05
파이썬 스타일 코딩

목차

01. 파이썬 스타일 코드의 이해

파이썬 스타일 코드의 개념

- **파이썬 스타일 코드(pythonic code) :** 파이썬 스타일의 코드 작성 기법
- 특별한 문법이 아니라, 파이썬에서 기본적으로 제공하는 문법들을 활용하여 코딩하는 것이 바로 파이썬 스타일 코드이다.

01. 파이썬 스타일 코드의 이해

파이썬 스타일 코드의 개념

- 파이썬 스타일 코드의 대표적인 예로, 다음과 같이 for문을 사용하여 여러 단어를 붙이는 경우

```
>>> colors = ['red', 'blue', 'green', 'yellow']
>>> result = ''
>>> for s in colors:
...     result += s
...
>>> print(result)
redbluegreenyellow
```

- 가장 간단한 코딩 방법

```
>>> colors = ['red', 'blue', 'green', 'yellow']
>>> result = ''.join(colors)
>>> print(result)
redbluegreenyellow
```

01. 파이썬 스타일 코드의 이해

파이썬 스타일 코드를 사용하는 이유

- 파이썬 스타일 코드가 생긴 이유는 바로 파이썬의 철학 때문이다.
- 파이썬은 기본적으로 '인간의 시간이 컴퓨터의 시간보다 더 중요하다.'라는 개념을 가지고 있다. 코드상으로 사람이 해야 하는 일을 최대한 줄이면서 같은 목표를 달성할 수 있는 문법 체계를 가지고 있다.
- 파이썬 스타일 코드가 익숙해지면 코드 자체도 간결해지고 코드 작성 시간도 줄일 수 있다

02. 문자열의 분리 및 결합

문자열의 분리: split( ) 함수

- **split( ) 함수** : 문자열의 값을 특정 값을 기준으로 분리하여 리스트 형태로 변환하는 방법이다.

```
>>> items = 'zero one two three'.split()          # 빈칸을 기준으로 문자열 분리하기
>>> print (items)
['zero', 'one', 'two', 'three']
```

- 위 코드를 보면 문자열 'zero one two three'를 split( ) 함수를 사용하여 리스트형의 변수로 변환하였다. split( ) 함수 안에는 매개변수로 아무것도 입력하지 않았다. split( ) 함수는 텍스트를 아주 간단히 리스트 형태로 나누어 분리할 수 있다는 점에서 널리 사용되고 있다.

02. 문자열의 분리 및 결합

문자열의 분리: split( ) 함수

- split( ) 함수에 매개변수를 넣어 텍스트를 어떻게 분리하는지 알아보자.

```
>>> example = 'python,jquery,javascript'          # ","를 기준으로 문자열 나누기
>>> example.split(",")
['python', 'jquery', 'javascript']
>>> a, b, c = example.split(",")                  # 리스트에 있는 각 값을 a, b, c 변수로 언패킹
>>> print(a, b, c)
python jquery javascript
>>> example = 'theteamlab.univ.edu'
>>> subdomain, domain, tld = example.split('.')   # "."을 기준으로 문자열 나누기 → 언패킹
>>> print(subdomain, domain, tld)
theteamlab univ edu
```

02. 문자열의 분리 및 결합

문자열의 결합: join( ) 함수

- **join( ) 함수 :** 문자열로 구성된 리스트를 합쳐 하나의 문자열로 반환할 때 사용한다.
- join( ) 함수를 사용하는 방법은 구분자.join(리스트형) 형태로 사용할 수 있다.

```
>>> colors = ['red', 'blue', 'green', 'yellow']
>>> result = ''.join(colors)
>>> result
'redbluegreenyellow'
```

➡ colors라는 리스트 변수에는 색이름이 문자열의 요소로 들어 있다. 이 변수에 join( ) 함수를 적용하면 각 색이름이 붙어 하나의 문자열의 값으로 반환된다. 만약 색깔 사이에 다양한 구분자를 넣고 싶다면 join( ) 함수 앞에 ',' 나 '-' 등을 추가하면 된다.

02. 문자열의 분리 및 결합

문자열의 결합: join( ) 함수

- 다양한 구분자를 삽입한 예이다.

```
>>> result = ' '.join(colors)              # 연결 시, 1칸을 띄고 연결
>>> result
'red blue green yellow'
>>> result = ', '.join(colors)             # 연결 시 ", "으로 연결
>>> result
'red, blue, green, yellow'
>>> result = '-'.join(colors)              # 연결 시 "-"으로 연결
>>> result
'red-blue-green-yellow'
```

03. 리스트 컴프리헨션

리스트 컴프리헨션 다루기

- 리스트 컴프리헨션(list comprehension)의 기본 개념은 기존 리스트형을 사용하여 간단하게 새로운 리스트를 만드는 기법이다. 리스트와 for문을 한 줄에 사용할 수 있는 장점이 있다.

일반적인 반복문 + 리스트

```
>>> result = []
>>> for i in range(10):
...     result.append(i)
...
>>> result
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

리스트 컴프리헨션

```
>>> result = [i for i in range(10)]
>>> result
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

03. 리스트 컴프리헨션

■ 리스트 컴프리헨션 용법 : 필터링

- 필터링은 if문과 함께 사용하는 리스트 컴프리헨션이다. 일반적으로 짝수만 저장하기 위해서는 다음과 같은 코드를 작성해야 한다.

일반적인 반복문 + 리스트

```
>>> result = []
>>> for i in range(10):
...     if i % 2 == 0:
...         result.append(i)
...
>>> result
[0, 2, 4, 6, 8]
```

리스트 컴프리헨션

```
>>> result = [i for i in range(10) if i % 2 == 0]
>>> result
[0, 2, 4, 6, 8]
```

03. 리스트 컴프리헨션

리스트 컴프리헨션 용법 : 필터링

- 다음 코드를 보면, 기존 리스트 컴프리헨션문 끝에 if i % 2 == 0 을 삽입하여 해당 조건을 만족할 때만 i를 추가할 수 있게 한다. 만약 else문과 함께 사용하여 해당 조건을 만족하지 않을 때는 다른 값을 할당할 수 있다. 다음 코드를 보자.

```
>>> result = [i if i % 2 == 0 else 10 for i in range(10)]
>>> result
[0, 10, 2, 10, 4, 10, 6, 10, 8, 10]
```

- 위 코드처럼 if문을 앞으로 옮겨 else문과 함께 사용하면, 조건을 만족하지 않을 때 else 뒤에 i의 값을 할당하는 코드를 작성할 수 있다.

03. 리스트 컴프리헨션

■ 리스트 컴프리헨션 용법 : 중첩 반복문

- 리스트 컴프리헨션에서도 기존처럼 리스트 2개를 섞어 사용할 수 있다.
- 다음 코드와 같이 2개의 for문을 만들 수 있다.

```
>>> word_1 = "Hello"
>>> word_2 = "World"
>>> result = [i + j for i in word_1 for j in word_2]          # 중첩 반복문
>>> result
['HW', 'Ho', 'Hr', 'Hl', 'Hd', 'eW', 'eo', 'er', 'el', 'ed', 'lW', 'lo', 'lr', 'll', 'ld', 'lW',
'lo', 'lr', 'll', 'ld', 'oW', 'oo', 'or', 'ol', 'od']
```

- 위 코드를 보면, word_1에서 나오는 값을 먼저 고정한 후, word_2의 값을 하나씩 가져와 결과를 생성한다.

03. 리스트 컴프리헨션

리스트 컴프리헨션 용법 : 중첩 반복문

- 중첩 반복문에서도 필터링을 적용할 수 있다. 다음과 같이 반복문 끝에i f문을 추가하면 된다.

```
>>> case_1 = ["A", "B", "C"]
>>> case_2 = ["D", "E", "A"]
>>> result = [i + j for i in case_1 for j in case_2 if not(i==j)]
>>> result
['AD', 'AE', 'BD', 'BE', 'BA', 'CD', 'CE', 'CA']
```

03. 리스트 컴프리헨션

■ 리스트 컴프리헨션 용법 : 이차원 리스트

- 비슷한 방식으로 이차원 리스트(two-dimensional list) 를 만들 수 있다. 앞 중첩 반복문의 예시 코드 결과는 일차원 리스트(one-dimensional list) 였다. 그렇다면 하나의 정보를 열 row 단위로 저장하는 이차원 리스트는 어떻게 만들 수 있을까? 먼저 다음 코드를 보자.

```
>>> words = 'The quick brown fox jumps over the lazy dog'.split()
>>> print(words)
['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
>>> stuff = [[w.upper(), w.lower(), len(w)] for w in words]      # 리스트의 각 요소를 대문자, 소문자, 길이로 변환하여 이차원 리스트로 변환
```

- 가장 간단한 방법은 위 코드처럼 대괄호 2개를 사용하는 것이다. 이 코드는 기존 문장을 split( ) 함수로 분리하여 리스트로 변환한 후, 각 단어의 대문자, 소문자, 길이를 하나의 리스트로 따로 저장하는 방식이다.

03. 리스트 컴프리헨션

리스트 컴프리헨션 용법 : 이차원 리스트

- 저장한 후 결과를 출력하면, 다음과 같이 이차원 리스트를 생성할 수 있다.

```
>>> for i in stuff:
...     print (i)
...
['THE', 'the', 3]
['QUICK', 'quick', 5]
['BROWN', 'brown', 5]
['FOX', 'fox', 3]
['JUMPS', 'jumps', 5]
['OVER', 'over', 4]
['THE', 'the', 3]
['LAZY', 'lazy', 4]
['DOG', 'dog', 3]
```

03. 리스트 컴프리헨션

리스트 컴프리헨션 용법 : 이차원 리스트

- 다른 방법으로 for문 2개를 붙여 사용할 수도 있다. 여기서 한 가지 주의할 점은 for문 2개를 붙여 사용하면 대괄호의 위치에 따라 for문의 실행이 달라진다는 것이다. 이전에 배웠던 for문은 앞에 있는 for문이 먼저 실행된 후, 뒤의 for문이 실행되었다. 그래서 다음 코드의 경우 A가 먼저 고정되고, D, E, A를 차례대로 붙여 결과가 출력된다.

```
>>> case_1 = ["A", "B", "C"]
>>> case_2 = ["D", "E", "A"]
>>> result = [i + j for i in case_1 for j in case_2]
>>> result
['AD', 'AE', 'AA', 'BD', 'BE', 'BA', 'CD', 'CE', 'CA']
```

03. 리스트 컴프리헨션

리스트 컴프리헨션 용법 : 이차원 리스트

- 이차원 리스트를 만들기 위해서는 대괄호를 하나 더 사용해야 한다. 그리고 그와 동시에 먼저 작동하는 for문의 순서가 달라진다. 다음 코드는 위의 코드와 달리 리스트 안에 [i + j for iin case_1]이 하나 더 존재한다. 따라서 먼저 나온 for문이 고정되는 것이 아니라, 뒤의 for문이 고정된다. 즉, A부터 고정되는 것이 아니라 case_2의 첫 번째 요소인 D가 고정되고 A, B, C가 차례로 D 앞에 붙는다. 결과를 보면 이차원 리스트 형태로 출력된 것을 확인할 수 있다.

```
>>> result = [[ i + j for i in case_1] for j in case_2]
>>> result
[['AD', 'BD', 'CD'], ['AE', 'BE', 'CE'], ['AA', 'BA', 'CA']]
```

03. 리스트 컴프리헨션

■ 리스트 컴프리헨션 용법 : 이차원 리스트

- 다음 두 코드는 꼭 구분해야 한다.

```
① [i + j for i in case_1 for j in case_2]
② [[i + j for i in case_1] for j in case_2]
```

➡ 첫 번째 코드는 일차원 리스트를 만드는 코드로, 앞의 for문이 먼저 실행된다. 두 번째 코드는 이차원 리스트를 만드는 코드로, 뒤의 for문이 먼저 실행된다. 이 두 코드의 차이를 꼭 이해하고 넘어가자.

03. 리스트 컴프리헨션

리스트 컴프리헨션의 성능

- 왜 리스트 컴프리헨션을 사용할까? 문법적 간단함의 장점 외에도 성능이 뛰어나다.

코드 8-1 loop.py(일반적인 반복문 + 리스트)

```
1  def sclar_vector_product(scalar, vector):
2      result = []
3      for value in vector:
4          result.append(scalar * value)
5      return result
6
7  iteration_max = 10000
8
9  vector = list(range(iteration_max))
10 scalar = 2
11
12 for _ in range(iteration_max):
13     sclar_vector_product(scalar, vector)
```

03. 리스트 컴프리헨션

리스트 컴프리헨션의 성능

코드 8-2 listcomprehension.py(리스트 컴프리헨션)

```
1 iteration_max = 10000
2
3 vector = list(range(iteration_max))
4 scalar = 2
5
6 for _ in range(iteration_max):
7     [scalar * value for value in range(iteration_max)]
```

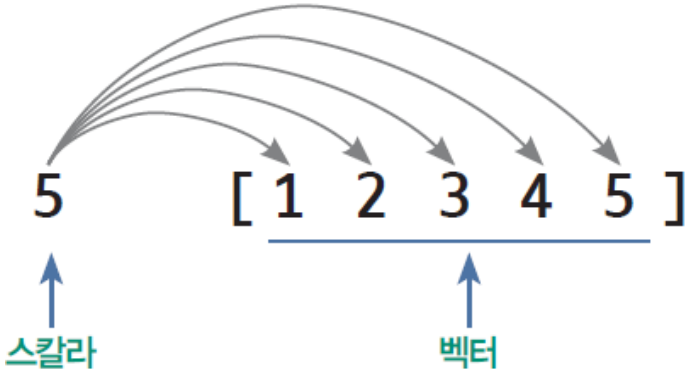
- [코드 8-1]과 [코드 8-2]를 실행하여 결과를 비교해 보자. 두 코드는 똑같은 목적이 있는데,

➡ 모두 대학 과정의 선형대수나 고등학교 과정의 행렬에서 배우는 벡터(vector)와 스칼라(scalar)의 곱셈을 실행하는 코드이다. 벡터-스칼라 곱셈은 하나의 스칼라값이 각 벡터의 값과 곱해져 최종값을 만든다.

03. 리스트 컴프리헨션

리스트 컴프리헨션의 성능

- 스칼라는 파이썬의 정수형 또는 실수형으로, 벡터는 리스트로 치환하여 생각하면 쉽게 이해할 수 있다.



[벡터와 스칼라]

03. 리스트 컴프리헨션

리스트 컴프리헨션의 성능

- 두 코드의 성능을 비교하기 위해 리눅스 계열에서 사용하는 time 명령어의 결과를 캡처한 것이다. 코드의 총 실행 시간을 확인할 수 있다.

```
real    0m10.230s
user    0m10.203s
sys     0m0.023s
```

(a) 일반적인 반복문 + 리스트

```
real    0m7.788s
user    0m7.762s
sys     0m0.021s
```

(b) 리스트 컴프리헨션

[코드의 실행 시간을 통한 성능 비교]

04. 다양한 방식의 리스트값 출력

■ 리스트값에 인덱스를 붙여 출력: enumerate( ) 함수

- enumerate( ) 함수는 리스트값을 추출할 때 인덱스를 붙여 함께 출력.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):    # 리스트에 있는 인덱스와 값을 언패킹
...     print(i, v)
...
0 tic
1 tac
2 toe
```

➡ 리스트형의 ['tic', 'tac', 'toe']에 enumerate( ) 함수를 적용하였다. enumerate( ) 함수를 적용하면 인덱스와 리스트의 값이 언패킹되어 추출되는데, 위 코드에서는 'tic', 'tac', 'toe'에 각각 0, 1, 2의 인덱스가 붙어 출력되는 것을 알 수 있다.

04. 다양한 방식의 리스트값 출력

리스트값에 인덱스를 붙여 출력: enumerate( ) 함수

- enumerate( ) 함수는 주로 딕셔너리형으로, 인덱스를 키로, 단어를 값으로 하여 쌍으로 묶어 결과를 출력하는 방식을 사용한다.

```
>>> {i:j for i,j in enumerate('TEAMLAB is an academic institute located in South Korea.'.split())}
{0: 'TEAMLAB', 1: 'is', 2: 'an', 3: 'academic', 4: 'institute', 5: 'located', 6: 'in', 7: 'South', 8: 'Korea.'}
```

04. 다양한 방식의 리스트값 출력

리스트값을 병렬로 묶어 출력: zip( ) 함수

- zip( ) 함수는 1개 이상의 리스트값이 같은 인덱스에 있을 때 병렬로 묶는 함수이다.

```
>>> alist = ['a1', 'a2', 'a3']
>>> blist = ['b1', 'b2', 'b3']
>>> for a, b in zip(alist, blist):                    # 병렬로 값을 추출
...     print(a, b)
...
a1 b1
a2 b2
a3 b3
```

- 위 코드는 alist와 blist를 zip( ) 함수로 묶는 코드이다. 두 리스트 모두 3개의 값이 있어, 같은 인덱스의 값을 묶어 출력한다.

04. 다양한 방식의 리스트값 출력

■ 리스트값을 병렬로 묶어 출력: zip( ) 함수

- zip( ) 함수로 묶으면 다양한 추가 기능을 만들 수 있다. 예를 들어, 다음 코드처럼 같은 위치에 있는 값끼리만 더할 수 있다.

```
>>> a, b, c = zip((1, 2, 3), (10, 20, 30), (100, 200, 300))
>>> print (a, b, c)
(1, 10, 100) (2, 20, 200) (3, 30, 300)
>>> [sum(x) for x in zip((1, 2, 3), (10, 20, 30), (100, 200, 300))]
[111, 222, 333]
```

➡ 같은 인덱스에 있는 숫자를 추출하여 sum( ) 함수를 적용하면 각각의 값을 더해 출력할 수 있다. 앞서 벡터-스칼라 곱셈을 봤는데, 이 기법으로 벡터 덧셈이나 매트릭스 덧셈 등을 유용하게 만들 수 있다.

04. 다양한 방식의 리스트값 출력

■ 리스트값을 병렬로 묶어 출력: zip( ) 함수

- enumerate( ) 함수와 zip( ) 함수를 같이 사용하면 다음 코드처럼 사용할 수 있다.

```
>>> alist = ['a1', 'a2', 'a3']
>>> blist = ['b1', 'b2', 'b3']

>>> for i, (a, b) in enumerate(zip(alist, blist)):
...     print (i, a, b)                    # (인덱스, alist[인덱스], blist[인덱스]) 표시
...
0 a1 b1
1 a2 b2
2 a3 b3
```

- 위 코드에서 alist와 blist를 zip( ) 함수로 묶고 enumerate( ) 함수를 적용하여 같은 인덱스의 값끼리 묶어 출력하였다.

05. 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
```

05. 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.

05. 람다 함수

여기서 잠깐! 람다 함수를 표현하는 다른 방식

- 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.

05. 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.

06. 별표의 활용

별표의 사용

- 별표(asterisk)는 곱하기 기호(*)를 뜻한다. 별표는 기본 연산자로, 단순 곱셈이나 제곱 연산에 많이 사용되었다.
- 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'>
```

06. 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.

⇒ 첫 번째 함수의 경우, 2, 3, 4, 5, 6이 변수 args에 할당된 것이다. 별표 여러 개의 변수를 담는 컨테이너로서의 속성을 부여하였기 때문이다

06. 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. 정수형인 a와 튜플형인 args가 매개변수에 입력된다. 핵심은 print(a, *args) 코드이다. 사실 args는 함수에 하나의 변수로 들어갔기 때문에 일반적이라면 다음과 같이 출력되어야 한다.

06. 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.

06. 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)
```

06. 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.

06. 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'>
```

06. 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
```