기초빅데이터프로그래밍

Numpy

서강대학교
SOGANG UNIVERSITY

Why NumPy?

- http://www.numpy.org
- **반복문 없이** 빠른 계산 가능한 다차원 배열 제공(내부적으로 C로 구현)
- 빠른 계산 속도 제공
- 원하는 형태의 data만 구별할 수 있는 indexing 기능 (정수 인덱싱, Boolean indexing)
- vectorized operation 지원
  - 배열의 각 원소에 대한 반복연산을 하나의 명령어로 처리함
- numpy 배열 생성 방법
  - python list 사용: np.array( )
  - numpy 제공함수 이용: zeros(), ones(), full(), eye()

NumPy

- NumPy: Numerical Python의 약어
- ndarray : numpy를 통해 생성되는 n차원의 배열 객체, 기존 파이썬과 다르게 **같은 종류의 데이터 만을 배열에 담을 수 있다.**
- ndarray의 생성 : NumPy의 **array**함수로 만든다

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

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

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

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

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

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

```
arr.shape
```

```
(5,)
```

```
print(arr) #배열로 표현
```

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

```
arr.ndim
```

```
1
```

```
arr.dtype
```

```
dtype('int32')
```

```
arr.size
```

```
5
```

```
arr.itemsize
```

```
4
```

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

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

```
arr.shape
```

```
(2, 4)
```

```
print(arr)
```

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

```
arr.ndim
```

```
2
```

```
arr.dtype
```

```
dtype('int32')
```

```
arr.size
```

```
8
```

numpy 제공함수 이용한 배열생성

- **zeros, ones** 함수로 0행렬, 모든 행렬 값이 1인 행렬을 만들 수 있다
- **empty**함수는 초기화되지 않은 행렬을 생성한다.

```
np.empty((3,3))
```

```
array([[  0.00000000e+000,   4.34777768e-321,   2.47032823e-323],
       [  8.91062644e-312,   0.00000000e+000,   3.23815565e-319],
       [  8.91067975e-312,   0.00000000e+000,   2.47032823e-323]])
```

```
np.ones((3,3))
```

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

```
np.full((3,4), 0.2)
```

```
array([[ 0.2,  0.2,  0.2,  0.2],
       [ 0.2,  0.2,  0.2,  0.2],
       [ 0.2,  0.2,  0.2,  0.2]])
```

```
np.zeros((3,3))
```

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

```
np.eye(3) # nxn identity
```

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

array()

- **array(object, dtype=None, copy=True, order=None, subok=False, ndmin=0)**

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. This argument can only be used to 'upcast' the array. For downcasting, use the .astype(t) method.
- 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 : {'C', 'F', 'A'}, optional
  - Specify the order of the array. If order is 'C', then the array will be in C-contiguous order (last-index varies the fastest). If order is 'F', then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is 'A' (default), then the returned array may be in any order (either C-, Fortran-contiguous, or even discontiguous), unless a copy is required, in which case it will be C-contiguous.
- 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.

```
import numpy as np
data=[[1,2,3,4,5], [5,6,7,8]]
arr=np.array(data)
arr
```

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

이제 지원되지 않으므로 아래와 같이 변경

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

```
array([list([1, 2, 3, 4, 5]), list([5, 6, 7, 8])], dtype=object)
```

- **데이터 생성 함수**
  - numpy.linspace
  - numpy.logspace
  - numpy.arange

numpy.linspace

- **numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)]**
  - Returns num evenly spaced samples, calculated over the interval [start, stop].
  - The endpoint of the interval can optionally be excluded.

Parameters:

- start : scalar -> array_like
  - The starting value of the sequence.
- stop : scalar -> array_like
  - The end value of the sequence, unless endpoint is set to False. In that case, the sequence consists of all but the last of num + 1 evenly spaced samples, so that stop is excluded. Note that the step size changes when endpoint is False.

- num : int, optional
  - Number of samples to generate. Default is 50. Must be non-negative.
- endpoint : bool, optional
  - If True, stop is the last sample. Otherwise, it is not included. Default is True.
- retstep : bool, optional
  - If True, return (samples, step), where step is the spacing between samples.
- dtype : dtype, optional
  - The type of the output array. If dtype is not given, infer the data type from the other input arguments.
  - New in version 1.9.0.

❖ Returns:

- samples : ndarray
  - There are num equally spaced samples in the closed interval [start, stop] or the half-open interval [start, stop) (depending on whether endpoint is True or False).
- step : float, optional
  - Only returned if retstep is True
  - Size of spacing between samples.

```
np.linspace(1,5)
```

```
array([ 1.        ,  1.08163265,  1.16326531,  1.24489796,  1.32653061,
        1.40816327,  1.48979592,  1.57142857,  1.65306122,  1.73469388,
        1.81632653,  1.89795918,  1.97959184,  2.06122449,  2.14285714,
        2.2244898 ,  2.30612245,  2.3877551 ,  2.46938776,  2.55102041,
        2.63265306,  2.71428571,  2.79591837,  2.87755102,  2.95918367,
        3.04081633,  3.12244898,  3.20408163,  3.28571429,  3.36734694,
        3.44897959,  3.53061224,  3.6122449 ,  3.69387755,  3.7755102 ,
        3.85714286,  3.93877551,  4.02040816,  4.10204082,  4.18367347,
        4.26530612,  4.34693878,  4.42857143,  4.51020408,  4.59183673,
        4.67346939,  4.75510204,  4.83673469,  4.91836735,  5.        ])
```

```
np.linspace(2.0, 3.0, num=5)
```

```
array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ])
```

```
np.linspace(2.0, 3.0, num=5, endpoint=False)
```

```
array([ 2. ,  2.2,  2.4,  2.6,  2.8])
```

```
np.linspace(2.0, 3.0, num=5, retstep=True)
```

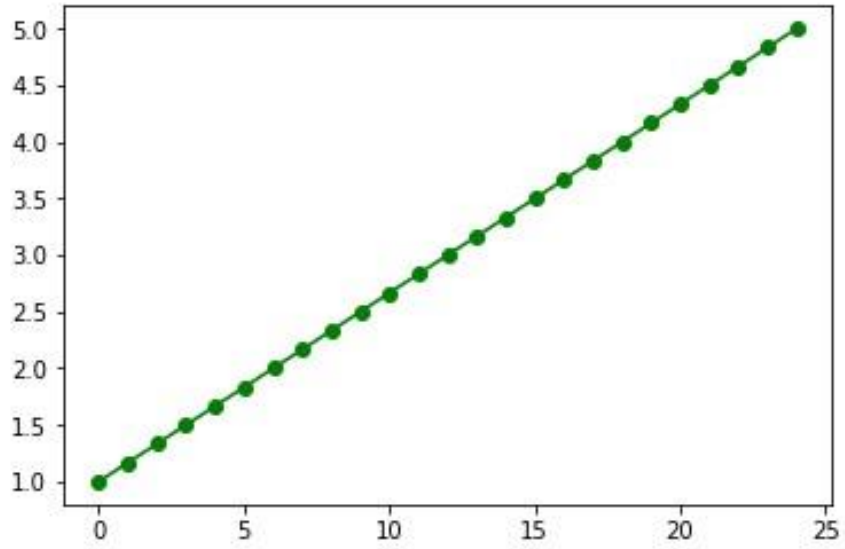
```
(array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ]), 0.25)
```

```
a=np.linspace(1,5,25)
```

```
print(a)
```

```
[ 1.          1.16666667  1.33333333  1.5         1.66666667  1.83333333
  2.          2.16666667  2.33333333  2.5         2.66666667  2.83333333
  3.          3.16666667  3.33333333  3.5         3.66666667  3.83333333
  4.          4.16666667  4.33333333  4.5         4.66666667  4.83333333
  5.        ]
```

```
import matplotlib.pylab as plt
plt.plot(a, 'go-')
plt.show()
```



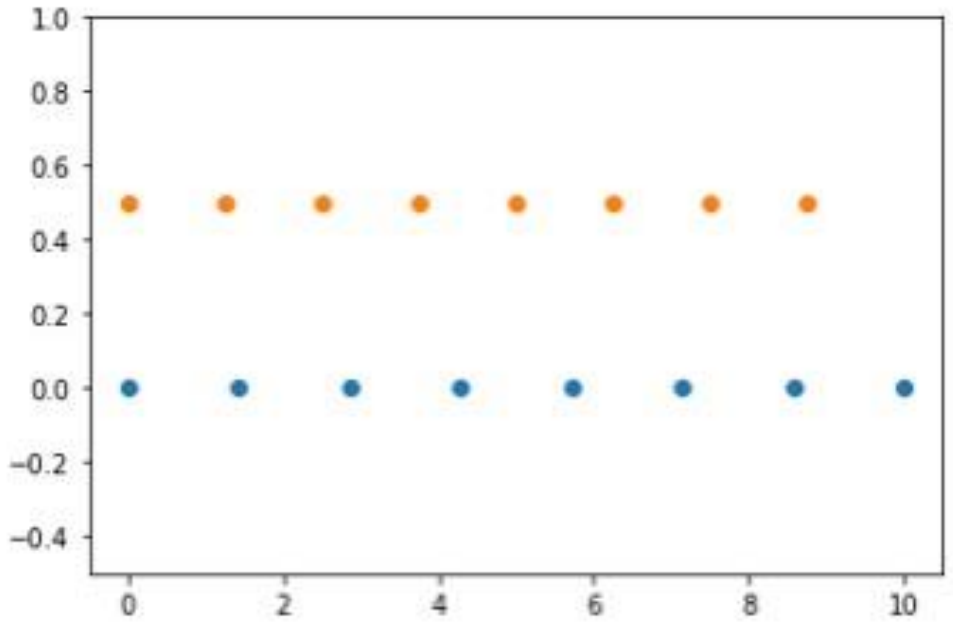
```
import matplotlib.pyplot as plt
N=8
y = np.zeros(N)
x1 = np.linspace(0, 10, N, endpoint=True)
x2 = np.linspace(0, 10, N, endpoint=False)
plt.plot(x1, y, 'o')
```

```
[<matplotlib.lines.Line2D at 0x2761c1f2198>]
```

```
plt.plot(x2, y+0.5, 'o')
```

```
[<matplotlib.lines.Line2D at 0x2761b9ebd30>]
```

```
plt.ylim([-0.5, 1])
plt.show()
```



logspace

- Numpy logspace return numbers spaced evenly **on a log scale**.
- logspace computes its start and end points as **base**start** and **base**stop** respectively. The base value can be specified, but is 10.0 by default.

a start value 10**0.02 =1.047
a stop value 10**2 =100

```
print(np.linspace(0.02, 2.0, num=20))
print(np.logspace(0.02, 2.0, num=20))
```

```
[ 0.02        0.12421053  0.22842105  0.33263158  0.43684211  0.54105263
  0.64526316  0.74947368  0.85368421  0.95789474  1.06210526  1.16631579
  1.27052632  1.37473684  1.47894737  1.58315789  1.68736842  1.79157895
  1.89578947  2.        ]
[   1.04712855    1.33109952    1.69208062    2.15095626    2.73427446
    3.47578281    4.41838095    5.61660244    7.13976982    9.07600522
   11.53732863   14.66613875   18.64345144   23.69937223   30.12640904
   38.29639507   48.68200101   61.88408121   78.6664358   100.        ]
```

```
print(np.logspace(np.log10(0.02), np.log10(2.0), num=20))
```

```
[ 0.02        0.0254855   0.03247553  0.04138276  0.05273302  0.06719637
  0.08562665  0.1091119   0.13903856  0.17717336  0.22576758  0.28768998
  0.36659614  0.46714429  0.59527029  0.75853804  0.96658605  1.23169642
  1.56951994  2.        ]
```

np.arange

- np.arange([start,] **stop**[, step,], dtype=None)

```
np.arange(10)
```

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

```
np.arange(1,10)
```

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

```
np.arange(1,10, 0.5)
```

```
array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,  5.5,  6. ,
        6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5])
```

```
np.arange(1,10, 3)
```

```
array([1, 4, 7])
```

```
np.arange(1,10, 2, dtype=np.float64)
```

```
array([ 1.,  3.,  5.,  7.,  9.])
```

np.ndarray

Parameters:

- 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.

ndarray의 차수, 모양, 타입

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

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

```
arr.ndim
```

```
2
```

```
arr.shape
```

```
(2, 4)
```

```
arr.dtype
```

```
dtype('int32')
```

```
arr.size
```

```
8
```

```
arr.itemsize #byte단위
```

```
4
```

```
arr.tolist()
```

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

```
arr.T
```

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

T	Same as self.transpose(), except that self is returned if self.ndim < 2.
dtype	Data-type of the array's elements.
size	Number of elements in the array.
itemsize	Length of one array element in bytes.
nbytes	Total bytes consumed by the elements of the array.
ndim	Number of array dimensions.
shape	Tuple of array dimensions.

```
a = np.ones(3, dtype=np.int32)
a
```

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

```
b = np.linspace(0, np.pi, 3)
b
```

```
array([ 0.        ,  1.57079633,  3.14159265])
```

```
b.dtype.name
```

```
'float64'
```

```
c = a+b
c
```

```
array([ 1.        ,  2.57079633,  4.14159265])
```

```
c.dtype.name
```

```
'float64'
```

```
d = np.exp(c*1j)
d
```

```
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
```

```
d.dtype.name
```

```
'complex128'
```

서강대학교 SOGANG UNIVERSITY

astype() : ndarray의 타입변환

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

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

```
arr.dtype
```

```
dtype('int32')
```

```
float_arr = arr.astype(np.float64)
float_arr.dtype
```

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

```
float_arr
```

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

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

```
array(['1', '2', '3', '4'],
      dtype='<U1')
```

```
int_arr = arr.astype(np.int64)
int_arr.dtype
```

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

```
int_arr
```

```
array([1, 2, 3, 4], dtype=int64)
```

Int형 자료를 float형으로 변환하여 새로운 배열을 생성함

문자열형 자료를 int형으로 변환하여 새로운 배열을 생성함

dtype : 자료형 이름과 원소가 차지하는 비트수로 이루어진다.

서강대학교 SOGANG UNIVERSITY

종류	Type Code	설명
int8, uint8	i1, u1	부호가 있는 8비트(1바이트) 정수형과 부호가 없는 8비트 정수형
int16, uint16	i2, u2	부호가 있는 16비트 정수형과 부호가 없는 16비트 정수형
int32, uint32	i4, u4	~
int64, uint64	i8, u8	~
float16	f2	반정밀도 부동소수점
float32	f4 또는 f	단정밀도 부동소수점, C언어의 float과 호환
float64	f8 또는 d	배정밀도 부동소수점, C언어의 double형과 파이썬 float객체와 호환
float128	f16 또는 g	확장 정밀도 부동소수점
complex64, complex 128, complex 256,	c8, c16, c32	각각 2개의 32, 64, 128비트 부동소수점형을 가지는 복소수
bool	?	True, False를 저장하는 불리언형
object	0	파이썬 객체형
string_	S	고정길이 문자열형(각 글자는 1바이트), 길이가 10인 문자열의 dtype = S10
unicode_	U	고정 길이 유니코드형(플랫폼에 따라 글자별 바이트수 다름) string_형과 같은형식 씀(ex) U10)

```
ar=np.array(['1','2','3','4'])
```

```
ar #int8
```

```
array(['1', '2', '3', '4'],
      dtype='<U1')
```

```
ar11=np.array([1,2,3,4],dtype=np.string_)
ar11
```

```
array([b'1', b'2', b'3', b'4'],
      dtype='|S1')
```

```
ar1=np.array(['a','b','c','d'])
ar1
```

```
array(['a', 'b', 'c', 'd'],
      dtype='<U1')
```

```
ar2=np.array(['11','22','32','44'])
ar2 #int16
```

```
array(['11', '22', '32', '44'],
      dtype='<U2')
```

```
ar3=np.array(["a","b","c","d"], dtype=np.string_)
ar3
```

```
array([b'a', b'b', b'c', b'd'],
      dtype='|S1')
```

배열 원소 개수 알아보기

일차원과 다차원의 원소 개수를 len()함수로 처리시 다른 결과가 나옴

```
import numpy as np
npa = np.array([[1,2,3], [4,5,6],[7,8,9]])
print(npa)
print(npa.size, npa.shape, npa.ndim, len(npa))
```

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

```
np1a = np.array([1, 999, 11, 23, 45, 45, 31])
print(np1a)
print(np1a.size, np1a.shape, np1a.ndim, len(np1a))
```

```
[  1 999  11  23  45  45  31]
7 (7,) 1 7
```

같은 크기의 배열간 산술연산

```
arr=np.array([[1,2,3],[10,20,30]])
print(arr)
arr+arr
```

```
[[ 1  2  3]
 [10 20 30]]

array([[ 2,  4,  6],
       [20, 40, 60]])
```

```
arr-arr
```

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

```
arr*0.5
```

```
array([[  0.5,   1. ,   1.5],
       [  5. ,  10. ,  15. ]])
```

```
1/arr
```

```
array([[ 1.        ,  0.5       ,  0.33333333],
       [ 0.1       ,  0.05      ,  0.03333333]])
```

numpy.random

- NumPy 의 랜덤 넘버 생성 관련 함수를 모아 놓은 것으로 다음과 같은 함수를 제공한다.

seed: pseudo random 상태 설정

shuffle: 조합(combination)

choice: 순열(permutation)

random_integers: uniform integer

rand: uniform

randn: Gaussian normal

- 컴퓨터에서 생성한 난수는 랜덤처럼 보이지만 정해진 알고리즘에 의해 생성되는 규칙적인 순열이다. seed 명령은 이러한 순열을 시작하는 초기값을 설정하여 난수가 정해진 순서로 나오게 만든다.

- np.random.seed(0)

- shuffle 명령은 주어진 배열의 순서를 뒤섞는다.

```
x = np.arange(10)
np.random.shuffle(x)
x
```

```
x=np.arange(10)
x
```

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

```
np.random.shuffle(x)
x
```

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

```
np.random.choice(x)
```

```
2
```

```
np.random.shuffle(x)
np.random.choice(x)
```

```
6
```

numpy.random.rand

numpy.random.rand(d0, d1, ..., dn)

- Random values in a given shape.

- Create an array of the given shape and propagate it with random samples from a **uniform distribution over [0, 1).**

Parameters:

- d0, d1, ..., dn : int, optional
- The dimensions of the returned array, should all be positive.
- If **no argument** is given **a single Python float** is returned.

Returns:

- out : ndarray, shape (d0, d1, ..., dn)
- Random values.

```
y=np.random.rand() # [0,1) uniform distribution
y
```

```
0.6133456987640892
```

```
y=np.random.rand(7)
y
```

```
array([ 0.0800172 ,  0.05933676,  0.41923288,  0.06668333,  0.06551149,
        0.63284029,  0.63446047])
```

```
y=np.random.rand(2,7)
y
```

```
array([[ 0.24307701,  0.43596357,  0.66831741,  0.31943357,  0.96614403,
         0.53821965,  0.75052437],
       [ 0.89556096,  0.42433654,  0.80967985,  0.26127949,  0.84305902,
         0.46167859,  0.37224801]])
```

```
y=np.random.rand(3,2,7)
y
```

```
array([[[ 0.42295466,  0.94729914,  0.54958094,  0.46207841,  0.72266712,
          0.7741601 ,  0.98360692],
        [ 0.86697688,  0.82325421,  0.62390495,  0.46612152,  0.49937559,
          0.54568176,  0.90496695]],

       [[ 0.35534179,  0.73052051,  0.87587516,  0.56478317,  0.15575129,
          0.70145355,  0.60097832],
        [ 0.46620323,  0.96544893,  0.69849624,  0.45982176,  0.22612589,
          0.23875577,  0.84345221]],

       [[ 0.93343081,  0.60005562,  0.16393261,  0.1204029 ,  0.76533883,
          0.94743599,  0.65055541],
        [ 0.9073561 ,  0.44664639,  0.46584579,  0.26518079,  0.40476183,
          0.86617323,  0.26406164]]])
```

numpy.random.randn

numpy.random.randn(d0, d1, ..., dn)

- Return a sample (or samples) from the "**standard normal**" distribution.
- If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "**normal" (Gaussian) distribution of mean 0 and variance 1** (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.
- This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard_normal instead.

Parameters:

- d0, d1, ..., dn : int, optional
- The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.

Returns:

- Z : ndarray or float
- A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

```
y=np.random.randn() # mean=0, variance=1 Gaussian Normal distribution
y
```

```
1.0657891986358792
```

```
y=np.random.randn(7)
y
```

```
array([-0.69993739,  0.14407911,  0.3985421 ,  0.02686925,  1.05583713,
       -0.07318342, -0.66572066])
```

```
y=np.random.randn(2,7)
y
```

```
array([[-0.04411241, -0.36326702, -0.01234481,  0.04212149,  1.95929589,
        -0.1984257 ,  0.33053441],
       [-1.43582841,  0.02752832,  1.12060466, -0.22403878, -0.42018339,
         0.99982969,  0.43103415]])
```

```
y=np.random.randn(3,2,7)
y
```

```
array([[[-0.65091287, -1.49874039, -1.23063497,  0.19400719, -0.99838235,
         -0.3676376 ,  1.73719932],
        [ 0.59361275, -0.54236358, -1.71967238, -0.57890879,  1.42694855,
          0.27699691,  0.78966713]],

       [[ 0.32207411,  0.70039238,  0.38871663, -0.04126386,  0.29588432,
         -0.42527999,  1.72763912],
        [-0.86835257, -0.82097758, -1.09974201,  0.08672983,  0.45628701,
          0.4310333 ,  2.07257353]],

       [[-0.53778512, -1.3784301 , -0.49240425,  2.32773811,  1.80440397,
         -0.24942133, -0.82086383],
        [-1.49333996,  0.52417595,  0.34511317,  0.72437468, -2.04038084,
         -1.0797781 , -0.69342441]]])
```

numpy.cumsum

numpy.cumsum(a, axis=None, dtype=None, out=None)

- Return the **cumulative sum** of the elements along a given axis.

Parameters:

- a : array_like
  - Input array.
- axis : int, optional
  - Axis along which the cumulative sum is computed. The default (None) is to compute the cumsum over the flattened array.
- dtype : dtype, optional
  - Type of the returned array and of the accumulator in which the elements are summed. If dtype is not specified, it defaults to the dtype of a, unless a has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.
- out : ndarray, optional
  - Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary. See doc.ufuncs (Section "Output arguments") for more details.

- Returns: cumsum_along_axis : ndarray.
  - A new array holding the result is returned unless out is specified, in which case a reference to out is returned. The result has the same size as a, and the same shape as a if axis is not None or a is a 1-d array.

cumprod([axis, dtype, out])	Return the cumulative product of the elements a long the given axis.

cumsum()

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

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

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

```
array([ 1,  3,  6, 10, 15, 21], dtype=int32)
```

```
np.cumsum(a, dtype=float)
```

```
array([  1.,   3.,   6.,  10.,  15.,  21.])
```

```
np.cumsum(a, axis=0) #sum over rows for each of the 3 columns
```

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

```
np.cumsum(a, axis=1) #sum over columns for each of the 2 rows
```

```
array([[ 1,  3,  6],
       [ 4,  9, 15]], dtype=int32)
```

```
a = np.array([[1,2,3],
              [4,5,6]])
a0 = a[:, 0]
a0
```

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

```
a1 = a[:, 2]
a1
```

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

```
ar0 =a[0, :]
ar0
```

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

```
ar1 = a[1, :]
ar1
```

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

2차원 배열에서
하나의 행을 선택하거나(a[1, :])
열을 선택하게(a[:, 1])하면
1차원 배열을 반환한다.

MATLAB은 2차원 배열반환

reshape()

```
import numpy as np
import pandas as pd
```

```
arr = np.arange(8)
arr
```

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

```
arr.reshape((4,2))
```

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

```
arr.reshape((2,4))
```

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

```
arr = np.arange(15)
arr.reshape((5,-1))
```

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

함수의 파라미터에 -1이 들어가면, 다른 나머지 차원 크기를 맞추고 남은 크기를 해당 차원에 할당해 준다

ravel() : return a contiguous flattened array

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

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

```
arr.ravel()
```

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

```
arr.ravel('F')
```

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

Row-major order: C
Column-major order : Fortran

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

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

```
np.ravel(arr, order='F')
```

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

```
np.ravel(arr)
```

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

```
np.ravel(arr, order='C')
```

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

concatenate() 배열 붙이기

```
a1 = np.array([[1,2,3], [4,5,6]])
a2 = np.array([[7,8,9], [10,11,12]])
np.concatenate([a1, a2], axis=0)
```

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

```
np.concatenate([a1, a2], axis=1)
```

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

repeat()

```
arr = np.arange(3)
arr
```

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

```
arr.repeat(4)
```

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

```
arr.repeat([2,3,4])
```

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

```
arr = np.random.randn(2,2)
arr
```

```
array([[ 1.05018478,  1.20771269],
       [ 1.23127659, -2.1905615 ]])
```

```
arr.repeat([2,3], axis=0)
```

```
array([[ 1.05018478,  1.20771269],
       [ 1.05018478,  1.20771269],
       [ 1.23127659, -2.1905615 ],
       [ 1.23127659, -2.1905615 ],
       [ 1.23127659, -2.1905615 ]])
```

```
arr.repeat([2,3], axis=1)
```

```
array([[ 1.05018478,  1.05018478,  1.20771269,  1.20771269,  1.20771269],
       [ 1.23127659,  1.23127659, -2.1905615 , -2.1905615 , -2.1905615 ]])
```

tile()

```
arr
```

```
array([[ 1.05018478,  1.20771269],
       [ 1.23127659, -2.1905615 ]])
```

```
np.tile(arr, 2)
```

```
array([[ 1.05018478,  1.20771269,  1.05018478,  1.20771269],
       [ 1.23127659, -2.1905615 ,  1.23127659, -2.1905615 ]])
```

```
np.tile(arr, (2,1))
```

```
array([[ 1.05018478,  1.20771269],
       [ 1.23127659, -2.1905615 ],
       [ 1.05018478,  1.20771269],
       [ 1.23127659, -2.1905615 ]])
```

```
np.tile(arr, (3,2))
```

```
array([[ 1.05018478,  1.20771269,  1.05018478,  1.20771269],
       [ 1.23127659, -2.1905615 ,  1.23127659, -2.1905615 ],
       [ 1.05018478,  1.20771269,  1.05018478,  1.20771269],
       [ 1.23127659, -2.1905615 ,  1.23127659, -2.1905615 ],
       [ 1.05018478,  1.20771269,  1.05018478,  1.20771269],
       [ 1.23127659, -2.1905615 ,  1.23127659, -2.1905615 ]])
```

ndarray 할당은 참조만 전달

ndarray은 참조만 할당하므로 새로운 ndarray로 처리하려면 copy( ) 함수를 사용해야 함

```
x=np.array([1,2,3,4])
o = x #할당
print(id(o), id(x)) #같은 id
```

```
2413251355632 2413251355632
```

```
o_sub = x[1:3]
print(o_sub)
print(id(o_sub))
```

```
[2 3]
2413251356832
```

```
o_sub[0] =99 # slicing과 원본이 같이 바뀜
print(o_sub, x)
```

```
[99  3] [ 1 99  3  4]
```

Slice을 해도 원 ndarray의 참조를 가지고 있어 갱신하면 원 값을 변경함

```
o_copy =x.copy()
print(id(o_copy), id(x))
o_copy[1] =77
print(o_copy, x)
```

```
2413251357312 2413251355632
[ 1 77  3  4] [ 1 99  3  4]
```

copy()를 이용해서 복사하면 새로운 ndarray 만들어지므로 독립적임

range와 numpy.arange 비교

```
pyL = range(1,10)
print(type(pyL), pyL)
```

```
<class 'range'> range(1, 10)
```

```
npA =np.arange(1,10, dtype=np.float)
print(type(npA), npA)
```

```
<class 'numpy.ndarray'> [ 1.  2.  3.  4.  5.  6.  7.  8.  9.]
```

```
npA =np.arange(1,10, dtype=np.int)
print(type(npA), npA)
```

```
<class 'numpy.ndarray'> [1 2 3 4 5 6 7 8 9]
```

numpy.arange는 다양한 타입으로 array를 생성할 수 있음

연산방식: 섭씨와 화씨 온도 변환

F(화씨) = c(섭씨) * 9 / 5 + 32 이 공식을 기준으로 연속적인 배열을 loop 문 없이 계산

```
cvalues = [25.3, 24.8, 26.9, 23.9]
c = np.array(cvalues)
print(c)
F = c * 9/5 +32
print(type(F), F)
```

```
[ 25.3  24.8  26.9  23.9]
<class 'numpy.ndarray'> [ 77.54  76.64  80.42  75.02]
```

기존 방식, list comprehension으로 loop문 실행

```
pyF = [x *9/5 +32 for x in cvalues]
print(type(pyF), pyF)
```

```
<class 'list'> [77.54, 76.64, 80.42, 75.02]
```

ndarray 특징은 array 원소 만큼 자동으로 순환 계산해서 ndarray로 반환함

실습

- Python의 list와 numpy의 ndarray의 성능을 비교하기 위해
- 1000000개 이상의 원소를 갖는 두 배열 X와 Y의 각 인덱스의 합을 구하는 경우에 실행시간을 비교하시오.

- 10,000,000개 원소의 경우 실행 예

```
t1 = pure_python_version()
t2 = numpy_version()
print('python list exe. time: ', t1)
print('numpy ndarray exe. time: ', t2)
print('numpy is in this example '+ str(t1/t2)+ ' faster')
```

```
python list exe. time:  5.441838264465332
numpy ndarray exe. time:  0.062976837158203125
numpy is in this example 86.41015506693319 faster
```

__getitem__ , __setitem__비교

numpy.ndarray 타입에서는 __getitem__에 논리연산 등 다양한 처리를 허용

```
pyl=range(1,10)
print(type(pyl), pyl)
npa=np.arange(1,10,dtype=np.float_)
print(type(npa), npa)
idx=np.array([True, False,True, False,True, False,True, False,True])
print(idx.dtype, idx)
```

```
<class 'range'> range(1, 10)
<class 'numpy.ndarray'> [ 1.  2.  3.  4.  5.  6.  7.  8.  9.]
bool [ True False  True False  True False  True False  True]
```

```
print(pyl.__getitem__(2))
print(npa.__getitem__(2))
print(npa.__getitem__(idx))
```

```
3
3.0
[ 1.  3.  5.  7.  9.]
```

```
npa.__setitem__(1,200)
print(npa)
npa.__setitem__(idx,222)
print(npa)
```

```
[   1.  200.    3.    4.    5.    6.    7.    8.    9.]
[ 222.  200.  222.    4.  222.    6.  222.    8.  222.]
```