Section 01 람다 함수

▪ 람다 함수의 사용

- **람다(lambda) 함수 :** 함수의 이름 없이, 함수처럼 사용할 수 있는 익명의 함수를 말한다.
- 일반적으로 람다 함수는 이름을 지정하지 않아도 사용할 수 있다.

코드 9-1 function.py(일반적인 함수)

```
1 def f(x, y):
2     return x + y
3
4 print(f(1, 4))
```

```
5
```


Section 01 람다 함수

▪ 람다 함수의 사용

코드 9-2 lambda.py(람다 함수)

```
1 f = lambda x, y: x + y
2 print(f(1, 4))
```

```
5
```

➡ 위의 두 코드는 모두 입력된 x, y의 값을 더하여 그 결과를 반환하는 함수로, 결과값도 5로 같다. 하지만 람다 함수는 별도의 def나 return을 작성하지 않는다. 단지 앞에는 매개변수의 이름을, 뒤에는 매개변수가 반환하는 결과값인 'x + y'를 작성하였다. 이는 기존의 f 함수와 구조는 같고 표현이 다를 뿐이다.

Section 01 람다 함수

람다 함수를 표현하는 다른 방식

여기서 잠깐!

- 람다 함수를 표현하는 다른 방식은 다음과 같다.

```
print((lambda x: x + 1)(5))
```

- 람다 함수 자체는 위 코드처럼 이름 없이 사용할 수도 있지만, 일반적으로 [코드 9-2]와 같이 어떤 변수에 람다 함수를 할당하여 함수와 비슷한 형태로 사용한다. 위 코드는 람다 함수에 별도의 이름을 지정하지는 않았지만, 괄호에 람다 함수를 넣고 인수(argument)로 5를 입력하였다. 이 코드의 결과는 6으로 출력되는 것을 확인할 수 있다.

Section 01 람다 함수

▪ 람다 함수의 다양한 형태

```
>>> f = lambda x, y: x + y
>>> f(1, 4)
5
>>>
>>> f = lambda x: x ** 2
>>> f(3)
9
>>>
>>> f = lambda x: x / 2
>>> f(3)
1.5
>>> f(3, 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: <lambda>() takes 1 positional argument but 2 were given
```

➡ 이 코드에서 맨 아래에 있는 코드를 보면, 앞에서 매개변수를 1개로 선언했는데 f(3, 5)와 같이 2개 이상의 값이 들어오면 오류가 발생한다. 이는 함수에서 매개변수의 개수를 넘어가는 인수가 입력될 때의 결과와 같다.

Section 02 맵리듀스

▪ map( ) 함수

- **map( ) 함수 :** 연속 데이터를 저장하는 시퀀스형에서 요소마다 같은 기능을 적용할 때 사용한다. 일반적으로 리스트나 튜플처럼 요소가 있는 시퀀스 자료형에 사용된다.

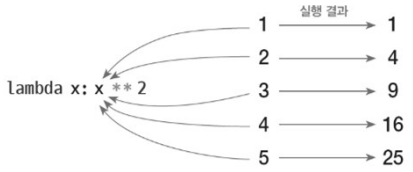
```
>>> ex = [1, 2, 3, 4, 5]
>>> f = lambda x: x ** 2
>>> print(list(map(f, ex)))
[1, 4, 9, 16, 25]
```

➡ 위 코드에서는 먼저 ex라는 이름의 리스트를 만들고, 입력된 값을 제곱하는 람다 함수 f를 생성하였다. 그리고 'map(함수 이름, 리스트 데이터)'의 구조에서 map(f, ex) 코드를 실행한다. 이는 '해당 코드로 함수 f를 ex의 각 요소에 맵핑(mapping)하라'는 뜻이다.

Section 02 맵리듀스

▪ map( ) 함수

- 위 코드는 실제로 다음과 같이 실행된다.



[map( ) 함수의 실행]

Section 02 맵리듀스

▪ map( ) 함수 : 제너레이터의 사용

- 한 가지 주의할 점은 파이썬 2.x와 3.x에서의 map( ) 함수 코드가 약간 다르다는 점이다. 파이썬 2.x에서는 map(f, ex)라고만 입력해도 리스트로 반환하지만, 3.x에서는 반드시 list(map(f, ex))처럼 list를 붙여야 리스트로 반환한다. 이것은 제너레이터(generator)라는 개념이 강화되면서 생긴 추가 코드이다.
- 제너레이터(generator)는 시퀀스 자료형의 데이터를 처리할 때, 실행 시점의 값을 생성하여 효율적으로 메모리를 관리할 수 있다는 장점이 있다.

Section 02 맵리듀스

▪ map( ) 함수 : 제너레이터의 사용

- 만약 list를 붙이지 않는다면, 다음 코드처럼 코딩할 수도 있다.

```
>>> ex = [1, 2, 3, 4, 5]
>>> f = lambda x: x ** 2
>>> for value in map(f, ex):
...     print(value)
...
1
4
9
16
25
```

Section 02 맵리듀스

▪ map( ) 함수 : 리스트 컴프리헨션과의 비교

- 최근에는 람다 함수나 map( ) 함수를 프로그램 개발에 사용하는 것을 권장하지 않는다. 굳이 두 함수를 쓰지 않더라도 리스트 컴프리헨션 기법으로 얼마든지 같은 효과를 낼 수 있기 때문이다.
- 만약 앞의 코드를 리스트 컴프리헨션으로 변경한다고 하면, 다음처럼 코딩하면 된다.

```
>>> ex = [1, 2, 3, 4, 5]
>>> [x ** 2 for x in ex]
[1, 4, 9, 16, 25]
```


Section 02 맵리듀스

▪ map( ) 함수 : 한 개 이상의 시퀀스 자료형 데이터의 처리

- map( ) 함수의 또 다른 특징은 2개 이상의 시퀀스 자료형 데이터를 처리하는 데도 문제가 없어, 여러 개의 시퀀스 자료형 데이터를 입력값으로 사용할 수 있다는 점이다. 만약 람다 함수를 작성한다면, zip( ) 함수처럼 2개의 시퀀스 자료형 데이터에서 같은 위치에 있는 값끼리 대응해 계산할 수 있다.
- 다음 코드의 경우에는 ex 변수와 같은 위치에 있는 값끼리 더한 결과가 출력된다.

```
>>> ex = [1, 2, 3, 4, 5]
>>> f = lambda x, y: x + y
>>> list(map(f, ex, ex))
[2, 4, 6, 8, 10]
```

- 위 코드를 리스트 컴프리헨션으로 변경하면, 다음과 같다.

```
>>> [x + y for x, y in zip(ex, ex) ]
[2, 4, 6, 8, 10]
```


Section 02 맵리듀스

▪ map( ) 함수 : 필터링(filtering) 기능

- map( ) 함수는 리스트 컴프리헨션처럼 필터링 기능을 사용할 수 있다. 한 가지 기억할 점은 리스트 컴프리헨션과 달리 else문을 반드시 작성해 해당 경우가 존재하지 않는 경우를 지정해주어야 한다는 점이다.
- 다음 코드에서 짝수일 때는 각 수를 제곱하고, 그렇지 않을 때는 해당 수를 그대로 출력하는 코드를 작성하였다. 비교를 위해 같은 기능의 리스트 컴프리헨션 코드를 바로 아래에 작성하였다. 실제로 리스트 컴프리헨션 코드가 약간 더 직관적이라는 사실을 알 수 있다.

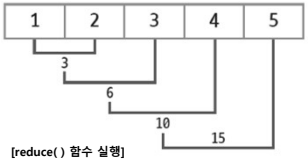
```
>>> list(map(lambda x: x ** 2 if x % 2 == 0  else x, ex))        # map() 함수
[1, 4, 3, 16, 5]
>>> [x ** 2 if x % 2 ==0 else x for x in ex]                     # 리스트 컴프리헨션
[1, 4, 3, 16, 5]
```


Section 02 맵리듀스

▪ reduce( ) 함수

- reduce( ) 함수는 map( ) 함수와 용법은 다르지만, 형제처럼 함께 사용하는 함수이다.
- reduce( ) 함수는 리스트와 같은 시퀀스 자료형에 차례대로 함수를 적용하여 모든 값을 통합하는 함수이다.

```
>>> from functools import reduce
>>> print(reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]))
15
```



[reduce( ) 함수 실행]

Section 02 맵리듀스

■ **reduce( ) 함수**

코드 9-3 reduce.py

```
1 x = 0
2 for y in [1, 2, 3, 4, 5]:
3     x += y
4
5 print(x)
```

```
15
```

➡ 이 경우에도 x의 결과값으로 15가 출력된다. 즉, 앞의 x는 기존의 x + y를 적용한 값을 계속 저장하는 변수이고, 뒤의 y는 리스트에 있는 값을 하나씩 할당받는 변수이다. 값을 차례대로 뽑는다는 것은 기존 map( ) 함수와 똑같지만, 리스트 변수의 모든 값을 람다 함수로 모두 적용한다는 차이가 있다.

Section 02 맵리듀스

람다 함수와 맵리듀스의 사용

여기서 잠깐!

- 람다 함수와 맵리듀스는 파이썬 2.x 버전에서 매우 많이 사용하던 함수이다. 최근에는 그 문법의 복잡성 때문에 권장하지 않지만, 여전히 기존 코드와 새롭게 만들어지는 코드에서는 많이 사용하고 있으므로 알아둘 필요가 있다.

Section 03 별표의 활용

■ **별표의 사용**

- 별표(asterisk)는 곱하기 기호(*)를 뜻한다. 별표는 기본 연산자로, 단순 곱셈이나 제곱 연산에 많이 사용되었다. 하지만 별표를 사용하는 특별한 경우가 있다. 바로 다음 코드와 같이 함수의 가변 인수(variable length arguments)를 사용할 때 변수명 앞에 별표를 붙인다.

가변 인수

```
>>> def asterisk_test(a, *args):
...     print(a, args)
...     print(type(args))
...
>>> asterisk_test(1, 2, 3, 4, 5, 6)
1 (2, 3, 4, 5, 6)
<class 'tuple'>
```


Section 03 별표의 활용

■ **별표의 사용**

키워드 가변 인수

```
>>> def asterisk_test(a, **kargs):
...     print(a, kargs)
...     print(type(kargs))
...
>>> asterisk_test(1, b=2, c=3, d=4, e=5, f=6)
1 {'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6}
<class 'dict'>
```

➡ 별표 한 개(*) 또는 두 개(**)를 변수명 앞에 붙여 여러 개의 변수가 함수에 한번에 들어갈 수 있도록 처리하였다. 즉, 첫 번째 함수의 경우, 2, 3, 4, 5, 6이 변수 args에 할당된 것이다. 그렇다면 별표의 어떤 특징 때문에 이것이 가능할까? 바로 여러 개의 변수를 담는 컨테이너로서의 속성을 부여하였기 때문이다

Section 03 별표의 활용

▪ 별표의 언패킹 기능

- 별표는 여러 개의 데이터를 담는 리스트, 튜플, 딕셔너리와 같은 자료형에서는 해당 데이터를 언패킹하는 기능을 한다.

```
>>> def asterisk_test(a, args):
...     print(a, *args)
...     print(type(args))
...
>>> asterisk_test(1, (2, 3, 4, 5, 6))
1 2 3 4 5 6
<class 'tuple'>
```

➡ 위 코드에서 asterisk_test 함수는 a와 args, 2개의 변수를 매개변수로 받는다. 여기서 주의할 점은 args 앞에 별표가 붙지 않았다는 점이다. 따라서 정수형인 a와 튜플형인 args가 매개변수에 입력된다. 핵심은 print(a, *args) 코드이다. 사실 args는 함수에 하나의 변수로 들어갔기 때문에 일반적이라면 다음과 같이 출력되어야 한다.

Section 03 별표의 활용

▪ 별표의 언패킹 기능

```
1 (2 3 4 5 6)
```

➡ 왜냐하면 튜플의 값은 하나의 변수이므로, 출력 시 괄호가 붙어 출력된다. 하지만 기대와 달리 1 2 3 4 5 6 형태로 출력되었다. 이는 일반적으로 print(a, b, c, d, e, f)처럼 각각의 변수를 하나씩 따로 입력했을 때 출력되는 형식이다. 이렇게 출력된 이유는 *args 때문이다. 즉, args 변수 앞에 별표가 붙어 이러한 결과가 나온 것이다. 이처럼 변수 앞의 별표는 해당변수를 언패킹한다. 즉, 하나의 튜플 (2, 3, 4, 5, 6)이 아닌 각각의 변수로 존재하는 2, 3, 4, 5, 6으로 변경된다.

Section 03 별표의 활용

▪ 별표의 언패킹 기능

```
>>> def asterisk_test(a, *args):
...     print(a, args)
...     print(type(args))
...
>>> asterisk_test(1, *(2, 3, 4, 5, 6))
1 (2, 3, 4, 5, 6)
<class 'tuple'>
```

➡ 위 코드에서 함수 호출 시 별표가 붙은 asterisk_test(1, *(2, 3, 4, 5, 6))의 형태로 값이 입력되었다. 즉, 입력값은 뒤의 튜플 변수가 언패킹되어 다음처럼 입력된 것이다.

```
>>> asterisk_test(1, 2, 3, 4, 5, 6)
```


Section 03 별표의 활용

▪ 별표의 언패킹 기능

```
>>> a, b, c = ([1, 2], [3, 4], [5, 6])
>>> print(a, b, c)
[1, 2] [3, 4] [5, 6]
>>>
>>> data = ([1, 2], [3, 4], [5, 6])
>>> print(*data)
[1, 2] [3, 4] [5, 6]
```

➡ 두 코드의 형태는 다르지만, 모두 기존의 튜플값을 언패킹하여 출력하는 결과는 같다.

Section 03 별표의 활용

▪ 별표의 언패킹 기능

- 별표의 언패킹 기능을 유용하게 사용할 수 있는 것 중 하나가 zip( ) 함수와 함께 사용할 때이다. 만약 이차원 리스트에서 행마다 한 학생의 수학·영어·국어 점수를 만들어 평균을 내고 싶다면, 2개의 for문을 사용하여 계산할 수 있다. 하지만 별표를 사용한다면 다음과 같이 하나의 for문만으로도 원하는 결과를 얻을 수 있다.

```
>>> for data in zip(*[[1, 2], [3, 4], [5, 6]]):
        print(data)
        print(type(data))

(1, 3, 5)
<class 'tuple'>
(2, 4, 6)
<class 'tuple'>
```

21/46

Section 03 별표의 활용

▪ 별표의 언패킹 기능

➡ 앞 코드에서 [[1, 2], [3, 4], [5, 6]]은 이차원 리스트로, 만약 언패킹한다면 [1, 2], [3, 4], [5, 6]으로 분리된다. 그리고 zip( ) 함수를 사용하여 같은 위치의 값을 튜플로 묶을 수 있다. 이로 인해 결과는 (1, 3, 5), (2, 4, 6)으로 나타난다. 필요에 따라 sum( ) 함수를 사용하여 각 인덱스값의 합계나 평균을 내기 유용하다.

22/46

Section 03 별표의 활용

▪ 별표의 언패킹 기능

- 키워드 가변 인수와 마찬가지 두 개의 별표(**)를 사용할 경우 딕셔너리형을 언패킹한다. 다음 코드는 딕셔너리형인 data 변수를 언패킹하여 키워드 매개변수를 사용하는 함수에 넣는 예제이다. 함수의 매개변수에 맞추어 유용하게 사용할 수 있는 코드 중 하나이다

```
>>> def asterisk_test(a, b, c, d,):
        print(a, b, c, d)

>>> data = {"b":1 , "c":2, "d":3}
>>> asterisk_test(10, **data)
10 1 2 3
```

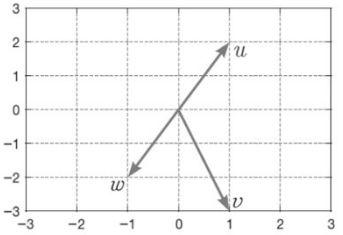
23/46

Section 04 선형대수학

▪ 벡터와 행렬의 개념 : 벡터(vector)

- 고등학교 수학에서는 어떤 정보를 표현할 때 크기와 방향을 모두 가지는 것을 벡터라고 하고, 크기만 가지는 것을 스칼라라고 하였다. 일반적으로 열 형태로 숫자를 표현하고 좌표평면에 나타낸다.

$$u = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad v = \begin{bmatrix} 1 \\ -3 \end{bmatrix} \quad w = \begin{bmatrix} -1 \\ -2 \end{bmatrix}$$



[벡터의 표현: 2개의 값]

24/46

Section 04 선형대수학

▪ 벡터와 행렬의 개념 : 벡터(vector)

- 여러 개의 정보를 표현할 때 사용하는 벡터는 앞에서 배운 리스트와 비슷하다. 여러 개의 데이터를 하나의 정보에 표현한다고 생각할 수 있다.

$$\mathbb{R}^4 \ni [1,\ 2,\ -1.0,\ 3.14] \quad \text{또는} \quad \mathbb{R}^4 \ni \begin{bmatrix} 1 \\ 2 \\ -1.0 \\ 3.14 \end{bmatrix}$$

[벡터의 표현: 3개 이상의 값]

- R은 실수real number를 뜻하며, 기호는 '[1, 2, -1.0, 3.14]는 4차원 실수 집합에 포함된다'라고 해석한다. 읽을 때는 4-벡터4-vector 또는 4차원 벡터로 읽는다.

Section 04 선형대수학

▪ 벡터와 행렬의 개념 : 행렬(matrix)

- 행렬은 원래 격자를 뜻하는데, 수학에서는 사각형으로 된 수의 배열을 지칭한다. 쉽게 말해, 1개 이상의 벡터 모임을 행렬이라고 생각하면 된다. 즉, 행렬에서 행 또는 열은 하나의 대상에 대한 정보를 표현한 것이며, 그 모음이 바로 행렬이다.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \qquad A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad \begin{aligned} a_{11} &= 1 \\ a_{23} &= 6 \\ a_{31} &= 7 \end{aligned}$$

[행렬의 표현] [행렬값의 표현]

- 행렬은 m개의 행과 n개의 열로 구성된다. 일반적으로 'm×n 행렬'이라고 표기하며 'm by n'으로 읽는다. 매트릭스의 각 요소의 값을 표시할 때는 행렬 A의 i행, j열의 값을 'A의 ij번째 값'이라고 하고, 'a_ij '로 표시한다.

Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 벡터

- 벡터를 파이썬 스타일 코드로 표현하는 방법에 대해 알아보자.

```
vector_a = [1, 2, 10]                  # 리스트로 표현한 경우
vector_b = (1, 2, 10)                  # 튜플로 표현한 경우
vector_c = {'x': 1, 'y': 1, 'z': 10}   # 딕셔너리로 표현한 경우
```

➡ 만약 각 데이터의 이름, 즉 x, y, z 와 같은 정보(ex. 키, 몸무게, 나이)를 함께 표현해야 한다면 딕셔너리로 표현하는 것도 좋은 방법이다. 데이터의 위치나 순서가 바뀌지 않아야 한다면 튜플로 저장하는 것이 좋다. 이처럼 벡터를 사용하는 목적에 따라 코드 표현은 다를 수 있다.

Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 벡터 : 벡터의 연산

- 벡터의 가장 기본적인 연산은 같은 위치에 있는 값끼리 연산하는 것이다.

$$[u_1,\ u_2,\ \cdots,\ u_n] + [v_1,\ v_2,\ \cdots,\ v_n] = [u_1 + v_1,\ u_2 + v_2,\ \cdots,\ u_n + v_n]$$

(a) 공식

$$[2,\ 2] + [2,\ 3] + [3,\ 5] = [7,\ 10]$$

(b) 예시

[벡터의 연산]

➡ 연산을 하기 위해서는 먼저 벡터의 크기가 같아야 한다. 벡터를 리스트로 생각한다면, 각 인덱스 값이 같은 위치에 있는 값끼리 연산을 하면 된다

Section 04 선형대수학

■ 파이썬 스타일 코드로 표현한 벡터 : 벡터의 연산

```
>>> u = [2, 2]
>>> v = [2, 3]
>>> z = [3, 5]
>>> result = [ ]
>>>
>>> for i in range(len(u)):
...     result.append(u[i] + v[i] + z[i])
...
>>> print(result)
[7, 10]
```

➡ 이 코드는 파이썬 특유의 간결성을 살리지 못하였다. 특히 벡터와 같은 수학 연산을 복잡하게 표현한다면, 다른 사람이 사용하기 어려울 수 있다. 위 코드는 리스트 컴프리헨션과zip( ) 함수와 같은 파이썬 스타일 코드를 최대한 사용하여 연산을 간단하게 표시할 수 있다.

29/46

Section 04 선형대수학

■ 파이썬 스타일 코드로 표현한 벡터 : 벡터의 연산

```
>>> u = [2, 2]
>>> v = [2, 3]
>>> z = [3, 5]
>>>
>>> result = [sum(t) for t in zip(u, v, z)]
>>> print (result)
[7, 10]
```

➡ 코드가 훨씬 간단해졌다. 여기서 보는 것처럼 리스트의 sum( ) 함수를 사용하여 zip( ) 함수로 묶인 튜플 t 변수의 합계를 구하였다. 변수 t에는 차례대로 (2, 2, 3), (2, 3, 5)가 들어간다. 확인을 위해 다음 코드를 수행하면 t에 어떤 값이 할당되었는지 알 수 있다

```
>>> [t for t in zip(u, v, z)]
[(2, 2, 3), (2, 3, 5)]
```

30/46

Section 04 선형대수학

■ 파이썬 스타일 코드로 표현한 벡터 : 별표를 사용한 함수화

- 만약 4개 이상의 변수일 경우에는 어떻게 해야 할까? 이때는 별표를 이용하여 다음과 같이 처리할 수 있다.

```
>>> def vector_addition(*args):
...     return [sum(t) for t in zip(*args)]
...
>>> vector_addition(u ,v, z)
[7, 10]
```

➡ 위 코드에서 함수 vector_addition을 만들고, 해당 함수에서는 *args를 사용하여 여러 개의 변수를 입력받는 가변 인수로 사용하였다. 그리고 실제 함수에서는 args에 별표(*)를 붙여 언패킹하였다. 따라서 원래 코드인 sum(t) for t in zip(u, v, z)와 같은 효과를 낼 수 있다.

31/46

Section 04 선형대수학

■ 파이썬 스타일 코드로 표현한 벡터 : 별표를 사용한 함수화

- 여전히 변수를 3개나 생성하는 문제는 어떻게 해결할 수 있을까? 이 문제는 이차원 리스트를 만든 후 별표의 언패킹으로 해결한다. 다음 코드를 보자.

```
>>> row_vectors = [[2, 2], [2, 3], [3, 5]]
>>> vector_addition(*row_vectors)
[7, 10]
```

➡ 이렇게 별표를 사용하여 훨씬 더 깔끔한 코드를 작성할 수 있다. 이러한 기법을 잘 사용하면 훨씬 더 간결한 파이썬 코드를 작성할 수 있다.

32/46

Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 벡터 : 스칼라-벡터 연산

- 간단히 숫자형 변수라고 할 수 있는 스칼라와 벡터는 곱셈 연산이 가능하며, 분배 법칙을 적용할 수 있다.

$$\alpha(u+v) = \alpha u + \alpha v$$

(a) 공식

$$2([1, 2, 3] + [4, 4, 4]) = 2(5, 6, 7) = [10, 12, 14]$$

(b) 예시

[스칼라- 벡터 연산]

Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 벡터 : 스칼라-벡터 연산

- 앞의 예시 수식을 다음과 같은 코드로 변경하자. 매우 간단하게 같은 인덱스에 있는 값들을 더한 후, 스칼라값인 alpha를 곱하면 원하는 결과가 출력된다.

```
>>> u = [1, 2, 3]
>>> v = [4, 4, 4]
>>> alpha = 2
>>>
>>> result = [alpha * sum(t) for t in zip(u, v)]
>>> result
[10, 12, 14]
```


Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 행렬

- 행렬도 벡터와 마찬가지로 리스트, 튜플, 딕셔너리 등을 사용하여 파이썬 스타일 코드로 표현할 수 있다. 행렬을 파이썬으로 표현하는 방법은 다음과 같다.

```
matrix_a = [[3, 6], [4, 5]]                    # 리스트로 표현한 경우
matrix_b = [(3, 6), (4, 5)]                    # 튜플로 표현한 경우
matrix_c = {(0 ,0): 3, (0 ,1): 6, (1 ,0): 4, (1 ,1): 5}    # 딕셔너리로 표현한 경우
```

- 이차원의 정보를 행렬이라고 한다. 이 행렬은 일차원의 벡터 정보를 모아 이차원 형태로 표현한 것으로, 다음과 같이 벡터의 정보를 모아 표현할 수 있다.

```
[[1번째 행], [2번째 행], [3번째 행]]
```


Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 행렬 : 행렬의 연산

- 행렬의 가장 기본적인 연산은 덧셈과 뺄셈이다. 2개 이상의 행렬을 연산하기 위해 각 행렬의 크기는 같아야 한다. 즉, A가 '2×2 행렬'이라면 B도 같은 '2×2 행렬'이어야 한다. 다음으로 인덱스가 같은 값끼리 연산이 일어난다.

$$A = \begin{bmatrix} 3 & 6 \\ 4 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 5 & 8 \\ 6 & 7 \end{bmatrix}$$

$$C = A + B = \begin{bmatrix} 3 & 6 \\ 4 & 5 \end{bmatrix} + \begin{bmatrix} 5 & 8 \\ 6 & 7 \end{bmatrix} = \begin{bmatrix} 8 & 14 \\ 10 & 12 \end{bmatrix}$$

[행렬의 연산]

Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 행렬 : 행렬의 연산

- 행렬의 연산을 파이썬 스타일 코드를 사용하여 표현하려면 어떻게 작성할까? 가장 쉬운 표현 방법은 별표(*)와 zip( ) 함수를 활용하는 것이다

```
>>> matrix_a = [[3, 6], [4, 5]]
>>> matrix_b = [[5, 8], [6, 7]]
>>> result = [[sum(row) for row in zip(*t)] for t in zip(matrix_a, matrix_b)]
>>>
>>> print(result)
[[8, 14], [10, 12]]
```

➡ 이 코드의 핵심은 'zip 함수( )를 어떻게 활용하는가'이다. 일단 리스트 컴프리헨션 안에 2개의 for문이 있다. 이 경우 바깥에 있는 for문이 먼저 실행되어 matrix_a와 matrix_b에서 zip ( ) 함수를 통해 같은 인덱스에 있는 값들이 추출된다. 즉, [3, 6]과 [5, 8]이 튜플로 묶여 ([3, 6], [5, 8])로 추출된다.

Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 행렬 : 행렬의 연산

- 확인을 위해 다음 코드를 실행해 보자.

```
>>> [t for t in zip(matrix_a, matrix_b)]
[([3, 6], [5, 8]), ([4, 5], [6, 7])]
```

➡ 이제 t는 2개의 리스트값을 가진 하나의 튜플로, 안쪽에 있는 리스트 컴프리헨션 구문에 들어간다. t는 1개의 튜플이므로 zip( ) 함수를 사용하기 위해 값을 언패킹해야 한다. 따라서 [sum(row) for row in zip(*t)]와 같이 언패킹한 상태로 zip( ) 함수를 사용하면, ([3, 6], [5, 8]) 의 값에서 같은 인덱스에 있는 값들이 추출되어 (3, 5), (6, 8)로 row 변수에 할당된다. 즉, [sum(row) for row in zip(*t)] 코드에서 같은 위치에 있는 값끼리 묶여 row라는 이름의 튜플이 생성된 후, sum( ) 함수가 적용된다. 이로 인하여 같은 위치의 값끼리 더해져 [[8, 14], [10, 12]]라는 원하던 결과가 나온다.

Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 행렬 : 행렬의 동치

- 행렬의 동치는 2개의 행렬이 서로 같은지를 나타내는 표현으로, 만약 행렬이 같다면 2개의 행렬이 동치라고 말한다. 다음은 두 행렬이 'A = B'이기 위한 조건을 나타낸 것이다.

$$A = \begin{bmatrix} 3 & 6 \\ 4 & 5 \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$b_{11} = 3,\ b_{12} = 6,\ b_{21} = 4,\ b_{22} = 5$$

[행렬의 동치]

Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 행렬 : 행렬의 동치

- 두 행렬이 동치임을 확인하는 코드는 어떻게 작성할까? '행렬의 연산'과 비슷한 코드를 작성하되, 불린형을 활용한다. 다음 코드를 보자.

```
>>> matrix_a = [[1, 1], [1, 1]]
>>> matrix_b = [[1, 1], [1, 1]]
>>> all([row[0] == value for t in zip(matrix_a, matrix_b) for row in zip(*t) for
value in row])
True
>>> matrix_b = [[5, 8], [6, 7]]
>>> all([all([row[0] == value for value in row]) for t in zip(matrix_a, matrix_b)
for row in zip(*t)])
False
```

➡ 코드의 핵심은 각 요소의 값을 비교하는 row[0] == value 코드이다.

Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 행렬 : 행렬의 동치

- all( ) 함수와 any( ) 함수는 리스트형과 튜플형에서 내부 값이 and 조건이나 or 조건으로 참인지 거짓인지를 반환하는 함수이다. 다음 코드를 보자.

```
>>> any([False, False, False])
False
>>> any([False, True, False])
True
>>> all([False, True, True])
False
>>> all([True, True, True])
True
```

➡ all( ) 함수는 안에 있는 모든 값이 참일 경우에만 True를 반환한다. 즉, and 조건으로 리스트의 값들이 모두 같은지를 확인한다. 반대로 any( ) 함수는 하나라도 참이 있으면 True를 반환하고, 모두가 거짓일 때만 False를 반환한다. 즉, or 조건으로 리스트에 있는 값들을 확인한다.

41/46

Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 행렬 : 행렬의 동치

- 행렬의 동치를 확인하는 코드로 돌아가자. 먼저 모든 값을 일차원 리스트로 만드는데, 이는 각 인덱스의 값이 같은지 다른지만 확인하므로 이차원 리스트로 구성할 필요가 없기 때문이다. 다음으로 마지막에 같은 인덱스에 있는 값들을 row라는 튜플에 할당한 후, 마지막 for문인 for value in row를 구성하여 row 안의 값을 다시 value에 할당한다.
- 여기서 각 인덱스의 값과 첫 번째 값, 즉 row[0]이 같은지 확인한다. 모두 같다면 all( ) 함수에 의해 해당 인덱스의 동일 여부가 True로 반환될 것이다. 그리고 모든 인덱스의 값이 같은지 확인하여 True 또는 False로 반환한 후, 마지막으로 all( ) 함수로 동치 여부를 확인한다. 만약 all( ) 함수가중간에 없다면, 다음과 같은 결과가 출력된다.

```
>>> [[row[0] == value for value in row] for t in zip(matrix_a, matrix_b) for row in zip(*t)]
[[True, False], [True, False], [True, False], [True, False]]
```

➡ 위 코드는 행렬의 개수가 늘어나더라도 문제없이 실행된다. 다양한 실험을 통해 코드가 어떻게 구성되어 있는지 확인한다.

42/46

Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 행렬 : 전치행렬(transpose matrix)

- 전치행렬은 주어진 m × n의 행렬에서 행과 열을 바꾸어 만든 행렬이다. 수식의 표현은 비교적 단순하다.

$$A=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \Rightarrow A^T=\begin{bmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & & \ddots & \vdots \\ a_{1m} & a_{2m} & \cdots & a_{nm} \end{bmatrix}$$

(a) 공식

$$A=\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \Rightarrow A^T=\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

(b) 예시

[전치행렬]

43/46

Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 행렬 : 전치행렬(transpose matrix)

- 전치행렬을 구현하기 위해 행과 열의 값을 변경해야 한다. 구현은 간단하다. 기본 행렬 리스트에서 별표와 zip( ) 함수로 각 행의 같은 위치의 인덱스값을 추출한 후, 이 값으로 리스트를 새롭게 구성하면 된다.

```
>>> matrix_a = [[1, 2, 3], [4, 5, 6]]
>>> result = [[element for element in t] for t in zip(*matrix_a)]
>>> result
[[1, 4], [2, 5], [3, 6]]
```

➡ 위 코드의 핵심은 for t in zip(*matrix_a)이다. 별표 때문에 리스트를 다음과 같이 언패킹한다.

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

44/46

Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 행렬 : 전치행렬(transpose matrix)

➡ 이렇게 언패킹한 상태에서 zip( ) 함수를 사용하면, 같은 위치의 값들을 t로 할당할 수 있다. 즉 [1, 4], [2, 5], [3, 6]이 묶인다. 이 값들이 그대로 리스트로 들어가면 전치행렬이 완성된다. 실제 코드를 보면 다음과 같이 언패킹되어 출력되는 것을 확인할 수 있다.

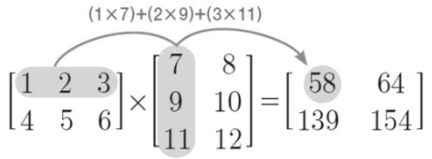
```
>>> [t for t in zip(*matrix_a)]
[(1, 4), (2, 5), (3, 6)]
```

45/46

Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 행렬 : 행렬의 곱셈

- 행렬의 곱셈은 앞 행렬의 열과 뒤 행렬의 행을 선형 결합하면 된다. 다음과 같이 대응되는 값들이 곱셈한다고 생각하면 된다.



[행렬의 곱셈]

46/46

Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 행렬 : 행렬의 곱셈

- 앞 행렬의 열과 뒤 행렬의 행의 크기가 같아야 한다. 행렬의 곱셈을 위한 조건을 만족하여야 연산이 된다.

[행렬의 곱셈을 하기 위한 조건]

47/46

Section 04 선형대수학

▪ 파이썬 스타일 코드로 표현한 행렬 : 행렬의 곱셈

- 코드로 구현하기 위해서는 전치행렬의 코드 기법을 사용하여 한 행렬에서는 열의 값을, 다른 행렬에서는 행의 값을 추출하여 곱하는 코드로 구성해야 한다.

```
>>> matrix_a = [[1, 1, 2], [2, 1, 1]]
>>> matrix_b = [[1, 1], [2, 1], [1, 3]]
>>> result = [[sum(a * b for a, b in zip(row_a, column_b)) for column_b in
zip(*matrix_b)] for row_a in matrix_a]
>>> result
[[5, 8], [5, 6]]
```

➡ 뒤에 있는 for row_a in matrix_a에서 행의 값이 뽑히고, for column_b in zip(*matrix_b)에서는 열의 값이 뽑힌다. 이는 앞의 전치행렬에서 사용한 코드와 비슷하다. 다음으로 sum(a * b for a, b in zip(row_a, column_b) 코드를 통해 행과 열에서 같은 위치의 인덱스값을 뽑은 다음, sum( ) 함수를 사용하여 합을 구하면 결과값이 출력된다.

48/46