작성자: 고우주 / 데이터쿱와(주)

파이썬 - numpy 실습

1. numpy 개요

1.1 numpy 개요

- numpy는 수치해석용 파이썬 패키지로 numerical python의 줄임말
- 다차원의 배열 자료구조 클래스인 ndarray 클래스를 지원
- 벡터와 행렬을 사용하는 선형대수 계산 사용

1.2 numpy 특징

- numpy의 배열 연산은 c로 구현된 내부 반복문을 사용
- 파이썬 반복문에 비해 빠른 속도
- 벡터화 연산(vectorized operation)을 이용
- 간단한 코드로도 복잡한 선형 대수 연산을 수행
- 배열 인덱싱(array indexing)을 사용한 질의(query) 기능

1.3 데이터 분석에서 빠른 연산을 위해 자주 사용하는 기능

- 배열에서 데이터 변경, 정제, 부분 집합, 필터링의 빠른 수행
- 정렬, 유일 원소 찾기, 집합 연산
- 통계 표현과 데이터의 수집/요약
- 여러 데이터의 병합, 데이터 정렬과 데이터 조작

1.4 numpy 패키지 설치

- numpy는 기본 패키지로 설치되어 있으나, 만약 별로 설치 한다면,
- 터미널과 cmd에서 설치 시 pip install numpy 실행 또는 Jupyter notebook에서 직접 실행

In [1]:

```
!pip install numpy
```

```
Requirement already satisfied: numpy in /anaconda3/lib/python3.7/site-packages (1.15.4)
You are using pip version 19.0.3, however version 19.1.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
```

1.5 numpy 패키지 불러오기

- 배열을 사용하기 위해 numpy 패키지를 import
- numpy는 "np"라는 축약어 사용이 관례

In [1]:

```
import numpy as np
```

2. 배열 (array)

numpy의 array라는 함수에 리스트[ ]를 넣으면 배열로 변환

2.1 1차원 배열

In [2]:

```
# 1차원 배열 생성
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
arr
```

Out[2]:

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

In [3]:

```
# 타입 확인
type(arr)
```

Out[3]:

```
numpy.ndarray
```

In [4]:

```
# array의 형태(크기)를 확인
arr.shape
```

Out[4]:

```
(10,)
```

In [5]:

```
# array의 자료형을 확인
arr.dtype
```

Out[5]:

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

2.2 벡터화 연산

In [6]:

```
# data 리스트 데이터 생성
data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [7]:

```
x = np.array(data)
x
```

Out[7]:

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

In [8]:

```
x + 2
```

Out[8]:

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

In [9]:

```
x * 2
```

Out[9]:

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

In [10]:

```
x // 2
```

Out[10]:

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

In [11]:

```
# array 배열 끼리 연산
a = np.array([1, 2, 3])
b = np.array([10, 20, 30])
```

In [12]:

```
2 * a + b
```

Out[12]:

```
array([12, 24, 36])
```

2.3 조건연산(True, False)

In [13]:

```
a == 2
```

Out[13]:

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

In [14]:

```
b > 10
```

Out[14]:

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

In [15]:

```
(a == 2) & (b > 10)
```

Out[15]:

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

2.4 2차원 배열

- 다차원 배열 자료 구조 ex) 1차원, 2차원, 3차원 배열
- 2차원 배열은 행렬(matrix)로 가로줄 행(row)과 세로줄 열(column)로 구성

In [16]:

```
# 2 x 3 배열 생성
c = np.array([[0, 1, 2], [3, 4, 5]])
c
```

Out[16]:

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

In [17]:

```
# 행의 갯수 확인
len(c)
```

Out[17]:

```
2
```

In [18]:

```
# 열의 갯수 확인
len(c[0])
```

Out[18]:

```
3
```

In [19]:

```
print(len(c))
print(len(c[0]))
```

```
2
3
```

2.5 3차원 배열

바깥쪽 리스트의 길이부터 가장 안쪽 리스트 길이의 순서로 표시

In [20]:

```
 # 2 x 3 x 4 배열 생성
d = np.array([[[1, 2, 3, 4],
               [5, 6, 7, 8],
               [9, 10, 11, 12]],
              [[11, 12, 13, 14],
               [15, 16, 17, 18],
               [19, 20, 21, 22]]])
d
```

Out[20]:

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

       [[11, 12, 13, 14],
        [15, 16, 17, 18],
        [19, 20, 21, 22]]])
```

In [21]:

```
# 인덱싱으로 배열크기 확인
len(d), len(d[0]), len(d[0][0])
```

Out[21]:

```
(2, 3, 4)
```

In [22]:

```
print((len(d), len(d[0]), len(d[0][0])))
```

```
(2, 3, 4)
```

2.6 배열의 차원과 크기

ndim 속성은 배열의 차원, shape 속성은 배열의 크기를 반환

In [23]:

```
# 배열 생성
ab = np.array([1, 2, 3])
ab
```

Out[23]:

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

In [24]:

```
# 차원
ab.ndim
```

Out[24]:

```
1
```

In [25]:

```
# 크기
ab.shape
```

Out[25]:

```
(3,)
```

In [26]:

```
print(ab.ndim)
print(ab.shape)
```

```
1
(3,)
```

In [27]:

```
abc = np.array([[0, 1, 2], [3, 4, 5]])
abc
```

Out[27]:

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

In [110]:

```
print(abc.ndim)
print(abc.shape)
```

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

In [111]:

```
# 2x3x4 3차원 배열 생성
abcd = np.array([[[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]],
                 [[11, 12, 13, 14],
                  [15, 16, 17, 18],
                  [19, 20, 21, 22]]])
abcd
```

Out[111]:

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

       [[11, 12, 13, 14],
        [15, 16, 17, 18],
        [19, 20, 21, 22]]])
```

In [112]:

```
print(abcd.ndim)
print(abcd.shape)
```

```
3
(2, 3, 4)
```

3.배열의 인덱싱(Indexing)

3.1 인덱싱(Indexing)

- 배열 객체로 구현한 다차원 배열의 원소 중 "하나"의 개체를 선택
- 콤마로 구분된 차원을 축(axis)이라 하며, 그래프의 (x, y)축과 동일

In [30]:

```
# 1차원 배열 생성
a = np.array([0, 1, 2, 3, 4])
a
```

Out[30]:

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

In [31]:

```
a[2]
```

Out[31]:

```
2
```

In [32]:

```
a[-1]
```

Out[32]:

```
4
```

In [33]:

```
# 2차원 배열 생성
b = np.array([[0, 1, 2], [3, 4, 5]])
b
```

Out[33]:

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

In [34]:

```
# 첫번째 행의 첫번째 열
b[0, 0]
```

Out[34]:

```
0
```

In [35]:

```
# 첫번째 행의 두번째 열
b[0, 1]
```

Out[35]:

```
1
```

In [36]:

```
# 마지막 행의 마지막 열
b[-1, -1]
```

Out[36]:

```
5
```

3.2 불리언 인덱싱(Blean Indexing)

불리안 배열 인덱싱 방식은 인덱스 배열의 원소가 True, False 두 값으로만 구성되며 인덱스 배열의 크기가 원래 ndarray 객체의 크기와 같아야 한다

In [37]:

```
# 1차원 배열 생성
a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
a
```

Out[37]:

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

In [38]:

```
idx = np.array([True, False, True, False, True,
                False, True, False, True, False])
idx
```

Out[38]:

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

In [39]:

```
a[idx]
```

Out[39]:

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

In [40]:

```
a % 2
```

Out[40]:

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

In [41]:

```
a % 2 == 0
```

Out[41]:

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

In [42]:

```
a[a % 2 == 0]
```

Out[42]:

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

3.3 슬라이싱(Slicing)

배열 객체로 구현한 다차원 배열의 원소 중 "복수 개"를 선택
일반적인 파이썬의 슬라이싱(slicing)과 comma(,)를 함께 사용

Slicing 사용 예

- [:] 배열 전체
- [0:n] 0번째부터 n-1번째까지, 즉 n번 항목은 포함하지 않는다.
- [:5] 0번째부터 4번째까지,5번은 포함하지 않는다.
- [2:] 2번째부터 끝까지
- [-1] 제일 끝에 있는 배열값 반환
- [-2] 제일 끝에서 두번째 값 반환

In [43]:

```
# 2차원 배열 생성
a = np.array([[0, 1, 2, 3],
              [4, 5, 6, 7]])
a
```

Out[43]:

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

In [44]:

```
# 첫번째 행 전체
a[0, :]
```

Out[44]:

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

In [45]:

```
# 두번째 행의 두번째 열부터 끝열까지
a[1, 1:]
```

Out[45]:

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

In [46]:

```
# 첫번째 행의 두번째 열, 두번째 행의 두번째 열까지
a[:2, :2]
```

Out[46]:

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

4. 데이터 타입

- ndarray클래스는 데이터가 같은 자료형
- array 명령으로 배열을 만들 때 자료형 지정은 dtype 사용

dtype 접두사	설명	사용 예
b	불리언	b (참 혹은 거짓)
i	정수	i8 (64비트)
u	부호 없는 정수	u8 (64비트)
f	부동소수점	f8 (64비트)
c	복소 부동소수점	c16 (128비트)
O	객체	O (객체에 대한 포인터)
S	바이트 문자열	S24 (24 글자)
U	유니코드 문자열	U24 (24 유니코드 글자)

4.1 데이터 타입 확인

In [47]:

```
# 정수 배열 입력
a = np.array([1, 2, 3])
a.dtype
```

Out[47]:

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

In [48]:

```
# 실수 배열 입력
b = np.array([1.0, 2.0, 3.0])
b.dtype
```

Out[48]:

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

In [49]:

```
# 배열에 하나라도 실수인자가 있으면 실수형
c = np.array([1, 2, 3.0])
c.dtype
```

Out[49]:

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

In [50]:

```
# 정수형을 실수형으로 바꾸기
d = np.array([1, 2, 3], dtype="f")
d.dtype
```

Out[50]:

```
dtype('float32')
```

4.2 numpy inf와 non

- 무한대를 표현하기 위한 np.inf(infinity)와 정의할 수 없는 숫자를 나타내는 np.nan(not a number)
  예) 1을 0으로 나누거나 0에 대한 로그 값을 계산하면 무한대인 np.inf 0을 0으로 나누면 np.nan이 나온다.

In [51]:

```
np.array([0, 1, -1, 0]) / np.array([1, 0, 0, 0])
```

```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:1: RuntimeWarning: divide by zero encou
ntered in true_divide
  """Entry point for launching an IPython kernel.
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:1: RuntimeWarning: invalid value encoun
tered in true_divide
  """Entry point for launching an IPython kernel.
```

Out[51]:

```
array([  0.,  inf, -inf,  nan])
```

In [52]:

```
# 로그함수
np.log(0)
```

```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:2: RuntimeWarning: divide by zero encou
ntered in log
```

Out[52]:

```
-inf
```

In [53]:

```
# 지수함수
np.exp(-np.inf)
```

Out[53]:

```
0.0
```

5. 배열 생성

NumPy는 단한 배열을 생성하는 명령을 제공

- zeros, ones
- zeros_like, ones_like
- empty
- arange
- linspace, logspace
- rand, randn

5.1 zeros

'0'으로 초기화된 배열을 생성

In [54]:

```
# 0으로 된  1차원 배열 생성
a = np.zeros(5)
a
```

Out[54]:

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

In [55]:

```
a.dtype
```

Out[55]:

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

In [56]:

```
# 0으로 된 2x3 2차원 배열 생성
b = np.zeros((2, 3))
b
```

Out[56]:

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

In [57]:

```
b.dtype
```

Out[57]:

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

In [58]:

```
# 0으로 된 5x2 정수형 배열 생성(명시하지 않으면 실수)
c = np.zeros((5, 2), dtype="i")
c
```

Out[58]:

```
array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]], dtype=int32)
```

5.2 ones

'1'로 초기화된 배열을 생성

In [59]:

```
d = np.ones((2, 3, 4), dtype="i8")
d
```

Out[59]:

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

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]])
```

In [60]:

```
d.dtype
```

Out[60]:

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

5.3 arrange

Python 기본 명령어 range와 같은 특정한 규칙에 따라 증가하는 수열을 생성

In [61]:

```
# 배열을  순차적으로 0 ... n-1 까지 생성
np.arange(10)
a
```

Out[61]:

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

In [62]:

```
# (시작, 끝(포함하지 않음), 단계간격)
np.arange(1, 21, 2)
```

Out[62]:

```
array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19])
```

5.4 전치연산

행과 열을 바꾸는 전치(transpose) 연산으로 t 속성 사용

In [63]:

```
# 3x2 배열 생성
a = np.array([[1, 2, 3], [4, 5, 6]])
a
```

Out[63]:

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

In [64]:

```
# 3x2 배열을 2x3 배열로 전치
a.T
```

Out[64]:

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

5.5 배열의 크기 변환

만들어진 배열의 내부 데이터는 보존한 채로 형태만 reshape 명령어나 메서드로 변형

In [65]:

```
# 12개 원소를 가진 1차원 배열 생성
a = np.arange(12)
a
```

Out[65]:

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

In [66]:

```
#  1차원 배열을 reshape 매서드로 3x4 행렬로 변형
b = a.reshape(3, 4)
b
```

Out[66]:

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

In [67]:

```
# -1을 사용하여 동일하게 변형
b.reshape(3, -1)
```

Out[67]:

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

6. 배열의 연산

6.1 벡터화 연산(vectorized operation)

x 벡터와 y 벡터의 두 벡터 연산 시

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ 10000 \end{bmatrix}, \quad y = \begin{bmatrix} 10001 \\ 10002 \\ 10003 \\ \vdots \\ 20000 \end{bmatrix},$$

일 때, 두 벡터의 합

$$z = x + y$$

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ 10000 \end{bmatrix} + \begin{bmatrix} 10001 \\ 10002 \\ 10003 \\ \vdots \\ 20000 \end{bmatrix} = \begin{bmatrix} 1 + 10001 \\ 2 + 10002 \\ 3 + 10003 \\ \vdots \\ 10000 + 20000 \end{bmatrix} = \begin{bmatrix} 10002 \\ 10004 \\ 10006 \\ \vdots \\ 30000 \end{bmatrix}$$

In [68]:

```
# arange 수열로 1...10001, 10001...20001까지 생성
x = np.arange(1, 10001)
y = np.arange(10001, 20001)
x, y
```

Out[68]:

```
(array([    1,     2,     3, ...,  9998,  9999, 10000]),
 array([10001, 10002, 10003, ..., 19998, 19999, 20000]))
```

In [69]:

```
# 두 벡터의 합
z = x + y
z
```

Out[69]:

```
array([10002, 10004, 10006, ..., 29996, 29998, 30000])
```

6.2 벡터 연산 - 지수, 제곱, 로그함수

In [70]:

```
# 1...5까지 1차원 배열 생성
a = np.arange(5)
a
```

Out[70]:

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

In [71]:

```
# 지수함수
np.exp(a)
```

Out[71]:

```
array([ 1.        ,  2.71828183,  7.3890561 , 20.08553692, 54.59815003])
```

In [72]:

```
# 10에 a승
10 ** a
```

Out[72]:

```
array([    1,    10,   100,  1000, 10000])
```

In [73]:

```
# 로그함수
np.log(a + 1)
```

Out[73]:

```
array([0.        , 0.69314718, 1.09861229, 1.38629436, 1.60943791])
```

6.3 스칼라와 벡터 연산

In [74]:

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

Out[74]:

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

In [75]:

```
# 스칼라 100과 x 벡터의 곱하기
100 * x
```

Out[75]:

```
array([  0, 100, 200, 300, 400, 500, 600, 700, 800, 900])
```

In [76]:

```
# 0...12까지 수열 생성 후 3x4 벡터 생성
x = np.arange(12).reshape(3, 4)
x
```

Out[76]:

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

In [77]:

```
# 스칼라 100과 x 벡터의 곱하기
100 * x
```

Out[77]:

```
array([[   0,  100,  200,  300],
       [ 400,  500,  600,  700],
       [ 800,  900, 1000, 1100]])
```

6.4 브로드캐스팅(Broadcasting)

- 벡터 연산 시에 두 벡터의 크기가 동일해야 한다.
- 서로 다른 크기를 가진 두 배열의 사칙 연산은 브로드캐스팅(broadcasting)으로 크기가 작은 배열을 자동으로 반복 확장하여 크기가 큰 배열에 맞춰준다.

$$x = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}, \quad x + 1 = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 1 = ?$$

브로드캐스팅은 다음과 같이 스칼라를 벡터와 같은 크기로 확장시켜서 덧셈 계산

$$\begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 1 = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

In [132]:

```
# 1차원 배열 생성
x = np.array([[0, 1, 2], [1, 2, 3], [2, 3, 4], [4, 5, 6]])
x
```

Out[132]:

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

In [133]:

```
# 1로 된 배열 생성
y = np.arange(3)
y
```

Out[133]:

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

In [134]:

```
# 연산하기
x + y
```

Out[134]:

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

In [135]:

```
# 벡터와 스칼라 연산
x + 1
```

Out[135]:

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

7. 차원축소 연산

차원 축소(Dimension Reduction) 연산은 행렬의 하나의 행에 있는 원소들을 하나의 데이터 집합으로 보고 그 집합의 평균을 구하면 1차원 벡터가 반환

numpy의 차원 축소 연산 메서드

- 최대/최소: min, max, argmin, argmax
- 통계: sum, mean, median, std, var
- 불리언: all, any

7.1 1차원 배열의 차원숙소 연산

In [82]:

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

Out[82]:

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

In [83]:

```
# numpy 합계 sum
np.sum(x)
```

Out[83]:

```
10
```

In [84]:

```
# 합계 sum
x.sum()
```

Out[84]:

```
10
```

In [85]:

```
# 최소값
x.min()
```

Out[85]:

```
1
```

In [86]:

```
# 최대값
x.max()
```

Out[86]:

```
4
```

In [87]:

```
# 위치 최소
x.argmin()
```

Out[87]:

```
0
```

In [88]:

```
# 위치 최대
x.argmax()
```

Out[88]:

```
3
```

In [89]:

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

Out[89]:

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

In [90]:

```
# 평균값
y.mean()
```

Out[90]:

```
1.75
```

In [91]:

```
# 중간값
np.median(y)
```

Out[91]:

```
1.5
```

7.2 2차원 배열의 차원숙소 연산

- 연산의 대상이 2차원 이상인 경우에는 어느 차원으로 계산을 할 지를 axis 인수를 사용
- axis=0인 경우는 열 연산, axis=1인 경우는 행 연산(디폴트 값은 axis=0)

In [92]:

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

Out[92]:

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

In [93]:

```
x.sum()
```

Out[93]:

```
10
```

In [94]:

```
# axis=0 인자를 사용하여 열 합계 구하기
x.sum(axis=0)
```

Out[94]:

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

In [95]:

```
# axis=1 인자를 사용하여 행 합계 구하기
x.sum(axis=1)
```

Out[95]:

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

8. numpy를 이용한 기술통계(descriptive statistics)

샘플 x의 집합이 아래와 같다면,
x = {18, 5, 10, 23, 19, -5, 10, 0, 0, 5, 2, 126, 8, 2, 5, 5, 15, -3, 4, -1, -20 ,8, 9, -4, 25, -12}

In [96]:

```
x = np.array([18, 5, 10, 23, 19, -5, 10, 0, 0, 5, 2, 126, 8, 2, 5, 5, 15, -3, 4, -1, -20 ,8, 9, -4, 25, -12])
x
```

Out[96]:

```
array([ 18,   5,  10,  23,  19,  -5,  10,   0,   0,   5,   2, 126,   8,
         2,   5,   5,  15,  -3,   4,  -1, -20,   8,   9,  -4,  25, -12])
```

In [97]:

```
# 데이터 개수
len(x)
```

Out[97]:

```
26
```

In [98]:

```
# 평균값
np.mean(x)
```

Out[98]:

```
9.76923076923077
```

In [99]:

```
# 분산값
np.var(x)
```

Out[99]:

```
637.9467455621302
```

In [100]:

```
# 표준편차
np.std(x)
```

Out[100]:

```
25.257607676938253
```

In [101]:

```
# 최대값
np.max(x)
```

Out[101]:

```
126
```

In [102]:

```
# 최소값
np.min(x)
```

Out[102]:

```
-20
```

In [103]:

```
# 중앙값
np.median(x)
```

Out[103]:

```
5.0
```

9. numpy 난수생성

데이터를 무작위로 섞거나 임의의 수 즉, 난수(random number)를 발생시키는 numpy의 random 서브패키지 사용 명령어는

- rand: 0부터 1사이의 균일 분포
- randn: 가우시안 표준 정규 분포
- randint: 균일 분포의 정수 난수

In [104]:

```
# rand로 1차원 벡터 난수 발생
np.random.rand(10)
```

Out[104]:

```
array([9.80114329e-01, 7.33486865e-01, 8.84953666e-04, 6.56479127e-01,
       3.60124550e-01, 2.82663073e-01, 5.02073274e-01, 6.37992651e-01,
       2.67115693e-01, 6.45095731e-01])
```

In [105]:

```
# rand로 3x2 벡터 난수 발생
np.random.rand(3, 2)
```

Out[105]:

```
array([[0.57697406, 0.10174202],
       [0.43666112, 0.30022151],
       [0.09264312, 0.39533411]])
```

randint 명령

numpy.random.randint(low, high=none, size=none)
(만약 high를 입력하지 않으면 0과 low사이의 숫자를, high를 입력하면 low와 high는 사이의 숫자를 출력하고, size는 난수의 숫자)

In [106]:

```
# 10부터 20까지 10개 인자의 배열 생성
np.random.randint(10, 20, size=10)
```

Out[106]:

```
array([19, 15, 14, 19, 18, 13, 11, 18, 18, 16])
```

In [107]:

```
# 10부터 20까지 3x5 벡터 생성
np.random.randint(10, 20, size=(3, 5))
```

Out[107]:

```
array([[11, 17, 19, 12, 17],
       [19, 18, 18, 19, 12],
       [12, 17, 14, 13, 10]])
```

In [108]:

```
# DataKubwa - nowave
```