단원 08
컴프리헨션과 매핑

인공지능소프트웨어학과

강환수 교수

Section 1. 컴프리헨션

-

컴프리헨션 개요

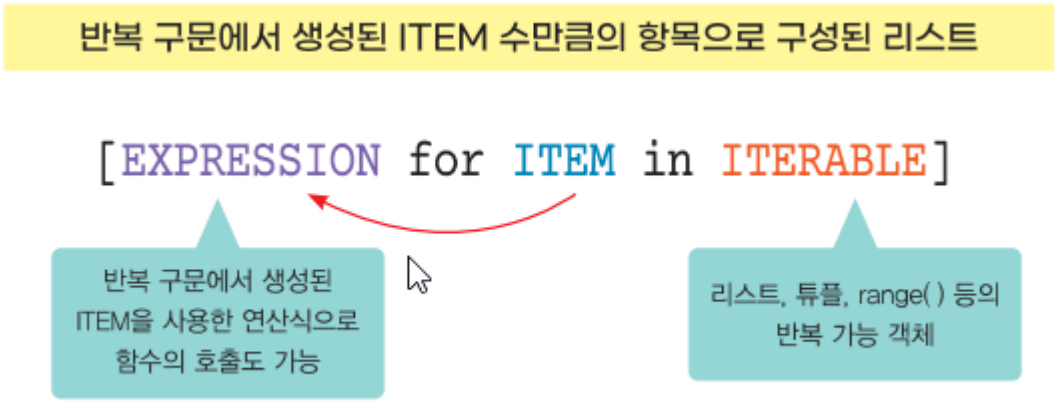
컴프리헨션(comprehension)은 리스트, 사전, 집합 등의 다양한 컨테이너를 간결하고 우아하게 만드는 방법

- **컴프리헨션**
  - 규칙성이나 조건을 가지는 항목으로 구성되는 모임을 직관적으로 보다 쉽게 만들 수 있음
    - 한번 이해하면 훨씬 읽기 쉬운 형식으로 작성 가능
  - 처음에는 다소 어렵다고 느낄 수 있지만
    - 파이썬 컴프리헨션은 코드를 보다 편하고, 파이썬스럽게(pythonic) 만드는 방법
  - 컴프리헨션은 함축, 축약, 내포, 내장 등으로도 불림

▲ 그림 1 다양한 컴프리헨션

리스트를 간단히 만드는 컴프리헨션

- **리스트 컴프리헨션**
  - 반복 가능한 ITERABLE의 각 요소 ITEM에 대해 실행되는 식 EXPRESSION을 포함하는 대괄호로 구성



▲ **그림 2** 리스트 컴프리헨션 구문 개요

```
>>> # 간단한 리스트 컴프리헨션
>>> [i for i in range(5)]
[0, 1, 2, 3, 4]
```

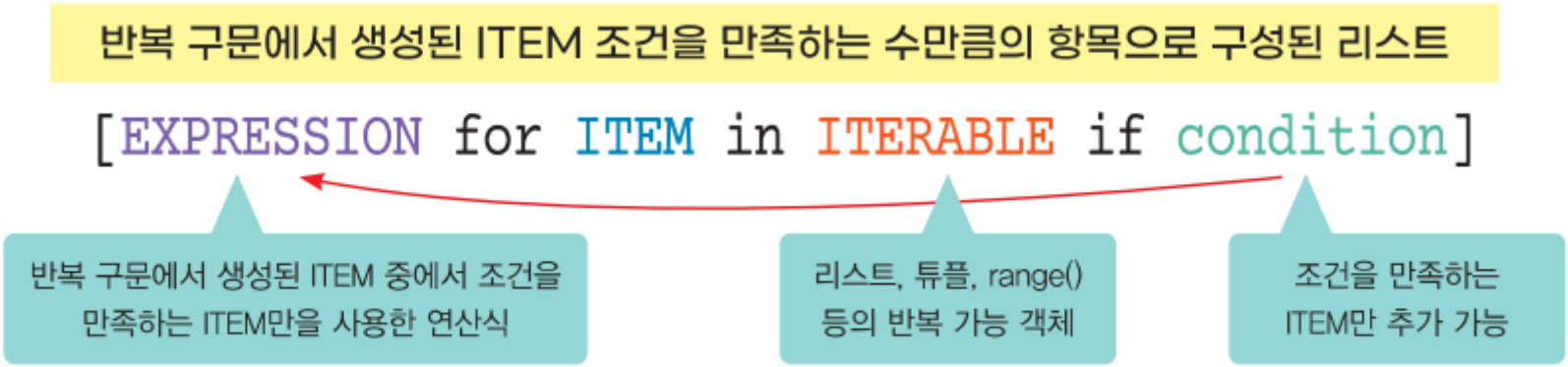
```
>>> # 리스트 컴프리헨션을 전통적인 빈 리스트와
>>> # 반복에서 추가 구문으로 작성
>>> lst = []
>>> for i in range(5):
...     lst.append(i)
...
>>> lst
[0, 1, 2, 3, 4]
```

▲ **그림 3** 리스트 컴프리헨션(왼쪽)과 전통적인 for in 문장으로 만드는 리스트(오른쪽)

조건에 따라 항목을 포함하는 컴프리헨션

컴프리헨션의 반복 항목 중에서 조건 condition에 따라 리스트에 삽입하거나 뺄 수 있는 구문

- **반복 이후에 조건을 추가**
  - 조건을 만족하는 항목만 리스트에 추가



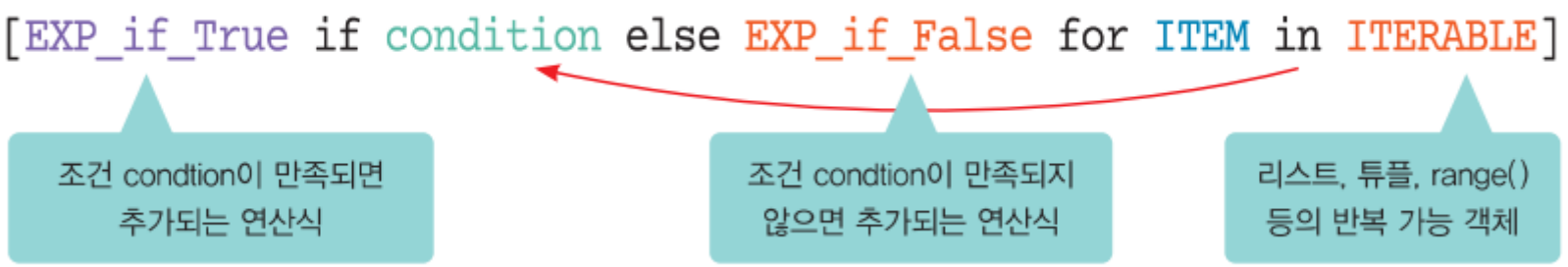
▲ **그림 4** 조건 if condition을 만족하는 ITEM만 추가 가능한 컴프리헨션

```
>>> # 반복 항목 중에서 짝수를 리스트에 삽입
>>> [i for i in range(6) if i%2 == 0]
[0, 2, 4]
>>>
>>> # 반복 항목 중에서 홀수를 리스트에 삽입
>>> [i for i in range(6) if i%2 == 1]
[1, 3, 5]
```

조건에 따라 항목을 변형하는 컴프리헨션

- **조건 연산식 구문을 사용**

반복 구문에서 생성된 모든 ITEM에서 조건에 따라 변형된 항목으로 구성된 리스트



▲ **그림 7** 항목을 조건에 따라 변형한 컴프리헨션

```
>>> # 짝수이면 0, 홀수이면 1을 리스트에 삽입
>>> [0 if i%2 == 0 else 1 for i in range(10)]
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
>>>
>>> # 짝수이면 even, 홀수이면 odd를 리스트에 삽입
>>> ['even' if i%2 == 0 else 'odd' for i in range(10)]
['even', 'odd', 'even', 'odd', 'even', 'odd', 'even', 'odd', 'even', 'odd']
```

중첩된 리스트를 만드는 컴프리헨션

- 리스트 컴프리헨션 내부의 항목으로 리스트를 구성하는 중첩된 리스트

반복 구문에서 생성된 모든 ITEM에서 내부 리스트를 구성하는 리스트 컴프리헨션


```
>>> # 변화되는 중첩된 2차원 리스트
>>> [[i] for i in range(3)]
[[0], [1], [2]]
>>> [[i, i+1] for i in range(3)]
[[0, 1], [1, 2], [2, 3]]
>>> [[i, i+1, i+2] for i in range(3)]
[[0, 1, 2], [1, 2, 3], [2, 3, 4]]
```

중첩된 반복을 사용한 다양한 리스트 컴프리헨션

- 리스트 컴프리헨션에서 중첩된 반복의 제어 변수 i와 j를 사용

중첩된 반복 구문에서 제어 변수를 활용하는 리스트 컴프리헨션

```
[EXPESSION for i in ITERABLE1 for j in ITERABLE2]
```

```
for i in ITERABLE1
    for j in ITERABLE2]
        ...
```

```
>>> # 중첩된 반복을 사용해 생성한 리스트
>>> a = [i + j for i in range(3) for j in range(2)]
>>> print(a)
[0, 1, 1, 2, 2, 3]
>>> b = [i * j for i in range(3) for j in range(2)]
>>> print(b)
[0, 0, 0, 1, 0, 2]
```

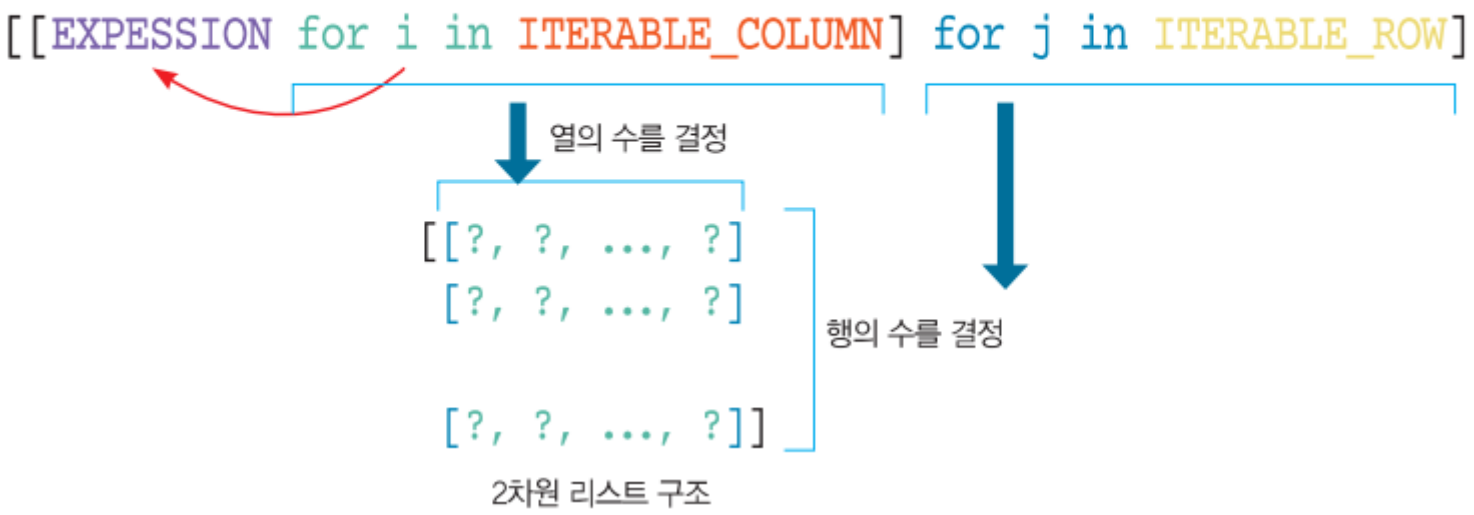
- i: 0 1 2
- J: 0 1 0 1 0 1

중첩된 컴프리헨션

리스트 컴프리헨션 내부에 다시 리스트 컴프리헨션을 사용

- **규칙성을 갖는 행과 열의 항목으로 2차원 리스트를 생성**

중첩된 리스트 컴프리헨션을 활용하는 리스트 컴프리헨션




```
>>> [[col for col in range(3)] for row in range(2)]
[[0, 1, 2], [0, 1, 2]]
>>> [[col+1 for col in range(4)] for row in range(3)]
[[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]
```

사전을 쉽게 만드는 딕셔너리 컴프리헨션

딕셔너리를 생성하는 간결하고 효율적인 방법으로 딕셔너리 컴프리헨션 제공

- **딕셔너리 컴프리헨션**

```
{key_expression : value_expression for element in iterable if condition}
```

▲ 그림 11 딕셔너리 컴프리헨션 구조

- Key_expression: 사전의 키 값
- value_expression: 키에 대한 값
- Iterable: 반복할 수 있는 객체(리스트, 튜플, range() 등)

```
>>> import math
>>> {i: i*2 for i in range(5)}
{0: 0, 1: 2, 2: 4, 3: 6, 4: 8}
>>> {i: i**2 for i in range(5)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

>>> words = ['apple', 'banana', 'cherry', 'avocado']
>>> first_letter_dict = {word: word[0].upper() for word in words}
>>> print(first_letter_dict)
{'apple': 'A', 'banana': 'B', 'cherry': 'C', 'avocado': 'A'}
```

집합 컴프리헨션

집합 컴프리헨션은 파이썬에서 집합을 생성하기 위해 사용되는 간편한 방법

- **외부에 중괄호 { }를 사용하여 집합을 생성**

```
>>> words = ["apple", "banana", "grape", "orange", "kiwi"]
>>> long_words = {word for word in words if len(word) >= 5}
>>> print(long_words)
{'grape', 'banana', 'orange', 'apple'}
```

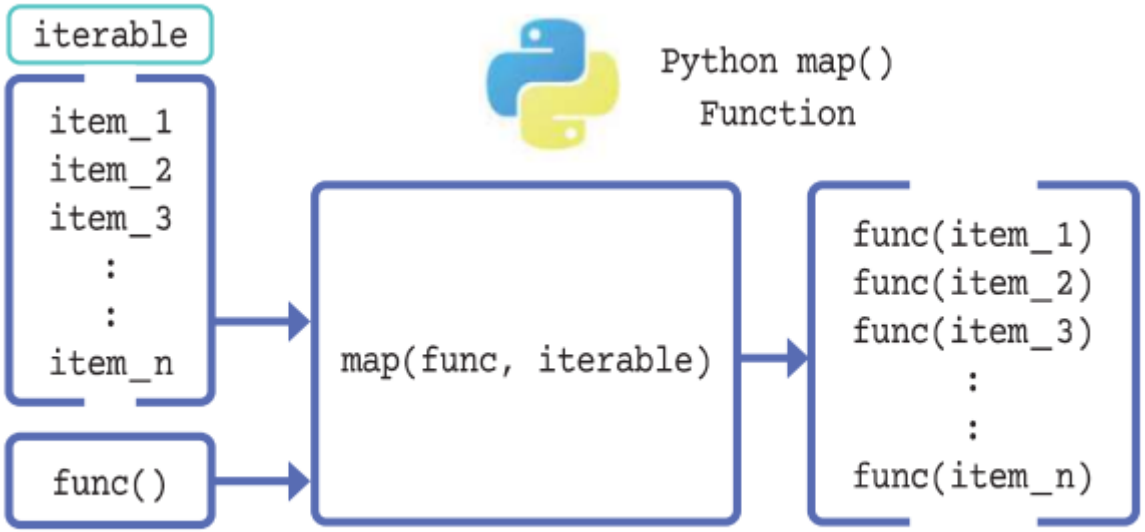
Section 2. 매핑과 필터

-

매핑 개요

함수 map()를 사용하는 매핑(mapping)

- **내장 함수 map()**
  - 반복 구문을 사용하지 않고 반복가능(iterable) 컨테이너(container)의 모든 항목(item)을 특정 함수의 인자로 처리할 수 있는 함수
    - 반복적인 개체의 각 항목에 변환 함수를 적용해 새로운 반복가능 개체로 변환



▲ 그림 12 내장 함수 map() 개요

함수 map()

- **함수 map(func, *iterables)**
  - 두 번째 인자로는 *iterables로 여러 개의 반복가능 객체

내장 함수 map(func, *iterables)

```
map(function, iterable_1[, iterable_2, iterable_3, ..., iterable_n])
```

문자열, range, 목록, 튜플, 집합, 딕셔너리 등이 가능하다.

- function: iterable의 항목을 변환하는 함수
- itrable_i: iterable의 항목을 함수의 인자로 사용, 함수의 매개변수 수만큼의 iterable 필요
- 반환 값: map 객체

```
>>> def my_mult(x, y):
...     return x * y
...
>>> list(map(my_mult, range(1, 5), range(1, 5)))
[1, 4, 9, 16]

>>> list(map(lambda inch: inch * 2.54, range(6)))
[0.0, 2.54, 5.08, 7.62, 10.16, 12.7]
```

필터 개요

- **내장 함수 filter()**
  - 여러 개의 항목에서 조건에 맞는 일부의 데이터만 추려낼 때 사용

내장 함수 filter(cond_function, iterable)

```
filter(cond_function, iterable)
```

- cond_function: 각 원소를 평가하는 조건을 확인하는 함수 또는 람다 표현식
- itrable: 조건을 확인할 반복 가능 객체
- 반환 값: iterable의 각 원소에 대해 cond_function을 평가하고, 조건을 만족하는 원소들로 구성된 새로운 filter 객체 반환

```
>>> nums = [-10, 3, 4, -3, 5]
>>> positive_nums = list(filter(lambda x: x > 0, nums))
>>> print(positive_nums)
[3, 4, 5]
```

Section 3. 이터레이터와 제너레이터

-

반복가능(iterable) 개요

- **iterable**
  - 반복가능하다는 것은 모임의 항목을 한 번에 하나씩 반환할 수 있는 개체
  - '반복가능'은 '순회가능'이라고도 표현할 수 있으며 원어로는 iterable
  - 모든 시퀀스 유형인 range, 목록, 문자열, 튜플 등과 시퀀스가 아닌 모임 객체인 사전과 집합, 파일 등
  - 또한, 메소드 __iter__() 또는 __getitem__() 메서드로 정의하는 클래스는 반복가능 객체

```
>>> hasattr(range, '__iter__')
True
>>> hasattr(range, '__getitem__')
True
```

- 정의: 객체를 반복(iterate)할 수 있도록 만드는 메소드입니다.
- 역할: 이터레이터 객체를 반환해야 하며, 이터레이터는 __next__() 메소드를 포함해야 합니다.

- 정의: 객체에서 인덱스로 접근할 수 있는 기능을 제공합니다.
- 역할: 리스트나 튜플처럼 인덱스를 통해 요소를 반환하는 데 사용됩니다.

이터레이터(iterator) 개요

이터레이터(iterator)는 항목 모임에서 반복할 수 있게 해주는 객체

- 이터레이터(반복자)
  - 이터레이터는 컨테이너를 순회하고 해당 항목을 참조할 수 있도록 구현
  - 반복자는 두 가지 주요 작업을 수행해야 함
    - 자신인 이터레이터를 반환
      - 메서드 __iter__()
    - 항목 모임인 컨테이너에서 한 번에 한 항목씩 데이터 반환
      - 메서드 __next__()

- 사용자 직접 정의하는 반복자 클래스를 생성하려면 이 두 메서드를 구현
  - 두 가지 특수 메서드 __iter__()와 __next__()를 구현

목록을 이터레이터 객체로 생성

목록, 튜플, 사전, 집합 등은 이터레이터가 아님, iterable 임

- **목록을 이터레이터로 생성**
  - 내장 함수 iter(iterable)

```
>>> iter_lst = iter([1, 2, 3])
>>> iter_lst
<list_iterator object at 0x000001FFB7F31780>
```

- **이터레이터는 마지막 값을 반환된 이후에 다시 next()가 호출되면**
  - StopIteration 예외가 발생

```
>>> next(iter_lst)
1
>>> next(iter_lst)
2
>>> next(iter_lst)
3
>>> next(iter_lst)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

제너레이터 개요

- 제너레이터
  - 반복가능 객체인 이터레이터를 만들어주는 함수
    - 대량의 값 시퀀스를 생성하려고 할 때 유용
  - 모든 값을 한 번에 생성하지 않으므로 메모리와 CPU 자원을 효율적으로 활용
    - 성능을 향상시키는 데 도움
- 제너레이터를 정의하는 방법
  - 함수 정의
    - return 문 대신 yield 문을 사용
      - yield 문은 값을 반환하고 함수의 상태를 일시적으로 저장
      - 이렇게 저장된 상태는 다음 호출 시에 사용되어 반복 가능한 객체를 생성

제너레리터 생성과 활용

- 쉽게 for in에서 활용 가능

❖ 함수가 yield 문을 만나면 실행이 일시 중지되었다가, next()가 호출되면 중단된 지점부터 실행을 재개함.

```
>>> def my_generator():
...     yield 1
...     yield 2
...
>>> g = my_generator()
>>> type(g)
<class 'generator'>
>>> dir(g)
['__class__', '__del__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gets
tate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__lt__', '__name__', '__ne__', '__ne
w__', '__next__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__sub
classhook__', 'close', 'gi_code', 'gi_frame', 'gi_running', 'gi_suspended', 'gi_yieldfrom', 'send', 'throw']
>>> next(g)
1
>>> next(g)
2
>>> next(g) # StopIteration
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> for i in my_generator():
...     print(i)
...
1
2
>>>
```

제너레이터 컴프리헨션

- 대괄호 [ ] 대신 소괄호 ( )를 사용하여 제너레이터를 간편히 생성
  - 튜플이 아니라 제너레이터가 생성
- 제너레이터 컴프리헨션 장점
  - 메모리를 효율적으로 사용
  - 한 번에 모든 값을 생성하지 않고 필요한 시점에 값을 생성하여 속도를 향상

```
>> squares = (x**2 for x in range(1, 11))
>>> for _ in range(10):
...     print(next(squares), end=' ')
...
1 4 9 16 25 36 49 64 81 100
```

sequence, iterable, iterator, generator 비교

종류	특징	구분	자료형 사례
sequence	순서가 있는 모임으로 인덱싱이 가능	정수로 참조하는 메서드 __getitem__과 __len__ 메서드 지원	str, range, list, tuple * dict, set는 sequence가 아님.
iterable	순차적으로 항목 반환 가능	메서드 __iter__ 또는 __getitem__ 지원	str, range, list, tuple, dict, set
iterator	내장 함수 next(obj) 호출 가능	메서드 __iter__와 __next__ 지원	range_iterator, list_iterator, tuple_iterator, dict_iterator, set_iterator generator
generator	함수에서 yield로 반환 또는 컴프리헨션으로 생성	iterator를 생성해주는 함수로, yield로 값을 반환하는 함수 또는 컴프리헨션으로 만드는 generator	generator

제너레이터 컴프리헨션의 메모리 효율성

제너레이터는 일반 리스트보다 메모리를 적게 사용

- **제너레이터는 값을 한 번에 모두 생성하지 않고 필요한 시점에 값을 생성하기 때문**
  - 모듈 sys의 sys.getsizeof() 함수를 사용하여 바이트 단위로 객체의 메모리 사용량을 확인

```
>>> import sys
>>>
>>> # 리스트 컴프리헨션
>>> list_comp = [x for x in range(1000000)]
>>> # generator 컴프리헨션
>>> gen_comp = (x for x in range(1000000))
>>>

>>> print(sys.getsizeof(list_comp))  # 리스트 컴프리헨션의 메모리 사용량 출력
8448728
>>> print(sys.getsizeof(gen_comp))   # generator 컴프리헨션의 메모리 사용량 출력
200
```

- **생성되는 반복되는 수의 영향을 받지 않음, 반복 수를 천만 개로 해도 동일**

```
>>> gen_comp = (x for x in range(10000000))
>>> print(sys.getsizeof(gen_comp))   # generator 컴프리헨션의 메모리 사용량 출력
200
```

심화 학습: 리스트와 제너레이터의 생성 시간과 메모리 사용

다음은 교재에 없는 내용임

- **코드**

```
# %% 심화 학습: 리스트와 제너레이터의 생성 시간과 메모리 사용 비교
import time
import sys

# 시간 측정 함수
def measure_time_and_size(expression):
    start_time = time.time()
    obj = expression()
    elapsed_time = time.time() - start_time
    return elapsed_time, sys.getsizeof(obj)

# 리스트 컴프리헨션과 제너레이터 컴프리헨션 비교
list_time, list_size = measure_time_and_size(lambda: [x for x in range(1000000)])
gen_time, gen_size = measure_time_and_size(lambda: (x for x in range(1000000)))

# 결과 출력
print(f"리스트 컴프리헨션 - 시간: {list_time:.6f}초, 메모리: {list_size}바이트")
print(f"제너레이터 컴프리헨션 - 시간: {gen_time:.6f}초, 메모리: {gen_size}바이트")
```

❖ 실제 람다함수를 호출

❖ 리스트를 생성하는 함수 자체를 인자로 넘김

- **결과**
  - 리스트 컴프리헨션 - 시간: 0.044000초, 메모리: 8448728바이트
  - 제너레이터 컴프리헨션 - 시간: 0.000000초, 메모리: 192바이트