

Ch9. 리스트, 튜플, 딕셔너리 심화

- 리스트 표현식 사용하기

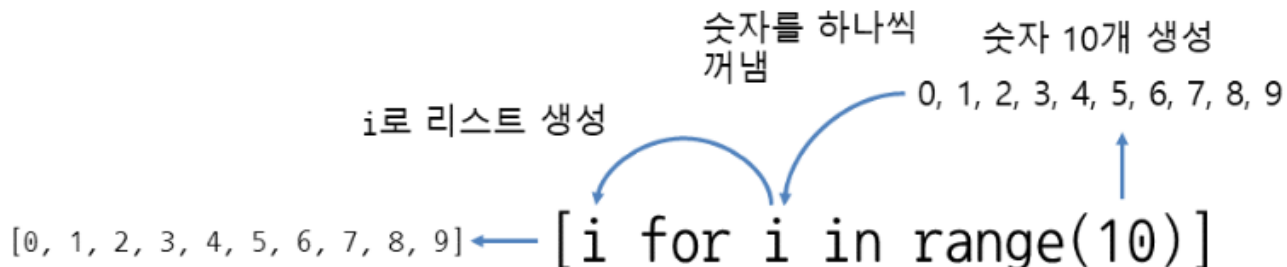
- 파이썬의 리스트가 특이한 점 : 리스트 안에 for 반복문과 if 조건문을 사용가능
- 리스트 안에 식, for 반복문, if 조건문 등을 지정하여 리스트를 생성하는 것을 리스트 컴프리헨션(list comprehension)이라고 함
- 리스트 내포, 리스트 내장, 리스트 축약, 리스트 해석 등으로 씀
- 컴프리헨션 능력, 이해력, 시험 등의 뜻, 어떤 것을 잡아서 담아둔다는 뜻도 존재
- 식으로 지정해서 생성된 것을 리스트로 잡아두는 것이 리스트 컴프리헨션임

- [식 for 변수 in 리스트]
- list(식 for 변수 in 리스트)

```
>>> a = [i for i in range(10)]      # 0부터 9까지 숫자를 생성하여 리스트 생성
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> b = list(i for i in range(10))  # 0부터 9까지 숫자를 생성하여 리스트 생성
>>> b
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- range(10)으로 0부터 9까지 생성하여 변수 i에 숫자를 꺼내고, 최종적으로 i를 이용하여 리스트를 만든다는 뜻

- 리스트 표현식 사용하기


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

- [i for i in range(10)]는 변수 i를 그대로 사용하지만, 다음과 같이 식 부분에서 i를 다른 값과 연산하면 각 연산의 결과를 리스트로 생성함

```
>>> c = [i + 5 for i in range(10)]    # 0부터 9까지 숫자를 생성하면서 값에 5를 더하여 리스트 생성
>>> c
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> d = [i * 2 for i in range(10)]    # 0부터 9까지 숫자를 생성하면서 값에 2를 곱하여 리스트 생성
>>> d
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

- 리스트 표현식에서 if 조건문 사용하기

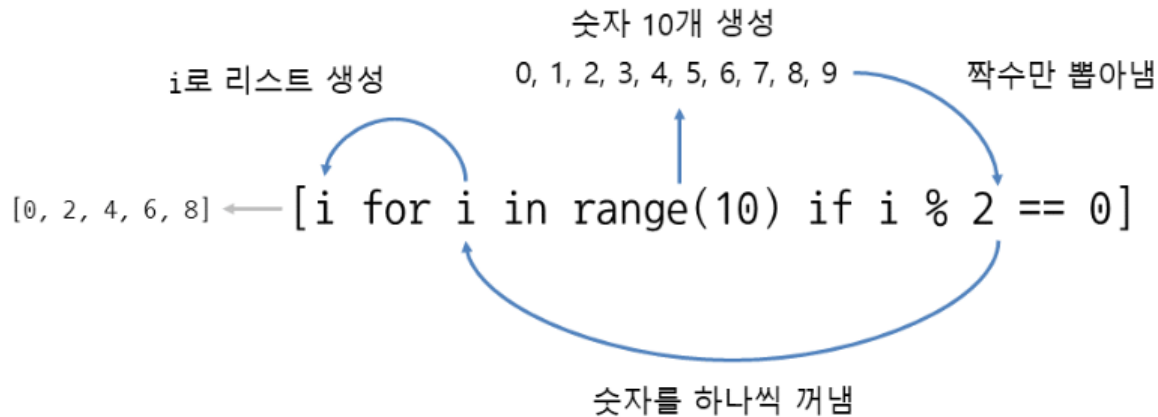
- if 조건문은 for 반복문 뒤에 지정함

- [식 for 변수 in 리스트 if 조건식]
 - list(식 for 변수 in 리스트 if 조건식)

```
>>> a = [i for i in range(10) if i % 2 == 0]    # 0~9 숫자 중 2의 배수인 숫자(짝수)로 리스트 생성
>>> a
[0, 2, 4, 6, 8]
```

- 리스트 표현식에서 if 조건문 사용하기

- for 반복문 뒤에 if 조건문을 지정하면 숫자를 생성한 뒤 if 조건문에서 특정 숫자만 뽑아내서 리스트를 생성함



```
>>> b = [i + 5 for i in range(10) if i % 2 == 1]    # 0~9 숫자 중 홀수에 5를 더하여 리스트 생성
>>> b
[6, 8, 10, 12, 14]
```

- for 반복문과 if 조건문을 여러 번 사용하기

```
[식 for 변수1 in 리스트1 if 조건식1      for 변수2 in 리스트2 if 조건식2      ...      for 변수n in 리스트n if 조건식n]
```

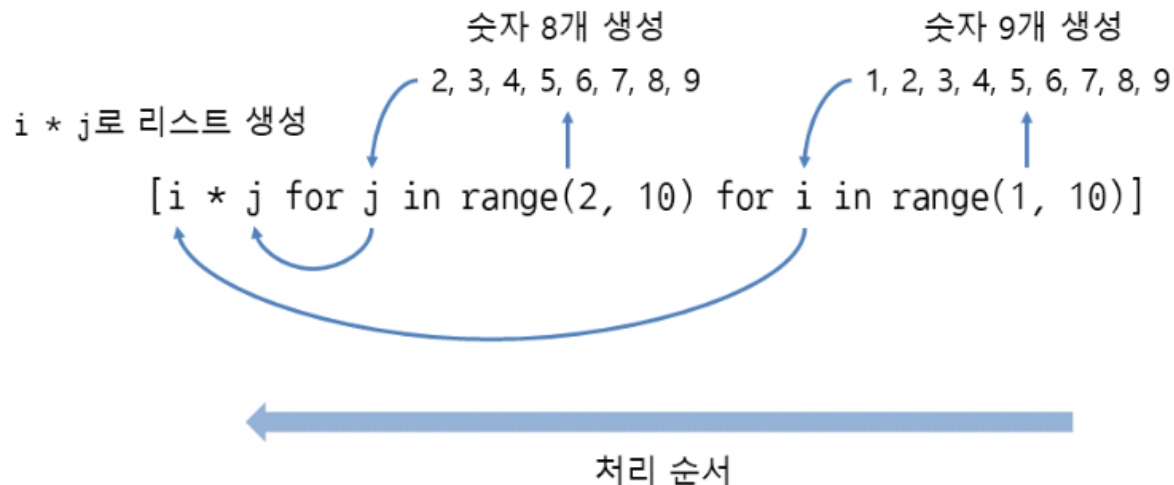
```
list(식 for 변수1 in 리스트1 if 조건식1      for 변수2 in 리스트2 if 조건식2      ...      for 변수n in 리스트n if 조건식n)
```

- 다음은 2단부터 9단까지 구구단을 리스트 생성함

```
>>> a = [i * j for j in range(2, 10) for i in range(1, 10)]
>>> a
[2, 4, 6, 8, 10, 12, 14, 16, 18, 3, 6, 9, 12, 15, 18, 21, 24, 27, 4, 8, 12, 16, 20, 24, 28, 32, 36, 5, 10, 15, 20, 25, 30, 35, 40, 45, 6, 12, 18, 24, 30, 36, 42, 48, 54, 7, 14, 21, 28, 35, 42, 49, 56, 63, 8, 16, 24, 32, 40, 48, 56, 64, 72, 9, 18, 27, 36, 45, 54, 63, 72, 81]
```

- for 반복문과 if 조건문을 여러 번 사용하기
 - 코드를 한 줄로 입력했지만 다음과 같이 여러 줄로 입력해도 됨
 - 가독성을 위해 들여쓰기를 해주는 것이 좋음

```
a = [i * j for j in range(2, 10)
      for i in range(1, 10)]
```



- 리스트에 map 사용하기

- map은 리스트의 요소를 지정된 함수로 처리해주는 함수임
(map은 원본 리스트를 변경하지 않고 새 리스트를 생성)

- `list(map(함수, 리스트))`
- `tuple(map(함수, 튜플))`

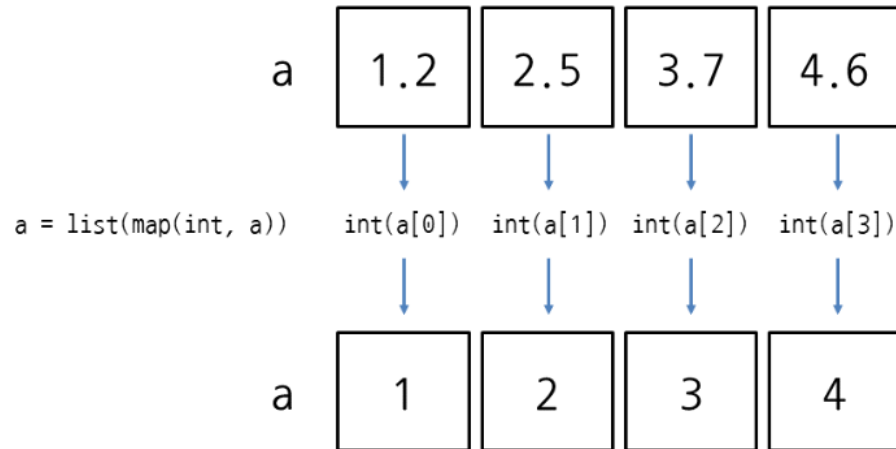
- 먼저 for 반복문을 사용해서 변환

```
>>> a = [1.2, 2.5, 3.7, 4.6]
>>> for i in range(len(a)):
...     a[i] = int(a[i])
...
>>> a
[1, 2, 3, 4]
```

- map을 사용하면 편리

```
>>> a = [1.2, 2.5, 3.7, 4.6]
>>> a = list(map(int, a))
>>> a
[1, 2, 3, 4]
```


- 리스트에 map 사용하기



- `range`를 사용해서 숫자를 만든 뒤 숫자를 문자열로 변환

```
>>> a = list(map(str, range(10)))
>>> a
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

- 리스트를 출력해보면 각 요소가 `'`(작은따옴표)로 묶인 것을 볼 수 있음

- 2차원 리스트 사용하기

- 2차원 리스트는 가로×세로 형태로 이루어져 있으며
행(row)과 열(column) 모두 0부터 시작함



The diagram illustrates a 2D list structure. A horizontal blue arrow labeled '가로 크기' (Horizontal Size) points to the right above the table. A vertical blue arrow labeled '세로 크기' (Vertical Size) points downwards to the left of the table. The table has 4 columns labeled '열 0', '열 1', '열 2', and '열 3' at the top, and 3 rows labeled '행 0', '행 1', and '행 2' on the left. The table cells are empty.

	열 0	열 1	열 2	열 3
행 0				
행 1				
행 2				

- 2차원 리스트를 만들고 요소에 접근하기

- 2차원 리스트는 리스트 안에 리스트를 넣어서 만들 수 있으며
안쪽의 각 리스트는 ,(콤마)로 구분함

- 리스트 = [[값, 값], [값, 값], [값, 값]]

- 숫자 2개씩 3묶음으로 리스트 생성

```
>>> a = [[10, 20], [30, 40], [50, 60]]  
>>> a  
[[10, 20], [30, 40], [50, 60]]
```

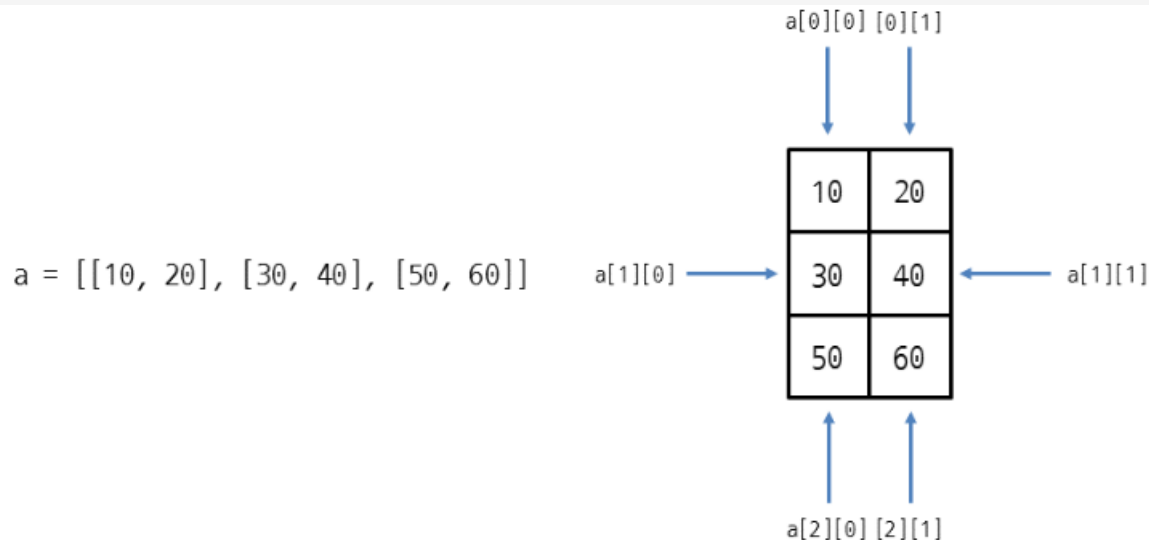
- 리스트를 한 줄로 입력했지만 가로, 세로를 알아보기 쉽게 세 줄로 입력해도 됨

```
a = [[10, 20],  
      [30, 40],  
      [50, 60] ]
```

- 2차원 리스트 요소에 접근하기

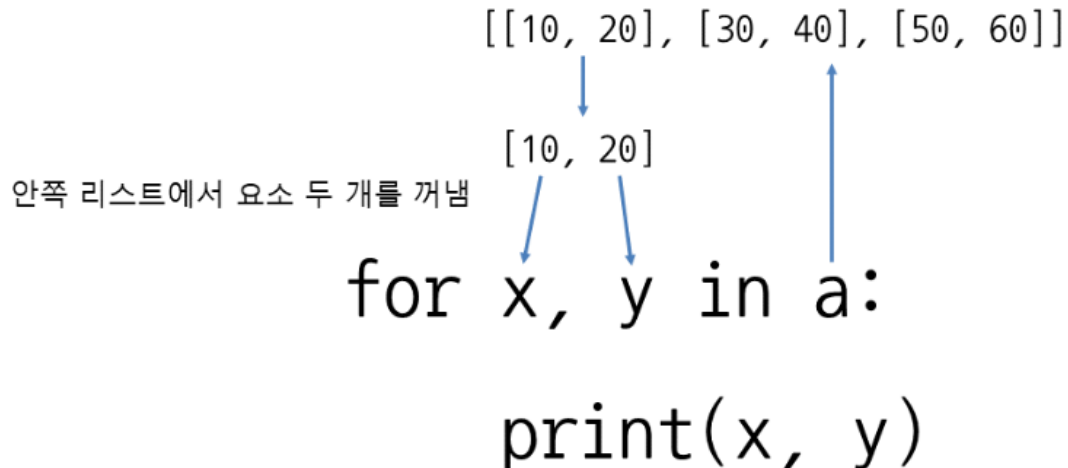
- 리스트[세로인덱스][가로인덱스]
- 리스트[세로인덱스][가로인덱스] = 값

```
>>> a = [[10, 20], [30, 40], [50, 60]]
>>> a[0][0]          # 세로 인덱스 0, 가로 인덱스 0인 요소 출력
10
>>> a[1][1]          # 세로 인덱스 1, 가로 인덱스 1인 요소 출력
40
>>> a[2][1]          # 세로 인덱스 2, 가로 인덱스 0인 요소 출력
60
>>> a[0][1] = 1000    # 세로 인덱스 0, 가로 인덱스 1인 요소에 값 할당
>>> a[0][1]
1000
```



- 반복문으로 2차원 리스트의 요소를 모두 출력하기

```
>>> a = [[10, 20], [30, 40], [50, 60]]
>>> for x, y in a:    # 리스트의 가로 한 줄(안쪽 리스트)에서 요소 두 개를 꺼냄
...     print(x, y)
...
10 20
30 40
50 60
```



- for 반복문을 두 번 사용하기

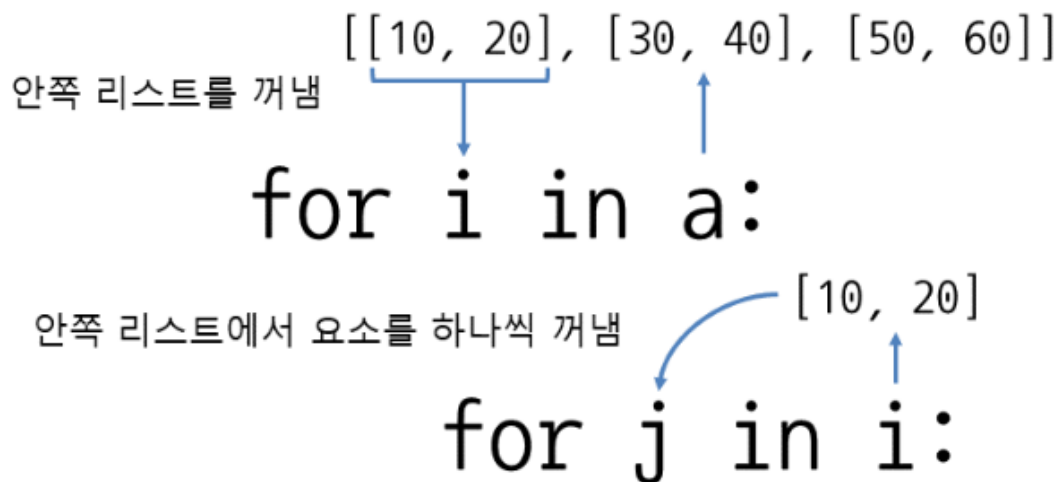
two_dimensional_list_for_for.py

```
a = [[10, 20], [30, 40], [50, 60]]

for i in a:      # a에서 안쪽 리스트를 꺼냄
    for j in i:  # 안쪽 리스트에서 요소를 하나씩 꺼냄
        print(j, end=' ')
    print()
```

실행 결과

```
10 20
30 40
50 60
```



- for와 range 사용하기

two_dimensional_list_for_for_range.py

```
a = [[10, 20], [30, 40], [50, 60]]

for i in range(len(a)):          # 세로 크기
    for j in range(len(a[i])):    # 가로 크기
        print(a[i][j], end=' ')
    print()
```

실행 결과

```
10 20
30 40
50 60
```

- len으로 2차원 리스트 a의 크기를 구하면 리스트 안에 들어있는 모든 요소의 개수가 아니라 안쪽 리스트의 개수(세로 크기)가 나온다는 점

```
for i in range(len(a)):          # 세로 크기
    for j in range(len(a[i])):    # 가로 크기
```

- 요소에 접근할 때는 리스트[세로인덱스][가로인덱스] 형식으로 접근함
- 세로 인덱스에 변수 i를, 가로 인덱스에 변수 j를 지정해줌

```
print(a[i][j], end=' ')
```

- while 반복문을 한 번 사용하기

two_dimensional_list_while.py

```
a = [[10, 20], [30, 40], [50, 60]]

i = 0
while i < len(a):    # 반복할 때 리스트의 크기 활용(세로 크기)
    x, y = a[i]      # 요소 두 개를 한꺼번에 가져오기
    print(x, y)
    i += 1           # 인덱스를 1 증가시킴
```

실행 결과

```
10 20
30 40
50 60
```

- while 반복문을 사용할 때도 리스트의 크기를 활용하면 편리함

```
i = 0
while i < len(a):    # 반복할 때 리스트의 크기 활용(세로 크기)
```

- 리스트에 인덱스를 지정하여 값을 꺼내 올 때는 다음과 같이 변수 두 개를 지정해주면 가로 한 줄(안쪽 리스트)에서 요소 두 개를 한꺼번에 가져올 수 있음

```
x, y = a[i]
```


- while 반복문을 두 번 사용하기

two_dimensional_list_while_while.py

```
a = [[10, 20], [30, 40], [50, 60]]

i = 0
while i < len(a):          # 세로 크기
    j = 0
    while j < len(a[i]):    # 가로 크기
        print(a[i][j], end=' ')
        j += 1             # 가로 인덱스를 1 증가시킴
    print()
    i += 1                 # 세로 인덱스를 1 증가시킴
```

실행 결과

```
10 20
30 40
50 60
```

```
i = 0
while i < len(a):          # 세로 크기
    j = 0
    while j < len(a[i]):    # 가로 크기
```

- 요소에 접근할 때는 리스트[세로인덱스][가로인덱스] 형식으로 접근함
- 세로 인덱스에 변수 i를, 가로 인덱스에 변수 j를 지정해줌

```
print(a[i][j], end=' ')
```

- while 반복문을 두 번 사용하기

list_create.py

```
a = []    # 빈 리스트 생성

for i in range(10):
    a.append(0)    # append로 요소 추가

print(a)
```

실행 결과

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- for 반복문으로 10번 반복하면서 append로 요소를 추가하면
1차원 리스트 생성가능

- for 반복문으로 2차원 리스트 만들기

two_dimensional_list_create.py

```
a = []    # 빈 리스트 생성

for i in range(3):
    line = []        # 안쪽 리스트로 사용할 빈 리스트 생성
    for j in range(2):
        line.append(0)    # 안쪽 리스트에 0 추가
    a.append(line)        # 전체 리스트에 안쪽 리스트를 추가

print(a)
```

실행 결과

```
[[0, 0], [0, 0], [0, 0]]
```

- 먼저 세로 크기만큼 반복하면서 안쪽 리스트로 사용할 빈 리스트 line을 생성

```
for i in range(3):
    line = []        # 안쪽 리스트로 사용할 빈 리스트 생성
```

```
    for j in range(2):
        line.append(0)    # 안쪽 리스트에 0 추가
    a.append(line)        # 전체 리스트에 안쪽 리스트를 추가
```

- append에 리스트를 넣으면 리스트 안에 리스트가 들어가는 특성을 이용함

- 리스트 표현식으로 2차원 리스트 만들기

```
>>> a = [[0 for j in range(2)] for i in range(3)]  
>>> a  
[[0, 0], [0, 0], [0, 0]]
```

- for 반복문을 한 번만 사용하고 싶다면 다음과 같이 식 부분에서 리스트 자체를 곱해주면 됨

```
>>> a = [[0] * 2 for i in range(3)]  
>>> a  
[[0, 0], [0, 0], [0, 0]]
```

- 톱니형 리스트 만들기

- 가로 크기를 알고 있다고 가정하고, 리스트를 만들어보자

jagged_list_create.py

```
a = [3, 1, 3, 2, 5]    # 가로 크기를 저장한 리스트
b = []                # 빈 리스트 생성

for i in a:            # 가로 크기를 저장한 리스트로 반복
    line = []          # 안쪽 리스트로 사용할 빈 리스트 생성
    for j in range(i):  # 리스트 a에 저장된 가로 크기만큼 반복
        line.append(0)
    b.append(line)      # 리스트 b에 안쪽 리스트를 추가

print(b)
```

실행 결과

```
[[0, 0, 0], [0], [0, 0, 0], [0, 0], [0, 0, 0, 0, 0]]
```

- 리스트 표현식 활용

```
>>> a = [[0] * i for i in [3, 1, 3, 2, 5]]
>>> a
[[0, 0, 0], [0], [0, 0, 0], [0, 0], [0, 0, 0, 0, 0]]
```

- 2차원 리스트의 할당과 복사 알아보기

- 2차원 리스트를 만든 뒤 다른 변수에 할당하고, 요소를 변경해보면
두 리스트에 모두 반영됨

```
>>> a = [[10, 20], [30, 40]]
>>> b = a
>>> b[0][0] = 500
>>> a
[[500, 20], [30, 40]]
>>> b
[[500, 20], [30, 40]]
```

- 리스트 a를 copy 메서드로 b에 복사한 뒤 b의 요소를 변경해보면
리스트 a와 b에 모두 반영됨

```
>>> a = [[10, 20], [30, 40]]
>>> b = a.copy()
>>> b[0][0] = 500
>>> a
[[500, 20], [30, 40]]
>>> b
[[500, 20], [30, 40]]
```

- 2차원 리스트의 할당과 복사 알아보기
 - 2차원 이상의 다차원 리스트는 리스트를 완전히 복사하려면 copy 메서드 대신 copy 모듈의 **deepcopy** 함수를 사용해야 함

```
>>> a = [[10, 20], [30, 40]]
>>> import copy          # copy 모듈을 가져옴
>>> b = copy.deepcopy(a)  # copy.deepcopy 함수를 사용하여 깊은 복사
>>> b[0][0] = 500
>>> a
[[10, 20], [30, 40]]
>>> b
[[500, 20], [30, 40]]
```

- 반복문으로 딕셔너리의 키-값 쌍을 모두 출력하기

```
>>> x = {'a': 10, 'b': 20, 'c': 30, 'd': 40}
>>> for i in x:
...     print(i, end=' ')
...
a b c d
```

- for i in x:처럼 for 반복문에 딕셔너리를 지정한 뒤에 print로 변수 i를 출력해보면 값은 출력되지 않고 키만 출력됨

```
for 키, 값 in 딕셔너리.items():
    반복할 코드
```

```
>>> x = {'a': 10, 'b': 20, 'c': 30, 'd': 40}
>>> for key, value in x.items():
...     print(key, value)
...
a 10
b 20
c 30
d 40
```

```
for key, value in {'a': 10, 'b': 20, 'c': 30, 'd': 40}.items():
    print(key, value)
```


- 딕셔너리의 키만 출력하기

- items로 키와 값을 함께 가져왔는데,
키만 가져오거나 값만 가져오면서 반복할 수도 있음

items: 키-값 쌍을 모두 가져옴
keys: 키를 모두 가져옴
values: 값을 모두 가져옴

- for 반복문에서 keys로 키를 가져오면서 반복

```
>>> x = {'a': 10, 'b': 20, 'c': 30, 'd': 40}
>>> for key in x.keys():
...     print(key, end=' ')
...
a b c d
```

- for 반복문에서 values를 사용하면 값만 가져오면서 반복

```
>>> x = {'a': 10, 'b': 20, 'c': 30, 'd': 40}
>>> for value in x.values():
...     print(value, end=' ')
...
10 20 30 40
```

- 딕셔너리 표현식 사용하기

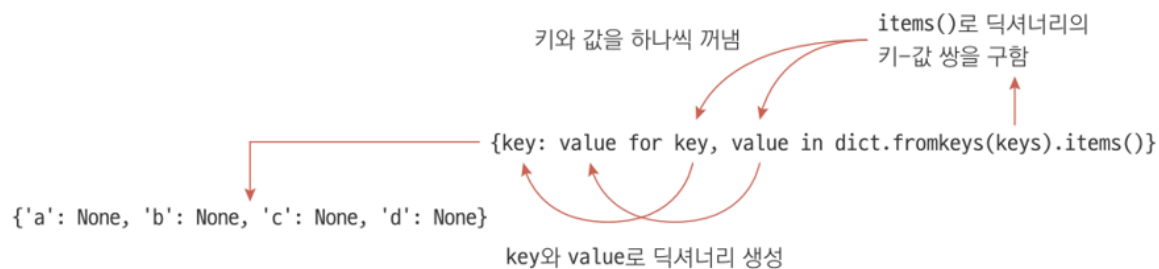
- 리스트와 마찬가지로 딕셔너리도 for 반복문과 if 조건문을 사용하여 딕셔너리를 생성할 수 있음

- {키: 값 for 키, 값 in 딕셔너리}
- dict({키: 값 for 키, 값 in 딕셔너리})

```
>>> keys = ['a', 'b', 'c', 'd']
>>> x = {key: value for key, value in dict.fromkeys(keys).items()}
>>> x
{'a': None, 'b': None, 'c': None, 'd': None}
```

- 딕셔너리 표현식을 사용할 때는 for in 다음에 딕셔너리를 지정하고 items를 사용
- 키, 값을 가져온 뒤에는 키: 값 형식으로 변수나 값을 배치하여 딕셔너리를 생성됨

```
x = {key: value for key, value in dict.fromkeys(keys).items()}
```



- 딕셔너리 표현식 사용하기

- keys로 키만 가져온 뒤 특정 값을 넣거나, values로 값을 가져온 뒤 값을 키로 사용할 수도 있음

```
>>> {key: 0 for key in dict.fromkeys(['a', 'b', 'c', 'd']).keys()}           # 키만 가져옴
{'a': 0, 'b': 0, 'c': 0, 'd': 0}
>>> {value: 0 for value in {'a': 10, 'b': 20, 'c': 30, 'd': 40}.values()} # 값을 키로 사용
{10: 0, 20: 0, 30: 0, 40: 0}
```

- 키와 값의 자리를 바꾸는 등 여러 가지로 응용할 수 있음

```
>>> {value: key for key, value in {'a': 10, 'b': 20, 'c': 30, 'd': 40}.items()} # 키-값 자리를 바꿈
{10: 'a', 20: 'b', 30: 'c', 40: 'd'}
```

- 딕셔너리 표현식에서 if 조건문 사용하기

- 딕셔너리 표현식은 딕셔너리에서 특정 값을 찾아서 삭제할 때 유용함
- 딕셔너리는 특정 키를 삭제하는 pop 메서드만 제공할 뿐 특정 값을 삭제하는 메서드는 제공하지 않음
- for 반복문으로 반복하면서 del로 삭제하는 방식을 떠올릴 수 있음

dict_del_by_value_error.py

```
x = {'a': 10, 'b': 20, 'c': 30, 'd': 40}

for key, value in x.items():
    if value == 20:      # 값이 20이면
        del x[key]      # 키-값 쌍 삭제

print(x)
```

실행 결과

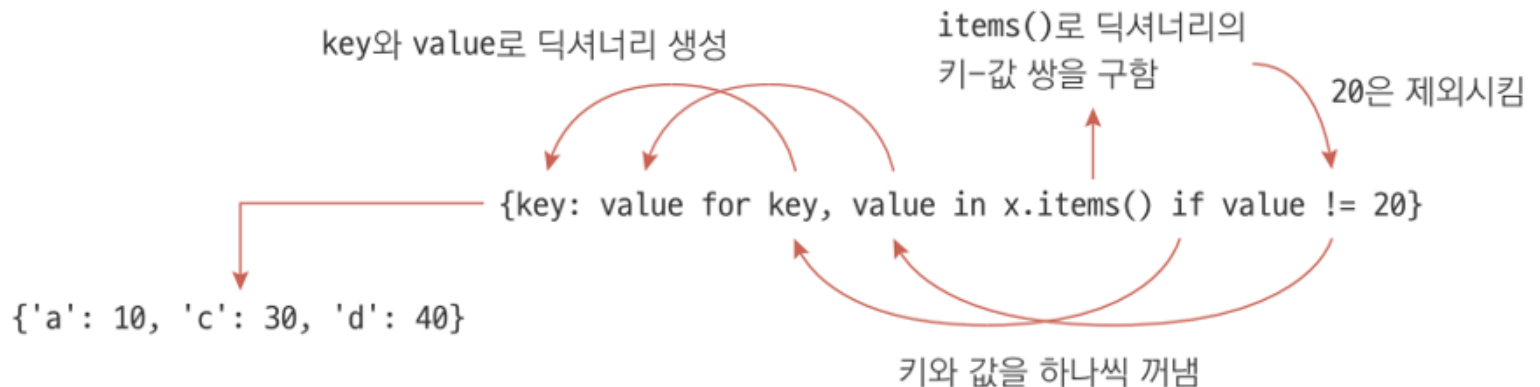
```
Traceback (most recent call last):
  File "C:\project\dict_del_by_value_error.py", line 3, in <module>
    for key, value in x.items():
RuntimeError: dictionary changed size during iteration
```

- 딕셔너리 표현식에서 if 조건문 사용하기

- 별 문제 없이 잘 삭제될 것 같지만 반복 도중에 딕셔너리의 크기가 바뀌었다는 에러가 발생함
- 딕셔너리는 for 반복문으로 반복하면서 키-값 쌍을 삭제하면 안 됨
- 딕셔너리 표현식에서 if 조건문을 사용하여 삭제할 값을 제외하면 됨

- {키: 값 for 키, 값 in 딕셔너리 if 조건식}
- dict({키: 값 for 키, 값 in 딕셔너리 if 조건식})

```
>>> x = {'a': 10, 'b': 20, 'c': 30, 'd': 40}
>>> x = {key: value for key, value in x.items() if value != 20}
>>> x
{'a': 10, 'c': 30, 'd': 40}
```



- 딕셔너리의 for 문 예

```
>>> snacks={'새우깡' : 700, '홈런볼' : 1000, '꼬깔콘' : 1500}
>>> for i in snacks:
    print(i, end = ' ')
```

새우깡 홈런볼 꼬깔콘

```
>>> for i in snacks:
    snacks[i] += 200
```

```
>>> for key, value in snacks.items():
    print (key, value, sep=' ', end = ' ')
```

새우깡 900 홈런볼 1200 꼬깔콘 1700

```
>>> snacks
{'새우깡': 900, '홈런볼': 1200, '꼬깔콘': 1700}
```

```
>>> from pprint import pprint
>>> pprint(snacks)
{'꼬깔콘': 1700, '새우깡': 900, '홈런볼': 1200}
```

```
snacks={'새우깡' : 700, '홈런볼' : 1000, '꼬깔콘' : 1500}
```

```
for key, value in snacks.items():
    if value == 1500:
        del snacks[key]
```

```
print(snacks)
```

Traceback (most recent call last):

File "D:/private/python/test0509_dic.py", line 3, in <module>
for key, value in snacks.items():

RuntimeError: dictionary changed size during iteration

반복 중에 딕셔너리 크기가 바뀜

```
snacks={'새우깡' : 700, '홈런볼' : 1000, '꼬깔콘' : 1500}
```

```
snacks={key:value for key,value in snacks.items() if value !=1500}
print(snacks)
```

```
{'새우깡' : 700, '홈런볼' : 1000}
```

- 딕셔너리 안에서 딕셔너리 사용하기

- 딕셔너리 = {키1: {키A: 값A}, 키2: {키B: 값B}}

- 예를 들어 지구형 행성의 반지름, 질량, 공전주기를 딕셔너리로 표현해보자

dict_dict.py

```
terrestrial_planet = {  
    'Mercury': {  
        'mean_radius': 2439.7,  
        'mass': 3.3022E+23,  
        'orbital_period': 87.969  
    },  
    'Venus': {  
        'mean_radius': 6051.8,  
        'mass': 4.8676E+24,  
        'orbital_period': 224.70069,  
    },  
    'Earth': {  
        'mean_radius': 6371.0,  
        'mass': 5.97219E+24,  
        'orbital_period': 365.25641,  
    },  
    'Mars': {  
        'mean_radius': 3389.5,  
        'mass': 6.4185E+23,  
        'orbital_period': 686.9600,  
    }  
}  
  
print(terrestrial_planet['Venus']['mean_radius'])    # 6051.8
```

실행 결과

6051.8

- 딕셔너리 안에서 딕셔너리 사용하기
 - 중첩 딕셔너리는 계층형 데이터를 저장할 때 유용함
 - 딕셔너리[키][키]
 - 딕셔너리[키][키] = 값
 - 딕셔너리가 두 단계로 구성되어 있으므로 대괄호를 두 번 사용함
 - 금성(Venus)의 반지름(mean radius)를 출력하려면
먼저 'Venus'를 찾아가고 다시 'mean_radius'의 값을 가져오면 됨

```
print(terrestrial_planet['Venus']['mean_radius'])    # 6051.8
```


- 딕셔너리의 할당과 복사

- 딕셔너리를 만든 뒤 다른 변수에 할당함

```
>>> x = {'a': 0, 'b': 0, 'c': 0, 'd': 0}
>>> y = x
```

- 변수 이름만 다를 뿐 딕셔너리 x와 y는 같은 객체임

```
>>> x is y
True
```

- x와 y는 같으므로 y['a'] = 99와 같이 키 'a'의 값을 변경하면
딕셔너리 x와 y에 모두 반영됨

```
>>> y['a'] = 99
>>> x
{'a': 99, 'b': 0, 'c': 0, 'd': 0}
>>> y
{'a': 99, 'b': 0, 'c': 0, 'd': 0}
```

- 딕셔너리의 할당과 복사

- 딕셔너리 x와 y를 완전히 두 개로 만들려면 copy 메서드로 모든 키-값 쌍을 복사해야 함

```
>>> x = {'a': 0, 'b': 0, 'c': 0, 'd': 0}
>>> y = x.copy()
```

```
>>> x is y
False
>>> x == y
True
```

- 딕셔너리 x와 y는 별개이므로 한쪽의 값을 변경해도 다른 딕셔너리에 영향을 미치지 않음
- 다음과 같이 딕셔너리 y에서 키 'a'의 값을 변경

```
>>> y['a'] = 99
>>> x
{'a': 0, 'b': 0, 'c': 0, 'd': 0}
>>> y
{'a': 99, 'b': 0, 'c': 0, 'd': 0}
```

- 중첩 딕셔너리의 할당과 복사 알아보기

- 중첩 딕셔너리를 만든 뒤 copy 메서드로 복사함

```
>>> x = {'a': {'python': '2.7'}, 'b': {'python': '3.6'}}
>>> y = x.copy()
```

- 이제 y['a']['python'] = '2.7.15'와 같이 y의 값을 변경해보면 x와 y에 모두 반영됨

```
>>> y['a']['python'] = '2.7.15'
>>> x
{'a': {'python': '2.7.15'}, 'b': {'python': '3.6'}}
>>> y
{'a': {'python': '2.7.15'}, 'b': {'python': '3.6'}}
```

- 중첩 딕셔너리의 할당과 복사 알아보기

- 중첩 딕셔너리를 완전히 복사하려면 copy 메서드 대신
copy 모듈의 deepcopy 함수를 사용해야 함

```
>>> x = {'a': {'python': '2.7'}, 'b': {'python': '3.6'}}
>>> import copy                # copy 모듈을 가져옴
>>> y = copy.deepcopy(x)       # copy.deepcopy 함수를 사용하여 깊은 복사
>>> y['a']['python'] = '2.7.15'
>>> x
{'a': {'python': '2.7'}, 'b': {'python': '3.6'}}
>>> y
{'a': {'python': '2.7.15'}, 'b': {'python': '3.6'}}
```

- 딕셔너리 y의 값을 변경해도 딕셔너리 x에는 영향을 미치지 않음
- copy.deepcopy 함수는 중첩된 딕셔너리에 들어있는 모든 딕셔너리를 복사하는 깊은 복사(deep copy)를 해줌