Ch6. NumPy (1) - Basic

Numpy (http://numpy.org)는 Numerical Python의 줄임말로, 고성능의 과학계산 컴퓨팅과 데이터 분석에 필요한 기본 패키지이다. Numpy의 특징은 다음과 같다.

- C 언어로 구현된 python 라이브러리
- 빠르고 효율적인 다차원 배열 객체 ndarray 로 데이터를 관리
- 디스크로부터 배열 기반의 데이터를 읽거나 쓸 수 있는 도구
- 벡터 및 행렬 연산에 있어서 매우 편리한 기능을 제공
- 선형대수 계산, 푸리에 변환, 난수 발생기
- 데이터분석을 할 때 사용되는 라이브러리인 pandas와 matplotlib의 기반이 됨
- Python과 C, C++ 그리고 Fotran 코드를 통합하는 도구

Numpy는 Python에 빠른 배열 처리 기능을 제공하며, 데이터 분석에서는 알고리즘에 사용할 데이터 컨테이너 (container)의 역할을 한다.

만약 데이터가 수치 데이터(numerical data) 라면, Numpy의 배열(array)은 Python의 기본 자료 구조보다 훨씬 효율적인 방법으로 데이터를 다룰 수 있게 해준다.

Table of Contents

1. 설치와 실행

아나콘다 (anaconda)를 설치한 경우, NumPy 와 같이 자주 사용되는 라이브러리가 함께 자동 설치 되기 때문에 import 하여 사용할 수 있다. 만일 직접 라이브러리를 설치하고자 하는 경우 아래와 같은 방법을 따르면 된다.

1.1. 라이브러리 설치

NumPy 라이브러리를 설치해야 사용 가능하다.
파이썬 라이브러리 설치는 설치 관리자인 pip, conda를 통해 할 수 있다.

- pip (Package Installer for Python) : pip는 python에 한정된 패키지 관리자
- conda : conda는 다른 언어 c, java 등 포함한 패키지 관리자, 가상 환경 생성 포함하는 패키지 관리자, 어떤 OS에서든 패키지 및 종속을 빠르게 설치한다는 장점이 있음

1) cmd 창을 이용하는 경우

In [ ]:
```
### pip를 사용한 패키지 설치와 관리 (cmd 창 이용)

pip install numpy                    # 패키지 설치
pip install numpy --upgrade          # 패키지 업데이트
pip show numpy                       # 패키지 정보
pip unistall numpy                   # 패키지 삭제
pip list                             # 설치된 패키지 리스트 조회
```

In [ ]:
```
# conda를 사용한 패키지 설차와 관리 (cmd 창 이용)

conda install numpy                  # 패키지 설치
conda update -numpy                  # 패키지 업데이트
conda remove -numpy                  # 패키지 삭제
conda list                           # 설치된 패키지 리스트 조회
conda activate                       # 가상 환경 활성화
conda deactivate                     # 가상환경 비활성화
```

2) 주피터 노트북을 이용하는 경우

- 기존 pip, conda 앞에 느낌표(!)를 사용

In [ ]:
```
!pip install numpy
!pip install numpy --upgrade
!pip show numpy
!pip list
```

In [ ]:
```
!conda list
```

1.2. 라이브러리 실행

- 설치된 라이브러리를 사용하기 위해서는 반드시 불러오는 과정을 거쳐야 하며 import를 사용한다.

    import 라이브러리명

In [2]:
```
# 라이브러리 불러오기
import numpy
import numpy as np
```

In [ ]:
```
#버전 확인
np.__version__
```

2. 다차원 배열 객체 (ndarray)

NumPy의 핵심 기능 중 하나는 N차원 배열 객체 또는 ndarray로, 파이썬에서 사용할 수 있는 대규모 데이터 집합을 담을 수 이는 빠르고 유연한 구조이다. 다차원 배열 객체를 다루기에 앞서 몇가지 특징을 알아보자.

1. ndarray의 모든 원소는 같은 자료형 (type)이어야만 한다.

In [47]:
```
A= [1, 'A',True]
```

In [48]:
```
A
```

Out[48]:
```
[1, 'A', True]
```

In [49]:
```
#모든 원소를 문자열로 인식
arr1 = np.array([1, 'A',True])
arr1
```

Out[49]:
```
array(['1', 'A', 'True'], dtype='<U11')
```

2. 사용되는 자료형은 아래와 같다.

> NumPy는 수치해석을 위한 라이브러리인 만큼 숫자형 자료형에 대해서는 파이썬 내장 숫자 자료형에 비해 더 세부적으로 구분해 놓았다. (자료형 뒤에 붙는 숫자는 몇 비트 크기 인지를 의미함)

- 부호가 있는 정수 int(8, 16, 32, 64)
- 부호가 없는 양의 정수 uint(8 ,16, 32, 64)
- 실수float(16, 32, 64, 128)
- 복소수 complex(64, 128, 256)
- 불 bool
- 문자열 string_
- 파이썬 오프젝트 object
- 유니코드 unicode_

3. 배열에 사용하는 기본 함수/매서드

Operation	Description
size	배열 원소 수
ndim	배열의 차원 수
shape	각 차원의 크기를 tuple로 표시

dtype	생성한 배열의 데이터형 확인
astype	자료형 변환
reshape	배열의 shape(각 차원의 크기) 변경

2.1. ndarray 생성

NumPy의 array 라는 함수에 리스트(list)를 넣으면 배열로 변환해 준다.

1차원 배열

In [50]:
```
data1 = [6, 7.6, 8, 0, 1,2]
data1
```

Out[50]:
```
[6, 7.6, 8, 0, 1, 2]
```

In [51]:
```
arr1 = np.array(data1)
arr1
```

Out[51]:
```
array([6. , 7.6, 8. , 0. , 1. , 2. ])
```

In [52]:
```
type(arr1)
```

Out[52]:
```
numpy.ndarray
```

In [54]:
```
## 배열의 차원, 크기 정보
print(arr1.size)
print(arr1.ndim)
print(arr1.shape)
```

```
6
1
(6,)
```

In [55]:
```
print(arr1.dtype)
```

```
float64
```

In [56]:
```
# 배열의 자료형을 직접 지정하여 배열을 생성할 수 있다.
arr2 = np.array(data1, dtype="int")
arr2
```

Out[56]:
```
array([6, 7, 8, 0, 1, 2])
```

In [57]:
```
# 배열의 자료형 확인
arr2.dtype
```

Out[57]:
```
dtype('int32')
```

In [58]:
```
# 배열의 데이터형 변환
print(arr1.astype(str))
print(arr1.astype(int))
```

```
print(arr1.astype(bool))
```

```
['6.0' '7.6' '8.0' '0.0' '1.0' '2.0']
[6 7 8 0 1 2]
[ True  True  True False  True  True]
```

2차원 배열

2차원 배열은 행렬(matrix)라고도 하는데, 행렬에서의 가로를 행(row), 세로를 열(column) 이라고 한다.

2차원 배열은 **리스트의 리스트 (list of list)**를 이용해서 만든다. 안쪽 리스트의 길이는 열의 수를 나타내고, 바깥쪽 리스트 의 길이는 행의 수를 나타낸다.

In [59]:
```
data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
arr2 = np.array(data2)
arr2
```

Out[59]:
```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

In [42]:
```
print(arr2.ndim)
print(arr2.shape)
print(arr2.size)
```

```
2
(2, 4)
8
```

In [43]:
```
# 각 차원의 크기 shape 변경
arr3=arr2.reshape(4,2)
arr3
```

Out[43]:
```
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
```

In [44]:
```
arr4=arr2.reshape(2, 2, 2)
arr4
```

Out[44]:
```
array([[[1, 2],
        [3, 4]],

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

In [45]:
```
arr5=arr2.reshape(-1, 1)   # -1: 원래 배열의 길이와 다른 차원의 크기로 부터 추정이 됨을 뜻함
arr5
```

Out[45]:
```
array([[1],
       [2],
       [3],
```

```
        [4],
        [5],
        [6],
        [7],
        [8]])
```

3차원 배열

3차원 배열은 **리스트의 리스트의 리스트 (list of list of list)**를 이용한다. 크기를 나타낼 경우에는 가장 바깥쪽 리스트의 길이부터 가장 안쪽 리스트의 길이의 순서로 표시한다. 예를 들어, 2 x 2 x 3 배열은 다음과 같다.

In [40]:
```
# 2 x 2 x 3 array
data3 = [[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]]
arr3 = np.array(data3)
arr3
```

Out[40]:
```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

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

In [36]:
```
print(arr3.ndim)
print(arr3.shape)
```

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

여러가지 배열 생성 방법

Operation	Description
zeros	주어진 길이나 모양에 0으로 이루어진 배열 생성
ones	주어진 길이나 모양에 1으로 이루어진 배열 생성
empty	ones, zeros와 비슷하나 값을 초기화하지는 않음
zeros_like / ones_like /empty_like	이미 있는 array와 동일한 모양과 데이터 형태를 유지한 상태에서 각각 '0', '1', '빈 배열' 생성
arange	range 함수와 유사하나 ndarray형태로 배열 생성
eye / identity	항등행렬 생성
rand/ randn	표준 정규 분포에서 랜덤 샘플링 후 배열 생성/ unif(0,1)에서 랜덤 샘플링 후 배열생성

In [33]:
```
"""
zeros : 주어진 길이나 모양으로 0으로 이루어진 배열 생성
"""
np.zeros(5)
```

Out[33]:
```
array([0., 0., 0., 0., 0.])
```

In [34]:
```
np.zeros((2,2))
```

Out[34]:
```
array([[0., 0.],
       [0., 0.]])
```

In [114]:
```
"""
ones : 주어진 길이나 모양으로 1으로 이루어진 배열 생성
"""
np.ones((2,3))
```

Out[114]:
```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

In [32]:
```
"""
zeros_like, ones_like, empty_like, full_like: 특정 배열이나 리스트와 같은 길이나 모양을 만들되 요소에 각각 0과 1, 빈 배열, 특정값의 배열 생성
"""
arr1
```

Out[32]:
```
array([6. , 7.6, 8. , 0. , 1. , 2. ])
```

In [18]:
```
np.zeros_like(arr1)
```

Out[18]:
```
array([0., 0., 0., 0., 0., 0.])
```

In [19]:
```
np.ones_like(arr1)
```

Out[19]:
```
array([1., 1., 1., 1., 1., 1.])
```

In [25]:
```
np.empty_like(arr1)
```

Out[25]:
```
array([4., 4., 4., 4., 4., 4.])
```

In [26]:
```
np.full_like(arr1,4)
```

Out[26]:
```
array([4., 4., 4., 4., 4., 4.])
```

In [80]:
```
"""
empty : 메모리를 할당하여 새로운 배열을 생성하는데,
생성된 배열에 어떤 값이 들어가 있는지는 알 수 없음.
주로, 배열을 초기화 시킬 때 사용
"""
np.empty((2, 3, 3))    # 3차원의 배열
```

Out[80]:
```
array([[[6.23042070e-307, 4.67296746e-307, 1.69121096e-306],
        [1.33511018e-306, 1.15711174e-306, 1.11256817e-306],
        [1.06811422e-306, 1.42417221e-306, 1.11260619e-306]],
```

```
       [[8.90094053e-307, 1.86919378e-306, 1.06809792e-306],
        [1.37962456e-306, 1.69111861e-306, 1.78020169e-306],
        [1.37961777e-306, 7.56599807e-307, 1.60995121e+295]]])
```

In [81]:
```
"""
arange : Numpy 버젼의 range 함수
특정한 규칙에 따라 증가하는 수열을 만듦
"""
np.arange(10)
```

Out[81]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [82]:
```
np.arange(3, 21, 2)
```

Out[82]:
```
array([ 3,  5,  7,  9, 11, 13, 15, 17, 19])
```

난수생성

배열을 만드는 또 다른 방법으로 무작위 난수 생성하여 배열을 만들 수 있다. 난수는 보통 아래와 같은 경우 필요하다.

- 시간과 비용 문제로 전수 조사를 못하므로 표본 조사를 해야 할 때
- 기계학습 할 때 데이터셋을 훈련용/검증용/테스트용으로 샘플링 할 때
- 다양한 확률 분포로 부터 데이터를 무작위로 생성해서 시뮬레이션(simulation) 할 때

먼저, 임의로 추출되기 때문에 난수는 생성할 때마다 다른 값을 추출하게 된다. 언제 시뮬레이션 하는지와 관계 없이 동일한 값을 추출하고자 하는 경우 seed를 사용해서 난수 생성 초기값 부여해야 한다.

In [131]:
```
np.random.seed(100)            # random seed 번호 지정
```

다양한 분포에서 난수를 생성할 수 있지만, 우리는 균일 분포 (uniform distribution)와 표준 정규 분포 (standard normal distribution)를 대표적으로배워 보겠다.

In [86]:
```
"""
rand / randn :  각각 uniform distribution, standard normal distribution 에서
랜덤 데이터를 생성
"""
np.random.rand(10)          # uniform dist 에서 10개 랜덤 추출
```

Out[86]:
```
array([0.47991799, 0.52782122, 0.09015178, 0.11230965, 0.33832805,
       0.94422707, 0.34581001, 0.04045125, 0.19823081, 0.24308086])
```

In [106]:
```
np.random.seed(100)
np.random.randn(10)          # normal
```

Out[106]:
```
array([-1.74976547,  0.3426804 ,  1.1530358 , -0.25243604,  0.98132079,
        0.51421884,  0.22117967, -1.07004333, -0.18949583,  0.25500144])
```

In [107]:
```
np.random.randn(3, 5)       #2차원 배열
```

Out[107]:
```
array([[-0.45802699,  0.43516349, -0.58359505,  0.81684707,  0.67272
081],
       [-0.10441114, -0.53128038,  1.02973269, -0.43813562, -1.11831
825],
       [ 1.61898166,  1.54160517, -0.25187914, -0.84243574,  0.18451
869]])
```

In [108]:
```
#정규분포를 활용하고 싶은 경우
mu, sigma = 0.0, 3.0
```

```
print(np.random.normal(loc=mu, scale=sigma, size=10))   #loc는 mean을, sc
ale는 sigma를 의미
```

```
[ 2.8112466   2.19300103  4.08466838 -0.97871418  0.16702804  0.6671
9883
 -4.32965099 -2.26905692  2.44936203  2.25133428]
```

2.2 인덱싱 (indexing) 과 슬라이싱 (slicing)

- NumPy 슬라이싱 (slicing)은 각 배열 차원별 최소-최대의 범위를 정하여 부분 집합을 구하는 것이며
- 인덱싱 (indexing)은 각 차원별로 선택되어지는 배열요소의 인덱스들을 일렬로 나열하여 부분집합을 구하는 방식이다.

1차원배열

1차원 배열은 기존 리스트, 문자열, 튜플에서의 방법과 동일하게 인덱싱(indexing) 과 슬라이싱(slicing) 할 수 있다.

In [110]:
```
arr = np.arange(10)
arr
```

Out[110]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [111]:
```
print(arr[-1])
print(arr[0:5])
```

```
9
[0 1 2 3 4]
```

In [112]:
```
arr[9] = 19
arr
```

Out[112]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8, 19])
```

다차원 배열

2차원 이상 다차원 배열의 경우에는 콤마(comma ,)를 이용하여 원소 하나하나에 접근할 수 있다. 콤마로 구분된 차원을 축(axis)라고 한다.

In [113]:
```
# 2차원 array
arr2 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr2
```

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

In [114]:
```
arr2[0]
```

Out[114]:
```
array([1, 2, 3])
```

In [115]:
```
arr2[0, 1]
```

Out[115]:
```
2
```

In [116]:
```
arr2[0][1]
```

Out[116]:
```
2
```

In [158]:
```
arr2[0:2, 0:2]
```

Out[158]:
```
array([[1, 2],
       [4, 5]])
```

In [159]:
```
arr2[:, 1]
```

Out[159]:
```
array([2, 5, 8])
```

In [160]:
```
arr2[0, :]
```

Out[160]:
```
array([1, 2, 3])
```

아래의 **팬시 인덱싱** *(fancy indexing)*은 정수 배열을 사용하는 인덱싱 방법이다.

In [122]:
```
arr2[[0,2]]
```

Out[122]:
```
array([[1, 2, 3],
       [7, 8, 9]])
```

In [174]:
```
arr2[[0,2],[1,1]]
```

Out[174]:
```
array([2, 8])
```

In [175]:
```
# 3차원 array
arr3 = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
arr3
```

Out[175]:
```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

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

In [176]:
```
arr3[0]
```

Out[176]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [177]:
```
arr3[-1]
```

Out[177]:
```
array([[ 7,  8,  9],
       [10, 11, 12]])
```

In [179]:
```
arr3[0][0:, 0:]
```

Out[179]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [178]:
```
arr3[0][0:2, 0:2]
```

Out[178]:
```
array([[1, 2],
       [4, 5]])
```

불리언 인덱싱 (Boolean indexing)

bool 자료형을 이용해 indexing 하는 것도 가능하다.

불리언(boolean) 배열 인덱싱 방식은 인덱스 배열의 원소가 True, False 두 값으로만 구성되며 인덱스 배열의 크기가 원래 ndarray 객체 크기와 같아야 한다.
True 위치에 해당하는 요소만 선택한다.

In [186]:
```
# example1
```

In [126]:
```
arr = np.array([
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]
])
```

In [127]:
```
b_arr = np.array([[False, True, False], [True, False, True], [False,
 True, False]])
```

In [128]:
```
print(arr[b_arr])
b_arr2 = (arr%2==0)
print(b_arr2)
print(arr[b_arr2])
print(arr[arr%2==0])
```

```
[2 4 6 8]
[[False  True False]
 [ True False  True]
 [False  True False]]
[2 4 6 8]
[2 4 6 8]
```

In [188]:
```
# example2
```

In [130]:
```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe']
)
```

In [131]:
```
names == 'Bob'
```

Out[131]:
```
array([ True, False, False,  True, False, False, False])
```

In [132]:
```
names[names == 'Bob']
```

Out[132]:
```
array(['Bob', 'Bob'], dtype='<U4')
```

In [133]:
```
#난수 생성
data = np.random.randn(7, 4)
data
```

Out[133]:
```
array([[-0.45594693,  1.18962227, -1.69061683, -1.35639905],
       [-1.23243451, -0.54443916, -0.66817174,  0.00731456],
       [-0.61293874,  1.29974807, -1.73309562, -0.9833101 ],
       [ 0.35750775, -1.6135785 ,  1.47071387, -1.1880176 ],
       [-0.54974619, -0.94004616, -0.82793236,  0.10886347],
       [ 0.50780959, -0.86222735,  1.24946974, -0.07961125],
       [-0.88973148, -0.88179839,  0.01863895,  0.23784462]])
```

In [134]:
```
data[names == 'Bob'] #7개의 행중에서, 1행 4행 선택됨
```

Out[134]:
```
array([[-0.45594693,  1.18962227, -1.69061683, -1.35639905],
       [ 0.35750775, -1.6135785 ,  1.47071387, -1.1880176 ]])
```

In [135]:
```
data[names == 'Bob', 2:]  #slicing
```

Out[135]:
```
array([[-1.69061683, -1.35639905],
       [ 1.47071387, -1.1880176 ]])
```

In [136]:
```
names != 'Bob'
```

Out[136]:
```
array([False,  True,  True, False,  True,  True,  True])
```

In [137]:
```
data[names != 'Bob', 3] #2,3,5,6,7 행 중에서 index3에 해당하는 값들
```

Out[137]:
```
array([ 0.00731456, -0.9833101 ,  0.10886347, -0.07961125,  0.23784462])
```

논리 / 비교 연산자 (==, !=, <, <=, >, >= etc) 를 이용하면 원하는 조건에 맞는 요소만 선택할수 있다.

In [138]:
```
data[data < 0]
```

Out[138]:
```
array([-0.45594693, -1.69061683, -1.35639905, -1.23243451, -0.54443916,
       -0.66817174, -0.61293874, -1.73309562, -0.9833101 , -1.6135785 ,
       -1.1880176 , -0.54974619, -0.94004616, -0.82793236, -0.86222735,
       -0.07961125, -0.88973148, -0.88179839])
```

In [140]:
```
data[data < 0] = 0
data
```

```
Out[140]: array([[0.        , 1.18962227, 0.        , 0.        ],
                 [0.        , 0.        , 0.        , 0.00731456],
                 [0.        , 1.29974807, 0.        , 0.        ],
                 [0.35750775, 0.        , 1.47071387, 0.        ],
                 [0.        , 0.        , 0.        , 0.10886347],
                 [0.50780959, 0.        , 1.24946974, 0.        ],
                 [0.        , 0.        , 0.01863895, 0.23784462]])
```

Ch6. NumPy (2) - 연산과 데이터 전처리

배열(array)의 연산 부분은 배열이 만들어진 가장 근본적인 이유이다.

Python 기본 자료형이 할 수 없는 연산을 배열을 이용하면 할 수 있다.

In [2]:
```
import numpy as np
```

Table of Contents

1. 연산자를 이용한 연산

NumPy는 숫자형 배열에 대해서 산술 연산을 할 때 파이썬 기본 자료형과는 달리 'for loops'를 사용하지 않고 연산을 하기 때문에 연산 속도가 매우 빠르다.

배열과 스칼라간의 연산

In [1]:
```
# 기존 list의 원소를 2배 해야하는 작업
a = [0, 1, 2, 3, 4, 5]
res = []

for i in a :
    res.append(i*2)

res
```

Out[1]:
```
[0, 2, 4, 6, 8, 10]
```

In [10]:
```
#같은 작업을 array를 이용할 경우
np.array(a) * 2
```

Out[10]:
```
array([ 0,  2,  4,  6,  8, 10])
```

In [11]:
```
print(np.array(a) +1)
print(np.array(a) / 2)
print(np.array(a) // 2)
print(np.array(a)%2)
print(np.array(a)**2)
```

```
[1 2 3 4 5 6]
[0.  0.5 1.  1.5 2.  2.5]
[0 0 1 1 2 2]
[0 1 0 1 0 1]
[ 0  1  4  9 16 25]
```

배열 간의 연산

크기가 같은 두 개의 숫자형 배열에 대해서 산술 연산

In [14]:
```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
```

In [15]:
```
print(a+b)
print(a-b)
print(a*b)
print(a/b)
print(a//b)
print(a%b)
print(a**b)
```

```
[5 7 9]
[-3 -3 -3]
[ 4 10 18]
[0.25 0.4  0.5 ]
[0 0 0]
[1 2 3]
[  1  32 729]
```

2. 유니버설 함수 (Universal functions)를 이용한 연산

유니버설 함수는 ndarray 안에 있는 데이터 원소별로 연산을 수행하는 함수이다.

주요 단항 연산 함수

Operation	Description
abs, fabs	절댓값
sqrt	제곱근

exp	지수함수
log / log10 / log2	자연로그 / 상용로그 / 밑이 2인 로그
sign	부호
ceil / floor/ rint 혹은 around	소수 절단 함수 (올림/ 내림/ 반올림)
sin, cos, tan	삼각 함수
modf	배열 원소의 정수와 소수점을 구분하여 2개의 배열 반환
gradient	배열 원소 간 기울기 구하기 (1차 편미분한 값들로 구성된 배열)
prod	배열 원소 간 곱
sum	배열 원소 간 합
diff	배열 원소 간 차분

In [27]:
```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = np.array([-1, 3, -4])
d = np.array([4.5, 6.3, 7.8, 4.4, 3])
```

In [41]:
```
print(np.abs(c))
print(np.fabs(c))      # abs와 유사
print(np.sqrt(b))      # b**0.5 와 동일
print(np.square(a))    # a**2 와 동일
```

```
[1 3 4]
[1. 3. 4.]
[2.         2.23606798 2.44948974]
[1 4 9]
```

In [42]:
```
print(np.exp(a))
print(np.log(a))
print(np.log10(a))
print(np.log2(a))
print(np.sign(c))
```

```
[ 2.71828183  7.3890561  20.08553692]
[0.         0.69314718 1.09861229]
[0.         0.30103    0.47712125]
[0.         1.         1.5849625]
[-1  1 -1]
```

In [60]:
```
print(np.ceil(d))
print(np.floor(d))
print(np.rint(d))
```

```
[5. 7. 8. 5. 3.]
[4. 6. 7. 4. 3.]
[4. 6. 8. 4. 3.]
```

In [61]:
```
print(np.sin(0))
print(np.cos(np.pi))
```

```
print(np.tan(0))
```

```
0.0
-1.0
0.0
```

In [45]:
```
print(np.modf(d))
print(np.modf(d)[0])
print(np.modf(d)[1])
```

```
(array([0.5, 0.3, 0.8, 0.4, 0. ]), array([4., 6., 7., 4., 3.]))
[0.5 0.3 0.8 0.4 0. ]
[4. 6. 7. 4. 3.]
```

In [48]:
```
# np.gradient 1차 미분 계산
# a, b, c where the gradient of b is (c-a)/2, the gradient of c is: (c-b)/1 , and the gradient of a is (b-a)/1

# 1-dim array
e = np.array([1, 2, 4, 10, 13, 20])
print(np.gradient(e)) # 1차 편미분:[(2-1), {(2-1)+(4-2)}/2, {(4-2)+(10-4)}/2, {(10-4)+(13-10)}/2, {(13-10)+(20-13)}/2, (20-13)]
```

```
[1.  1.5 4.  4.5 5.  7. ]
```

In [16]:
```
# 2-dim array
f= np.array([[1, 2, 4, 8], [10, 13, 20, 15]])

np.gradient(f)              # 1st array : the gradient in rows direction ,  2nd array : in colums direction
```

Out[16]:
```
[array([[ 9., 11., 16.,  7.],
        [ 9., 11., 16.,  7.]]), array([[ 1. ,  1.5,  3. ,  4. ],
        [ 3. ,  5. ,  1. , -5. ]])]
```

In [17]:
```
f
```

Out[17]:
```
array([[ 1,  2,  4,  8],
       [10, 13, 20, 15]])
```

In [19]:
```
print(np.gradient(f, axis=0))
print(np.gradient(f, axis=1))
```

```
[[ 1.   1.5  3.   4. ]
 [ 3.   5.   1.  -5. ]]
```

주요 다항 연산 함수

Operation	Description
add	두 배열의 합
subtract	두 배열의 차
multuiply, prod	두 배열의 곱
divide	두 배열 나눗셈
power	두 배열 거듭 제곱
mod	첫 번째 배열의 원소를 두 번째 배열 원소로 나눈 나머지
maximum / minimum	두 배열의 원소들을 각각 비교하여 큰 값 / 작은 값 추출

In [30]:
```
print(a)
print(b)
```

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

In [24]:
```
print(np.add(a, b))
print(np.subtract(a, b))
print(np.multiply(a, b))
print(np.divide(a, b))
```

```
[5 7 9]
[-3 -3 -3]
[ 4 10 18]
[0.25 0.4  0.5 ]
```

In [29]:
```
print(np.power(a, b))
print(np.mod(b, a))
print( np.maximum(a, c))
print( np.minimum(a, c))
```

```
[  1  32 729]
[0 1 0]
[1 3 3]
[-1  2 -4]
```

3. 배열을 사용한 데이터 전처리

NumPy 배열을 사용하면 반복문을 작성하지 않고 간결한 배열연산을 통해 많은 종류의 데이터 처리 작업을 할 수 있다.

배열연산을 사용해서 반복문을 명시적으로 제거하는 기법을 흔히 벡터화(vectorized)라고 부른다.

3.1. 차원 축소 연산

행렬의 하나의 행에 있는 원소들을 하나의 데이터 집합으로 보고 그 집합의 평균을 구하면 각 행에 대해 하나의 숫자가 나오게 된다.

예를 들어, 10 x 5 크기의 2차원 배열에 대해 행-평균을 구하면 10개의 숫자를 가진 1차원 벡터가 나오게 된다. 이러한 연산을 차원 축소 (dimension reduction) 연산이라고 한다.

차원 축소 연산 매서드

Operation	Description
min, max, argmin, argmax	최대값, 최소값, 최소값이 위치한 index, 최대값이 위치한 index
sum, mean, median, var, std, corrcoef	전체 원소 합, 평균, 중앙값, 분산, 표준편차, 상관계수
cumsum, cumprod	누적합, 누적곱

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

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

In [4]:
```
print(np.max(arr))
print(np.min(arr))          # arr.min()와 동일
print(np.argmin(arr))       # arr.argmin() 와 동일
```

```
4
1
0
```

In [34]:
```
print( arr.mean())
print( arr.std())
print( arr.var())
```

```
2.5
1.118033988749895
1.25
```

In [85]:
```
print( arr.cumsum())
print( arr.cumprod())
```

```
[ 1  3  6 10]
[ 1  2  6 24]
```

- 연산의 대상이 2차원 이상인 경우에는 어느 차원으로 계산할지 axis 인수를 사용하여 지시한다.
- 2차원인 경우, axis = 0 인 경우는 행 (row) 방향 연산, axis = 1 인 경우에는 열 (column) 방향 연산을 한다.

In [86]:
```
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
arr2
```

Out[86]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [88]:
```
print( np.sum(arr2, axis=0))
print( arr2.sum(axis=1))
```

```
[5 7 9]
[ 6 15]
```

3.2. 불리언 배열을 위한 매서드 (method)

In [94]:
```
print(int(True))
print(int(False))
```

```
1
0
```

In [97]:
```
print(bool(0))
print(bool(1))
print(bool(-123))
```

```
False
True
True
```

- sum 매서드를 입히면 조건을 만족하는 원소의 개수를 세준다.

In [35]:
```
arr = np.random.randn(20)
```

In [36]:
```
arr
```

Out[36]:
```
array([-0.55989018, -0.03760781,  0.14896506,  0.7395734 ,  0.995583
14,
       -0.17403201, -0.47094374, -0.36462523, -0.60559826,  0.413694
39,
        0.09449215,  1.04186612,  0.78362064,  0.14679515, -0.067422
82,
        0.94123799,  1.47319092,  0.93228214,  1.02675149,  0.526831
73])
```

In [37]:
```
arr > 0
```

Out[37]:
```
array([False, False,  True,  True,  True, False, False, False, False
, True,  True,  True,  True,  True, False,  True,  True, True , True,
True])
```

In [102]:
```
(arr > 0).sum()
```

Out[102]:
```
9
```

- any 매서드는 적어도 하나 이상의 True가 존재하는지를 평가하고,
  all 매서드는 모든 원소가 다 True인지를 평가한다.

In [38]:
```
bools = np.array([False, False, True, False])
bools
```

Out[38]:
```
array([False, False,  True, False])
```

In [106]:
```
np.any(bools)        # 적어도 하나 이상의 원소가 True 이냐?
```

Out[106]:
```
True
```

In [105]:
```
np.all(bools)        # 모든 원소가 True 이냐?
```

Out[105]:
```
False
```

참고 논리 연산 함수

Operation	Description
isnan	배열 원소가 NaN이면 True 반환
isinf	배열 원소가 무한이면 True 반환
isfinite	배열 원소가 유한이면 True 반환
equal	두 배열의 원소가 동일하면 True 반환
any	True가 하나라도 존재하면 True 반환
all	모든 원소가 True이면 True 반환
where	조건절, 해당하는 조건을 만족하는 요소 위치 반환 혹은 조건 만족여부에 따라 요소값 변경

In [157]:
```
a = np.array([0, 1, 2, np.nan, 4, np.inf, np.NINF, np.PINF])
print(a)
print(np.isnan(a))
print(np.isinf(a))
print(np.isfinite(a))
```

```
[  0.   1.   2.  nan   4.  inf -inf  inf]
[False False False  True False False False False]
[False False False False False  True  True  True]
```

```
[ True  True  True False  True False False False]
```

In [170]:
```
a = np.array([4, 2, 6, 3, 2, 7, 8])
b = np.array([4, 5, 6, 4, 2, 8, 7])
c = np.array([4, 5, 6, 4, 2, 8, 7])
```

In [163]:
```
print(np.equal(a, b))
print(np.not_equal(a, b))
print(np.array_equal(b, c))
```

```
[ True False  True False  True False False]
[False  True False  True False  True  True]
True
```

In [176]:
```
# 알아두면 유용한 함수 where
a = np.array([4, 2, 6, 3, 2, 7, 8])

np.where(a < 5)            # 조건 만족하는 index 반환
```

Out[176]:
```
(array([0, 1, 3, 4], dtype=int64),)
```

In [177]:
```
np.where(a < 5 , 0, a)   # 조건을 만족하는 경우 0으로 요소를 변경하고, 만족하지 않는 경우 a의 값을 취함
```

Out[177]:
```
array([0, 0, 6, 0, 0, 7, 8])
```

In [181]:
```
# 판단 연산
p = "P"
q = np.array(["P", "Q"])
r = np.array(["R", "S"])
print(p in q)
print(p in r)
print(p not in q)
print(p not in r)
```

```
True
False
False
True
```

3.3. 배열 전치와 재형성

하나의 배열을 이용하여, 배열의 모양을 바꾸고, 요소를 반복하는 방법등에 대해서 알아보자.

배열전치

전치(transpose) 란 2차원 배열의 행 (row)과 열 (column)을 서로 바꾸는 것을 말한다. 이는 배열의 T 속성으로 가능하다.

배열 모양 재형성

배열의 모양을 변경하려면 reshape 매서드 혹은 resize 매서드를 이용해서 새로운 모양을 나타내는 튜플 (tuple)을 적용하면 된다.

Operation	Description
reshape	배열 형태 변경 (원본 객체변경없음)
resize	배열 형태 변경 (원본 객체변경)

In [5]:
```
arr = np.arange(15)
arr
```

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

In [8]:
```
arr1 = arr.reshape((3,5))
arr1
```

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

In [9]:
```
arr.resize((3,5))
arr
```

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

In [10]:
```
arr2=arr.T
```

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

In [11]:
```
arr1.transpose()    #.T와 동일한 기능을 한다
```

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

배열 1차원으로 펼치기

다차원 배열을 무조건 1차원으로 펼치기 위해서는 flatten 명령어를 사용하면 된다.

In [13]:
```
arr1 = arr.reshape((3,5))
arr1
```

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

```
In [14]: arr1.flatten()
Out[14]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

* Tips: 배열 사용에서 주의할 점은 길이가 5인 1차원 배열과 크기가 1 x 5 인 행렬(matrix)은 완벽히 다르다는 것이다.

```
In [15]: x = np.arange(5)
         x
Out[15]: array([0, 1, 2, 3, 4])

In [16]: y=x.reshape((1, 5))
         y
Out[16]: array([[0, 1, 2, 3, 4]])
```

배열 원소 반복

큰 배열을 만들기 위해서 배열을 반복하거나 복제하는 함수로 tile을 사용할 수 있다.

```
In [25]: arr = np.arange(3)
         arr
Out[25]: array([0, 1, 2])

In [26]: np.tile(arr, 2)
Out[26]: array([0, 1, 2, 0, 1, 2])

In [27]: np.tile(arr, (1,2))
Out[27]: array([[0, 1, 2, 0, 1, 2]])

In [64]: np.tile(arr, (2,1))
Out[64]: array([[0, 1, 2],
                [0, 1, 2]])

In [66]: np.tile(arr, (2, 2))
Out[66]: array([[0, 1, 2, 0, 1, 2],
                [0, 1, 2, 0, 1, 2]])
```

3.4. 배열 결합과 분리

배열 결합

두 배열을 결합하는데 사용되는 함수는 아래와 같다.

Operation	Description
concatenate	하나의 축에 따라 이어 붙인다
vstack	배열을 axis=0 을 따라 붙이기 (2차원의 경우 수직 방향)
hstack	배열을 axis=1 을 따라 붙이기 (2차원의 경우 수평 방향)
dstack	배열을 axis=2 을 따라 붙이기

In [3]:
```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = np.array([[0, 1, 2], [3, 4, 5]])
d = np.array([[6, 7, 8], [9, 10, 11]])
```

In [4]:
```
print(np.hstack((a, b)))
print(np.vstack((a, b)))
```

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

In [20]:
```
print(np.concatenate((a, b)))        # arrays are flattened
print(np.concatenate((c, d)))        # default : axis=0
print(np.concatenate((c,  d),  axis=0))
print(np.concatenate((c, d),   axis=1))
```

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

In [88]:
```
# 2-dim array dstack
print(c) ; print(d)
```

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

[[ 6  7  8]
 [ 9 10 11]]
```

In [15]:
```
np.dstack((c, d))
```

Out[15]:
```
array([[[ 0,  6],
        [ 1,  7],
        [ 2,  8]],

       [[ 3,  9],
        [ 4, 10],
        [ 5, 11]]])
```

In [70]:
```
print(np.vstack((c,d)))
print(np.hstack((c,d)))
```

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

In [102]:
```
# 3-dim array
e = np.array([[ [0, 1, 2], [3, 4, 5] ]])
f = np.array([[ [6, 7, 8], [9, 10, 11] ]])
print(e)
print(f)
```

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

In [105]:
```
print(e.shape , f.shape)  # axis=0, axis=1, axis=2 순서대로의 길이
```

```
(1, 2, 3) (1, 2, 3)
```

In [82]:
```
print( np.dstack((e,f)) )
```

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

In [88]:
```
# 2-dim array dstack
print(c) ; print(d)
```

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

[[ 6  7  8]
 [ 9 10 11]]
```

In [99]:
```
print(np.dstack((c, d)))
 # print(np.dstack(( c.reshape(2,3,1), d.reshape(2,3,1))))
```

```
[[[ 0 6]
  [ 1 7]
  [ 2 8]]

 [[ 3 9]
  [ 4 10]
  [ 5 11]]]
```

배열 분리

Operation	Description
split	선택한 방향으로 분할
hsplit	(axis=1) 방향으로 분할
vsplit	(axis=0) 방향으로 분할

In [29]:
```
a = np.arange(1, 13).reshape((4, 3))
print(a)
```

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

In [34]:
```
np.split(a, 4)    # default : axis= 0
```

Out[34]:
```
[array([[1, 2, 3]]),
 array([[4, 5, 6]]),
 array([[7, 8, 9]]),
 array([[10, 11, 12]])]
```

In [35]:
```
np.split(a, 2)
```

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

In [36]:
```
np.split(a, 4, axis=0)
```

```
Out[36]: [array([[1, 2, 3]]),
          array([[4, 5, 6]]),
          array([[7, 8, 9]]),
          array([[10, 11, 12]])]
```

```
In [91]: np.split(a, 3, axis=1)
```

```
Out[91]: [array([[ 1],
                 [ 4],
                 [ 7],
                 [10]]), array([[ 2],
                 [ 5],
                 [ 8],
                 [11]]), array([[ 3],
                 [ 6],
                 [ 9],
                 [12]])]
```

```
In [95]: np.hsplit(a, 3)
```

```
Out[95]: [array([[ 1],
                 [ 4],
                 [ 7],
                 [10]]), array([[ 2],
                 [ 5],
                 [ 8],
                 [11]]), array([[ 3],
                 [ 6],
                 [ 9],
                 [12]])]
```

```
In [37]: np.vsplit(a, 2)
```

```
Out[37]: [array([[1, 2, 3],
                 [4, 5, 6]]), array([[ 7,  8,  9],
                 [10, 11, 12]])]
```

3.5. 배열 변경

operation	description
append	추가
insert	삽입
delete	삭제

append

```
In [25]: a = np.arange(1, 10).reshape(3, 3)
         b = np.arange(10, 19).reshape(3, 3)
         print(a)
```

```
print(b)
```

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

In [26]:
```
# axis 지정하지 않은 경우 - 1D로 변형
np.append(a, b)
```

Out[26]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 1
6, 17, 18])
```

In [27]:
```
# axis 지정
print(np.append(a, b, axis=0))
print(np.append(a, b, axis=1))
```

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

In [28]:
```
a = np.arange(1, 10).reshape(3, 3)
b = np.arange(10, 16).reshape(2, 3)
print(a)
print(b)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[10 11 12]
 [13 14 15]]
```

insert

In [48]:
```
a = np.arange(1, 10).reshape(3, 3)
print(a)
```

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

In [49]:
```
# axis 지정하지 않은 경우 - 1D로 변형
np.insert(a, 1, 999)      # 배열 a에 index1에 999 추가
```

Out[49]:
```
array([  1, 999,   2,   3,   4,   5,   6,   7,   8,   9])
```

In [110]:
```
# axis 지정
print(np.insert(a, 1, 999, axis=0))
print(np.insert(a, 1, 999, axis=1))
```

```
[[  1   2   3]
 [999 999 999]
 [  4   5   6]
 [  7   8   9]]
[[  1 999   2   3]
 [  4 999   5   6]
 [  7 999   8   9]]
```

delete

In [50]:
```
a = np.arange(1, 10).reshape(3, 3)
print(a)
```

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

In [111]:
```
# axis 지정하지 않은 경우 - 1D로 변형
np.delete(a, 1)
```

Out[111]:
```
array([1, 3, 4, 5, 6, 7, 8, 9])
```

In [112]:
```
# axis 지정
print(np.delete(a, 1, axis=0))
print(np.delete(a, 1, axis=1))
```

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

3.6. 정렬 (sort)

sort 명령이나 매서드를 사용하여 배열 안의 원소를 정렬(sorting) 시킬 수 있다.

2차원 이상인 경우에는 axis 인수를 사용하여 정렬시킬 방향을 결정시킨다.

즉, axis로 기존에 지정한 축 방향을 무시하여 정렬하라는 뜻이다. 디폴트 값은 -1로 가장 안쪽 차원이다.

In [52]:
```
a = np.array([2,1,3,4])
a
```

Out[52]:
```
array([2, 1, 3, 4])
```

In [53]:
```
np.sort(a)
```

```
Out[53]: array([1, 2, 3, 4])
```

```
In [54]: a = np.array([[4, 3, 5], [1, 2, 1]])
         a
```

```
Out[54]: array([[4, 3, 5],
                [1, 2, 1]])
```

```
In [55]: np.sort(a)
```

```
Out[55]: array([[3, 4, 5],
                [1, 1, 2]])
```

```
In [56]: np.sort(a, axis=0)
```

```
Out[56]: array([[1, 2, 1],
                [4, 3, 5]])
```

```
In [46]: np.sort(a, axis=1)
```

```
Out[46]: array([[3, 4, 5],
                [1, 1, 2]])
```

```
In [58]: a = np.array([[4, 3, 5], [1, 4, 1]])
         print(a)
         np.sort(a, axis=0)
```

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

```
Out[58]: array([[1, 3, 1],
                [4, 4, 5]])
```

- 만약 자료를 정렬하는 것이 아니라 순서(order)만 알고 싶다면 argsort 명령을 사용하면 된다.

```
In [59]: a = np.array([4, 3, 1, 2])
         j = np.argsort(a)          # 첫번째로 작은 원소는 index2에 위치, 두번째 작은 원소는 index 3에 위치 , ...
         j
```

```
Out[59]: array([2, 3, 1, 0], dtype=int64)
```

```
In [48]: a[j]
```

```
Out[48]: array([1, 2, 3, 4])
```

3.7. 집합 함수 (set)

NumPy는 1차원 ndarray를 위한 몇 가지 기본 집합연산을 제공한다.

배열 집합연산

Operation	Description
unique	고유한 원소만 반환
intersect1d	교집합
union1d	합집합
setdiff1d	차집합
in1d(x, y)	x 배열 원소가 y 배열에 포함되는지 여부

In [63]:
```
set1 = np.array([5, 2, 4, 1, 3])
set2 = np.array([1, 6, 8, 3, 5])
```

In [64]:
```
np.intersect1d(set1, set2)
```

Out[64]:
```
array([1, 3, 5])
```

In [65]:
```
np.union1d(set1, set2)
```

Out[65]:
```
array([1, 2, 3, 4, 5, 6, 8])
```

In [66]:
```
print(np.setdiff1d(set1, set2))
print(np.setdiff1d(set2, set1))
```

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

In [53]:
```
print(np.in1d(set1, set2))
print(np.in1d(set2, set1))
```

```
[ True False False  True  True]
[ True False False  True  True]
```

In [122]:
```
names = np.array(["Charles", "Kilho", "Hayoung", "Charles", "Hayoung
", "Kilho"])
print(np.unique(names))
```

```
['Charles' 'Hayoung' 'Kilho']
```

4. 선형대수

데이터를 다루는 통계학 학문에서는 행렬(matrix)을 가지고 연산을 하는 선형대수는 중요한 개념이다. Numpy의 배열은 행렬의 곱셈, 분할, 행렬식 등을 간편하게 연산할 수 있도록 도와준다.

주요 선형 대수 함수

Operation	Description
diag	diagonal matrix
dot	multiplication of matrix
trace	trace of matrix
linalg.det	determenation of matrix
linalg.solve	solution of equation Ax = b
linalg.inv	inverse matrix

In [56]:
```
a = np.diag([1, 1])
b = np.array([[1, 2], [3, 4]])
```

In [133]:
```
print(a)
print(b)
```

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

In [134]:
```
print(a*b)                    # 원소 간 곱
print(np.multiply(a, b))
print(np.dot(a, b))           # 내적
```

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

In [137]:
```
np.trace(b)
```

Out[137]: 5

In [139]:
```
print(np.linalg.inv(arr1))
print(np.linalg.inv(arr2))
```

```
[[1. 0.]
 [0. 1.]]
[[-2.   1. ]
 [ 1.5 -0.5]]
```

In [141]:
```
print(np.linalg.det(b))
```

```
-2.0000000000000004
```

In [143]:
```
# 4x+3y=23 & 3x+2y=16
a = np.array([[4, 3], [3, 2]])
b = np.array([23, 16])
x = np.linalg.solve(a, b)
```

```
print(x)
```

```
[2. 5.]
```

5. 브로드캐스팅 (Broadcasting)

선형 대수에서 벡터 (또는 행렬)끼리 덧셈 혹은 뺄셈을 하려면 두 벡터 (혹은 행렬)의 크기가 같아야 한다.

그러나, NumPy에서는 서로 다른 크기를 가진 두 배열의 사칙 연산도 지원한다.

이 기능을 **브로드캐스팅 (broadcasting)**이라고 하는데 크기가 작을 배열의 원소가 자동 반복되어 크기가 큰 배열에 맞춰지는 것을 말한다.

In [59]:
```
x = np.arange(5)
x
```

Out[59]:
```
array([0, 1, 2, 3, 4])
```

In [60]:
```
y = np.ones_like(x)
y
```

Out[60]:
```
array([1, 1, 1, 1, 1])
```

In [61]:
```
x + y
```

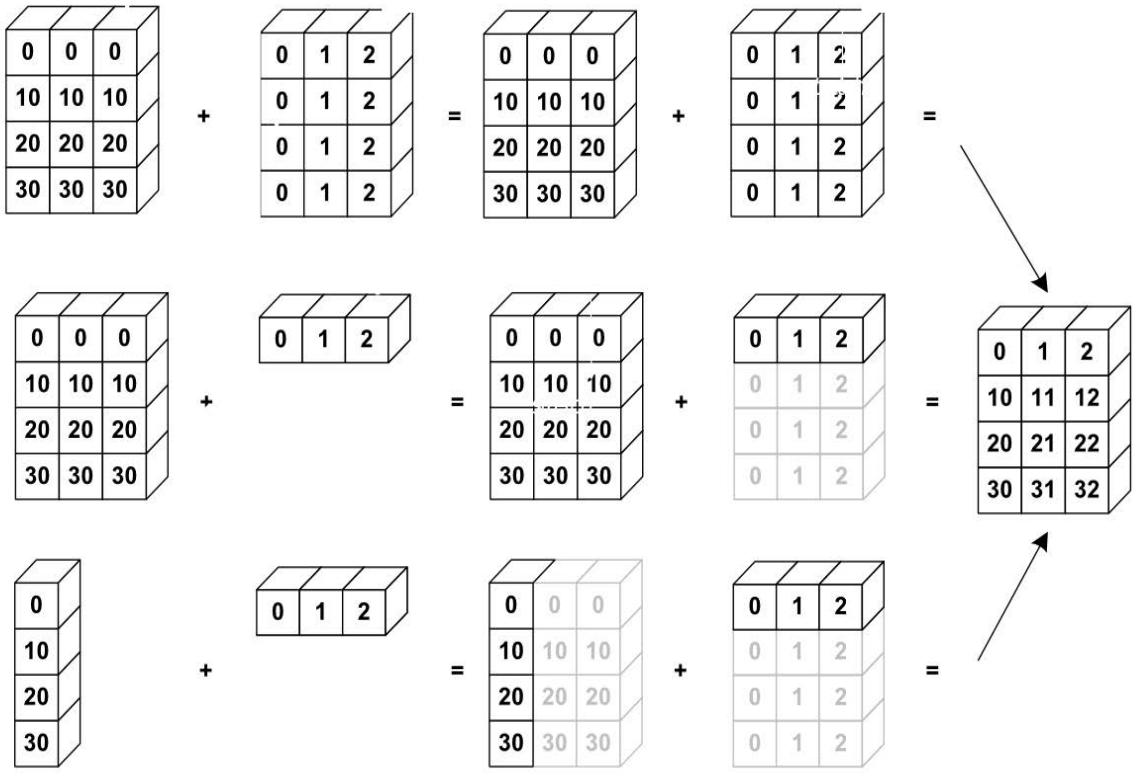
Out[61]:
```
array([1, 2, 3, 4, 5])
```

In [62]:
```
x + 1                    # 크기 모양이 다름에도 연산 가능
```

Out[62]:
```
array([1, 2, 3, 4, 5])
```

브로드캐스팅은 더 높은 차원에서도 작동한다. 다음 그림을 참조하라.

* 그림 출처 : https://i.stack.imgur.com/JcKv1.png



이 그림을 코드로 구현하면 아래와 같다.

In [65]:
```
a = np.tile(np.arange(0, 40, 10), (3, 1)).T  #참고 : tile 함수를 원소를 반복하는 함수이다.
a
```

Out[65]:
```
array([[ 0,  0,  0],
       [10, 10, 10],
       [20, 20, 20],
       [30, 30, 30]])
```

In [66]:
```
b = np.array([0, 1, 2])
b
```

Out[66]:
```
array([0, 1, 2])
```

In [67]:
```
a+b
```

Out[67]:
```
array([[ 0,  1,  2],
       [10, 11, 12],
```

```
       [20, 21, 22],
       [30, 31, 32]])
```

In [68]:
```
a = np.arange(0, 40, 10)
a
```

Out[68]:
```
array([ 0, 10, 20, 30])
```

In [69]:
```
a = a.reshape(4,1)
a
```

Out[69]:
```
array([[ 0],
       [10],
       [20],
       [30]])
```

In [70]:
```
a + b
```

Out[70]:
```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

In [46]:
```
#또 다른 예제
arr1 = np.array([[2, 3, 4], [6, 7, 8]])
arr3 = np.array([100, 200, 300])
print( arr1.shape, arr3.shape)
```

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

In [47]:
```
arr1+arr3
```

Out[47]:
```
array([[102, 203, 304],
       [106, 207, 308]])
```

Ch6. NumPy (3) - Reading and Writing files

NumPy를 활용하여 배열을 입출력하는 방법에 대해서 알아보자.

In [49]:
```
import numpy as np
```

Table of Contents

1. 텍스트 파일로 읽고 쓰기

- NumPy로 만들어진 배열을 np.loadtxt()와 np.savetxt()를 사용하여 텍스트파일로 읽고 쓸수 있다.
- 만들어진 배열을 txt, csv, dat 파일로 저장하고, 불러 올 수 있으며 이를 통해 콤마(,)와 공백(space)으로 분리된 구조화된 데이터를 다룰 수있다.
- 두 함수는 default로 space를 구분자로 사용하기 때문에 만일 콤마(,)를 구분자로 사용하려는 경우 delimiter=',' 를 지정해야 한다.
- 사용시 주의할 점은 1차원 배열 (즉, 벡터)은 n*1 형태로 저장된다는 점과 파일에서 n*1 이나 1*n은 모두 1차원 벡터로 읽힌다는 점이다.

operation	description
savetxt	여러개의 배열을 csv, txt, dat 파일로 저장
loadtxt	csv, txt, dat 파일을 배열로 불러오기

In [46]:
```
x = np.array([0, 1, 2, 3, 4])
y = np.array([5, 6, 7, 8, 9])
```

In [47]:
```
#.csv 파일로 저장
np.savetxt("C:/Users/hansolji/Downloads/xy_save.csv",
           (x, y),
           delimiter=",",
           header="--xy save start--", #머릿말 지정
           footer="--xy save end--", #꼬릿말 지정
           fmt="%1.2f") #소수점 두 자리 표현

#.txt 파일로 저장
np.savetxt("C:/Users/hansolji/Downloads/xy_save.txt",
```

```
            (x, y),
            header="--xy save start--", #머릿말 지정
            footer="--xy save end--", #꼬릿말 지정
            fmt="%1.2f") #소수점 두 자리 표현

#.dat 파일로 저장
np.savetxt("C:/Users/hansolji/Downloads/xy_save.dat",
            (x, y),
            header="--xy save start--", #머릿말 지정
            footer="--xy save end--", #꼬릿말 지정
            fmt="%1.2f") #소수점 두 자리 표현

#txt 형태로 저장하나 콤마(,) 를 구분자로 지정
np.savetxt("C:/Users/hansolji/Downloads/xy_save2.txt",
            (x, y),
            delimiter=",",
            header="--xy save start--", #머릿말 지정
            footer="--xy save end--", #꼬릿말 지정
            fmt="%1.2f") #소수점 두 자리 표현
```

In [52]:
```
#파일 불러오기
xy_save_csv = np.loadtxt("C:/Users/hansolji/Downloads/xy_save.csv",
delimiter=",")
xy_save_txt = np.loadtxt("C:/Users/hansolji/Downloads/xy_save.txt")
xy_save_dat = np.loadtxt("C:/Users/hansolji/Downloads/xy_save.dat")
xy_save_txt_deli = np.loadtxt("C:/Users/hansolji/Downloads/xy_save2.
txt", delimiter=",")
```

In [53]:
```
xy_save_csv
```

Out[53]:
```
array([[0., 1., 2., 3., 4.],
       [5., 6., 7., 8., 9.]])
```

In [54]:
```
xy_save_txt
```

Out[54]:
```
array([[0., 1., 2., 3., 4.],
       [5., 6., 7., 8., 9.]])
```

In [55]:
```
xy_save_dat
```

Out[55]:
```
array([[0., 1., 2., 3., 4.],
       [5., 6., 7., 8., 9.]])
```

In [56]:
```
xy_save_txt_deli
```

Out[56]:
```
array([[0., 1., 2., 3., 4.],
       [5., 6., 7., 8., 9.]])
```

In [57]:
```
#x : 1-D 배열이었지만, n*1 형태로 저장됨
np.savetxt("C:/Users/hansolji/Downloads/x_save.txt", x)
```

In [59]:
```
#n*1 형태로 저장된 파일이었으나, 1-D 배열로 불러들여짐
x_save_txt= np.loadtxt("C:/Users/hansolji/Downloads/x_save.txt")
```

```
x_save_txt
```

```
Out[59]: array([0., 1., 2., 3., 4.])
```

2. Binary 파일 읽고 쓰기

- NumPy에서 지원하는 바이너리 파일을 .npy, .npz라고 한다.
- .npy는 1개의 ndarray를, .npz는 여러개의 ndarray를 저장하는 데 사용된다.
- 불러온 배열 file을 더이상 사용할 일이 없는 경우엔 메모리 효율 관리를 위해 close로 닫는다.

operation	description
save	1개의 배열을 .npy 포맷 바이너리 파일로 저장
savez	여러개의 배열을 1개의 압축되지 않은 .npz 포맷 파일로 저장
savez_compressed	savez와 동일하나 데이터를 압축(compress)하여 .npz 포맷 파일로 저장
load	저장된 .npy, .npz파일을 배열로 불러오기

```
In [60]: x = np.array([0, 1, 2, 3, 4])
         x
```

```
Out[60]: array([0, 1, 2, 3, 4])
```

```
In [61]: # save : np.save(path, x) -> .npy 파일로 저장됨
         np.save("C:/Users/hansolji/Downloads/x_save", x)
```

```
In [62]: # load : np.load(path)
         x_save_load = np.load("C:/Users/hansolji/Downloads/x_save.npy")
         x_save_load
```

```
Out[62]: array([0, 1, 2, 3, 4])
```

```
In [63]: x = np.array([0, 1, 2, 3, 4])
         y = np.array([5, 6, 7, 8, 9])
         print(x, y)
```

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

```
In [66]: # save : np.savez(path, x, y) -> .npz 파일로 저장됨
         np.savez("C:/Users/hansolji/Downloads/xy_savez", x=x, z=y)
```

```
In [67]: xy_savez_load = np.load("C:/Users/hansolji/Downloads/xy_savez.npz")
         xy_savez_load
```

```
Out[67]: <numpy.lib.npyio.NpzFile at 0x1dd81be7b08>
```

```
In [69]: print(type(xy_savez_load))
         print(xy_savez_load['x'])
```

```
print(xy_savez_load['z'])
```

```
<class 'numpy.lib.npyio.NpzFile'>
[0 1 2 3 4]
[5 6 7 8 9]
```

In [71]:
```
#파일닫기
xy_savez_load.close()
```

In [ ]:
```
print(xy_savez_load['x'])  #닫혀진 후에는 더 이상 사용이 불가능하다.
```

In [3]:
```
import numpy as np
```

Examples

Q1. 다음의 배열 **a** 에 대해서 아래의 문항에 답해보자.

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

1) 배열 a를 다른 함수를 사용하여 만들어보자.

2) 배열 a 원소의 유형을 출력하고, 원소의 유형을 float64로 바꾼 배열 b를 출력해보자.

3) 배열 a를 2차원 3*2 배열 c로 만들어 보자.

4) a를 이용해서 사실 아래와 같은 2차원 배열 d 를 만들고 싶었다. 어떻게 만들 수 있을까?

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

Q2. 다음은 배열 **e**와 리스트 **e_list** 이다.

In [8]:
```
e = np.array([[1, 2, 3, 4, 5], [2, 4, 5, 6, 7], [5, 7, 8, 9, 9]])
e_list = [[1, 2, 3, 4, 5], [2, 4, 5, 6, 7], [5, 7, 8, 9, 9]]
```

1) 배열 e 와 리스트 e_list 에서 각각 [[1,2,3,4,5], [2,4,5,6,7]] 에 해당하는 부분만 출력하고 싶을때 어떻게 슬라이싱 할 수 있을까?

2) 배열 e 에서 각각 [[1,5],[2,7]] 부분만 다시 뽑아내고 싶다. 어떻게 할 수 있을까?

3) 배열 e에 대해서 아래의 결과들을 예상해 보자.

In [10]:
```
e
```

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

In [ ]:
```
e[:,[2,3]]
```

In [ ]:
```
e[:,2]
```

In [ ]:
```
e[:][2]
```

In [ ]:
```
e[ [-1,-2], 1 ]
```

In [ ]:
```
e[[-1,-2]][1]
```

Q3. A와 B의 dot product 의 대각원소(digonal) 을 얻는 코드를 작성해 보자.

In [41]:
```
A= np.array([[1,2],[2,3]])
B= np.random.randn(2, 2)
```

Q4. 다음의 Z array 를 n번째 컬럼에 의해서 정렬(sort) 해보자.

- 만일 2번째 컬럼에 의해서 정렬한다면, array가 2번째 컬럼의 크기순으로 확장되어 정렬된다. 2
- 번째 컬럼 중에서 가장 작은 값이 있는 행이 array의 첫번째 행이되고, 2번째 컬럼 중에서 가장큰 값이 있는 행이 array의 마지막 행이됨

In [5]:
```
Z = np.random.randint(0,10,(3,3))
print(Z)
```

```
[[7 2 3]
 [7 3 9]
 [3 9 4]]
```

Q5. 다음의 a,b,c 를 이용한 결과 중 다른 하나는 무엇인가?

In [72]:
```
a = np.array([1, 2, 3])
b = np.array([0, 4, 8])
c = np.linspace(0, 20, 6).astype("int")
```

In [ ]:
```
np.vstack((a,b))
```

In [ ]:
```
np.vstack((a, np.split(c, 2, axis=0)[0]))
```

In [ ]:
```
np.concatenate((a,b), axis=0)
```

In [ ]:
```
np.hstack((a,b)).reshape(2,3)
```

Q6. 3개의 배열 A1, A2, A3 를 C:/Users/Downloads 경로에 Aarray 라는 이름의 압축된 *.npz 형식 파일로 저장하는 코드를 작성하라.

- A1, A2, A3 각각의 배열은 파일에 동일한 이름으로 저장할 것. 즉, A1은 파일에 A1으로, A2는 파일에 A2로, A3은 파일에 A3으로 저장되게 코드 작성하여라.

In [ ]:
```
A1=np.random.randint(0,10,(3,3))
A2=np.random.randint(0,2,(3,5))
A3=np.random.randint(0,2,(3,5))
```