Numpy

파이썬 프로그래밍

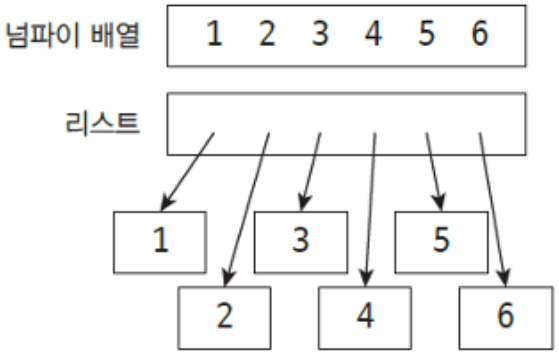
목차

Numpy 개요

- Numpy (Numerical Python)
  - 수치 해석용 파이썬 패키지
  - 다차원 배열 자료구조인 ndarray 클래스를 지원

- 기존 파이썬의 리스트 (List)
  - 가변 크기의 동적 할당
  - 많은 데이터를 리스트에서 처리하는 경우,
  - 속도가 느리고 많은 메모리를 차지함

- numpy 배열 (ndarray: N-dimensional array)
  - 고정된 크기의 배열: C 언어로 loop 처리함
  - 같은 종류의 데이터를 가짐
  - 적은 메모리를 사용
  - 데이터 처리 속도가 빠름
  - 배열 생성 후 크기 변경이 안됨
    - 변경해야 되는 경우, 새로운 배열을 생성함



Numpy 배열 생성

- ndarray 클래스
  - N차원 배열 객체(N-dimensional array)
  - 한가지 타입의 데이터만 저장 가능
  - 다차원 행렬 자료 구조를 지원
  - 인덱싱과 슬라이싱을 지원함
  - C언어 배열(array)의 특성인 연속적인 메모리에 배치됨
    - 메모리 최적화, 계산 속도 향상

Numpy 배열 생성: array()함수

- 1차원 배열 생성
  - array(list) 함수

```
import numpy as np

a = np.array([0, 1, 2, 3, 4]) # 리스트 형태의 자료를 입력 받아 1차원 배열 생성
list1 = [10, 20, 30, 40]
b = np.array(list1) # Python의 list를 매개변수로 받아서 1차원 배열 생성
c = np.array(['a', 'b', 'c'])
print(a)
print(type(a))
```

```
[0 1 2 3 4]
<class 'numpy.ndarray'>
```

- 2차원 배열 생성

```
arr2 = np.array([[1, 2, 3],
                 [4, 5, 6]])
arr2
```

Out 7

	0	1	2
0	1	2	3
1	4	5	6

Numpy 다양한 배열 생성

- zeros() 함수

```
numpy.zeros(shape, dtype=float, order='C', *, like=None)
```

  - 0으로 채워지는 배열 생성
  - shape: int 또는 튜플 형식
  - dtype: 데이터 타입(float, int 등)
  - order: 메모리 저장 방식
    - 'C': row-major (행 우선, C-style)
    - 'F': column-major (열 우선, Fortran-style)

```
arr = np.zeros(5)
print(arr)
```

```
[0. 0. 0. 0. 0.]
```

```
arr.shape
```

```
(5,)
```

```
arr.dtype
```

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

```
arr1 = np.zeros((2,3), dtype=np.int64)
print(arr1)
```

```
[[0 0 0]
 [0 0 0]]
```

```
arr1.shape
```

```
(2, 3)
```

```
arr1.dtype
```

```
dtype('int64')
```

Numpy의 다양한 배열 생성 방법

- 배열 생성
  - ones(), eye() 같은 다양한 함수를 사용해서 배열 초기화
  - ones(shape, dtype=None, order='C')
  - eye(N, M=None, k=0, dtype=<class 'float'>): 단위 행렬 생성

```
import numpy as np

a = np.ones(10) # 1로 이루어진 크기가 10인 배열 생성
print(a)

b = np.eye(4) # 단위행렬 생성
print(b)
```

```
# np.ones(10)
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

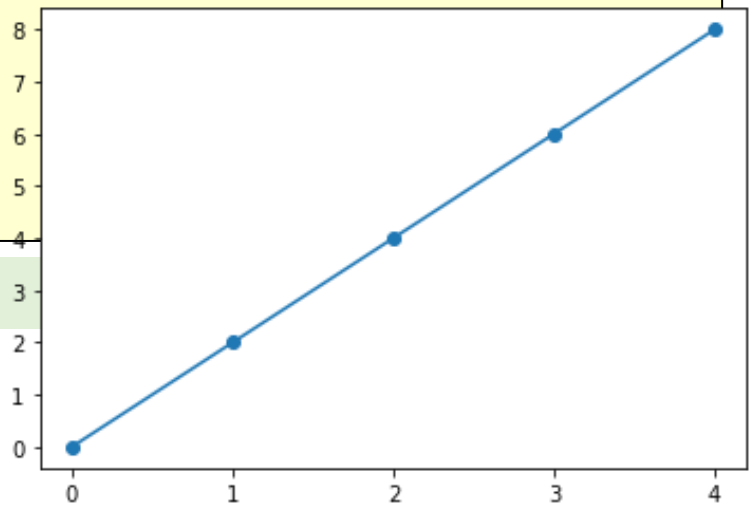
```
[[1. 0. 0. 0.]  # np.eye(4)
[0. 1. 0. 0.]
[0. 0. 1. 0.]
[0. 0. 0. 1.]]
```

Numpy의 arange()

- arange(start, stop, step)
  - start부터 stop 미만까지 step 간격으로 배열 생성 (데이터 간격 지정)
  - 데이터의 간격을 기준으로 데이터 생성

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

xtick = np.arange(5)
a = np.arange(0, 10, 2)
print(a)

plt.plot(a, 'o', linestyle='-')
plt.xticks(xtick)
plt.show()
```

```
[0 2 4 6 8]
```



ndarray 생성 방법

1차원 배열 생성

In [9]:
```
import numpy as np
data1 = [1, 2, 3, 4, 5]
arr = np.array(data1)
arr
```

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

2차원 배열 생성

In [10]:
```
data2 = [[1, 2, 3], [4, 5, 6]] # 2차원 배열 생성
arr1 = np.array(data2)
arr1
```

Out[10]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

ones((rows, cols))

ones((rows, cols)): 행렬의 값이 1인 행렬 생성

In [18]:
```
onearray = np.ones((2, 3)) #
print(onearray)
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
```

zeros((rows, cols))

zeros((rows, cols): 행렬의 값이 0인 행렬 생성

In [17]:
```
zeroarray = np.zeros((3, 3))
print(zeroarray)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

arange(start, end, step)

범위 내의 값을 순차적으로 가지는 배열 생성

In [16]:
```
array3 = np.arange(10)
array3
```

Out[16]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [20]:
```
array4 = np.arange(10, 20, 2)
array4
```

Out[20]:
```
array([10, 12, 14, 16, 18])
```

numpy의 ndarray 속성

속성	설 명
ndim	배열 축 혹은 차원의 갯수 (1차원, 2차원 배열)
shape	배열의 차원으로 (m, n) 형식의 튜플 형태이다. 이 때, m과 n은 각 차원의 원소의 크기를 알려주는 정수 값이다.
size	배열의 원소의 개수이다. 이 개수는 shape내의 원소의 크기의 곱과 같다. 즉 (m, n) shape 배열의 size는 (m x n) 이다.
dtype	배열내의 원소의 형을 기술하는 객체이다. numpy는 파이썬 표준 형을 사용할 수 있으나 numpy 자체의 자료형인 bool_, character, int_, int8, int16, int32, int64, float, float8, float16_, float32, float64, complex_, complex64, object_ 형을 사용할 수 있다.
itemsize	배열내의 원소의 크기를 바이트 단위로 기술한다. 예를 들어 int32 자료형의 크기는 32bit/8bit=4 바이트가 된다.
data	배열의 실제 원소를 포함하고 있는 버퍼.

배열 속성 예제

```
import numpy as np
arr = np.random.randint(1, 100, size=(3,4))

print(arr)
print("type: ", type(arr))
print("shape: ", arr.shape)
print("ndim: ", arr.ndim)
print("size: ", arr.size)
print("dtype: ", arr.dtype)

arr1 = arr.astype(np.float64)
print(arr1.dtype)
print(arr1)
```

astype(): 배열의 데이터 타입을 변환하고 새로운 배열을 리턴 (int64->float64)로 변환

```
[[99 34 59 70]
 [37 30 65 74]
 [65 16 19 88]]
type: <class 'numpy.ndarray'>
shape: (3, 4)
ndim: 2
size: 12
dtype: int64
float64
[[99. 34. 59. 70.]
 [37. 30. 65. 74.]
 [65. 16. 19. 88.]]
```

Numpy 데이터 타입

- numpy가 지원하는 데이터 타입(dtype 속성)
  - 숫자형
    - 정수형: int8, int16, int32, int64
    - 부호 없는 정수형: uint8, unit16, unit32, uint64
  - bool: boolean 타입
    - True, False
  - 부동 소수형
    - float16, float32, float64
  - 복소수형
    - complex64, complex128
  - np.object: 파이썬 객체 타입
  - np.string_: 고정자리 스트링 타입
  - np.unicode_: 고정자리 유니코드 타입

Numpy 데이터형 설정

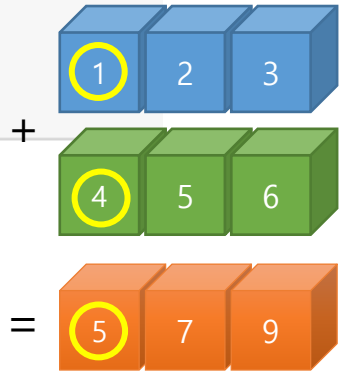
- 배열의 데이터 타입 지정 및 연산

```
In [27]:
1 a = np.array([1, 2, 3], dtype='int32')
2 b = np.array([4, 5, 6], dtype='int64')
3 a.dtype

Out[27]: dtype('int32')
```

```
In [28]:
1 b.dtype

Out[28]: dtype('int64')
```

```
In [29]:
1 c = a + b
2 print(c)
3 print(c.dtype)

[5 7 9]
int64
```

크기가 큰 데이터 타입으로 자동 변환

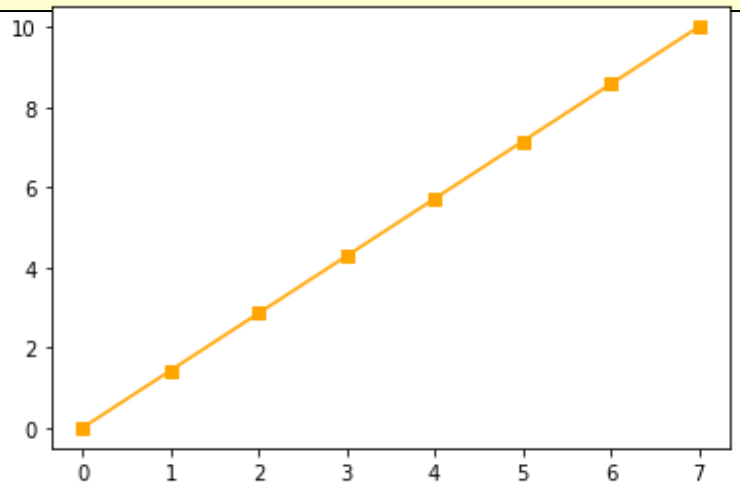


linspace(start, stop, n)

- linspace(start, stop, n)
  - start ~ end의 범위까지 n개의 균일한 구간으로 나누어 데이터를 생성하고 배열을 생성 (구간의 개수를 지정)
  - linear space

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

a = np.linspace(0, 10, 8) # 0~10사이의 값을 8구간으로 균등하게 생성
print(a)
plt.plot(a, marker='s', color='orange')
plt.show()
```

```
[ 0.          1.42857143  2.85714286
4.28571429  5.71428571  7.14285714
  8.57142857 10.        ]
```

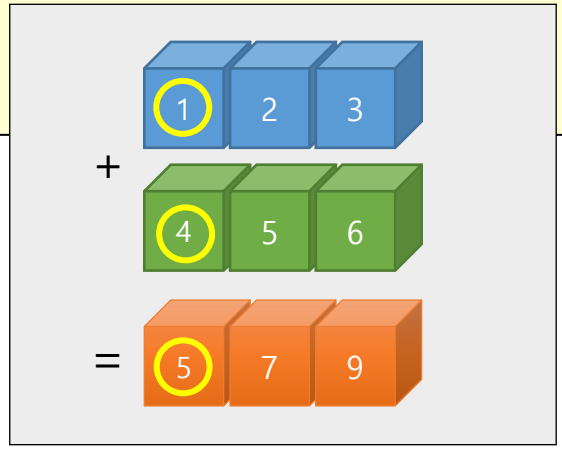


Numpy 산술 연산 #1

- 덧셈, 뺄셈 연산: shape이 같아야 함

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

c = a + b # 행렬 덧셈 연산: 같은 위치에 있는 값끼리 더함
d = a - b # 행렬 뺄셈 연산
e = a + 10 # 행렬 a의 값을 10씩 증가시킴

print('a+b: ', c)
print('a-b: ', d)
print('a+10: ', e)
```

```
a+b: [5 7 9]
a-b: [-3 -3 -3]
a+10: [11 12 13]
```



Numpy 산술 연산 #2

```
import numpy as np
a = np.arange(1, 10).reshape(3,3)
print('a')
print(a)

b = np.arange(9, 0, -1).reshape(3,3)
print('b')
print(b)

print('a+b')
print(a+b)

print('a-b')
print(a-b)

print('a*b')
print(a*b)

print('a/b')
print(a/b)
```

```
a
[[1 2 3]
 [4 5 6]
 [7 8 9]]
b
[[9 8 7]
 [6 5 4]
 [3 2 1]]
a+b
[[10 10 10]
 [10 10 10]
 [10 10 10]]
a-b
[[-8 -6 -4]
 [-2 0 2]
 [ 4 6 8]]
a*b
[[ 9 16 21]
 [24 25 24]
 [21 16 9]]
a/b
[[0.11111111 0.25 0.42857143]
 [0.66666667 1. 1.5 ]
 [2.33333333 4. 9. ]]
```

같은 위치의 숫자와 연산 수행

Numpy array(ndarray)를 list로 변환 #1

- Numpy의 ndarray를 Python의 list로 변환
  - 1차원 배열: list(ndarray) 사용

```
import numpy as np

a1 = np.array([1, 2, 3])

b1 = list(a1)
b2 = a1.tolist()

print('list(a1):', b1)
print('ndarray.tolist():', b2)
```

list(a1)과 a1.tolist()의 결과는 동일

```
list(a1): [1, 2, 3]
ndarray.tolist(): [1, 2, 3]
```

Numpy array(ndarray)를 list로 변환 #2

- Numpy의 ndarray를 Python의 list로 변환
  - 2차원 배열: ndarray.tolist()함수 사용

```
# 3x4형태의 랜덤 배열 생성
randarray = np.random.randint(1, 100, size=(3, 4))
print('randarray')
print(randarray)
```

randint()를 이용하여 3x4 배열 생성

```
randarray
[[62 67 52 50]
 [67 10 25  6]
 [27 14 14 87]]
```

```
b2 = list(randarray)
print(b2)
```

2차원 배열을 list()함수로 변환시 내부 원소는 변환 안됨

```
[array([62, 67, 52, 50]), array([67, 10, 25,  6]), array([27, 14, 14, 87])]
```

```
b3 = randarray.tolist()
print(b3)
```

2차원 배열을 tolist()함수로 변환

```
[[62, 67, 52, 50], [67, 10, 25, 6], [27, 14, 14, 87]]
```

다차원 배열의 인덱싱

- 배열의 인덱싱
  - Python 리스트 인덱싱: [행][열]
    - 행, 열을 별도의 대괄호로 분리

```
python_list = [[1, 2, 3],
               [4, 5, 6]]

for i in range(len(python_list)):
    for j in range(len(python_list[0])):
        # 파이썬 2차원 리스트 접근 [row][col]
        print(python_list[i][j], end=' ')
    print()
```

```
1 2 3
4 5 6
```

다차원 배열의 인덱싱

- Numpy 배열의 인덱싱
  - 하나의 대괄호 내부에서 콤마로 분리: [행, 열]

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

print('b[0, 0]:', b[0, 0])
print('b[1, 2]:', b[1, 2])
print('b[-1, -1]:', b[-1, -1]) # 마지막 행, 마지막 열 원소
print('b[0][1]:', b[0][1])     # Python 리스트 접근 방식
```

```
b[0, 0]: 0
b[1, 2]: 5
b[-1, -1]: 5
1
```

b[0,0] 0	b[0,1] 1	b[0,2] 2
b[1,0] 3	b[1,1] 4	b[1,2] 5

Numpy의 2차원 배열 인덱스

1차원 배열 Indexing/Slicing

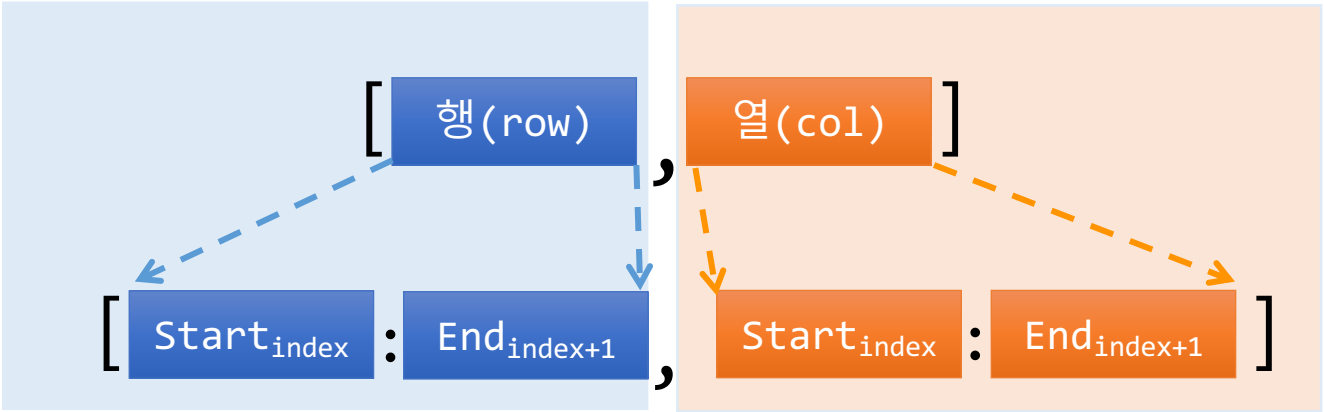
- 범위 지정:
  - data[$from_{index}$: to_{index}]: to_{index}는 범위에 포함되지 않음
  - $from_{index}$: 생략 가능 (생략한 경우, 0으로 간주함)
  - to_{index} : 생략 가능 (생략한 경우, 마지막 인덱스로 설정)
- 1차원 배열
  - [:] : 인덱스 없이 콜론(:) 만 사용하면 전체 데이터를 선택

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

print(data[:]) # 전체 데이터 선택
print(data[:3]) # index 0 ~ 2까지
print(data[0:1]) # index 0 (1은 포함 안됨)
print(data[-2:]) # index -2에서 끝까지
```

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

2차원 배열 Indexing/Slicing #1

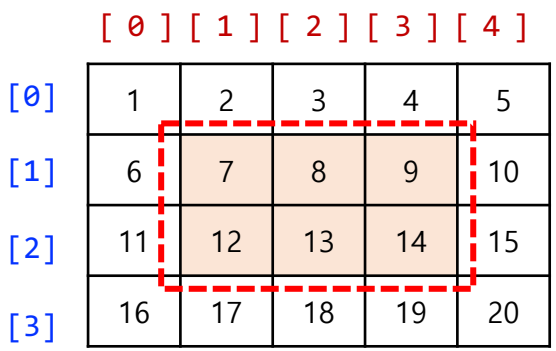
- Indexing/Slicing
  - 콤마를 기준으로 2개의 인자를 받음: [행(row), 열(col)]
  - 콜론(:) 사용: 콜론 좌측은 시작 index, 우측은 마지막 index+1의 값
  - [row_start_{index}: $row\_end_{index+1}$, col_start_{index}: $col\_end_{index+1}$]
    - 예) [1:3, 1:4]



2차원 배열 Indexing/Slicing #2

- [row start$_{index}$: row end$_{index+1}$, col start$_{index}$: col end$_{index+1}$]
  - 예) [1:3, 1:4]

```
a = np.array([[1, 2, 3, 4, 5],
              [6, 7, 8, 9, 10],
              [11, 12, 13, 14, 15],
              [16, 17, 18, 19, 20]])

print(a[1, 2])        # 1행 2열의 원소 출력 (8)
print(a[1:2])         # 1행의 원소 출력 [6, 7, 8, 9, 10]
print(a[:, -1])       # 모든 행의 마지막 열 출력 [5, 10, 15, 20]
print(a[1:3, 1:4])    # 1~2행, 1~3열 출력
```



a[1,2] → 8

a[1:2] → 6 7 8 9 10

a[:, -1] → 5 10 15 20

a[1:3, 1:4] →

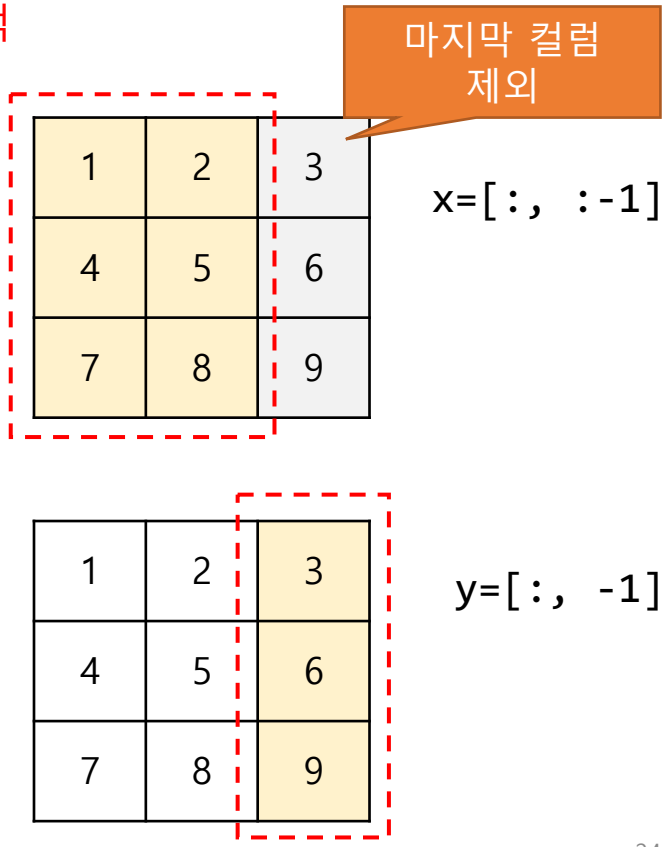
7	8	9
12	13	14

2차원 배열 Indexing/Slicing #3

- 2차원 배열 분리
  - X = [: , :-1] or X = [:, :2]
    - 2차원 배열에서 모든 행 선택, 마지막 컬럼을 제외한 모든 컬럼(컬럼 0, 1) 선택
  - Y = [: , -1] or Y = [:, 2]
    - 2차원 배열에서 모든 행 선택, 마지막 col만 선택

```
import numpy as np

data = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])

x, y = data[:, : -1], data[:, -1]
print(x)
print()
print(y)
```

```
[[1 2]
 [4 5]
 [7 8]]

[3 6 9]
```



reshape()함수 #1

- reshape(rows, cols)
  - 원래의 배열을 동일한 크기의 다른 차원(rows x cols)으로 변경함
  - 전체 사이즈 48: (12x4) 배열 -> (8x6)형태로 변경
    - reshape(8, 6)
  - reshape(-1, 4)
    - -1: 자동 맞춤
    - 열의 수 4를 고정, 행은 자동으로 12가 됨 ->(12, 4)가 됨

```
a = np.arange(1, 21)
print(a)
```

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

```
a = a.reshape(4, 5)
print(a)
```

(4 x 5) 형태의 2차원 배열로 변경

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

reshape()함수 #2

- reshape() 함수 예제

```
b = a.reshape(2, 10)
print(b)
```

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

```
c = b.reshape(4, -1)
print(c)
```

(2 x 10) 형태의 2차원 배열을
(4 x 5) 형태의 배열로 변경

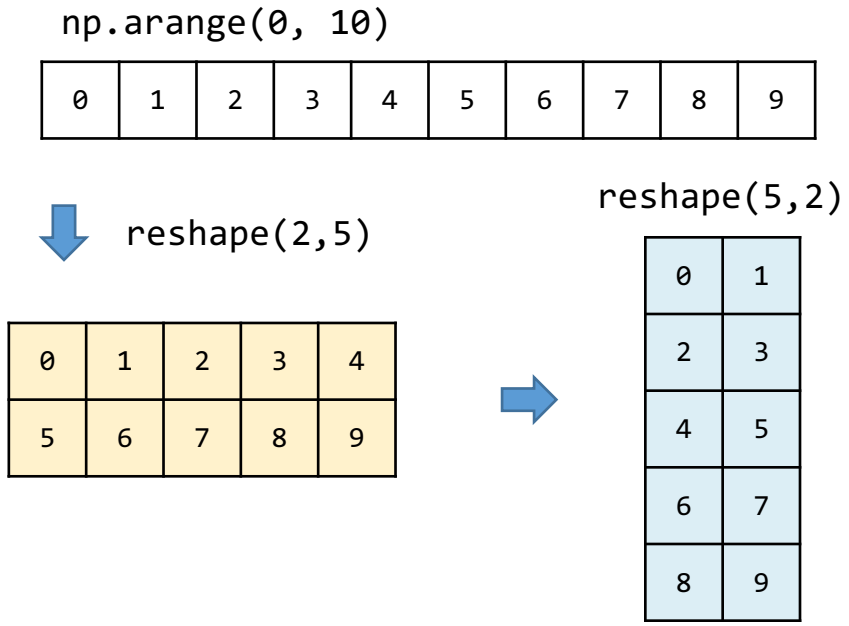
```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]
```

reshape() 예제

- 배열 생성과 동시에 reshape() 적용

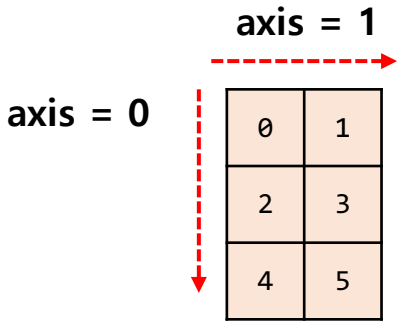
```
array1 = np.arange(0, 10).reshape(2, 5)
print(array1)
print()

array2 = array1.reshape(5, 2)
print(array2)
```

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

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



다차원 배열의 축(axis) #1

- Numpy의 모든 집계 함수는 축(axis)을 기준으로 계산
  - 집계함수: max(), min(), sum(), mean(), cumsum(), var(), std() 등
  - axis를 지정하지 않으면 axis= None
- axis
  - axis = None
    - 전체 행렬을 하나의 배열로 간주
  - axis = 0: (열 연산)
    - 각 열 단위로 연산 수행
  - axis = 1: (행 연산)
    - 행 단위로 연산 수행




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

print(x)
print('sum(axis=None): ', x.sum(axis=None)) # 전체 합계
print('sum(axis=0): ', x.sum(axis=0)) # 열 합계
print('sum(axis=1): ', x.sum(axis=1)) # 행 합계
```

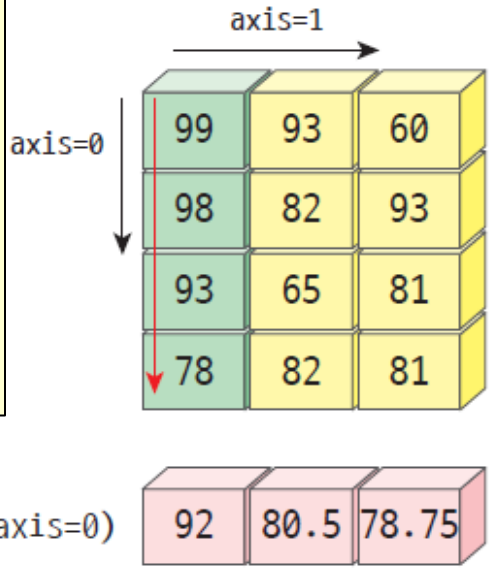
```
[[0 1]
 [2 3]
 [4 5]]
sum(axis=None):  15
sum(axis=0):  [6 9]
sum(axis=1):  [1 5 9]
```

다차원 배열의 축(axis) #2

- 행이나 열 단위로 계산

```
scores = np.array([[99, 93, 60],
                   [98, 82, 93],
                   [93, 65, 81],
                   [78, 82, 81]])

mean_col = scores.mean(axis=0)  # col단위 계산(평균)
mean_row = scores.mean(axis=1)  # row단위 계산
print(mean_col)
print(mean_row.round(2))
```

```
[92.  80.5 78.75]
[84.  91.  79.67 80.33]
```



브로드캐스팅(broadcasting)

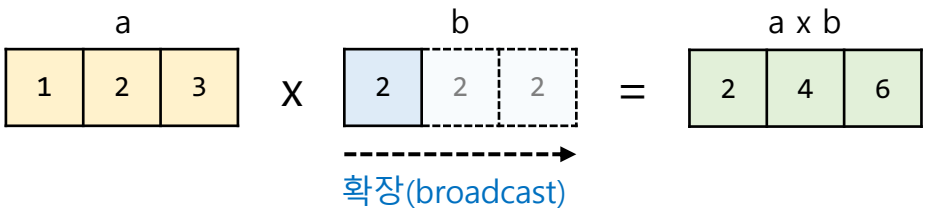
- Broadcasting
  - 특정 조건에 맞는 경우, 크기가 다른 형태의 배열끼리 연산을 수행
    - 크기가 작은 쪽의 배열을 큰 쪽의 배열 크기로 확장하여 연산
  - Broadcasting 조건
    - 각 배열의 차원의 크기가 일치하거나
    - 둘 중 하나의 길이가 1인 경우, 두 배열은 broadcasting 가능
- Broadcasting이 안되는 경우

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([1, 2])
print(a + b)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-78-7a6abddfdd86> in <module>
      3 a = np.array([1, 2, 3])
      4 b = np.array([1, 2])
----> 5 print(a + b)

ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

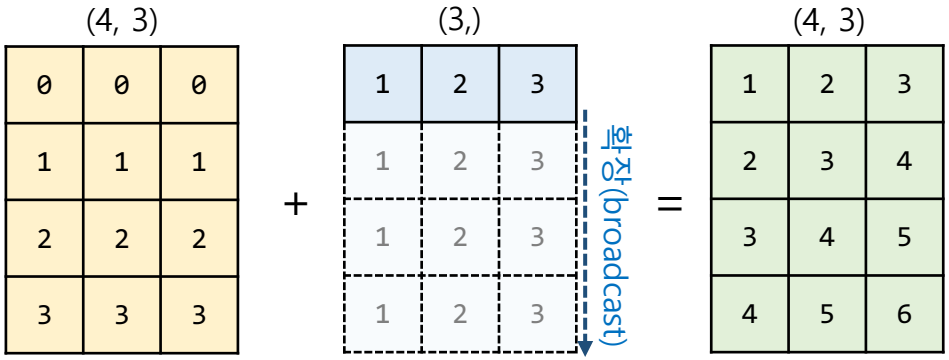
브로드캐스팅(broadcasting) #1

- 배열과 스칼라 값의 연산




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

print(a * b)
```

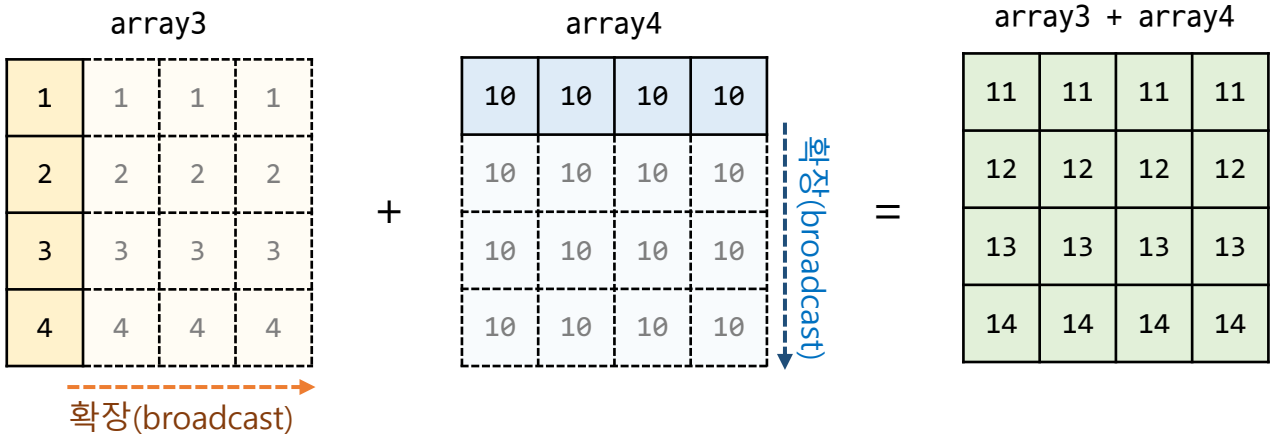
- 2차원 배열과 1차원 배열의 연산




```
array1 = np.array([[0, 0, 0],
                   [1, 1, 1],
                   [2, 2, 2],
                   [3, 3, 3]])

array2 = np.array([1, 2, 3])

print(array1 + array2)
```

브로드캐스팅(broadcasting) #2

- (4,1) 배열과 (4,) 배열 연산

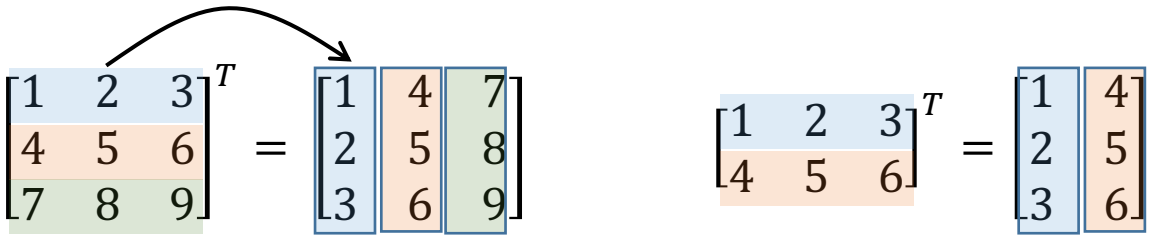



```
array3 = np.array([1, 2, 3, 4]).reshape(4, 1)
array4 = np.array([10, 10, 10, 10])

print(array3)
print('-' * 20)
print(array4)
print('-' * 20)
print(array3 + array4)
```

```
[[1]
 [2]
 [3]
 [4]]
--------------------
[10 10 10 10]
--------------------
[[11 11 11 11]
 [12 12 12 12]
 [13 13 13 13]
 [14 14 14 14]]
```

배열 조작: 전치 행렬

- 전치 행렬
  - 행과 열을 교환하여 얻는 행렬
  - numpy.transpose() 함수 또는 ndarray.T 속성 사용



$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

```
import numpy as np
a = np.arange(1, 10).reshape(3,3)
print('a')
print(a)

b = np.transpose(a)
print('np.transpose(a)')
print(b)

c = a.T
print('a.T')
print(c)
```

```
a
[[1 2 3]
 [4 5 6]
 [7 8 9]]
np.transpose(a)
[[1 4 7]
 [2 5 8]
 [3 6 9]]
a.T
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

배열 조작: 배열 배치(ravel, flatten)

- 다차원 배열을 1차원 배열로 변환
  - `ravel()`, `flatten()` 함수: 결과는 동일
- 배치 방법
  - `order='C'` (C-style 배치): 행 우선 배치 (numpy의 기본값)
  - `order='F'` (Fortran-style): 열 우선 배치

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

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

```
c = arr.ravel(order='C')
print(c)
```

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

행 우선 배치

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

```
f = arr.ravel(order='F')
print(f)
```

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

열 우선 배치

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

flatten()과 ravel() 차이점

- flatten()
  - 배열의 복사본을 반환: ndarray 객체의 함수 (ravel() 보다 느림)
- ravel()
  - 원래 배열의 view를 반환
    - view: 동일한 데이터를 가진 새로운 배열 보기 (메모리 내용을 보는 방식 변경)
  - ravel()이 반환한 뷰를 수정하면 원본 배열이 수정: Numpy 라이브러리 함수

```
# ravel() 예제
a1 = np.array([[1, 2],
               [3, 4]])

a2 = a1.ravel()
print(a2)
print('ravel()함수가 반환한 view 수정 이후 ')
a2[0] = 100

print(a2)
print(a1)
```

a2[0]의 값 및 원본 배열도 변경됨

```
[1 2 3 4]
ravel()이 반환한 view 수정 이후
[100   2   3   4]
[[100   2]
 [  3   4]]
```

```
# flatten() 예제
a1 = np.array([[1, 2],
               [3, 4]])

a2 = a1.flatten()
print(a2)
print('flatten()함수가 반환한 배열 수정 이후')
a2[0] = 100

print(a2)
print(a1)
```

a2[0]의 값만 변경

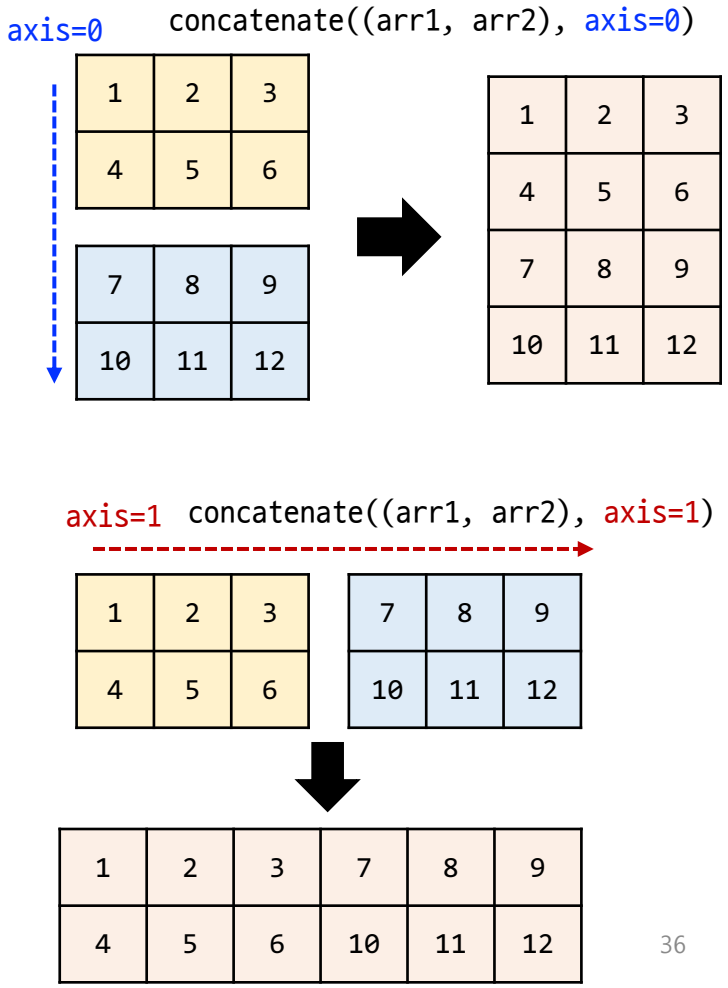
```
[1 2 3 4]
flatten()함수가 반환 배열 수정 이후
[100   2   3   4]
[[1 2]
 [3 4]]
```

배열 붙이기: concatenate()

- 배열 이어 붙이기
  - numpy.concatenate((a1, a2, ...), axis=0)
    - 설정한 축(axis)을 따라 배열을 이어 붙임

```
import numpy as np
arr1 = np.array([[1, 2, 3],
                 [4, 5, 6]])
arr2 = np.array([[7, 8, 9],
                 [10, 11, 12]])

arr3 = np.concatenate((arr1, arr2), axis=0)
print(arr3)

arr4 = np.concatenate((arr1, arr2), axis=1)
print(arr4)
```

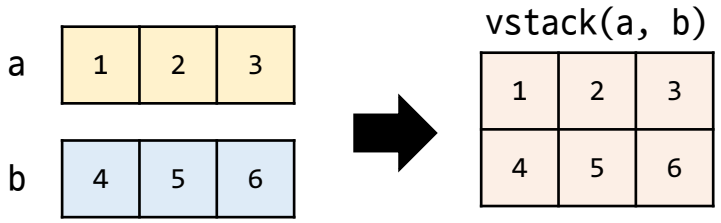
```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

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



배열 붙이기: vstack(), hstack()

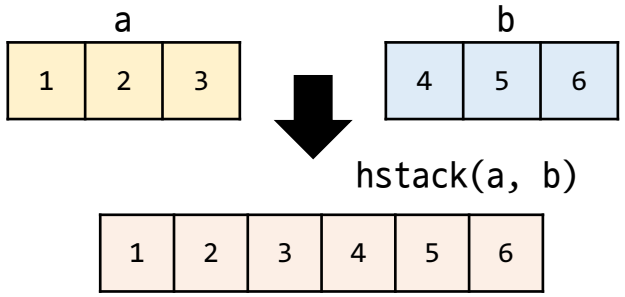
- numpy.vstack((a1, a2, ...))
  - 배열을 수직으로 연결
  - concatenate(axis=0)과 동일

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

c = np.vstack((a, b))
print(c)
```



- numpy.hstack((a1, a2, ...))
  - 배열을 수평으로 연결
  - concatenate(axis=1)과 동일

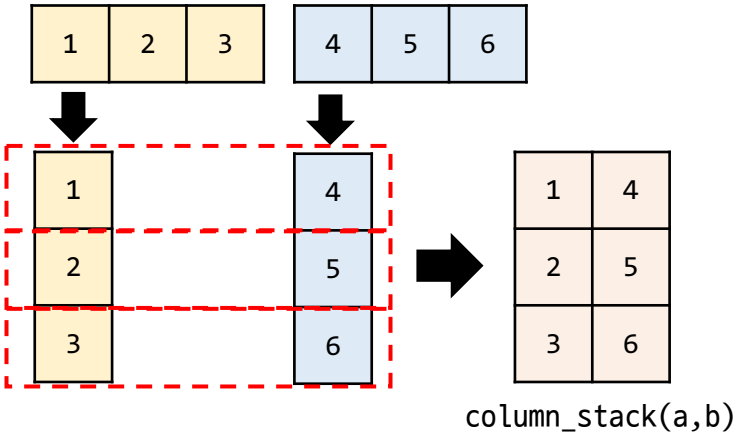
```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

d = np.hstack((a, b))
print(d)
```



배열 붙이기: column_stack()

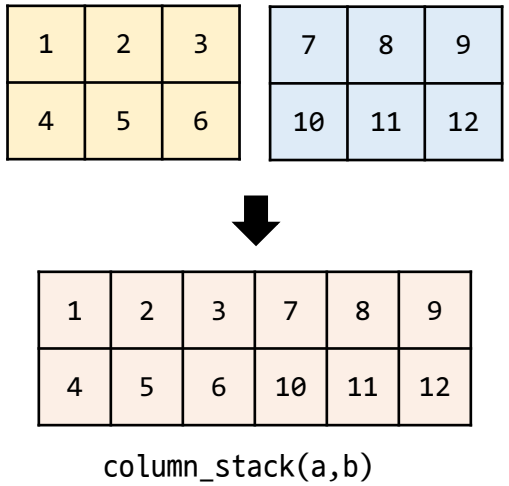
- column_stack((a1, a2, ...))
  - 1차원 배열을 열(column)방향으로 세워서 합침

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = np.column_stack((a, b))
print(c)
```



  - 2차원 배열은 hstack(), concatenate(axis=1)과 동일

```
a2 = np.array([[1, 2, 3],
               [4, 5, 6]])
b2 = np.array([[7, 8, 9],
               [10, 11, 12]])

# np.concatemate((a2, b2), axis=1) 동일
c2 = np.column_stack((a2, b2))
print(c2)
```



https://numpy.org/doc/stable/reference/generated/numpy.column_stack.html

배열 정렬: 1차원 배열

- 1차원 배열 정렬
  - np.sort(배열) 함수: 기본적으로 오름차순 정렬만 지원

```
import numpy as np
x = np.array([4, 2, 6, 5, 1, 3, 0])
sort_x = np.sort(x) # 정렬된 배열을 리턴(원본 배열은 변경 안됨)
print(x)
print(sort_x)
```

```
[4 2 6 5 1 3 0]
[0 1 2 3 4 5 6]
```

  - 역순 정렬: [::-1] 사용
    - 처음부터 끝까지 역순(-1)으로 정렬 (뒤집기)

```
x = np.array([4, 2, 6, 5, 1, 3, 0])
print(np.sort(x)[::-1])
```

```
[6 5 4 3 2 1 0]
```

배열 정렬: 2차원 배열

2	1	6
0	7	4
5	3	2

- 2차원 배열
  - axis=1: 각 행(row)을 오름차순 정렬 (좌에서 우)

```
x2 = np.array([[2, 1, 6],
               [0, 7, 4],
               [5, 3, 2]])
x2_axis1 = np.sort(x2, axis=1)
print(x2_axis1)
```

```
[[1 2 6]
 [0 4 7]
 [2 3 5]]
```

```
# 각 행을 내림차순
x2_axis1_desc = np.sort(x2, axis=1)[:,::-1]
print(x2_axis1_desc)
```

```
[[6 2 1]
 [7 4 0]
 [5 3 2]]
```

  - axis=0: 각 열(column)을 오름차순 정렬 (위에서 아래)

```
x2_axis0 = np.sort(x2, axis=0)
print(x2_axis0)
```

```
[[0 1 2]
 [2 3 4]
 [5 7 6]]
```

```
# 각 컬럼별 내림차순 정렬
x2_axis0_desc = np.sort(x2, axis=0)[::-1]
print(x2_axis0_desc)
```

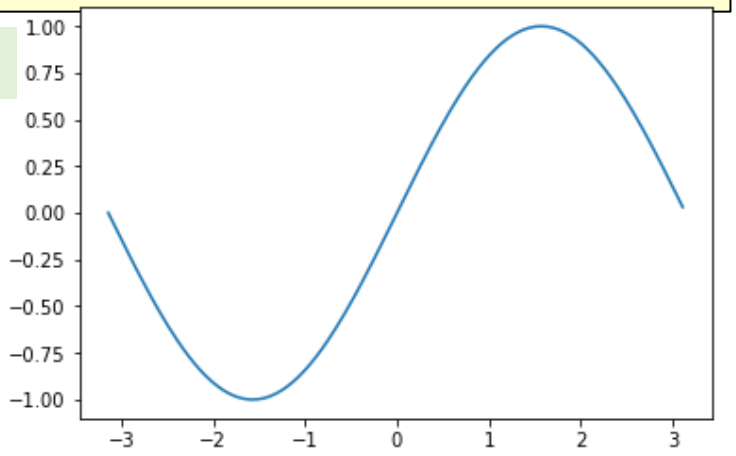
```
[[5 7 6]
 [2 3 4]
 [0 1 2]]
```

Numpy array의 다양한 활용

- 초기값 설정 지정
  - 배열에 어떤 연산이나 함수를 적용하면 배열의 모든 값이 한꺼번에 계산됨

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

a = np.zeros(10) +5 # 배열의 값이 한꺼번에 변경됨
print(a)

b = np.arange(-np.pi, np.pi, np.pi/100)
plt.plot(b, np.sin(b)) # plot([x축], [y축])
plt.show()
```

```
[5. 5. 5. 5. 5. 5. 5. 5. 5. 5.]
```



Numpy 활용

- numpy 활용

```
import numpy as np

print(np.pi)
print(np.sqrt(2))
print(np.sin(0))
print(np.cos(np.pi))
```

```
3.141592653589793
1.4142135623730951
0.0
-1.0
```

```
import numpy as np

a = np.random.rand(5)
print(a)
print(type(a))
print(np.random.choice(6, 10))
```

random.rand(n)
- 0~1 사이의 n개의 실수를 랜덤하게 생성
- [0, 1)

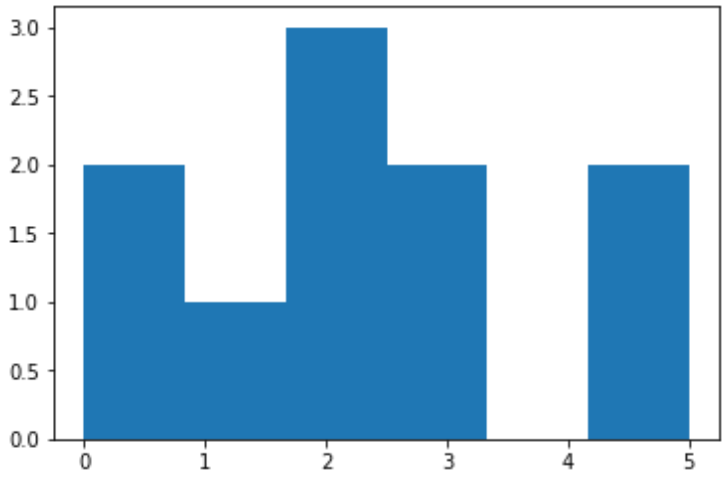
random.choice(n, m)
- 0~n-1 까지의 숫자를 랜덤하게 m개 또는 (m, n)회 뽑음
- arange(n)과 같음

```
[0.45234669 0.09113568 0.89656678 0.92394257 0.5419088 ]
<class 'numpy.ndarray'>
[2 4 1 2 1 1 2 5 3 5]
```

ndarray: N-dimensional array

numpy를 활용한 그래프 그리기

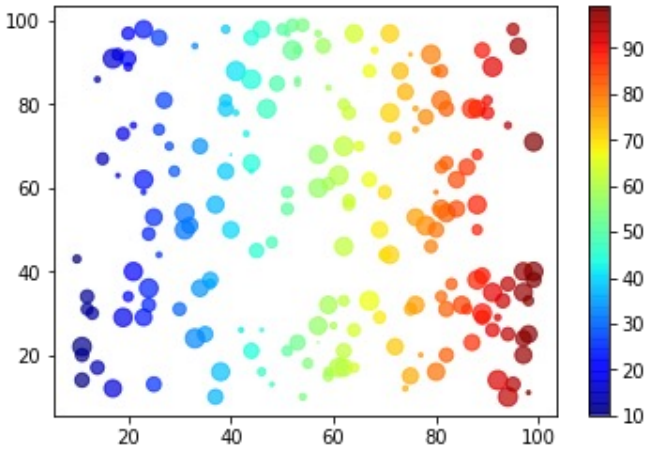
numpy 사용	기존 코드
import matplotlib.pyplot as plt import numpy as np dice = np.random.choice(6, 10) plt.hist(dice, bins=6) plt.show()	import matplotlib.pyplot as plt import random dice = [] for i in range(10): dice.append(random.randint(1, 6)) plt.hist(dice, bins=6) plt.show()
. 더 간결한 코드 . 빠른 수행 속도	



numpy를 사용한 산점도(버블 차트)

- 반복문 사용이 없음

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

# np.random.randit(low, high, size)  [Low, High)
# return: it or ndarray of ints
x = np.random.randint(10, 100, 200) # 200: 데이터 개수
y = np.random.randint(10, 100, 200)
size = np.random.rand(200) * 100 # rand(200): 0~1.0미만의 수를 200개 생성
# c: color, cmp: color map
plt.scatter(x, y, s=size, c=x, cmap='jet', alpha=0.7)
plt.colorbar()
plt.show()
```



random.choice()함수

- random.choice(n, m, replace=True)
  - replace=False: 0부터 n-1까지의 숫자 중 중복없이 m회 선택
  - n >= m 이 아닌 경우 ValueError 발생함

```
print(np.random.choice(10, 9, replace=False))

>> [5 4 8 2 6 3 7 0 9]
```

- 랜덤값에 확률 적용
  - p 속성: 각 경우의 수가 발생할 확률 설정, 확률의 합은 1
    - 총 n개 설정함 (0 ~ n-1까지의 확률)
  - random.choice(n, m, p=[0.1, 0.2, … 0.1])

```
import numpy as np

print(np.random.choice(6, 10, p=[0.1, 0.2, 0.3, 0.2, 0.1, 0.1]))
```

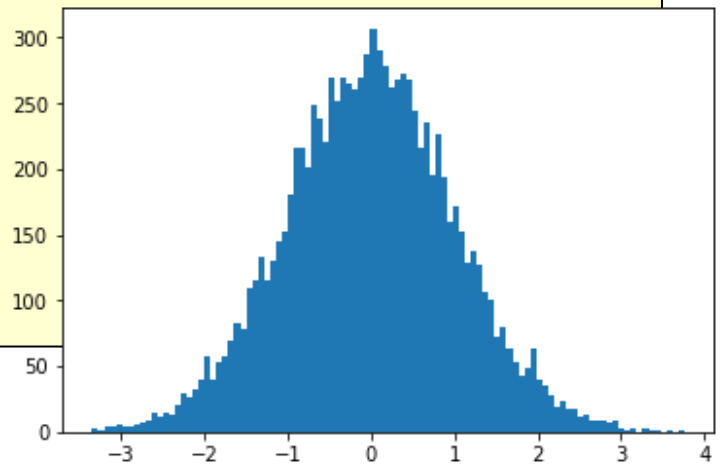
```
[3 1 4 5 2 0 3 5 3 2]
```

난수 기반 배열 생성: normal()

- random.normal()
  - 정규 분포 확률 밀도
  - np.random.normal(loc=0.0, scale=1.0, size=None)
    - 정규 분포 확률 밀도에서 표본 추출
    - loc: 정규분포의 평균, scale: 표준편차

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

mean = 0
std = 1
a = np.random.normal(mean, std, (2, 3))
print(a)

data = np.random.normal(mean, std, 10000)
plt.hist(data, bins=100)
plt.show()
```

```
[[ 1.4192442 -2.0771293 1.84898108]
[-0.12303317 1.04533993 1.94901387]]
```

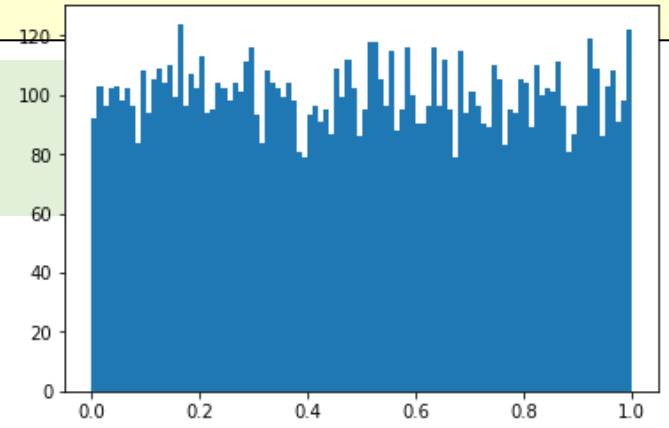


난수 기반 배열 생성: rand()

- np.random.rand(d_0, d_1, … d_n)
  - shape이 (d_0, d_1, … d_n)인 배열 생성 후 난수로 초기화
  - 난수: [0, 1)의 균등 분포(uniform distribution) 형상으로 표본 추출
  - Gaussian normal

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

a = np.random.rand(3,2) # 3x2형태의 배열 생성 후 난수로 초기화
print(a)

data = np.random.rand(10000) # 표본 10000개의 배열을 100개 구간으로 분포
plt.hist(data, bins=100) # 그래프상에 균등 분포를 보여줌
plt.show()
```

```
[[0.14124807 0.74326924]
 [0.84273887 0.90848195]
 [0.29233325 0.28027208]]
```

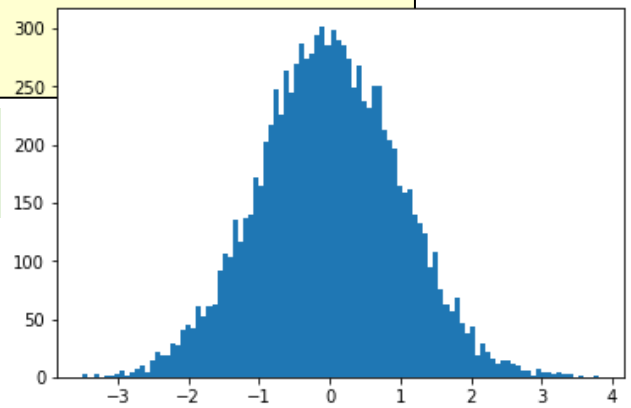


난수 기반 배열 생성: randn()

- np.random.randn(d_0, d_1, …, d_n)
  - shape이 (d_0, d_1, …, d_n) 형태인 배열 생성 후 난수로 초기화
  - 난수: 표준 정규분포 (standard normal distribution)에서 표본 추출

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

a = np.random.randn(2,4)
print(a)
data = np.random.randn(10000)

plt.hist(data, bins=100)
plt.show()
```

```
[[ 2.11186326 -0.85351437 -0.68341882 1.02986644]
[ 1.12187888 0.44885116 -1.21960995 -0.35457379]]
```

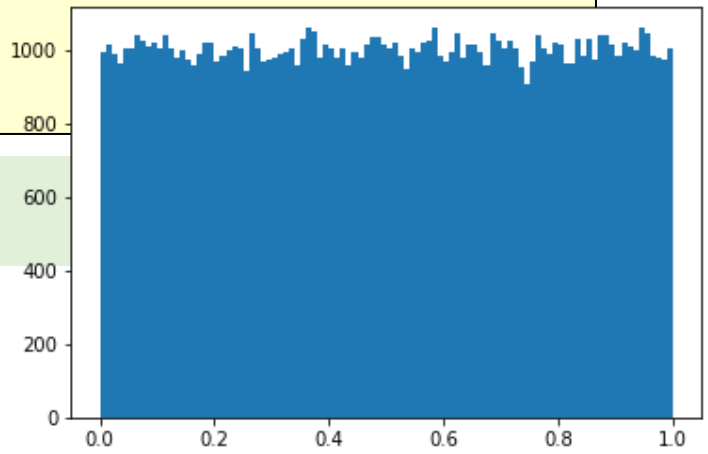


난수 기반 배열 생성: random()

- np.random.random(size=None)
  - 난수 [0, 1)의 균등 분포(uniform distribution)에서 표본 추출
  - random.rand()와 차이점: size 입력 형태(tuple)만 다름

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

a = np.random.random((2, 4))
print(a)

data = np.random.random(100000)
plt.hist(data, bins=10)
plt.show()
```

```
[[0.11719313 0.94157099 0.98318574 0.99957587]
 [0.46066523 0.37051647 0.2391908 0.8762925 ]]
```



Numpy mask 기능

- mask 기능
  - 어떤 조건에 부합하는 데이터만 선별적으로 저장하기 위한 기능

```
import numpy as np
a = np.arange(-5, 5)
print(a)
print(a[a<0])

mask1 = abs(a)>3
mask2 = abs(a) % 2 == 0
print('mask 테스트')
print(a[mask1])
print(a[mask1 + mask2]) # or 조건 (둘 중 하나의 조건이 참인 경우)
print(a[mask1 * mask2]) # and 조건 (두 조건 모두 참인 경우 )
```

```
[-5 -4 -3 -2 -1 0 1 2 3 4]
[-5 -4 -3 -2 -1]
mask 테스트
[-5 -4 4]
[-5 -4 -2 0 2 4]
[-4 4]
```

Questions?