03. IT신기술특론

-Python Library(1)

2018. 09. 17

Spring 2018

LK Lab.
Prepared by Dae-Kyeong Kim
Ph.D
Dept. of CSE
University of Gachon

Agenda

- 본 강좌는 4차 산업혁명의 요소인 AI, 딥러닝, IoT, 블록체인 중 하나인 딥러닝과 관련된 단순퍼셉트론, 다층퍼셉트론, 심층신경망, 순환신경망 등 다양한 기법에 관해 설명합니다.

  - 딥러닝의 이론을 학습하는 데 필요한 수학 지식을 익힐 수 있다.
  - 파이썬 개발환경을 구축 및 파이썬 라이브러리를 사용할 수 있다.
  - 신경망 기본형과 심층신경망(딥러닝)을 활용할 수 있다.
  - 시계열 데이터처리를 위한 RNN학습과 응용을 할 수 있다.

학습일정 및 내용

주차	기간	수업내용및학습활동	비고
1	03/02~03/08	실라버스 파이썬기초-1파이썬프로그램실행,2데이터형,3변수,4데이터구조	
2	03/09~03/15	파이썬기초-5연산,6기본구문,7함수,8클래스	
3	03/16~03/22	파이썬기초 9라이브러리-NumPy, 딥러닝을위한라이브러리, TensorFlow, 케라스(Keras), 씨아노(Theano)	
4	03/23~03/29	퍼셉트론	4차 산업혁명과 딥러닝
5	03/30~04/05	신경망	
6	04/06~04/12	신경망학습	
7	04/13~04/19	오차역전파법	
8	04/20~04/26	중간고사	신경망

학습일정 및 내용

주차	기간	수업내용및학습활동	비고
9	04/27~05/03	학습관련기술들1	
10	05/04~05/10	학습관련기술들2	
11	05/11~05/17	RNN1	
12	05/18~05/24	RNN2	심층신경망
13	05/25~05/31	CNN1	
14	06/01~06/07	CNN2	
15	06/08~06/14	딥러닝의미래	순환신경망
16	06/15~06/21	기말고사	

CONTENTS

1. NumPy 개요

- 파이썬 기반 데이터 분석 환경에서 NumPy 는 행렬 연산을 위한 핵심 라이브러리
- NumPy는 "**Numerical Python**"의 약자로 대규모 다차원 배열과 행렬 연산에 필요한 다양한 함수를 제공
- 특히 메모리 버퍼에 배열 데이터를 저장하고 처리하는 효율적인 인터페이스를 제공
- 파이썬 list 객체를 개선한 NumPy의 ndarray 객체를 사용하면 더 많은 데이터를 더 빠르게 처리할 수 있다.
- 배열 클래스. 딥러닝 구현시 배열, 열, 행렬 계산에 많이 사용.

- NumPy는 다음과 같은 특징을 갖습니다.
  - 강력한 N 차원 배열 객체
  - 정교한 브로드케스팅(Broadcast) 기능
  - C/C ++ 및 포트란 코드 통합 도구
  - 유용한 선형 대수학, 푸리에 변환 및 난수 기능
  - 범용적 데이터 처리에 사용 가능한 다차원 컨테이너

http://cs231n.github.io/python-numpy-tutorial/

1.넘파이 가져오기

- NumPy는 과학 연산을 위한 파이썬 핵심 라이브러리
- NumPy는 고성능 다차원 배열과 이런 배열을 처리하는 다양한 함수와 툴을 제공
- 파이썬에서 NumPy를 사용할 때, 다음과 같이 numpy 모듈을 "np"로 임포트하여 사용

```
(C:₩DEV₩Anaconda3) C:₩Users₩Teacher>python
Python 3.6.1 |Anaconda 4.4.0 (64-bit)| (default, May 11 2017, 13:25:24) [MSC v.1900 64 bit (AMD64)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>>
```

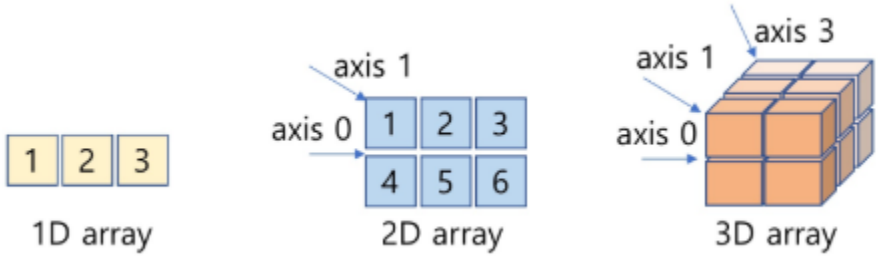
- NumPy 라이브러리 버전 확인

```
>>> np.__version__
'1.12.1'
>>>
```

http://cs231n.github.io/python-numpy-tutorial/

2. 넘파이 배열 생성하기

- NumPy 배열은 <그림 >과 같이 다차원 배열을 지원
- NumPy 배열의 구조는 "**Shape**"으로 표현
- Shape은 배열의 구조를 파이썬 튜플 자료형을 이용하여 정의
- 예를 들어 28X28 컬러 사진은 높이가 28, 폭이 28, 각 픽셀은 3개 채널(RGB)로 구성된 데이터 구조를 갖는다.
- 즉 컬러 사진 데이터는 Shaep이 (28, 28, 3)인 3차원 배열
- 다차원 배열은 입체적인 데이터 구조를 가지며, 데이터의 차원은 여러 갈래의 데이터 방향을 갖는다.
- 다차원 배열의 데이터 방향을 axis로 표현 가능
- 행방향(높이), 열방향(폭), 채널 방향은 각각 axis=0, axis=1 그리고 axis=2로 지정
- NumPy 집계(Aggregation) 함수는 배열 데이터의 집계 방향을 지정하는 axis 옵션을 제공



NumPy 1차원, 2차원 및 3차원 배열과 Axis

2. 넘파이 배열 생성하기

- NumPy 배열은 numpy.ndarray 객체

```
>>> x = np.array([1.0, 2.0, 3.0])
>>> print(x)
[ 1.  2.  3.]
>>> type(x)
<class 'numpy.ndarray'>
```

- NumPy 객체의 정보를 출력하는 용도로 다음 pprint 함수를 공통으로 사용

```
>>> def pprint(arr):
...     print("type:{}".format(type(arr)))
...     print("shape: {}, dimension: {}, dtype:{}".format(arr.shape, arr.ndim, arr.dtype))
...     print("Array's Data:₩n", arr)
...
>>>
```

2. 넘파이 배열 생성하기

2.1 파이썬 배열로 NumPy 배열 생성

- 파이썬 배열을 인자로 NumPy 배열을 생성
- 파라미터로 list 객체와 데이터 타입(dtype)을 입력하여 NumPy 배열을 생성
- dtype을 생략할 경우, 입력된 list 객체의 요소 타입이 설정

- 파이썬 1차원 배열(list)로 NumPy 배열 생성

```
>>> arr = [1, 2, 3]
>>> a = np.array([1, 2, 3])
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (3,), dimension: 1, dtype:int32
Array's Data:
 [1 2 3]
>>>
```


2. 넘파이 배열 생성하기

2.1 파이썬 배열로 NumPy 배열 생성

- 파이썬 2차원 배열로 NumPy 배열 생성, 원소 데이터 타입 지정

```
>>> arr = [(1,2,3), (4,5,6)]
>>> a= np.array(arr, dtype = float)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:float64
Array's Data:
 [[ 1.  2.  3.]
 [ 4.  5.  6.]]
>>>
```

2. 넘파이 배열 생성하기

2.1 파이썬 배열로 NumPy 배열 생성

- 파이썬 3차원 배열로 NumPy 배열 생성, 원소 데이터 타입 지정

```
>>> arr = np.array([[[1,2,3], [4,5,6]], [[3,2,1], [4,5,6]]], dtype = float)
>>> a= np.array(arr, dtype = float)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 2, 3), dimension: 3, dtype:float64
Array's Data:
 [[[ 1.  2.  3.]
  [ 4.  5.  6.]]

 [[ 3.  2.  1.]
  [ 4.  5.  6.]]]
>>>
```


2. 넘파이 배열 생성하기

2.2 배열 생성 및 초기화

- Numpy는 원하는 shape으로 배열을 설정하고, 각 요소를 특정 값으로 초기화하는 zeros, ones, full, eye 함수를 제공
- 또한, 파라미터로 입력한 배열과 같은 shape의 배열을 만드는 zeros_like, ones_like, full_like 함수도 제공
- 이 함수를 이용하여 배열 생성하고 초기화할 수 있다.

- **np.zeros 함수**
- zeros(shape, dtype=float, order='C')
- 지정된 shape의 배열을 생성하고, 모든 요소를 0으로 초기화

```
>>> a = np.zeros((3, 4))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (3, 4), dimension: 2, dtype:float64
Array's Data:
 [[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
>>>
```


2. 넘파이 배열 생성하기

2.2 배열 생성 및 초기화

- **np.ones 함수**
- np.ones(shape, dtype=None, order='C')
- 지정된 shape의 배열을 생성하고, 모든 요소를 1로 초기화

```
>>> a = np.ones((2,3,4),dtype=np.int16)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 3, 4), dimension: 3, dtype:int16
Array's Data:
 [[[1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]]

 [[1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]]]
>>>
```

2. 넘파이 배열 생성하기

2.2 배열 생성 및 초기화

- **np.full 함수**
- np.full(shape, fill_value, dtype=None, order='C')
- 지정된 shape의 배열을 생성하고, 모든 요소를 지정한 "fill_value"로 초기화

```
>>> a = np.full((2,2),7)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 2), dimension: 2, dtype:int32
Array's Data:
 [[7 7]
 [7 7]]
>>>
```


2. 넘파이 배열 생성하기

2.2 배열 생성 및 초기화

- **np.eye 함수**
- np.eye(N, M=None, k=0, dtype=<class 'float'>)
- (N, N) shape의 단위 행렬(Unit Matrix)을 생성

```
>>> np.eye(4)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
>>>
```

2. 넘파이 배열 생성하기

2.2 배열 생성 및 초기화

- **np.empty 함수**
- empty(shape, dtype=float, order='C')
- 지정된 shape의 배열 생성
- 요소의 초기화 과정에 없고, 기존 메모리값을 그대로 사용
- 배열 생성비용이 가장 저렴하고 빠름
- 배열 사용 시 주의가 필요(초기화를 고려)

```
>>> a = np.empty((4,2))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (4, 2), dimension: 2, dtype:float64
Array's Data:
 [[  0.00000000e+000    0.00000000e+000]
 [  0.00000000e+000    0.00000000e+000]
 [  0.00000000e+000    1.83792420e-321]
 [  1.31668765e-252    1.16816378e-307]]
>>>
```

2. 넘파이 배열 생성하기

2.2 배열 생성 및 초기화

- **like 함수**
- numpy는 지정된 배열과 shape이 같은 행렬을 만드는 like 함수를 제공합니다.
- np.zeros_like
- np.ones_like
- np.full_like
- np.enpty_like

```
>>> a = np.array([[1,2,3], [4,5,6]])
>>> b = np.ones_like(a)
>>> pprint(b)
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:int32
Array's Data:
 [[1 1 1]
 [1 1 1]]
>>>
```


2. 넘파이 배열 생성하기

2.3 데이터 생성 함수

- NumPy는 주어진 조건으로 데이터를 생성한 후, 배열을 만드는 데이터 생성 함수를 제공합니다.
- numpy.linspace
- numpy.arange
- numpy.logspace

- **np.linspace 함수**
- numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
- start부터 stop의 범위에서 num개를 균일한 간격으로 데이터를 생성하고 배열을 만드는 함수
- 요소 개수를 기준으로 균등 간격의 배열을 생성

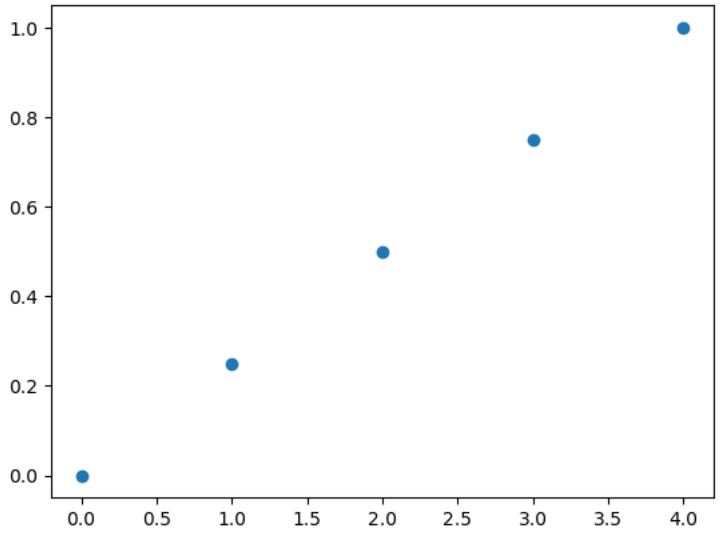
```
>>> a = np.linspace(0, 1, 5)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (5,), dimension: 1, dtype:float64
Array's Data:
 [ 0.    0.25  0.5   0.75  1.  ]
>>>
```

2. 넘파이 배열 생성하기

2.3 데이터 생성 함수

- **np.linspace 함수**
- numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
- start부터 stop의 범위에서 num개를 균일한 간격으로 데이터를 생성하고 배열을 만드는 함수
- 요소 개수를 기준으로 균등 간격의 배열을 생성

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(a, 'o')
[<matplotlib.lines.Line2D object at 0x00000206243AD978>]
>>> plt.show()
```

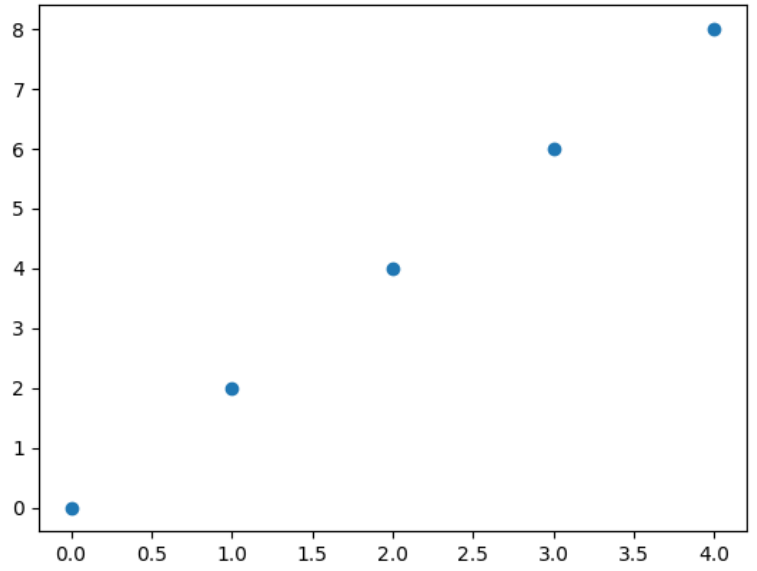


2. 넘파이 배열 생성하기

2.3 데이터 생성 함수

- **np.arange 함수**
- numpy.arange([start,] stop[, step,], dtype=None)
- start부터 stop 미만까지 step 간격으로 데이터 생성한 후 배열을 만듦
- 범위내에서 간격을 기준 균등 간격의 배열
- 요소의 객수가 아닌 데이터의 간격을 기준으로 배열 생성

```
>>> a = np.arange(0, 10, 2, np.float)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (5,), dimension: 1, dtype:float64
Array's Data:
 [ 0.  2.  4.  6.  8.]
>>>

>>> import matplotlib.pyplot as plt
>>> plt.plot(a, 'o')
[<matplotlib.lines.Line2D object at 0x0000020623B2ADA0>]
>>> plt.show()
```

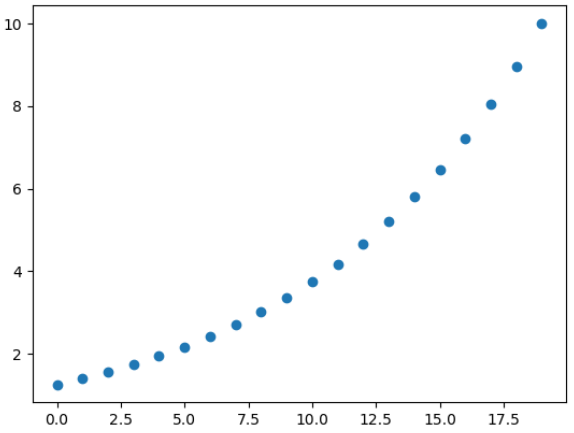


2. 넘파이 배열 생성하기

2.3 데이터 생성 함수

- **np.logspace 함수**
- numpy.logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None)
- 로그 스케일의 linspace 함수
- 로그 스케일로 지정된 범위에서 num 개수만큼 균등 간격으로 데이터 생성한 후 배열 만듦

```
>>> a = np.logspace(0.1, 1, 20, endpoint=True)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (20,), dimension: 1, dtype:float64
Array's Data:
 [  1.25892541   1.40400425   1.565802     1.74624535   1.94748304
    2.1719114    2.42220294   2.70133812   3.0126409    3.35981829
    3.74700446   4.17881006   4.66037703   5.19743987   5.79639395
    6.46437163   7.2093272    8.04013161   8.9666781   10.         ]
>>>

>>> import matplotlib.pyplot as plt
>>> plt.plot(a, 'o')
[<matplotlib.lines.Line2D object at 0x0000020623E9C710>]
>>> plt.show()
```



2. 넘파이 배열 생성하기

2.4 난수 기반 배열 생성

- NumPy는 난수 발생 및 배열 생성을 생성하는 numpy.random 모듈을 제공합니다. 이 절에서는 이 모듈의 함수 사용법을 소개합니다. numpy.random 모듈은 다음과 같은 함수를 제공합니다.
- np.random.normal
- np.random.rand
- np.random.randn
- np.random.randint
- np.random.random

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

2. 넘파이 배열 생성하기

2.4 난수 기반 배열 생성

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

```
>>> mean = 0
>>> std = 1
>>> a = np.random.normal(mean, std, (2, 3))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:float64
Array's Data:
 [[ 1.34847043   0.86691391   0.10589266]
 [ 0.45577562   1.91859771   1.25051526]]
>>>
```

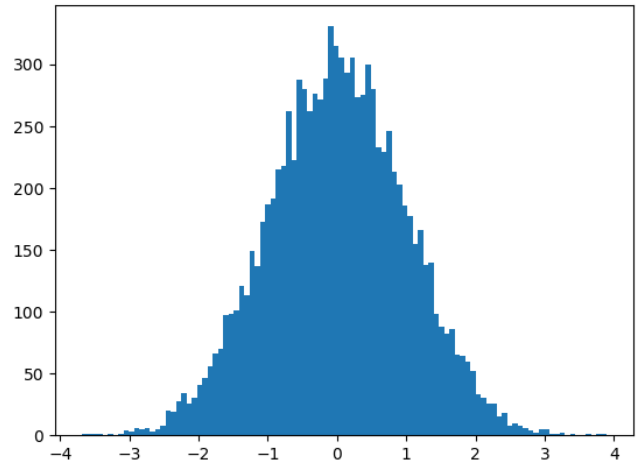

2. 넘파이 배열 생성하기

2.4 난수 기반 배열 생성

- **np.random.normal**

•np.random.normal이 생성한 난수는 정규 분포의 형상을 갖는다.
•아래 예제는 정규 분포로 10000개 표본을 뽑은 결과를 히스토그램으로 표현한 예이다.
•표본 10000개의 배열을 100개 구간으로 구분할 때, 정규 분포 형태를 보이고 있다

```
>>> data = np.random.normal(0, 1, 10000)
>>> import matplotlib.pyplot as plt
>>> plt.hist(data, bins=100)
(array([   1.,    1.,    1.,    1.,    0.,    1.,    0.,    1.,    4.,
          3.,    6.,    5.,    6.,    3.,    5.,    8.,   20.,   19.,
..
          8.,   10.,    8.,    6.,    4.,    2.,    5.,    5.,    1.,
          1.,    2.,    0.,    1.,    0.,    0.,    1.,    0.,    1.,    1.]), array([-3.69381374, -3.61786661, -3.54191947,
-3.46597233, -3.39002519,
       -3.31407805, -3.23813091, -3.16218378, -3.08623664, -3.0102895 ,
       -2.93434236, -2.85839522, -2.78244808, -2.70650095, -2.63055381,
...
        3.5211644 ,  3.59711154,  3.67305868,  3.74900582,  3.82495295,
        3.90090009]), <a list of 100 Patch objects>)
>>> plt.show()
```



2. 넘파이 배열 생성하기

2.4 난수 기반 배열 생성

- **np.random.rand**
- numpy.random.rand(d0, d1, ..., dn)
- Shape이 (d0, d1, ..., dn) 인 배열 생성 후 난수로 초기화
- 난수: [0. 1)의 균등 분포(Uniform Distribution) 형상으로 표본 추출
- Gaussina normal

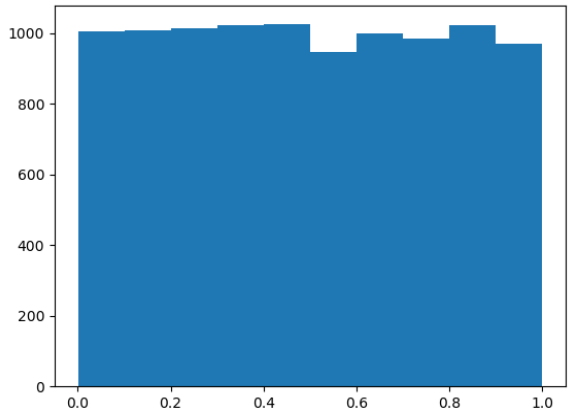
```
>>> a = np.random.rand(3,2)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (3, 2), dimension: 2, dtype:float64
Array's Data:
 [[ 0.83302772   0.78445019]
 [ 0.51604274   0.83432893]
 [ 0.33137418   0.07590928]]
>>>
```

2. 넘파이 배열 생성하기

2.4 난수 기반 배열 생성

- **np.random.rand**
- np.random.rand는 균등한 비율로 표본 추출
- 다음 예제는 균등 분포로 10000개를 표본 추출한 결과를 히스토그램으로 표현한 예
- 표본 10000개의 배열을 10개 구간으로 구분했을때 균등한 분포 형태를 보이고 있다.

```
>>> data = np.random.rand(10000)
>>> import matplotlib.pyplot as plt
>>> plt.hist(data, bins=10)
(array([ 1006.,  1009.,  1013.,  1022.,  1026.,   947.,  1001.,   984.,
         1022.,   970.]), array([  1.57091974e-04,   1.00136491e-01,   2.00115890e-01,
         3.00095288e-01,   4.00074687e-01,   5.00054086e-01,
         6.00033485e-01,   7.00012884e-01,   7.99992283e-01,
         8.99971681e-01,   9.99951080e-01]), <a list of 10 Patch objects>)
>>> plt.show()
```



2. 넘파이 배열 생성하기

2.4 난수 기반 배열 생성

- **p.random.randn**
- numpy.random.randn(d0, d1, ..., dn)
- (d0, d1, ..., dn) shape 배열 생성 후 난수로 초기화
- 난수: 표준 정규 분포(standard normal distribution)에서 표본 추출

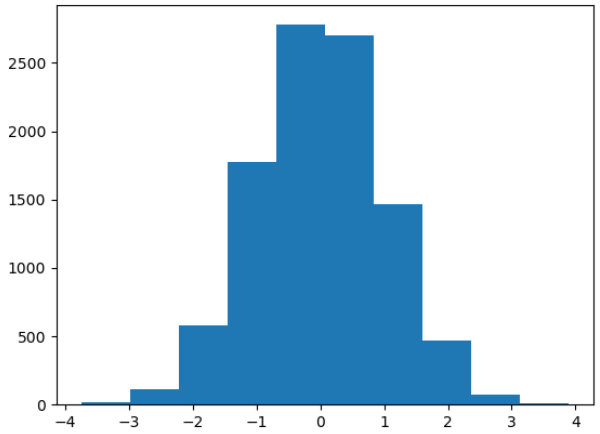
```
>>> a = np.random.randn(2, 4)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 4), dimension: 2, dtype:float64
Array's Data:
 [[ 1.076367    1.692743   -2.0075129  -1.39008854]
 [ 1.18616278  0.3596576  -0.92842398 -0.32057655]]
>>>
```

2. 넘파이 배열 생성하기

2.4 난수 기반 배열 생성

- **p.random.randn**
- np.random.randn은 정규 분포로 표본 추출
- 다음 예제는 정규 분포로 10000개를 표본 추출한 결과를 히스토그램으로 표현한 예
- 표본 10000개의 배열을 10개 구간으로 구분했을때 정규 분포 형태를 보이고 있다.

```
>>> data = np.random.randn(10000)
>>> import matplotlib.pyplot as plt
>>> plt.hist(data, bins=10)
(array([   18.,   117.,   577.,  1778.,  2780.,  2705.,  1468.,   469.,
          75.,    13.]), array([-3.75755727, -2.99170692, -2.22585656, -1.46000621, -0.69415585,
        0.0716945 ,  0.83754486,  1.60339521,  2.36924556,  3.13509592,
        3.90094627]), <a list of 10 Patch objects>)
>>> plt.show()
```



2. 넘파이 배열 생성하기

2.4 난수 기반 배열 생성

- **p.random.randint**
- numpy.random.randint(low, high=None, size=None, dtype='l')
- 지정된 shape으로 배열을 만들고 low 부터 high 미만의 범위에서 정수 표본 추출

```
>>> a = np.random.randint(5, 10, size=(2, 4))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 4), dimension: 2, dtype:int32
Array's Data:
 [[7 8 9 7]
 [9 8 5 7]]
>>>
```

```
>>> a = np.random.randint(1, size=10)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (10,), dimension: 1, dtype:int32
Array's Data:
 [0 0 0 0 0 0 0 0 0 0]
>>>
```

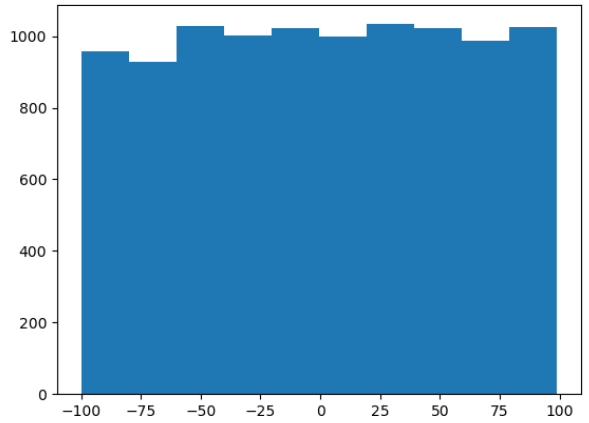
2. 넘파이 배열 생성하기

2.4 난수 기반 배열 생성

- **p.random.randint**

100에서 100의 범위에서 정수를 균등하게 표본 추출
다음 예제에서 균등 분포로 10000개를 표본 추출한 결과를 히스토그램으로 표현한 예.
표본 10000개의 배열을 10개 구간으로 구분했을 때 균등한 분포 형태를 보이고 있다.

```
>>> data = np.random.randint(-100, 100, 10000)
>>> import matplotlib.pyplot as plt
>>> plt.hist(data, bins=10)
(array([ 957.,  929., 1029., 1001., 1021.,  998., 1034., 1021.,
         986., 1024.]), array([-100. , -80.1, -60.2, -40.3, -20.4,  -0.5,  19.4,  39.3,
         59.2,  79.1,  99. ]), <a list of 10 Patch objects>)
>>> plt.show()
```



2. 넘파이 배열 생성하기

2.4 난수 기반 배열 생성

- **np.ramdom.random**
- np.random.random(size=None)¶
- 난수: [0., 1.)의 균등 분포(Uniform Distribution)에서 표본 추출

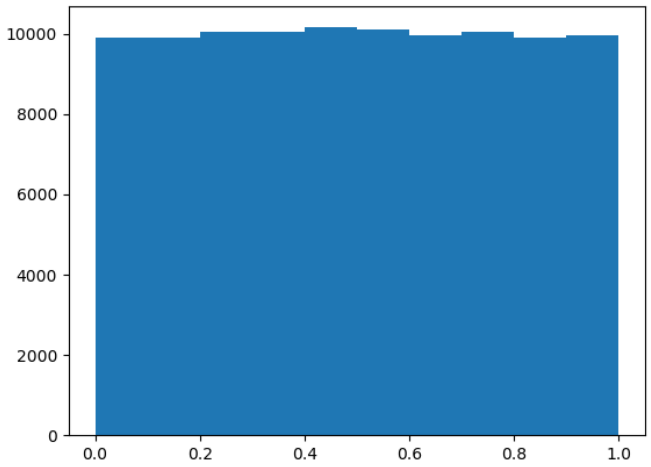
```
>>> a = np.random.random((2, 4))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 4), dimension: 2, dtype:float64
Array's Data:
 [[ 0.31624937   0.16938419   0.78851881   0.84421213]
 [ 0.85589263   0.02745719   0.02203479   0.02622456]]
>>>
```


2. 넘파이 배열 생성하기

2.4 난수 기반 배열 생성

- **np.ramdom.random**
- np.random.random은 균등 분포로 표본을 추출.
- 다음 예제는 정규 분포로 10000개를 표본 추출한 결과를 히스토그램으로 표현한 예.
- 표본 10000개의 배열을 10개 구간으로 구분했을때 정규 분포 형태를 보이고 있다.

```
>>> data = np.random.random(100000)
>>> import matplotlib.pyplot as plt
>>> plt.hist(data, bins=10)
(array([ 9891.,  9894., 10051., 10052., 10159., 10095.,  9949.,
        10054.,  9898.,  9957.]), array([ 1.58830560e-05,  1.00013854e-01,  2.00011826e-01,
         3.00009797e-01,  4.00007768e-01,  5.00005739e-01,
         6.00003710e-01,  7.00001682e-01,  7.99999653e-01,
         8.99997624e-01,  9.99995595e-01]), <a list of 10 Patch objects>)
>>> plt.show()
```



2. 넘파이 배열 생성하기

2.5 약속된 난수

- 난수는 특정 시작 숫자를 설정함으로써 난수 발생을 할 수 있다. 난수의 시작점을 설정하는 함수가 np.random.seed 다.

- random 모듈의 함수는 실행할 때 마다 무작위 수를 반환

```
>>> np.random.random((2, 2))
array([[ 0.68822065,  0.30232855],
       [ 0.29526186,  0.76374219]])
>>> np.random.randint(0, 10, (2, 3))
array([[3, 1, 3],
       [8, 0, 2]])
>>> np.random.random((2, 2))
array([[ 0.42142068,  0.8870935 ],
       [ 0.12229069,  0.88120449]])
>>> np.random.randint(0, 10, (2, 3))
array([[4, 8, 2],
       [2, 1, 8]])
>>>
```


2. 넘파이 배열 생성하기

2.5 약속된 난수

- np.random.seed 함수를 이용한 무작위수 재연
- np.random.seed(100)을 기준으로 동일한 무작위수로 초기화된 배열이 만들어 지고 있다.

```
>>> np.random.seed(100)
>>> np.random.random((2, 2))
array([[ 0.54340494,  0.27836939],
       [ 0.42451759,  0.84477613]])
>>> np.random.randint(0, 10, (2, 3))
array([[4, 2, 5],
       [2, 2, 2]])
>>> # seed 값 재 설정
...
>>> np.random.seed(100)
>>> np.random.random((2, 2))
array([[ 0.54340494,  0.27836939],
       [ 0.42451759,  0.84477613]])
>>> np.random.randint(0, 10, (2, 3))
array([[4, 2, 5],
       [2, 2, 2]])
>>>
```

3. NumPy 입출력

…

- NumPy는 배열 객체를 바이너리 파일 혹은 텍스트 파일에 저장하고 로딩하는 기능을 제공

함수명	기능	파일포멧
np.save()	NumPy 배열 객체 1개를 파일에 저장	바이너리
np.savez()	NumPy 배열 객체 복수개를 파일에 저장	바이너리
np.load()	NumPy 배열 저장 파일로 부터 객체 로딩	바이너리
np.loadtxt()	텍스트 파일로 부터 배열 로딩	텍스트
np.savetxt()	텍스트 파일에 NumPy 배열 객체 저장	텍스트

- 예제로 다음과 같은 a, b 두 개 배열을 사용

```
>>> a = np.random.randint(0, 10, (2, 3))
>>> b = np.random.randint(0, 10, (2, 3))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:int32
Array's Data:
 [[1 0 8]
 [4 0 9]]
>>> pprint(b)
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:int32
Array's Data:
 [[6 2 4]
 [1 5 3]]
>>>
```

3. NumPy 입출력

1. 배열 객체 저장

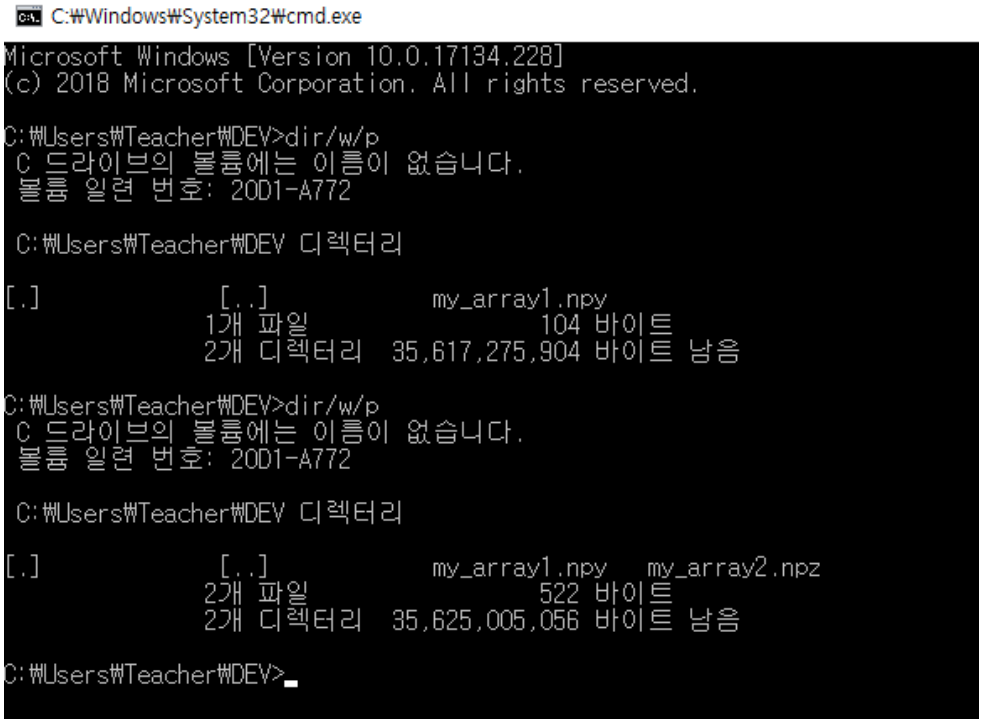
- np.save 함수와 np.savez 함수를 이용하여 배열 객체를 파일로 저장할 수 있음
- np.save: 1개 배열 저장, 확장자: npy
- np.savez: 복수 배열을 1개의 파일에 저장, 확장자: pnz
- 배열 저장 파일은 바이너리 형태

1개 배열 파일에 저장

```
>>> np.save("./my_array1", a)
```

복수 배열을 1개 파일에 저장

```
>>> np.savez("my_array2", a, b)
```



3. NumPy 입출력

2.파일로 부터 배열 객체 로딩

npy와 npz 파일은 np.load 함수로 읽을 수 있다.

```
>>> np.load("./my_array1.npy")
array([[2, 5, 7],
       [1, 3, 7]])
>>> npzfiles = np.load("./my_array2.npz")
>>> npzfiles.files
['arr_0', 'arr_1']
>>> npzfiles['arr_0']
array([[2, 5, 7],
       [1, 3, 7]])
>>> npzfiles['arr_1']
array([[0, 1, 9],
       [1, 7, 2]])
>>>
```


3. NumPy 입출력

3. 텍스트 파일 로딩

- 텍스트 파일을 np.loadtxt 로 로딩 할 수 있다.
- np.loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0)
  - fname: 파일명
  - dtype: 데이터 타입
  - comments: comment 시작 부호
  - delimiter: 구분자
  - skiprows: 제외 라인 수(header 제거용)

```
>>> np.loadtxt("./simple.csv")
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> np.loadtxt("./simple.csv", dtype=np.int)
array([[1, 2, 3],
       [4, 5, 6]])
>>>
```

```
C:₩Users₩Teacher₩DEV>copy con simple.csv
1 2 3
4 5 6
        1개 파일이 복사되었습니다.

C:₩Users₩Teacher₩DEV>cat simple.csv
1 2 3
4 5 6

C:₩Users₩Teacher₩DEV>
```

3. NumPy 입출력

3. 텍스트 파일 로딩

- **문자열을 포함하는 텍스트 파일 로딩**
- president_height.csv 파일은 숫자와 문자를 모두 포함하는 데이터 파일.
- dtype을 이용하여 컬럼 타입을 지정하여 로딩.
- delimiter와 skiprows를 이용하여 구분자와 무시해야 할 라인을 지정

```
>>> data = np.loadtxt("./height.csv", delimiter=",", skiprows=1, dtype={'names':
("Num","Name","Height"),'formats':('i', 'S20', 'f')})
>>> data[:3]
array([(1, b'Mr.Hong',  189.), (2, b'Mr.Park',  170.)],
      dtype=[('Num', '<i4'), ('Name', 'S20'), ('Height', '<f4')])
>>>
```

```
C:₩Users₩Teacher₩DEV>copy con height.csv
Num,Name,Height
1,Mr.Hong,189
2,Mr.Park,170

        1개 파일이 복사되었습니다.

C:₩Users₩Teacher₩DEV>cat height.csv
Num,Name,Height
1,Mr.Hong,189
2,Mr.Park,170
```


3. NumPy 입출력

4. 배열 객체 텍스트 파일로 저장

np.savetxt 함수를 이용하여 배열 객체를 텍스트 파일로 저장할 수 있다.
np.savetxt(fname, X, fmt='%.18e', delimiter=' ', newline='₩n', header='', footer='', comments='# ')

```
>>> data = np.random.random((3, 4))
>>> pprint(data)
type:<class 'numpy.ndarray'>
shape: (3, 4), dimension: 2, dtype:float64
Array's Data:
 [[ 0.15954889  0.92842296  0.12991699  0.67435897]
 [ 0.40469834  0.34016756  0.52156657  0.77371504]
 [ 0.82916683  0.96248644  0.61407079  0.46028263]]
>>>
>>>
>>> np.savetxt("./saved.csv", data, delimiter=",")
>>>
```

```
C:₩Users₩Teacher₩DEV 디렉터리

[.]             [..]            height.csv      my_array1.npy
my_array2.npz   saved.csv       simple.csv
               5개 파일                 885 바이트
               2개 디렉터리  35,621,416,960 바이트 남음

C:₩Users₩Teacher₩DEV>cat saved.csv
1.59548887440735920e-01,9.28422957058865572e-
01,1.29916991103899315e-
01,6.74358973977665976e-01
4.04698339204071277e-01,3.40167562508095766e-
01,5.21566572638469727e-
01,7.73715039474358046e-01
8.29166833216708143e-01,9.62486441667629866e-
01,6.14070793035856655e-
01,4.60282633547186904e-01

C:₩Users₩Teacher₩DEV>
```

4. 데이터 타입

...

- NumPy는 다음과 같은 데이터 타입을 지원
- 배열을 생성할 때 dtype속성으로 다음과 같은 데이터 타입을 지정할 수 있다.
- np.int64 : 64 비트 정수 타입
- np.float32 : 32 비트 부동 소수 타입
- np.complex : 복소수 (128 float)
- np.bool : 불린 타입 (Trur, False)
- np.object : 파이썬 객체 타입
- np.string_ : 고정자리 스트링 타입
- np.unicode_ : 고정자리 유니코드 타입

5. 배열 상태 검사(Inspecting)

...

- NumPy는 배열의 상태를 검사하는 다음과 같은 방법을 제공

배열 속성 검사 항목	배열 속성 확인 방법	예시	결과
배열 shape	np.ndarray.shape 속성	arr.shape	(5, 2, 3)
배열 길이	일차원의 배열 길이 확인	len(arr)	5
배열 차원	np.ndarray.ndim 속성	arr.ndim	3
배열 요소 수	np.ndarray.size 속성	arr.size	30
배열 타입	np.ndarray.dtype 속성	arr.dtype	dtype('float64')
배열 타입 명	np.ndarray.dtype.name 속성	arr.dtype.name	float64
배열 타입 변환	np.ndarray.astype 함수	arr.astype(np.int)	배열 타입 변환

5. 배열 상태 검사(Inspecting)

...

- NumPy 배열 객체는 다음과 같은 방식으로 속성을 확인

```
>>> arr = np.random.random((5,2,3))
>>> type(arr)
<class 'numpy.ndarray'>
>>> arr.shape
(5, 2, 3)
>>> # 배열의 길이
... len(arr)
5
>>>
>>>
>>> #데모 배열 객체 생성
... arr = np.random.random((5,2,3))
>>> #배열 타입 조회
... type(arr)
<class 'numpy.ndarray'>
>>> # 배열의 shape 확인
... arr.shape
(5, 2, 3)
```


5. 배열 상태 검사(Inspecting)

...

- NumPy 배열 객체는 다음과 같은 방식으로 속성을 확인

```
>>> # 배열의 길이
... len(arr)
5
>>> # 배열의 차원 수
... arr.ndim
3
>>> # 배열의 요소 수: shape(k, m, n) ==> k*m*n
... arr.size
30
>>> # 배열 타입 확인
... arr.dtype
dtype('float64')
>>> # 배열 타입명
... arr.dtype.name
'float64'
```


5. 배열 상태 검사(Inspecting)

…

- NumPy 배열 객체는 다음과 같은 방식으로 속성을 확인

```
>>> # 배열 요소를 int로 변환
... # 요소의 실제 값이 변환되는 것이 아님
... # View의 출력 타입과 연산을 변환하는 것
... arr.astype(np.int)
array([[[0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0]]])
```


5. 배열 상태 검사(Inspecting)

...

- NumPy 배열 객체는 다음과 같은 방식으로 속성을 확인

```
>>> # np.float으로 타입을 다시 변환하면 np.int 변환 이전 값으로 모든 원소 값이 복원됨
... arr.astype(np.float)
array([[[ 0.51854214,  0.06016704,  0.74072382],
        [ 0.31779295,  0.02336782,  0.63928198]],

       [[ 0.20385555,  0.09472152,  0.34458304],
        [ 0.57245853,  0.88902976,  0.14194716]],

       [[ 0.97143972,  0.68756585,  0.67412703],
        [ 0.85203383,  0.36536352,  0.42586769]],

       [[ 0.95212365,  0.77137236,  0.97339845],
        [ 0.88858888,  0.32378506,  0.20370252]],

       [[ 0.9360477 ,  0.6931599 ,  0.53844117],
        [ 0.95343717,  0.67163876,  0.67260179]]])
>>>
```


6. 도움말

…

- NumPy의 모든 API는 np.info 함수를 이용하여 도움말을 확인

```
>>> np.info(np.ndarray.dtype)
Data-type of the array's elements.

Parameters
----------
None

Returns
-------
d : numpy dtype object

See Also
--------
numpy.dtype

Examples
--------
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
>>>
```


7. 배열 연산

7.1 배열 일반 연산

- NumPy는 기본 연산자를 포함 모든 산술 연산 함수는 np 모듈에 포함되어 있다.

산술 연산(Arithmetic Operations)
NumPy는 기본 연산자를 연산자 재정의하여 배열(행렬) 연산에 대한 직관적으로 표현을 강화
모든 산술 연산 함수는 np 모듈에 포함

```
>>> # arange로 1부터 10 미만의 범위에서 1씩 증가하는 배열 생성
... # 배열의 shape을 (3, 3)으로 지정
...
>>> a = np.arange(1, 10).reshape(3, 3)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int32
Array's Data:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
>>>
```


7. 배열 연산

7.1 배열 일반 연산

- NumPy는 기본 연산자를 포함 모든 산술 연산 함수는 np 모듈에 포함되어 있다.

산술 연산(Arithmetic Operations)
NumPy는 기본 연산자를 연산자 재정의하여 배열(행렬) 연산에 대한 직관적으로 표현을 강화
모든 산술 연산 함수는 np 모듈에 포함

```
>>> # arange로 9부터 0까지 범위에서 1씩 감소하는 배열 생성
... # 배열의 shape을 (3, 3)으로 지정
...
>>> b = np.arange(9, 0, -1).reshape(3, 3)
>>> pprint(b)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int32
Array's Data:
 [[9 8 7]
 [6 5 4]
 [3 2 1]]
>>>
```


7. 배열 연산

7.1 배열 일반 연산

산술 연산(Arithmetic Operations)

배열 연산: 덧셈, +

```
>>> a+b
array([[10, 10, 10],
       [10, 10, 10],
       [10, 10, 10]])
>>> np.add(a, b)
array([[10, 10, 10],
       [10, 10, 10],
       [10, 10, 10]])
>>>
```

배열 연산: 뺄셈, -

```
>>> a - b
array([[-8, -6, -4],
       [-2,  0,  2],
       [ 4,  6,  8]])
>>> np.subtract(a, b)
array([[-8, -6, -4],
       [-2,  0,  2],
       [ 4,  6,  8]])
>>>
```


7. 배열 연산

7.1 배열 일반 연산

산술 연산(Arithmetic Operations)

배열 연산: 곱셈, *

```
>>> a*b
array([[ 9, 16, 21],
       [24, 25, 24],
       [21, 16,  9]])
>>> np.multiply(a, b)
array([[ 9, 16, 21],
       [24, 25, 24],
       [21, 16,  9]])
>>>
```

배열 연산: 나눗셈, /

```
>>> a/b
array([[ 0.11111111,  0.25      ,  0.42857143],
       [ 0.66666667,  1.        ,  1.5       ],
       [ 2.33333333,  4.        ,  9.        ]])
>>> np.divide(a, b)
array([[ 0.11111111,  0.25      ,  0.42857143],
       [ 0.66666667,  1.        ,  1.5       ],
       [ 2.33333333,  4.        ,  9.        ]])
>>>
```


7. 배열 연산

7.1 배열 일반 연산

산술 연산(Arithmetic Operations)

배열 연산: 지수

```
>>> np.exp(b)
array([[  8.10308393e+03,   2.98095799e+03,   1.09663316e+03],
       [  4.03428793e+02,   1.48413159e+02,   5.45981500e+01],
       [  2.00855369e+01,   7.38905610e+00,   2.71828183e+00]])
>>>
```

배열 연산: 제곱근

```
>>> np.sqrt(a)
array([[ 1.        ,  1.41421356,  1.73205081],
       [ 2.        ,  2.23606798,  2.44948974],
       [ 2.64575131,  2.82842712,  3.        ]])
>>>
```

배열 연산: sin

```
>>> np.sin(a)
array([[ 0.84147098,  0.90929743,  0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 ],
       [ 0.6569866 ,  0.98935825,  0.41211849]])
>>>
```


7. 배열 연산

7.1 배열 일반 연산

산술 연산(Arithmetic Operations)

배열 연산: cos

```
>>> np.cos(a)
array([[ 0.54030231, -0.41614684, -0.9899925 ],
       [-0.65364362,  0.28366219,  0.96017029],
       [ 0.75390225, -0.14550003, -0.91113026]])
>>>
```

배열 연산: tan

```
>>> np.tan(a)
array([[ 1.55740772, -2.18503986, -0.14254654],
       [ 1.15782128, -3.38051501, -0.29100619],
       [ 0.87144798, -6.79971146, -0.45231566]])
>>>
```

배연 연산: log

```
>>> np.log(a)
array([[ 0.        ,  0.69314718,  1.09861229],
       [ 1.38629436,  1.60943791,  1.79175947],
       [ 1.94591015,  2.07944154,  2.19722458]])
>>>
```


7. 배열 연산

7.1 배열 일반 연산

산술 연산(Arithmetic Operations)

배연 연산: dot product, 내적

```
>>> np.dot(a, b)
array([[ 30,  24,  18],
       [ 84,  69,  54],
       [138, 114,  90]])
>>>
```


7. 배열 연산

7.1 배열 일반 연산

비교 연산(Comparison)

배열의 요소별 비교 (Element-wise)

```
>>> a == b
array([[False, False, False],
       [False,  True, False],
       [False, False, False]], dtype=bool)
>>> a > b
array([[False, False, False],
       [False, False,  True],
       [ True,  True,  True]], dtype=bool)
>>>
```

배열 비교 (Array-wise)

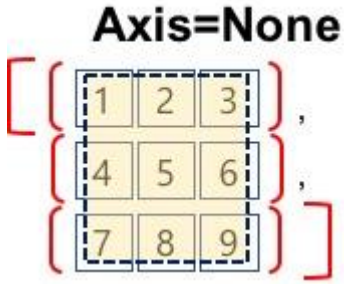
```
>>> np.array_equal(a, b)
False
>>>
```


7. 배열 연산

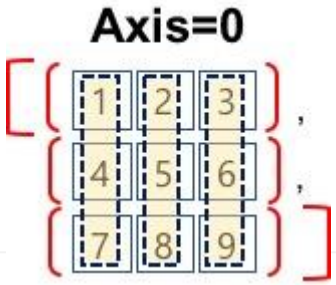
7.1 배열 일반 연산

집계 함수(Aggregate Functions)
NumPy의 모든 집계 함수는 집계 함수는 AXIS를 기준으로 계산
집계함수에 AXIS를 지정하지 않으면 axis=None입니다. axis=None, 0, 1은 다음과 같은 기준으로 생각

- **axis=None**
  aixs=None은 전체 행렬을 하나의 배열로 간주하고 집계 함수의 범위를 전체 행렬로 정의



- **axis=0aixs=0은 행을 기준으로 각 행의 동일 인덱스의 요소를 그룹으로 함**
- **각 그룹을 집계 함수의 범위로 정의**

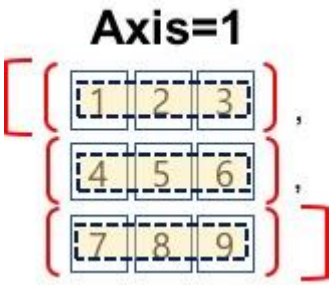


7. 배열 연산

7.1 배열 일반 연산

집계 함수(Aggregate Functions)
NumPy의 모든 집계 함수는 집계 함수는 AXIS를 기준으로 계산
집계함수에 AXIS를 지정하지 않으면 axis=None입니다. axis=None, 0, 1은 다음과 같은 기준으로 생각

- **axis=1aixs=1은 열을 기준으로 각 열의 요소를 그룹으로 함**
- **각 그룹을 집계 함수의 범위로 정의**




```
>>> # arange로 1부터 10미만의 범위에서 1씩 증가하는 배열 생성
... # 배열의 shape을 (3, 3)으로 지정
...
>>> a = np.arange(1, 10).reshape(3, 3)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int32
Array's Data:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
```


7. 배열 연산

7.1 배열 일반 연산

집계 함수(Aggregate Functions)

[ndarray 배열 객체].sum(), np.sum(): 합계
지정된 axis를 기준으로 요소의 합을 반환

```
>>> a.sum(), np.sum(a)
(45, 45)
>>> a.sum(axis=0), np.sum(a, axis=0)
(array([12, 15, 18]), array([12, 15, 18]))
>>> a.sum(axis=1), np.sum(a, axis=1)
(array([ 6, 15, 24]), array([ 6, 15, 24]))
>>>
```

[ndarray 배열 객체].min(), np.min(): 최소값
지정된 axis를 기준으로 요소의 최소값을 반환

```
>>> a.min(), np.min(a)
(1, 1)
>>> a.min(axis=0), np.min(a, axis=0)
(array([1, 2, 3]), array([1, 2, 3]))
>>> a.min(axis=1), np.min(a, axis=1)
(array([1, 4, 7]), array([1, 4, 7]))
>>>
```


7. 배열 연산

7.1 배열 일반 연산

집계 함수(Aggregate Functions)

[ndarray 배열 객체].max(), np.max(): 최대값
지정된 axis를 기준으로 요소의 최대값을 반환

```
>>> a.max(), np.max(a)
(9, 9)
>>> a.max(axis=0), np.max(a, axis=0)
(array([7, 8, 9]), array([7, 8, 9]))
>>> a.max(axis=1), np.max(a, axis=1)
(array([3, 6, 9]), array([3, 6, 9]))
>>>
```

[ndarray 배열 객체].cumssum(), np.cumsum(): 누적 합계
지정된 axis를 기준으로 각 요소의 누적 합의 결과를 반환

```
>>> a.cumsum(), np.cumsum(a)
(array([ 1,  3,  6, 10, 15, 21, 28, 36, 45], dtype=int32), array([ 1,  3,  6, 10, 15, 21, 28, 36, 45], dtype=int32))
>>> a.cumsum(axis=0), np.cumsum(a, axis=0)
(array([[ 1,  2,  3],
        [ 5,  7,  9],
        [12, 15, 18]], dtype=int32), array([[ 1,  2,  3],
        [ 5,  7,  9],
        [12, 15, 18]], dtype=int32))
>>> a.cumsum(axis=1), np.cumsum(a, axis=1)
(array([[ 1,  3,  6],
        [ 4,  9, 15],
        [ 7, 15, 24]], dtype=int32), array([[ 1,  3,  6],
        [ 4,  9, 15],
        [ 7, 15, 24]], dtype=int32))
>>>
```

7. 배열 연산

7.1 배열 일반 연산

집계 함수(Aggregate Functions)

[ndarray 배열 객체].mean(), np.mean(): 평균
지정된 axis를 기준으로 요소의 평균을 반환

```
>>> a.mean(), np.mean(a)
(5.0, 5.0)
>>> a.mean(axis=0), np.mean(a, axis=0)
(array([ 4.,  5.,  6.]), array([ 4.,  5.,  6.]))
>>> a.mean(axis=1), np.mean(a, axis=1)
(array([ 2.,  5.,  8.]), array([ 2.,  5.,  8.]))
>>>
```

np.mean(): 중앙값
지정된 axis를 기준으로 요소의 중앙값을 반환

```
>>> np.median(a)
5.0
>>> np.mean(a, axis=0)
array([ 4.,  5.,  6.])
>>> np.mean(a, axis=1)
array([ 2.,  5.,  8.])
>>>
```


7. 배열 연산

7.1 배열 일반 연산

집계 함수(Aggregate Functions)

np.corrcoef(): (상관계수)Correlation coefficient

```
>>> np.corrcoef(a)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>>
```

[ndarray 배열 객체].std(), np.std(): 표준편차

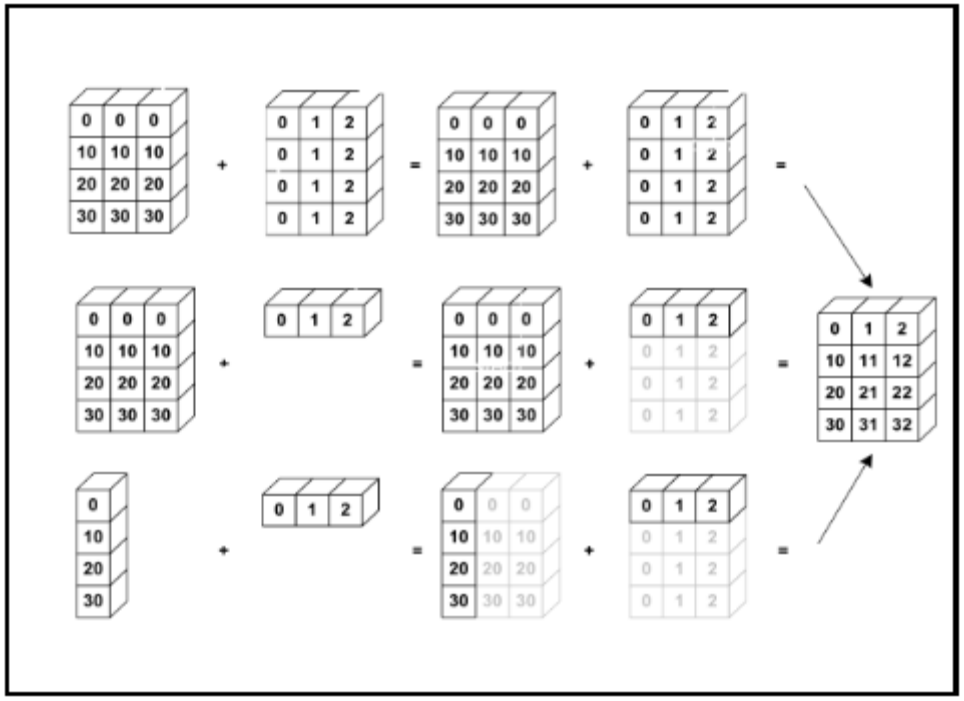
지정된 axis를 기준으로 요소의 표준 편차를 계산

```
>>> a.std(), np.std(a)
(2.5819888974716112, 2.5819888974716112)
>>> a.std(axis=0), np.std(a, axis=0)
(array([ 2.44948974,  2.44948974,  2.44948974]), array([ 2.44948974,  2.44948974,  2.44948974]))
>>> a.std(axis=1), np.std(a, axis=1)
(array([ 0.81649658,  0.81649658,  0.81649658]), array([ 0.81649658,  0.81649658,  0.81649658]))
>>>
```


7. 배열 연산

7.2 브로드캐스팅

- Shape이 같은 두 배열에 대한 이항 연산은 배열의 요소 별로 수행
- 두 배열 간의 Shape이 다를 경우 두 배열 간의 형상을 맞추는 <그림 >의 Broadcasting 과정을 거침



브로드캐스트 작동 원리
참조: **https://mathematica.stackexchange.com/questions/99171/how-to-implement-the-general-array-broadcasting-method-from-numpy**

7. 배열 연산

7.2 브로드캐스팅

```
>>> # 데모 배열 생성
...
>>> a = np.arange(1, 25).reshape(4, 6)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (4, 6), dimension: 2, dtype:int32
Array's Data:
 [[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
>>> b = np.arange(25, 49).reshape(4, 6)
>>> pprint(b)
type:<class 'numpy.ndarray'>
shape: (4, 6), dimension: 2, dtype:int32
Array's Data:
 [[25 26 27 28 29 30]
 [31 32 33 34 35 36]
 [37 38 39 40 41 42]
 [43 44 45 46 47 48]]
>>>
```


7. 배열 연산

7.2 브로드캐스팅

Shape이 다른 두 배열의 연산
Shape이 다른 두 배열 사이의 이항 연산에서 브로드케스팅 발생
두 배열을 같은 Shape으로 만든 후 연산을 수행

Case 1: 배열과 스칼라
배열과 스칼라 사이의 이항 연산 시 스칼라를 배열로 변형

```
>>> # 데모 배열 생성
...
>>> a = np.arange(1, 25).reshape(4, 6)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (4, 6), dimension: 2, dtype:int32
Array's Data:
 [[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
>>> a+100
array([[101, 102, 103, 104, 105, 106],
       [107, 108, 109, 110, 111, 112],
       [113, 114, 115, 116, 117, 118],
       [119, 120, 121, 122, 123, 124]])
>>>
```

a + 100은 다음과 같은 과정을 거쳐 처리

```
>>> # step 1: 스칼라 배열 변경
...
>>> new_arr = np.full_like(a, 100)
>>> pprint(new_arr)
type:<class 'numpy.ndarray'>
shape: (4, 6), dimension: 2, dtype:int32
Array's Data:
 [[100 100 100 100 100 100]
 [100 100 100 100 100 100]
 [100 100 100 100 100 100]
 [100 100 100 100 100 100]]
>>> # step 2: 배열 이항 연산
...
>>> a+new_arr
array([[101, 102, 103, 104, 105, 106],
       [107, 108, 109, 110, 111, 112],
       [113, 114, 115, 116, 117, 118],
       [119, 120, 121, 122, 123, 124]])
>>>
```

7. 배열 연산

7.2 브로드캐스팅

Shape이 다른 두 배열의 연산

Shape이 다른 두 배열 사이의 이항 연산에서 브로드케스팅 발생
두 배열을 같은 Shape으로 만든 후 연산을 수행

Case 2: Shaep이 다른 배열들의 연산

```
>>> # 데모 배열 생성
...
>>> a = np.arange(5).reshape((1, 5))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (1, 5), dimension: 2, dtype:int32
Array's Data:
 [[0 1 2 3 4]]
>>> b = np.arange(5).reshape((5, 1))
>>> pprint(b)
type:<class 'numpy.ndarray'>
shape: (5, 1), dimension: 2, dtype:int32
Array's Data:
 [[0]
 [1]
 [2]
 [3]
 [4]]
>>> a+b
array([[0, 1, 2, 3, 4],
       [1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7],
       [4, 5, 6, 7, 8]])
>>>
```


8. 배열 복사

...

- ndarray 배열 객체에 대한 slice, subset, indexing이 반환하는 배열은 새로운 객체가 아닌 기존 배열의 뷰
- 반환한 배열의 값을 변경하면 원본 배열에 반영
- 기본 배열로부터 새로운 배열을 생성하기 위해서는 copy 함수로 명시적으로 사용
- copy 함수로 복사된 원본 배열과 사본 배열은 완전히 다른 별도의 객체

[ndarray 배열 객체].copy(), np.copy()

```
>>> #데모용 배열
...
>>> a = np.random.randint(0, 9, (3, 3))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int32
Array's Data:
 [[8 1 7]
 [7 6 5]
 [5 3 0]]
```


8. 배열 복사

...

[ndarray 배열 객체].copy(), np.copy()

...

```
>>> copied_a1 =np.copy(a)
>>> # 복사된 배열의 요소 업데이트
... copied_a1[:, 0]=0 #배열의 전체 row의 첫번째 요소 0으로 업데이트
>>> pprint(copied_a1)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int32
Array's Data:
 [[0 1 7]
 [0 6 5]
 [0 3 0]]
>>> # 복사본 배열 변경이 원보에 영향을 미치지 않음
...
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int32
Array's Data:
 [[8 1 7]
 [7 6 5]
 [5 3 0]]
```


8. 배열 복사

...

[ndarray 배열 객체].copy(), np.copy()

...

```
>>> #np.copy()를 이용한 복사
...
>>> copied_a2 = np.copy(a)
>>> pprint(copied_a2)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int32
Array's Data:
 [[8 1 7]
 [7 6 5]
 [5 3 0]]
>>>
```


9. 배열 정렬

…

- ndarray 객체는 axis를 기준으로 요소 정렬하는 sort 함수를 제공

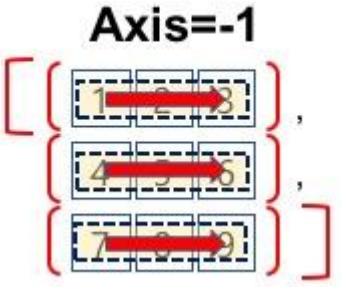
```
>>> #배열 생성
...
>>> unsorted_arr = np.random.random((3, 3))
>>> pprint(unsorted_arr)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:float64
Array's Data:
 [[ 0.06225557   0.52386238   0.63103379]
 [ 0.51632189   0.95317424   0.68346181]
 [ 0.47117225   0.23284275   0.01425411]]
>>> #데모를 위한 배열 복사
...
>>> unsorted_arr1 = unsorted_arr.copy()
>>> unsorted_arr2 = unsorted_arr.copy()
>>> unsorted_arr3 = unsorted_arr.copy()
>>>
```


9. 배열 정렬

…

- ndarray 객체는 axis를 기준으로 요소 정렬하는 sort 함수를 제공

•[ndarray 객체].sort() axis의 기본 값은 -1
•axis=-1은 현재 배열의 마지막 axis를 의미
•현재 unsorted_arr의 마지막 axis는 1
•[ndarray 객체].sort()와 [ndarray 객체].sort(axis=1)의 결과는 동일



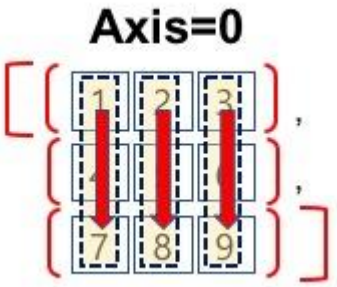

```
>>> #배열 정렬
…
>>> unsorted_arr1.sort()
>>> pprint(unsorted_arr1)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:float64
Array's Data:
 [[ 0.06225557  0.52386238  0.63103379]
 [ 0.51632189  0.68346181  0.95317424]
 [ 0.01425411  0.23284275  0.47117225]]
>>>
```


9. 배열 정렬

...

- ndarray 객체는 axis를 기준으로 요소 정렬하는 sort 함수를 제공

[axis=0의 정렬 방향은 다음 그림과 같다.



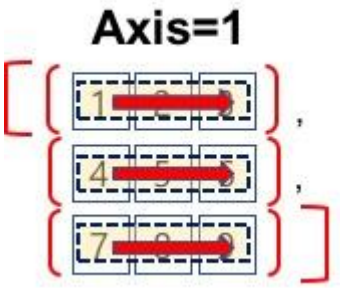

```
>>> #배열 정렬, axis=0
...
>>> unsorted_arr2.sort(axis=0)
>>> pprint(unsorted_arr2)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:float64
Array's Data:
 [[ 0.06225557   0.23284275   0.01425411]
 [ 0.47117225   0.52386238   0.63103379]
 [ 0.51632189   0.95317424   0.68346181]]
>>>
```


9. 배열 정렬

...

- ndarray 객체는 axis를 기준으로 요소 정렬하는 sort 함수를 제공

axis=1의 정렬 방향은 다음 그림과 같다.




```
>>> #배열 정렬, axis=1
...
>>> unsorted_arr3.sort(axis=1)
>>> pprint(unsorted_arr3)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:float64
Array's Data:
 [[ 0.06225557   0.52386238   0.63103379]
 [ 0.51632189   0.68346181   0.95317424]
 [ 0.01425411   0.23284275   0.47117225]]
>>>
```


10. 서브셋, 슬라이싱, 인덱싱

1 요소 선택

- 배열의 각 요소는 axis 인덱스 배열로 참조할 수 있음
- 1차원 배열은 1개 인덱스, 2차원 배열은 2개 인덱스, 3차원 인덱스는 3개 인덱스로 요소를 참조
- 인덱스로 참조한 요소는 값 참조, 값 수정이 모두 가능

```
>>> # 데모 배열 생성
...
>>> a0 = np.arange(24) # 1차원 배열
>>> pprint(a0)
type:<class 'numpy.ndarray'>
shape: (24,), dimension: 1, dtype:int32
Array's Data:
 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
>>> a1 = np.arange(24).reshape((4, 6)) #2차원 배열
>>> pprint(a1)
type:<class 'numpy.ndarray'>
shape: (4, 6), dimension: 2, dtype:int32
Array's Data:
 [[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
```

10. 서브셋, 슬라이싱, 인덱싱

1 요소 선택

```
>>> a2 = np.arange(24).reshape((2, 4, 3)) # 3차원 배열
>>> pprint(a2)
type:<class 'numpy.ndarray'>
shape: (2, 4, 3), dimension: 3, dtype:int32
Array's Data:
 [[[ 0  1  2]
  [ 3  4  5]
  [ 6  7  8]
  [ 9 10 11]]

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


10. 서브셋, 슬라이싱, 인덱싱

1 요소 선택

1 차원 배열 요소 참조 및 변경

```
>>> a0[5] # 5번 인덱스 요소 참조
5
>>> # 5번 인덱스 요소 업데이트
...
>>> a0[5] = 1000000
>>> pprint(a0)
type:<class 'numpy.ndarray'>
shape: (24,), dimension: 1, dtype:int32
Array's Data:
 [      0       1       2       3       4 1000000       6       7       8
        9      10      11      12      13      14      15      16      17
       18      19      20      21      22      23]
>>>
```


10. 서브셋, 슬라이싱, 인덱싱

1 요소 선택

2 차원 배열 요소 참조 및 변경

```
>>> pprint(a1)
type:<class 'numpy.ndarray'>
shape: (4, 6), dimension: 2, dtype:int32
Array's Data:
 [[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
>>> # 1행 두번째 컬럼 요소 참조
...
>>> a1[0, 1]
1
>>> # 1행 두번째 컬럼 요소 업데이트
...
>>> a1[0, 1]=10000
>>> pprint(a1)
type:<class 'numpy.ndarray'>
shape: (4, 6), dimension: 2, dtype:int32
Array's Data:
 [[    0 10000     2     3     4     5]
 [    6     7     8     9    10    11]
 [   12    13    14    15    16    17]
 [   18    19    20    21    22    23]]
>>>
```

10. 서브셋, 슬라이싱, 인덱싱

1 요소 선택

3 차원 배열 요소 참조 및 변경

```
>>> pprint(a2)
type:<class 'numpy.ndarray'>
shape: (2, 4, 3), dimension: 3, dtype:int32
Array's Data:
 [[[ 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 번째 행, 첫번째 컬럼, 두번째 요소 참조
...
>>> a2[1, 0, 1]
13
>>> a2[1, 0, 1]=10000
>>> pprint(a2)
type:<class 'numpy.ndarray'>
shape: (2, 4, 3), dimension: 3, dtype:int32
Array's Data:
 [[[    0     1     2]
  [    3     4     5]
  [    6     7     8]
  [    9    10    11]]

 [[   12 10000    14]
  [   15    16    17]
  [   18    19    20]
  [   21    22    23]]]
>>>
```

10. 서브셋, 슬라이싱, 인덱싱

2 슬라이싱(Slicing)

- 여러개의 배열 요소를 참조할 때 슬라이싱을 사용
- 슬라이싱은 axis 별로 범위를 지정하여 실행
- 범위는 fromindex:toindexfromindex:toindex 형태로 지정
- from_index는 범위의 시작 인덱스이며, to_index는 범위의 종료 인덱스
- 요소 범위를 지정할 때 to_index는 결과에 포함되지 않음.
- Fromindex:toindexfromindex:toindex의 범위 지정에서 from_index는 생략 가능
- 생략할 경우 0을 지정한 것으로 간주
- to_index 역시 생략 가능
- 이 경우 마지막 인덱스로 설정
- 따라서 "::" 형태로 지정된 범위는 전체 범위를 의미
- from_index와 to_index에 음수를 지정하면 이것은 반대 방향을 의미
- 예를 들어서 -1은 마지막 인덱스를 의미

10. 서브셋, 슬라이싱, 인덱싱

2 슬라이싱(Slicing)

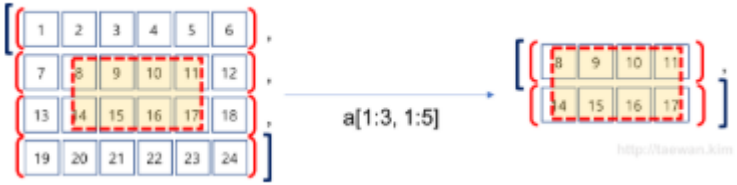
- 슬라이싱은 원본 배열의 뷰
- 따라서 슬라이싱 결과의 요소를 업데이트하면 원본에 반영

```
>>> # 데모 배열 생성
...
>>> a1 = np.arange(1, 25).reshape((4, 6))
>>> #2차원 배열
...
>>> pprint(a1)
type:<class 'numpy.ndarray'>
shape: (4, 6), dimension: 2, dtype:int32
Array's Data:
 [[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
>>>
```

10. 서브셋, 슬라이싱, 인덱싱

2 슬라이싱(Slicing)

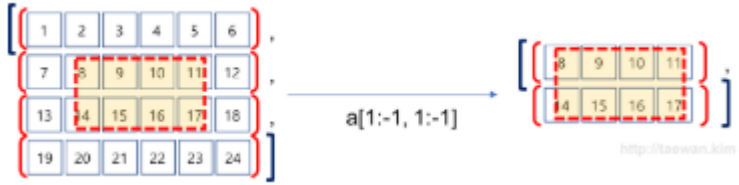
가운데 요소 가져오기




```
>>> a1[1:3, 1:5]
array([[ 8,  9, 10, 11],
       [14, 15, 16, 17]])
>>>
```

음수 인덱스를 이용한 범위 설정음수 인덱스는 지정한 axis의 마지막 요소로 부터 반대 방향의 인덱스
-1은 마지막 요소의 인덱스를 의미
다음 슬라이싱은 위 슬라이싱과 동일한 결과를 만듦

```
>>> a1[1:-1, 1:-1]
array([[ 8,  9, 10, 11],
       [14, 15, 16, 17]])
>>>
```



10. 서브셋, 슬라이싱, 인덱싱

2 슬라이싱(Slicing)

슬라이싱 업데이트

```
>>> # 데모 대상 배열 조회
...
>>> pprint(a1)
type:<class 'numpy.ndarray'>
shape: (4, 6), dimension: 2, dtype:int32
Array's Data:
 [[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
>>> # 슬라이싱 배열
...
>>> slide_arr = a1[1:3, 1:5]
>>> pprint(slide_arr)
type:<class 'numpy.ndarray'>
shape: (2, 4), dimension: 2, dtype:int32
Array's Data:
 [[ 8  9 10 11]
 [14 15 16 17]]
>>>
```


10. 서브셋, 슬라이싱, 인덱싱

2 슬라이싱(Slicing)

슬라이싱 업데이트

```
>>> # 슬라이싱 결과 배열에 슬라이싱을 적용하여 4개 요소 참조
...
>>> slide_arr[:, 1:3]
array([[ 9, 10],
       [15, 16]])
>>> pprint(slide_arr)
type:<class 'numpy.ndarray'>
shape: (2, 4), dimension: 2, dtype:int32
Array's Data:
 [[ 8  9 10 11]
 [14 15 16 17]]
>>> # 슬라이싱을 적용하여 참조한 4개 요소 업데이트 및 슬라이싱 배열 조회
...
>>> slide_arr[:, 1:3]=99999
>>> pprint(slide_arr)
type:<class 'numpy.ndarray'>
shape: (2, 4), dimension: 2, dtype:int32
Array's Data:
 [[    8 99999 99999    11]
 [   14 99999 99999    17]]
```


10. 서브셋, 슬라이싱, 인덱싱

2 슬라이싱(Slicing)

슬라이싱 업데이트

```
>>> # 원본 배열에 반영된 결과 확인
...
>>> pprint(a1)
type:<class 'numpy.ndarray'>
shape: (4, 6), dimension: 2, dtype:int32
Array's Data:
 [[    1     2     3     4     5     6]
 [    7     8 99999 99999    11    12]
 [   13    14 99999 99999    17    18]
 [   19    20    21    22    23    24]]
>>>
```


10. 서브셋, 슬라이싱, 인덱싱

3 블린 인덱싱(Boolean Indexing)

- NumPy는 불린 인덱싱은 배열 각 요소의 선택 여부를 True, False 지정하는 방식
- 해당 인덱스의 True만을 조회

```
>>> # 데모 배열 생성
...
>>> a1 = np.arange(1, 25).reshape((4, 6))
>>>
>>> #2차원 배열
...
>>> pprint(a1)
type:<class 'numpy.ndarray'>
shape: (4, 6), dimension: 2, dtype:int32
Array's Data:
 [[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
>>>
```


10. 서브셋, 슬라이싱, 인덱싱

3 블린 인덱싱(Boolean Indexing)

a1 배열에서 요소의 값이 짝수인 요소들의 총 합은?

```
>>> # 짝수인 요소 확인
... # numpy broadcasting을 이용하여 짝수인 배열 요소 확인
...
>>> even_arr = a1%2==0
>>> pprint(even_arr)
type:<class 'numpy.ndarray'>
shape: (4, 6), dimension: 2, dtype:bool
Array's Data:
 [[False  True False  True False  True]
 [False  True False  True False  True]
 [False  True False  True False  True]
 [False  True False  True False  True]]
>>> # a1[a1%2==0] 동일한 의미입니다.
...
>>> a1[even_arr]
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24])
>>> np.sum(a1)
300
>>>
```


10. 서브셋, 슬라이싱, 인덱싱

3 블린 인덱싱(Boolean Indexing)

Boolean Indexing의 응용
2014년 시애클 강수량 데이터: ./seattle2014.csv
2014년 시애틀 1월 평균 강수량은?

```
C:₩Users₩Teacher₩DEV>copy con seattle2014.csv
STATION,STATION_NAME,DATE,PRCP,SNWD,SNOW,TMAX,TMIN,AWND,WDF2,WDF5,WSF2,WSF5,WT01,WT05,WT02,WT03
GHCND:USW00024233,SEATTLE TACOMA INTERNATIONAL AIRPORT WA US,20140101,0,0,0,72,33,12,340,310,36,40,-9999,-
9999,-9999,-9999
GHCND:USW00024233,SEATTLE TACOMA INTERNATIONAL AIRPORT WA US,20140102,41,0,0,106,61,32,190,200,94,116,-
9999,-9999,-9999,-9999
        1개 파일이 복사되었습니다.

C:₩Users₩Teacher₩DEV>dir/w/p
 C 드라이브의 볼륨에는 이름이 없습니다.
 볼륨 일련 번호: 20D1-A772

 C:₩Users₩Teacher₩DEV 디렉터리

[.]             [..]            height.csv      my_array1.npy   my_array2.npz   saved.csv
seattle2014.csv  simple.csv
               6개 파일               1,233 바이트
               2개 디렉터리  34,785,116,160 바이트 남음

C:₩Users₩Teacher₩DEV>
```

10. 서브셋, 슬라이싱, 인덱싱

3 블린 인덱싱(Boolean Indexing)

Boolean Indexing의 응용

```
>>> # 데이터 로딩
...
>>> import pandas as pd
>>> rains_in_seattle = pd.read_csv("./seattle2014.csv")
>>> rains_arr = rains_in_seattle['PRCP'].values
>>> print("Data Size:", len(rains_arr))
Data Size: 2
>>> # 날짜 배열
...
>>> days_arr = np.arange(0, 365)
>>> # 1월의 날수 boolean index 생성
...
>>> condition_jan = days_arr < 31
>>> # 40일 조회
...
>>> condition_jan[:40]
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True, False, False, False, False, False,
       False, False, False, False], dtype=bool)
>>>
```


10. 서브셋, 슬라이싱, 인덱싱

3 블린 인덱싱(Boolean Indexing)

Boolean Indexing의 응용

```
>>> # 데이터 로딩
...
>>> import pandas as pd
>>> rains_in_seattle = pd.read_csv("./seattle2014.csv")
>>> rains_arr = rains_in_seattle['PRCP'].values
>>> print("Data Size:", len(rains_arr))
Data Size: 2
>>> # 날짜 배열
...
>>> days_arr = np.arange(0, 365)
>>> # 1월의 날수 boolean index 생성
...
>>> condition_jan = days_arr < 31
>>> # 40일 조회
...
>>> condition_jan[:40]
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True, False, False, False, False, False,
       False, False, False, False], dtype=bool)
>>>
```


10. 서브셋, 슬라이싱, 인덱싱

4 팬시 인덱싱(Fancy Indexing)

배열에 인덱스 배열을 전달하여 요소를 참조하는 방법

```
>>> arr = np.arange(1, 25).reshape((4, 6))
>>> pprint(arr)
type:<class 'numpy.ndarray'>
shape: (4, 6), dimension: 2, dtype:int32
Array's Data:
 [[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
>>>
```

Facncy Case 1

```
>>> [arr[0,0], arr[1, 1], arr[2, 2], arr[3, 3]]
[1, 8, 15, 22]
>>> # 두 배열을 전달==> (0, 0), (1,1), (2,2), (3, 3)
...
>>> arr[[0, 1, 2, 3], [0, 1, 2, 3]]
array([ 1,  8, 15, 22])
>>>
```


10. 서브셋, 슬라이싱, 인덱싱

4 팬시 인덱싱(Fancy Indexing)

Facncy Case 2

```
>>> # 전체 행에 대해서, 1, 2번 컬럼 참조
...
>>> arr[:, [1, 2]]
array([[ 2,  3],
       [ 8,  9],
       [14, 15],
       [20, 21]])
>>>
```


11. 배열 변환

1 전치(Transpose)

- 배열을 변환하는 방접으로는 전치, 배열 shape 변환, 배열 요소 추가, 배열 결합, 분리 등이 있다.

Tranpose는 행렬의 인덱스가 바뀌는 변환

$$[1 \quad 2]^T = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

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

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

다음은 NumPy에서 행렬을 전치하기 위하여 [numpy.ndarray 객체].T 속성을 사용
이 속성은 전치된 행렬을 반환

11. 배열 변환

1 전치(Transpose)

```
>>> # 행렬 생성
...
>>> a = np.random.randint(1, 10, (2, 3))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:int32
Array's Data:
 [[6 8 9]
 [3 5 8]]
>>> #행렬의 전치
...
>>> pprint(a.T)
type:<class 'numpy.ndarray'>
shape: (3, 2), dimension: 2, dtype:int32
Array's Data:
 [[6 3]
 [8 5]
 [9 8]]
>>>
```


11. 배열 변환

2 배열 형태 변경

- [numpy.ndarray 객체]는 배열의 Shape을 변경하는 ravel 메서드와 reshape 메서드를 제공
- ravel은 배열의 shape을 1차원 배열로 만드는 메서드
- reshape은 데이터 변경없이 지정된 shape으로 변환하는 메서드

[numpy.ndarray 객체].ravel()

- 배열을 1차원 배열로 반환하는 메서드
- numpy.ndarray 배열 객체의 View를 반환
- ravel 메서드가 반환하는 배열은 원본 배열이 뷰
- 따라서 ravel 메서드가 반환하는 배열의 요소를 수정하면 원본 배열 요소에도 반영

```
>>> # 데모 배열 생성
...
>>> a = np.random.randint(1, 10, (2, 3))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:int32
Array's Data:
 [[3 6 5]
 [5 5 9]]
>>>
```


11. 배열 변환

2 배열 형태 변경

revel이 반환하는 배열은 a 행렬의 view
반환 행렬의 데이터를 변경하면 a 행렬도 변경.

```
>>> a.ravel()
array([3, 6, 5, 5, 5, 9])
>>>
>>> b = a.ravel()
>>> pprint(b)
type:<class 'numpy.ndarray'>
shape: (6,), dimension: 1, dtype:int32
Array's Data:
 [3 6 5 5 5 9]
>>> b[0]=99
>>> pprint(b)
type:<class 'numpy.ndarray'>
shape: (6,), dimension: 1, dtype:int32
Array's Data:
 [99  6  5  5  5  9]
>>> # b 배열 변경이 a 행렬에 반영되어 있습니다.
...
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:int32
Array's Data:
 [[99  6  5]
 [ 5  5  9]]
>>>
```


11. 배열 변환

2 배열 형태 변경

[numpy.ndarray 객체].reshape()
[numpy.ndarray 객체]의 shape을 변경
실제 데이터를 변경하는 것은 아닌, 배열 객체의 shape 정보만을 수정

```
>>> # 대상 행렬 속성 확인
...
>>> a = np.random.randint(1, 10, (2, 3))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:int32
Array's Data:
 [[3 3 3]
 [4 7 3]]
>>> result = a.reshape((3, 2, 1))
>>> pprint(result)
type:<class 'numpy.ndarray'>
shape: (3, 2, 1), dimension: 3, dtype:int32
Array's Data:
 [[[3]
  [3]]

 [[3]
  [4]]

 [[7]
  [3]]]
>>>
```


11. 배열 변환

3 배열 요소 추가 삭제

- 배열의 요소를 변경, 추가, 삽입 및 삭제하는 resize, append, insert, delete 함수를 제공

resize()

- np.resize(a, new_shape)
- np.ndarray.resize(new_shape, refcheck=True)
- 배열의 shape과 크기를 변경

np.resize와 np.reshape 함수는 배열의 shape을 변경한다는 부분에서 유사
차이점은 reshape 함수는 배열 요소 수를 변경하지 않는다.
Reshape 전후 배열의 요소 수는 같다.
반면에 resize는 shape을 변경하는 과정에서 배열 요소 수를 줄이거나 늘린다.

일반적인 resize 사용 방법

```
>>> #배열 생성
...
>>> a = np.random.randint(1, 10, (2, 6))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 6), dimension: 2, dtype:int32
Array's Data:
 [[5 7 5 4 1 9]
 [8 4 4 4 3 9]]
>>>
```

11. 배열 변환

3 배열 요소 추가 삭제

```
>>> # shape 변경 - 요소 수 변경 없음
...
>>> a.resize((6, 2))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (6, 2), dimension: 2, dtype:int32
Array's Data:
 [[5 7]
 [5 4]
 [1 9]
 [8 4]
 [4 4]
 [3 9]]
>>>
```


11. 배열 변환

3 배열 요소 추가 삭제

요소 수가 늘어난 변경

```
>>> #배열 생성
...
>>> a = np.random.randint(1, 10, (2, 6))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 6), dimension: 2, dtype:int32
Array's Data:
 [[5 6 6 1 5 2]
 [6 5 7 4 5 8]]
>>> # 요소숙 12개에서 20개로 늘어남
... # 늘어난 요소는 0으로 채워짐
...
>>> a.resize((2, 10))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 10), dimension: 2, dtype:int32
Array's Data:
 [[5 6 6 1 5 2 6 5 7 4]
 [5 8 0 0 0 0 0 0 0 0]]
>>>
```


11. 배열 변환

3 배열 요소 추가 삭제

요소 수가 즐어든 변경

```
>>> #배열 생성
...
>>> a = np.random.randint(1, 10, (2, 6))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 6), dimension: 2, dtype:int32
Array's Data:
 [[2 2 2 3 8 8]
 [2 5 3 4 6 8]]
>>> # 요소숙 12개에서 9개로 줄임
... # 이전 데이터 삭제
...
>>> a.resize((3, 3))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int32
Array's Data:
 [[2 2 2]
 [3 8 8]
 [2 5 3]]
>>>
```


11. 배열 변환

3 배열 요소 추가 삭제

append()

- np.append(arr, values, axis=None)
- 배열의 끝에 값을 추가

np.append 함수는 arr의 끝에 values(배열)을 추가
axis로 배열이 추가되는 방향을 지정

```
>>> # 데모 배열 생성
...
>>> a = np.arange(1, 10).reshape(3, 3)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int32
Array's Data:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> b = np.arange(10, 19).reshape(3, 3)
>>> pprint(b)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int32
Array's Data:
 [[10 11 12]
 [13 14 15]
 [16 17 18]]
>>>
```


11. 배열 변환

3 배열 요소 추가 삭제

case 1: axis을 지정하지 않을 경우

axis를 지정하지 않으면 배열은 1차원 배열로 변형되어 결합

```
>>> # axis 지정 없이 추가
...
>>> result = np.append(a, b)
>>> pprint(result)
type:<class 'numpy.ndarray'>
shape: (18,), dimension: 1, dtype:int32
Array's Data:
 [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18]
>>> #원본 배열을 변경하는 것이 아니며 새로운 배열이 생성
...
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int32
Array's Data:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
>>>
```


11. 배열 변환

3 배열 요소 추가 삭제

case 2: axis=0 지정

```
>>> # axis = 0
... # axis 0 방향으로 b 배열 추가
...
>>> result = np.append(a, b, axis=0)
>>> pprint(result)
type:<class 'numpy.ndarray'>
shape: (6, 3), dimension: 2, dtype:int32
Array's Data:
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]
 [13 14 15]
 [16 17 18]]
>>>
```

- axis = 0 설정 시, shape[0]를 제외한 나머지 shape은 같아야 함.

11. 배열 변환

3 배열 요소 추가 삭제

shape[0]를 제외한 나머지 shape이 다를 경우 append는 오류를 발생.

```
>>> different_sahpe_arr = np.arange(10, 20).reshape(2, 5)
>>> pprint(different_sahpe_arr)
type:<class 'numpy.ndarray'>
shape: (2, 5), dimension: 2, dtype:int32
Array's Data:
 [[10 11 12 13 14]
 [15 16 17 18 19]]
>>> # 기준 축을 제외한 shape이 다른 배열의 append: 오류 발생
...
>>> np.append(a, different_sahpe_arr, axis=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:₩DEV₩Anaconda3₩lib₩site-packages₩numpy₩lib₩function_base.py", line 5003, in append
    return concatenate((arr, values), axis=axis)
ValueError: all the input array dimensions except for the concatenation axis must match exactly
>>>
```


11. 배열 변환

3 배열 요소 추가 삭제

case 3: axis=1 지정

- axis = 1 설정 시, shape[1]을 제외한 나머지 shape은 같아야 함
- shape[1]를 제외한 나머지 shape이 다를 경우 append는 오류를 발생

```
>>> # axis = 1
... # axis 1 방향으로 b 배열 추가
...
>>> result = np.append(a, b, axis=1)
>>> pprint(result)
type:<class 'numpy.ndarray'>
shape: (3, 6), dimension: 2, dtype:int32
Array's Data:
 [[ 1  2  3 10 11 12]
 [ 4  5  6 13 14 15]
 [ 7  8  9 16 17 18]]
>>>
```


11. 배열 변환

3 배열 요소 추가 삭제

axis = 1 설정 시, shape[1]를 제외한 나머지 shape은 같아야 함
shape[1]를 제외한 나머지 shape이 다를 경우 append는 오류를 발생

```
>>> # shape이 다른 배열 생성
...
>>> different_sahpe_arr = np.arange(10, 20).reshape(5, 2)
>>> pprint(different_sahpe_arr)
type:<class 'numpy.ndarray'>
shape: (5, 2), dimension: 2, dtype:int32
Array's Data:
 [[10 11]
 [12 13]
 [14 15]
 [16 17]
 [18 19]]
>>> # 기준 축을 제외한 shape이 다른 배열의 append: 오류 발생
...
>>> np.append(a, different_sahpe_arr, axis=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:₩DEV₩Anaconda3₩lib₩site-packages₩numpy₩lib₩function_base.py", line 5003, in append
    return concatenate((arr, values), axis=axis)
ValueError: all the input array dimensions except for the concatenation axis must match exactly
>>>
```

11. 배열 변환

3 배열 요소 추가 삭제

insert()

np.insert(arr, obj, values, axis=None)
axis를 지정하지 않으며 1차원 배열로 변환
추가할 방향을 axis로 지정

```
>>> # 데모 배열 생성
...
>>> a = np.arange(1, 10).reshape(3, 3)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int32
Array's Data:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> #  a 배열을 일차원 배열로 변환하고 1번 index에 99 추가
...
>>> np.insert(a, 1, 999)
array([  1, 999,   2,   3,   4,   5,   6,   7,   8,   9])
>>> # a 배열의 axis 0 방향 1번 인덱스에 추가
... # index가 1인 row에 999가 추가됨
...
```


11. 배열 변환

3 배열 요소 추가 삭제

insert()

```
>>> np.insert(a, 1, 999, axis=0)
array([[  1,   2,   3],
       [999, 999, 999],
       [  4,   5,   6],
       [  7,   8,   9]])
>>> # a 배열의 axis 1 방향 1번 인덱스에 추가
... # index가 1인 column에 999가 추가됨
...
>>> np.insert(a, 1, 999, axis=1)
array([[  1, 999,   2,   3],
       [  4, 999,   5,   6],
       [  7, 999,   8,   9]])
>>>
```


11. 배열 변환

3 배열 요소 추가 삭제

delete()

np.delete(arr, obj, axis=None)
axis를 지정하지 않으며 1차원 배열로 변환
삭제할 방향을 axis로 지정
delete 함수는 원본 배열을 변경하지 않으며 새로운 배열을 반환

```
>>> # 데모 배열 생성
...
>>> a = np.arange(1, 10).reshape(3, 3)
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int32
Array's Data:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> #  a 배열을 일차원 배열로 변환하고 1번 index 삭제
...
>>> np.delete(a, 1)
array([1, 3, 4, 5, 6, 7, 8, 9])
```


11. 배열 변환

3 배열 요소 추가 삭제

delete()

```
>>> # a 배열의 axis 0 방향 1번 인덱스인 행을 삭제한 배열을 생성하여 반환
...
>>> np.delete(a, 1, axis=0)
array([[1, 2, 3],
       [7, 8, 9]])
>>> # a 배열의 axis 1 방향 1번 인덱스인 열을 삭제한 배열을 생성하여 반환
...
>>> np.delete(a, 1, axis=1)
array([[1, 3],
       [4, 6],
       [7, 9]])
>>>
```


11. 배열 변환

4 배열 결합

- 배열과 배열을 결합하는 np.concatenate, np.vstack, np.hstack 함수를 제공

배열 결합
np.concatenate
concatenate((a1, a2, ...), axis=0)
a1, a2.....: 배열

```
>>> # 데모 배열
...
>>> a = np.arange(1, 7).reshape((2, 3))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:int32
Array's Data:
 [[1 2 3]
 [4 5 6]]
>>> b = np.arange(7, 13).reshape((2, 3))
>>> pprint(b)
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:int32
Array's Data:
 [[ 7  8  9]
 [10 11 12]]
>>>
```


11. 배열 변환

4 배열 결합

배열 결합

np.concatenate

```
>>> # axis=0 방향으로 두 배열 결합, axis 기본값=0
...
>>> result = np.concatenate((a, b))
>>> result
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
>>> # axis=0 방향으로 두 배열 결합, 결과 동일
...
>>> result = np.concatenate((a, b), axis=0)
>>> result
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
>>> # axis=1 방향으로 두 배열 결합, 결과 동일
...
>>> result = np.concatenate((a, b), axis=1)
>>> result
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
>>>
```


11. 배열 변환

4 배열 결합

수직 방향 배열 결합

np.vstack

np.vstack(tup)

tup: 튜플

튜플로 설정된 여러 배열을 수직 방향으로 연결 (axis=0 방향)

np.concatenate(tup, axis=0)와 동일

```
>>> # 데모 배열
...
>>> a = np.arange(1, 7).reshape((2, 3))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:int32
Array's Data:
 [[1 2 3]
 [4 5 6]]
>>> b = np.arange(7, 13).reshape((2, 3))
>>> pprint(b)
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:int32
Array's Data:
 [[ 7  8  9]
 [10 11 12]]
>>>
```

11. 배열 변환

4 배열 결합

수직 방향 배열 결합

np.vstack

```
>>> np.vstack((a, b))
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
>>> # 4개 배열을 튜플로 설정
...
>>> np.vstack((a, b, a, b))
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12],
       [ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
>>>
```


11. 배열 변환

4 배열 결합

수평 방향 배열 결합
np.hstack
np.hstack(tup)
tup: 튜플
튜플로 설정된 여러 배열을 수평 방향으로 연결 (axis=1 방향)
np.concatenate(tup, axis=1)와 동일

```
>>> # 데모 배열
...
>>> a = np.arange(1, 7).reshape((2, 3))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:int32
Array's Data:
 [[1 2 3]
 [4 5 6]]
>>> b = np.arange(7, 13).reshape((2, 3))
>>> pprint(b)
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:int32
Array's Data:
 [[ 7  8  9]
 [10 11 12]]
>>>
```

11. 배열 변환

4 배열 결합

수평 방향 배열 결합

np.hstack

```
>>> np.hstack((a, b))
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
>>> np.hstack((a, b, a, b))
array([[ 1,  2,  3,  7,  8,  9,  1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12,  4,  5,  6, 10, 11, 12]])
>>>
```

11. 배열 변환

5 배열 분리

- NumPy는 배열을 수직, 수평으로 분할하는 함수를 제공
- np.hsplit(): 지정한 배열을 수평(행) 방향으로 분할
- np.vsplit(): 지정한 배열을 수직(열) 방향으로 분할

배열 수평 분할

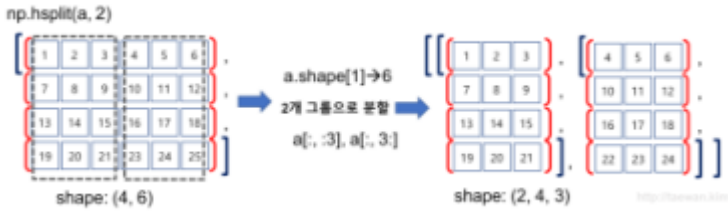
np.hsplit(ary, indices_or_sections)
배열을 수평 방향(컬럼 방향)으로 분할하는 함수

```
>>> # 분할 대상 배열 생성
...
>>> a = np.arange(1, 25).reshape((4, 6))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (4, 6), dimension: 2, dtype:int32
Array's Data:
 [[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
>>>
```


11. 배열 변환

5 배열 분리

수평 방향으로 배열을 두 그룹으로 분할

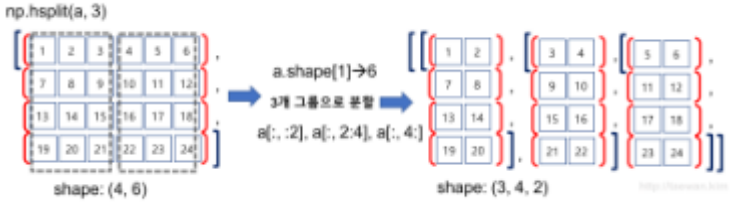



```
>>> # 수평으로 두 그룹으로 분할하는 함수
...
>>> result = np.hsplit(a, 2)
>>> result
[array([[ 1,  2,  3],
        [ 7,  8,  9],
        [13, 14, 15],
        [19, 20, 21]]), array([[ 4,  5,  6],
        [10, 11, 12],
        [16, 17, 18],
        [22, 23, 24]])]
>>>
```

11. 배열 변환

5 배열 분리

수평 방향으로 배열을 세 그룹으로 분할



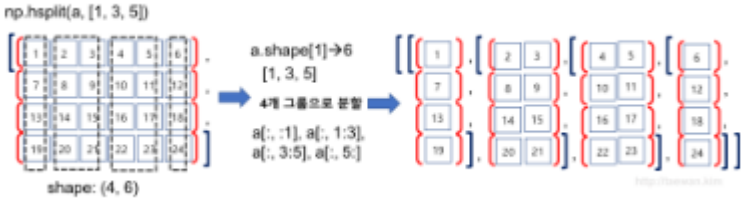

```
>>> # 수평으로 두 그룹으로 분할하는 함수
...
>>> result = np.hsplit(a, 3)
>>> result
[array([[ 1,  2],
        [ 7,  8],
        [13, 14],
        [19, 20]]), array([[ 3,  4],
        [ 9, 10],
        [15, 16],
        [21, 22]]), array([[ 5,  6],
        [11, 12],
        [17, 18],
        [23, 24]])]
>>>
```


11. 배열 변환

5 배열 분리

수평 방향으로 여러 구간으로 구분

np.hsplit의 두 번째 파라미터에 구간 설정 배열을 전달하여 여러 배열로 구분




```
>>> np.hsplit(a, [1, 3 ,5])
[array([[ 1],
        [ 7],
        [13],
        [19]]), array([[ 2,  3],
        [ 8,  9],
        [14, 15],
        [20, 21]]), array([[ 4,  5],
        [10, 11],
        [16, 17],
        [22, 23]]), array([[ 6],
        [12],
        [18],
        [24]])]
>>>
```

11. 배열 변환

5 배열 분리

배열 수직 분할

np.vsplit(ary, indices_or_sections)

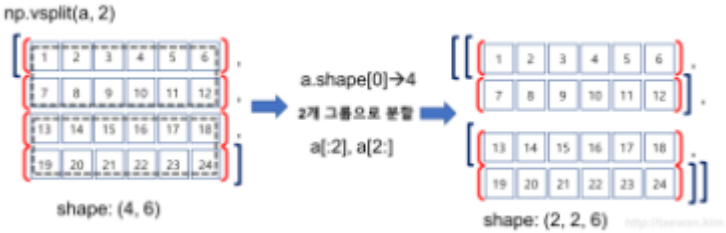
배열을 수직 방향(행 방향)으로 분할하는 함수

```
>>> # 분할 대상 배열 생성
...
>>> a = np.arange(1, 25).reshape((4, 6))
>>> pprint(a)
type:<class 'numpy.ndarray'>
shape: (4, 6), dimension: 2, dtype:int32
Array's Data:
 [[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
>>>
```


11. 배열 변환

5 배열 분리

수직 방향으로 배열을 두 개 그룹으로 분할

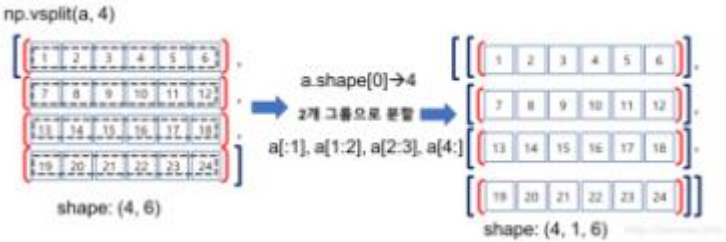



```
>>> result=np.vsplit(a, 2)
>>> result
[array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]]), array([[13, 14, 15, 16, 17, 18],
       [19, 20, 21, 22, 23, 24]])]
>>> np.array(result).shape
(2, 2, 6)
>>>
```


11. 배열 변환

5 배열 분리

수직 방향으로 배열을 4 개 그룹으로 분할




```
>>> result=np.vsplit(a, 4)
>>> result
[array([[1, 2, 3, 4, 5, 6]]), array([[ 7,  8,  9, 10, 11, 12]]), array([[13, 14, 15, 16, 17, 18]]), array([[19, 20, 21, 22,
23, 24]])]
>>> np.array(result).shape
(4, 1, 6)
>>>
```


11. 배열 변환

5 배열 분리

수직 방향으로 여러 구간으로 구분
np.hsplit의 두 번째 파라미터에 구간 설정 배열을 전달하여 여러 배열로 구분


```
>>> # row를 1, 2-3, 4번째 라인으로 구분
...
>>> np.vsplit(a, [1, 3])
[array([[1, 2, 3, 4, 5, 6]]), array([[ 7,  8,  9, 10, 11, 12],
       [13, 14, 15, 16, 17, 18]]), array([[19, 20, 21, 22, 23, 24]])]
>>>
```

12. Quiz-numpy

문제1. 아래의 a 배열에 모든 원소에 5를 더한 결과를 출력하시오

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

```
import numpy as np

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

print(a)
print(a+5)
```

문제2. 아래의 배열의 원소들의 평균값을 출력하시오

a = numpy.array([1,2,3,4,5,5,6,10])

```
import numpy as np

a = np.array([1,2,3,4,5,5,6,10])
print(np.mean(a))
```


12. Quiz-numpy

문제3. a 배열의 중앙값을 출력하시오

import numpy as np
a = np.array([1,2,3,4,5,5,6,10])

```
import numpy as np

a = np.array([1,2,3,4,5,5,6,10])
print(np.median(a))
```

문제4. a 배열의 최대값과 최소값을 출력하시오

import numpy as np
a = np.array([1,2,3,4,5,5,6,10])

```
import numpy as np

a = np.array([1,2,3,4,5,5,6,10])

print(np.max(a))
print(np.min(a))
```


12. Quiz-numpy

문제5. a 배열의 표준편차와 분산을 출력하시오

import numpy as np
a = np.array([1,2,3,4,5,5,6,10])

```
import numpy as np

a = np.array([1,2,3,4,5,5,6,10])

print(np.std(a))
print(np.var(a))
```

문제6. 아래의 행렬식을 numpy로 구현하시오

$$\begin{bmatrix} 1 & 3 & 7 \\ 1 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 5 \\ 7 & 5 & 0 \end{bmatrix}$$

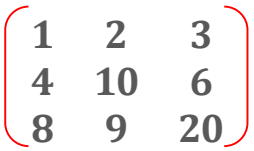
```
import numpy as np

a=np.array([[1,3,7],[1,0,0]])
b=np.array([[0,0,5],[7,5,0]])

print(a+b)
```

```
[[ 1  3 12]
 [ 8  5  0]]
```


12. Quiz-numpy

문제7. 아래의 numpy 배열을 생성하고 원소 중에 10만 출력해보시오

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 10 & 6 \\ 8 & 9 & 20 \end{pmatrix}$$



```
import numpy as np

a=np.array([[1,2,3],[4,10,6],[8,9,20]])

print(a[1][1])
```

문제8.아래의 행렬 연산을 파이썬으로 구현하시오

$$\begin{pmatrix} 1 & 2 \\ & \\ 3 & 4 \end{pmatrix} \text{X} \begin{pmatrix} 10 & 20 \end{pmatrix}$$

```
import numpy as np

a= np.array([[1,2],[3,4]])
b= np.array([10,20])

print(a*b)
```

```
[[10 40]
 [30 80]]
```

```
import numpy as np

a= np.array([[1,2],[3,4]])
b= np.array([10,20])

print(np.dot(a,b))
```

```
[ 50 110]
```


12. Quiz-numpy

문제9. 아래의 그림의 행렬 연산을 numpy로 구현하시오

```
[ 0]
[10]
[20]    +   [0 1 2]
[30]
```

```
import numpy as np

a=np.array([[0],[10],[20],[30]])
b=np.array([0,1,2])
print(a)
print(b)
print(a+b)
```

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


12. Quiz-numpy

문제10. 아래의 행렬 연산을 numpy와 numpy를 이용하지 않았을때 2가지 방법으로 구현하시오

$$\begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix} - \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

```
import numpy as np

a= np.array([[10,20],[30,40]])
b= np.array([[5,6],[7,8]])

print(a-b)
```

```
[[ 5 14]
 [23 32]]
```

```
a= [[10,20],[30,40]]
b= [[5,6],[7,8]]
res=[[0,0],[0,0]]
for i in range(len(a)):
    for j in range(len(a[0])):
        res[i][j]=a[i][j]-b[i][j]
print(res)
```

```
[[5, 14], [23, 32]]
```


12. Quiz-numpy

shape / reshape

```
import numpy as np

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

print(A.shape)
print(A.shape[0])
print(A.shape[1])
```

```
(2, 3)
2
3
```

```
import numpy as np

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

print(A)

B = A.reshape((3, 2))

print(B)
```

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


12. Quiz-numpy

concatenate
axis = 0 : Y축 (세로 방향)으로 설정한다.
axis = 1 : X축 (가로 방향)으로 설정한다.

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
C_Y = np.concatenate((A, B), axis = 0)
print(C_Y)
C_X = np.concatenate((A, B), axis = 1)
print(C_X)
```

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


Thank you