



# Python Programming for Beginners

## NumPy

2023학년도 2학기

Suk-Hwan Lee

**Artificial Intelligence**

*Creating the Future*

Dong-A University

Division of Computer Engineering &  
Artificial Intelligence

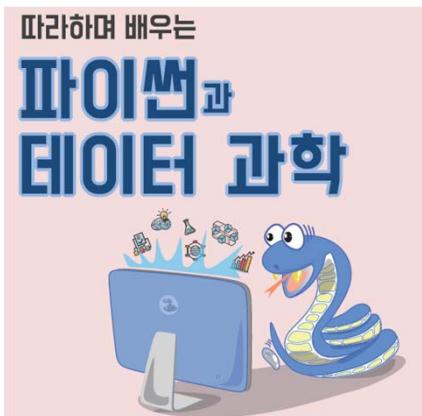
## References

### NumPy Tutorials

- <https://numpy.org/doc/stable/index.html>
- <https://docs.scipy.org/doc/numpy-1.17.0/user/quickstart.html#array-creation>
- <http://aikorea.org/cs231n/python-numpy-tutorial/>
- <https://towardsdatascience.com/the-ultimate-beginners-guide-to-numpy-f5a2f99aef54>
- [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)
- “따라하며 배우는 파이썬과 데이터 과학”, 생능출판사 강의자료

### ➤ NumPy

- 파이썬의 과학계산을 위한 가장 기본적인 라이브러리
- 행렬, 벡터 연산을 위한 사실상의 표준 라이브러리로 빠른 처리속도가 장점
- 다차원 배열과 행렬 객체가 포함



## NumPy 소개

- Wikipedia : <https://en.wikipedia.org/wiki/NumPy>

## NumPy

From Wikipedia, the free encyclopedia

NumPy (pronounced /nʌmpai/ (*NUM-py*) or sometimes /nʌmpi/<sup>[3][4]</sup> (*NUM-pee*)) is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.<sup>[5]</sup> The ancestor of NumPy, Numeric, was originally created by Jim Hugunin with contributions from several other developers. In 2005, Travis Oliphant created NumPy by incorporating features of the competing Numarray into Numeric, with extensive modifications. NumPy is open-source software and has many contributors.

Contents [hide]
1 History
2 Features
2.1 The ndarray data structure
2.2 Limitations
3 Examples
4 See also
5 References
6 Further reading
7 External links



### Linear algebra

```
>>> from numpy.random import rand
>>> from numpy.linalg import solve, inv
>>> a = np.array([[1, 2, 3], [3, 4, 6.7], [5, 9.0, 5]])
>>> a.transpose()
array([[ 1.,  3.,  5.],
       [ 2.,  4.,  9.],
       [ 3.,  6.7,  5.]])
>>> inv(a)
array([-2.27683616,  0.96045198,  0.07909605],
      [ 1.04519774, -0.56497175,  0.1299435 ],
      [ 0.39548023,  0.05649718, -0.11299435]])
>>> b = np.array([3, 2, 1])
>>> solve(a, b) # solve the equation ax = b
array([-4.83050847,  2.13559322,  1.18644068])
>>> c = rand(3, 3) * 20 # create a 3x3 random matrix of values within [0, 1] scaled by 20
>>> c
array([[ 3.98732789,  2.47702609,  4.71167924],
       [ 9.24410671,  5.5240412 ,  10.6468792 ],
       [10.38136661,  8.44968437,  15.17639591]])
>>> np.dot(a, c) # matrix multiplication
array([[ 53.61964114,  38.8741616 ,  71.53462537],
       [118.4935668 ,  86.14012835,  158.40440712],
       [155.04043289, 104.3499231 , 195.26228855]])
```

### Tensors

```
>>> M = np.zeros(shape=(2, 3, 5, 7, 11))
>>> T = np.transpose(M, (4, 2, 1, 3, 0))
>>> T.shape
(11, 5, 3, 7, 2)
```

### Incorporation with OpenCV

```
>>> import numpy as np
>>> import cv2
>>> r = np.reshape(np.arange(256*256)%256,(256,256)) # 256x256 pixel array with a horizontal
   gradient from 0 to 255 for the red color channel
>>> g = np.zeros_like(r) # array of same size and type as r but filled with 0s for the green color
   channel
>>> b = r.T # transposed r will give a vertical gradient for the blue color channel
>>> cv2.imwrite('gradients.png', np.dstack([b,g,r])) # OpenCV images are interpreted as BGR, the
   depth-stacked array will be written to an 8bit RGB PNG-file called 'gradients.png'
True
```

## NumPy 설치 및 특징

### Installing NumPy

To install NumPy, I strongly recommend using a scientific Python distribution. If you're looking for the full instructions for installing NumPy on your operating system, you can [find all of the details here](#).

If you already have Python, you can install NumPy with

```
conda install numpy
```

아나콘다, 미니콘다라는 패키지를 설치하면  
자동 설치가 됨

```
pip install numpy
```

구글 colab 환경에서는 설치가 필요없음

### How to import NumPy

Any time you want to use a package or library in your code, you first need to make it accessible.

In order to start using NumPy and all of the functions available in NumPy, you'll need to import it. This can be easily done with this import statement:

```
import numpy as np
```

(We shorten “numpy” to “np” in order to save time and also to keep code standardized so that anyone working with your code can easily understand and run it.)

### NumPy

- 다차원 행렬 자료구조인 ndarray를 핵심으로 선형대수 연산이 필요한 알고리듬에 사용
- ndarray는 Numpy의 핵심인 **다차원 행렬 자료구조 클래스** 입니다.
- 실제로 파이썬이 제공하는 List 자료형과 동일한 출력 형태를 갖습니다.

➤ What's the difference between a Python list and a NumPy array?

- NumPy gives you an enormous range of fast and efficient numerically-related options.
- While **a Python list can contain different data types within a single list, all of the elements in a NumPy array should be homogenous.**
- The mathematical operations that are meant to be performed on arrays wouldn't be possible if the arrays weren't homogenous.

➤ What is array?

- An array can be indexed by a tuple of nonnegative integers, by booleans, by another array, or by integers.
- The **rank** of the array is **the number of dimensions**. The **shape** of the array is **a tuple of integers giving the size of the array along each dimension**.

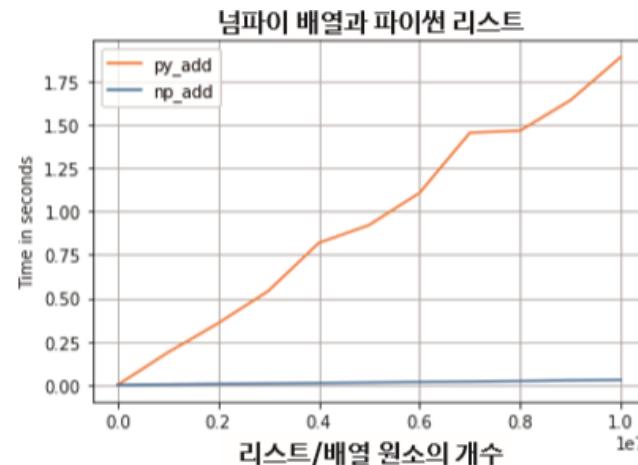
### ➤ 리스트보다 NumPy의 배열이 훨씬 빠르다

- 리스트는 여러 개의 값을 저장할 수 있는 자료구조로서 강력하고 활용도가 높다. 리스트는 다양한 자료형의 데이터를 여러 개 저장할 수 있으며 데이터를 변경하거나 추가, 제거할 수 있다.

```
>>> scores = [10, 20, 30, 40, 50, 60]
```

- 하지만 데이터 과학에서는 파이썬의 기본 리스트로 충분하지 않다. 데이터를 처리할 때는 리스트와 리스트 간의 다양한 연산이 필요한데, 파이썬 기본 리스트는 이러한 기능이 부족하며 리스트를 다루는 일은 연산 속도도 빠르지 않다. 따라서 데이터 과학자들은 기본 리스트 대신에 Numpy를 선호한다.

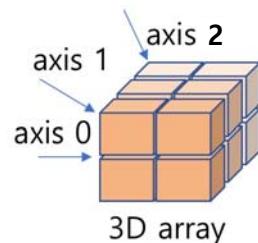
- NumPy는 대용량의 배열과 행렬 연산을 빠르게 수행하며, 고차원적인 수학 연산자와 함수를 포함하고 있는 파이썬 라이브러리이다.
- 표에서 알 수 있듯이 NumPy의 배열은 주황색으로 표시된 파이썬의 리스트에 비하여 처리속도가 매우 빠름을 알 수 있다.



[출처] “따라하며 배우는 파이썬과 데이터 과학”, 생능출판사 강의자료

### ➤ 리스트보다 NumPy의 배열이 훨씬 빠르다

- NumPy의 핵심적인 객체는 다차원 배열이다. 예를 들어서 정수들의 2차원 배열(테이블)을 NumPy를 이용해서 생성할 수 있다. 배열의 각 요소는 인덱스index라고 불리는 정수들로 참조된다.
- NumPy에서 차원은 축axis이라고도 한다.



### ➤ 리스트와 NumPy 배열은 무엇이 다른가

- 데이터 과학자들은 왜 NumPy를 많이 사용할까?
- NumPy는 성능이 우수한 ndarray 객체를 제공한다.
- ndarray의 장점을 정리하면 아래와 같다.

- ndarray는 C 언어에 기반한 배열 구조이므로 메모리를 적게 차지하고 속도가 빠르다.
- ndarray를 사용하면 배열과 배열 간에 수학적인 연산을 적용할 수 있다.
- ndarray는 고급 연산자와 풍부한 함수들을 제공한다.

**[Note]** NumPy 배열의 구조는 “Shape”으로 표현됩니다. Shape은 배열의 구조를 파이썬 튜플 자료형을 이용하여 정의합니다. 예를 들어 28X28 컬러 사진은 높이가 28, 폭이 28, 각 픽셀은 3개 채널(RGB)로 구성된 데이터 구조를 갖습니다. 즉 컬러 사진 데이터는 Shape이 **(28, 28, 3)**인 3차원 배열입니다. 다차원 배열은 입체적인 데이터 구조를 가지며, 데이터의 차원은 여러 갈래의 데이터 방향을 갖습니다. 다차원 배열의 데이터 방향을 axis로 표현할 수 있습니다. 행방향(높이), 열방향(폭), 채널 방향은 각각 **axis=0, axis=1** 그리고 **axis=2**로 지정됩니다. NumPy 집계(Aggregation) 함수는 배열 데이터의 집계 방향을 지정하는 axis 옵션을 제공합니다.

[출처] “따라하며 배우는 파이썬과 데이터 과학”, 생능출판사 강의자료

### ➤ 리스트와 NumPy 배열은 무엇이 다른가

- NumPy 배열의 장점을 이해하기 위한 간단한 예제를 살펴보자. 다음과 같이 학생들의 중간고사와 기말고사 성적을 저장하고 있는 리스트가 있다고 하자.

```
mid_scores = [10, 20, 30] # 파이썬 리스트 mid_scores
final_scores = [70, 80, 90] # 파이썬 리스트 final_scores
```

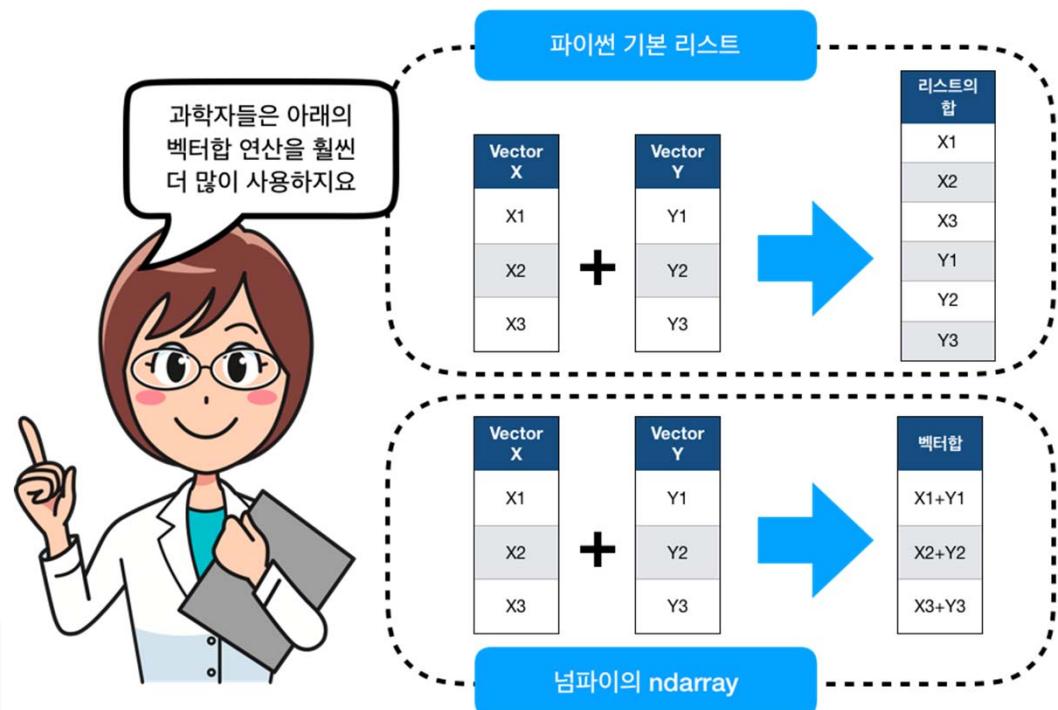
- Mid scores는 학생 3명의 중간고사 성적을 저장하고 final scores은 기말고사 성적을 저장하고 있는데, 다음 표와 같이 각 학생들의 중간고사 성적과 기말고사 성적을 합하여 오른쪽의 총점(total)이라는 리스트를 만들고 싶다.

	중간고사 성적	기말고사 성적	총점
학생 #1	10	70	80
학생 #2	20	80	100
학생 #3	30	90	120

- 그런데, 파이썬 리스트 더하기 연산자는 두 리스트를 연결하므로, `mid_scores + final_scores`을 적용하면 다음과 같이 2개의 리스트를 연결한 리스트가 만들어진다.

```
>>> total = mid_scores + final_scores # 원소간의 합이 아닌 리스트를 연결함
>>> total
[10, 20, 30, 70, 80, 90]
```

- 이것은 분명히 우리가 원하는 연산이 아니다.
- 하지만 NumPy 배열에 +연산을 하면, 대응되는 값끼리 합쳐진 결과를 얻을 수 있다.



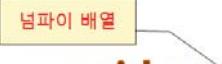
## NumPy 설치 및 특징

### ➤ NumPy의 별칭 만들기, 그리고 간단한 배열 연산하기

- 파이썬에서 NumPy를 사용하려면, 다음과 같이 NumPy 패키지를 불러와야 한다.
- `import ~ as`에서 `as` 뒤에 나타나는 이름은 `as` 앞의 이름을 대체하는 별칭이다. 보통 numpy의 별칭 alias으로 `np`를 사용한다.
- 앞으로 사용하게 될 NumPy 사용 프로그램에서 이 코드의 호출은 필수이다.

```
import numpy as np
```

- NumPy 배열을 만들려면 NumPy가 제공하는 `array()` 함수를 이용한다. `array()` 함수에 파이썬 리스트를 전달하면 NumPy 배열이 생성된다.

넘파이 배열  파이썬 리스트 

```
mid_scores = np.array([10, 20, 30])
```

```
import numpy as np
```

```
np.__version__
```

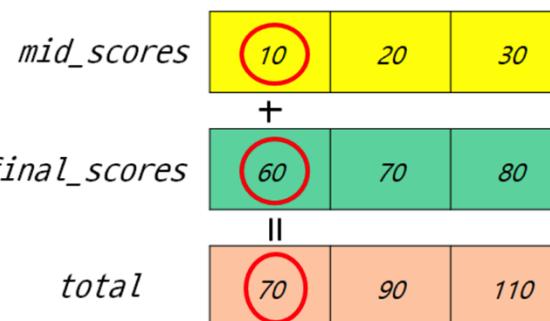
```
'1.21.2'
```

- NumPy 버전 확인

[출처] “따라하며 배우는 파이썬과 데이터 과학”, 생능출판사 강의자료

```
mid_scores = np.array([10, 20, 30])  
final_scores = np.array([60, 70, 80])
```

- 2개의 배열을 합하여 학생들의 총점을 계산해보자. 이번에는 ndarray의 덧셈 연산을 수행한다.



```
total = mid_scores + final_scores  
print('시험성적의 합계 :', total)      # 각 요소별 합계가 나타난다  
print('시험성적의 평균 :', total/2)    # 모든 요소를 2로 나눈다
```

```
시험성적의 합계 : [ 70  90 110]  
시험성적의 평균 : [35. 45. 55.]
```

## ➤ NumPy의 핵심 다차원배열을 알아보자

- NumPy의 핵심이 되는 다차원배열 ndarray은 다음과 같은 속성을 가지고 있다.
- 이러한 속성을 이용하여 프로그램의 오류를 찾거나 배열의 상세한 정보를 손쉽게 조회할 수 있다.

```
>>> a = np.array([1, 2, 3])      # 넘파이 ndarray 객체의 생성
>>> a.shape      # a 객체의 형태(shape)
(3,)
>>> a.ndim       # a 객체의 차원
1
>>> a.dtype      # a 객체 내부 자료형
dtype('int32')
>>> a.itemsize   # a 객체 내부 자료형이 차지하는 메모리 크기(byte)
4
>>> a.size       # a 객체의 전체 크기(항목의 수)
3
```

- 각각의 속성에 관련한 상세한 설명은 다음 표와 같다.

속성	설명
<code>ndim</code>	배열 축 혹은 차원의 개수
<code>shape</code>	배열의 차원으로 $(m, n)$ 형식의 튜플 형이다. 이때, $m$ 과 $n$ 은 각 차원의 원소의 크기를 알려주는 정수
<code>size</code>	배열 원소의 개수이다. 이 개수는 shape내의 원소의 크기의 곱과 같다. 즉 $(m, n)$ 형태 배열의 size는 $m \cdot n$ 이다.
<code>dtype</code>	배열내의 원소의 형을 기술하는 객체이다. 넘파이는 파이썬 표준 자료형을 사용할 수 있으나 넘파이 자체의 자료형인 <code>bool_</code> , <code>character</code> , <code>int_</code> , <code>int8</code> , <code>int16</code> , <code>int32</code> , <code>int64</code> , <code>float</code> , <code>float8</code> , <code>float16_</code> , <code>float32</code> , <code>float64</code> , <code>complex_</code> , <code>complex64</code> , <code>object</code> 형을 사용할 수 있다.
<code>itemsize</code>	배열내의 원소의 크기를 바이트 단위로 기술한다. 예를 들어 <code>int32</code> 자료형의 크기는 $32/8 = 4$ 바이트가 된다.
<code>data</code>	배열의 실제 원소를 포함하고 있는 버퍼
<code>stride</code>	배열 각 차원별로 다음 요소로 점프하는 데에 필요한 거리를 바이트로 표시한 값을 모은 튜플



### 잠깐 – 네이처에 소개된 넘파이

2020년 9월 저명한 과학 저널 [네이처Nature](#)에 넘파이에 대한 리뷰 논문이 게재되었다. 기초과학 분야 논문을 주로 게재하는 네이처에 소프트웨어 모듈이 소개되는 것은 매우 이례적인 일이다. 논문은 [텐서tensor](#)라 불리는 다차원 배열을 효율적으로 다루는 넘파이가 과학 분야에 파이썬 생태계를 제공했고, 과학 각 분야의 수치 데이터 처리 기술들이 상호운용성을 가질 수 있도록 하는 역할을 했다고 밝히고 있다. 블랙홀의 모습을 최초로 찍어낸 [이벤트 호라이즌Event Horizon](#) 연구팀이 넘파이 배열을 이용해 연구의 각 단계별 데이터를 저장하고 다루었던 사례도 소개하였다.

영국 왕립 천문학회는 넘파이를 바탕으로 만들어진 Astropy에게 상을 수여하면서 "Astropy 프로젝트는 수백 명의 젊은 과학자들에게 버전 제어, 단위 검사, 코드 리뷰, 이슈 트래킹과 같은 전문가 수준의 소프트웨어 개발 경험을 제공했다. 현대의 연구자에게 이것들은 핵심적 기술들인데 물리학이나 천문학 분야의 정규 대학 교육에서는 종종 누락되고 있다."라고 밝혔다.

## LAB<sup>10-1</sup> ndarray 객체를 생성하고 속성을 알아보자

넘파이의 다차원 배열을 이해하기 위해 다음과 같은 일들을 수행해 보자.

- 0에서 9까지의 정수 값을 가지는 ndarray 객체 array\_a를 넘파이를 이용하여 작성하여 내용을 출력해 보라.
- range() 함수를 사용하여 0에서 9까지의 정수 값을 가지는 ndarray 객체 array\_b를 만들고 아래의 결과와 같이 나타나도록 하여라.
- 문제 2의 코드를 수정하여 0에서 9까지의 정수 값 중에서 다음과 같이 짝수를 가지는 ndarray 객체 array\_c를 생성하여라.
- array\_c의 shape, ndim, dtype, size, itemsize를 출력해 보라.

### 원하는 결과

```
실습 1 : array_a = [0 1 2 3 4 5 6 7 8 9]
```

```
실습 2 : array_b = [0 1 2 3 4 5 6 7 8 9]
```

```
실습 3 : array_c = [0 2 4 6 8]
```

```
실습 4:
```

```
array_c의 shape : (5,)
```

```
array_c의 ndim : 1
```

```
array_c의 ctype : int64
```

```
array_c의 size : 5
```

```
array_c의 itemsize : 8
```

```
실습 1 : array_a = [0 1 2 3 4 5 6 7 8 9]
```

```
실습 2 : array_b = [0 1 2 3 4 5 6 7 8 9]
```

```
실습 3 : array_c = [0 2 4 6 8]
```

```
실습 4:
```

```
array_c의 shape : (5,)
```

```
array_c의 ndim : 1
```

```
array_c의 dtype : int32
```

```
array_c의 size : 5
```

```
array_c의 itemsize : 4
```

```
import numpy as np
```

```
# 실습 1
```

```
array_a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print('실습 1 : array_a =', array_a)
```

```
# 실습 2
```

```
array_b = np.array(range(10))
print('실습 2 : array_b =', array_b)
```

```
# 실습 3
```

```
array_c = np.array(range(0,10,2))
print('실습 3 : array_c =', array_c)
```

```
# 실습 4
```

```
print('실습 4: ')
print('array_c의 shape :', array_c.shape)
print('array_c의 ndim :', array_c.ndim)
print('array_c의 ctype :', array_c.dtype)
print('array_c의 size :', array_c.size)
print('array_c의 itemsize :', array_c.itemsize)
```

### ❖ 강력한 NumPy 배열의 연산을 알아보자

- NumPy 배열에는 + 연산자나 \* 연산자와 같은 수학적인 연산자를 얼마든지 적용할 수 있다. 예를 들어서 어떤 회사가 좋은 성과를 거두어 전 직원의 월급을 100만원씩 올려주기로 하였다. 어떻게 하면 될까? 현재 직원들의 월급이 [220, 250, 230]이라고 하자. 이것을 NumPy 배열에 저장한다.

```
import numpy as np  
salary = np.array([220, 250, 230])
```

- NumPy 배열에 저장된 모든 값에 100을 더하려면 다음과 같이 배열에 스칼라 값 100을 더하면 된다.

```
salary = salary + 100  
print(salary)
```

[320, 350, 330]

- 위의 코드와 같이 넘파이 배열 salary에 100을 더하면 salary 배열의 모든 요소에 100이 더해진다. 만일 모든 직원들의 월급을 2배 올려주려면 어떻게 하면 될까? 다음과 같이 넘파이의 배열에 2를 곱하면 된다

```
salary = np.array([220, 250, 230])  
salary = salary * 2  
print(salary)
```

[440, 500, 460]

- 물론 곱셈연산은 2.1과 같은 실수 값을 적용할 수도 있다

```
salary = np.array([220, 250, 230])  
salary = salary * 2.1  
print(salary)
```

[462. 525. 483.]

## ❖ 강력한 NumPy 배열의 연산을 알아보자



### 잠깐 – 넘파이의 계산은 왜 빠를까?

넘파이가 계산을 쉽고 빠르게 할 수 있는 데에는 이유가 있다. 넘파이는 각 배열마다 타입이 하나만 있다고 생각한다. 다시 말하면, 넘파이의 배열 안에는 동일한 타입의 데이터만 저장할 수 있다. 즉 정수면 정수, 실수면 실수만을 저장할 수 있는 것이다. 파이썬의 리스트처럼 여러 가지 타입을 섞어서 저장할 수는 없다. 만약 여러분들이 여러 가지 타입을 섞어서 넘파이의 배열에 전달하면 넘파이는 이것을 전부 문자열로 변경한다. 예를 들어서 다음 배열은 문자열 배열이 된다.

```
>>> tangled = np.array([ 100, 'test', 3.0, False])
>>> print(tangled)
['100' 'test' '3.0' 'False']
```

이렇게 동일한 자료형으로만 데이터를 저장하면 각각의 데이터 항목에 필요한 저장공간이 일정하다. 따라서 몇 번째 위치에 있는 항목이든 그 순서만 안다면 바로 접근할 수 있기 때문에 빠르게 데이터를 다룰 수 있는 것이다. 이렇게 원하는 위치에 바로 접근하여 데이터를 읽고 쓰는 일을 **임의 접근random access**라고 한다. 우리가 주 기억 장치로 많이 쓰는 기억장치가 **임의 접근 기억장치random access memory**이고 줄여서 RAM이라 한다. 임의 접근이 가능하기 때문에 기억장치가 회전하면서 원하는 위치의 데이터를 읽는 하드디스크보다 빠르다.

## LAB10-2 여러 사람의 BMI를 빠르고 간편하게 계산하기

넘파이를 이용하여 다수의 인원에 대해 BMI 계산을 효율적으로 적용해 보자.

수정 병원에서는 연구를 위하여 모집한 다수의 실험 대상자들의 키와 몸무게를 측정하였다. 하나의 리스트는 실험 대상자들의 키를 저장한 리스트로서 `heights`라고 하자. 또 하나의 리스트는 몸무게를 저장한 리스트로서 `weights`라고 하자.

수정 병원의 실험 대상자들의 BMI를 한 번에 계산할 수 있는 방법은 무엇일까?

### 원하는 결과

```
대상자들의 키: [1.83 1.76 1.69 1.86 1.77 1.73]
대상자들의 몸무게: [86 74 59 95 80 68]
대상자들의 BMI
[25.68007405 23.88946281 20.65754 27.45982194 25.53544639 22.72043837]
```

```
import numpy as np

heights = [ 1.83, 1.76, 1.69, 1.86, 1.77, 1.73 ]
weights = [ 86, 74, 59, 95, 80, 68 ]

np_heights = np.array(heights)
np_weights = np.array(weights)

bmi = np_weights/(np_heights**2)
print('대상자들의 키:', np_heights)
print('대상자들의 몸무게:', np_weights)
print('대상자들의 BMI')
print(bmi)
```

## ❖ 인덱싱 Indexing과 슬라이싱 Slicing

- 지금부터는 다음과 같은 성적이 저장된 1차원 배열에서 요소들을 꺼내는 방법을 살펴보자.

```
>>> scores = np.array([88, 72, 93, 94, 89, 78, 99])
```

- 넘파이 배열에서 특정한 요소를 추출하려면 인덱스를 사용한다.
- 파이썬 리스트와 마찬가지로 인덱스는 **0부터 시작한다**.
- 따라서 인덱스로 2를 지정하면 93이 출력된다.

```
>>> scores[2]
93
```

- 마지막 요소에 접근하려면 리스트와 마찬가지로 인덱스로 **-1**을 주면 된다.

```
>>> scores[-1]
99
```

- 넘파이 배열에서는 다음과 같이 슬라이싱도 가능하다.
- 마지막 항목 인덱스가 4일때는 4-1인 scores[3]항목까지 슬라이싱 된다는 것에 주의하도록 하자.

```
>>> scores[1:4]      # 첫 번째, 두 번째, 세 번째, 항목을 슬라이싱 함
array([72, 93, 94])
```

인덱싱은 특정한 요소를 얻는 방법

<b>scores</b>	0	1	2	3	4	5	6
	88	72	93	94	89	78	99
	-7	-6	-5	-4	-3	-2	-1

```
>>> scores[2]
93
```

슬라이싱은 요소 집합을 얻는 방법

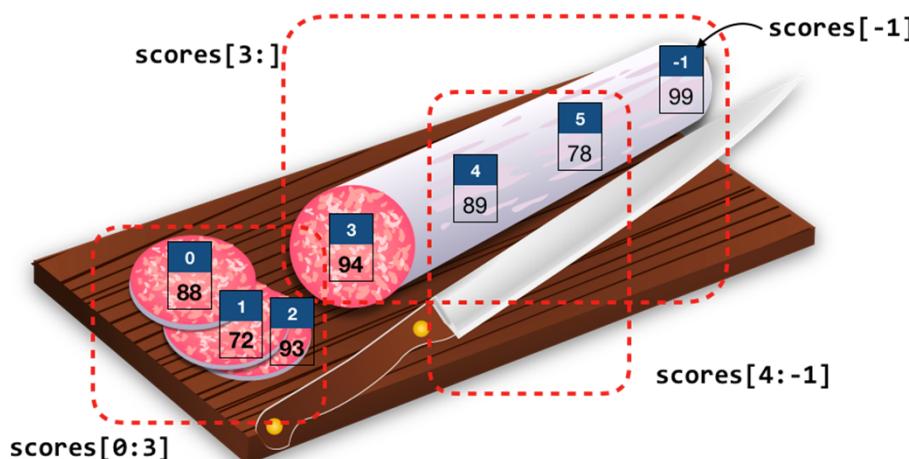
<b>scores</b>	0	1	2	3	4	5	6
	88	72	93	94	89	78	99
	-7	-6	-5	-4	-3	-2	-1

```
>>> scores[1:4]
[72, 93, 94]
```

## ❖ 인덱싱 Indexing과 슬라이싱 Slicing

- 다음과 같이 시작 인덱스나 종료 인덱스는 생략이 가능하다.
- 이것은 파이썬의 리스트와 동일하다.
- 또한 scores[4:-1]과 같은 음수 인덱싱도 가능하다.

```
>>> scores[3:]      # 마지막 인덱스를 생략하면 디폴트 값은 -1임
array([94, 89, 78, 99])
>>> scores[4:-1]    # 마지막 인덱스로 -1을 사용할 경우 -1의 앞에 있는 78까지 슬라이싱함
array([89, 78])
```



## ❖ 논리적인 인덱싱

- 논리적인 인덱싱 logical indexing이란 어떤 조건을 주어서 배열에서 원하는 값을 추려내는 것이다. 예를 들어서 사람들의 나이가 저장된 넘파이 배열 ages가 있다고 하자.

```
>>> ages = np.array([18, 19, 25, 30, 28])
```

- ages에서 20살 이상인 사람만 고르려고 하면 다음과 같은 조건식을 써준다.

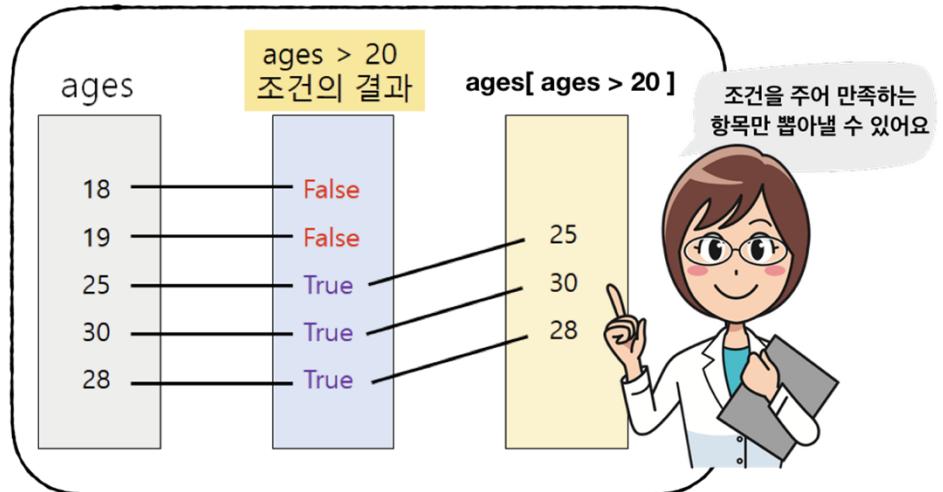
```
>>> y = ages > 20
>>> y
array([False, False, True, True, True])
```

### ❖ 논리적인 인덱싱

- 결과는 부울형의 넘파이 배열이 된다.
- ages 배열의 첫 번째와 두 번째 요소는 20보다 크지 않으므로 False가 되고 나머지 요소들은 모두 20보다 크므로 True가 되었다.
- 그런데 실제로는 배열 중에서 20살 이상인 사람들을 뽑아내는 연산이 많이 사용된다.
- 이때는 위의 부울형 배열을 인덱스로 하여 배열 ages에 보내면 된다.

```
>>> ages[ ages > 20 ]
array([25, 30, 28])
```

배열 [ 조건 ] 은 조건을 만족하는 배열 내의 항목들로 이루어진 배열



### 잠깐 - BMI가 25가 넘는 사람만 추출해 보자

앞서 실습을 통해 여러 사람의 BMI를 출력해 보았다. 이제 BMI가 25가 넘는 사람의 BMI만 출력하도록 해보자. 키와 몸무게 값을 담은 넘파이 배열 np\_heights와 np\_weights가 이미 만들어져 있다면 다음과 같이 구할 수 있다.

```
bmi = np_weights/(np_heights**2)
print(bmi[bmi > 25])      # BMI가 25 넘는 사람의 BMI만을 출력
```

## ❖ 2차원 배열의 인덱싱

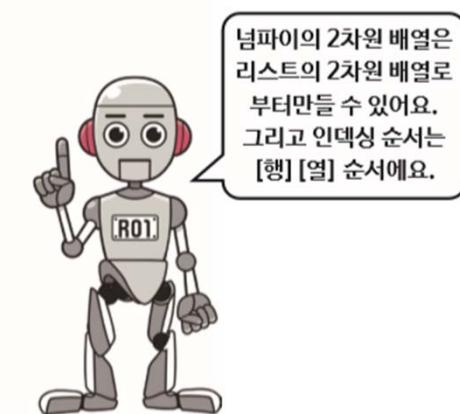
- 넘파이를 사용하면 2차원 배열도 쉽게 만들 수 있다.
- 파이썬의 2차원 리스트는 “리스트의 리스트”라고 할 수 있다.
- 수학에서의 행렬과는 비슷하지만 리스트는 행렬 연산을 지원하지 않는다.

```
>>> import numpy as np
>>> y = [[1,2,3], [4,5,6], [7,8,9]] # 2차원 배열(리스트 자료형)
>>> y
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

- 넘파이 2차원 배열은 다음과 같이 np.array()를 호출하여 생성할 수 있다. 넘파이의 2차원 배열은 수학에서의 행렬과 같이 다룰 수 있다.
- 따라서 역행렬이나 행렬식을 구하는 등의 행렬 연산들이 넘파이 배열에 쉽게 적용될 수 있도록 구현되어 있다.

```
>>> np_array = np.array(y) # 2차원 배열(넘파이 디차원 배열)
>>> np_array
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

- 2차원 배열에서 특정한 위치에 있는 요소는 어떻게 꺼낼까?
- 2차원 배열도 인덱스를 사용한다. 다만 2차원이기 때문에 인덱스가 2개 필요하다. 첫 번째 인덱스는 행의 번호이다.
- 두 번째 인덱스는 열의 번호이다. 예를 들어서 np\_array[0][2]는 30| 된다.



```
>>> np_array[0][2]
3
```

### ❖ 2차원 배열의 인덱싱

- 넘파이의 2차원 배열에서 np\_array[0][2]와 같은 형태로도 특정한 요소를 꺼낼 수 있다.
- 하지만 넘파이에서는 많이 사용되는 표기법이 있다. 0번째 행과 2번째 열에 있는 요소에 접근할 때는 콤마를 사용하여 np\_array[0, 1]로 써주어도 된다.
- 콤마 앞에 값은 행을 나타내며, 콤마 뒤에 값은 열을 나타낸다.

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

- 2차원 넘파이 배열은 행렬이라는 점을 명심하자. 행렬에서는 행의 번호와 열의 번호만 있으면 특정한 요소를 꺼낼 수 있다.
- 따라서 넘파이 스타일로 [row, col] 인덱스를 사용하면 row 행을 가져온 뒤에 거기서 col 번째 항목을 찾는 것이 아니라 바로 특정 항목에 접근하게 된다.

```
>>> np_array[0, 0]
1
>>> np_array[2, -1]
9
```

[출처] “따라하며 배우는 파이썬과 데이터 과학”, 생능출판사 강의자료

- 그리고 리스트와 유사하게 인덱스 표기법을 사용하여 배열의 요소를 변경할 수도 있다.

```
>>> np_array[0, 0] = 12    # ndarray의 첫 요소를 변경함
>>> np_array
array([[12,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9]])
```

- 파이썬 리스트와 달리, 넘파이 배열은 모든 항목이 동일한 자료형을 가진다는 것을 명심하여야 한다. 예를 들어 정수 배열에 부동 소수점 값을 삽입하고 하면 소수점 이하값은 자동으로 사라진다.

```
>>> np_array[2, 2] = 1.234  # 마지막 요소의 값을 실수로 변경하려고 하면 실패
>>> np_array
array([[12,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  1]])
```

## ❖ 넘파이 스타일의 2차원 배열 잘라내기

- 넘파이에서 슬라이싱은 큰 행렬에서 작은 행렬을 꺼집어내는 것으로 이해하면 된다. 다음은 2차원 행렬의 일부를 추출하는 코드이다.

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

```
np_array[0:2, 2:4]
```

```
array([[3, 4],
       [7, 8]])
```

- 위의 표기법은 0에서 1까지의 행에서 2에서 3까지의 열로 이루어진 행렬을 지정한 것이다. 따라서 위와 같은 행렬이 출력된다.
- NumPy의 2차원 행렬에서 하나의 행을 지정하는 방식은 다음과 같이 할 수 있다.

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

- 또한 다음과 같은 넘파이 스타일의 표기법도 가능하다.

```
>>> np_array[1, 1:3]
array([6, 7])
```

- 아래의 그림은 4x4 크기의 np\_array라는 ndarray와 이 ndarray의 인덱싱 및 슬라이싱 결과를 보여준다

np_array	np_array[0]	np_array[1, :]	np_array[:, 2]
(0, 0) (0, 1) (0, 2) (0, 3)	(0, 0) (0, 1) (0, 2) (0, 3)	(0, 0) (0, 1) (0, 2) (0, 3)	(0, 0) (0, 1) (0, 2) (0, 3)
(1, 0) (1, 1) (1, 2) (1, 3)	(1, 0) (1, 1) (1, 2) (1, 3)	(1, 0) (1, 1) (1, 2) (1, 3)	(1, 0) (1, 1) (1, 2) (1, 3)
(2, 0) (2, 1) (2, 2) (2, 3)	(2, 0) (2, 1) (2, 2) (2, 3)	(2, 0) (2, 1) (2, 2) (2, 3)	(2, 0) (2, 1) (2, 2) (2, 3)
(3, 0) (3, 1) (3, 2) (3, 3)	(3, 0) (3, 1) (3, 2) (3, 3)	(3, 0) (3, 1) (3, 2) (3, 3)	(3, 0) (3, 1) (3, 2) (3, 3)

np_array[0:2, 0:2]	np_array[0:2, 2:4]	np_array[:, :, 2]	np_array[1::2, 1::2]
(0, 0) (0, 1) (0, 2) (0, 3)	(0, 0) (0, 1) (0, 2) (0, 3)	(0, 0) (0, 1) (0, 2) (0, 3)	(0, 0) (0, 1) (0, 2) (0, 3)
(1, 0) (1, 1) (1, 2) (1, 3)	(1, 0) (1, 1) (1, 2) (1, 3)	(1, 0) (1, 1) (1, 2) (1, 3)	(1, 0) (1, 1) (1, 2) (1, 3)
(2, 0) (2, 1) (2, 2) (2, 3)	(2, 0) (2, 1) (2, 2) (2, 3)	(2, 0) (2, 1) (2, 2) (2, 3)	(2, 0) (2, 1) (2, 2) (2, 3)
(3, 0) (3, 1) (3, 2) (3, 3)	(3, 0) (3, 1) (3, 2) (3, 3)	(3, 0) (3, 1) (3, 2) (3, 3)	(3, 0) (3, 1) (3, 2) (3, 3)

## ❖ 넘파이 스타일의 2차원 배열 잘라내기

`np_array[a:b][c:d]`와 `np_array[a:b, c:d]` 차이 이해



## 잠깐 - 파이썬 리스트 슬라이싱과 넘파이 스타일 슬라이싱의 차이

다음과 같이 파이썬 리스트 슬라이싱과 넘파이 스타일의 슬라이싱을 적용했을 때, 슬라이싱에 사용된 범위와 간격은 동일하지만 전혀 다른 결과가 나온다. 이 이유를 잘 이해하는 것이 중요하다.

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

print(np_array[::-2][::2]) # 첫 슬라이싱: 0행, 2행 선택, 두 번째 슬라이싱: 그 중 0행 선택
print(np_array[::-2,:2]) # 행 슬라이싱: 0행, 2행 선택, 열 슬라이싱: 0열 2열 선택
```

```
[[1 2 3 4]]
[[ 1  3]
 [ 9 11]]
```

```
np_array = np.arange(1,17).reshape(4,4)
np_array
```

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

```
np_array[0:2]
```

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

```
np_array[0:2][0]
np_array[0:2,0]
```

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

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

```
np_array[0:2][1]
np_array[0:2,1]
```

```
array([5, 6, 7, 8])
```

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

```
np_array[::-2][::2]
np_array[::-2, ::2]
```

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

```
array([[ 1,  3],
       [ 9, 11]])
```

### ❖ 2차원 배열에서 논리적인 인덱싱

- 2차원 배열에서도 어떤 조건을 주어서 조건에 맞는 값들만 추려낼 수 있다.
- 이 코드는 np\_array에 1에서 9까지의 값이 들어있는 2차원 배열에 대해서 np\_array > 5 계산식의 결과를 보여준다.
- 이 계산의 결과 1, 2, 3, 4, 5까지의 값은 1>5, 2>5,..., 5>5 의 연산의 결과인 False 값이 나타남을 알 수 있다. 나머지 값인 6, 7, 8, 9는 모두 5보다 크기 때문에 True 값이 나타남을 알 수 있다.

```
>>> np_array = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> np_array > 5
array([[False, False, False],
       [False, False, True],
       [ True,  True,  True]])
```

- 2차원 배열에 대해서 비교연산자를 사용하면 위와 같이 True, False로 이루어진 배열이 반환되는데, 이것을 이용하여 특정한 값들을 뽑아낼 수도 있다.
- 다음과 같이 코드를 살펴보자. 이 코드를 살펴보면 np\_array의 큰 괄호 내부에서 np\_array > 5 연산을 적용하였다.
- 이렇게 할 경우 np\_array 배열내의 모든 원소들 중에서 5보다 큰 값만이 추출되어 [6, 7, 8, 9]와 같은 1차원 행렬이 출력되는 것을 볼 수 있다.

```
>>> np_array[ np_array > 5 ]
array([6, 7, 8, 9])
```

[출처] “따라하며 배우는 파이썬과 데이터 과학”, 생능출판사 강의자료

- 좀 더 나아가 아래와 같이 세번째 열을 추려내는 연산을 살펴보면 세번째 열의 모든 원소 [3, 6, 9]가 나타남을 볼 수 있다.

```
>>> np_array[:, 2]
array([3, 6, 9])
```

- 이제 이 슬라이싱의 결과 값 [3, 6, 9]중에서 5를 넘는 값이 있는지를 부울형으로 반환한다. 이 경우 간단하게 다음과 같은 연산을 통해서 False, True, True를 얻을 수 있다.

```
>>> np_array[:, 2] > 5
array([False, True, True])
```

### ❖ 2차원 배열에서 논리적인 인덱싱

- 이제 크기 비교 연산자를 약간 수정하여 다음과 같은 짹수 구하기 연산자를 적용해 보자.
- 위에서 살펴본 바와 같이 %2의 결과가 0인 경우가 짹수인 경우에 해당하므로 결과는 짹수 값이 있는 곳만 True가 나타나고 나머지는 모두 False가 나타남을 볼 수 있다.

```
>>> np_array[:] % 2 == 0
array([[False,  True,  False],
       [ True,  False,  True],
       [False,  True,  False]])
```

- 이제 이 값이 True인 원소만을 추출하면 손쉽게 다음과 같은 1차원 배열을 얻을 수 있다. 이를 응용하면 특정수의 배수를 추출하는 등의 필터링 작업을 손쉽게 할 수 있다

```
>>> np_array[ np_array % 2 == 0 ]
array([2, 4, 6, 8])
```

[출처] “따라하며 배우는 파이썬과 데이터 과학”, 생능출판사 강의자료

### LAB10-3 2차원 배열 연습하기

문자 데이터를 저장하고 있는 어떤 2차원 넘파이 배열 x에서 'c' 문자가 몇 개 있는지 알고 싶다. 'c'만을 추출하여 배열을 만들어 보라.

```
x = np.array( [['a', 'b', 'c', 'd'],
                ['c', 'c', 'g', 'h']])
```

그리고 다음과 같은 두 개의 2차원 배열에 정수를 담아 두 배열을 차이 결과를 확인해 보라.

```
mat_a = np.array( [[10, 20, 30], [10, 20, 30]])
mat_b = np.array( [[2, 2, 2], [1, 2, 3]])
```

#### 원하는 결과

```
['c' 'c' 'c']
[[ 8 18 28]
 [ 9 18 27]]
```

```
print(x [ x == 'c' ])
print(mat_a - mat_b)
```

## ❖ Boolean Indexing

- NumPy의 불린 인덱싱은 배열 각 요소의 선택 여부를 True, False 지정하는 방식입니다. 해당 인덱스의 True만을 조회합니다.
- a1 배열에서 요소의 값이 짝수인 요소들의 집합은?

```
# 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:int64
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]]
```

※ pprint(): 81페이지 정의

```
# 짝수인 요소 확인
# numpy broadcasting 을 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

```
# Numpy 객체 출력 용도의 pprint 함수 정의
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)
```

## ❖ Boolean Indexing

### ➤ Boolean Indexing의 응용

- 2014년 시애클 강수량 데이터: ./data/seattle2014.csv
- 2014년 시애클 1월 평균 강수량은?

```
!head -n 3 ./data/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,20140
101,0,0,0,72,33,12,340,310,36,40,-9999,-9999,-9999,-9999
GHCND:USW00024233,SEATTLE TACOMA INTERNATIONAL AIRPORT WA US,20140
102,41,0,0,106,61,32,190,200,94,116,-9999,-9999,-9999
```

```
# 데이터 로딩
import pandas as pd
rains_in_seattle = pd.read_csv("./data/seattle2014.csv")
rains_arr = rains_in_seattle['PRCP'].values
print("Data Size:", len(rains_arr))
```

Data Size: 365

```
# 날짜 배열
days_arr = np.arange(0, 365)
```

<https://github.com/jakevdp/PythonDataScienceHandbook/blob/master/notebooks/data/Seattle2014.csv>

```
# 1월의 날수 boolean index 생성
condition_jan = days_arr < 31
```

```
# 40일 조회
condition_jan[:40]
```

```
array([ True,  True,  True,  True,  True,  True,  True,  True,  Tr
ue,
       True,  True,  True,  True,  True,  True,  True,  True,  Tr
ue,
       True,  True,  True,  True,  True,  True,  True,  True,  Tr
ue,
       True,  True,  True, False, False, False, False, Fals
e,
       False, False, False, False], dtype=bool)
```

```
# 1월의 강수량 추출
rains_jan = rains_arr[condition_jan]
```

```
# 강수량 데이터 수 (1월: 31일)
len(rains_jan)
```

31

```
# 1월 강수량 총합
np.sum(rains_jan)
```

940

```
# 1월 평균 강수량
np.mean(rains_jan)
```

30.322580645161292

## ❖ Fancy Indexing

- 배열에 인덱스 배열을 전달하여 요소를 참조하는 방법입니다.

```
arr = np.arange(1, 25).reshape((4, 6))
pprint(arr)
```

```
shape: (4, 6), dimension: 2, dtype:int64
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]]
```

### Fancy 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])
```

### Fancy Case 2

```
# 전체 행에 대해서, 1, 2번 컬럼 참조
arr[:, [1, 2]]
```

```
array([[ 2,  3],
       [ 8,  9],
       [14, 15],
       [20, 21]])
```

## ❖ 배열과 벡터



### 잠깐 – 배열의 차원과 벡터의 차원

차원dimension은 어떤 수치 데이터를 다룰 때에 고려해야 하는 속성이 몇 개인지에 따라 결정된다. 예를 들어 3차원 공간의 위치는 x축 위의 위치, y축 위의 위치, 그리고 z축 위의 위치를 모두 고려해야만 정확한 한 점을 가리킬 수 있기 때문에 "3차원" 좌표로 표현한다.

차원이라는 용어를 많이 사용하는 데이터 구조로 배열array이 있다. 가장 단순한 배열은 데이터가 하나의 줄로 나열되어 있는 것이다. 이것을 1차원 배열이라고 부른다. 수학과 과학 분야에서 이런 수치 데이터를 벡터vector라고 부른다. 벡터는 원소의 개수가 차원이 된다. 따라서  $[1, 4]$ ,  $[3, 2, 1]$ 은 모두 1차원 배열이지만 벡터로 간주하면 각각 1차원 데이터가 2개와 3개씩 있는 2차원 벡터, 3차원 벡터이다. 벡터의 차원은 몇 개의 항목이 있는지에 따라 결정되고, 배열의 차원은 몇 가지 방향으로 줄을 지어 있는지를 표현하는 것이다.

배열의 차원을 한 단계 높이면 1차원 배열을 여러 줄로 겹쳐 놓은 형태가 된다. 이때 어떤 항목을 지목하는 인덱스를 표현하려면 두 가지 방향으로 각각 하나씩의 위치값이 필요하고, 이 방향을 각각 배열의 축이라 부른다. 배열의 차원은 축의 개수이므로, 이런 배열은 2차원 배열이라 부른다. 수학에서는 각각의 축을 행row과 열column이라고 부르며, 이런 모양의 데이터를 행렬matrix라고 부른다.

같은 방식으로 배열의 차원을 높여 보자. 3차원 배열은 입체적인 모양이 될 것이다. 이런 구조가 표현하는 데이터는 벡터나 행렬이 아니라 텐서tensor라고 부른다. 텐서는 3차원 배열뿐만 아니라 모든 차원의 배열을 포괄하는 개념이므로 벡터는 1차원 텐서, 행렬은 2차원 텐서라 할 수 있다.

머릿속에 그림을 그리기는 어렵지만 코드로는 4차원, 5차원 배열을 쉽게 만들 수 있다. 3차원 배열을 나열한 배열, 그렇게 만든 4차원 배열을 또 나열한 배열이다. 그리고 이 모든 것이 텐서이다.

## LAB10-4 넘파이 배열의 형태 알아내고 슬라이싱하여 연산

검사 대상자들의 키와 몸무게가 다음과 같이 2차원 넘파이 배열에 저장되었다고 하자.

```
import numpy as np

x = np.array([[ 1.83, 1.76, 1.69, 1.86, 1.77, 1.73 ],
              [ 86.0, 74.0, 59.0, 95.0, 80.0, 68.0 ]])
y = x[0:2, 1:3]
z = x[0:2][1:3]
```

x와 y, 그리고 z의 형태가 어떠한지 확인해 보라.

그리고 이 정보를 바탕으로 각 대상자들의 BMI 값을 저장한 배열을 생성해 보라.

### 원하는 결과

```
x shape : (2, 6)
y shape : (2, 2)
z shape : (1, 6)
z values = : [[86. 74. 59. 95. 80. 68.]]
BMI data
[0.00024743 0.0003214 0.00048549 0.00020609 0.00027656 0.00037413]
```

```
import numpy as np

x = np.array([[ 1.83, 1.76, 1.69, 1.86, 1.77, 1.73 ],
              [ 86.0, 74.0, 59.0, 95.0, 80.0, 68.0 ]])
y = x[0:2, 1:3]
z = x[0:2][1:3]

print('x shape :', x.shape)
print('y shape :', y.shape)
print('z shape :', z.shape)
print('z values = :', z)

bmi = x[0] / x[1]**2
print('BMI data')
print(bmi)
```

## LAB10-5 2차원 배열에서 특정 조건을 만족하는 행만 추출하기

선수들의 키와 몸무게가 하나의 리스트를 구성하고 있으며, 또 이들의 리스트로 이루어진 데이터 player가 있다.

```
players = [[170, 76.4],
           [183, 86.2],
           [181, 78.5],
           [176, 80.1]]
```

이것을 바탕으로 넘파이 2차원 배열을 만들어 보고, 선수들 가운데 몸무게가 80을 넘는 선수들만 골라서 정보를 출력해 보자. 또 키가 180 이상인 선수들의 정보도 추출해 보자.

### 원하는 결과

```
몸무게가 80 이상인 선수 정보
[[183.  86.2]
 [176.  80.1]]
키가 180 이상인 선수 정보
[[183.  86.2]
 [181.  78.5]]
```

```
import numpy as np

players = [[170, 76.4],
           [183, 86.2],
           [181, 78.5],
           [176, 80.1]]

np_players = np.array(players)

print('몸무게가 80 이상인 선수 정보')
print(np_players[ np_players[:, 1] >= 80.0 ])

print('키가 180 이상인 선수 정보')
print(np_players[ np_players[:, 0] >= 180.0 ])
```

```
array([[170. ,  76.4],
       [183. ,  86.2],
       [181. ,  78.5],
       [176. ,  80.1]])
```

## ❖ ndarray 생성

- NumPy는 배열을 쉽게 생성하는 함수도 지원하고 있다.
- `zeros((n,m))`:  $n \times m$  배열(혹은 행렬)을 생성해서 초기값은 0
- `eye(n)`:  $n \times n$  크기의 단위행렬 identity matrix를 생성
  - ✓ 정방행렬 중에서 대각선 성분이 1이고 나머지 성분이 0인 행렬

대화창 실습 : 초기값을 가지는 행렬의 생성

```
>>> np.zeros((2, 3))
array([[0., 0., 0.],
       [0., 0., 0.]])
```

모든 값이 0인 2x3 크기 행렬

```
>>> np.ones((2, 3))
array([[1., 1., 1.],
       [1., 1., 1.]])
```

모든 값이 1인 2x3 크기 행렬

```
>>> np.full((2, 3), 100)
array([[100, 100, 100],
       [100, 100, 100]])
```

모든 값이 100인 2x3 행렬

`>>> np.eye(3)`

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

3x3 크기의 단위행렬

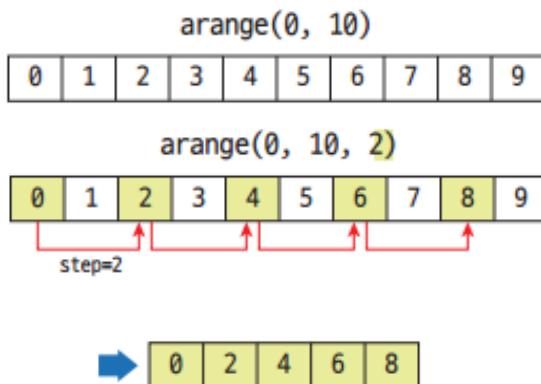
2x3 크기의 랜덤행렬

```
>>> np.random.random((2, 3))
array([[0.87143684, 0.44538612, 0.11908973],
       [0.87548557, 0.57502347, 0.87641631]])
```

### ❖ arange() 함수

- `numpy.arange([start,] stop[, step,], dtype=None)`



[그림 11-8] arange(0, 10, 2) 실행 결과

대화창 실습 : arange() 함수를 이용한 배열 생성

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

실수 step 값을 사용할 수 있음

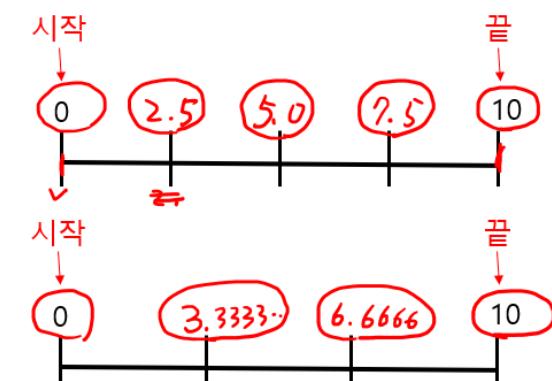
```
>>> np.arange(0.0, 1.0, 0.2)
array([0. , 0.2, 0.4, 0.6, 0.8])
```

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(0, 1.0, 0.3))
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    list(range(0, 1.0, 0.3))
TypeError: 'float' object cannot be interpreted as an integer
```

동일한 간격을 가진 연속된 값을 생성

대화창 실습 : linspace() 함수를 이용한 배열 생성

```
>>> np.linspace(0, 10, 5)
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
>>> np.linspace(0, 10, 4)
array([ 0.        ,  3.33333333,  6.66666667, 10.        ])
```



## 배열 생성 및 데이터 생성 함수

### ❖ `arange()` 함수와 `range()` 함수

- 시작값을 지정하려면 다음과 같이 한다.

```
>>> np.arange(1, 6)  
array([1, 2, 3, 4, 5])
```

- 증가되는 값을 지정하려면 다음과 같이 한다.

```
>>> np.arange(1, 10, 2)  
array([1, 3, 5, 7, 9])
```

- for 반복문을 위해 많이 사용했던 `range()` 함수와 비슷하다. 그런데 `range()`를 통해 생성된 것은 반복 가능 객체이고 이것으로 리스트를 만들수 있다.

```
>>> range(5)  
range(0, 5)  
>>> range(0, 5, 2)  
range(0, 5, 2)  
>>> list(range(5))  
[0, 1, 2, 3, 4]
```

- 따라서 `range()`를 써서 `arange()`와 같이 넘파이 배열을 만들고 싶다면 다음과 같은 일을 수행하면 된다.

```
>>> np.array(range(5))  
array([0, 1, 2, 3, 4])
```

[출처] “따라하며 배우는 파이썬과 데이터 과학”, 생능출판사 강의자료

### ❖ `linspace()` 함수와 `logspace()` 함수

- `linspace()`는 상당히 많이 사용되는 함수이다. `linspace()`는 시작값부터 끝값 까지 균일한 간격으로 지정된 개수만큼의 배열을 생성한다.



데이터 생성을 시작할 값 - 생략할 수 없음

**numpy.linspace( start, stop, num=50 )**

데이터 생성 개수 - 기본값은 50개

start에서 stop까지의 데이터를 생성해요  
하지만 정수가 아니라 실수 데이터가 생성되고  
start에서 stop까지의 간격을 균일하게 쪼개어  
num 개의 실수를 생성하지요

데이터 생성을 멈출 값으로 생략할 수 없음  
데이터는 stop-1이 아니라 stop까지 생성된다.

- `logspace()` 함수는 로그 스케일로 수들을 생성한다.



데이터 생성을  $10^{start}$  부터 시작한다.

**numpy.logspace( start, stop, num=50 )**

데이터 생성 개수 - 기본값은 50개

$10^{start}$  부터  $10^{stop}$  까지의 실수를  
로그 스케일로 볼 때 균등한 간격으로  
num 개수만큼 생성합니다.

데이터 생성을  $10^{stop}$  까지 한다.

여기서는 10을 베이스로 잡았지만, base 키워드 매개변수에 설정한 인자에 따라 바꿀수도 있다. <sup>21</sup>

### ❖ linspace() 함수와 logspace() 함수

`linspace(0, 10, 100)`이라고 호출하면 0에서 10까지 총 100개의 수들이 생성

```
>>> np.linspace(0, 10, 100)
array([ 0.          ,  0.1010101 ,  0.2020202 ,  0.3030303 ,  0.4040404 ,
       ...
       8.08080808,  8.18181818,  8.28282828,  8.38383838,  8.48484848,
       8.58585859,  8.68686869,  8.78787879,  8.88888889,  8.98989899,
       9.09090909,  9.19191919,  9.29292929,  9.39393939,  9.49494949,
       9.59595956,  9.6969697 ,  9.7979798 ,  9.8989899 ,  10.        ])
```

`logspace(x, y, n)`: 생성되는 수의 시작은  $10^x$ 부터  $10^y$  까지가 되며, n 개의 수가 생성

```
>>> np.logspace(0, 5, 10)
array([1.00000000e+00, 3.59381366e+00, 1.29154967e+01, 4.64158883e+01,
       1.66810054e+02, 5.99484250e+02, 2.15443469e+03, 7.74263683e+03,
       2.78255940e+04, 1.00000000e+05])
```

[출처] “따라하며 배우는 파이썬과 데이터 과학”, 생능출판사 강의자료

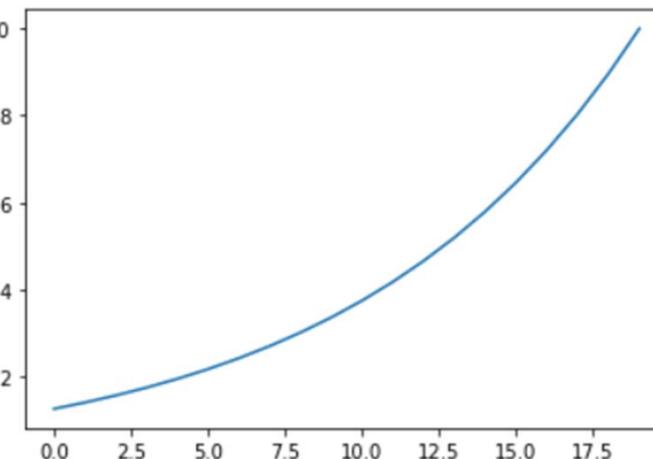
```
def pprint(arr):
    print("type:{}".format(type(arr)))
    print("shape: {}, dimension: {}, dtype:{}".format(arr.shape, arr.ndim, arr.dtype))
    print("Array's Data:\n", arr)
```

```
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  2.17191114
  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)
```

```
[<matplotlib.lines.Line2D at 0x165c7e8e2e0>]
```



### ❖ 배열의 형태를 바꾸는 **reshape()** 함수와 **flatten()** 함수

- reshape() 함수는 상당히 많이 사용되는 함수이다. 데이터의 개수는 유지한 채로 배열의 차원과 형태를 변경한다. 이 함수의 인자인 shape을 튜플의 형태로 넘겨주는 것이 원칙이지만 reshape(x, y)라고 하면 reshape((x, y))와 동일하게 처리된다.



**new\_array = old\_array.reshape( shape )**

이전 배열 old\_array의 형태가 (n,m)이고, 새롭게 얻고 싶은 형태 shape이 (l,k)라고 하면  
 $n \times m = l \times k$ 를 만족해야만 형태를 바꿀 수 있습니다.

변경하여 얻고 싶은 형태를 넘겨 줌  
 1차원: (n, )  
 2차원: (n, m)  
 3차원: (n, m, l)

```
>>> y.reshape(6, -1)
array([[ 0,  1],
       [ 2,  3],
       [ 4,  5],
       [ 6,  7],
       [ 8,  9],
       [10, 11]])
```

인수로 -1을 전달하면 데이터의 개수에 맞춰서 자동으로 배열의 형태가 결정

```
>>> y.reshape(7, 2)
```

...

**y.reshape(7, 2)**

**ValueError: cannot reshape array of size 12 into shape (7,2)**

reshape()에 의해 생성될 배열의 형태가 호환되지 않을 경우 발생하는 오류

**flatten()**은 평탄화 함수로 2차원 이상의 고차원 배열을 1차원 배열로 만들어 준다.

```
>>> y.flatten() # 2차원 배열을 1차원 배열로 만들어 준다
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
>>> y = np.arange(12)
>>> y
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

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

## reshape 함수

[출처] 유틸 파이썬, “11장 넘파이”

### ❖ ndarray의 reshape

- reshape() 메소드를 이용하여 1차원 배열을 2행5열의 다차원 행렬로 변환
- reshape()을 이용한 배열의 재구성

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

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

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

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

```
np.arange(0, 10).reshape(2, 3)
```

```
-----  
ValueError                                Traceback (most recent call last)  
~\AppData\Local\Temp\ipykernel_31612\179680428.py in <module>  
----> 1 np.arange(0, 10).reshape(2, 3)  
  
ValueError: cannot reshape array of size 10 into shape (2,3)
```

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

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

(10,) : 1차원 배열(10개의 원소)



0	1	2	3	4
5	6	7	8	9

```
np.arange(0, 24).reshape(4, 3, 2)
```

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

```
a = np.arange(6).reshape(3, 2)  
a
```

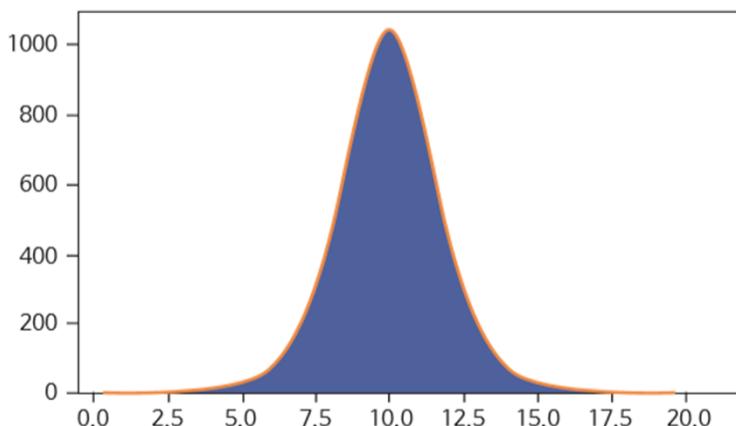
```
array([[0, 1],  
       [2, 3],  
       [4, 5]])
```

```
np.transpose(a)
```

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

### ❖ 난수를 생성해보자

- 데이터 과학자가 다룰 데이터는 상당히 크며, 몇십만 개 이상의 데이터를 다루어야 하는 경우도 많다.
- 10,000개 이상의 임의의 데이터를 어떻게 생성할 것인가?
- 직접 10,000개 이상의 데이터를 입력할 수도 있지만 어떤 확률 분포에서 난수를 생성하여서 실험 데이터로 사용할 수도 있을 것이다.



- 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.ranfd
  - ✓ np.random.randint
  - ✓ np.random.random

## 난수 생성

### np.random.normal

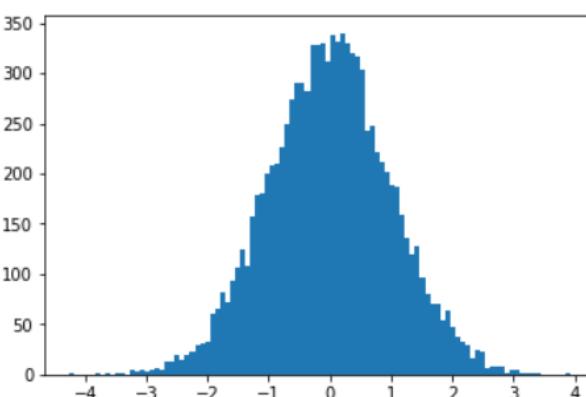
- normal(loc=0.0, scale=1.0, size=None)
- 정규 분포 확률 밀도에서 표본 추출
- loc: 정규 분포의 평균
- scale: 표준편차

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

※ pprint(): 23,81페이지 정의

```
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:float64
Array's Data:
[[ 1.4192442 -2.0771293  1.84898108]
 [-0.12303317  1.04533993  1.94901387]]
```

```
data = np.random.normal(0, 1, 10000)
import matplotlib.pyplot as plt
plt.hist(data, bins=100)
plt.show()
```



- np.random.normal()이 생성한 난수는 정규 분포의 형상을 갖습니다.
- 예제는 정규 분포로 10000개 표본을 뽑은 결과를 히스토그램으로 표현한 예입니다.
- 표본 10000개의 배열을 100 개 구간으로 구분할 때, 정규 분포 형태를 보이고 있습니다.

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

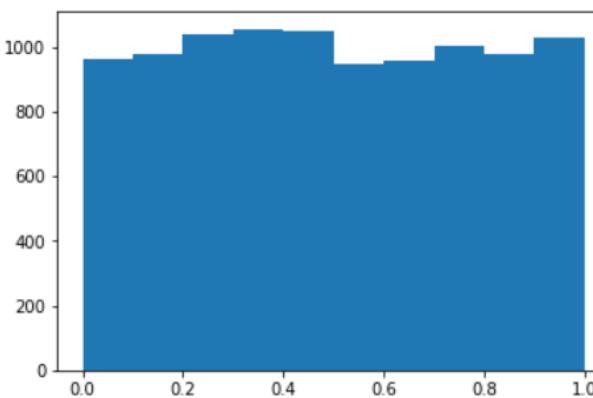
### np.random.rand

- numpy.random.rand(d0, d1, ..., dn)
- Shape 0이 (d0, d1, ..., dn)인 배열 생성 후 난수로 초기화
- 난수: [0, 1]의 Uniform Distribution 형상으로 표본 추출

```
a = np.random.rand(3,2)
pprint(a)
```

```
type:<class 'numpy.ndarray'>
shape: (3, 2), dimension: 2, dtype:float64
Array's Data:
[[ 0.1258167  0.25474262]
 [ 0.25514046  0.0918946 ]
 [ 0.19843316  0.73586066]]
```

```
data = np.random.rand(10000)
import matplotlib.pyplot as plt
plt.hist(data, bins=10)
plt.show()
```



- np.random.rand는 균등한 비율로 표본 추출
- 다음 예제는 균등 분포로 10000 개를 표본 추출한 결과를 히스토그램으로 표현한 예입니다.
- 표본 10000개의 배열을 10개 구간으로 구분했을 때 균등한 분포 형태를 보이고 있습니다.

## 난수 생성

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

### np.random.randn

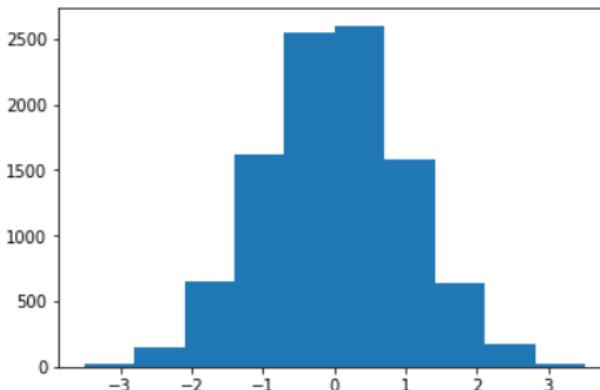
- numpy.random.randn(d0, d1, ..., dn)
- (d0, d1, ..., dn) shape 배열 생성 후 난수로 초기화
- 난수: 표준 정규 분포(standard normal distribution)에 서 표본 추출

```
a = np.random.randn(2, 4)
 pprint(a)
```

※ pprint(): 23,81 페이지 정의

```
type:<class 'numpy.ndarray'>
shape: (2, 4), dimension: 2, dtype:float64
Array's Data:
[[ 0.81791892  0.74967685 -0.20023471  0.76089888]
 [-1.13037451 -0.52569743 -1.33934774  0.75105868]]
```

```
data = np.random.randn(10000)
import matplotlib.pyplot as plt
plt.hist(data, bins=10)
plt.show()
```



- np.random.randn은 정규 분포로 표본 추출
- 다음 예제는 정규 분포로 10000개를 표본 추출한 결과를 히스토그램으로 표현한 예입니다.
- 표본 10000개의 배열을 10개 구간으로 구분했을 때 정규 분포 형태를 보이고 있습니다.

### np.random.randint

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

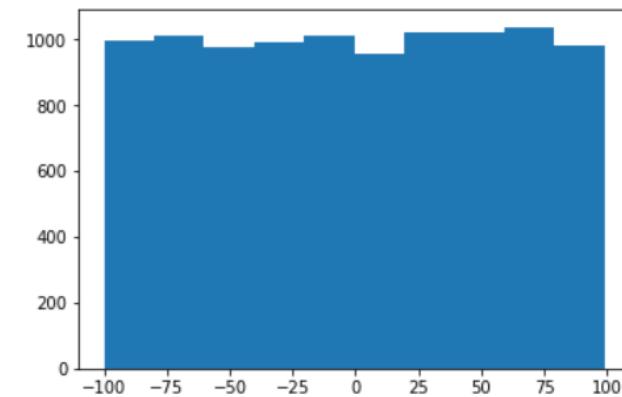
```
a = np.random.randint(5, 10, size=(2, 4))
 pprint(a)
```

```
type:<class 'numpy.ndarray'>
shape: (2, 4), dimension: 2, dtype:int64
Array's Data:
[[5 5 6 6]
 [7 9 7 9]]
```

```
a = np.random.randint(1, size=10)
 pprint(a)
```

```
type:<class 'numpy.ndarray'>
shape: (10,), dimension: 1, dtype:int64
Array's Data:
[0 0 0 0 0 0 0 0 0 0]
```

```
data = np.random.randint(-100, 100, 10000)
import matplotlib.pyplot as plt
plt.hist(data, bins=10)
plt.show()
```



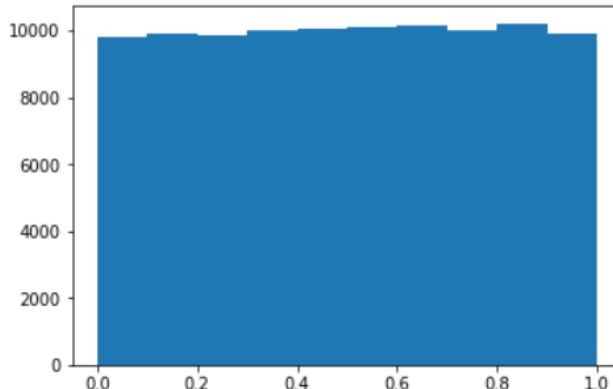
### np.random.random

- 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:
[[ 0.92646678  0.02811114  0.97379431  0.86712785]
 [ 0.18829149  0.78809537  0.52076073  0.71967828]]
```

```
data = np.random.random(100000)
import matplotlib.pyplot as plt
plt.hist(data, bins=10)
plt.show()
```



- np.random.random은 균등 분포로 표본을 추출합니다.
- 다음 예제는 정규 분포로 10000개를 표본 추출한 결과를 히스토그램으로 표현한 예입니다.
- 표본 10000개의 배열을 10개 구간으로 구분했을 때 정규 분포 형태를 보이고 있습니다.

## 난수 생성

### ❖ 난수를 생성해보자

- 넘파이에서 난수의 시드seed를 설정하는 문장은 다음과 같다.
- 시드가 설정되면 다음과 같은 문장을 수행하여 5개의 난수를 얻을 수 있다
- 난수는 컴퓨터가 규칙을 가지고 생성한 수로 정확한 의미의 난수는 아니지만 난수에 가까운 수로 의사난수pseudo random number라고 부른다.
- 무작위 수를 만드는 난수는 특정 시작 숫자로부터 난수처럼 보이는 수열을 만드는 알고리즘의 결과물입니다. 따라서 시작점을 설정함으로써 난수 발생을 재연할 수 있습니다. 난수의 시작점을 설정하는 함수가 np.random.seed입니다.

```
>>> np.random.seed(100)
```



[출처] “따라하며 배우는 파이썬과 데이터 과학”, 생능출판사 강의자료

```
>>> np.random.rand(5)  
array([0.54340494, 0.27836939, 0.42451759, 0.84477613, 0.00471886])
```

```
>>> np.random.rand(5, 3)  
array([[0.12156912, 0.67074908, 0.82585276],  
[0.13670659, 0.57509333, 0.89132195],  
[0.20920212, 0.18532822, 0.10837689],  
[0.21969749, 0.97862378, 0.81168315],  
[0.17194101, 0.81622475, 0.27407375]])
```

난수로 이루어진 2차원  
배열(5x3)

```
>>> a = 10  
>>> b = 20  
>>> (b - a) * np.random.rand(5) + a  
array([14.31704184, 19.4002982 , 18.17649379, 13.3611195 , 11.75410454])
```

10에서 20사이에 있는 난  
수 5개 생성

```
>>> np.random.randint(1, 7, size=10)  
array([4, 3, 4, 1, 1, 2, 6, 6, 2, 6])
```

```
>>> np.random.randint(1, 11, size=(4, 7))  
array([[10, 2, 6, 9, 8, 5, 3],  
[ 7, 3, 2, 9, 5, 3, 2],  
[ 3, 1, 6, 2, 9, 8, 2],  
[ 7, 5, 2, 8, 3, 3, 6]])
```

1부터 (11-1)=10사이  
의 4행 7열 난수 생성

## 난수 생성

- random 모듈의 함수는 실행할 때마다 무작위 수를 반환합니다.

```
np.random.random((2, 2))
```

```
array([[ 0.37177011,  0.80381439],  
       [ 0.98299691,  0.91079526]])
```

```
np.random.randint(0, 10, (2, 3))
```

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

```
np.random.random((2, 2))
```

```
array([[ 0.4573231 ,  0.1649216 ],  
       [ 0.76895461,  0.96333133]])
```

```
np.random.randint(0, 10, (2, 3))
```

```
array([[4, 5, 4],  
       [5, 1, 1]])
```

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

- np.random.seed 함수를 이용한 무작위수 재연
- np.random.seed(100)을 기준으로 동일한 무작위수로 초기화된 배열이 만들 어지고 있습니다.

```
# seed 값을 설정하여 아래에서 난수가 재연 가능하도록 함  
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 재설정  
np.random.seed(100)
```

```
# 위 난수의 재연  
np.random.random((2, 2))
```

```
array([[ 0.54340494,  0.27836939],  
       [ 0.42451759,  0.84477613]])
```

```
#위 난수의 재연  
np.random.randint(0, 10, (2, 3))
```

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

## 첨 부

- **ndarray** 클래스
- **ndarray** 객체 배열 및 연산
- **브로드캐스팅**

## ndarray 클래스

1) NumPy의 ndarray 클래스 이해



2) np.array (외 zeros, ones, empty, arange, linspace) 함수로  
ndarray 배열 객체를 생성 다루기

## 1) numpy.ndarray 클래스 이해

```
class numpy.ndarray(shape, dtype=float, buffer=None, offset=0,
strides=None, order=None) [source]
```

An array object represents a multidimensional, homogeneous array of fixed-size items. An associated data-type object describes the format of each element in the array (its byte-order, how many bytes it occupies in memory, whether it is an integer, a floating point number, or something else, etc.)

Arrays should be constructed using `array`, `zeros` or `empty` (refer to the See Also section below). The parameters given here refer to a low-level method (`ndarray(...)`) for instantiating an array.

For more information, refer to the `numpy` module and examine the methods and attributes of an array.

Parameters: (for the `_new_` method; see Notes below)

`shape : tuple of ints`

Shape of created array.

`dtype : data-type, optional`

Any object that can be interpreted as a numpy data type.

`buffer : object exposing buffer interface, optional`

Used to fill the array with data.

`offset : int, optional`

Offset of array data in buffer.

`strides : tuple of ints, optional`

Strides of data in memory.

`order : {'C', 'F}, optional`

Row-major (C-style) or column-major (Fortran-style) order.

First mode, `buffer` is None:

```
>>> np.ndarray(shape=(2,2), dtype=float, order='F')
array([[0.0e+000, 0.0e+000], # random
       [      nan, 2.5e-323]])
```

Second mode:

```
>>> np.ndarray((2,), buffer=np.array([1,2,3]),
...             offset=np.int_().itemsize,
...             dtype=int) # offset = 1*itemsize, i.e. skip first element
array([2, 3])
```

## ndarray 클래스

### numpy.ndarray : attribute

#### T : ndarray

The transposed array.

#### data : buffer

Python buffer object pointing to the start of the array's data.

#### dtype : dtype object

Data-type of the array's elements.

#### flags : dict

Information about the memory layout of the array.

#### flat : numpy.flatiter object

A 1-D iterator over the array.

#### imag : ndarray

The imaginary part of the array.

#### real : ndarray

The real part of the array.

#### size : int

Number of elements in the array.

#### itemsize : int

Length of one array element in bytes.

#### nbytes : int

Total bytes consumed by the elements of the array.

#### ndim : int

Number of array dimensions.

#### shape : tuple of ints

Tuple of array dimensions.

#### strides : tuple of ints

Tuple of bytes to step in each dimension when traversing an array.

#### ctypes : ctypes object

An object to simplify the interaction of the array with the ctypes module.

#### base : ndarray

Base object if memory is from some other object.

속성	설명
ndim	<ul style="list-style-type: none"><li>배열 축 혹은 차원의 갯수.</li></ul>
shape	<ul style="list-style-type: none"><li>배열의 차원으로 (m, n) 형식의 튜플 형이다. 이 때, m과 n은 각 차원의 원소의 크기를 알려주는 정수 값이다.</li></ul>
size	<ul style="list-style-type: none"><li>배열의 원소의 갯수이다. 이 갯수는 shape내의 원소의 크기의 곱과 같다.</li><li>즉 (m, n) shape 배열의 size는 <math>m \times n</math>이다.</li></ul>
dtype	<ul style="list-style-type: none"><li>배열내의 원소의 형을 기술하는 객체이다. numpy는 파이썬 표준 형을 사용할 수 있으나</li><li>numpy 자체의 자료형인 bool_, character, int_, int8, int16, int32, int64, float,</li><li>float8, float16_, float32, float64, complex_, complex64, object_ 형을 사용할 수 있다.</li></ul>
itemsize	<ul style="list-style-type: none"><li>배열내의 원소의 크기를 바이트 단위로 기술한다.</li><li>예를 들어 int32 자료형의 크기는 <math>32/8=4</math> 바이트가 된다.</li></ul>
data	<ul style="list-style-type: none"><li>배열의 실제 원소를 포함하고 있는 버퍼.</li></ul>

[출처] <https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html>

[출처] “따라하며 배우는 파이썬과 데이터 과학”, 생능출판사 강의자료

## ndarray 클래스

### numpy.ndarray : Methods

<code>all([axis, out, keepdims, where])</code>	Returns True if all elements evaluate to True.
<code>any([axis, out, keepdims, where])</code>	Returns True if any of the elements of <i>a</i> evaluate to True.
<code>argmax([axis, out])</code>	Return indices of the maximum values along the given axis.
<code>argmin([axis, out])</code>	Return indices of the minimum values along the given axis.
<code>argpartition(kth[, axis, kind, order])</code>	Returns the indices that would partition this array.
<code>argsort([axis, kind, order])</code>	Returns the indices that would sort this array.
<code>astype(dtype[, order, casting, subok, copy])</code>	Copy of the array, cast to a specified type.
<code>byteswap([inplace])</code>	Swap the bytes of the array elements
<code>choose(choices[, out, mode])</code>	Use an index array to construct a new array from a set of choices.
<code>clip([min, max, out])</code>	Return an array whose values are limited to [min, max].
<code>compress(condition[, axis, out])</code>	Return selected slices of this array along given axis.

[출처] <https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html>

<code>conj()</code>	Complex-conjugate all elements.
<code>conjugate()</code>	Return the complex conjugate, element-wise.
<code>copy([order])</code>	Return a copy of the array.
<code>cumprod([axis, dtype, out])</code>	Return the cumulative product of the elements along the given axis.
<code>cumsum([axis, dtype, out])</code>	Return the cumulative sum of the elements along the given axis.
<code>diagonal([offset, axis1, axis2])</code>	Return specified diagonals.
<code>dot(b[, out])</code>	Dot product of two arrays.
<code>dump(file)</code>	Dump a pickle of the array to the specified file.
<code>dumps()</code>	Returns the pickle of the array as a string.
<code>fill(value)</code>	Fill the array with a scalar value.
<code>flatten([order])</code>	Return a copy of the array collapsed into one dimension.
<code>getfield(dtype[, offset])</code>	Returns a field of the given array as a certain type.

# ndarray 클래스

## numpy.ndarray : Methods

<code>item(*args)</code>	Copy an element of an array to a standard Python scalar and return it.
<code>itemset(*args)</code>	Insert scalar into an array (scalar is cast to array's dtype, if possible)
<code>max([axis, out, keepdims, initial, where])</code>	Return the maximum along a given axis.
<code>mean([axis, dtype, out, keepdims, where])</code>	Returns the average of the array elements along given axis.
<code>min([axis, out, keepdims, initial, where])</code>	Return the minimum along a given axis.
<code>newbyteorder([new_order])</code>	Return the array with the same data viewed with a different byte order.
<code>nonzero()</code>	Return the indices of the elements that are non-zero.
<code>partition(kth[, axis, kind, order])</code>	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.
<code>prod([axis, dtype, out, keepdims, initial, ...])</code>	Return the product of the array elements over the given axis
<code>ptp([axis, out, keepdims])</code>	Peak to peak (maximum - minimum) value along a given axis.

[출처] <https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html>

<code>put(indices, values[, mode])</code>	Set <code>a.flat[n] = values[n]</code> for all n in indices.
<code>ravel([order])</code>	Return a flattened array.
<code>repeat(repeats[, axis])</code>	Repeat elements of an array.
<code>reshape(shape[, order])</code>	Returns an array containing the same data with a new shape.
<code>resize(new_shape[, refcheck])</code>	Change shape and size of array in-place.
<code>round([decimals, out])</code>	Return <code>a</code> with each element rounded to the given number of decimals.
<code>searchsorted(v[, side, sorter])</code>	Find indices where elements of <code>v</code> should be inserted in <code>a</code> to maintain order.
<code>setfield(val, dtype[, offset])</code>	Put a value into a specified place in a field defined by a data-type.
<code>setflags([write, align, uic])</code>	Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.
<code>sort([axis, kind, order])</code>	Sort an array in-place.
<code>squeeze([axis])</code>	Remove axes of length one from <code>a</code> .

**numpy.ndarray : Methods**

<b>std([axis, dtype, out, ddof, keepdims, where])</b>	Returns the standard deviation of the array elements along given axis.
<b>sum([axis, dtype, out, keepdims, initial, where])</b>	Return the sum of the array elements over the given axis.
<b>swapaxes(axis1, axis2)</b>	Return a view of the array with <i>axis1</i> and <i>axis2</i> interchanged.
<b>take(indices[, axis, out, mode])</b>	Return an array formed from the elements of <i>a</i> at the given indices.
<b>tobytes([order])</b>	Construct Python bytes containing the raw data bytes in the array.
<b>tofile(fid[, sep, format])</b>	Write array to a file as text or binary (default).
<b>tolist()</b>	Return the array as an <i>a.ndim</i> -levels deep nested list of Python scalars.

<b>tostring([order])</b>	A compatibility alias for <b>tobytes</b> , with exactly the same behavior.
<b>trace([offset, axis1, axis2, dtype, out])</b>	Return the sum along diagonals of the array.
<b>transpose(*axes)</b>	Returns a view of the array with axes transposed.
<b>var([axis, dtype, out, ddof, keepdims, where])</b>	Returns the variance of the array elements, along given axis.
<b>view([dtype][, type])</b>	New view of array with the same data.

## 2) ndarray 객체 배열 다루기

A 2-dimensional array of size 2 x 3, composed of 4-byte integer elements:

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]], np.int32)
>>> type(x)
<class 'numpy.ndarray'>
>>> x.shape
(2, 3)
>>> x.dtype
dtype('int32')
```

The array can be indexed using Python container-like syntax:

```
>>> # The element of x in the *second* row, *third* column, namely, 6.
>>> x[1, 2]
6
```

- 컨테이너(Container)란 자료형(Data type)의 저장 모델로 종류에 무관하게 데이터를 저장할 수 있음을 뜻한다.
- 문자열, 튜플, 리스트, 사전, 집합 등은 종류에 무관(Container)한 형식이며, 정수, 실수, 복소수 등은 단일 종류(Literal)한 형식이다.
- 컨테이너 메서드(Container Method)는 위와 같이 종류에 무관하게 저장할 수 있는 자료형의 매직 메서드(Magic Method)를 뜻한다.

For example *slicing* can produce views of the array:

```
>>> y = x[:, 1]
>>> y
array([2, 5])
>>> y[0] = 9 # this also changes the corresponding element in x
>>> y
array([9, 5])
>>> x
array([[1, 9, 3],
       [4, 5, 6]])
```

- NumPy slicing creates a view instead of a copy as in the case of builtin Python sequences such as string, tuple and list. Care must be taken when extracting a small portion from a large array which becomes useless after the extraction, because the small portion extracted contains a reference to the large original array whose memory will not be released until all arrays derived from it are garbage-collected. In such cases an explicit `copy()` is recommended.
- <https://numpy.org/doc/stable/reference/arrays.indexing.html>

## numpy.array

```
numpy.array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0,
like=None)
```

Create an array.

Parameters: `object : array_like`

An array, any object exposing the array interface, an object whose `__array__` method returns an array, or any (nested) sequence.

`dtype : data-type, optional`

The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence.

`copy : bool, optional`

If true (default), then the object is copied. Otherwise, a copy will only be made if `__array__` returns a copy, if `obj` is a nested sequence, or if a copy is needed to satisfy any of the other requirements (`dtype`, `order`, etc.).

`order : {'K', 'A', 'C', 'F}, optional`

Specify the memory layout of the array. If `object` is not an array, the newly created array will be in C order (row major) unless 'F' is specified, in which case it will be in Fortran order (column major). If `object` is an array the following holds.

Returns: `out : ndarray`

An array object satisfying the specified requirements.

`out : ndarray` ←

`empty`

Return a new uninitialized array.

`ones`

Return a new array setting values to one.

`zeros`

Return a new array setting values to zero.

`full`

Return a new array of given shape filled with value.

`subok : bool, optional`

If True, then sub-classes will be passed-through, otherwise the returned array will be forced to be a base-class array (default).

`ndmin : int, optional`

Specifies the minimum number of dimensions that the resulting array should have. Ones will be pre-pended to the shape as needed to meet this requirement.

`like : array_like`

Reference object to allow the creation of arrays which are not NumPy arrays. If an array-like passed in as `like` supports the `__array_function__` protocol, the result will be defined by it. In this case, it ensures the creation of an array object compatible with that passed in via this argument.

*New in version 1.20.0.*

## ndarray 객체 배열

- 셀 상의 변수 출력하는 환경 설정

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

- One way we can initialize NumPy arrays is from Python lists, using nested lists for two- or higher-dimensional data.

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

```
a  
a.shape
```

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

```
(3, 4)
```

```
a[0]
```

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

```
print(a[0])
```

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

## More information about arrays

1D array, 2D array, ndarray, vector, matrix

- An array referred to as a “ndarray,” which is shorthand for “N-dimensional array.”
- 1-D, or one-dimensional array, 2-D, or two-dimensional array, and so on. The NumPy ndarray class is used to represent both matrices and vectors. A vector is an array with a single column, while a matrix refers to an array with multiple columns.

In NumPy, dimensions are called axes. This means that if you have a 2D array that looks like this:

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

Your array has 2 axes. The first axis has a length of 2 and the second axis has a length of 3.

Just like in other Python container objects, the contents of an array can be accessed and modified by indexing or slicing the array. Different arrays can share the same data, so changes made on one array might be visible in another.

## ndarray 객체 배열

### How to create a basic array

```
np.array()  
np.zeros()  
np.ones()  
np.empty()  
np.arange()  
np.linspace()  
dtype
```

np.zeros() : create an array filled with 0s  
np.ones() : create an array filled with 1s  
np.empty() : creates an array whose initial content is random and depends on the state of the memory  
np.arange() : create an array with a range of elements  
np.linspace() : create an array with values that are spaced linearly in a specified interval

- To create a NumPy array, you can use the function `np.array()`.

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

Command

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



NumPy Array

1
2
3

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



1	2
3	4
5	6

```
b = np.zeros(2)  
c = np.ones(2)  
d = np.empty(2)
```

b, c, d

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

```
# np.arange(first number, last number, step size)  
e = np.arange(2, 9, 2)  
f = np.linspace(0, 10, 5)
```

e  
f

array([2, 4, 6, 8])

array([ 0. , 2.5, 5. , 7.5, 10. ])

### Specifying data type

- While the default data type is floating point (float64), you can explicitly specify which data type you want using `dtype`.

```
array = np.ones(2, dtype=int)  
array
```

array([1, 1])

## ndarray 객체 배열

- numpy의 ndarray의 속성 예시

```
>>> np.shape(np.eye(3))
(3, 3)
>>> np.shape([[1, 2]])
(1, 2)
>>> np.shape([0])
(1,)
>>> np.shape(0)
()
```

```
np.eye(3)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

```
>>> a = np.array([(1, 2), (3, 4)], dtype=[('x', 'i4'), ('y', 'i4')])
>>> np.shape(a)
(2,)
>>> a.shape
(2,)
```

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128) # 128-bit complex
>>> x.size
30
>>> np.prod(x.shape)
30
```

- Type code : i2 = int16, i4 = int32, i8 = int64
- the names of the first elements of each entry in the array can be accessed using 'x' and the second elements can be accessed using 'y':
- <https://stackoverflow.com/questions/51364975/datatype-in-numpy/51365260>

[출처] <https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html>

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

```
import numpy as np
```

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

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

```
a.shape
a.ndim
a.dtype.name
a.itemsize
a.size
type(a)
```

```
(3, 5)
```

```
2
```

```
'int32'
```

```
4
```

```
15
```

```
numpy.ndarray
```

```
b = np.array([6, 7, 8])
b
type(b)
```

```
array([6, 7, 8])
```

```
numpy.ndarray
```

## ndarray 객체 배열

대화창 실습 : numpy의 ndarray 사용

```
>>> import numpy as np  
>>> a = np.array([1, 2, 3]) # NumPy ndarray 객체의 생성  
>>> a  
array([1, 2, 3])  
>>> a.shape      # a 객체의 형태(shape)  
(3,)  
>>> a.ndim       # a 객체의 차원  
1  
>>> a.dtype      # a 객체 내부 자료형  
dtype('int32')  
>>> a.itemsize    # a 객체 내부 자료형이 차지하는 메모리 크기(byte)  
4  
>>> a.size       # a 객체의 전체 크기(항목의 수)  
3
```

대화창 실습 : NumPy와 데이터 형

```
>>> a = np.array([1, 2, 3], dtype = 'int32')  
>>> b = np.array([4, 5, 6], dtype = 'int64')  
>>> a.dtype  
dtype('int32')  
>>> b.dtype  
dtype('int64')  
>>> c = a + b  
>>> c.dtype  
dtype('int64')
```

[출처] 윤종 파이썬, “11장 넘파이”

NumPy의 가장 핵심적인 객체, 다차원 배열을 처리하며 다음과 같은 속성들이 있다

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

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

shape은 생성된 배열 객체의 형태  
를 튜플 타입으로 반환함

a.shape : (3,

a.ndim : 1

a.dtype : dtype('int32')

a.size : 3

a.itemsize : 4

[그림 11-2] 넘파이의 ndarray a와 그 속성들

정수는 8, 16, 32비트 64비트 자료  
를 사용할 수 있다.

32비트와 64비트 정수의 덧셈 결과  
는 자동 스케일업으로 64비트가 됨

## ndarray 객체 배열

### 주의 : ndarray 배열을 생성할 때 주의할 점

1. 넘파이의 배열 ndarray를 생성할 때, 반드시 대괄호를 사용하여 리스트 형식의 데이터를 만들어서 array() 함수의 인자로 넣어야 한다.

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

만일 리스트 형식으로 하지 않고 쉼표로 구분해서 입력할 경우 다음과 같은 오류가 발생된다.

```
>>> a = np.array(1, 2, 3, 4) # 잘못된 입력
```

```
...
```

```
ValueError: only 2 non-keyword arguments accepted
```

2. 넘파이의 ndarray는 리스트와는 달리, 서로 다른 자료형의 값을 원소로 가질 수 없다.

```
>>> a = np.array([1, 'two', 3, 4], dtype = np.int32)
```

```
...
```

```
ValueError: invalid literal for int() with base 10: 'two'
```

만일 다음과 같이 자료형을 명시하지 않을 경우 'two'라는 원소의 자료형인 str 형으로 자동 형 변환이 일어난다. 이 경우 모든 원소들은 문자열 형이되어 정수의 덧셈, 뺄셈 등의 연산을 사용할 수 없다.

```
>>> a = np.array([1, 'two', 3, 4])
```

```
>>> a
```

```
array(['1', 'two', '3', '4'], dtype = '<U21')
```

[출처] 유틸 파이썬, “11장 넘파이”



### NOTE : 넘파이 배열의 데이터 타입을 지정하는 두 가지 방법

넘파이 배열의 데이터 타입을 지정하는 방법에는 두 가지가 있다.

1. dtype = np.int32 와 같이 np의 int32 속성 값으로 지정하기

```
>>> a = np.array([1, 2, 3, 4], dtype = np.int32)
```

2. dtype = 'int32' 와 같이 문자열 형식으로 속성 값 지정하기

```
>>> a = np.array([1, 2, 3, 4], dtype = 'int32')
```

## ndarray 객체 배열

[출처] 유틸 파이썬, “11장 넘파이”

### ▶ ndarray의 메소드와 주요 함수

#### ndarray

대화창 실습 : ndarray의 메소드

```
>>> a = np.array([1, 2, 3]) # 1차원 ndarray 배열 생성  
>>> a.max()      # 가장 큰 값을 반환  
3  
>>> a.min()      # 가장 작은 값을 반환  
1  
>>> a.mean()     # 평균 값을 반환  
2.0
```

#### flatten()

대화창 실습 : ndarray의 flatten() 메소드

```
>>> a = np.array([[1, 1], [2, 2], [3, 3]])  
>>> a.flatten() # ndarray 배열의 평탄화 메소드  
array([1, 1, 2, 2, 3, 3])
```

#### append() 함수

대화창 실습 : ndarray의 append() 함수

```
>>> a = np.array([1, 2, 3])  
>>> b = np.array([[4, 5, 6], [7, 8, 9]])  
>>> np.append(a, b)  
array([1, 2, 3, 4, 5, 6, 7, 8, 9])  
>>> np.append([a], b, axis = 0) # [a]를 통해 2차원 배열로 만들어야 함  
array([[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9]])
```

#### rand() 함수

대화창 실습 : ndarray의 rand() 함수

```
>>> np.random.rand(3, 3) # (3, 3) shape의 난수 생성  
array([[0.63208882, 0.72259476, 0.15125742],  
       [0.60731818, 0.20682056, 0.51958311],  
       [0.644331 , 0.91940484, 0.83990604]])
```

#### randint() 함수

Ndarray의 randint()함수

```
>>> np.random.randint(0, 10, size = 10) # 0에서 10까지의 10개의 난수생성  
array([2, 0, 8, 2, 7, 0, 1, 3, 1, 3])
```

### 3) ndarray 연산

대화창 실습 : ndarray의 뒷셈(shape)이 같을 경우

```
>>> a = np.array([1, 2, 3])      # 1, 2, 3 원소를 가지는 1차원 ndarray
>>> b = np.array([4, 5, 6])      # 4, 5, 6 원소를 가지는 1차원 ndarray
>>> c = a + b                  # 1차원 ndarray의 뒷셈
>>> c
array([5, 7, 9])
```

대화창 실습 : ndarray의 뒷셈(shape)이 다를 경우

```
>>> a = np.array([1, 2])        # 2개의 원소를 가지는 1차원 배열 : (2,) shape
>>> b = np.array([4, 5, 6])     # 3개의 원소를 가지는 1차원 배열 : (3,) shape
>>> c = a + b                  # shape이 다른 1차원 배열의 합
...
ValueError: operands could not be broadcast together with shapes (2,) (3,)
```

대화창 실습 : 2차원 ndarray의 사칙연산

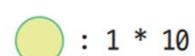
```
>>> a = np.array([[1, 2], [3, 4]])    # 2차원 배열 a
>>> b = np.array([[10, 20], [30, 40]]) # 2차원 배열 b
>>> a + b
array([[11, 22],
       [33, 44]])
>>> a - b
array([[ -9, -18],
       [-27, -36]])
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix} = \begin{bmatrix} 1*10 & 2*20 \\ 3*30 & 4*40 \end{bmatrix}$$

```
>>> a * b
array([[ 10,  40],
       [ 90, 160]])
>>> a / b
array([[ 0.1,  0.1],
       [ 0.1,  0.1]])
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix} = \begin{bmatrix} 10 & 40 \\ 90 & 160 \end{bmatrix}$$

a \* b



[그림 11-4] 넘파이의 a \* b 연산

### 3) ndarray 연산

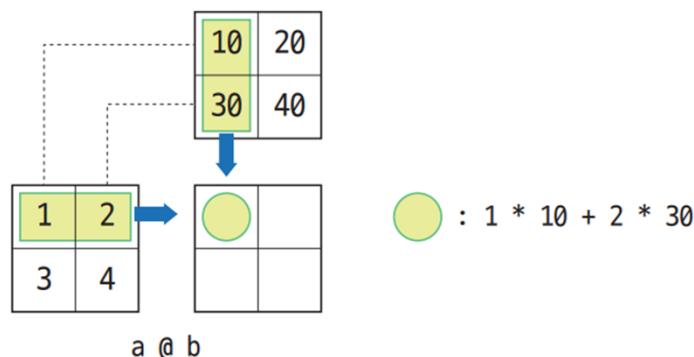
대화창 실습 : 행렬 곱 함수 `matmul()` 실습

```
>>> np.matmul(a, b)
array([[ 70, 100],
       [150, 220]])

>>> a @ b
array([[ 70, 100],
       [150, 220]])

>>> a[0,0] * b[0,0] + a[0,1] * b[1,0]
70
>>> a[0,0] * b[0,1] + a[0,1] * b[1,1]
100
>>> a[1,0] * b[0,0] + a[1,1] * b[1,0]
150
>>> a[1,0] * b[0,1] + a[1,1] * b[1,1]
220
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} @ \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix} = \begin{bmatrix} 1*10+2*30 & 1*20+2*40 \\ 3*10+4*30 & 3*20+4*40 \end{bmatrix}$$



[그림 11-5] 넘파이의  $a @ b$  연산

대화창 실습 : 단위행렬에 대한 행렬 곱

```
>>> a = [[1, 2], [3, 4]]
>>> b = [[1, 0], [0, 1]] # b는 단위행렬
>>> np.matmul(a, b)      # a 행렬과 단위행렬 b의 곱의 결과
array([[ 1,  2],
       [ 3,  4]])
```

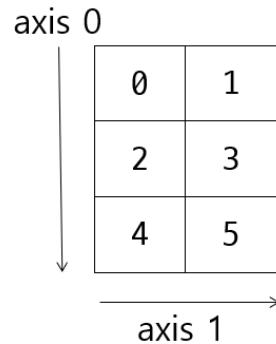
대화창 실습 : 2차원 ndarray의 덧셈, 뺄셈, 곱셈, 나눗셈, 제곱 연산

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a + 1                  # 행렬의 각 성분에 대한 덧셈
array([[2, 3],
       [4, 5]])
>>> a - 1                  # 행렬의 각 성분에 대한 뺄셈
array([-1, 1],
      [-2, 3])
>>> a * 100                # 행렬의 각 성분에 대한 곱셈
array([[100, 200],
       [300, 400]])
>>> a / 100                # 행렬의 각 성분에 대한 나눗셈
array([[0.01, 0.02],
       [0.03, 0.04]])
>>> a ** 2                  # 행렬의 각 성분에 대한 제곱연산
array([[ 1,  4],
       [ 9, 16]])
```

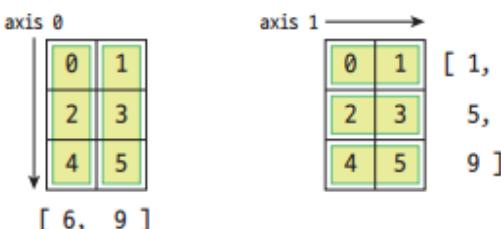
## ndarray 객체 배열의 축

[출처] 유틸 파이썬, “11장 넘파이”

### 6) 다차원 배열의 축



[그림 14-6] 2차원 배열과 축 1과 축 2의 방향



[그림 11-11] axis 0과 axis 1에 대하여 각각 sum() 함수를 수행한 결과

```
a = np.arange(6).reshape(3, 2)  
a
```

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

```
a.sum() # 행렬의 모든 원소의 합
```

```
15
```

```
a.sum(axis = 0) # 0축 방향(행 방향) 원소의 합
```

```
array([6, 9])
```

```
a.sum(axis = 1) # 1축 방향(행 방향) 원소의 합
```

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

```
a.min(axis = 0) # 0축 방향 원소의 최솟값
```

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

```
a.min(axis = 1) # 1축 방향 원소의 최솟값
```

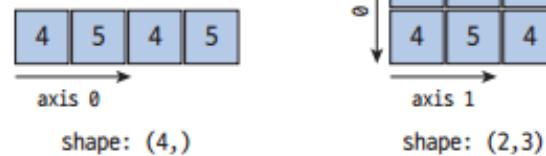
```
array([0, 2, 4])
```

```
a.max(axis = 0) # 0축 방향 원소의 최댓값
```

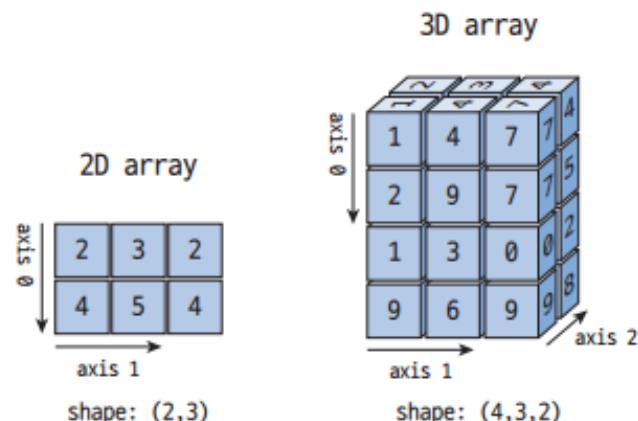
```
array([4, 5])
```

```
a.max(axis = 1) # 1축 방향 원소의 최댓값
```

```
array([1, 3, 5])
```

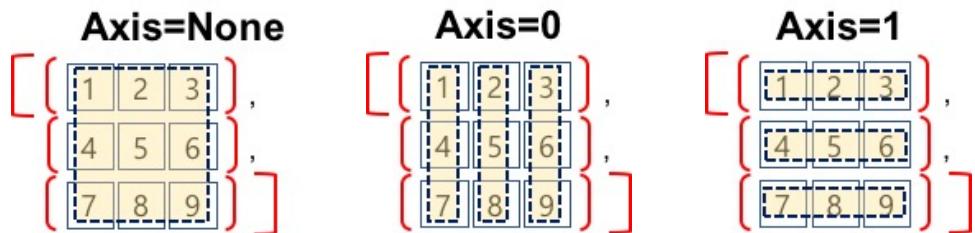


[그림 11-12] 1차원, 2차원, 3차원 배열의 형태와 각 배열의 축 방향



## Aggregate Functions (집계 함수)

- NumPy의 모든 집계 함수는 axis를 기준으로 계산됩니다. 집계함수에 axis를 지정하지 않으면 axis=None입니다. axis=None, 0, 1은 다음과 같은 기준으로 생각할 수 있습니다.



- axis = None : 전체 행렬을 하나의 배열로 간주하고 집계 함수의 범위를 전체 행렬로 정의합니다.
- axis=0 : 행을 기준으로 각 행의 동일 인덱스의 요소를 그룹으로 하고, 각 그룹을 집계 함수의 범위로 정의합니다.
- axis=1 : 열을 기준으로 각 열의 요소를 그룹으로 하고, 각 그룹을 집계 함수의 범위로 정의합니다.

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

※ pprint() : 23,81 페이지 정의

```
# orange로 1부터 100만의 범위에서 1씩 증가하는 배열 생성
# 배열의 shape을 (3, 3)으로 지정
a = np.arange(1, 10).reshape(3, 3)
pprint(a)

shape: (3, 3), dimension: 2, dtype:int64
Array's Data
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

[ndarray 배열 객체].sum(), np.sum(): 합계

- 지정된 axis를 기준으로 요소의 합을 반환합니다.

```
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 객체 배열의 축

```
# arange로 1부터 10이만의 범위에서 1씩 증가하는 배열 생성
# 배열의 shape를 (3, 3)으로 지정
a = np.arange(1, 10).reshape(3, 3)
pprint(a)
```

```
shape: (3, 3), dimension: 2, dtype:int64
Array's Data
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

### [ndarray 배열 객체].min(), np.min(): 최소값 (np.max() : 최대값)

- 지정된 axis를 기준으로 요소의 최소값을 반환합니다.

```
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]))
```

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

### [ndarray 배열 객체].cumsum(), np.cumsum(): 누적 합계 (Cumulative sum)

- 지정된 axis를 기준으로 각 요소의 누적 합의 결과를 반환합니다.

```
a.cumsum(), np.cumsum(a)
```

```
(array([ 1,  3,  6, 10, 15, 21, 28, 36, 45]),
 array([ 1,  3,  6, 10, 15, 21, 28, 36, 45]))
```

```
a.cumsum(axis=0), np.cumsum(a, axis=0)
```

```
(array([[ 1,  2,  3],
        [ 5,  7,  9],
        [12, 15, 18]]), array([[ 1,  2,  3],
        [ 5,  7,  9],
        [12, 15, 18]]))
```

```
a.cumsum(axis=1), np.cumsum(a, axis=1)
```

```
(array([[ 1,  3,  6],
        [ 4,  9, 15],
        [ 7, 15, 24]]), array([[ 1,  3,  6],
        [ 4,  9, 15],
        [ 7, 15, 24]]))
```

### [ndarray 배열 객체].mean(), np.mean(): 평균

np.median(): 중앙값

np.corrcoef(): (상관계수)Correlation coefficient

[ndarray 배열 객체].std(), np.std(): 표준편차

- 배열의 요소를 변경, 추가, 삽입 및 삭제하는 `resize`, `append`, `insert`, `delete` 함수를 제공합니다.

## Resize()

- `np.resize(a, new_shape)`
- `np.ndarray.resize(new_shape, refcheck=True)`
- 배열의 `shape`과 크기를 변경합니다.
- np.resize와 np.reshape 함수는 배열의 `shape`을 변경한다는 부분에서 유사합니다. 차이점은 `reshape` 함수는 배열 요소 수를 변경하지 않습니다. `reshape` 전후 배열의 요소 수는 같습니다. 반면에 `resize`는 `shape`을 변경하는 과정에서 배열 요소 수를 줄이거나 늘립니다.
- 일반적인 `resize` 사용 방법

※ `pprint()`: 23,81 페이지 정의

```
# 배열 생성
a = np.random.randint(1, 10, (2, 6))
pprint(a)
```

```
type:<class 'numpy.ndarray'>
shape: (2, 6), dimension: 2, dtype:int64
Array's Data:
[[1 5 4 2 7 4]
 [4 8 4 4 9 9]]
```

```
# shape 변경 - 요소 수 변경 없음
a.resize((6, 2))
pprint(a)
```

```
type:<class 'numpy.ndarray'>
shape: (6, 2), dimension: 2, dtype:int64
Array's Data:
[[1 5]
 [4 2]
 [7 4]
 [4 8]
 [4 4]
 [9 9]]
```

- 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:int64
Array's Data:
[[5 1 8 4 5 3]
 [2 8 9 2 2 6]]
```

```
# 요소수 12개에서 20개로 늘어남
# 늘어난 요소는 0으로 채워짐
a.resize((2, 10))
pprint(a)
```

```
type:<class 'numpy.ndarray'>
shape: (2, 10), dimension: 2, dtype:int64
Array's Data:
[[5 1 8 4 5 3 2 8 9 2]
 [2 6 0 0 0 0 0 0 0 0]]
```

## 배열의 요소 추가/삭제

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

- 배열의 요소를 변경, 추가, 삽입 및 삭제하는 `resize`, `append`, `insert`, `delete` 함수를 제공합니다.

### Resize()

- `np.resize(a, new_shape)`
  - `np.ndarray.resize(new_shape, refcheck=True)`
  - 배열의 shape과 크기를 변경합니다.
- 
- 요소 수가 줄어드는 변경

```
# 배열 생성
a = np.random.randint(1, 10, (2, 6))
pprint(a)
```

※ pprint(): 23,81 페이지 정의

```
type:<class 'numpy.ndarray'>
shape: (2, 6), dimension: 2, dtype:int64
Array's Data:
[[7 4 5 9 8 6]
 [9 7 5 1 3 1]]
```

```
# 요소수 12개에서 9개로 줄임
# 이전 데이터 삭제
a.resize((3, 3))
pprint(a)
```

```
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int64
Array's Data:
[[7 4 5]
 [9 8 6]
 [9 7 5]]
```

### append()

- `np.append(arr, values, axis=None)`
- 배열의 끝에 값을 추가
- `np.append` 함수는 `arr`의 끝에 `values`(배열)을 추가합니다. `axis`로 배열이 추가되는 방향을 지정할 수 있습니다.

```
# 데모 배열 생성
a = np.arange(1, 10).reshape(3, 3)
pprint(a)
b = np.arange(10, 19).reshape(3, 3)
pprint(b)
```

```
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int64
Array's Data:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int64
Array's Data:
[[10 11 12]
 [13 14 15]
 [16 17 18]]
```

- 뒷페이지 연계 코드

## 배열의 요소 추가/삭제

### append()

a	b
<code>[[1 2 3]</code>	<code>[[10 11 12]</code>
<code>[4 5 6]</code>	<code>[13 14 15]</code>
<code>[7 8 9]]</code>	<code>[16 17 18]]</code>

#### case 1: axis을 지정하지 않을 경우

- axis를 지정하지 않으면 배열은 1차원 배열로 변형되어 결합됩니다.

```
# axis 지정 없이 추가  
result = np.append(a, b)  
 pprint(result)
```

```
type:<class 'numpy.ndarray'>  
shape: (18,), dimension: 1, dtype:int64  
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 아니라 새로운 배열이 생성됩니다.  
 pprint(a)

```
type:<class 'numpy.ndarray'>  
shape: (3, 3), dimension: 2, dtype:int64  
Array's Data:  
[[1 2 3]  
[4 5 6]  
[7 8 9]]
```

※ pprint(): 23,81페이지 정의

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

#### case 2: axis=0 지정

- axis = 0 설정 시, shape[0]를 제외한 나머지 shape은 같아야 합니다.
- shape[0]를 제외한 나머지 shape이 다를 경우 append는 오류를 발생함

```
# 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:int64  
Array's Data:  
[[ 1  2  3]  
[ 4  5  6]  
[ 7  8  9]  
[10 11 12]  
[13 14 15]  
[16 17 18]]
```

```
different_sahpe_arr = np.arange(10, 20).reshape(2, 5)  
 pprint(different_sahpe_arr)
```

```
type:<class 'numpy.ndarray'>  
shape: (2, 5), dimension: 2, dtype:int64  
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)
```

```
ValueError: Traceback (most recent call last)  
<ipython-input-13-62a27e99a457> in <module>()  
      1 # 기준 축을 제외한 shape이 다른 배열의 append: 오류 발생  
----> 2 np.append(a, different_sahpe_arr, axis=0)  
/usr/local/lib/python3.5/dist-packages/numpy/lib/function_base.py  
in append(arr, values, axis)  
    5150     values = ravel(values)  
    5151     axis = arr.ndim-1  
-> 5152     return concatenate((arr, values), axis=axis)  
ValueError: all the input array dimensions except for the concatenation axis must match exactly
```

## 배열의 요소 추가/삭제

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

### case 3: axis=1 지정

- axis = 1 설정 시, shape[1]을 제외한 나머지 shape은 같아야 합니다.
- shape[1]을 제외한 나머지 shape이 다를 경우 append는 오류를 발생함

```
# axis = 1  
# axis 1 방향으로 b 배열 추가  
result = np.append(a, b, axis=1)  
pprint(result)
```

※ pprint(): 23,81 페이지 정의

```
type:<class 'numpy.ndarray'>  
shape: (3, 6), dimension: 2, dtype:int64  
Array's Data:  
[[ 1  2  3 10 11 12]  
[ 4  5  6 13 14 15]  
[ 7  8  9 16 17 18]]
```

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

```
# shape[0] 다른 배열 생성  
different_shape_arr = np.arange(10, 20).reshape(5, 2)  
pprint(different_shape_arr)
```

```
type:<class 'numpy.ndarray'>  
shape: (5, 2), dimension: 2, dtype:int64  
Array's Data:  
[[10 11]  
[12 13]  
[14 15]  
[16 17]  
[18 19]]
```

```
# 기준 축을 제외한 shape[0] 다른 배열의 append: 오류 발생  
np.append(a, different_shape_arr, axis=1)
```

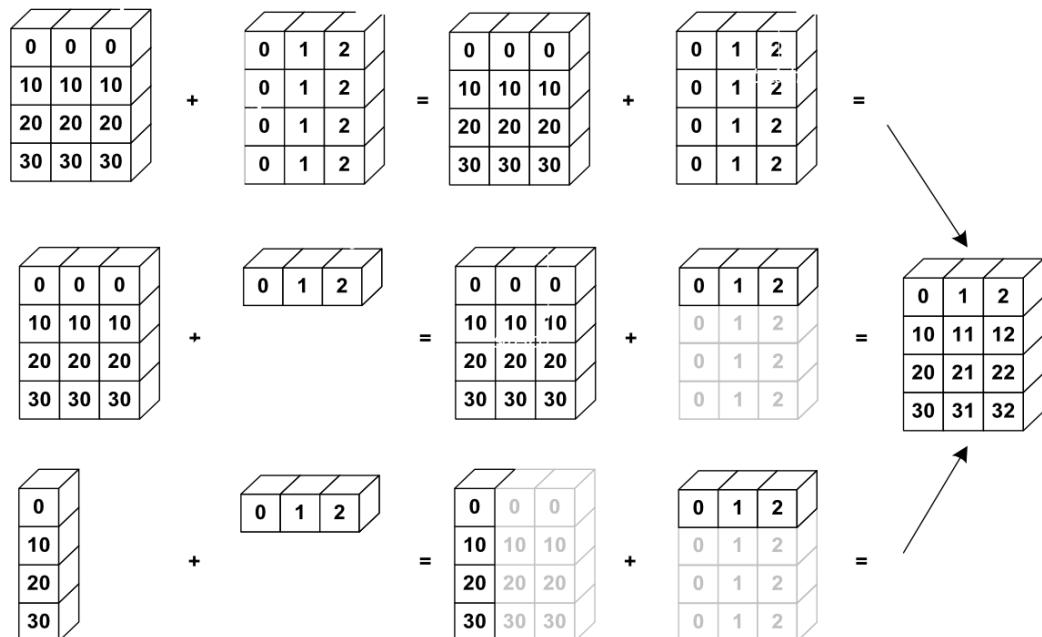
```
ValueError: Traceback (most recent call last)  
<ipython-input-16-c3bfec5cee97> in <module>()  
      1 # 기준 축을 제외한 shape[0] 다른 배열의 append: 오류 발생  
----> 2 np.append(a, different_shape_arr, axis=1)  
/usr/local/lib/python3.5/dist-packages/numpy/lib/function_base.py  
in append(arr, values, axis)  
    5150         values = ravel(values)  
    5151         axis = arr.ndim-1  
-> 5152     return concatenate((arr, values), axis=axis)  
ValueError: all the input array dimensions except for the concatenation axis must match exactly
```

## Broadcasting (브로드캐스팅)

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

### ❖ 브로드캐스팅

- Shape이 같은 두 배열에 대한 이항 연산은 배열의 요소별로 수행됩니다.  
두 배열 간의 Shape이 다를 경우 두 배열 간의 형상을 맞추는  
Broadcasting 과정을 거칩니다.



# 대모 배열 생성

```
a = np.arange(1, 25).reshape(4, 6)
 pprint(a)
 b = np.arange(25, 49).reshape(4, 6)
 pprint(b)
```

shape: (4, 6), dimension: 2, dtype:int64

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]]
```

shape: (4, 6), dimension: 2, dtype:int64

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]]
```

### Shape이 같은 배열의 이항 연산

- Shape이 같은 두 배열을 이항 연산 할 경우 위치가 같은 요소 단위로 수행됨

a+b

```
array([[ 26,  28,  30,  32,  34,  36],
       [ 38,  40,  42,  44,  46,  48],
       [ 50,  52,  54,  56,  58,  60],
       [ 62,  64,  66,  68,  70,  72]])
```

## Broadcasting (브로드캐스팅)

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

### Shape0 다른 배열의 이항 연산

- Shape0 다른 두 배열 사이의 이항 연산에서 **브로드캐스팅** 발생
- 두 배열을 같은 Shape으로 만든 후 연산을 수행

#### ➤ Case 1: 배열과 스칼라

- 배열과 스칼라 사이의 이항 연산 시 스칼라를 배열로 변형합니다.

```
# 데모 배열 생성
a = np.arange(1, 25).reshape(4, 6)
pprint(a)
```

※ pprint(): 23,81 페이지 정의

```
shape: (4, 6), dimension: 2, dtype:int64
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 처리됨

```
# step 1: 스칼라 배열 변경
new_arr = np.full_like(a, 100)
pprint(new_arr)
```

```
shape: (4, 6), dimension: 2, dtype:int64
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]])
```

#### ➤ Case 2: Shape0 다른 배열들의 연산

# 데모 배열 생성

```
a = np.arange(5).reshape((1, 5))
pprint(a)
b = np.arange(5).reshape((5, 1))
pprint(b)
```

```
shape: (1, 5), dimension: 2, dtype:int64
Array's Data
[[0 1 2 3 4]]
shape: (5, 1), dimension: 2, dtype:int64
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]])
```

## 인덱싱과 슬라이싱

### ❖ 슬라이싱

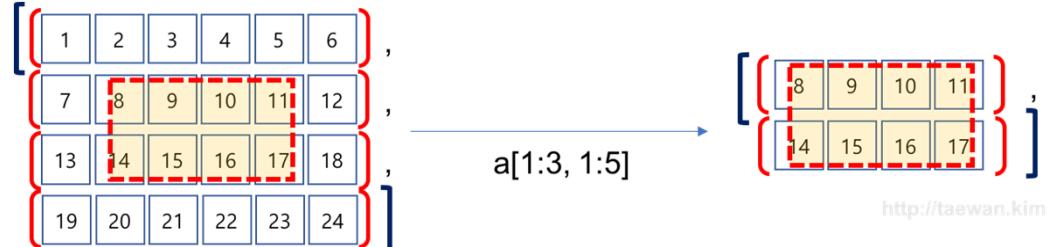
- 여러개의 배열 요소를 참조할 때 슬라이싱을 사용합니다. 슬라이싱은 axis 별로 범위를 지정하여 실행합니다. 범위는 `from_index : to_index` 형태로 지정합니다. `from_index`는 범위의 시작 인덱스이며, `to_index`는 범위의 종료 인덱스입니다. 요소 범위를 지정할 때 `to_index`는 결과에 포함되지 않습니다.
- `from_index : to_index`의 범위 지정에서 `from_index`는 생략 가능합니다. 생략 할 경우 0을 지정한 것으로 간주합니다. `to_index` 역시 생략 가능합니다. 이 경우 마지막 인덱스로 설정됩니다. 따라서 ":" 형태로 지정된 범위는 전체 범위를 의미합니다.
- `from_index`와 `to_index`에 음수를 지정하면 이것은 반대 방향을 의미합니다. 예를 들어서 -1은 마지막 인덱스를 의미합니다.
- 슬라이싱은 원본 배열의 뷰입니다. 따라서 슬라이싱 결과의 요소를 업데이트 하면 원본에 반영됩니다.

```
# 대모 배열 생성
a1 = np.arange(1, 25).reshape((4, 6)) #2차원 배열
pprint(a1)

shape: (4, 6), dimension: 2, dtype:int64
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]]
```

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

### ❖ 가운데 요소 가져오기



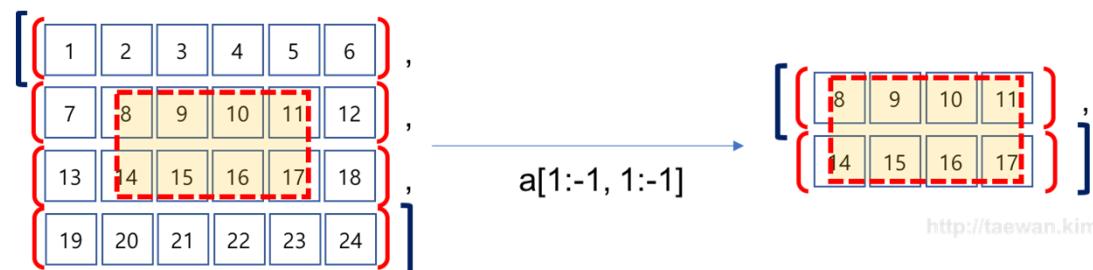
<http://taewan.kim>

a1[1:3, 1:5]

```
array([[ 8,  9, 10, 11],
       [14, 15, 16, 17]])
```

### • 음수 인덱스를 이용한 범위 설정

- ✓ 음수 인덱스는 지정한 axis의 마지막 요소로 부터 반대 방향의 인덱스입니다.
- ✓ -1은 마지막 요소의 인덱스를 의미합니다.
- ✓ 다음 슬라이싱은 위 슬라이싱과 동일한 결과를 만듭니다.



<http://taewan.kim>

a1[1:-1, 1:-1]

```
array([[ 8,  9, 10, 11],
       [14, 15, 16, 17]])
```

## 인덱싱과 슬라이싱

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

### ❖ 슬라이싱 업데이트

```
# 대문 대상 배열 조회  
pprint(a1)
```

```
shape: (4, 6), dimension: 2, dtype:int64  
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)
```

```
shape: (2, 4), dimension: 2, dtype:int64  
Array's Data  
[[ 8  9 10 11]  
[14 15 16 17]]
```

```
# 슬라이싱 결과 배열에 슬라이싱을 적용하여 4개 요소 합조  
pprint(slide_arr[:, 1:3])
```

```
type:<class 'numpy.ndarray'>  
shape: (2, 2), dimension: 2, dtype:int32  
Array's Data:  
[[ 9 10]  
[15 16]]
```

```
# 슬라이싱을 적용하여 합조한 4개 요소 업데이트 및 슬라이싱 배열 조회  
slide_arr[:, 1:3]=99999  
pprint(slide_arr)
```

```
shape: (2, 4), dimension: 2, dtype:int64  
Array's Data  
[[      8 99999 99999      11]  
[     14 99999 99999      17]]
```

```
# 원본 배열에 반영된 결과 확인  
pprint(a1)
```

```
shape: (4, 6), dimension: 2, dtype:int64  
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]]
```

## 추가 자료

- **ndarray** 객체 배열의 축
- 배열 복사와 정렬
- **NumPy** 입출력

## ndarray 객체 배열의 축

### 다차원 배열의 축

#### numpy.append()

```
numpy.append(arr, values, axis=None)
```

Append values to the end of an array.

```
>>> np.append([1, 2, 3], [[4, 5, 6], [7, 8, 9]])  
array([1, 2, 3, ..., 7, 8, 9])
```

When *axis* is specified, *values* must have the correct shape.

```
>>> np.append([[1, 2, 3], [4, 5, 6]], [[7, 8, 9]], axis=0)  
array([[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9]])  
>>> np.append([[1, 2, 3], [4, 5, 6]], [7, 8, 9], axis=0)  
Traceback (most recent call last):  
...  
ValueError: all the input arrays must have same number of dimensions, but  
the array at index 0 has 2 dimension(s) and the array at index 1 has 1  
dimension(s)
```

#### numpy.insert()

```
numpy.insert(arr, obj, values, axis=None)
```

Insert values along the given axis before the given indices.

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

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

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

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

```
np.insert(a, 1, 4, axis = 1)
```

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

*obj* : int, slice or sequence of ints

Object that defines the index or indices before which values is inserted.

## ndarray 객체 배열의 축

### 다차원 배열의 축

numpy.insert()

numpy.insert(arr, obj, values, axis=None)

Insert values along the given axis before the given indices.

```
>>> a = np.array([[1, 1], [2, 2], [3, 3]])
>>> a
array([[1, 1],
       [2, 2],
       [3, 3]])
>>> np.insert(a, 1, 5)
array([1, 5, 1, ..., 2, 3, 3])
>>> np.insert(a, 1, 5, axis=1)
array([[1, 5, 1],
       [2, 5, 2],
       [3, 5, 3]])
```

Difference between sequence and scalars:

```
>>> np.insert(a, [1], [[1],[2],[3]], axis=1)
array([[1, 1, 1],
       [2, 2, 2],
       [3, 3, 3]])
>>> np.array_equal(np.insert(a, 1, [1, 2, 3], axis=1),
...                  np.insert(a, [1], [[1],[2],[3]], axis=1))
True
```

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

```
>>> np.insert(b, slice(2, 4), [5, 6])
array([1, 1, 5, ..., 2, 3, 3])
```

```
>>> np.insert(b, [2, 2], [7.13, False]) # type casting
array([1, 1, 7, ..., 2, 3, 3])
```

```
>>> x = np.arange(8).reshape(2, 4)
>>> idx = (1, 3)
>>> np.insert(x, idx, 999, axis=1)
array([[ 0, 999,   1,   2, 999,   3],
       [ 4, 999,   5,   6, 999,   7]])
```

## ndarray 객체 배열의 축

### numpy.insert()

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

```
# 대모 배열 생성
a = np.arange(1, 10).reshape(3, 3)
pprint(a)
```

```
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int64
Array's Data:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
# a 배열을 일차원 배열로 변환하고 1번 index에 999 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
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]])
```

### numpy.delete()

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

```
# 대모 배열 생성
a = np.arange(1, 10).reshape(3, 3)
pprint(a)
```

```
type:<class 'numpy.ndarray'>
shape: (3, 3), dimension: 2, dtype:int64
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])
```

```
# 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]])
```

## ❖ 배열 복사

- ndarray 배열 객체에 대한 slice, subset, indexing이 반환하는 배열은 새로운 객체가 아닌 기존 배열의 뷔임.
- 반환한 배열의 값을 변경하면 원본 배열에 반영됨. 따라서 기본 배열로부터 새로운 배열을 생성하기 위해서는 copy 함수로 명시적으로 사용해야 함. copy 함수로 복사된 원본 배열과 사본 배열은 완전히 다른 별도의 객체임.

### [ndarray 배열 객체].copy(), np.copy()

```
#데모용 배열
a = np.random.randint(0, 9, (3, 3))
pprint(a)
```

```
shape: (3, 3), dimension: 2, dtype:int64
Array's Data
[[1 8 2]
 [3 1 7]
 [2 8 4]]
```

```
copied_a1 = np.copy(a)
```

```
# 복사된 배열의 요소 업데이트
copied_a1[:, 0]=0 #배열의 전체 row의 첫번째 요소 0으로 업데이트
pprint(copied_a1)
```

```
shape: (3, 3), dimension: 2, dtype:int64
Array's Data
[[0 8 2]
 [0 1 7]
 [0 8 4]]
```

```
# 복사본 배열 변경이 원본에 영향을 미치지 않음
pprint(a)
```

```
shape: (3, 3), dimension: 2, dtype:int64
Array's Data
[[1 8 2]
 [3 1 7]
 [2 8 4]]
```

```
#np.copy()를 이용한 복사
copied_a2 = np.copy(a)
pprint(copied_a2)
```

```
shape: (3, 3), dimension: 2, dtype:int64
Array's Data
[[1 8 2]
 [3 1 7]
 [2 8 4]]
```

## 배열 복사와 정렬

### ❖ 배열 정렬

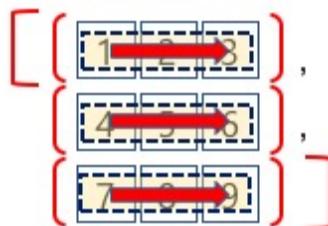
- ndarray 객체는 axis를 기준으로 요소 정렬하는 sort 함수를 제공함

```
#배열 생성
unsorted_arr = np.random.random((3, 3))
pprint(unsorted_arr)

shape: (3, 3), dimension: 2, dtype:float64
Array's Data
[[ 0.2850756   0.39471278  0.62599575]
 [ 0.36466771  0.2226558   0.78950711]
 [ 0.16414313  0.95836532  0.43830591]]
```

```
#데모를 위한 배열 복사
unsorted_arr1 = unsorted_arr.copy()
unsorted_arr2 = unsorted_arr.copy()
unsorted_arr3 = unsorted_arr.copy()
```

### Axis=-1



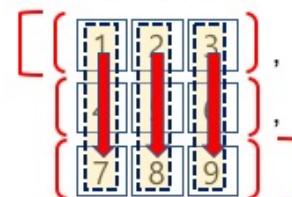
- [ndarray 객체].sort() axis의 기본 값은 -1입니다.
- axis=1은 현재 배열의 마지막 axis를 의미함.
- 현재 unsorted\_arr의 마지막 axis는 1입니다.
- [ndarray 객체].sort()와 [ndarray 객체].sort(axis=1)의 결과는 동일합니다.

```
#배열 정렬
unsorted_arr1.sort()
pprint(unsorted_arr1)
```

```
shape: (3, 3), dimension: 2, dtype:float64
Array's Data
[[ 0.2850756   0.39471278  0.62599575]
 [ 0.2226558   0.36466771  0.78950711]
 [ 0.16414313  0.43830591  0.95836532]]
```

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

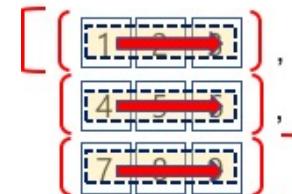
### Axis=0



```
#배열 정렬, axis=0
unsorted_arr2.sort(axis=0)
pprint(unsorted_arr2)
```

```
shape: (3, 3), dimension: 2, dtype:float64
Array's Data
[[ 0.16414313  0.2226558   0.43830591]
 [ 0.2850756   0.39471278  0.62599575]
 [ 0.36466771  0.95836532  0.78950711]]
```

### Axis=1



```
#배열 정렬, axis=1
unsorted_arr3.sort(axis=1)
pprint(unsorted_arr3)
```

```
shape: (3, 3), dimension: 2, dtype:float64
Array's Data
[[ 0.2850756   0.39471278  0.62599575]
 [ 0.2226558   0.36466771  0.78950711]
 [ 0.16414313  0.43830591  0.95836532]]
```

## 배열 결합과 분리

### ❖ 배열 결합

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

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

```
# 데모 배열
a = np.arange(1, 7).reshape((2, 3))
 pprint(a)
b = np.arange(7, 13).reshape((2, 3))
 pprint(b)
```

```
shape: (2, 3), dimension: 2, dtype:int64
Array's Data
[[1 2 3]
 [4 5 6]]
shape: (2, 3), dimension: 2, dtype:int64
Array's Data
[[ 7  8  9]
 [10 11 12]]
```

```
# 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]])
```

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

### 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)
b = np.arange(7, 13).reshape((2, 3))
 pprint(b)
```

```
shape: (2, 3), dimension: 2, dtype:int64
Array's Data
[[1 2 3]
 [4 5 6]]
shape: (2, 3), dimension: 2, dtype:int64
Array's Data
[[ 7  8  9]
 [10 11 12]]
```

```
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]])
```

### 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)
b = np.arange(7, 13).reshape((2, 3))
pprint(b)

shape: (2, 3), dimension: 2, dtype:int64
Array's Data
[[1 2 3]
 [4 5 6]]
shape: (2, 3), dimension: 2, dtype:int64
Array's Data
[[ 7  8  9]
 [10 11 12]]
```

```
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]])
```

## ❖ 배열 분리

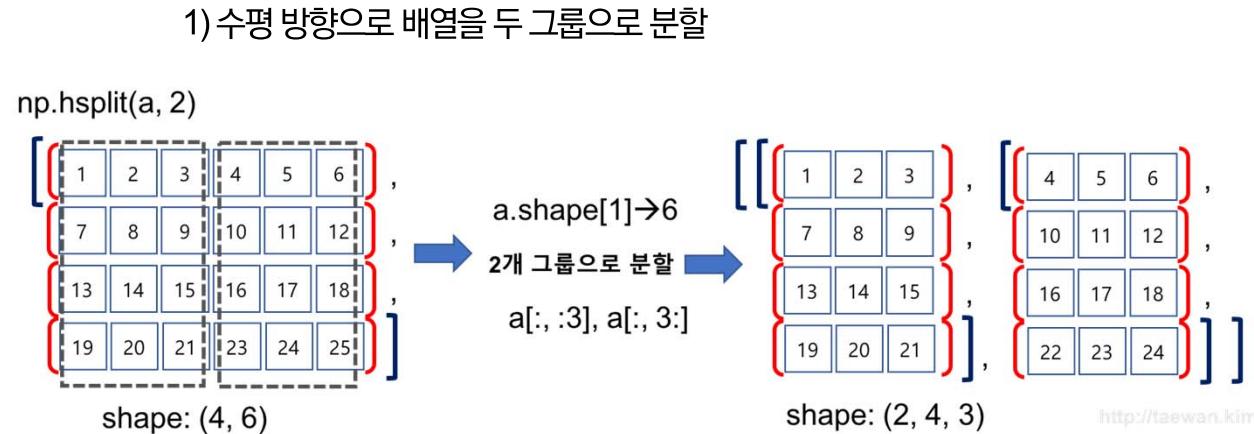
- NumPy는 배열을 수직, 수평으로 분할하는 함수를 제공합니다.
  - ✓ np.hsplit(): 지정한 배열을 수평(행) 방향으로 분할
  - ✓ np.vsplit(): 지정한 배열을 수직(열) 방향으로 분할

### 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)

shape: (4, 6), dimension: 2, dtype:int64
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]]
```



```
# 수평으로 두 그룹으로 분할하는 함수
result = np.hsplit(a, 2)
result

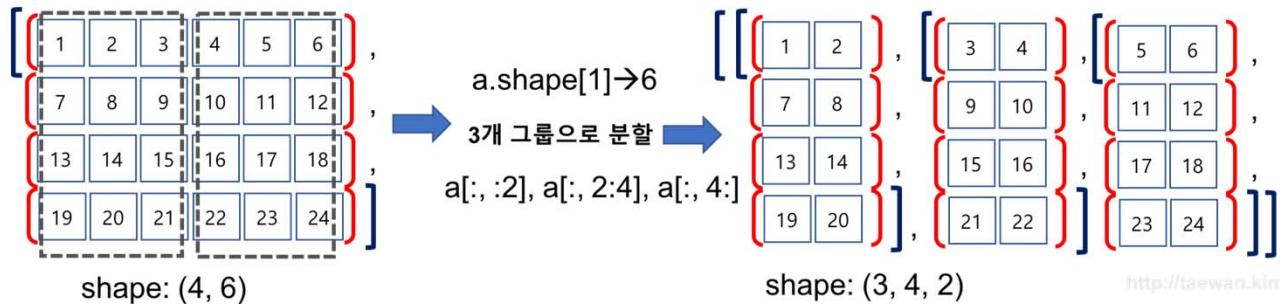
[array([[ 1,  2,  3],
       [ 7,  8,  9],
       [13, 14, 15],
       [19, 20, 21]]), array([[ 4,  5,  6],
       [10, 11, 12],
       [16, 17, 18],
       [22, 23, 24]])]
```

## 배열 결합과 분리

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

### 2) 수평 방향으로 배열을 세 그룹으로 분할

np.hsplit(a, 3)



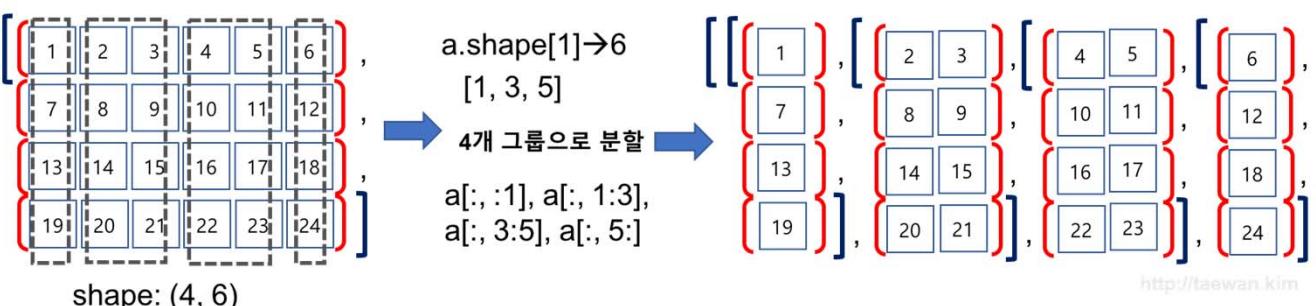
```
# 수평으로 두 그룹으로 분할하는 함수  
result = np.hsplit(a, 3)  
result
```

```
[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]])]
```

### 3) 수평 방향으로 여러 구간으로 구분

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

np.hsplit(a, [1, 3, 5])



```
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]])]
```

## 배열 수직 분할

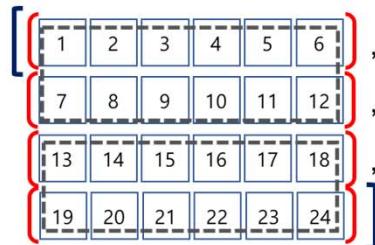
- np.vsplit(ary, indices\_or\_sections)
- 배열을 수직 방향(행 방향)으로 분할하는 함수

```
# 분할 대상 배열 생성
a = np.arange(1, 25).reshape((4, 6))
pprint(a)
```

```
shape: (4, 6), dimension: 2, dtype:int64
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]]
```

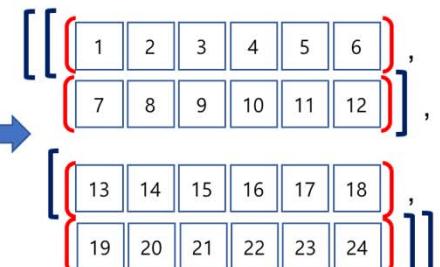
## 1) 수직방향으로 배열을 두 그룹으로 분할

np.vsplit(a, 2)



shape: (4, 6)

a.shape[0]→4

2개 그룹으로 분할  
a[:2], a[2:]shape: (2, 2, 6) <http://taewan.kim>

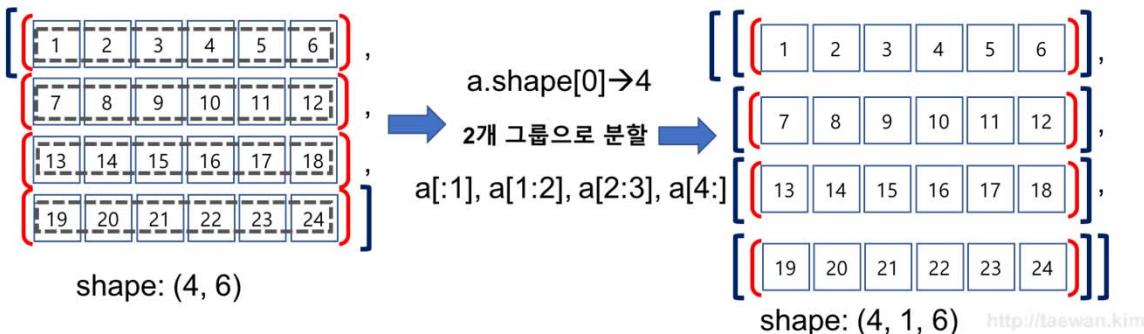
```
result=np.vsplit(a, 2)
result
```

```
[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
```

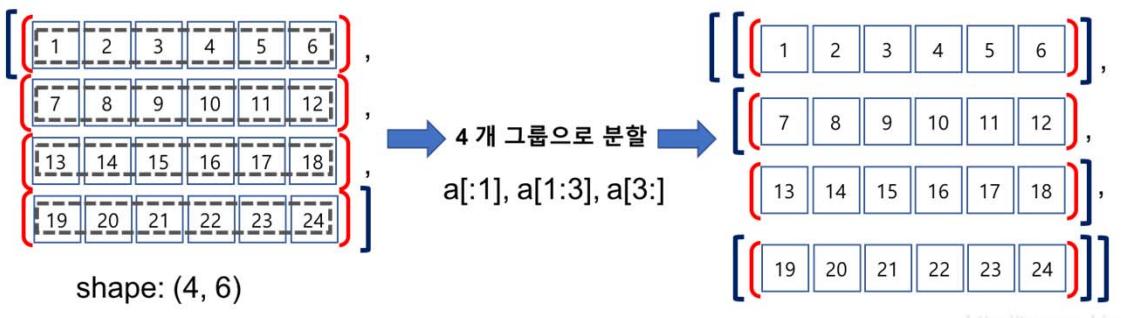
(2, 2, 6)

## 2) 수직 방향으로 배열을 4개 그룹으로 분할

`np.vsplit(a, 4)`

## 3) 수직 방향으로 여러 구간으로 구분

- `np.hsplit`의 두 번째 파라미터에 구간 설정 배열을 전달하여 여러 배열로 구분합니다.

`np.vsplit(a, [1, 3])`

```
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)
```

# row은 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]])]
```

## NumPy 입출력

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

함수명	기능	파일포맷
np.save()	NumPy 배열 객체 1개를 파일에 저장	바이너리
np.savez()	NumPy 배열 객체 복수개를 파일에 저장	바이너리
np.load()	NumPy 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 사용합니다.

```
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)

a = np.random.randint(0, 10, (2, 3))
b = np.random.randint(0, 10, (2, 3))
pprint(a)
pprint(b)
```

a = np.random.randint(0, 10, (2, 3))  
b = np.random.randint(0, 10, (2, 3))  
pprint(a)  
pprint(b)

type:<class 'numpy.ndarray'>  
shape: (2, 3), dimension: 2, dtype:int64  
Array's Data:  
[[8 7 4]  
[4 3 8]]  
type:<class 'numpy.ndarray'>  
shape: (2, 3), dimension: 2, dtype:int64  
Array's Data:  
[[6 9 8]  
[9 7 7]]

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

### ❖ 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: npz
- ✓ 배열 저장 파일은 **바이너리** 형태입니다.

#### • 1개 파일 저장

```
# a 배열 파일에 저장
np.save("./my_array1", a)
```

```
# 파일 조회
!ls -al my_array1*
-rw-r--r-- 1 root root 128 1월 17 2018 my_array1.npy
```

#### • 복수 배열을 1개 파일에 저장

```
# a, b 두 개 배열을 파일에 저장
np.savez("my_array2", a, b)
```

```
# 파일 조회
!ls -al my_array2*
-rw-r--r-- 1 root root 466 1월 17 2018 my_array2.npz
```

Jupyter lab : Magic Command로 실행

```
%ls my_array1*
```

## NumPy 입출력

### ❖ 파일로부터 배열 객체 로딩

- npy와 npz 파일은 np.load 함수로 읽을 수 있습니다.

```
# 1 // 배열 로딩  
np.load("./my_array1.npy")
```

```
array([[8, 7, 4],  
       [4, 3, 8]])
```

```
# 복수 파일 로딩  
npzfiles = np.load("./my_array2.npz")  
npzfiles.files  
['arr_0', 'arr_1']
```

```
npzfiles['arr_0']  
array([[8, 7, 4],  
       [4, 3, 8]])
```

```
npzfiles['arr_1']  
array([[6, 9, 8],  
       [9, 7, 7]])
```

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

### ❖ 텍스트 파일 로딩

- 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)

```
# 데이터 파일 위치  
!ls -al ./data/simple.csv
```

```
-rw-r--r-- 1 root root 15 1월 14 06:32 ./data/simple.csv
```

```
# 데이터 파일 내용  
!cat ./data/simple.csv
```

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

```
# 기본 데이터 파일은 float를 설정합니다.  
# 파일 데이터 배열로 로딩  
np.loadtxt("./data/simple.csv")
```

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

```
#dtype 속성으로 데이터 타입 변경 가능합니다.  
# 파일 데이터 배열로 로딩 및 데이터 타입 지정  
np.loadtxt("./data/simple.csv", dtype=np.int)
```

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

[출처] [http://taewan.kim/post/numpy\\_cheat\\_sheet/](http://taewan.kim/post/numpy_cheat_sheet/)

### ❖ 문자열을 포함하는 텍스트 파일 로딩

- 텍스트를 포함한 파일 경우 dtype으로 컬럼 명과 데이터 타입을 설정해야 함

```
# 대상 데이터 파일 조회  
!ls -al ./data/president_height.csv
```

```
-rw-r--r-- 1 root root 987 1월 13 07:04 ./data/president_height.c  
sv
```

```
# 데이터 파일 내용  
# 헤더 라인: 1개  
# 문자열 포함  
!head -n 3 ./data/president_height.csv
```

```
order,name,height(cm)  
1,George Washington,189  
2,John Adams,170
```

- president\_height.csv 파일은 숫자와 문자를 모두 포함하는 데이터 파일입니다.
- dtype을 이용하여 컬럼 타입을 지정하여 로딩합니다.
- delimiter와 skiprows를 이용하여 구분자와 무시해야 할 라인을 지정합니다.

```
# dtype와 dict 형식으로 데이터 파일 지정  
data = np.loadtxt("./data/president_height.csv", delimiter=",", skiprows=1, dtype={  
    'names': ("order", "name", "height"),  
    'formats': ('i', 'S20', 'f')  
})  
# 배열 데이터 출력  
data[:3]  
  
array([(1, b'George Washington', 189.), (2, b'John Adams', 170.),  
       (3, b'Thomas Jefferson', 189.)],  
      dtype=[('order', '<i4'), ('name', 'S20'), ('height', '<f4')])
```

### ❖ 배열 객체 텍스트 파일로 저장

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

- ✓ X : 1D or 2D array\_like,
- ✓ delimiter : String or character separating columns,
- ✓ header, footer : String that will be written at the beginning of the file or the end of the file.

fmt : str or sequence of strs, optional  
newline : String or character separating lines  
header, footer : String that will be written at the beginning of the file or the end of the file.

```
# 대모 데이터 생성
data = np.random.random((3, 4))
pprint(data)
```

```
type:<class 'numpy.ndarray'>
shape: (3, 4), dimension: 2, dtype:float64
Array's Data:
[[ 0.21554899  0.56103576  0.71822224  0.42060378]
 [ 0.59906291  0.51097642  0.37703684  0.48276954]
 [ 0.1889987   0.62604535  0.88074236  0.01603881]]
```

```
# 배열 객체 텍스트 파일로 저장
np.savetxt("./data/saved.csv", data, delimiter=",")
```

```
#파일 조회
!ls -al ./data/saved.csv
```

```
-rw-r--r-- 1 root root 300 1월 17 2018 ./data/saved.csv
```

```
#파일 내용 조회
!cat ./data/saved.csv
```

```
2.155489877688239186e-01,5.610357577617570701e-01,7.182222391102696113e-01,4.206037
807135534212e-01
5.990629079875264829e-01,5.109764156401283008e-01,3.770368358788609431e-01,4.827695
415663437739e-01
1.889986982230907886e-01,6.260453490415701649e-01,8.807423613788849526e-01,1.603880
803818769074e-02
```

```
# 데이터 파일 로딩
np.loadtxt('./data/saved.csv', delimiter=',')
```

```
array([[ 0.21554899,  0.56103576,  0.71822224,  0.42060378],
       [ 0.59906291,  0.51097642,  0.37703684,  0.48276954],
       [ 0.1889987 ,  0.62604535,  0.88074236,  0.01603881]])
```