

게임 수학

동명대학교 게임공학과

강영민

무엇을 다룰 것인가

- 벡터와 행렬

- 벡터의 의미

- 벡터(vector)는 ‘나르다’라는 의미의 라틴어 동사 ‘vehere’에서 유래
 - ‘무엇인가를 나르는 것’이라는 의미
 - 벡터라는 것은 무엇인가를 옮겨 놓는 역할을 수행한다.

- 수학과 물리학에서의 개념

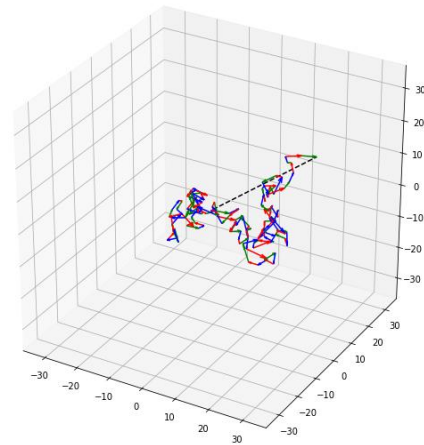
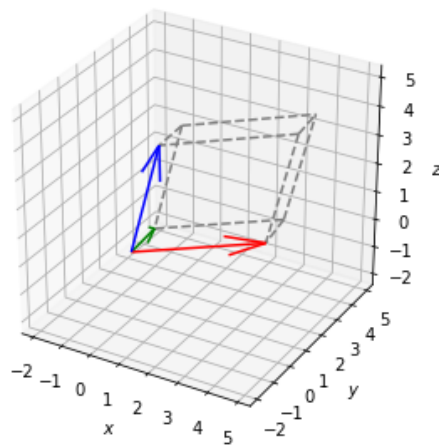
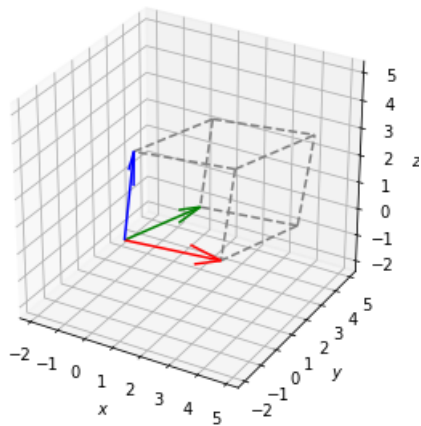
- 크기와 방향으로 결정되는 양(量, quantity)
 - 방향량(方向量)이라고도 함.
 - 예: 힘(force)은 크기만으로는 그 성질을 온전히 표현할 수 없고, 방향도 같이 고려해야 하므로 벡터로 표현된다.

왜?

- 게임은 벡터와 행렬로 데이터를 표현하고 조작한다

벡터

수학자들은 자연 현상의 많은 것들이 수로 표현될 수 있음을 알았는데, 하나의 숫자로 충분히 표현할 수 있지만 하나 이상의 수가 필요한 경우도 있다. 이러한 양을 벡터(vector)라고 부른다.



행렬은 왜?

- 행렬은 2차원으로 배열된 수
 - 벡터의 집합으로 해석 가능
 - 벡터와 곱해져서 새로운 벡터 만들 수 있음
 - 벡터를 바꾸어 놓는 연산자 – 트랜스포머?
 - 행렬은 벡터를 조작하는 도구

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1j} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2j} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{ij} & \cdots & a_{in} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mj} & \cdots & a_{mn} \end{bmatrix}$$

실습은 어떻게 진행하나

- 파이썬 프로그래밍
 - 수학적 개념을 실제로 코드로 구현해 봄
 - 벡터의 연산, 행렬의 연산 결과를 숫자와 시각 정보로 확인
- 파이썬 코딩을 하는 방법
 - 다양한 방법이 존재
 - Python interpreter 설치 + 내게 맞는 IDE (예 – visual studio code)
 - 우리가 사용할 방법
 - Google Colab 사용

구글 코랩 사용법

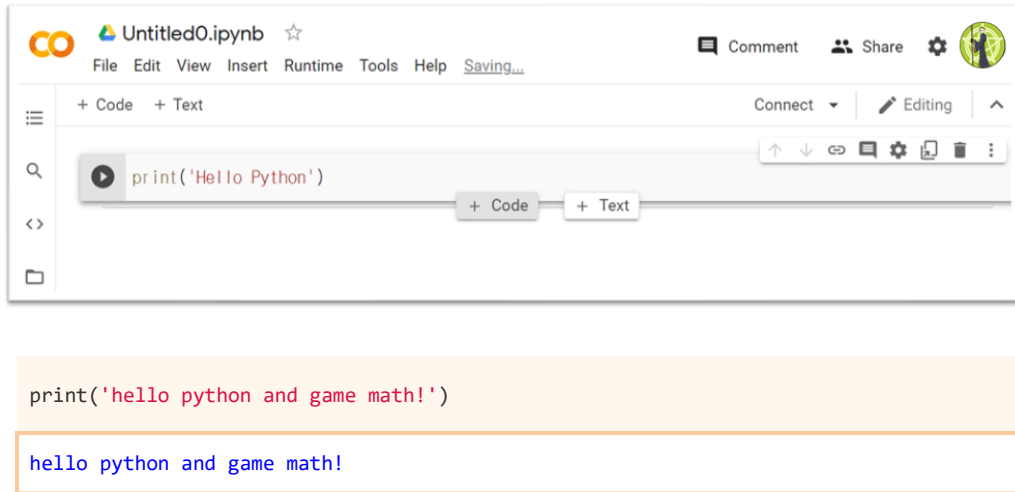
- 장점

- 파이썬을 설치하지 않고도 웹 환경의 주피터 노트북에서 파이썬 사용 가능
- 다른 사용자와의 파일 공유가 가능하며 협업을 통한 개발도 손쉽게 할 수 있다.
- 넘파이, 판다스, 사이킷런, 텐서플로우 등의 패키지가 미리 설치된 환경 제공
- 클라우드에서 제공하는 GPU와 TPU를 사용하여 고성능 계산을 할 수 있다.

- 사용방법

- 구글 계정이 있다면 아래 사이트에 접속해서 사용
- <https://colab.research.google.com>

사용 예시

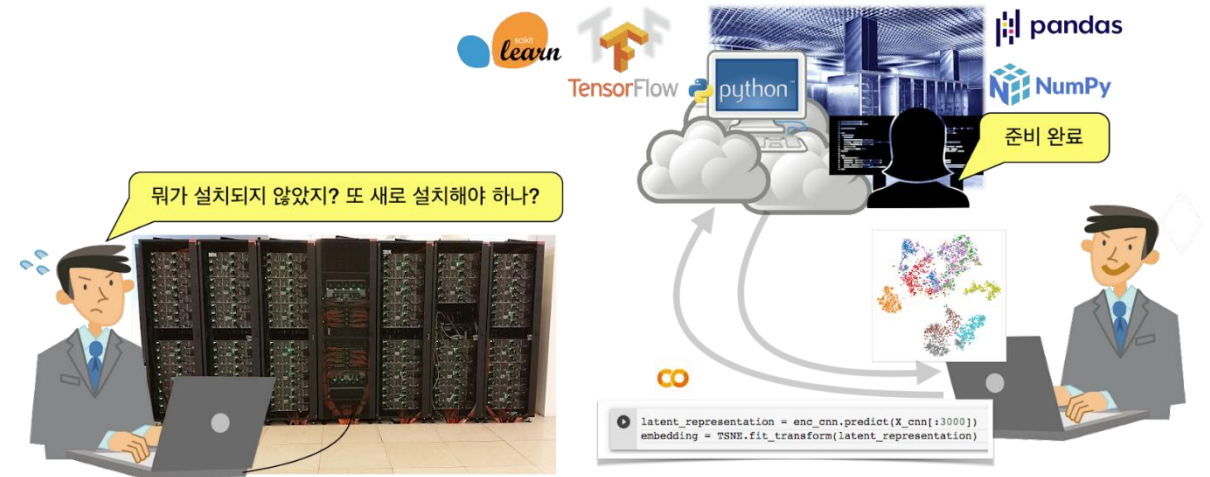


The screenshot shows a Jupyter Notebook titled 'Untitled0.ipynb'. The code cell contains the following Python code:

```
print('Hello Python')
```

The output of the code cell is displayed below the code:

```
hello python and game math!
```



시스템 관리 작업에 신경 쓸 필요 없이 관심 있는 이론과, 이를 구현하는 코드에만 집중할 수 있을 것이다.

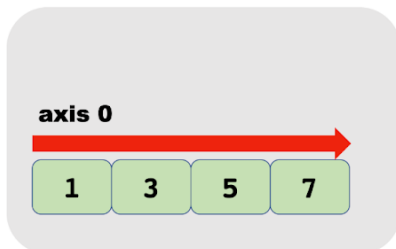
사용할 패키지 - 넘파이

- 넘파이는 파이썬에서 수치 데이터를 다루는 패키지
 - 가장 기본적이고 강력한 패키지
 - 데이터 분석이나 기계 학습에서 활발히 활용
 - 데이터 분석을 위한 패키지인 판다스Pandas
 - 머신러닝을 위한 Scikit-learn, Tensorflow 등이 넘파이 위에서 작동

```
import numpy as np
my_array = np.array( [ 1, 2, 3 ] )
```

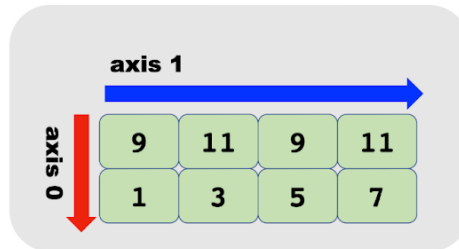
numpy의 별칭으로 np를 지정함
np는 numpy의 별칭

1D 배열



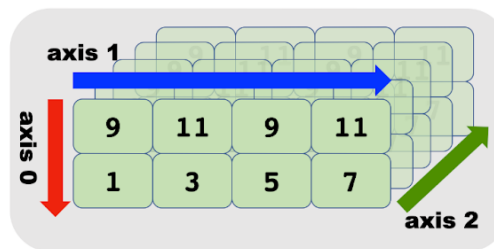
shape = (4,)

2D 배열



shape = (2,4)

3D 배열



shape = (2,4,5)

넘파이 배열과 파이썬 리스트의 차이

- 리스트는 벡터 데이터가 아님

```
store_a = [20, 10, 30] # 매장 A의 매출  
store_b = [70, 90, 70] # 매장 B의 매출  
list_sum = store_a + store_b  
list_sum
```

```
[20, 10, 30, 70, 90, 70]
```

- 넘파이의 배열은 벡터, 행렬을 표현하는 데에 적합

```
import numpy as np  
np_store_a = np.array(store_a) # 넘파이를 임포트하고 np라는 별칭 지정  
np_store_b = np.array(store_b) # store_a 리스트를 넘파이 배열로 변환  
array_sum = np_store_a + np_store_b # store_b 리스트를 넘파이 배열로 변환  
array_sum
```

```
array([ 90, 100, 100])
```

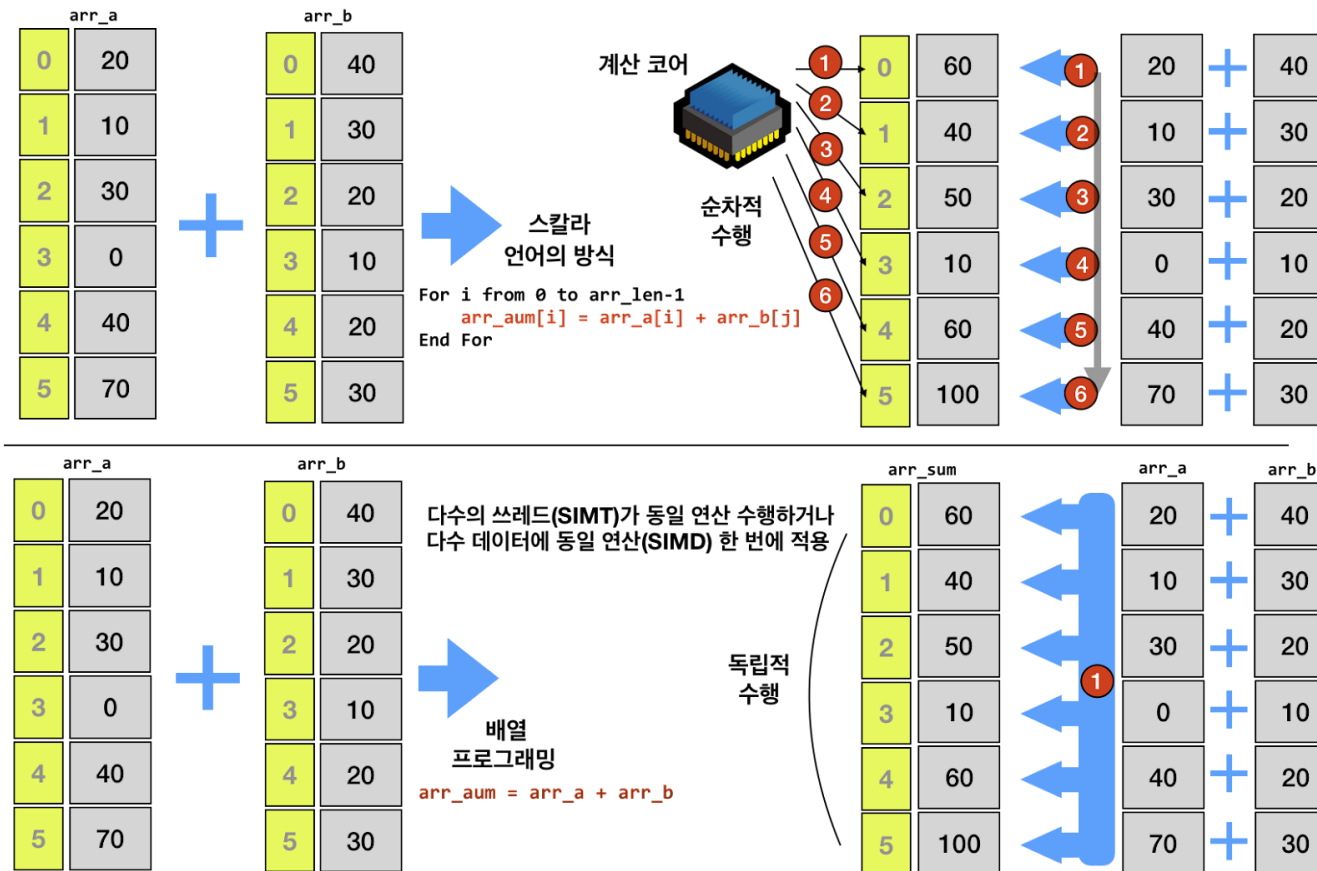
넘파이 배열의 다양한 속성

- 아래 넘파이 배열 속성을 점검하여 계산이나 오류 파악에 도움을 얻을 수 있다

속성	설명
ndim	배열 축 혹은 차원의 수
shape	배열의 차원으로 (m, n) 형식의 튜플. 이 때, m과 n은 각 차원별 원소의 크기
size	배열 원소의 수
dtype	배열에 담긴 원소의 자료형
itemsize	배열에 담긴 원소의 크기를 바이트 단위로 나타냄
data	배열의 실제 원소들을 가지고 있는 메모리 버퍼
stride	배열 각 차원별로 다음 원소로 점프하는 데에 필요한 거리를 바이트로 표시

넘파이의 성공 요인

- 배열 프로그래밍 – array programming



배열 프로그래밍의 성능 확인 1/2

```
import numpy as np
import time
arr_a = np.random.rand(100)
arr_b = np.random.rand(100)
result = np.zeros(100)
start = time.time()
for i in range(len(values)):
    result[i] = arr_a[i] * arr_b[i]
end = time.time()
print('소요시간: ', end - start)
```

명시적인 작업 지시 : C 언어 스타일

소요시간: 0.0012476444244384766

배열 프로그래밍의 성능 확인 2/2

```
start = time.time()
result = arr_a * arr_b          # 벡터화 연산 *
end = time.time()
print('소요시간: ', end - start)
```

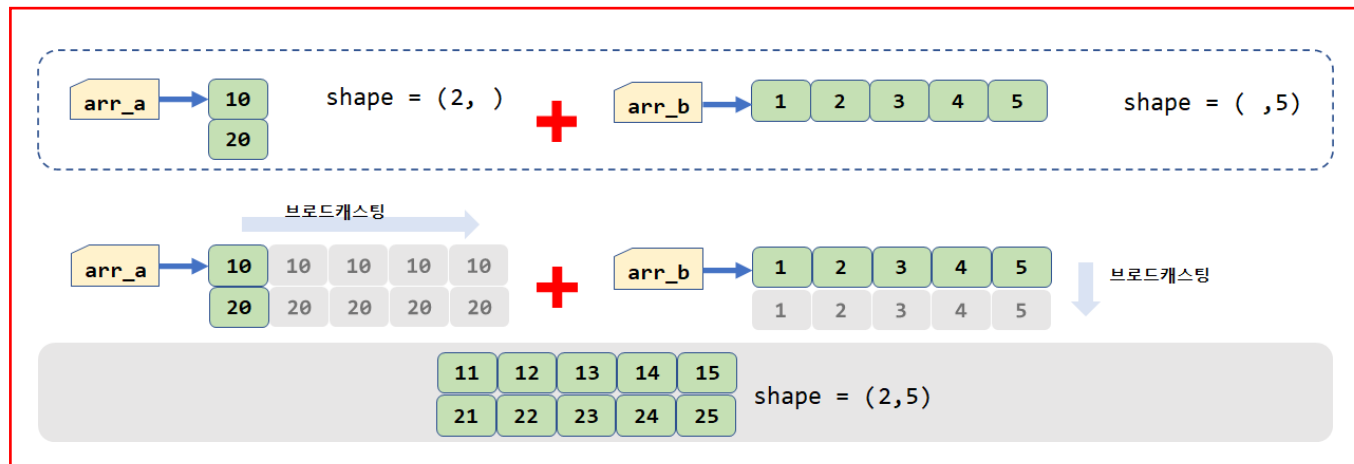
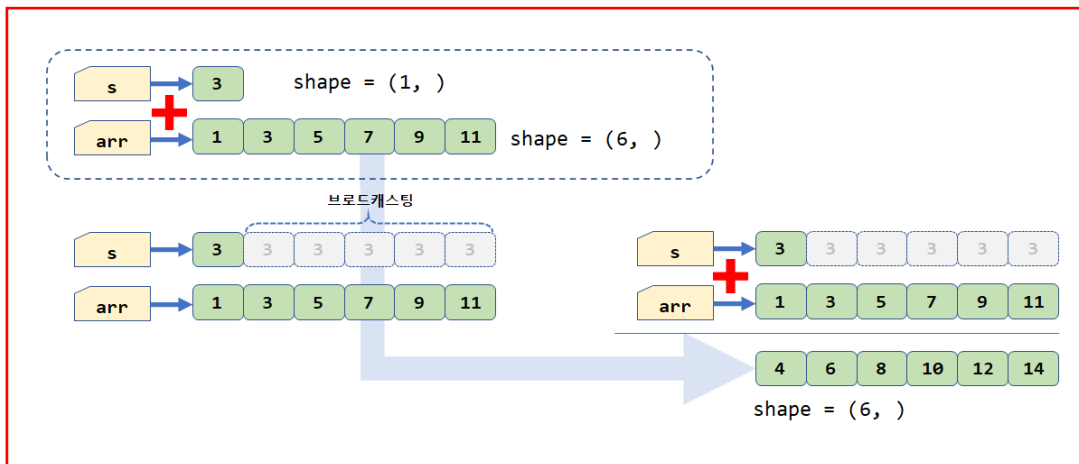
소요시간: 0.0000588893890380859

데이터 개수에 따른 성능 차이

N	10K	20K	40K	100K	200K	400K	1M	2M	3M	4M	5M
성능	46	148	126	143	150	179	187	195	177	182	206

넘파이 장점 – 브로드캐스팅

벡터화 연산은 두 피연산자 동일한 shape을 갖고 있을 때 대응되는 원소들에 대해 각각 연산이 적용된다. 그런데, **브로드캐스팅**은 이러한 벡터화 연산이 다른 shape을 가지고 있을 때에도 연산이 가능하게 만들어 준다.



브로드캐스팅에 의한 성능 향상

```
▶ import numpy as np
import time

n = 1000
arr_a = np.random.rand(n, 1)
arr_b = np.random.rand(1, n)
print(arr_a.shape, arr_b.shape)
```

```
▶ # 브로드캐스팅과 벡터화 없이 더하기
start = time.time()                # 연산 시작 시간
res_naive = np.empty((n, n))       # 결과 저장 공간
arr_a_ex = np.empty((n, n))        # 차원을 일치시킴
arr_b_ex = np.empty((n, n))        # 차원을 일치시킴
for i in range(0, n):
    for j in range(0, n):
        arr_a_ex[j,i] = arr_a[j, 0] # 확장된 공간 채움
        arr_b_ex[i,j] = arr_b[0, j] # 확장된 공간 채움

for i in range(n):
    for j in range(n):
        # 두 배열의 원소별 합
        res_naive[i, j] = arr_a_ex[i, j] + arr_b_ex[i, j]
end = time.time()                  # 연산 종료 시간
print(n, 'x', n, ": ", end-start)
```

```
→ 1000 x 1000 : 1.60617995262146
```

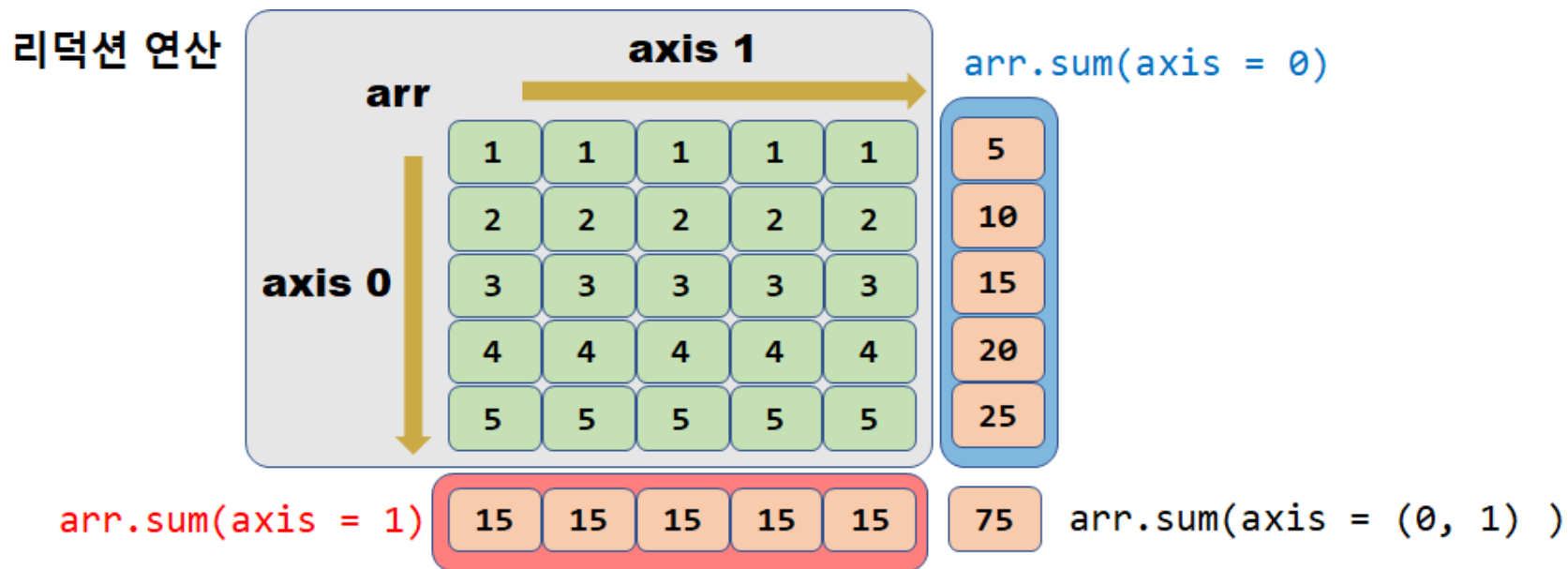


```
▶ # 브로드캐스팅과 벡터화 연산으로 더하기
start = time.time()                # 연산 시작 시간
res = arr_a + arr_b                # 브로드캐스팅과 벡터화 연산 적용
end = time.time()                  # 연산 종료 시간
print(n, 'x', n, ": ", end-start)
```

```
→ 1000 x 1000 : 0.0021255016326904297
```

넘파이 장점 – 리덕션

리덕션이라는 것은 여러 개의 수로 이루어진 배열을 하나의 수로 집계^{aggregation}하는 것이다. 넘파이는 이 집계를 위한 함수로 `sum()`, `mean()`, `min()`, `max()` 같은 함수를 제공



리덕션에 의한 성능향상

```
▶ import numpy as np
import time

n = 1000
arr = np.random.rand(n, n)
```

```
▶ sum_naive = 0
start = time.time()
for i in range(n):
    for j in range(n):
        sum_naive += arr[i,j]
end = time.time()
print(n, 'x', n, "배열의 원소 합", sum_naive, "계산 시간: ", end-start)
```

```
→ 1000 x 1000 배열의 원소 합 500048.24990126066
계산 시간: 0.39148378372192383
```



```
▶ start = time.time()
sum_reduction = np.sum(arr, axis = (0,1))
end = time.time()
print(n, 'x', n, "배열의 원소 합", sum_reduction, "계산 시간: ", end-start)
```

```
→ 1000 x 1000 배열의 원소 합 500048.2499012628
계산 시간: 0.00128173828125
```

벡터의 표현

동명대학교 게임공학과

강영민

벡터의 개념

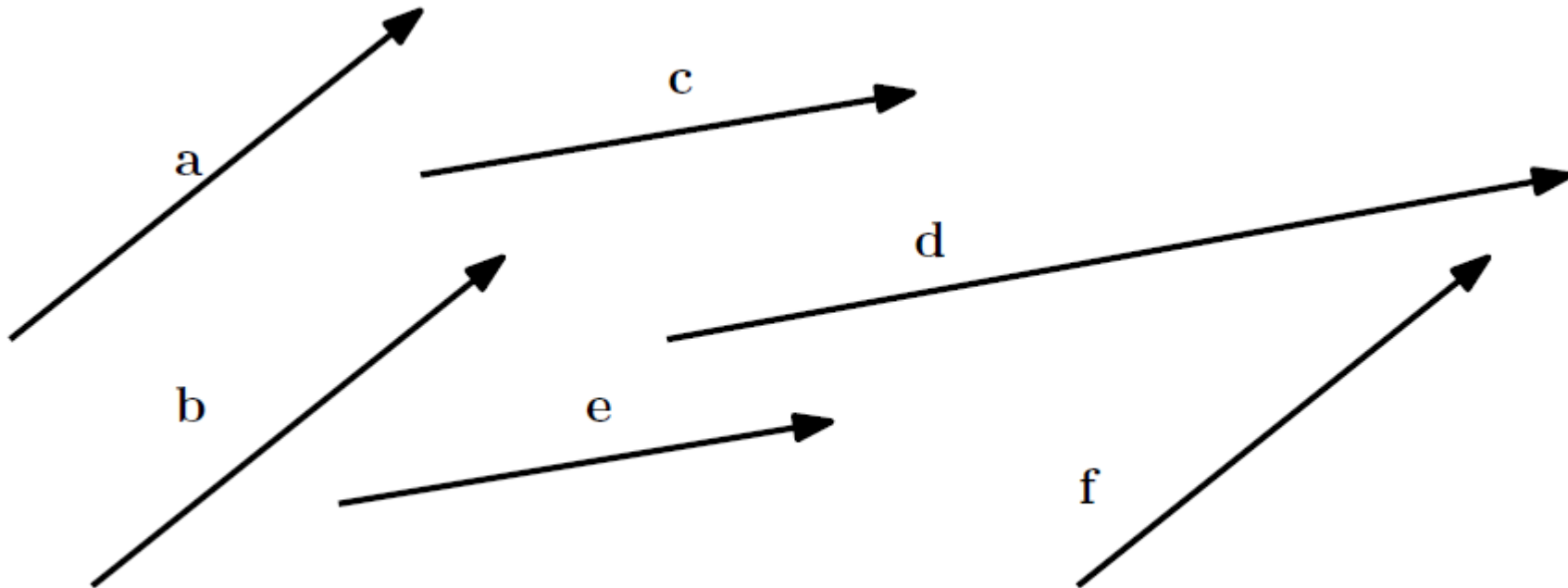
- 물리적 현상 등을 표현할 때 그 대상을 양^{quantity}로 표현
- 이 양은 스칼라 혹은 벡터
 - 스칼라: 오로지 크기만으로 완전히 그 양을 표현할 수 있음
 - 질량, 소요시간, 길이, 열량
 - 대소 비교가 가능
 - 벡터: 크기와 함께 방향도 중요한 양
 - 하나의 숫자로 표현할 수 없음
 - 힘, 속도, 변위
- 속도^{velocity}와 속력^{speed}
 - 속도: 벡터
 - 속력: 스칼라

이과가 싫어합니다



동등벡터 찾기

- 다음 중 동등벡터들을 찾아 묶어 보자

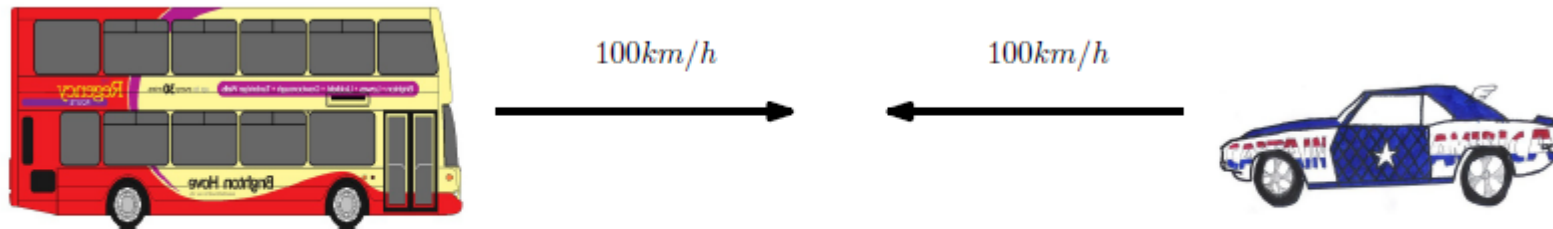


동등벡터 찾기

벡터의 표현

- 속도와 속력

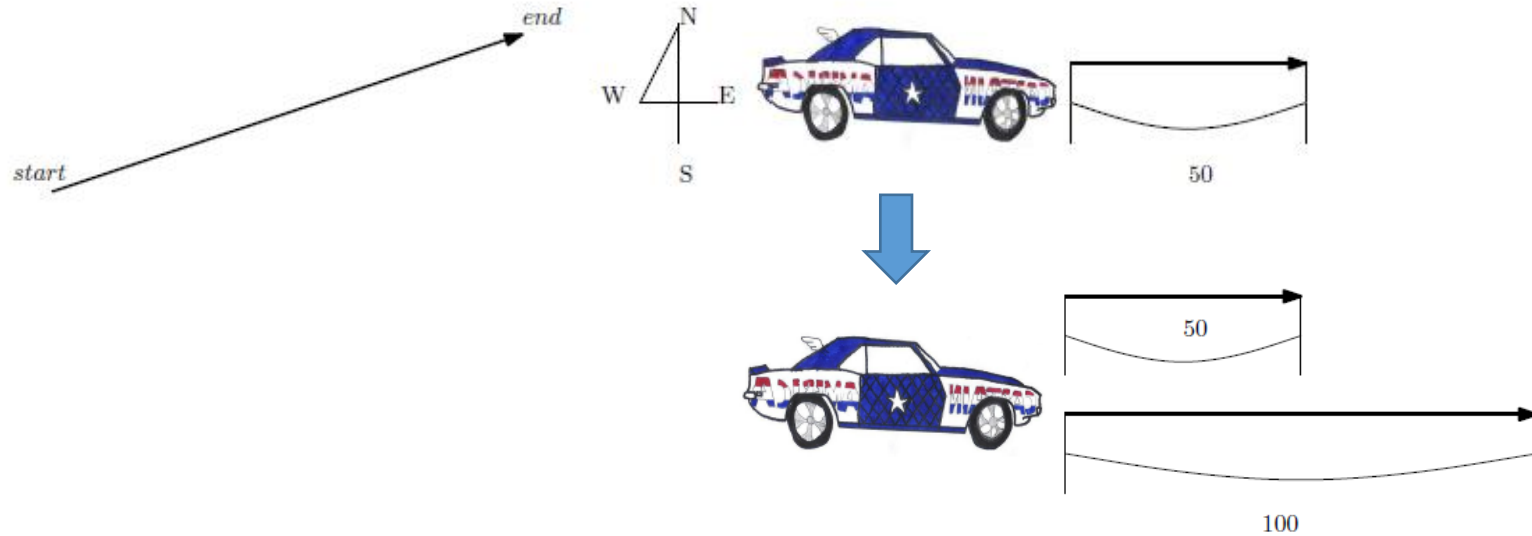
- 속력은 숫자로 표현하면 그만이지만, 속도는 숫자 하나로 표현 불가



동일한 속력으로 서로 마주 보며 달리는 차량의 속도

화살표를 이용한 벡터 표현

- 벡터를 표현하는 가장 직관적인 방법
 - 화살표: 시작하는 점과 끝나는 점으로 구성
 - 화살표의 방향은 벡터가 작용하는 방향
 - 화살표의 길이는 벡터가 가진 크기



속력이 두 배로 늘어난 자동차의 속도

동등벡터 equivalent vector

- 벡터의 표기법 \vec{a} , a

- 동등벡터

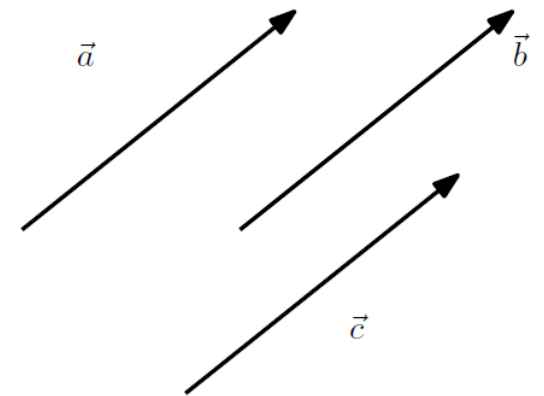
- 크기와 방향이 같은 벡터들은 모두 동등한 벡터로 간주

- 동등하다: 같은 양을 의미

- 예) 물 5kg과 쌀 5kg은 같은 대상을 가리키는 것은 아니지만, “무게 ” 라는 양의 측면에서 동등

- 두 벡터가 동등하다는 것은 두 벡터가 같은 존재라는 것이 아님

- 벡터로 표현되는 양이 같다는 의미



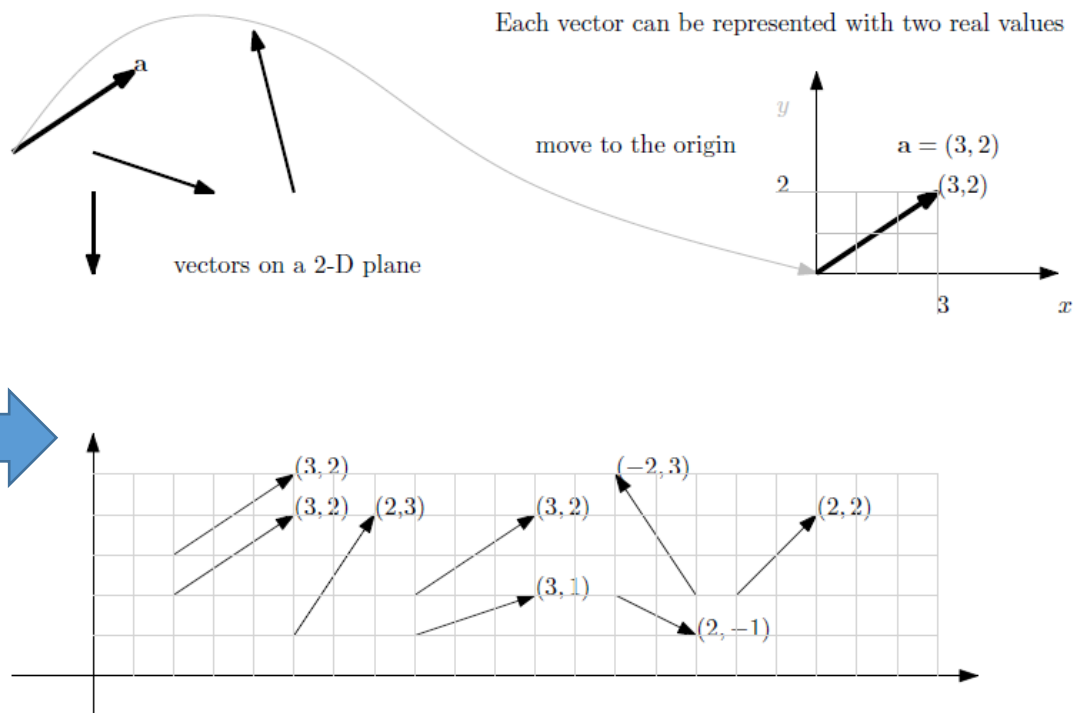
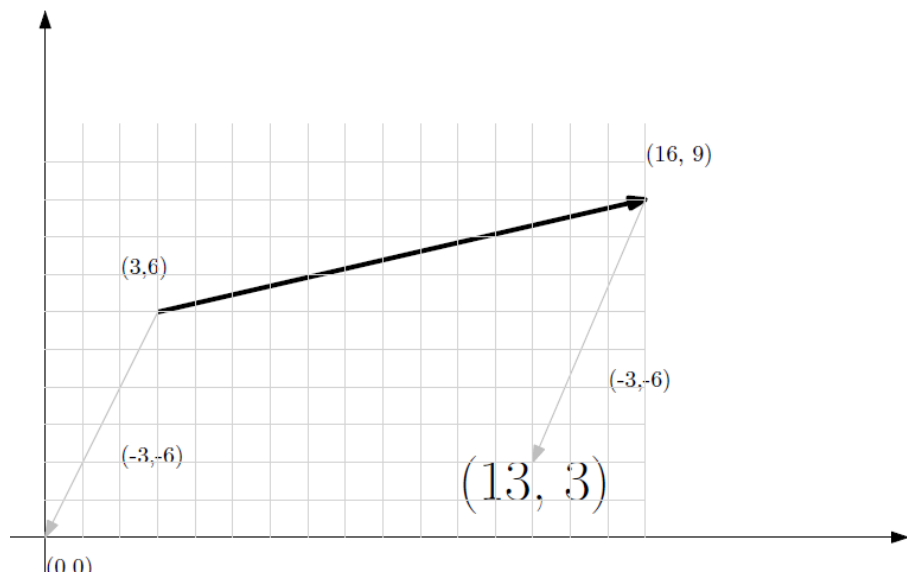
동등벡터

벡터의 수학적 표현

- 벡터를 조금 더 형식을 갖춰 표현하는 방법
 - 수학적으로 연산이 편리하도록 표현
 - 벡터의 화살표 표현
 - 화살표가 그려지는 공간의 차원에 따라 좌표의 원소 개수 결정
 - 벡터의 차원 = 공간의 차원
 - 벡터가 n 차원이라면
 - 이 벡터는 n 개의 숫자로 표현되는 좌표 형태로 다룰 수 있을 것
 - n -튜플^{tuple}
- $$\mathbf{v} = (v_1, v_2, v_3, \dots, v_n)$$
- n 개의 차원을 가진 공간에서 그려지는 화살표 = n 차원 벡터 = n -튜플

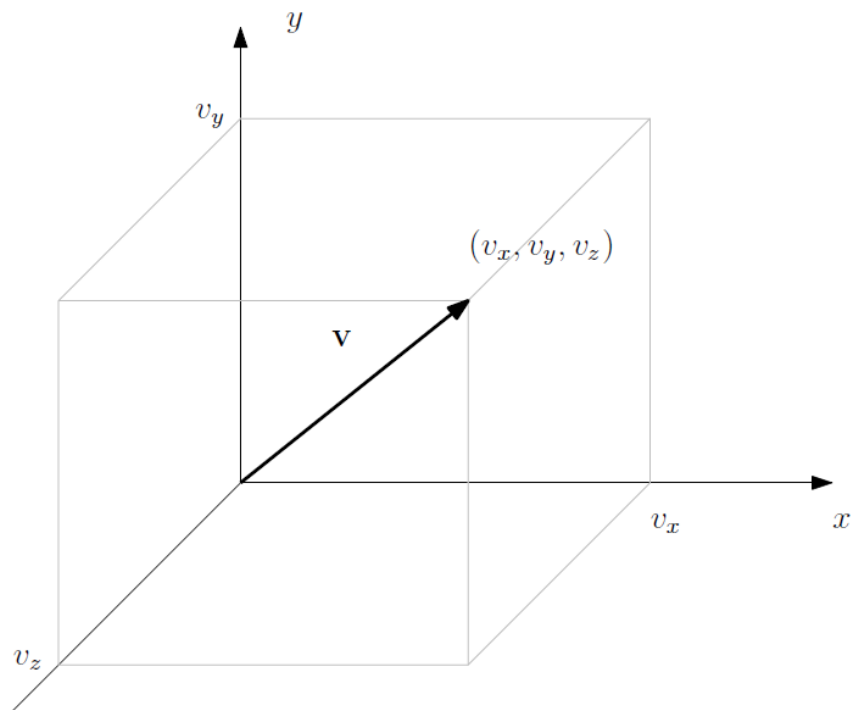
2차원 벡터의 예

- 시작점 (x_s, y_s), 끝점 (x_e, y_e)
 - 네 개의 숫자로 표현할 수 있음
 - 동등 벡터는 모두 같은 형태로 표현되어야 함
 - 이를 위해서는 시작점을 원점으로 옮기면 동등 벡터는 모두 같은 끝점을 가짐
 - 이 끝점이 이들 동등벡터를 대표하는 양

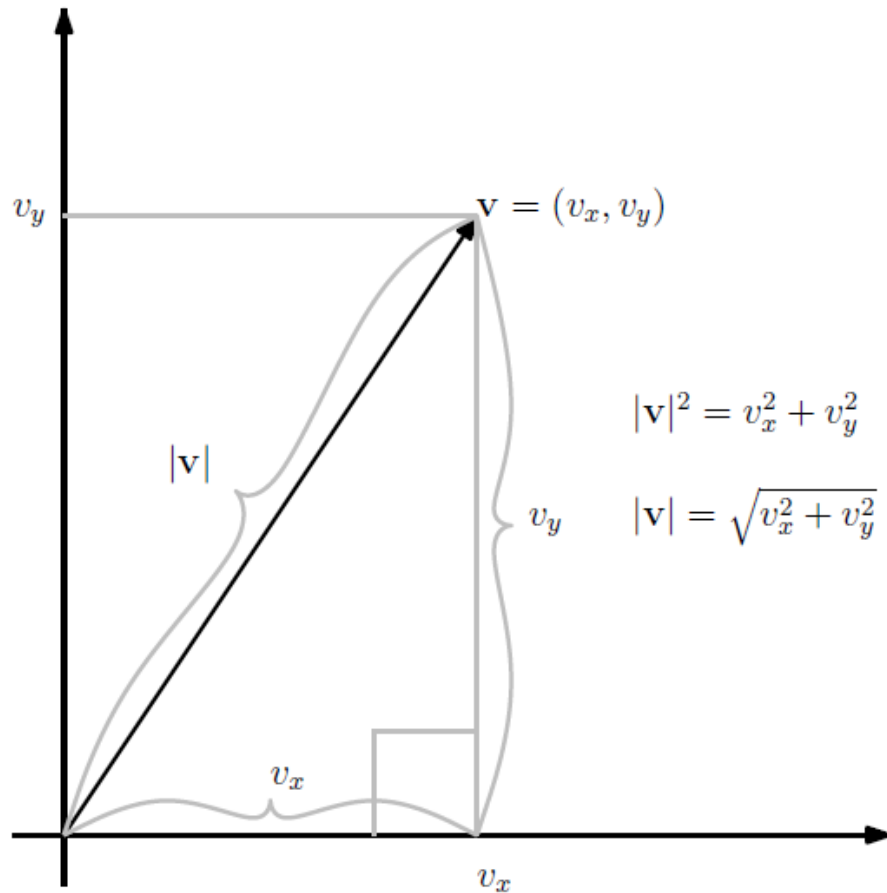


3차원 벡터

- 3차원 벡터는 2차원 벡터에 축을 하나 더하기만 하면 된다



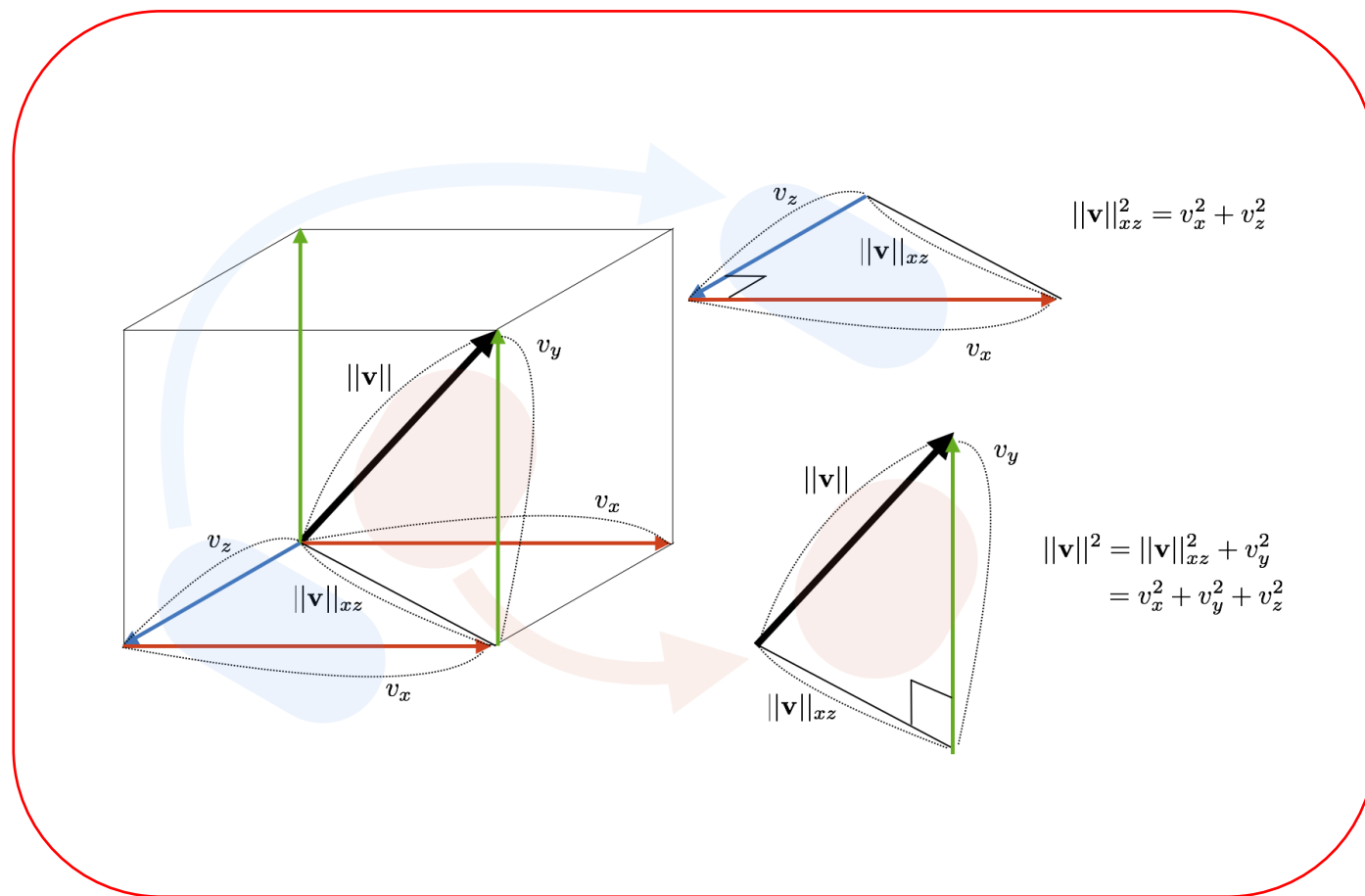
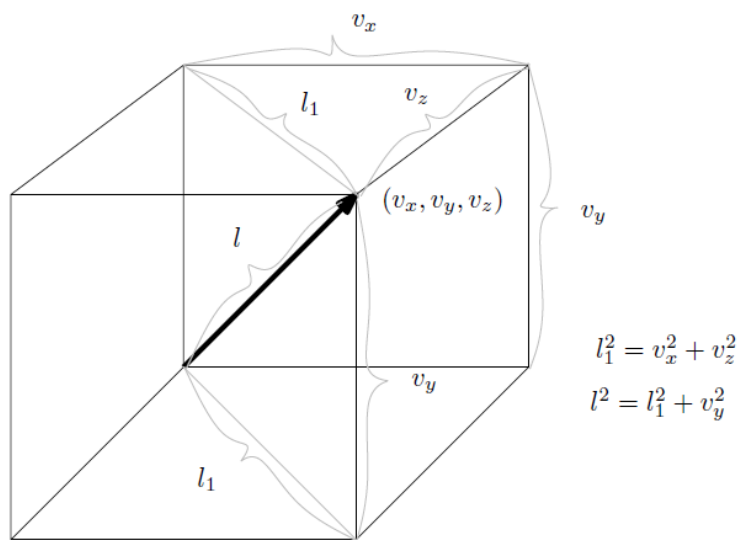
2차원 벡터의 크기



- 왼쪽에 보이는 벡터 \mathbf{v} 의 크기는?
 - 좌표값은 각 축으로의 길이
 - 2차원의 예에서
 - X좌표는 밑변
 - Y좌표는 높이
 - 벡터의 크기는 이러한 밑변과 높이를 가지는 직각삼각형의 빗변 길이
 - 피타고라스 정리로 간단히 계산
- 크기는 언제나 양의 값
- 이 값을 노름(norm, 놈)이라 함

3차원 벡터의 크기

$$\mathbf{v} = (v_x, v_y, v_z)$$
$$\|\mathbf{v}\| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$



벡터 노름 - 일반화

- 피타고라스 정리를 이용하여 구한 벡터의 노름
 - 특별한 예: 2차 노름, l_2 노름
- 노름의 일반화

$$||\mathbf{x}||_p = \left(\sum_{i=1}^n |x_i|^p \right)^{\frac{1}{p}}$$

- p 값에 따라 l_p -노름이라 부름
- p=1
 - 맨해튼 거리라고도 함
- p가 무한대
 - 원소중에 제일 큰 값을 얻음 (max)

벡터 노름 계산해 보기

```
▶ import numpy as np          # numpy의 별명을 np로 함  
  
v = np.array([1, 2, 3])      # 세 개의 원소를 가진 배열로 3차원 벡터 표현  
print(v.ndim, v.shape)      # 넘파이 배열의 ndim과 shape 출력
```

```
➡ 1 (3,)
```

```
▶ def norm(v, p=2) :  
    dim = v.shape[0]          # 원소의 개수를 담고 있음 = 벡터의 차원  
    acc = 0  
    for i in range(dim):      # 벡터의 차원만큼 모든 원소의 p 제곱 구해 누적  
        acc += v[i]**p  
    return acc**(1/p)         # 전체에 1/p 제곱을 적용하여 p-노름 계산
```

```
▶ v_norm_1 = norm(v, p=1)     # 맨해튼 노름 (L1)  
v_norm_2 = norm(v, p=2)     # 유클리드 노름 (L2)  
v_norm_3 = norm(v, p=3)     # 3-노름  
v_norm_4 = norm(v, p=4)     # 4-노름  
print(v_norm_1, v_norm_2, v_norm_3, v_norm_4)
```

```
➡ 6.0 3.7416573867739413 3.3019272488946263 3.1463462836457885
```

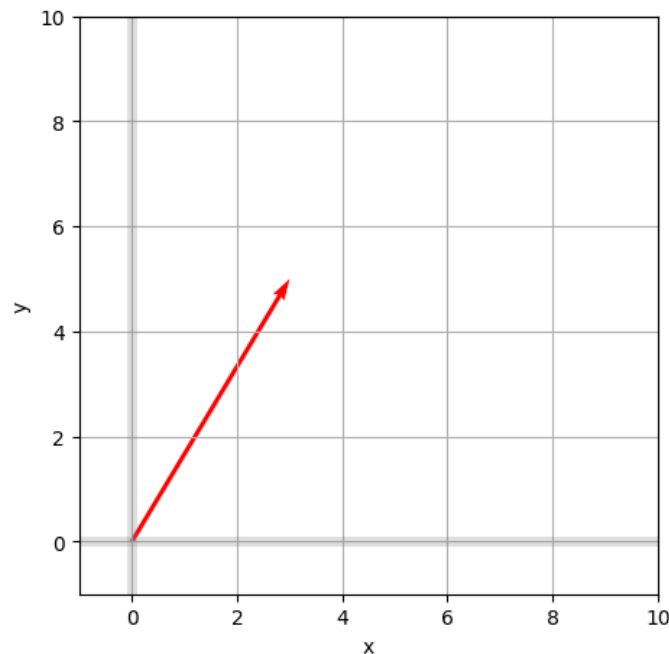
$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^n |x_i|^p \right)^{\frac{1}{p}}$$

가시화 도구

<https://github.com/dknife/linalg/blob/main/tool/visualizer.py>

```
!wget https://raw.githubusercontent.com/dknife/linalg/main/tool/visualizer.py  
from visualizer import *  
!rm visualizer.py
```

```
# (3,5) 벡터를 가시화 하자  
  
my_axis = axis2d(x=[-1,10], y=[-1,10])  
my_vector = np.array([3,5])  
draw_vec2d(my_axis, my_vector)
```



**자 이제 한 학기 동안
수학을 눈으로 살펴 봅시다.**