

180618

아나콘다 설치하기

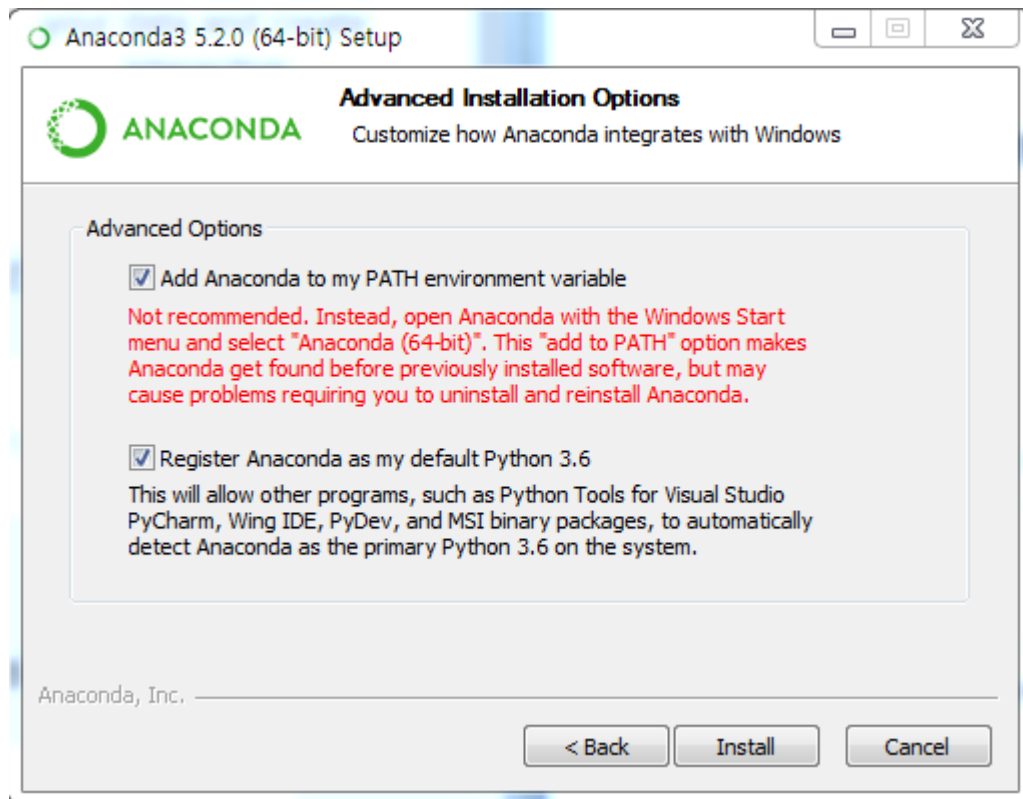
anaconda.com 에서 다운로드 받기 (64bit / 32bit 따라서)

원래는 파이썬 전에 설치하는게 좋음! 중복되는 파일이 있기도!

그래도 path 잘 설정하면 중복 설치해도 문제 없음

설치 진행 중 기존 파이썬이 설치되어있는 사람은 이 부분만 주의하기!

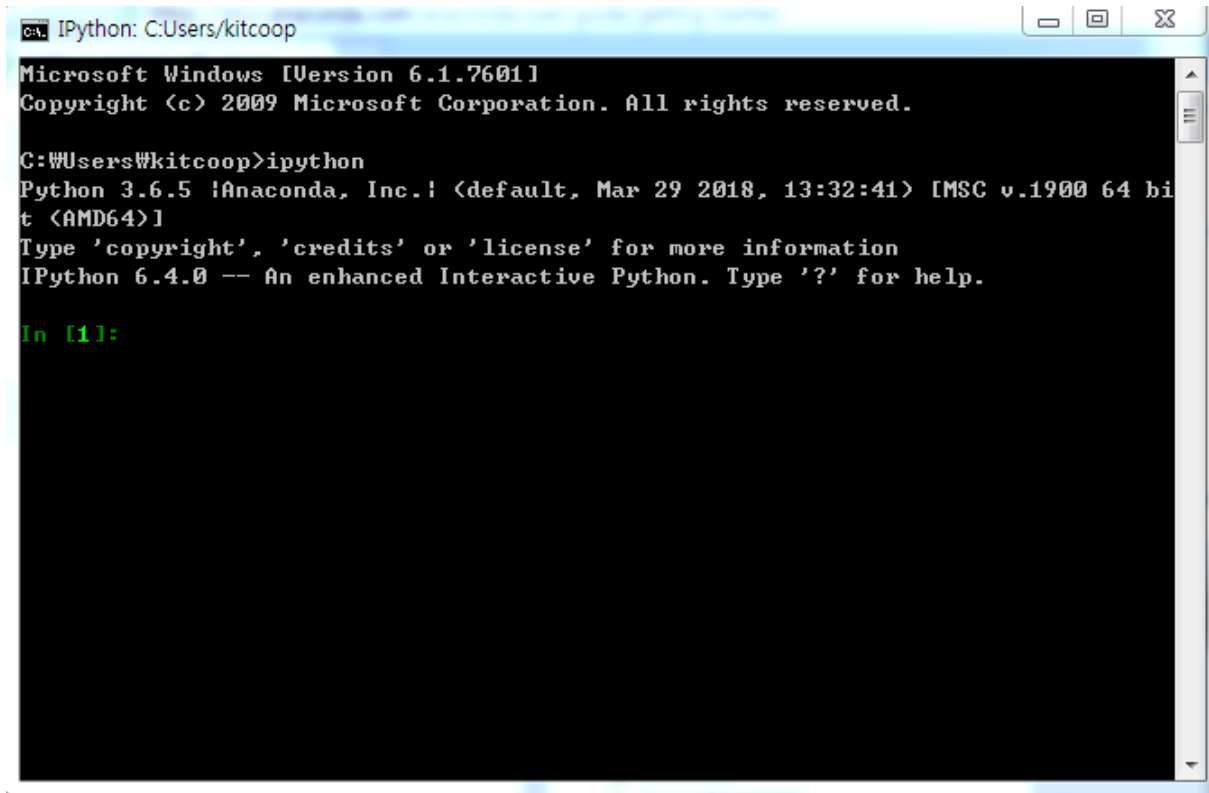
-> 두 개 다 체크



시작-실행-cmd-ipython 호출

cf. ipython notebook 를 cmd에서 실행하면 쥬피터 노트북 실행

-- 안 되면 리부팅 하기!



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\kitcoop>ipython
Python 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 6.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

default 디렉토리에 모듈 하나 만들어놓고 계속 호출하기!

ex. 자주쓰는 내부 모듈 -- profile.py로 저장

```
import numpy as np
import pandas as pd
from pandas import Series, DataFrame
from numpy import nan as NA
import matplotlib.pyplot as plt
# import os
# os.getcwd() #디폴트 디렉토리 위치 확인
# os.chdir("") #디폴트 디렉토리 변경 ~~ 아나콘다
```

cf. 디렉토리 원기호 ww 두 번 쓰기!

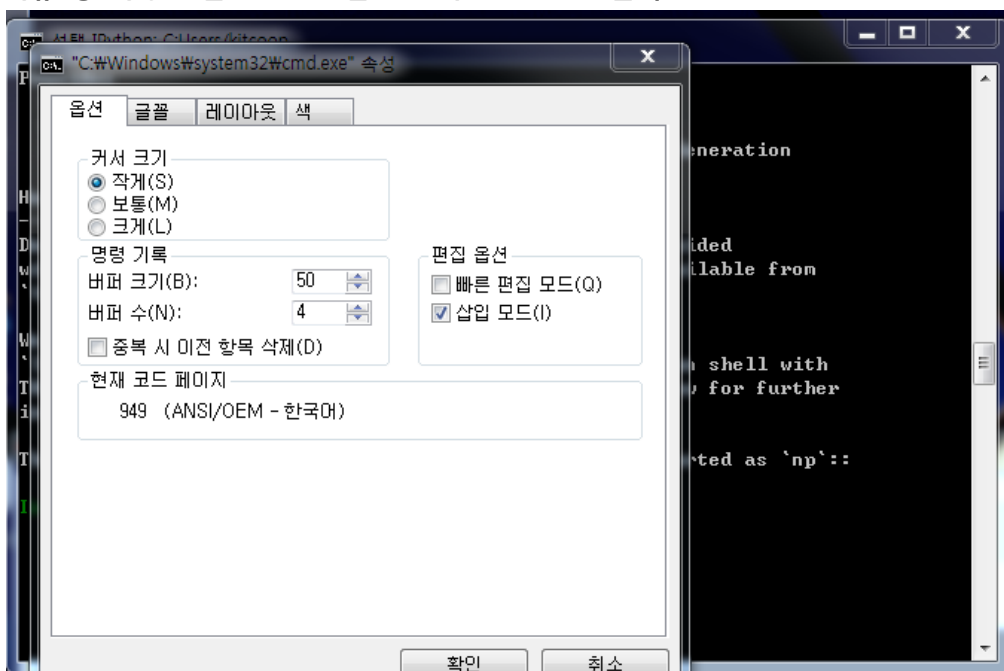
기타 -- 필요하면 추가



%를 안 해도 됨!

run profile만 해도 됨

메뉴 창 속성 - 옵션 - 편집옵션 - 빠른편집모드 클릭



드래그 + 오른쪽 마우스 클릭 -> ctrl+c 역할

교재 : 파이썬 라이브러리를 활용한 데이터 분석, 웨스 맥키니 지음

Chapter 4 NumPy 기본 : 배열과 벡터계산

NumPy

: Numerical Python의 줄임말 ~ 산술연산

: 고성능의 과학계산 컴퓨팅과 데이터 분석에 필요한 기본 패키지

NumPy 기능

- 빠르고 메모리를 효율적으로 사용하며, 벡터 산술연산과 세련된 브로드캐스팅 기능을 제공하는 다차원 배열인 ndarray

- 반복문을 작성할 필요 없이 전체 데이터 배열에 대해 빠른 연산을 제공하는 표준 수학 함수
- 배열 데이터를 디스크에 쓰거나 읽을 수 있는 도구와 메모리에 올려진 파일을 사용하는 도구
- 선형대수, 난수 발생기, 푸리에 변환 기능
- C, C++, 포트란으로 쓰여진 코드를 통합하는 도구

cf. ndarray - 단 하나의 데이터 타입 사용!

NumPy에서 중요하게 사용되는 기능

- 벡터 배열상에서 데이터 개조, 정제, 부분집합, 필터링, 변형, 다른 종류 연산의 빠른 수행
- 일반적인 배열 처리 알고리즘(정렬, 고유 원소 찾기, 집합연산 등)
- 데이터의 수집 및 요약과 통계의 효과적 표현
- 데이터 간의 관계조작(다른 종류의 데이터 묶음을 병합하고 엮기)
- 조건문을 포함한 반복문 대신 사용할 수 있는 조건절을 표현할 수 있는 배열 표현
- 데이터 그룹 전체에 적용할 수 있는 수집, 변형, 함수 적용 같은 데이터 처리

→ 데이터 분석을 할 때 사용되는 pandas, matplotlib의 기반으로 사용

(NumPy자체는 고수준 데이터 분석기능을 제공 X)

4.1 NumPy ndarray : 다차원 배열 객체

ndarray

- N차원의 배열 객체
- 파이썬에서 대규모 데이터 집합을 담을 수 있는 빠르고 유연한 자료 구조
- 전체 데이터 블록에 수학적 연산 수행 할 수 있음

np.array() 생성 예시

```
In [3]: data=np.random.randn(2, 3)
```

2행, 3열 형태로 출력

randn : 평균이 0이고 표준편차가 1인 난수 생성

```
In [4]: data
```

```
Out[4]:
```

```
array([[ 0.29746833, -1.506904 , -1.33678286],  
       [ 0.10452144, -0.83368878, -1.06133133]])
```

```
In [5]: data*10
```

```
Out[5]:
```

```
array([[ 2.97468326, -15.06904001, -13.3678286 ],  
       [ 1.0452144 , -8.33688775, -10.61331328]])
```

```
In [6]: data+data # 벡터연산 가능 like R
```

```
Out[6]:
```

```
array([[ 0.59493665, -3.013808 , -2.67356572],  
       [ 0.20904288, -1.66737755, -2.12266266]])
```

```
In [7]: data.shape
```

```
Out[7]: (2, 3)
```

shape : 차원의 크기를 알려줌

```
In [8]: data.dtype
```

```
Out[8]: dtype('float64')
```

dtype : 배열에 저장된 자료형을 알려줌

4.1.1 ndarray 생성

array 함수

- 순차적인 객체를 받아 넘겨받은 데이터가 들어있는 새로운 NumPy 배열을 생성함

dtype= np.int / np.float 등등 옵션을 통해 데이터 타입 지정 가능

array()

In [9]: **data1=[6,7.5,8,0,1]**

In [10]: **arr1=np.array(data1)**

In [11]: **arr1**

Out[11]: array([6. , 7.5, 8. , 0. , 1.])

In [12]: **data2=[[1,2,3,4],[5,6,7,8]]**

같은 길이의 리스트가 담겨있는 순차 데이터는 다차원 배열로 변환 가능

In [13]: **arr2=np.array(data2)**

In [14]: **arr2**

Out[14]:

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

In [15]: **arr2.ndim**

Out[15]: 2

#ndim : arr2의 차원 (행*열 -> 2차원)

In [16]: **arr2.shape**

Out[16]: (2, 4)

In [17]: **arr1.dtype**

Out[17]: dtype('float64')

In [18]: **arr2.dtype**

Out[18]: dtype('int32')

명시적으로 지정하지 않는 한, np.array()는 생성될 때 적절한 자료형을 추정함.

-> dtype 객체에 저장됨

zeros(), ones()

In [19]: **np.zeros(10)**

Out[19]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [20]: **np.zeros((3,6))**

Out[20]:

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

zeros나 ones는 주어진 길이나 모양에 각각 0 / 1이 들어있는 **배열**을 생성한다.

2차원인 이상인 경우 괄호 두 번 쓰는 것 생각하기!

np.zeros()/np.ones()에 들어가는 차원(, ...) 나열이기 때문!

empty

- empty : 주어진 크기의 초기화되지 않은 배열 생성 ~ 의미없는 그냥 난수
- > *0으로 0으로 만들거나 등등에 사용됨. (단순히 비어있는 채로 생성은 불가능)

In [23]: **np.empty((2,3,2))**

Out[23]:

```
array([[[7.e-322, 0.e+000],
        [0.e+000, 0.e+000],
        [0.e+000, 0.e+000]],
       [[0.e+000, 0.e+000],
        [0.e+000, 0.e+000],
        [0.e+000, 0.e+000]]])
```

2차원인 이상인 경우 괄호 두 번 쓰는 것 생각하기!

np.empty(**(2,3,2)**) - 2층 **3행 2열** (vs. R **2행 3열** 2층)

-> **python**은 차원이 커질수록 **앞으로 쌓임**)

cf.

똑같이.t() / transpose()로 층/행 전환 가능, 따라서 행/열 입력 순서 잘 외우기

층/행/열 in python (cf. 행/열/층 in R)

arange() - range 함수의 배열버전

arange([start,] stop[, step], dtype=None) - stop말고는 생략가능

In [35]: **np.arange(15)**

Out[35]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])

-----참고

```
>>> np.arange(3)
```

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

```
>>> np.arange(3.0)
```

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

```
>>> np.arange(3,7)
```

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

```
>>> np.arange(3,7,2)
```

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

? help

사용 방법은 help처럼 마지막에 ? 사용 후 엔터

함수	설명
array	기본적으로 deep copy(메모리 복사) 수행 [기존 메모리 영역과 분리됨]
asarray	얕은 복사 . 데이터 수정 시 연결된 데이터도 수정됨.
arange	range와 유사, ndarray를 반환 (not list)
ones, ones_like	ones - 차원을 입력하면 입력한 모양의 내용을 모두 1로 초기화하는 배열 생성 ones_like - 기존의 데이터(리스트, 배열 등)을 입력하면 모양이 같은 배열을 새로 생성하여 내용을 모두 1로 초기화한다.
zeros, zero_like	ones, ones_like와 같지만 내용을 0으로 채운다.
empty, empty_like	메모리를 할당하여 새로운 배열을 생성하지만 ones/zeros처럼 값을 초기화하지는 않는다.
eye, identity	NxN 크기의 단위행렬을 생성한다. (대각선 - 1, 나머지 - 0)

cf. in 책 -> 복사된다 = deep copy / 복사가 되지 않는다 = 얕은 복사

실습1.

하나의 리스트 생성 후 array와 asarray를 이용해 새로운 배열을 생성- 깊은 복사 / 얕은 복사?

```
In [59]: l1
```

```
Out[59]: [1, 2, 3, 4]
```

```
In [60]: a1=np.array(l1)
```

```
In [61]: a2=np.asarray(l1)
```

```
-----  
In [62]: l1[0]=8  #수정
```

```
-----  
In [63]: l1
```

```
Out[63]: [8, 2, 3, 4]
```

```
In [64]: a1
```

```
Out[64]: array([1, 2, 3, 4]) # 깊은 복사
```

```
In [65]: a2
```

```
Out[65]: array([1, 2, 3, 4]) # 깊은 복사  
-----
```

-> 리스트를 인자로 사용시 둘 다 깊은복사(deep copy).

데이터 타입이 array로 바뀌기 때문!

실습2.

하나의 배열 생성 후 array와 asarray를 이용해 새로운 배열을 생성- 깊은 복사 / 얕은 복사?

```
In [69]: aa1=np.array([1,2,3,4])
```

```
In [70]: aa2=np.array(aa1)
```

```
In [71]: aa3=np.asarray(aa1)
```

```
-----  
In [72]: aa1[3]=10  # 수정
```

```
-----  
In [73]: aa1
```

```
Out[73]: array([ 1,  2,  3, 10])
```

```
In [74]: aa2
```

```
Out[74]: array([1, 2, 3, 4]) # 깊은 복사
```

```
In [75]: aa3
```

```
Out[75]: array([ 1,  2,  3, 10]) # 얕은 복사  
-----
```

->배열을 인자로 사용시 array는 깊은 복사, asarray는 얕은 복사.

0619

4.1.2 ndarray의 자료형

```
In [14]: arr1=np.array([1,2,3], dtype=float)
```

```
In [15]: arr2=np.array([1,2,3],dtype=int)
```

```
In [16]: arr1.dtype
```

```
Out[16]: dtype('float64')
```

```
In [17]: arr2.dtype
```

```
Out[17]: dtype('int32')
```

array 생성시 dtype 작성 가능.

NumPy 자료형

종류	Type Code	설명
int / uint	i / u	부호가 있는 정수 / 부호가 없는 정수
float	f	소수
complex	c	복소수
bool	?	True/False 불리언형
object	O	파이썬 객체형
str	S	고정 길이 문자열형(각 글자는 1바이트)
unicode_	U	고정 길이 유니코드형

astype 메서드 -- 데이터 타입 변경

```
In [54]: arr=np.array([3.7,-1.2,-2.6,-0.5,10.9])
```

```
In [55]: arr
```

```
Out[55]: array([ 3.7, -1.2, -2.6, -0.5, 10.9])
```

```
In [56]: arr.astype(int)
```

```
Out[56]: array([ 3, -1, -2,  0, 10])
```

+ 소수점 -> 정수형 변경시 소수점 아래자리는 버려진다. (not 반올림) -- 주의하기

astype 변경 - deecopy 발생, 메모리 낭비 발생 가능. 신중히 처리하거나 기존 메모리 지우기

4.1.3 배열과 스칼라 간의 연산

- 배열 : for 반복문을 작성하지 않고 데이터를 일괄처리 할 수 있음 = 벡터화

```
In [106]: arr=np.array([[1.,2.,3.],[4.,5.,6.]])
```

```
# list 형태 후 array! / R에서 c() -> matrix / data.frame처럼
```

```
In [107]: arr
```

```
Out[107]:
```

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

```
In [108]: arr*arr
```

```
Out[108]:
```

```
array([[ 1.,  4.,  9.],  
       [16., 25., 36.]])
```

```
In [109]: arr-arr
```

```
Out[109]:
```

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

```
In [110]: 1/arr
```

```
Out[110]:
```

```
array([[1.         , 0.5         , 0.33333333],  
       [0.25        , 0.2         , 0.16666667]])
```

```
In [111]: arr**0.5
```

```
Out[111]:
```

```
array([[1.         , 1.41421356, 1.73205081],  
       [2.         , 2.23606798, 2.44948974]])
```

```
# 스칼라 값에 대한 산술연산은 각 요소로 전달된다
```

4.1.4 색인과 슬라이싱 기초

리스트와 가장 큰 차이점은 배열 슬라이싱은 원본 배열의 뷰 개념 -- 수정시 원본도 수정됨

1차원 배열 - 표면적으로 파이썬의 리스트와 유사하게 동작함

```
In [112]: arr=np.arange(10)
```

```
In [113]: arr
```

```
Out[113]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [115]: arr[5:8]
```

```
Out[115]: array([5, 6, 7])
```

```
In [116]: arr[5:8]=12
```

```
In [117]: arr
```

```
Out[117]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

데이터는 복사되지 않고, 뷰에 대한 변경은 그대로 원본 배열에 반영됨 (얕은 복사)

.copy() 메서드 -- 깊은 복사

1.하나의 배열을 만든다

2.배열의 슬라이싱으로 새로운 배열을 만들고, copy 메서드를 사용하여 만든 후
각각 생성된 슬라이싱 조각이 변경될 때 원본의 데이터도 변경되는지 확인하자.

```
In [2]: arr10=np.array([1,2,3])
```

```
In [3]: arr11=arr10[:]
```

```
In [4]: arr12=arr10[:].copy()
```

```
In [5]: arr11[0]=8
```

```
In [6]: arr11
```

```
Out[6]: array([8, 2, 3])
```

```
In [7]: arr10
```

```
Out[7]: array([8, 2, 3])
```

```
In [8]: arr12
```

```
Out[8]: array([1, 2, 3])
```

```
In [10]: arr11[1]=88
```

```
In [11]: arr12[2]=888
```

```
In [14]: arr10
```

```
Out[14]: array([ 8, 88,  3])
```

```
In [13]: arr11
```

```
Out[13]: array([ 8, 88,  3])
```

```
In [12]: arr12
```

```
Out[12]: array([ 1,  2, 888])
```

다차원 배열

```
In [118]: arr2d=np.arange(1,10).reshape(3,3)
```

```
In [119]: arr2d
```

```
Out[119]:
```

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

```
In [120]: arr2d[2]    # 1차원 배열(차원축소)
```

```
Out[120]: array([7, 8, 9])
```

```
In [121]: arr2d[0][2] # 스칼라(차원축소)
```

```
Out[121]: 3
```

```
In [122]: arr2d[0,2]
```

```
Out[122]: 3
```

- 다차원 배열에서 정수 색인시 차원의 축소가 진행됨

```
In [123]: arr3d=np.arange(1,13).reshape(2,2,3)
```

```
In [124]: arr3d
```

```
Out[124]:
```

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

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

```
In [125]: arr3d[0]    #차원의 축소 발생 -> 스칼라 값과 배열 모두 대입 가능
```

```
Out[125]:
```

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

```
In [127]: old_values=arr3d[0].copy()
```

```
In [128]: arr3d[0]=42
```

```
In [129]: arr3d
```

```
Out[129]:
```

```
array([[[42, 42, 42],
        [42, 42, 42]],
```

```
       [[ 7,  8,  9],
        [10, 11, 12]])
```

```
In [130]: arr3d[0]=old_values
```

```
In [132]: arr3d
```

```
Out[132]:
```

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

슬라이스 색인

- 얇은 복사, 항상 같은 차원의 결과값 리턴
- 행렬끼리의 곱에서 차원 축소 발생하면 수행 X, 따라서 차원 중요
- 기본 : 행출력

1차원

```
n [139]: arr[1:6] # end범위만 조심!
```

```
ut[139]: array([ 1,  2,  3,  4, 12])
```

#다차원

```
In [140]: arr2d
```

```
Out[140]:
```

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

```
In [141]: arr2d[:2]
```

```
Out[141]:
```

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

arr2d[0:2,:] / arr2d[0:2] 와 같은 의미

```
In [143]: arr2d[:2, 1:]
```

```
Out[143]:
```

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

색인을 여러 개 넘겨서 다차원을 슬라이싱 하는 것도 가능하다.

cf. 정수색인과 슬라이스를 함께 사용하면 한 차원 낮은 배열을 반환한다.

```
In [144]: arr2d[1, :2]
```

```
Out[144]: array([4, 5])
```

```
In [145]: arr2d[2,:1]
```

```
Out[145]: array([7])
```

```
In [146]: arr2d[2,:1].ndim
```

```
Out[146]: 1
```

```
In [148]: arr2d.ndim
```

```
Out[148]: 2
```

3차원 배열도 살펴보기

첫 번째 층의 전체 행과 전체 컬럼

```
In [28]: arr3[0,:,:]
```

```
Out[28]:
```

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

각 층의 첫번째 행만? # 차원축소 발생

```
In [31]: arr3[:,0,:]
```

```
Out[31]:
```

```
array([[ 0,  1,  2,  3,  4],  
       [10, 11, 12, 13, 14],  
       [20, 21, 22, 23, 24]])
```

cf.

```
In [30]: arr3[:,0,:]
```

```
Out[30]:
```

```
array([[0, 1, 2, 3, 4],  
       [5, 6, 7, 8, 9]]) # 첫 번째 층이 반환됨 -- 원래 의도와 달라짐
```

+ 각 층의 처음행 원소만 가져오고 싶은데, arr3[:,0] 은 앞에 [:] 로 다 가져와서 [0]을 사용하면 첫 차원의 인덱스 설정됨

+ [][][] 형태에서 전체[:]를 표현하면 다른 결과값 나올 수도 있다.

4.1.5 불리언 색인

In [149]: **names**

Out[149]: array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')

In [150]: **data=np.random.randn(7,4)**

In [151]: **data**

Out[151]:

```
array([[ 0.07976583, -1.20992107,  1.13439923, -1.4485478 ],
       [-1.16569999, -1.10488425, -0.59356941,  0.79831317],
       [-1.17623513,  0.09100287, -0.51041449, -0.33809355],
       [-2.82863079, -2.00223493, -0.27617305, -1.06262861],
       [ 1.74349993, -0.52266151,  0.99256825,  0.31020221],
       [-1.33033988,  1.26749417,  0.29296222,  0.29330798],
       [-0.94207558,  1.94372162,  1.07690065, -0.6123475 ]])
```

각각의 이름이 data배열의 각 row에 대응한다고 가정.

In [152]: **names=='Bob'** **# 불리언 배열**

Out[152]: array([True, False, False, True, False, False, False])

In [153]: **data[names=='Bob']** **# 불리언 색인**

Out[153]:

```
array([[ 0.07976583, -1.20992107,  1.13439923, -1.4485478 ],
       [-2.82863079, -2.00223493, -0.27617305, -1.06262861]])
```

In [154]: **data[names=='Bob',2:]** **# 불리언 색인도 슬라이스, 정수 색인과 혼용 가능**

Out[154]:

```
array([[ 1.13439923, -1.4485478 ],
       [-0.27617305, -1.06262861]])
```

In [155]: **names!='Bob'** **# 'Bob'이 아닌 요소 선택**

Out[155]: array([False, True, True, False, True, True, True])

In [156]: **data[~(names=='Bob')]** **# ~를 사용해서 조건절 부정**

Out[156]:

```
array([[ -1.16569999, -1.10488425, -0.59356941,  0.79831317],
       [-1.17623513,  0.09100287, -0.51041449, -0.33809355],
       [ 1.74349993, -0.52266151,  0.99256825,  0.31020221],
       [-1.33033988,  1.26749417,  0.29296222,  0.29330798],
       [-0.94207558,  1.94372162,  1.07690065, -0.6123475 ]])
```



```
In [157]: mask = (names=='Bob') | (names=='Will')
# 두 가지 이상의 조건을 선택할 때 %(and)와 | (or) 같은 논리연산자 사용 가능
and, or은 불가능
```

```
In [158]: mask
Out[158]: array([ True, False,  True,  True,  True, False, False])
```

```
In [159]: data[mask]
Out[159]:
array([[ 0.07976583, -1.20992107,  1.13439923, -1.4485478 ],
       [-1.17623513,  0.09100287, -0.51041449, -0.33809355],
       [-2.82863079, -2.00223493, -0.27617305, -1.06262861],
       [ 1.74349993, -0.52266151,  0.99256825,  0.31020221]])
```

항상 데이터 복사가 이루어짐! 얇은 복사

```
In [160]: data[data<0] = 0 # 모든 음수를 0으로 만들기
```

```
In [161]: data
Out[161]:
array([[0.07976583, 0.          , 1.13439923, 0.          ],
       [0.          , 0.          , 0.          , 0.79831317],
       [0.          , 0.09100287, 0.          , 0.          ],
       [0.          , 0.          , 0.          , 0.          ],
       [1.74349993, 0.          , 0.99256825, 0.31020221],
       [0.          , 1.26749417, 0.29296222, 0.29330798],
       [0.          , 1.94372162, 1.07690065, 0.          ]])
```

```
In [162]: data[names!='Joe'] = 7
```

1차원 불리언 배열을 선택해서 전체 로우나 칼럼 쉽게 선택 가능

```
In [163]: data
Out[163]:
array([[7.          , 7.          , 7.          , 7.          ],
       [0.          , 0.          , 0.          , 0.79831317],
       [7.          , 7.          , 7.          , 7.          ],
       [7.          , 7.          , 7.          , 7.          ],
       [7.          , 7.          , 7.          , 7.          ],
       [0.          , 1.26749417, 0.29296222, 0.29330798],
       [0.          , 1.94372162, 1.07690065, 0.          ]])
```

[문제]

1부터 증가하는 3X5X6의 임시 배열을 만든 후

```
a1=np.arange(1,91).reshape(3,5,6)
```

1) 각 층의 첫 번째 행의 모든 데이터를 10으로 변경

```
a1[:,0:1]=10 #차원 유지
```

```
arr[:,0]=10 #차원축소 발생 (수정시엔 문제 없음)
```

cf. `arr[:,0]`

전체를 앞에 쓰면서 분리하면 앞에 전체범위를 무시.

왜냐면 `arr[:,0]` 먼저 수행--> 다시 `arr[:,0]`은 `arr[0]`과 같은 의미. 즉, `arr` 결과는 첫 번째 층만 출력. 분리할 때엔 모두라는 선택범위가 먼저 있으면 안 됨.

2) 각 층의 첫 번째 열의 모든 데이터를 10으로 변경

```
a1[:,0:1]=10 # 차원, 모양 유지
```

```
a1[:,0]=10 # 차원축소 발생, col모양이 아닌 row모양으로 나타남
```

3) 첫 번째 층의 4번째 행의 3,4,5번째 열의 데이터를 5로 변경

```
a1[:,1,3:4,2:5]=5 #차원 유지
```

```
a1[0,3,2:5]=5 # 차원축소 발생
```

4) 모든 층과 행의 6번째 열의 데이터를 20으로 변경

(불리언 색인 활용)

```
b1=np.array(['a','b','c','d','e','f'])
```

리스트는 벡터연산 X, 불리언 색인은 배열 형태로 만들어야 각 값에 대한 불리언 값 반환

```
a1[:,b1=="f"]=20 # 불리언 색인은 차원축소 X
```

강사님은 [0,0,0,0,0,1]

5)[[5, 5],

[27,28]] 의 값을 100으로 변경

```
a1[0,3:,2:4]=100
```

0620

4.1.6 팬시 색인

`array[[행의 정수]][[열의 정수]]`

여러 개의 행 범위 - 열 범위인 경우 각각 순서쌍 위치의 원소가 반환됨.

위와 다르게 나열된 행과 열이 크로스 된 형태의 데이터를 제공하려면

`array[[행의 정수]][:, [열의 정수]]`

다음의 방법과 같은 의미

`array[np.ix_([행 범위], [열 범위])]`

`np.ix_` : under bar 까지 함께 있는 것 기억하기! cf. pandas에서 .ix메서드

- 정수 배열을 사용한 색인
- 색인기호[]가 두 번 중복된 형태로, 색인은 정수만 표현 가능(슬라이스 표현 불가)
- 항상 깊은 복사가 이루어 짐
- 슬라이스 색인과는 다르게 차원의 축소가 가능함--항상 차원의 축소는 아님
/특정 컬럼, 행 선택시 차원의 축소가 대체적으로 발생함.

특정 순서로 로우를 선택하고 싶으면 원하는 순서가 명시된 정수가 담긴 ndarray나 리스트 명시
like in R 벡터색인 ~ `a1[c(1,3),c(6,4)]` c -> list로 바꾼 것 뿐

기억하기

[리스트형태로 입력 -- 대괄호가 두 개 !]-원래 색인 []

reverse 색인도 음의 정수로 표현 가능 ex. -1 : 뒤에서 첫 번째, -4 : 뒤에서 네 번째
vs R (- : 제외)

In [79]: `arr=np.arange(32).reshape((8,4))`

In [80]: `arr[[-3,-5,-7]]`

Out[80]:

```
array([[20, 21, 22, 23],
       [12, 13, 14, 15],
       [ 4,  5,  6,  7]])
```

2차원 배열에서의 팬시색인 연습

In [12]: **a1**

Out[12]:

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

In [13]: **a1[[0,1]]** -- **결의 [] : 색인 / 안의 [,] : 행 선택 리스트**

Out[13]:

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

In [14]: **a1[[0,1],[0,1]]** # **[[1,2] , [6,7]] 을 하고 싶은데 안 됨. R과의 차이점!**
 # **순서쌍 -- [0,0] + [1,1]을 출력하도록 되어있음**

Out[14]: array([1, 7])

In [16]: **a1[[0,1]][[0,1]]**

분리한 색인 형태도 안 됨!

a1[[0,1]](행 팬시색인)의 [[0,1]](행 팬시색인)이므로 행만 색인해서 출력

Out[16]:

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

In [18]: **a1[[0,1]][:,][0,1]** **### 팬시색인의 핵심 ###**

분리해서 사용함. 뒤의 색인은 앞에 있는 것 받고, 그 중 열 부분을 선택한 것 !
뒤의 색인에서 행의 범위에 : 쓰는 것 주의!

Out[18]:

```
array([[1, 2],
       [6, 7]])
```

In [29]: **a1[np.ix_([0,1],[0,1])]** # **같은 의미의 함수! 언더바까지 기억 잘 하기**

Out[29]:

```
array([[1, 2],
       [6, 7]])
```

[[8,9], [18,19]] 만 출력하려면?

In [21]: **a1[[1,3]][2,3]**

Out[21]:

```
array([[ 8,  9],
       [18, 19]])
```

띄워져 있는 건 팬시색인을 통해 해야 함 ! R이랑 다른 점 유의하기

네 번째 열만 출력

In [25]: **a1[:,3] # 차원축소 발생**

Out[25]: array([4, 9, 14, 19])

In [26]: **a1[:,:[3]] # 차원의 축소가 발생하지 않음 -- [] 중복 팬시 색인 형태**

Out[26]:

```
array([[ 4],
       [ 9],
       [14],
       [19]])
```

cf.

In [28]: **a1[:,3]**

Out[28]: array([16, 17, 18, 19, 20])

-- 네 번째 행 (not 네 번째 열) 출력.

(a1[:,3]의 [3]은 a1[3]과 같다.)

과정 살펴보기

ex. a1[:,3]

a1[:,3]인 경우 모든 행 -->차원 유지, 모든 범위

a1[:,3]은 a1[:,3]의 [3], 즉 a1[3]으로 a1의 네 번째 행.

--> 따라서 뒤에 색인에 행 부분은 앞 색인의 결과 차원을 생각하고 쓰기

차원 축소 관련 정리

In [33]: **a1[:,0:1]** #슬라이스 색인 -> 차원 유지

Out[33]:

```
array([[ 1],
       [ 6],
       [11],
       [16]])
```

In [34]: **a1[:,:[0]]** # 0을 리스트로 묶으면 차원 유지됨

Out[34]:

```
array([[ 1],
       [ 6],
       [11],
       [16]])
```

In [35]: **a1[:,:[0]]** # 차원축소 발생 (정수색인)

Out[35]: array([1, 6, 11, 16])

3차원에서 팬시 함수 쓰기 연습

In [37]: `a2=np.arange(1,41).reshape(2,4,5)`

In [38]: `a2`

Out[38]:

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

       [[21, 22, 23, 24, 25],
        [26, 27, 28, 29, 30],
        [31, 32, 33, 34, 35],
        [36, 37, 38, 39, 40]]])
```

7,12만 선택하기

In [40]: `a2[0,[1,2],[1]]` #차원축소

Out[40]: `array([7, 12])`

cf. 분리할 때

In [42]: `a2[0][[1,2]][1]`

Out[42]: `array([11, 12, 13, 14, 15])` # 원하는 값과 다른 값

cf의 명령어는 처음 []을 정수 색인 -> 차원축소 일어남!

`a2[0]` : `a2`의 첫 층

`a2[0][[1,2]]` : `a2`의 첫 층의 2nd, 3rd 행

`a2[0][[1,2]][1]` : `a2`의 첫 층의 2nd, 3rd 행의 2nd 행

-----> 해결

색인을 분리해서 쓸 때에는 차원의 유지 조심해야 함.

따라서 앞쪽에서의 분리된 색인이 두 개 이상이거나, 전체일 때 생각한 값이 안 나올 수 있음.

분리해서 쓸 때 꼭 슬라이스 표현 꼭 써주기!

1. 층 차원축소 X 표현

In [82]: `a2[[0]][:,[1,2]][:,:1]`

결과값에 대해 차원축소를 발생시키지 않으려면, 분리된 마지막 색인에 `[:,:1]`로 써주기

Out[82]: `array([[7, 12]])`

In [56]: `a2[0:1][:,[1,2]][:,:1]`

Out[82]: `array([[7, 12]])`

2. 층 차원축소 O 표현

In [83]: **a2[0][[1,2]][:,1]**

Out[83]: array([7, 12])

-----일반적인 삼차원 팬시 색인 표현!-----

In [51]: **a2[:, :, [0]]**

Out[51]:

```
array([[[ 1],
         [ 6],
         [11],
         [16]],

       [[21],
         [26],
         [31],
         [36]])
```

4. 1. 7. 배열 전치와 축 바꾸기

T

arr.T

- 배열의 행과 열의 구조를 바꿈
- 색인데이터를 복사(deep copy)하지 않고, 데이터 모양이 바뀐 뷰를 반환하는 특별한 기능
- T메서드를 사용한 간단한 배열 전치

cf.

행렬의 inner product는 **np.dot 함수를** 이용해 간단히 계산 가능

```
arr=np.random.randn(6,3)
```

```
np.dot(arr.T,arr)
```

transpose 메서드

arr.transpose((i,j,k,...))

- 얇은 복사(뷰)
- 다차원일 때 사용하는 함수, 여러 개의 층을 동시에 변경할 수 있음
- transpose((i,j,k)) -- 튜플형태로 전달 (튜플형태가 아니어도 작동)

swapaxes 메서드 -- 두 개의 축 번호를 받아서 배열을 뒤바꿈

arr.swapaxes(j,k)

- 데이터를 복사하지 않고 원래 데이터에 대한 뷰를 반환함
- 바꿀 축만 명시하면 되므로 함수 인수 순서는 상관 없음

축번호

2차원 $m \times n$

행의 축번호 - 0 / 열의 축번호 - 1

3차원 $m \times n \times l$

층의 축번호 - 0 / 행의 축번호 - 1 / 열의 축번호 - 2

arr.transpose((1,0,2)) 층 자리에 행을 넣겠다 -> 행 자리에 층을 -> 열은 그대로

arr.transpose((0,1,2)) -- 원래대로 출력

arr.transpose((0,2,1)) -- 행-열을 바꿔 출력

In [61]: **a2**

Out[61]:

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

       [[21, 22, 23, 24, 25],
        [26, 27, 28, 29, 30],
        [31, 32, 33, 34, 35],
        [36, 37, 38, 39, 40]]])
```

In [62]: **a2.transpose((0,2,1))**

Out[62]:

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

       [[21, 26, 31, 36],
        [22, 27, 32, 37],
        [23, 28, 33, 38],
        [24, 29, 34, 39],
        [25, 30, 35, 40]]])
```

연습문제

1. 5*4 형태의 임시 배열을 생성한 후 p(1,0), p(3,1) 의 값을 출력하자

```
q1=np.arange(1,21).reshape((5,4))
```

```
q1[[1,3], [0,1]]
```

2. 위의 배열에서 arr[1:3,2:4]의 형태와 동일하게 팬시색인을 통해 출력해보자

```
q1[[1,2]][:,[2,3]]
```

3. 3*5*4 형태의 임시 배열을 생성한 후 각각 행과 열의 값을 바꿔서 출력

```
q3=np.arange(1,61).reshape(3,5,4)
```

1)transpose

```
q3.transpose((0,2,1)) # 웬만하면 문법상 튜플 형태 사용하기!
```

2)swapaxes

```
q3.swapaxes(1,2) 혹은 q3.swapaxes(2,1)
```

4. 3*5*4 형태의 임시 배열을 생성한 후 각각 층(0)과 열(2)의 값을 서로 바꾸는 전치치환 출력

1) transpose

```
q3.transpose((2,1,0))
```

2) swapaxes

```
q3.swapaxes(0,2) 혹은 q3.swapaxes(2,0)
```

5. 0~59까지의 값을 갖는 3*5*4의 배열에서 다음의 값만 return 하도록 팬시색인을 통해 출력해 보자

```
array([[[ 9, 10],
        [13, 14]],

       [[29, 30],
        [33, 34]],

       [[49, 50],
        [53, 54]]])
```

```
q5=np.arange(60).reshape(3,5,4)
```

```
q5[:,:[2,3]][:,:,[1,2]]
```

4.2 유니버설 함수(범용 함수)

unfc

- ndarray 안에 있는 데이터 원소별로 연산을 수행하는 함수
- 하나 이상의 스칼라 값을 받아서 하나 이상의 스칼라 결과 값을 반환하는 간단한 함수를 고속으로 수행할 수 있는, 벡터화된 래퍼 함수

단항 유니버설 함수

함수	설명
abs, fabs	각 원소의 절대값을 구한다. 복소수가 아닌 경우 빠른 연산을 위해 fabs를 사용한다.
sqrt	각 원소의 제곱근을 계산한다. (= arr**0.5)
square	각 원소의 제곱을 계산한다. (= arr**2)
exp	각 원소에서 지수 를 계산한다.
log, log10, log2, log1p	각각 자연로그, 로그10, 로그2, 로그(1+x)
sign	각 원소의 부호를 계산한다. 1(양수) 0(영) -1(음수)
ceil	각 원소의 값보다 같거나 큰 정수 중 가장 작은 정수를 반환한다.
floor	각 원소의 값보다 작거나 같은 정수 중 가장 큰 정수를 반환한다.
rint	각 원소의 소수자리를 반올림한다. dtype은 유지된다.
modf	각 원소의 몫과 나머지를 각각의 배열로 반환한다.
isnan	각각의 원소가 숫자인지 아닌지를(NaN, Not a Number) 나타내는 불리언 배열을 반환한다.
isfinite, isinf	배열의 각 원소가 유한한지 무한한지 나타내는 불리언 배열을 반환한다.
cos, cosh, sin, sinh, tan, tanh	일반 삼각 함수와 쌍곡 삼각 함수
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	역삼각함수
logical_not	각 원소의 논리 부정(not) 값을 계산한다. ~arr과 동일하다. (Numpy에서 -arr => ~arr로 바뀜. pandas는 동일)

ex.

```
In [101]: arr=np.arange(10)
```

```
In [102]: np.sqrt(arr)
```

```
Out[102]:
```

```
array([0.          , 1.          , 1.41421356, 1.73205081, 2.          ,
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.          ])
```

이항 유니버설 함수

함수	설명
add	두 배열에서 같은 위치의 원소끼리 더한다
subtract	첫 번째 배열에서 두 번째 배열의 원소를 뺀다.
multiply	배열의 원소끼리 곱한다.
divide, floor_divide	첫 번째 배열에서 두 번째 배열의 원소를 나눈다. floor_divide는 몫만 취한다.
power	첫 번째 배열의 원소에 두 번째 배열의 원소만큼 제공한다.
maximum, fmax	각 배열의 두 원소 중 큰 값을 반환한다. fmax는 NaN을 무시한다.
minimum, fmin	각 배열의 두 원소 중 작은 값을 반환한다. fmin는 NaN을 무시한다.
mod	첫 번째 배열의 원소에 두 번째 배열의 원소를 나눈 나머지를 구한다.
copysign	첫 번째 배열 원소의 기호를 두 번째 배열 원소의 기호로 바꾼다.
greater, greater_equal, less, less_equal, equal, not_equal	각각 두 원소 간 $>$, $>=$, $<$, $<=$, $=$, $!=$ 의 비교연산 결과를 불리언 배열로 반환한다.
logical_and, logical_or, logical_xor	각각 두 원소 간의 논리연산 $&$, $ $, $^$ 결과를 반환한다.

ex,

```
In [104]: x=np.random.randn(8)
```

```
In [105]: y=np.random.randn(8)
```

```
In [106]: x
```

```
Out[106]:
```

```
array([ 0.08800789, -1.37506061,  0.62205514, -0.08306478, -1.10902029,
        1.31719463, -0.12558175,  1.2525783 ])
```

```
In [107]: y
```

```
Out[107]:
```

```
array([ 0.95422533, -1.03572446, -0.35276758,  0.60461614,
        0.96840077,
        1.64900188, -0.58849783,  0.54167185])
```

```
In [108]: np.maximum(x,y)
```

```
Out[108]:
```

```
array([ 0.95422533, -1.03572446,  0.62205514,  0.60461614,  0.96840077,
        1.64900188, -0.12558175,  1.2525783 ])
```

4.3 배열을 사용한 데이터 처리

4.3.1 배열연산으로 조건절 표현하기

참고 -- 리스트 내포 표현식

cf. 삼항표현식 `res = '합격' if jumsu >= 60 else '불합격'`

컴프리헨션 리스트 = `[수식 / for 항목 in range() / if 조건식]`

- 큰 배열을 처리할 경우 느린 속도 제공
- 다차원 배열은 사용 불가
- 위의 단점을 where 함수를 이용해 보완

```
In [87]: xarr=np.array([1.1,1.2,1.3,1.4,1.5])
```

```
In [88]: yarr=np.array([2.1,2.2,2.3,2.4,2.5])
```

```
In [89]: cond=np.array([True,False,True,True,False])
```

```
In [90]: result = [(x if c else y)
```

```
...:               for x, y, c in zip(xarr, yarr, cond)]
```

```
In [91]: result
```

```
Out[91]: [1.1, 2.2, 1.3, 1.4, 2.5]
```

-----cond가 참일 때 xarr, 거짓일 때 yarr을 가져오기

where 함수

np.where(조건 혹은 불리언 배열, 참일 때 return 값, 거짓일 때 return 값)

- x if 조건 else y 같은 삼항식의 벡터화된 버전 ~ 배열로 반환
- 다차원도 가능
- decode(오라클), ifelse()(R) 함수와 비슷

ex.

```
In [116]: result=np.where(cond,xarr,yarr)
```

```
In [117]: result
```

```
Out[117]: array([1.1, 2.2, 1.3, 1.4, 2.5])
```

np.where의 두, 세 번째 인자는 배열이 아니라도 괜찮다.

```
In [124]: arr=np.random.randn(4,4)
```

```
In [125]: arr
```

```
Out[125]:
```

```
array([[ 0.74341159, -0.90940428, -0.88456582,  0.12589991],
       [ 1.54192811,  0.88906359,  0.62644188,  0.84966901],
       [ 0.47172494,  0.292392   ,  1.05307354, -0.43770275],
       [ 1.14600904,  0.11722636, -0.50288879, -0.81580458]])
```

```
In [126]: np.where(arr>0,2,-2)
```

```
Out[126]:
```

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

```
In [127]: np.where(arr>0,2,arr)    # 본인 출력도 가능
```

```
Out[127]:
```

```
array([[ 2.         , -0.90940428, -0.88456582,  2.         ],
       [ 2.         ,  2.         ,  2.         ,  2.         ],
       [ 2.         ,  2.         ,  2.         , -0.43770275],
       [ 2.         ,  2.         , -0.50288879, -0.81580458]])
```

중첩된 where 사용 가능

cond1, cond2는 불리언 배열

```
np.where(cond1 & cond2, 0,
        np.where(cond1, 1,
                  np.where(cond2, 2, 3)))
```

아래와 같은 뜻

```
result = []
```

```
for i in range(n):
```

```
    if cond1[i] and cond2[i]:
```

```
        result.append(0)
```

```
    elif cond1[i]:
```

```
        result.append(1)
```

```
    elif cond2[i]:
```

```
        result.append(2)
```

```
    else:
```

```
        result.append(3)
```

4.3.2 수학 메서드와 통계 메서드

-- R과 축 부분에서 헛갈리기 쉬운 부분 ! 유의하기

axis 인자

- mean이나 sum 같은 함수는 선택적으로 axis인자를 받는다.

- **axis=0** **0:행** -- 서로 다른 **행들의 계산** -- **열끼리 계산** (in R : 열)

axis=1 **1:열** -- 서로 다른 **열들의 계산** -- **행끼리 계산** (in R : 행)

In [132]: **ar1=np.arange(1,21).reshape(5,4)**

In [133]: **ar1**

Out[133]:

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

In [134]: **ar1.mean()** **#메서드! 출력 위해서 () 같이 쓰기**

Out[134]: 10.5

In [135]: **np.mean(ar1)** **# 함수**

Out[135]: 10.5

In [136]: **ar1.sum()**

Out[136]: 210

In [139]: **ar1.sum(0)**

Out[139]: array([45, 50, 55, 60])

In [140]: **ar1.sum(1)**

Out[140]: array([10, 26, 42, 58, 74])

In [5]: **a2=np.arange(9).reshape(3,3)**

In [6]: **a2.cumsum(axis=0)**

Out[6]:

```
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]], dtype=int32)
```

In [7]: **a2.cumprod(axis=1)**

Out[7]:

```
array([[ 0,  0,  0],
       [ 3, 12, 60],
       [ 6, 42, 336]], dtype=int32)
```


3차원에서의 축 적용

```
In [8]: arr=np.arange(18).reshape(2,3,3)
```

```
In [9]: arr
```

```
Out[9]:
```

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

```
In [10]: arr.sum(0)    # 3차원에서 0 : 층 -> 층별 : 서로 다른 층끼리의 합 (행 / 열 그대로)
```

```
Out[10]:
```

```
array([[ 9, 11, 13],
       [15, 17, 19],
       [21, 23, 25]])
```

```
In [11]: arr.sum(1)    # 3차원에서 1 : 행 -> 행별 : 서로 다른 행끼리의 합 (층 / 열 그대로)
```

```
Out[11]:
```

```
array([[ 9, 12, 15],
       [36, 39, 42]])
```

```
In [12]: arr.sum(2)    # 3차원에서 2 : 열 -> 열별 : 서로 다른 열끼리의 합 (층 / 행 그대로)
```

```
Out[12]:
```

```
array([[ 3, 12, 21],
       [30, 39, 48]])
```

Numpy에서 var, sd 함수

```
In [15]: arr=np.arange(9).reshape(3,3)
```

```
In [16]: arr.var(0)
```

```
Out[16]: array([6., 6., 6.])
```

```
In [22]: np.square(arr-arr.mean(0)).sum(0)/2    # 값이 다름!
```

```
Out[22]: array([9., 9., 9.])
```

in Numpy : np.var, np.sd에서 자유도 항상 신경써서 작성하기!

--> **ddof 옵션** 사용하기

자유도가 n-1 일 때, **ddof : 1(빼려는 값)**을 써야 함 !! 기본 값이 0임

(cf. pandas-> 기본값 ddof=1)

```
In [33]: arr.var(0,ddof=1)
```

```
Out[33]: array([9., 9., 9.])
```

```
In [34]: arr.var(0,ddof=1, keepdims=True)    #keepdims : 차원방지 옵션
```

```
Out[34]: array([[9., 9., 9.]])
```

기본 배열 통계 메서드

메서드	설명
sum	배열 전체 혹은 특정 축에 대한 모든 원소의 합을 계산한다. 크기가 0인 배열의 sum은 0
mean	산술평균을 구한다. 크기가 0인 배열의 mean은 NaN
std, var	각각 표준편차와 분산을 구한다. 선택적으로 자유도를 줄 수 있음 Numpy에서 제공하는 기본 값은 n으로 나누는 것이다. 자유도 - ddof option 사용
min, max	최소 값 / 최대 값
argmin, argmax	최소 원소의 인덱스 값, 최대 원소의 인덱스 값
cumsum	각 원소의 누적 합
cumprod	각 원소의 누적 곱

★★★★★4.3.3 불리언 배열을 위한 메서드★★★★★

- 불리언 배열에 대한 sum메서드를 실행하면 True인 원소의 개수를 반환한다.

```
In [19]: arr=np.random.randn(100)
```

```
In [20]: (arr > 0)
```

```
Out[20]:
```

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

```
In [22]: (arr>0).sum() # group by 등보다 간편, 잘 활용됨! / True 개수 세기
```

```
Out[22]: 53
```

```
In [23]: (arr>0).mean(0) # True의 비율
```

```
Out[23]: 0.53
```

```
In [23]: bools=np.array([False,False,True,False])
```

```
In [24]: bools.any() # 하나 이상의 True값이 있는지 검사
```

```
Out[24]: True
```

```
In [25]: bools.all() #모든 원소가 True인지 검사
```

```
Out[25]: False
```

4.3.4정렬

- 리스트형처럼 sort메서드를 이용해 정렬 가능
- 출력 안 하고 원본을 바로 바꿈
- 출력하려면 다시 배열 이름 작성해야 함

#1차원 배열

```
In [26]: arr=np.random.randn(8)
```

```
In [27]: arr
```

```
Out[27]:
```

```
array([ 1.04791214,  1.01833341,  1.63544794, -1.28657428, -0.81290615,
        -0.60883545, -0.61964091, -0.99952912])
```

```
In [28]: arr.sort()
```

```
In [29]: arr
```

```
Out[29]:
```

```
array([-1.28657428, -0.99952912, -0.81290615, -0.61964091, -0.60883545,
        1.01833341,  1.04791214,  1.63544794])
```

#다차원 배열

- sort 메서드에 넘긴 축의 값에 따라 1차원 부분을 정렬

```
In [36]: arr=np.random.randn(3,5)
```

```
In [37]: arr
```

```
Out[37]:
```

```
array([[ -0.75868058, -0.38795527, -0.58306152,  0.38321699, -0.1590607 ],
       [-1.19226726,  0.20051711, -1.1316573 , -1.36858687,  1.97588999],
       [ 0.70407251, -0.73380732,  1.65310992,  0.5401295 ,  0.47129792]])
```

```
In [38]: arr.sort(0)
```

#3*5 형태 ~ 0 : 행 -- 서로 다른 행끼리 계산 -- 열기준 정렬

```
In [39]: arr
```

```
Out[39]:
```

```
array([[ -1.19226726, -0.73380732, -1.1316573 , -1.36858687, -0.1590607 ],
       [-0.75868058, -0.38795527, -0.58306152,  0.38321699,  0.47129792],
       [ 0.70407251,  0.20051711,  1.65310992,  0.5401295 ,  1.97588999]])
```

배열의 분위수를 구하는 쉽고 빠른 방법-- 우선 배열을 정렬한 후 특정 분위의 값을 선택하는 것

```
In [40]: large_arr=np.random.randn(1000)
```

```
In [41]: large_arr.sort()
```

```
In [42]: large_arr[int(0.05*len(large_arr))] # 5%quantile
```

```
Out[42]: -1.7535366871835618
```

```
In [43]: int(0.05*len(large_arr))
```

```
Out[43]: 50
```

4.3.5 집합 함수

- 1차원 ndarray를 위한 몇 가지 기본 집합연산

```
In [46]: names=np.array(['Bob','Joe','Will','Bob','Will','Joe','Joe'])
```

```
In [47]: np.unique(names)           #np.unique() 함수 -- set + 정렬 역할
```

```
Out[47]: array(['Bob', 'Joe', 'Will'], dtype='<U4')
```

```
In [48]: ints=np.array([3,3,3,2,2,1,1,4,4])
```

```
In [50]: np.unique(ints)
```

```
Out[50]: array([1, 2, 3, 4])
```

cf. np.unique를 순수 파이썬으로 구현하면 정렬 + set() 함수

```
In [3]: sorted(set(names))
```

```
Out[3]: ['Bob', 'Joe', 'Will']
```

```
In [6]: np.in1d(names,['Bob','Joe'])
```

np.in1d() 함수

-- 첫 번째 배열의 각 원소가 두 번째 배열의 원소를 포함하는지 나타내는 불리언 배열 반환

```
Out[6]: array([ True,  True, False,  True, False,  True,  True])
```

cf. np.where(names=='Bob',True, np.where(names=='Joe', True, False)) 이거랑 같다

배열 집합연산

함수	설명
unique(x)	배열 x에서 중복된 원소를 제거한 후 정렬하여 반환한다.
intersect1d(x,y)	배열 x와 y에 공통적으로 존재하는 원소를 정렬하여 반환한다.
union1d(x,y)	두 배열의 합집합을 반환한다.
in1d(x,y)	x의 원소 중 y의 원소를 포함하는지를 나타내는 불리언 배열을 반환한다.
setdiff1d(x,y)	x와 y의 차집합을 반환한다.
setxor1d(x,y)	한 배열에는 포함되지만 두 배열 모두에는 포함되지 않는 원소들의 집합인 대칭차집합을 반환한다.

```
In [779]: x=np.array([1,2,5,6,2,7,4])
          y=np.array([2,8,6,7])
          np.all(np.in1d(y,x))  # y가 x의 부분집합인지 확인
```

```
Out [779]: False
```

4.4 배열의 파일 입.출력

4.4.1 배열을 바이너리 형식으로 디스크에 저장하기

np.save('파일명', 배열이름)

np.load('파일명')

- 파일확장자 지정 가능 - raw 바이너리 형식의 .npy 파일이 기본값
- > .txt로 저장해야 파이썬이 아닌 곳에서 안 깨지게 볼 수 있다! **np.savetxt(" .txt")**
- load()시 m*n형태가 아닌 불규칙한 자료는 list형태로 불러내기
- 저장된 위치는 디폴트 위치

```
In [8]: arr=np.arange(10)
```

```
In [9]: np.save('some_array', arr)
```

```
In [10]: np.load('some_array.npy')
```

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

np.savez('파일명', x1=배열명1, x2=배열명2, ...)

- 여러 개의 배열을 압축된 형식으로 저장할 수 있음
- 저장하려는 배열은 키워드 인자(key) 형태로 전달됨
- npz형태로 저장 -> npz 파일은 저장된 키워드 형태로 불러올 수 있다. (딕셔너리 형태)

```
In [11]: np.savez('array_archive.npz', a=arr, b=arr)
```

```
In [12]: arch=np.load('array_archive.npz')    # 압축된 상태, 바로 출력 X
```

```
In [13]: arch['b']    # 키와 함께 출력 가능 ~ 딕셔너리 형태
```

```
Out[13]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

4.4.2 텍스트 파일 불러오기와 저장하기

// pandas에서 read_csv, read_table 함수 위주로 쓰긴 할 것//

np.loadtxt()

- 구분자를 지정하거나, 특정 칼럼에 대한 변환 함수를 지정하거나 로우를 건너뛰는 등의 다양한 기능을 제공함.

- 불규칙한 자료 로딩 불가.

option

delimiter : 구분자(sep) 옵션

skiprows : 몇 줄 skip 할 건지 옵션 -- 몇 개 (부분 skip 불가)

usecols : 어떤 col 선택할 건지 옵션, 리스트[] 형태로 씀 -- 몇 번째(0=1st, 1=2nd, ...)

In [25]: **!type arr_test.txt** **#for 결과값과 비교, 원본 출력**

0.5, 0.2, 0.1, 0.5

0.1, 0.5, 0.3, 0.4

0.5, 0.4, 0.2, 0.8

! - 밖에 있는 프롬프트(cmd)에 전달할 때 쓰는 용어. not in 아나콘다, but in cmd.

ex. !ipconfig

type - 출력 명령어 in 윈도우 (리눅스 - cat)

In [26]: **arr=np.loadtxt('arr_test.txt', delimiter=',')**

In [27]: **arr**

Out[27]:

```
array([[0.5, 0.2, 0.1, 0.5],
       [0.1, 0.5, 0.3, 0.4],
       [0.5, 0.4, 0.2, 0.8]])
```

In [30]: **arr=np.loadtxt('arr_test.txt',delimiter=',',skiprows=2)**

In [31]: **arr**

Out[31]: array([0.5, 0.4, 0.2, 0.8])

In [32]: **arr=np.loadtxt('arr_test.txt',delimiter=',',usecols=[0,3,1])**

In [33]: **arr**

Out[33]:

```
array([[0.5, 0.5, 0.2],
       [0.1, 0.4, 0.5],
       [0.5, 0.8, 0.4]])
```


+ 추가(2019) 선형대수학 - 행렬의 연산

-일반연산자 * 를 사용해서는, 배열의 같은 위치의 원소 곱셈을 연산

함수	설명
diag()	정방행렬의 대각/비대각 원소를 반환 1차원 배열을 대각원소로 하는 n*n대각행렬 반환
dot()	행렬의 곱
trace()	대각원소의 합
linarg.det()	행렬식
linarg.eig()	정방행렬의 고유값과 고유벡터
linarg.inv()	정방행렬의 역행렬
linarg.pinv()	무어-펜로즈 유사역원 역행렬
linarg.qr()	QR분해
linarg.svd()	특이값 분해(SVD)
linarg.solve()	선형연립방정식 $Ax=b$ 의 해 (A는 정방행렬이어야)
linarg.lstsq()	최소제곱해(회귀분석 등)

```
In [791]: A=np.array([[1,3,5,7],[5,3,4,7]])
          B=np.array([[1,3],[5,7],[5,3],[4,7]])
          C=np.array([[4,9,1,3],[7,5,2,7],[5,3,4,3],[1,4,7,8]])
          print("{0}\n\n{1}\n\n{2}".format(A,B,C))
```

```
[[1 3 5 7]
 [5 3 4 7]]
```

```
[[1 3]
 [5 7]
 [5 3]
 [4 7]]
```

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

```
In [792]: np.dot(A,B)
```

```
Out [792]: array([[69, 88],
 [68, 97]])
```

```
In [793]: np.diag(C)      # 행렬이 정방행렬이면 대각행렬 반환
```

```
Out [793]: array([4, 5, 4, 8])
```

```
In [794]: np.diag(A[1])   # 행렬이 정방행렬이 아닐 때, 대각행렬로 하고 나머지가 0인 행렬 반환
```

```
Out [794]: array([[5, 0, 0, 0],
 [0, 3, 0, 0],
 [0, 0, 4, 0],
 [0, 0, 0, 7]])
```

```
In [795]: np.trace(A)
```

```
Out [795]: 4
```

```
In [796]: np.linalg.det(C) #행렬식
```

```
Out [796]: -1440.0000000000005
```

```
In [800]: np.linalg.inv(C) # 역행렬
```

```
Out [800]: array([[ -0.04583333,  0.07083333,  0.15833333, -0.10416667],
                  [ 0.15277778, -0.06944444, -0.02777778,  0.01388889],
                  [-0.0125    , -0.1625    ,  0.225     ,  0.0625    ],
                  [-0.05972222,  0.16805556, -0.20277778,  0.07638889]])
```

```
In [801]: np.linalg.eig(C) # 고유값, 고유벡터
```

```
Out [801]: (array([18.62476623+0.j          , -4.26801477+0.j          ,
                  3.32162427+2.66122552j,  3.32162427-2.66122552j])),
          array([[ 0.48933944+0.j          ,  0.63546053+0.j          ,
                  0.49149024+0.11067431j,  0.49149024-0.11067431j],
                  [ 0.57741493+0.j          , -0.65846911+0.j          ,
                  0.12143635+0.17873993j,  0.12143635-0.17873993j],
                  [ 0.39288156+0.j          , -0.25785862+0.j          ,
                  0.31364382-0.37577172j,  0.31364382+0.37577172j],
                  [ 0.52228632+0.j          ,  0.31002785+0.j          ,
                  -0.67817177+0.j          , -0.67817177-0.j          ]]))
```

```
In [805]: y=np.array([2,4,6,4,7,1])
          x=np.array([[1,2,3,5],[1,4,5,2],[1,2,3,4],[1,2,3,4],[1,3,5,6],[1,3,2,5]])
          np.linalg.lstsq(x,y) # 최소제곱추정량
```

C:\ProgramData\Anaconda3\envs\insun\lib\site-packages\ipykernel_launcher.py:3: Future Warning: `rcond` parameter will change to the default of machine precision times ``max(M, N)`` where M and N are the input matrix dimensions.
To use the future default and silence this warning we advise to pass `rcond=None`, to keep using the old, explicitly pass `rcond=-1`.
This is separate from the ipykernel package so we can avoid doing imports until

```
Out [805]: (array([ 1.28552097, -1.61163735,  1.87280108,  0.10554804]),
          array([9.03924222]),
          4,
          array([15.51703159,  3.48152053,  1.40440877,  0.35830352]))
```

I 선형대수학

이 자료는 혁신성장 청년인재 집중양성 사업 강의 자료로
개인 학습 자료로만 사용가능 합니다.

```
In [48]: xt=x.T
          xpx=np.dot(xt,x)
          np.dot(np.dot(np.linalg.inv(xpx),xt),y) # 최소제곱추정량
```

```
Out [48]: array([ 1.1991453 ,  0.44871795,  0.45854701, -0.35811966])
```

```
In [49]: A=np.array([[1,3,4,5],[2,5,4,8],[3,6,7,4],[6,7,2,1]])
          b=np.array([4,3,6,7])
          print("Ax=b의 해 x : {}".format(np.linalg.solve(A,b)))
```

```
Ax=b의 해 x : [ 6.7704918 -5.55737705  2.08196721  1.1147541 ]
```

난수 생성 - 확률분포 난수

함수	설명
random.seed()	난수 발생기의 시드값 설정
random.permutation()	순서를 임의로 바꾸거나 임의의 순열 반환
random.shuffle()	리스트나 배열의 순서 섞기
random.rand() random.uniform()	균등분포(0,1) / 균등분포(a,b)에서 난수 추출
random.randint()	주어진 최소~최대 범위 내에서 난수 추출
random.randn() random.normal()	표준정규분포(0,1)과 정규분포(μ, σ^2)에서 난수 추출
random.binomial()	이항분포 (n,p)에서 난수 추출
random.beta	베타분포(a, b)에서 0~1사이의 난수 추출
random.chisquare()	카이제곱분포(df)에서 난수 추출
random.gamma()	감마분포(k, θ)에서 난수 추출

```
In [824]: norm_sample=np.random.normal(loc=10, scale=2, size=(5,4)) #  $N(10, 2^2)$ 
          norm_sample
```

```
Out [824]: array([[ 9.71836612,  7.81827815,  8.04571882, 10.04837975],
                  [ 9.05563971, 15.05704669,  6.77228096, 13.58098562],
                  [11.9513319 ,  8.85274566, 12.72204763, 10.81413283],
                  [ 9.24736834, 12.66691469,  5.50278645, 15.00919291],
                  [10.86949214, 10.83952032,  9.04214954, 10.72951332]])
```

```
In [826]: np.random.seed(1234) # 시드값 고정
          a=np.random.uniform(low=0,high=10,size=4) #  $U(0, 10)$ 
          b=np.random.binomial(n=10, p=0.3, size=(3,4)) #  $B(10, 0.3)$ 
          c=np.random.chisquare(df=4,size=(2,5)) #  $\chi^2(4)$ 
          d=np.random.gamma(shape=1, scale=1/4, size=5) #  $G(1, 1/4)$ 
          print("난수추출 from 분포\n균등분포\n{0}\n이항분포\n{1}\n카이제곱분포\n{2}\n감마분포\n{3}").format(a,b,c,d)
```

```
난수추출 from 분포
균등분포
[1.9151945  6.22108771  4.37727739  7.85358584]
이항분포
[[4 2 2 4]
 [6 5 2 3]
 [4 4 2 3]]
카이제곱분포
[[2.54301505  3.33880592  8.04050259  0.72212863  3.8581016 ]
 [4.97622925  5.39954513  0.49863571  2.40801918  4.28270762]]
감마분포
[0.06172849  0.64712594  0.14591215  0.60009347  0.01541812]
```