

데이터 과학을 위한 파이썬 프로그래밍

2판



Chapter 09

파이썬 스타일 코드 II



목차

1. 람다 함수
2. 맵리듀스
3. 별표의 활용
4. 선형대수학

학습목표

- 람다 함수를 사용하는 방식과 다양한 형태에 대해 알아본다.
- `map()` 함수와 `reduce()` 함수에 대해 학습한다.
- 별표를 사용하는 방법과 별표의 언패킹 기능에 대해 이해한다.
- 파이썬 스타일 코드로 벡터와 행렬을 표현한다

01 람다 함수

1. 람다 함수의 사용

- 람다(lambda) 함수: 함수의 이름 없이 함수처럼 사용할 수 있는 익명의 함수
- [코드 9-1]과 [코드 9-2] 비교

[코드 9-1]

function.py(일반적인 함수)

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

[실행결과]

5

1. 람다 함수의 사용

[코드 9-2]

lambda.py(람다 함수)


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

[실행결과]

5

- 두 코드 모두 입력된 x, y 의 값을 더하여 그 결과를 반환하는 함수로 결과 값도 5로 같지만, 람다 함수는 별도의 `def`나 `return`을 입력하지 않음
 - 람다 함수는 앞에는 매개변수의 이름을, 뒤에는 매개변수가 반환하는 결과값인 ' $x + y$ '를 입력함
- ☞ 기존의 `f` 함수와 구조는 같고 표현이 다름.

1. 람다 함수의 사용

여기서  잠깐! 람다 함수를 표현하는 다른 방식

람다 함수를 표현하는 또 다른 방식이 있다. 다음을 보자.

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

람다 함수 자체는 위 코드처럼 이름 없이 사용할 수 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 출력된다.

2. 람다 함수의 다양한 형태

- 람다 함수는 다양한 형태로 사용할 수 있는데 기본적으로 함수를 선언하는 것과 완전히 같은 방식임.

```
>>> 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
```

02 맵리듀스

1. map() 함수

- 맵리듀스(MapReduce): 파이썬뿐만 아니라 빅데이터를 처리하기 위한 기본 알고리즘으로 많이 사용함.
- 맵리듀스의 종류: map() 함수, reduce() 함수
- map() 함수: 연속 데이터를 저장하는 시퀀스 자료형에서 요소마다 같은 기능을 적용할 때 주로 사용함.

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

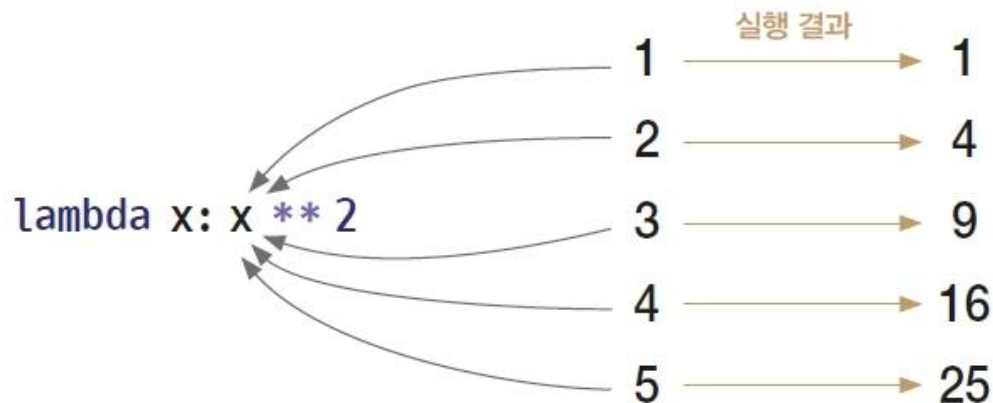


그림 9-1 map() 함수의 실행

1. map() 함수

1.1 제너레이터의 사용

- 파이썬 2.x와 3.x에서의 map() 함수 코드가 약간 다르다는 점 주의!

파이썬 2.x	map(f, ex)라고만 입력해도 리스트로 반환
파이썬 3.x	list(map(f, ex))처럼 list를 붙여야 리스트로 반환

☞ 제너레이터(generator)라는 개념이 강화되면서 생긴 추가 코드.

- **제너레이터:** 시퀀스 자료형의 데이터를 처리할 때 실행 시점의 값을 생성하여 효율적으로 메모리를 관리할 수 있다는 장점이 있음.

1. map() 함수

- list를 붙이지 않는다면 다음 코드처럼 코딩하는 것이 좋음.

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

1. map() 함수

1.2 리스트 컴프리헨션과의 비교

- 최근에는 람다 함수나 map() 함수를 프로그램 개발에 사용하는 것을 권장하지 않음.

☞ 리스트 컴프리헨션 기법으로 얼마든지 같은 효과를 낼 수 있기 때문

- 리스트 컴프리헨션으로 변경한 코드

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

1. map() 함수

1.3 한 개 이상의 시퀀스 자료형 데이터의 처리

- map() 함수는 2개 이상의 시퀀스 자료형 데이터를 처리하는 데도 문제가 없어, 여러 개의 시퀀스 자료형 데이터를 입력 값으로 사용할 수 있음
- 다음 코드를 보면 ex 변수와 같은 위치에 있는 값끼리 더한 결과가 출력됨.

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

- 리스트 컴프리헨션으로 변경한 코드

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

1. map() 함수

1.4 필터링(filtering) 기능

- map() 함수는 리스트 컴프리헨션처럼 필터링 기능을 사용할 수 있음
- 리스트 컴프리헨션과 달리 else문을 반드시 작성해 해당 경우가 존재하지 않는 경우를 지정해 주어야 함.
- 짝수일 때는 각 수를 제공하고, 그렇지 않을 때는 해당 수를 그대로 출력하는 코드 작성.

```
>>> 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]
```


2. reduce() 함수

- **reduce() 함수**: 리스트와 같은 시퀀스 자료형에 차례대로 함수를 적용한 다음 모든 값을 통합시켜 주는 함수
- map() 함수와 용법은 다르지만 형제처럼 함께 사용하는 함수임.

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

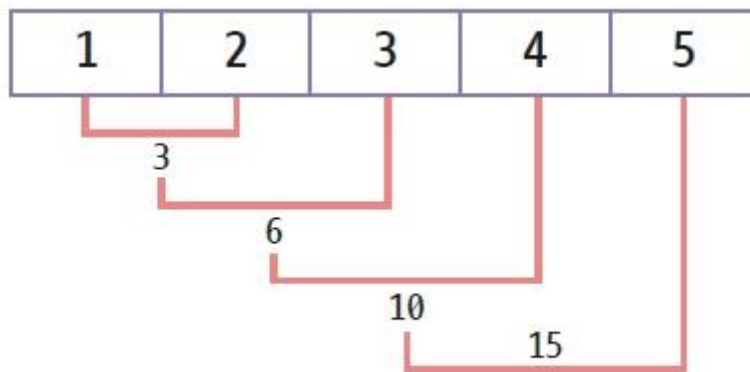


그림 9-2 reduce() 함수 실행


2. reduce() 함수

[코드 9-3]

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

[실행결과]

15

여기서  잠깐! 람다 함수와 맵리듀스의 사용

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

03

별표의 활용

1. 별표의 사용

- **별표(asterisk):** 곱하기 기호(*)를 뜻함.
- 별표는 기본 연산자로, 단순 곱셈이나 제곱 연산에 많이 사용됨.
- 별표를 사용하는 특별한 경우: 함수의 가변 인수(variable length arguments)를 사용할 때 변수명 앞에 별표를 붙임.

- 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'>
```

1. 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 한 번에 들어갈 수 있도록 처리함.
- 첫 번째 함수의 경우, 2, 3, 4, 5, 6이 변수 args에 할당됨.

2. 별표의 언패킹 기능

- 별표는 여러 개의 데이터를 담은 리스트, 튜플, 딕셔너리와 같은 자료형에서는 해당 데이터를 언패킹(unpacking)하는 기능도 있음.

```
>>> 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'>
```

2. 별표의 언패킹 기능

- 핵심은 `print(a, *args)` 코드.

- `args`는 함수에 하나의 변수로 들어갔기 때문에 일반적이라면 다음과 같이 출력되어야 함.

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

- BUT! 튜플의 값은 하나의 변수이므로 출력 시 괄호가 붙어 출력되지만 결과는 1 2 3 4 5 6 형태로 출력됨.

- 이것은 일반적으로 `print(a, b, c, d, e, f)`처럼 각각의 변수를 하나씩 따로 입력했을 때 출력되는 형식인데 이렇게 출력된 이유는 `args` 변수 앞에 별표가 붙었기 때문.

2. 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'>
```

- 함수 호출 시 별표가 붙은 `asterisk_test(1, *(2, 3, 4, 5, 6))`의 형태로 값이 입력됨. 즉, 입력 값은 뒤의 튜플 변수가 언패킹되어 다음처럼 입력된 것.

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


2. 별표의 언패킹 기능

- 두 코드의 형태는 다르지만 모두 기존의 튜플값을 언패킹하여 출력하는 것으로 결과는 같음.

```
>>> 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]
```

- 별표의 언패킹 기능을 유용하게 사용하는 경우 중 하나가 zip() 함수와 함께 사용할 때임.

2. 별표의 언패킹 기능

- 만약 이차원 리스트에서 행(row)마다 한 학생의 수학·영어·국어 점수가 있고 이것에 대해 평균을 내고 싶다면, 2개의 for문을 사용하여 계산할 수 있지만 별표를 사용한다면 다음과 같이 하나의 for문으로도 원하는 결과를 얻을 수 있음.

```
>>> 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'>
```

2. 별표의 언패킹 기능

- 키워드 가변 인수와 마찬가지로 두 개의 별표(**)를 사용할 경우 딕셔너리형을 언패킹함.
- 딕셔너리형인 data 변수를 언패킹하여 키워드 매개변수를 사용하는 함수에 넣는 예제

```
>>> 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
```

04

선형대수학

1. 벡터와 행렬의 개념

1.1 벡터(vector)

- 벡터는 '배달하다, 운반하다'의 뜻을 가진 라틴어에서 유래된 용어로, 고등학교 수학에서 어떤 정보를 표현할 때 크기와 방향을 모두 가지는 것을 벡터라고 하고, 크기만 가지는 것을 스칼라라고 부름
- 일반적으로 열 형태로 숫자를 표현하고 좌표평면에 나타냄.

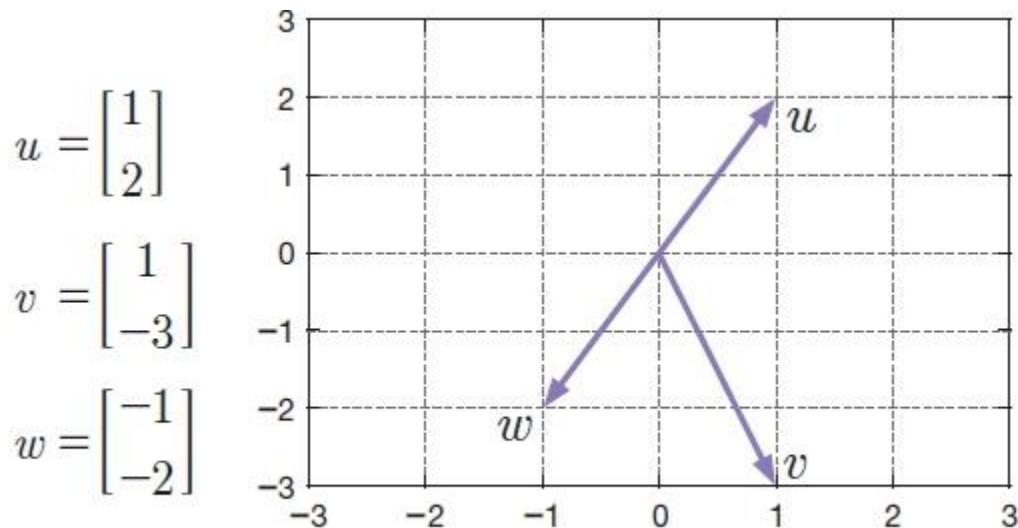


그림 9-3 벡터의 표현: 2개의 값

1. 벡터와 행렬의 개념

- 컴퓨터공학과에서의 벡터는 여러 개의 데이터를 하나의 정보로 표현한다는 관점에서 리스트와 비슷함.
- 수학에서의 벡터는 이차원 평면상에 정보를 나타내므로 값이 2개임.
- 벡터가 어떤 정보를 표현하는 방법이라는 관점에서 볼 때 3개 이상의 정보를 사용해야 하는 경우도 많음

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

그림 9-4 벡터의 표현: 3개 이상의 값

1. 벡터와 행렬의 개념

1.2 행렬(matrix)

- 행렬: 원래 격자를 뜻하는 말로, 수학에서는 사각형으로 된 수의 배열을 지칭함. 1개 이상의 벡터 모임
- 행렬에서 행 또는 열이 하나의 대상에 대한 정보를 표현한 것이며, 그 모임이 바로 행렬임.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

그림 9-5 행렬의 표현

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

그림 9-6 행렬값의 표현

$$a_{11} = 1$$

$$a_{23} = 6$$

$$a_{31} = 7$$

- 행렬의 구성: m개의 행과 n개의 열로 구성

2. 파이썬 스타일 코드로 표현한 벡터

- 벡터를 파이썬으로 표현하는 방법

```
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
```

- 가장 기본적인 방법은 리스트 형태로 표현하는 것. 튜플이나 딕셔너리 형태도 가능.
- 만약 각 데이터의 이름, 즉 x, y, z와 같은 정보(ex 키, 몸무게, 나이)를 함께 표현해야 한다면 딕셔너리로 표현하는 것도 좋은 방법임.
- 데이터의 위치나 순서가 바뀌지 않아야 한다면 튜플로 저장하는 것이 좋음.
- 벡터를 사용하는 목적에 따라 코드 표현은 다를 수 있으며 여기서는 기본적으로 리스트를 사용해 벡터의 연산을 실행함.

2. 파이썬 스타일 코드로 표현한 벡터

2.1 벡터의 연산

- 벡터의 가장 기본적인 연산: 같은 위치에 있는 값끼리 연산하는 것

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

(a) 공식

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

(b) 예시

그림 9-7 벡터의 연산

- [그림 9-7]의 수식을 코드로 작성

```
>>> 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]
```

2. 파이썬 스타일 코드로 표현한 벡터

- 리스트 컴프리헨션 과 zip() 함수와 같은 파이썬 스타일 코드를 이용해 간단한 연산으로 나타낸 코드

```
>>> 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 간단해짐. sum() 함수를 사용하여 zip() 함수로 묶인 튜플 t 변수의 합계를 구함. 변수 t에는 차례대로 (2, 2, 3), (2, 3, 5)가 들어감.
- 확인을 위해 다음 코드를 수행하면 t에 어떤 값이 할당되었는지 알 수 있음.

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

2. 파이썬 스타일 코드로 표현한 벡터

2.2 별표를 사용한 함수화

- 4개 이상의 변수를 사용해야 할 경우에는 어떻게 할까?

👉 별표를 이용하여 다음과 같이 처리할 수 있음.

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

- 여전히 변수를 3개나 생성하는 문제는 어떻게 해결할 수 있을까?

👉 이차원 리스트를 만든 후 별표의 언패킹으로 해결.

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

2. 파이썬 스타일 코드로 표현한 벡터

2.3 스칼라 - 벡터 연산

- 스칼라, 벡터: 숫자형 변수, 곱셈 연산이 가능하며 분배 법칙 적용 가능.

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

(a) 공식

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

(b) 예시

그림 9-8 스칼라-벡터 연산

- [그림 9-8]의 수식을 코드로 작성

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

3. 파이썬 스타일 코드로 표현한 행렬

- 행렬도 벡터와 마찬가지로 리스트, 튜플, 딕셔너리 등을 사용하여 파이썬 스타일 코드로 표현 할 수 있음.
- 행렬을 파이썬 스타일 코드로 표현하기

```
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
```

3. 파이썬 스타일 코드로 표현한 행렬

- **행렬**: 이차원의 정보, 일차원의 벡터 정보를 모아 이차원 형태로 표현한 것
- 벡터의 정보 모아 표현하기

```
[[1번째 행], [2번째 행], [3번째 행]]
```

- 행렬은 딕셔너리로 표현할 때 많은 경우의 수를 나타냄.
- 행과 열의 좌표 정보를 넣을 수 있고, 이름 정보를 넣을 수도 있음.
- 여기서는 가장 일반적인 표현법인 리스트를 사용함.

3. 파이썬 스타일 코드로 표현한 행렬

3.1 행렬의 연산

- 행렬의 가장 기본적인 연산: 덧셈과 뺄셈
- 2개 이상의 행렬을 연산하기 위해 각 행렬의 크기는 같아야 함.
- 그 다음 인덱스가 같은 값끼리 연산이 일어남.

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

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

그림 9-9 행렬의 연산

3. 파이썬 스타일 코드로 표현한 행렬

- 행렬의 연산을 파이썬 스타일 코드를 사용하여 표현하기

☞ 가장 쉬운 표현 방법은 별표(*)와 zip() 함수를 활용하는 것.

```
>>> 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]]
```

- 코드의 핵심: 'zip() 함수를 어떻게 활용하는가'

- 리스트 컴프리헨션 안에 2개의 for문이 있음.
- 뒤에 있는 for문이 먼저 실행되어 matrix_a와 matrix_b에서 zip() 함수를 통해 같은 인덱스에 있는 값들이 추출됨. 즉, [3, 6]과 [5, 8]이 튜플로 묶여 ([3, 6], [5, 8])로 추출됨.

3. 파이썬 스타일 코드로 표현한 행렬

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

- **t**: 2개의 리스트 값을 가진 하나의 튜플로 앞의 for문에 있는 리스트 컴프리헨션 구문에 들어감.
 - t는 1개의 튜플이므로 zip() 함수를 사용하기 위해 값을 언패킹해야 함.
 - [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)의 형태로 row 변수에 할당됨.
 - [sum(row) for row in zip(*t)] 코드에서 같은 위치에 있는 값끼리 묶여 row라는 이름의 튜플이 생성된 후 sum() 함수가 적용됨.
 - 이로 인하여 같은 위치의 값끼리 더해져 [[8, 14], [10, 12]]라는 결과가 나옴.

3. 파이썬 스타일 코드로 표현한 행렬

3.2 행렬의 동치

- 2개의 행렬이 서로 같은지를 나타내는 표현
- 만약 행렬이 같다면 '2개의 행렬이 동치'라고 말함.
- [그림 9-10]은 두 행렬이 'A = B'이기 위한 조건을 나타낸 것.

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

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

그림 9-10 행렬의 동치

3. 파이썬 스타일 코드로 표현한 행렬

- 두 행렬이 동치임을 확인하는 코드

👉 '행렬의 연산'과 비슷한 코드를 작성하되 불린형을 활용.

```
>>> 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
```

3. 파이썬 스타일 코드로 표현한 행렬

- **all() 함수:** 안에 있는 모든 값이 참일 경우에만 True를 반환함.
- **any() 함수:** 하나라도 참이 있으면 True를 반환 하고, 모두가 거짓일 때만 False를 반환함.

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

3. 파이썬 스타일 코드로 표현한 행렬

- 마지막에 같은 인덱스에 있는 값들을 row라는 튜플에 할당한 후, 마지막 for문인 for value in row 코드로 row 안의 값을 다시 value에 할당함.
- 그리고 모든 인덱스의 값이 같은지 확인하여 True 또는 False로 반환한 후, 마지막으로 all() 함수로 동치 여부를 확인함.
- 만약 all() 함수가 중간에 없다면 다음과 같은 결과가 출력됨.

```
>>> [[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]]
```

3. 파이썬 스타일 코드로 표현한 행렬

3.3 전치행렬(transpose matrix)

- 주어진 $m \times n$ 의 행렬에서 행과 열을 바꾸어 만든 행렬

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

(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

그림 9-11 전치행렬

- 전치행렬을 구현하기 위해 행과 열의 값을 변경해야 함.

3. 파이썬 스타일 코드로 표현한 행렬

```
>>> 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 핵심: `for t in zip(*matrix_a)`
 - 별표 때문에 리스트를 다음과 같이 언패킹함.

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

- 이렇게 언패킹한 상태에서 `zip()` 함수를 사용하면 같은 위치의 값들을 `t`로 할당할 수 있음. 즉, `[1, 4]`, `[2, 5]`, `[3, 6]`이 묶임.
- 이 값들이 그대로 리스트로 들어가면 전치행렬이 완성됨.

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

3. 파이썬 스타일 코드로 표현한 행렬

3.4 행렬의 곱셈

- 앞 행렬의 행과 뒤 행렬의 열을 선형 결합하면 됨.
- [그림 9-12]와 같이 대응되는 값들끼리 곱셈 연산하면 됨.

$$(1 \times 7) + (2 \times 9) + (3 \times 11)$$
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$$

그림 9-12 행렬의 곱셈

- [그림 9-13]과 같이 행렬의 곱셈을 위한 조건을 만족하여야 연산이 됨.



그림 9-13 행렬의 곱셈을 하기 위한 조건

3. 파이썬 스타일 코드로 표현한 행렬

- 코드 구현 - 전치행렬의 코드 기법을 사용하여 한 행렬에서는 열의 값을, 다른 행렬에서는 행의 값을 추출하여 곱하는 코드로 구성해야 함.

```
>>> 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]]
```

Thank you!