Python에서의 선형 대수

- 이 단원에서는 선형 대수의 여러 개념을 Python 기본 함수와 리스트 등을 이용하여 구현하는 연습을 한다.
- 벡터를 생성하고, 벡터와 행렬 연산을 수행할 수 있는 함수를 작성한다.
- 이 후, Python 모듈인 numpy를 이용하여 직접 만들었던 선형 대수의 기능과 비교해 본다.

벡터 - Vectors

- 벡터는 벡터 공간의 원소를 벡터라 하며,
- 백터들은 서로 더하거나 스칼라에 의해 곱해질 수 있다.
- 벡터를 숫자들의 리스트라고 생각해 보자.

In [1]:

```
height_weigth_age = [70,    # inches
                     170,   # pounds
                     40 ]   # years
```

In [2]:

```
grades = [95,    # exam1
          80,    # exam2
          75,    # exam3
          62 ]   # exam4
```

- Python의 list는 벡터 연산을 제공하지 않기 때문에, 벡터 연산을 추가해 보자.

벡터 합과 차

- 원소별로 계산 : 다음을 구현하고 싶음
  - [1, 2] 더하기 [3, 4] = [4, 6]
  - [5, 3] 빼기 [1, 7] = [4, -4]

In [3]:

```
def vector_add(v, w):
    """adds two vectors componentwise"""
    return [v_i + w_i for v_i, w_i in zip(v,w)]
```

In [4]:

```
def vector_subtract(v, w):
    """subtracts two vectors componentwise"""
    return [v_i - w_i for v_i, w_i in zip(v,w)]
```

여러 개의 벡터들의 합

- 벡터들의 리스트가 있을 때, 리스트 내의 벡터들을 원소 별로 합하기

In [5]:

```
def vector_sum(vectors):
    result = vectors[0]
    for vector in vectors[1:]:
        result = vector_add(result, vector)
    return result
```

In [6]:

```
def vector_sum(vectors):
    return reduce(vector_add, vectors)
```

스칼라 곱

- 목표 : 3*[1, 2, 3] = [3, 6, 9]

In [7]:

```
def scalar_multiply(c, v):
    return [c* vi for vi in v]
```

- 컴포넌트별 평균 : vector_mean([1,2],[2,4],[3,6]) == [3,6]

In [8]:

```
def vector_mean(vectors):
    n = len(vectors)
    return scalar_multiply(1/n, vector_sum(vectors))
```

- 단, Python 2.x의 경우 위의 나누기에 실수 나누기를 적용하기 위해서는 파일 위쪽에 다음을 표기
- Python 3.x는 실수 나누기가 적용되기 때문에 상관 없음

In [9]:

```
from __future__ import division
```

dot product

- 목표 :
  - dot([1, 2, 3], [0, 1, 2]) = sum([1*0, 2*1, 3*2]) = sum([0, 2, 6]) = 8

In [10]:

```
def dot(v, w):
    """v_1 * w_1 + ... + v_n * w_n"""
    return sum(v_i * w_i for v_i, w_i in zip(v, w))
```

제곱합

- 벡터 원소들의 제곱의 합

In [11]:

```
def sum_of_squares(v):
    """v_1 * v_1 + ... + v_n * v_n"""
    return dot(v, v)
```

- 벡터의 크기(magnitude) : 제곱합의 제곱근

In [12]:

```
import math
def magnitude(v):
    return math.sqrt(sum_of_squares(v))
```

벡터 사이의 거리

- 한 벡터에서 다른 벡터를 뺀 후, 크기를 구하는 것과 동일

In [13]:

```
def distance(v, w):
   return magnitude(vector_subtract(v, w))
```

NumPy

- numpy 모듈에는 지금까지 행한 벡터 연산들이 구현되어 있음
- numpy 패키지 (모듈)는 Python을 사용하는 거의 모든 수치 계산에 사용된다.
- Python을 위한 벡터, 행렬 및 고차원 데이터 구조를 제공한다.
- http://www.numpy.org/ (http://www.numpy.org/)

numpy 모듈을 이용하기 위해서는 다음과 같이 import를 먼저 진행한다. 한 번만 불러오면 된다.

In [14]:

```
import numpy as np
```

다음은 dot product 예제이다.

In [15]:

```
# dot product
np.dot([1,2], [3,4])
```

Out[15]:

11

Numpy array

- numpy의 다양한 기능은 array라는 데이터구조를 바탕으로 이루어진다.
- numpy는 다양한 방법을 통해 만들 수 있다.
  - 파이썬 리스트 또는 튜플을 이용하는 방법
  - arange, linspace 등과 같이 numpy 배열을 생성하는 데 사용되는 함수를 사용하는 방법
  - 파일에서 데이터를 읽어들이는 방법
- 다차원 배열을 구현
- 수학적 계산에 특화

Python list로부터 numpy array 만들기

In [39]:

```
import numpy as np  # 이미 한 번 import하였으면 다시 하지 않아도 된다.
a = np.array([0, 1, 2, 3])

a
```

Out[39]:

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

In [38]:

```
print(a)
```

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

In [17]:

```
a.ndim  # 1 차원
```

Out[17]:

```
1
```

In [18]:

```
a.shape  # 형태 : (4,)
```

Out[18]:

```
(4,)
```

In [19]:

```
len(a)  # 4
```

Out[19]:

```
4
```

In [37]:

```
# matrix: Python list로 이루어진 list를 이용하여 matrix 만들기
b = np.array([[0,1,2], [3,4,5]])

b
```

Out[37]:

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

In [21]:

```
b.ndim  # 2차원
```

Out[21]:

```
2
```

In [22]:

```
b.shape # 형태 (2,3)
```

Out[22]:

```
(2, 3)
```

In [23]:

```
len(b)  # 2 : 첫번째 차원의 길이
```

Out[23]:

```
2
```

array는 list와 비슷해 보이지만, numpy에서 array라는 별도의 데이터구조를 이용하는 몇 가지 이유가 있다.

- Python list는 동적으로 할당되며, list내의 원소들이 서로 다른 데이터형을 가질 수 있다.
- 이러한 점은 벡터나 행렬 계산을 느리게 혹은 불가능하게 한다.
- 반면 numpy array내의 원소들의 데이터 형은 일정하고(homogeneous), 변하지 않기 때문에, 메모리 효율적이고, 행렬이나 벡터 계산을 빠르게 할 수 있다.
- 이미 데이터 타입이 결정된 array의 원소를 다른 데이터 타입으로 변경하면 에러가 발생한다.

In [25]:

```
b.dtype
```

Out[25]:

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

In [26]:

```
b[0,0] = "a"
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-26-9c6a29519c7a> in <module>()
----> 1 b[0,0] = "a"

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

asarray는 복사를 하여 새로운 array를 만드는 대신, 참조를 한다. 따라서, 아래 예제에서 y의 변화는 x에 영향을 미친다.

In [62]:

```
x = np.array([0, 1, 2, 3])

y = np.asarray(x)

y[0] = 10

x
```

Out[62]:

```
array([10,  1,  2,  3])
```

array 생성

numpy에서는 다양한 방법을 통해 array를 생성할 수 있도록 도와준다.

In [40]:

```
a = np.arange(10)  # range 함수와 비슷

a
```

Out[40]:

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

In [41]:

```
b = np.arange(1, 9, 2)

b
```

Out[41]:

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

In [42]:

```
c = np.linspace(0, 1, 6)  #시작, 끝, 숫자 개수

c
```

Out[42]:

```
array([0. , 0.2, 0.4, 0.6, 0.8, 1. ])
```

In [43]:

```
d = np.ones((3, 3))  # 1로 이루어진 다차원 배열

d
```

Out[43]:

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

In [44]:

```
e = np.zeros((2, 2))  # 0으로 이루어진 다차원 배열

e
```

Out[44]:

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

In [45]:

```
f = np.eye(3)  # identity 행렬

f
```

Out[45]:

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

In [46]:

```
g = np.diag(np.array([1, 2, 3, 4]))  # 대각 행렬

g
```

Out[46]:

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

array 생성(2)

numpy.random 모듈을 이용한 랜덤 array 생성

In [47]:

```
np.random.rand(4)       # uniform in [0, 1]
```

Out[47]:

```
array([0.67490839, 0.44209155, 0.12471685, 0.51862441])
```

In [48]:

```
np.random.randn(4)      # standard normal
```

Out[48]:

```
array([-0.36397872, -0.55903025, -2.23781386,  0.89005866])
```

In [50]:

```
2.5 * np.random.randn(4) + 3   # 평균 3, 표준편차 2.5인 정규분포
```

Out[50]:

```
array([ 6.75050912,  3.08513051,  2.36146634, -1.97171902])
```

Indexing

Python list와 비슷하게 [ ]를 이용하여 원소에 접근한다.

In [65]:

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

In [66]:

```
v[0]
```

Out[66]:

```
1
```

In [67]:

```
M[1,1]
```

Out[67]:

```
4
```

In [68]:

```
M[1]
```

Out[68]:

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

In [69]:

```
M[1,:] # row 1
```

Out[69]:

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

In [70]:

```
M[:,1] # column 1
```

Out[70]:

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

In [71]:

```
M[0,0] = 10

M
```

Out[71]:

```
array([[10,  2],
       [ 3,  4]])
```

In [73]:

```
M[1,:] = -1

M
```

Out[73]:

```
array([[10,  2],
       [-1, -1]])
```

In [75]:

```
M[:,1] = 777

M
```

Out[75]:

```
array([[ 10, 777],
       [ -1, 777]])
```

In [77]:

```
A = np.array([[n+m*10 for n in range(5)] for m in range(5)])

A
```

Out[77]:

```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

In [79]:

```
A[1:3, 1:3]
```

Out[79]:

```
array([[11, 12],
       [21, 22]])
```

In [80]:

```
A[::2, ::2]
```

Out[80]:

```
array([[ 0,  2,  4],
       [20, 22, 24],
       [40, 42, 44]])
```

In [81]:

```
row_indices = [1, 2, 3]
A[row_indices]
```

Out[81]:

```
array([[10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

In [83]:

```
col_indices = [1, 2, -1]            # -1은 마지막 원소를 나타냄
A[row_indices, col_indices]
```

Out[83]:

```
array([11, 22, 34])
```

In [85]:

```
B = np.array([n for n in range(5)])
B
```

Out[85]:

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

In [87]:

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

Out[87]:

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

In [89]:

```
# same thing
row_mask = np.array([1,0,1,0,0], dtype=bool)
B[row_mask]
```

Out[89]:

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

In [91]:

```
x = np.arange(0, 10, 0.5)
x
```

Out[91]:

```
array([0. , 0.5, 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])
```

In [94]:

```
x[(5 < x) * (x < 7.5)]
```

Out[94]:

```
array([5.5, 6. , 6.5, 7. ])
```

numpy와 numpy.linalg를 이용한 선형대수

- numpy는 벡터와 행렬 연산에 있어 강력한 기능을 가지고 있다.

벡터 합과 차

In [97]:

```
a = np.array([3,1,-1])
b = np.arange(3)
c = a + b
d = b - a
```

- 스칼라 곱

In [98]:

```
4 * a
```

Out[98]:

```
array([12,  4, -4])
```

- 제곱합

In [99]:

```
np.sum(a**2)
```

Out[99]:

11

- dot product

In [100]:

```
np.dot(a, b)
```

Out[100]:

-1

- 벡터 원소별 곱셈

In [118]:

```
a * b
```

Out[118]:

array([0, 1, -2])

- 벡터 사이의 거리

In [101]:

```
np.linalg.norm(a-b)
```

Out[101]:

4.242640687119285

In [126]:

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

M
```

Out[126]:

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

In [127]:

```
N = np.array([[-1,1],[2,1]])

N
```

Out[127]:

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

- 역행렬

In [104]:

```
np.linalg.inv(M)
```

Out[104]:

```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

- 행렬식

In [48]:

```
np.linalg.det(M)
```

Out[48]:

```
-2.0000000000000004
```

- transpose

In [125]:

```
M.T
```

Out[125]:

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

- 해 찾기

In [49]:

```
c = np.array([2,1])
np.linalg.solve(M, c)
```

Out[49]:

```
array([-3. ,  2.5])
```

- 행렬곱

In [50]:

```
np.matmul(M,N)
```

Out[50]:

```
array([[3, 3],
       [5, 7]])
```

- 행렬의 원소별 곱셈

In [106]:

```
M * N
```

Out[106]:

```
array([[-1,  2],
       [ 6,  4]])
```

복사와 참조

- 성능 향상을 위해 파이썬에서의 많은 경우 복사를 하지 않고 참조를 하는 경우가 많다.

In [130]:

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

A
```

Out[130]:

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

In [131]:

```
# B는 A를 참조만 한다.
B = A
```

In [132]:

```
# B의 변화는 A에 영향을 미침
B[0,0] = 10

B
```

Out[132]:

```
array([[10,  2],
       [ 3,  4]])
```

- 만약 이런 현상을 원치 않는다면, copy()를 이용하여 복사한다.

In [134]:

```
C = np.copy(A)
C[0, 0] = -1

C
```

Out[134]:

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

In [135]:

```
# A는 바뀌지 않음
A
```

Out[135]:

```
array([[10,  2],
       [ 3,  4]])
```