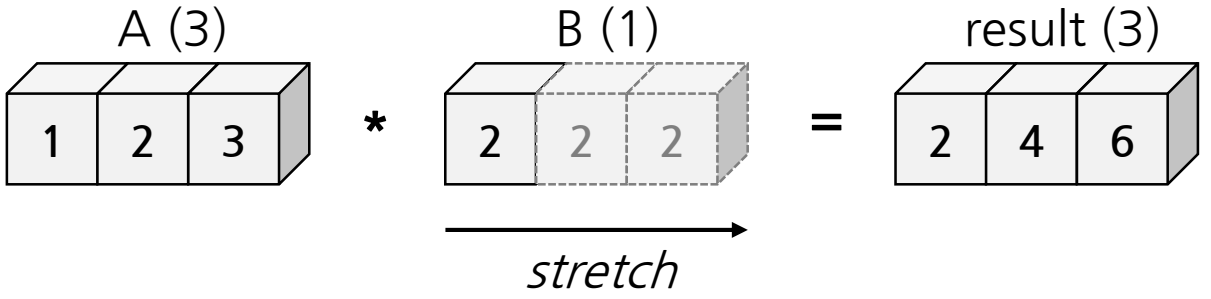
4) 넘파이의 범용 함수들

- 범용 함수(ufunc)는 배열 데이터의 연산(array broadcasting), 형 변환(type casting) 및 몇 가지 다른 표준 기능을 지원하는 넘파이 배열(ndarrays)에서 작동하는 함수 입니다.
- 범용함수들은 넘파이 배열의 요소별로 작동합니다.
- ufunc는 고정 된 수의 특정 입력을 가져 와서 고정 된 수의 특정 출력을 생성합니다.

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

1) 배열과 스칼라 연산

- 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 = 10
```

```
1  a + b
```

```
array([11, 12, 13])
```

```
1  np.add(a,b)
```

```
array([11, 12, 13])
```

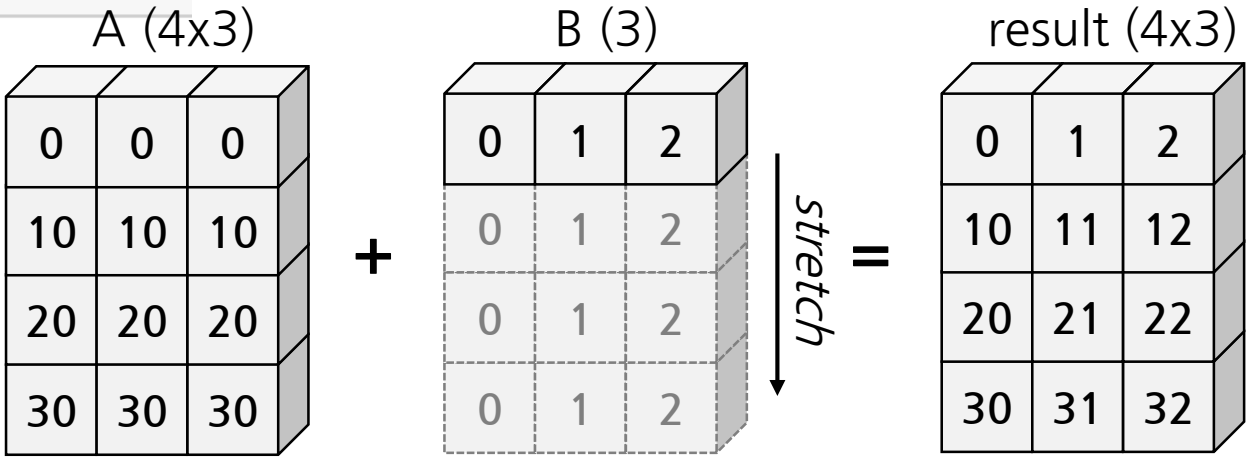
브로드 캐스팅 예

2) 브로드 캐스팅 규칙

- '브로드 캐스팅하려면 연산의 두 배열에 대한 후미 축의 크기가 동일한 크기이거나 둘 중 하나가 1이어야 한다.'

```
1  from numpy import array
2  a = array([[ 0.0,  0.0,  0.0],
3             [10.0, 10.0, 10.0],
4             [20.0, 20.0, 20.0],
5             [30.0, 30.0, 30.0]])
6  b = array([1.0, 2.0, 3.0])
7  a + b
```

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

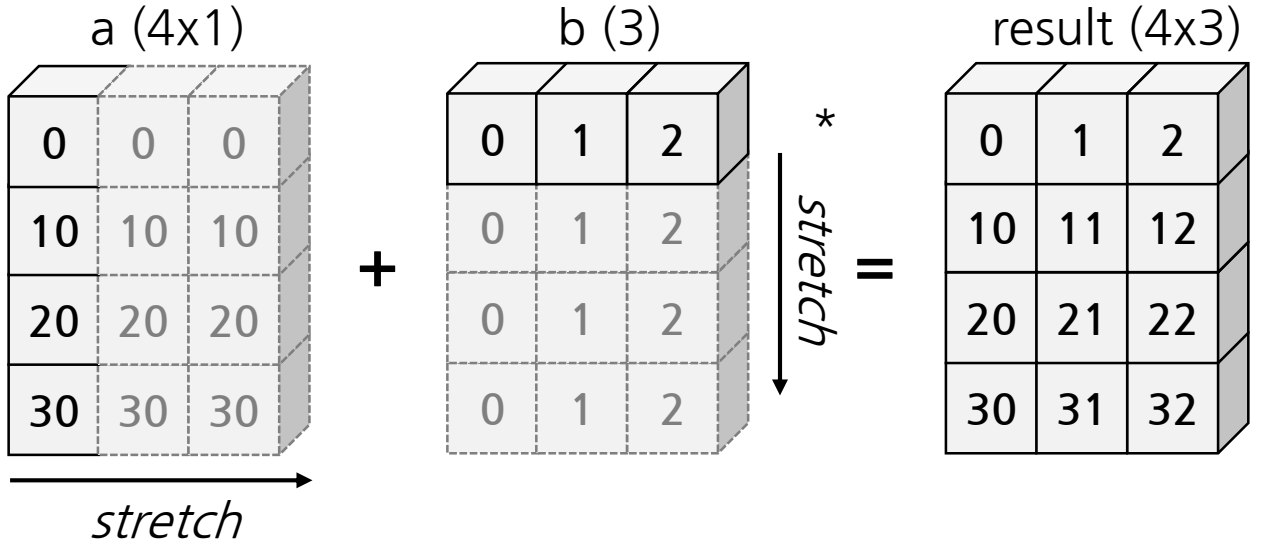


2) 브로드 캐스팅 규칙

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

```
1  from numpy import array, newaxis
2  a = array([0.0, 10.0, 20.0, 30.0])
3  b = array([1.0, 2.0, 3.0])
4  a[:,newaxis] + b
```

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



3.1. 인덱싱과 슬라이싱

3절. 배열 합치기/분리하기

- 인덱스는 맨 처음의 인덱스는 0이며 이후로 1씩 증가하도록 양수로 지정하는 것이 일반적
- 맨 뒤의 항목부터 음수를 이용해 지정할 수 있음

```
1  import numpy as np
2  a = np.arange(10)**3
3  a
```

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

데이터	0	1	8	27	64	128	216	343	512	729
양수 인덱스	0	1	2	3	4	5	6	7	8	9
음수 인덱스	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

- 도움말 문서들
  - •인덱싱 : https://docs.scipy.org/doc/numpy/user/basics.indexing.html
  - •배열 인덱싱 : https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html
  - •newaxis : https://docs.scipy.org/doc/numpy/reference/constants.html#numpy.newaxis

1) 인덱싱

- np_array_obj[*index*]

```
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-5-7c7cb9812849> in <module>
----> 1 a[10]

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

2) 슬라이싱

- np_array_obj[*start* : *stop*]
  - *start*부터 *stop*까지(*stop* 위치 포함 안함) 요소
- np_array_obj[*start* : *stop* : *step*]
  - *start*부터 *stop*까지(*stop* 위치 포함 안함) *step* 마다 요소

```
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)
```

start와 stop을 생략하면 자동 지정됨

```
1 a[ : :-1]
```

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

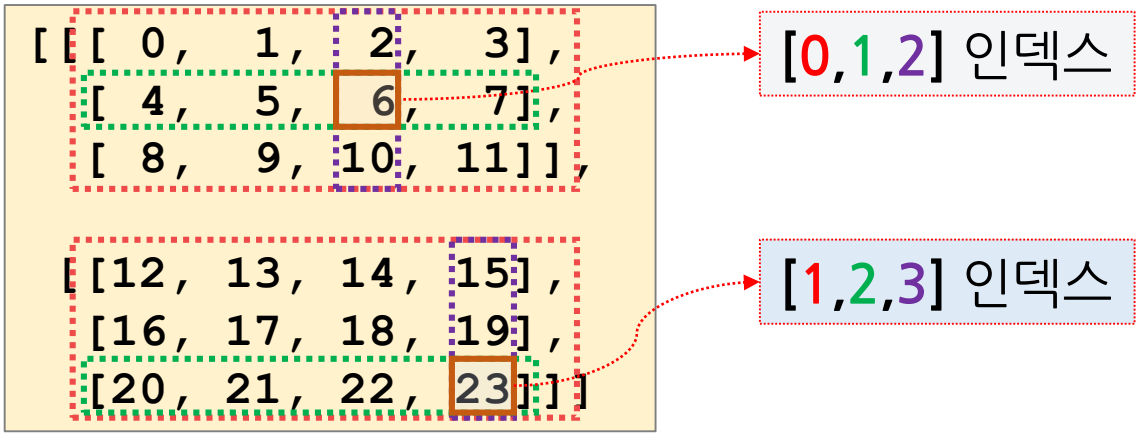
```
1 a[:6:2] = -1000   # a[0:6:2]와 동일, 처음부터 6까지 매 2번째마다 -1000
2 a
```

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

할당문을 가지면 값을 변경할 수 있음

3) 다차원 배열 인덱싱

- 2차원 배열 인덱싱
  - np_array_obj[*축1인덱스, 축2인덱스*]
- 3차원 배열 인덱싱
  - np_array_obj[*축1인덱스, 축2인덱스, 축3인덱스*]



3차원 배열 인덱싱

```
1 def f(x,y):
2     return 10*x+y
3
4 b = np.fromfunction(f, (5,4), dtype=int)
5 b
```

```
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
```

```
1 b[1,3]
```

```
13
```

```
1 b[-2,-3]
```

```
31
```

```
1 c = np.arange(24).reshape(2,3,4)          # 3차원
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]]])
```

```
1 c[0,1,2]
```

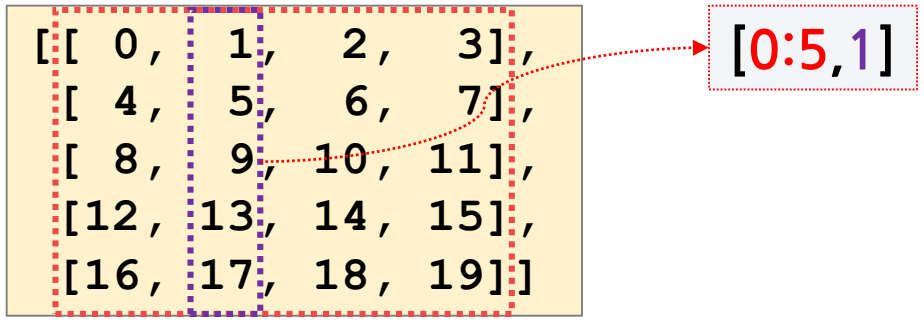
```
6
```

```
1 c[1,2,3]
```

```
23
```

4) 다차원 배열 슬라이싱

- np_array_obj[*start*:*stop*] 형식과 np_array_obj[*start*:*stop*:*step*] 형식 사용
- 차원이 여러 개 인 경우 콤마로 구분해서 각 차원 별로 start, stop 인덱스를 지정




```
1 B[0:5, 1]
```

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

```
1 B[ :, 1]
```

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

```
1 B[1:3, :]
```

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

5) 축 인덱스의 생략

- 축의 수보다 더 적은 수의 인덱스가 제공되면 누락 된 인덱스는 모든 항목을 선택함
- 넘파이에서 도트를 사용하여 b[i,...] 형식으로 작성

```
1 b[-1]
```

```
array([40, 41, 42, 43])
```

```
1 c[0]
```

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

```
1 c[0,0]
```

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

```
1 c[0,...]
```

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

```
1 c[:,:,0]
```

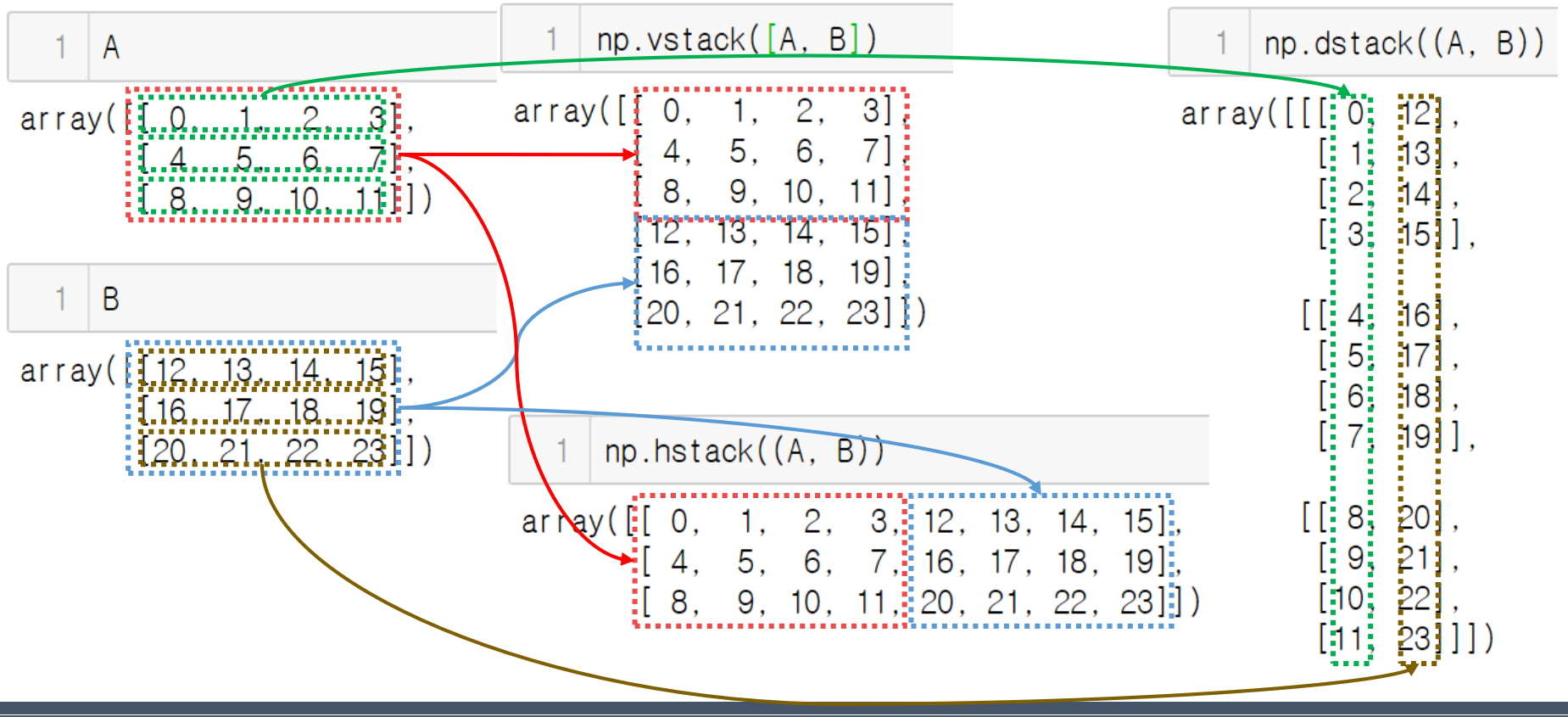
```
array([[ 0,  4,  8],
       [12, 16, 20]])
```

```
1 c[...,0]
```

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

1) hstack(), vstack(), dstack()

- hstack()은 배열을 옆에 추가하는 방식으로 쌓아 합침
- vstack()은 배열을 아래에 추가하는 방식으로 쌓아 합침
- dstack()은 3번째 축(depth)을 쌓아 합침



2) column_stack()

- 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))
4  np.column_stack((a,b,c))
```

column_stack() 함수는 1차원 배열들을 입력받아 2차원 배열을 만듦

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

```
1  a = np.array((1,2,3,4))
2  b = np.array((5,6,7,8))
3  np.hstack((a,b))
```

hstack() 함수는 1차원 배열을 입력받으면 그 결과는 1차원

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

3) newaxis 속성

- 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.column_stack((a[:,np.newaxis],b[:,np.newaxis]))
```

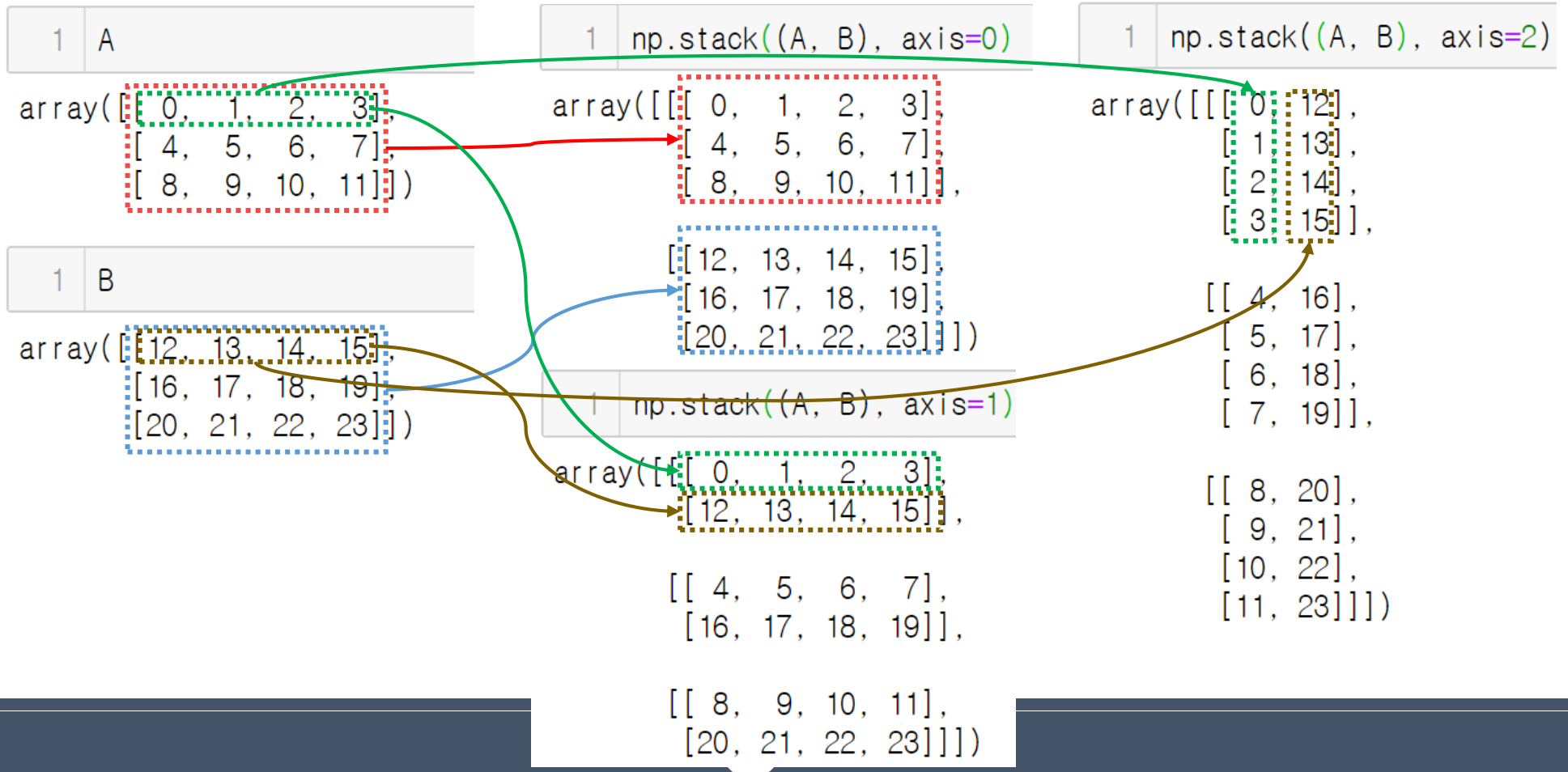
```
array([[1, 5],
       [2, 6],
       [3, 7],
       [4, 8]])
```

```
1 np.hstack((a[:,np.newaxis],b[:,np.newaxis]))
```

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

4) stack()

- stack() 함수는 축 속성 axis의 값에 따라 배열을 합침
- axis 매개 변수는 결과의 차원에서 새 축의 인덱스를 지정
- axis=0이면 첫 번째 차원, axis=-1이면 마지막 차원



3.3. r_() 함수와 c_() 함수

3절. 배열 합치기/분리하기

- r_[] 함수와 c_[] 는 한 개의 축을 따라 번호를 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.r_[[a],[b],[c]]
```

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

차원을 추가하고 싶다면 기존 데이터를 [] 기호로 묶어준다.

```
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.

1) hsplit(), vsplit(), dsplit()

- hsplit()함수를 사용하여 반환 할 동등한 모양의 배열 수를 지정하거나 나누기를 해야 하는 열을 지정하여 가로 축을 따라 배열을 나눌 수 있음
- vsplit() 함수는 세로축을 따라 분할
- dsplit() 함수는 세 번째 축(depth)을 따라 여러 개의 배열로 나눔. dsplit()은 3차원 이상 배열에서만 동작

```
1  a
```

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

```
1  a_vsplit = np.vsplit(a, 3)
```

```
1  a_vsplit
```

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

```
1  a_vsplit[0]
```

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

```
1  a_vsplit[1]
```

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

```
1  a_vsplit[2]
```

```
array([[ 8,  9, 10, 11]])
```

```
1  a_hsplit = np.hsplit(a, 4)
```

```
1  a_hsplit[0]
```

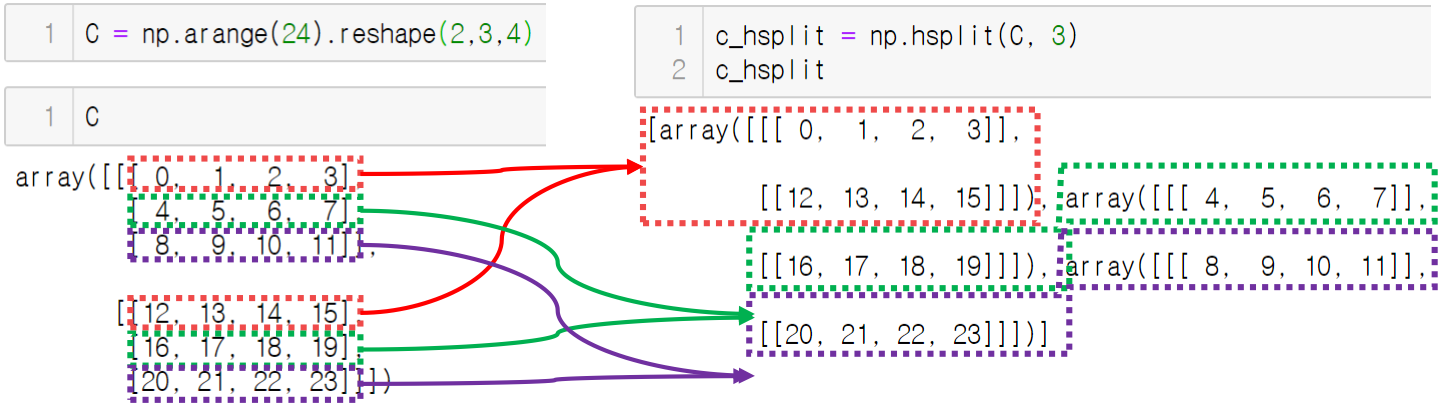
```
array([[0],
       [4],
       [8]])
```

```
1  a_hsplit[1]
```

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

2) hsplit()과 axis=1

- hsplit() 함수는 split() 함수의 axis 매개 변수가 1일 때와 같음
- 배열의 차원에 상관없이 항상 두 번째 축을 이용해 분할



3) 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))
```

```
1  a_hsplit[0]
```

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

```
1  a_hsplit[1]
```

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

```
1  a_hsplit[2]
```

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

```
1  a_hsplit[3]
```

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

4) split()

- split()은 여러 개의 작은 배열로 분할
- vsplit(), hsplit(), dsplit()과 비슷하지만 axis 파라미터를 가질 수 있음
- axis=0이면 vsplit()과 동일하게 동작
- axis=1인 경우 hsplit()과 동일하게 동작
- axis=2는 3차원 이상에서 동작하며 dsplit()과 동일하게 동작

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

```
1  b
```

```
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  np.vsplit(b, 3)
```

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

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

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

5) array_split()

- array_split() 함수와 split() 함수의 유일한 차이점은 array_split()이 *indices_or_sections*를 축을 똑같이 나누지 않는 정수로 사용할 수 있다는 것
- n개의 섹션으로 분할 되어야 하는 길이 l의 배열의 경우 (l//n)+1 크기의 l%n 하위 배열과 나머지 크기 l//n을 반환

```
1  a
```

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

```
1  np.array_split(a, 2, axis=0)
```

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

```
1  np.split(a, 2, axis=0)
```

ValueError: array split does not result in an equal division

```
TypeError                                 Traceback (most recent call
 last)
```

4.1. 모두 복사 안함

4절. 복사와 뷰

- 단순한 할당은 배열 객체나 데이터의 사본을 만들지 않음

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

```
True
```

```
b.shape = 3,4
a.shape
```

```
(3, 4)
```

```
b[::2] = 0
b
```

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

```
a
```

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

```
def f(x):
    print(id(x))
```

```
id(a)
```

```
1768677945952
```

```
f(a)
```

```
1768677945952
```

```
L = [1,2,3]
```

```
def func(data=None):
    data.append(4)
```

```
func(data=L)
```

```
L
```

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

4.2. 얕은 복사 뷰(view)

4절. 복사와 뷰

- 뷰는 동일한 데이터를 공유 할 수 있는 다른 객체
- view() 함수는 동일한 데이터를 보는 새로운 배열 객체를 생성
- s[:,:] 형식으로 배열을 자르면 뷰가 반환. 배열을 자르고 할당할 경우 원본 배열의 값이 바뀔 수 있음

```
1  c = a.view()
2  c is a
```

```
False
```

```
1  c.base is a
```

```
True
```

```
1  c.flags.owndata
```

```
False
```

```
1  c.shape = 2,6
2  a.shape, c.shape
```

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

```
1  a[0,:] = [1,2,3,4]
2  c
```

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

다른 모양 같은 데이터

```
1  a
```

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

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

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

```
1  s = a[ : , 1:3]
2  s
```

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

슬라이싱 하면 뷰가 반환

```
1  s[ : , 1] = 10
```

```
1  a
```

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

4.3. 깊은 복사 카피(copy)

4절. 복사와 뷰

- copy() 함수는 배열 및 해당 데이터의 전체 복사본을 생성

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

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

```
1  d = a.copy()
2  d is a
```

```
False
```

```
1  d.base is a
```

```
False
```

```
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]])
```

5.1. 인덱스 배열로 인덱싱

5절. 고급 인덱싱

- 배열의 인덱싱을 단일 숫자가 아닌 넘파이 배열을 이용

```
a = np.arange(12)**2
```

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

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

```
j = np.array( [ [ 3, 4], [ 9, 7 ] ] )
a[j]
```

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

이미지 저장을 팔레트의 인덱스 배열을 이용해 저장하는 예

```
palette = np.array( [ [0,0,0],                # black
                      [255,0,0],              # red
                      [0,255,0],              # green
                      [0,0,255],              # blue
                      [255,255,255] ] )       # white
```

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

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

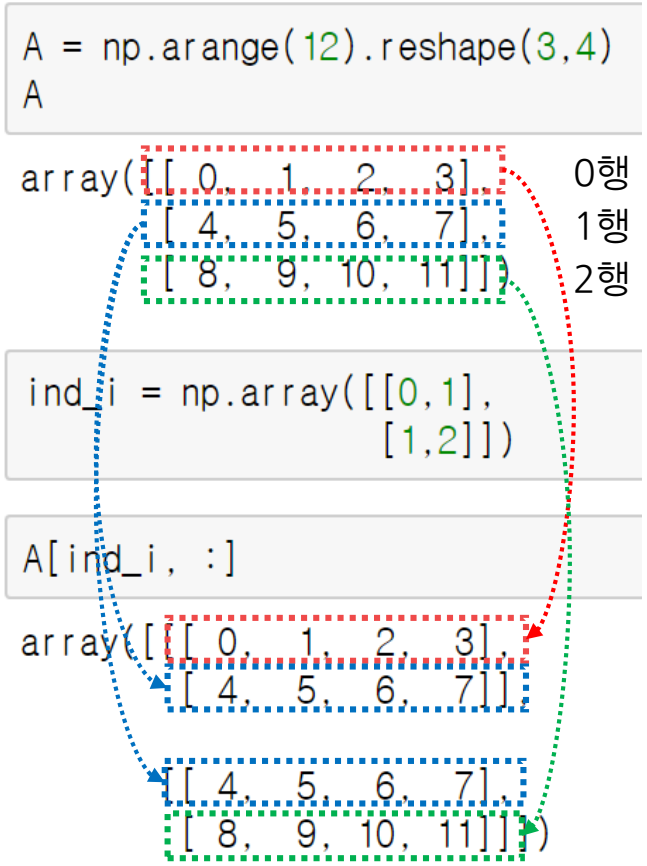
```
image_data = palette[image_index]
```

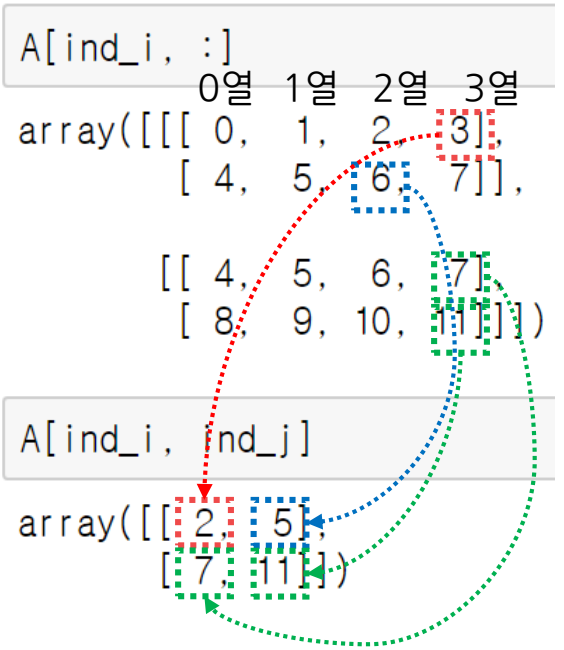
```
from matplotlib import pyplot as plt
%matplotlib inline
plt.imshow(image_data, interpolation='nearest')
plt.show()
```

5.2. 다차원 인덱싱

5절. 고급 인덱싱

- 하나 이상의 차원에 대해 인덱스를 제공
- 각 차원에 대한 인덱스 배열은 동일한 모양이어야 함





최댓값 검색

5절. 고급 인덱싱

- 다차원 인덱스 배열을 이용한 최댓값 검색의 예

```
1  data = np.sin(np.arange(20)).reshape(5,4)
```

```
1  data
```

```
array([[ 0.        ,  0.84147098,  0.90929743,  0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 ,  0.6569866 ],
       [ 0.98935825,  0.41211849, -0.54402111, -0.99999021],
       [-0.53657292,  0.42016704,  0.99060736,  0.65028784],
       [-0.28790332, -0.96139749, -0.75098725,  0.14987721]])
```

```
1  ind = data.argmax(axis=0)
2  ind
```

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

다차원 인덱싱 사용 예

```
1  data_max = data[ind, range(data.shape[1])]
```

```
1  data_max
```

```
array([0.98935825, 0.84147098, 0.99060736, 0.6569866 ])
```

```
1  data.max(axis=0)
```

```
array([0.98935825, 0.84147098, 0.99060736, 0.6569866 ])
```

```
1  np.all(data_max == data.max(axis=0))
```

```
True
```

5.3. 인덱싱을 이용한 값 변경

5절. 고급 인덱싱

- 배열을 대상으로 인덱싱을 사용해서 값 변경 가능

```
1  a = np.arange(5)
2  a
```

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

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

배열 인덱스를 이용한 값 변경

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

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

이것은 축을 이용한 것임

```
----------
IndexError                                Traceback (most recent ca
ll last)
<ipython-input-23-b9b0f372fa8b> in <module>()
----> 1 a[1,3,4] = 10

IndexError: too many indices for array
```

```
1  a = np.arange(5)  # array([0, 1, 2, 3, 4])
2  a[[0,0,2]] = [10,20,30]
3  a
```

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

동일 인덱스를 포함할 경우 가장 마지막 값이 할당

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

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

+= 연산자는 예상대로 동작하지 하지 않을 수 있음

5.4. 부울 배열을 이용한 인덱싱

5절. 고급 인덱싱

- 논리 배열 인덱스를 이용해서 인덱싱이 가능함

```
1  a = np.arange(20).reshape(4,5)
2  a
```

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

```
1  b = a % 2 == 0
2  b
```

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

```
1  a[b]
```

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
1  a[b] = a[b]**2
2  a
```

```
array([[  0,   1,   4,   3,  16],
       [  5,  36,   7,  64,   9],
       [100,  11, 144,  13, 196],
       [ 15, 256,  17, 324,  19]])
```

```
1  a[b] = 0
2  a
```

```
array([[ 0,  1,  0,  3,  0],
       [ 5,  0,  7,  0,  9],
       [ 0, 11,  0, 13,  0],
       [15,  0, 17,  0, 19]])
```

5.5. ix_() 함수

5절. 고급 인덱싱

- 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  ax,bx = np.ix_(a,b)
```

```
1  ax
```

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

```
1  bx
```

```
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  ax,bx,cx = np.ix_(a,b,c)
2  ax
```

```
array([[[2]],

       [[3]],

       [[4]],

       [[5]]])
```

```
1  bx
```

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

차원과 shape

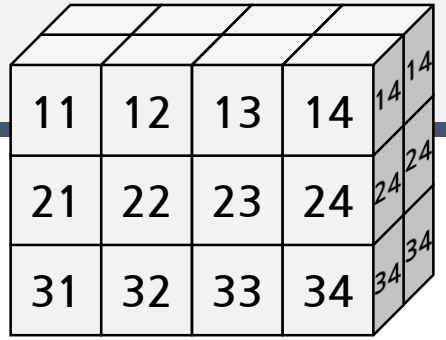
1차원 shape=(4,)

0	1	2	3

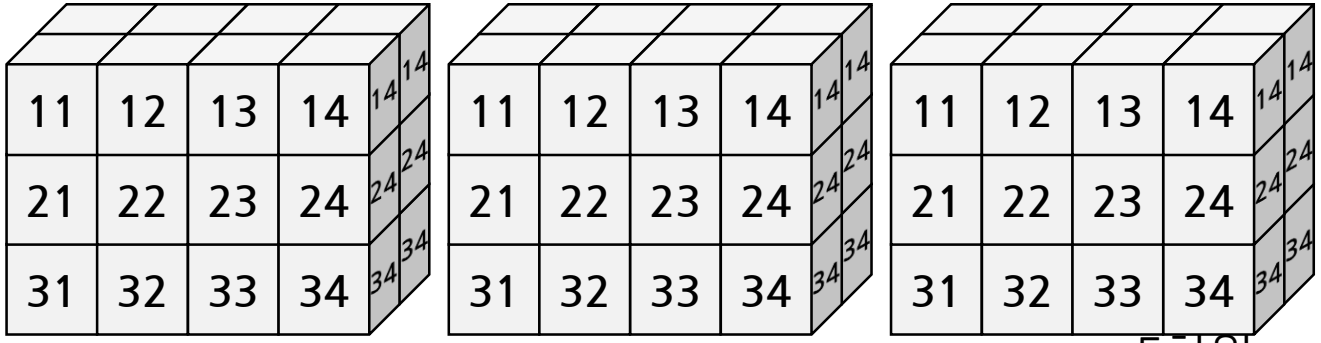
2차원 shape=(3,4)

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

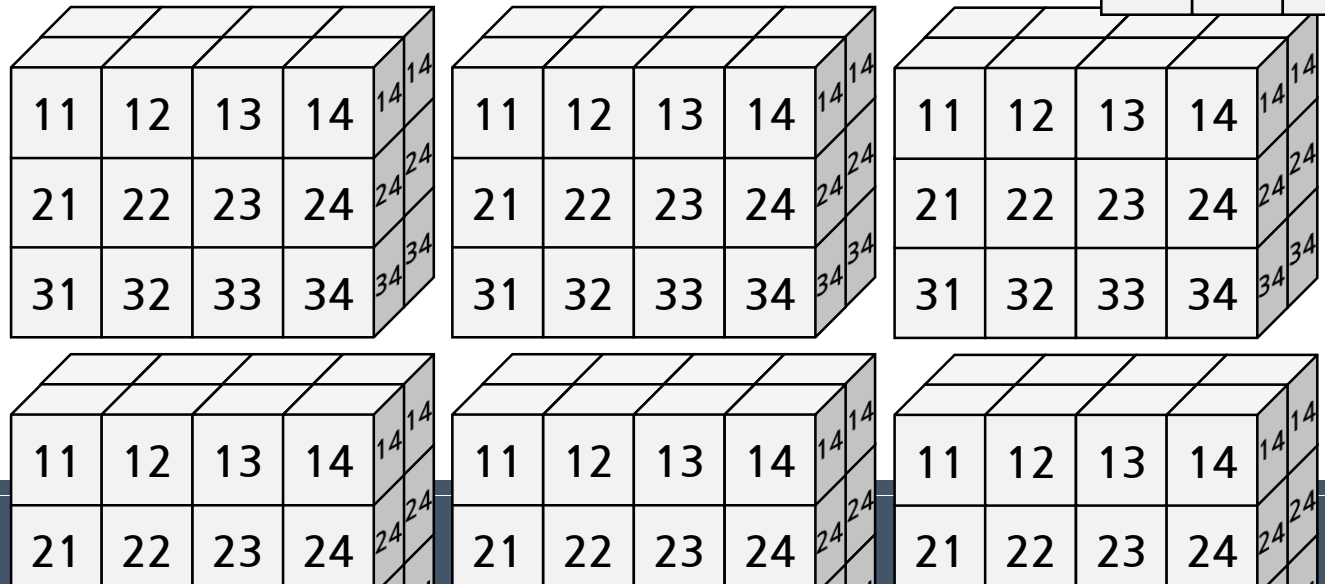
3차원 shape=(2,3,4)



4차원 shape=(3,2,3,4)



5차원 shape=(2,3,2,3,4)



6.1. 배열 조작

6절. 선형대수학

- 행렬 곱 : @ 또는 dot()
- 역행렬(Inverse of a matrix): np.linalg.inv(x)
- 단위행렬(Identity matrix) : np.eye(n)
- 대각합(Trace): np.trace(x)
- 연립방정식 해 풀기(Solve a linear matrix equation): np.linalg.solve(a, b)
- 대각행렬(Diagonal matrix): np.diag(x)
- 내적(Dot product, Inner product): np.dot(a, b)

Least Square Method

- 최소제곱법(Least Square Method; LSM) 또는 최소자승법(Least Mean Square; LMS)
- 어떤 모델의 파라미터를 구하는 한 방법으로서, 데이터와의 잔차(residual)의 합을 최소화하도록 모델의 파라미터를 구하는 방법
- 모델
  - 직선? 2차 곡선? 3차 곡선?
  - 모델을 직선으로 가정
- 파라미터
  - 모델이 결정되면 정해짐
  - f(x) = ax + b로 잡으면 a, b가 파라미터(식 1)

$$\sum_{i=1}^{n}(y_i - ax_i - b)^2 \qquad (1)$$

- 데이터
  - 측정한 데이터
- 잔차(Residual)
  - 어떤 데이터가 추정된 모델로부터 얼마나 떨어진 값인가를 나타내는 용어
  - 데이터 (xi, yi)의 잔차 ri = yi - f(xi) (식 2)
    - yi는 관측 된 값, f(xi)는 추정된 모델에 따른 값

$$\sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n}(y_i - f(x_i))^2 \qquad (2)$$

LSM 계산 - 대수적 방법

6절. 선형대수학

- 모델 추정 문제를 행렬식 형태로 표현한 후에 선형대수학을 적용
  - 예) 직선 f(x) = ax +b 추정 문제는 결국 다음 식들을 만족하는 a, b를 찾는 것.

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

  - 행렬식 표현

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

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

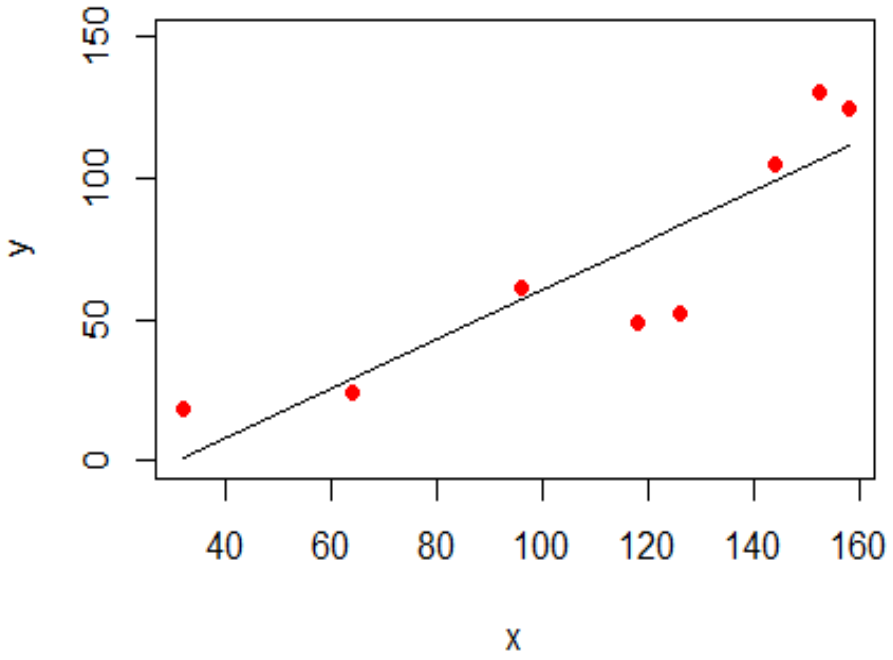
  - A가 정방행렬이 아니면 역행렬은 존재하지 않지만 pseudo inverse라는 걸 이용

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

6.2. 선형 연립방정식

6절. 선형대수학

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\
&\ldots \\
a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n
\end{aligned}
$$



$$
\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}
$$

간소화

$$AX = B$$

X를 구하려면?

$$X = A^{-1}B$$

$$\begin{bmatrix} a \\ b \end{bmatrix} = (A^T A)^{-1} A^T B$$

A가 정방행렬이 아니고 행의 수가 열의 수보다 크므로 left pseudo inverse를 이용

7.1. shape 자동 지정하기

7절. 유용한 정보와 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)
```

shape의 값이 -1이면 자동 지정됨

```
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]]])
```

7.2. 히스토그램

7절. 유용한 정보와 팁

- histogram() 함수는 한 쌍의 벡터, 즉 배열의 막대그래프와 빈 벡터를 반환
  - matplotlib의 hist() 함수는 히스토그램을 자동으로 그래프를 그림
  - numpy.histogram() 함수는 데이터 만 생성함

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

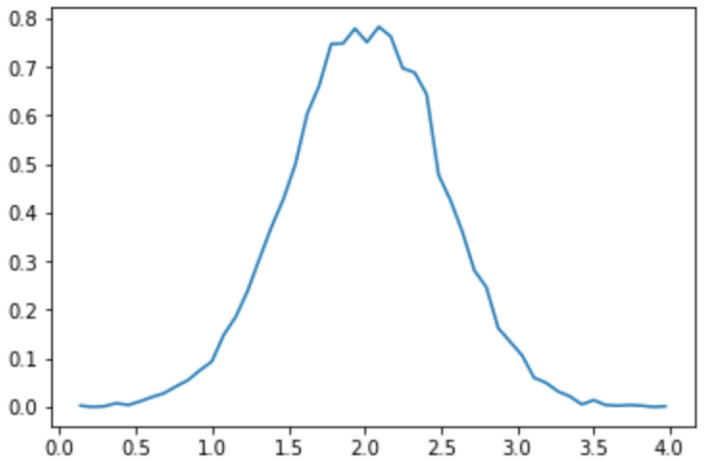
```
1  mu, sigma = 2, 0.5
2  v = np.random.normal(mu,sigma,10000)
```

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

```
1  n, bins
```

```
(array([0.00255555, 0.        , 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.        , 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()
```



연습문제(실습형)

1) 문제

다음 코드는 iris 데이터의 독립변수(sepal_length, sepal_width, petal_length, petal_length) 정보만 갖도록 한 코드입니다. 이를 이용해서 주어진 문제를 해결하세요.

```
import numpy as np
from sklearn import datasets
iris = datasets.load_iris()
iris_data = iris.data
```

1. 각 변수별 평균을 출력하세요.

```
array([5.84333333, 3.05733333, 3.758     , 1.19933333])
```

연습문제(실습형)

2. 처음 다섯 개 행을 출력하세요.

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2]])
```

3. 처음 다섯 개 행에서 마지막 열을 제외한 나머지 열을 출력하세요.

```
array([[5.1, 3.5, 1.4],
       [4.9, 3. , 1.4],
       [4.7, 3.2, 1.3],
       [4.6, 3.1, 1.5],
       [5. , 3.6, 1.4]])
```

연습문제(실습형)

4. 처음 다섯개 행에서 마지막 열만 출력하세요

```
array([0.2, 0.2, 0.2, 0.2, 0.2])
```

5. 3번 배열과 4번 배열을 원래 보양이 되도록 합치세요. 실행 결과는 2번의 결과와 같아야 합니다.

6. 처음 다섯 개 행을 이용해서 각 열 별로 평균보다 큰 값들만 출력하세요

연습문제(문제풀이형)

1. 다음 보기의 데이터가 있을 경우 print(a.sum(0))의 결과는?

```
import numpyas np
a = np.arange(12).reshape(3,4)
a
```

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

```
print(a.sum(0))
```

① 66
② [6 22 38]
③ [12 15 18 21]
④ 0

연습문제(문제풀이형)

2. a 데이터가 다음과 같을 때 다음 중 차원을 바꿀 수 있는 방법을 모두 고르세요

```
import numpyas np
a = np.floor(10*np.random.random((3,4)))
a
```

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

```
a.shape
```

```
(3, 4)
```

① a.ravel()
② a.reshape(6,2)
③ a.resize((2,6))
④ a.T(4,3)

연습문제(문제풀이형)

3. 다음 데이터에서 첫번째 행과 두번째 행만 뽑아내고 싶습니다. 빈칸에 들어갈 내용으로 옳은 것을 고르세요.

```
import numpy as np
a = np.arrange(12).reshape(3,4)

_____________________________
```

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

① a[:2]
② a[:, :2]
③ a[1,2]
④ a[1][2]

연습문제(문제풀이형)

4. 다음 코드의 실행 결과는 ?

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

① [0 2 0 0 5]

② [1 0 3 0 0]

③ [1 2 3 4 5]

④ 프로그램 오류, a[1,3,4] = 0 으로 코드를 수정해야 한다.

연습문제(문제풀이형)

5. 다음 두 배열 a, b를 출력의 예시처럼 만들고 싶을 때 빈 칸에 입력해야 할 내용으로 바른 것은?

```
import numpyas np
a= np.array([1,2,3,4,5])
b=np.array([6,7,8,9,10])
_____________
```

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

① np.r_[a, b]

② np.c_[a, b]

③ np.ix_(a, b)

④ [[a],[b]]