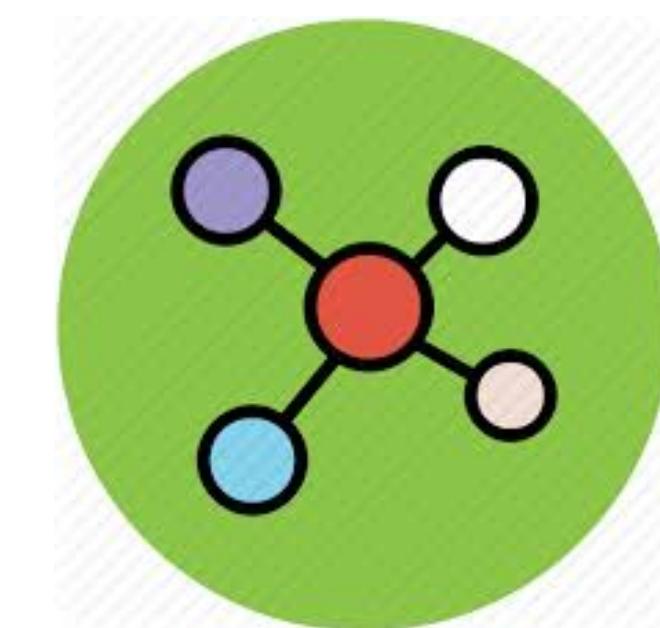




# 복합자료형

## Compound Data Types



박진수 교수

서울대학교·경영대학

jinsoo@snu.ac.kr



# 학습 목차

시퀀스형

▶ 리스트형

▶ 튜플형

매핑형

▶ 딕셔너리형

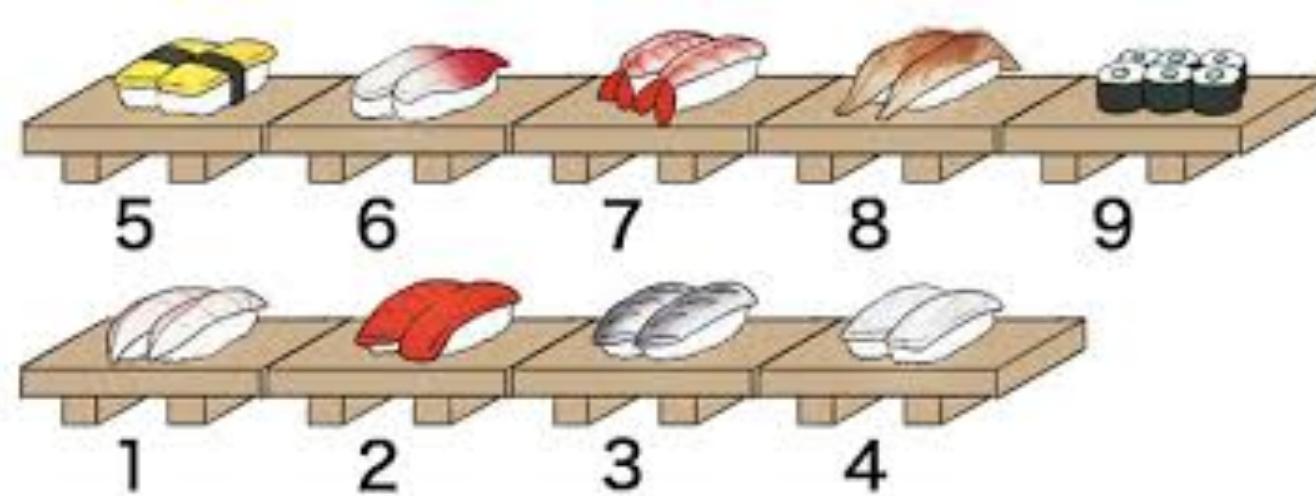
집합형

▶ 세트형

# 시퀀스형

---

## Sequence Types





## ● 시퀀스형(sequence types)이란?

- 0개 이상의 객체를 배열 형태로 참조하는 자료형(즉, 담고 있는 각 객체는 순서가 있다)
- 순회형(**iterable**)
  - 시퀀스형은 순회(iterative)할 수 있기 때문에 순회가 가능한 자료형(**iterable**)이며 다음과 같은 기능을 지원
    - 분할([ ]), 멤버십 연산자(**in**), 크기 함수(**len()**) 등
- 가장 많이 사용하는 내장(built-in) 시퀀스형
  - 문자열(**str**)
  - 리스트(**list**)
  - 튜플(**tuple**)
- 기타 시퀀스형
  - 내장 시퀀스형 : **bytearray**, **bytes**
  - 표준 라이브러리 : **collections.namedtuple**



# 리스트형



## List Type





# 리스트형

## ● 리스트(list)란?

### ● 가변자료형(mutable)

- ▶ 생성한 후 담고 있는 내용을 변경할 수 있다(즉, 담고 있는 객체를 삭제, 변경, 삽입하는 것이 가능)
- ▶ 이러한 유연성 때문에 파이썬에서 가장 흔히 사용하는 자료형

### ● 순서를 가지는 0개 이상의 객체를 참조하는 시퀀스형

- ▶ 각 객체는 쉼표( , )로 구분
- ▶ 리스트 자신을 포함해 어떠한 자료형도 담을 수 있다

### ● 출력 형식

- ▶ 리스트는 항상 대괄호([ ]) 형태로 출력





# 리스트 생성

- **[ ]**
  - **[ ]** : 빈(empty) 리스트를 반환
  - 대괄호(**[ ]**) 내에 쉼표(,)로 구분한 객체를 입력
    - 예) **[ x ]**는 단일 객체(singleton) 리스트를 반환
    - 예) **[ x, y, z ]**는 3개의 객체로 구성된 다중 객체 리스트를 반환
- **list()** 생성자
  - **list()** : 빈(empty) 리스트를 반환
  - **list(순회형)** : 순회형(iterable) 객체를 얇은 복사(shallow copy)해서 반환
    - 형변환 클래스(**str()**, **int()**, **float()**)처럼 다른 자료형의 객체를 리스트로 변환할 때 사용하는데 순회형만 리스트로 변환이 가능
      - 순회형 : 문자열, 리스트, 튜플, 세트, 딕셔너리, 순회자(iterator) 객체 등
- 기타 방법 : 고급 기능
  - **리스트 축약(list comprehension)**
    - 예) **[ x for x in 순회형 ]**



## 예시 : 리스트 생성

```
number_list = [1, 0, -5, -1.23, 27]
string_list= list('가나다라마바사')
complex_list = [list(), number_list, ['x', 3, string_list], True, None]

print(number_list)
print(string_list)
print(complex_list)

len(number_list)
len(string_list)
len(complex_list)
```



문자열은 불변자료형이라 생성한 후 내용의 변경이 불가능하다

```
s = 'abcde'  
s[0] = 'A' # a를 A로 바꾼다.
```

리스트는 가변자료형이라 생성한 후 내용의 변경이 가능하다

```
L = ['a', 'b', 'c', 'd', 'e']  
L[0] = 'A' # 'a'를 'A'로 바꾼다.  
print(L)
```



# 리스트 인덱스

문자열 인덱스와  
같은 원리로 작동

```
band = ['보컬', ['기타1', '기타2', '베이스'], '드럼', '키보드']
```

```
band[0]      # 첫 번째 객체를 추출
```

```
band[-1]     # 마지막 객체를 추출
```

```
band[1]      # 두 번째 객체를 추출
```

```
band[1][1]    # 두 번째 객체의 두 번째 객체를 추출
```

```
band[1][-1][0] # 두 번째 객체의 마지막 객체(문자열)의 첫 번째 문자를 추출
```

```
band[99]     # 리스트 범위 밖의 객체를 지정
```



중첩 리스트는 행렬(matrix)과 같은 다차원 배열을 표현할 때 사용할 수 있다

63	25	77	83	76
45	34	25	55	80
74	36	59	68	29

리스트가 담고 있는 각 객체는  
연속적인 인덱스 사용으로 접근할 수 있다

```
matrix = [  
    [63, 25, 77, 83, 76],  
    [45, 34, 25, 55, 80],  
    [74, 36, 59, 68, 29]  
]
```

```
matrix[0]
```

```
matrix[1] [-1]
```

```
matrix[-1] [2]
```



## ● 리스트를 생성하고 수정하기

- 아래의 값을 순서대로 갖는 리스트를 생성
  - 정수 5
  - 문자열 'I love Python'
- 이 리스트 내의 각 객체에 대해 다음과 같은 연산을 하여, 다시 이 리스트에 할당
  - 첫 번째 객체에 100을 곱한다
  - 두 번째 객체는 화이트스페이스를 기준으로 분할
- 그 결과를 출력

- 실행 결과 예시

```
[500, ['I', 'love', 'python']]
```



# 리스트 관련 연산자

- 결합(concatenation) 연산자



+

- 반복 결합 연산자



\*

- 확장 연산자



+ =

- 분할 연산자



[:], [:]

- 삭제 연산자



**del**

- 객체에 대한 참조를 삭제하는 연산자(i.e., 식별자를 삭제)

- 멤버십 연산자



**in / not in**





# 리스트 결합/반복 결합 연산자

## • + (결합) 연산자

- 두 리스트를 결합(concatenation)
- 즉, 두 대상 리스트를 붙인다

```
[1, 2, 3] + [4, 5, 6]
```

```
['a', 'b'] + ['c', 'd']
```

## • \* (반복 결합) 연산자

- 문자열의 반복 결합 연산자와 같은 원리
- 즉, 횟수만큼 리스트를 반복해서 연결

```
[0] * 5
```

```
[1, 2, 3] * 3
```



# 리스트 확장 연산자

## += 연산자

- 두 리스트 또는 리스트에 순회형을 병합해서 확장
- += 의 우변에는 **순회형**이 와야 한다
  - 순회형(**iterable**) : 문자열, 리스트, 튜플, 세트, 딕셔너리, 순회자(**iterator**) 객체 등
- extend(순회형)** 메소드도 같은 기능을 수행

▶ 뒤에 나오는 <객체 추가 vs. 객체 확장> 참조

```
list1 = ['프로도', '네오']
list2 = ['라이언', '콘']
list1 += list2
print(list1)
```

```
list1 += 307
```

```
list1 += [307]
print(list1)
```

```
list1 += '제이지'
print(list1)
```

```
list1[-3:] = []
list1 += ['제이지']
print(list1)
```



# 리스트 분할 연산자

## [:, [:]]

- 리스트 분할(슬라이스, slice)을 통해 리스트의 객체들을 추출
- 문자열 분할 연산자와 같은 기능을 수행

```
L = ['a', 'b', 'c', 'd', ['x', 'y', 'z'], 'e', 'f']
```

```
L[:]
```

```
L[:3]
```

```
L[-3:]
```

```
L[2:4] = ['가', '나']
print(L)
```

```
L[1:2] = ['l', 'm', 'n'] # 분할 연산자
```

```
print(L)
```

```
L[1] = ['p', 'q'] # 인덱스
print(L)
```

```
L[1:-1] = ['다', '라']
print(L)
```

```
L[:] = []
print(L)
```



# 리스트 객체 삭제 연산자

## • **del** 객체

- 리스트의 객체(요소) 한 개를 삭제
- 리스트의 객체(요소) 여러 개를 삭제
- 리스트의 전체 객체(요소)를 삭제

```
L = ['a', 'b', 'c', 'd', ['x', 'y', 'z'], 'e', 'f']
```

```
del L[4] [-1] # 다섯 번째 객체의 마지막 객체를 삭제  
print(L)
```

```
del L[4]           # 다섯 번째 객체를 삭제  
print(L)
```

```
del L[1:3]         # 두 번째와 세 번째 객체를 삭제  
print(L)
```

```
del L[:]           # 리스트의 모든 객체를 삭제  
print(L)
```

- **del**문은 객체에 대한 참조를 삭제하는 연산자기 때문에  
리스트 객체 뿐만 아니라 변수 등 파이썬에서 사용하는  
모든 자료형의 객체참조를 삭제하는데 사용



# 멤버십 연산자

## ● **in** 연산자

- 특정 객체가 속해 있는지 존재 여부를 확인
  - 속해 있다면 '참'(True)을 반환

## ● **not in** 연산자

- 특정 객체가 속해 있지 않다면 '참'(True)을 반환

```
L = ['a', 'b', ['c', 'd', 'e']]
```

```
'a' in L
```

```
'c' in L
```

```
'c' in L[-1]
```

```
'b' not in L
```

```
'f' not in L
```



# 특정 단어/문자 검색 : 문자열

## ● 멤버십 연산자

- 멤버십 연산자는 문자열에서 부분문자열(substring)이 포함되어 있는지 여부를 확인하고 싶을 때도 사용 가능
- 💡 문자열, 리스트, 튜플, 딕셔너리, 세트등 순회형(iterable)형에서 데이터 전체를 순회하면서 작업 할 때 **for**문을 사용할 수 있는데 이 때도 멤버십 연산자 사용
- i.e., **for ... in** 순회형



## 선형탐색(linear search)

- in** 연산자는 **선형탐색** 방식이므로 리스트(**list**)나 튜플(**tuple**)의 크기가 클 경우 처리 속도가 느린다
- 하지만 딕셔너리(**dict**)나 세트(**set**)의 경우 크기에 상관없이 처리 속도가 매우 빠르다

```
obama = "Don't just play on your phone, program it"
```

```
'j' in obama
```

```
'd' in obama
```

```
'program' in obama
```

```
python = '맛있는 파이썬을 즐거운 마음으로 배우자'
```

```
'즐' not in python
```

```
'파이썬' in python
```



## ● 리스트 연산자 사용하기

- 다음과 같은 값을 가진 2차원 리스트를 생성
  - `[[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15], [16, 17, 18, 19, 20]]`
- 해당 리스트에서 짝수를 모두 제거
- 그 후, 2차원 리스트 내부에 있는 리스트를 모두 결합하여 1차원 리스트 형태로 출력

## ● 실행 결과 예시

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```



문법	설명
<code>L.copy()</code>	▶ 리스트 <code>L</code> 의 얕은 복사본(shallow copy)을 반환
<code>L.append(x)</code>	▶ 객체 <code>x</code> 를 리스트 <code>L</code> 의 끝에 추가
<code>L.extend(m)</code> <code>L += m</code>	▶ 순회형 <code>m</code> 의 모든 객체들을 하나씩 리스트 <code>L</code> 의 끝에 추가 ▶ <code>+=</code> 연산자도 같은 기능을 수행
<code>L.insert(i, x)</code>	▶ 객체 <code>x</code> 를 리스트 <code>L</code> 의 인덱스 <code>i</code> 에 삽입
<code>L.pop()</code> <code>L.pop(i)</code>	▶ 리스트 <code>L</code> 의 가장 오른쪽에 있는 객체(마지막 객체)를 반환한 후 삭제 ▶ 리스트 <code>L</code> 의 인덱스 <code>i</code> 에 있는 객체를 반환한 후 삭제 ▶ 빈 리스트거나 설정한 인덱스 번호가 리스트 범위 밖이면 <code>IndexError</code> 가 발생
<code>L.remove(x)</code>	▶ 리스트 <code>L</code> 의 가장 왼쪽에 있는 객체(첫번째 객체) <code>x</code> 를 삭제 ▶ 객체 <code>x</code> 가 존재하지 않으면 <code>ValueError</code> 가 발생



문법	설명
<code>L.count(x)</code>	▶ 객체 <code>x</code> 가 리스트 <code>L</code> 에 몇 개 있는지 그 개수를 반환
<code>L.index(x, 시작, 끝)</code>	<ul style="list-style-type: none"><li>▶ 리스트 <code>L</code>에 있는 객체 <code>x</code> 중 가장 왼쪽에 있는 객체 <code>x</code>의 인덱스를 반환</li><li>▶ 리스트 <code>L</code>의 <code>시작</code>과 <code>끝</code> 인덱스가 정해졌다면, 그 인덱스 범위 안에 있는 객체 <code>x</code> 중 가장 왼쪽에 있는 객체 <code>x</code>의 인덱스를 반환(분할 연산자처럼 <code>시작</code>부터 <code>끝</code> <u>바로 앞</u>까지만 검색)</li><li>▶ 객체 <code>x</code>가 없으면 <code>ValueError</code> 가 발생</li></ul>
<code>L.reverse()</code>	<ul style="list-style-type: none"><li>▶ 리스트 <code>L</code>에 있는 객체들의 순서를 거꾸로 뒤집는다</li></ul>
<code>L.sort(key=None, reverse=False)</code>	<ul style="list-style-type: none"><li>▶ 리스트 <code>L</code>을 정렬</li><li>▶ 이 메소드는 <code>sorted()</code>함수처럼 <code>key</code>와 <code>reverse</code> 매개변수를 선택적으로 사용 가능<ul style="list-style-type: none"><li>- <code>key</code>의 전달인자로 함수를 사용해서 원하는 방식으로 리스트 <code>L</code>을 정렬할 수 있고 <code>reverse</code>의 전달인자가 참(<code>True</code>)이면 정렬이 역순으로 이루어진다</li></ul></li></ul>



*L.copy()* 리스트 *L*의 얕은 복사본(shallow copy)을 반환

```
x = [1, 2, 3, ['x', 'y', 'z']]  
y = x.copy() # 리스트 x를 얕은 복사해서 변수 y에 할당  
print(y)
```



# 리스트 추가 메소드

*L.append(x)* 객체 *x*를 리스트 *L*의 끝에 추가

```
L = []
L.append('a')
L.append('b')
print(L)
L.append(['d', 'e'])
print(L)
```

*L.extend(m)* 순회형 *m*의 모든 객체들을 하나씩 리스트 *L*의 끝에 추가  
*L += m*

```
L.extend(['f', 'g'])
print(L)
```

*L.insert(i,x)* 객체 *x*를 리스트 *L*의 인덱스 *i*에 삽입

```
L.insert(4, 'b')
print(L)
```



# 리스트 삭제 메소드

*L.pop()*

- 리스트 *L*의 가장 오른쪽에 있는 객체(마지막 객체)를 반환한 후 삭제
- 빈 리스트면 *IndexError* 가 발생

```
print(L)
L.pop()
print(L)

['a', 'b', ['d', 'e'], 'f', 'b', 'g']
```

*L.pop(i)*

- 리스트 *L*의 인덱스 *i*에 있는 객체를 반환한 후 삭제
- 빈 리스트거나 설정한 인덱스 번호가 리스트 범위 밖이면 *IndexError* 가 발생

```
L.pop(2)
print(L)
L.pop(4)
```

*L.remove(x)*

- 리스트 *L*의 가장 왼쪽에 있는 객체(첫 번째 객체) *x*를 삭제
- 객체 *x*가 존재하지 않으면 *ValueError* 가 발생

```
L.remove('b')
print(L)
L.remove('c')
```



# 리스트 질의 메소드

`L.count(x)` 객체 `x`가 리스트 `L`에 몇 개 있는지 그 개수를 반환

```
L = ['a', 'b', ['c', 'd', 'e'], 'f', 'b', 'c', 'g']
```

```
L.count('b')
```

```
L.count('c')
```

```
L.count('e')
```

```
L[2].count('e')
```



# 리스트 질의 메소드

`L.index(x, 시작, 끝)`

- 리스트 `L`에 있는 객체 `x` 중 가장 왼쪽에 있는 객체 `x`의 인덱스를 반환
- 리스트 `L`의 **시작**과 **끝** 인덱스가 정해졌다면, 그 인덱스 범위 안에 있는 객체 `x` 중 가장 왼쪽에 있는 객체 `x`의 인덱스를 반환(분할 연산자처럼 **시작**부터 **끝** 바로 앞까지만 검색)
- 객체 `x`가 없으면 `ValueError` 가 발생

```
L = ['a', 'b', ['c', 'd', 'e'], 'f', 'b', 'c', 'g']
```

```
L.index('b')
```

```
L.index('b', 3)
```

```
L.index('a', 3, 6)
```



# 리스트 정렬 메소드

`L.reverse()` 리스트 `L`에 있는 객체들의 순서를 거꾸로 뒤집는다

```
L = ['a', 'b', ['c', 'd', 'e'], 'f', 'b', 'c', 'g']
L.reverse()
print(L)
```

`L.sort(key=None, reverse=False)`

- 리스트 `L`을 정렬 (기본값은 오름차순)
- 이 메소드는 `sorted()`함수처럼 `key`와 `reverse` 매개변수를 선택적으로 사용 가능
  - `key`의 전달인자로 함수를 사용해서 원하는 방식으로 리스트 `L`을 정렬할 수 있고
  - `reverse`의 전달인자가 참(`True`)이면 정렬이 역순으로 이루어진다

`L.sort()` 왜 오류가 날까?

```
L.pop(4)
print(L)
```

```
L.insert(2, 'M')
print(L)
```

`L.sort()` 왜 대문자 'M'이 정렬한 리스트의 맨 앞에 있을까?



대소문자 구분없이 정렬할 수는 없을까?

```
print(L)
```

```
L.sort(key=str.lower)  
print(L)
```

```
L.sort(reverse=True)  
print(L)
```

```
L.sort(key=str.lower, reverse=True)  
print(L)
```

- **L.sort()** 메소드는 **sorted()** 함수처럼 **key**와 **reverse** 매개변수를 선택적으로 사용 가능
  - ⦿ **key**의 전달인자로 함수를 사용해서 원하는 방식으로 리스트 **L**을 정렬할 수 있고
  - ⦿ **reverse**의 전달인자가 참(**True**)이면 정렬이 역순으로 이루어진다



# 정렬 방법 : `sort()` 메소드 vs. `sorted()` 함수

`L.sort()` 메소드는 원본 리스트 `L` 자체를 정렬한다

`sorted()` 함수는 원본 리스트 `L` 자체를 정렬하지 않고, 대신 정렬한 리스트를 반환한다

```
x = ['드럼', '기타', '키보드', '베이스']  
y = sorted(x)
```

```
print(y)
```

```
print(x)
```

```
x.sort()  
print(x)
```



# 정렬 방법 : **sort()** 메소드 vs. **sorted()** 함수

리스트 *L*을 절댓값을 기준으로 내림차순 정렬하려면?

```
L = [-5, 0, 9, 1, 3, -12]
```



## ● 리스트 메소드 사용하기

- 사용자로부터 네 개의 문자열을 입력받는다
- 각 문자열에서 소문자 'a'의 개수를 세어 아래 예시와 같이 중첩 리스트를 구성
  - 예를 들어, 사용자가 순서대로 'abcd', 'ababab', 'abba', 'bcd'를 입력하면,
  - 중첩 리스트 [[ 'abcd', 1], ['ababab', 3], ['abba', 2], ['bcd', 0]]를 만든다
- 그 결과를 출력

문자열 **count** 메소드를 사용하면 문자열에서 특정 문자의 개수를 확인할 수 있다

- 예) 'abacdac'.count('ac') # 결과 : 2

## ● 실행 결과 예시

```
첫 번째 문자열을 입력하세요: I love Python
두 번째 문자열을 입력하세요: Append adds an item to the list
세 번째 문자열을 입력하세요: What animal is it?
네 번째 문자열을 입력하세요: Abracadabra
[['I love Python', 0], ['Append adds an item to the list', 2], ['What animal is it?', 3], ['Abracadabra', 4]]
```



# 튜플형



## Tuple Type



## ● 튜플(tuple)이란?

### ● 불변자료형(immutable)

- 생성한 후 담고 있는 내용을 변경할 수 없다(즉, 담고 있는 객체를 삭제, 변경, 삽입하는 것이 불가능)
- 순서가 있는 시퀀스형 자료의 객체를 변경할 필요가 있으면 처음부터 `list`를 사용하거나 `list()`생성자를 사용해 튜플을 리스트로 변환

### ● 순서를 가지는 0개 이상의 객체를 참조하는 시퀀스형

- 각 객체는 쉼표( , )로 구분
- 튜플 자신을 포함해 어떠한 자료형도 담을 수 있다

### ● 출력 형식

- 튜플은 항상 소괄호(()) 형태로 출력
- 생성할 때는 소괄호 없이도 생성이 가능





# 튜플 생성

- ()
  - ◎ () : 빈(empty) 튜플을 반환
  - ◎ 소괄호(()) 내에 쉼표(,)로 구분한 객체를 입력
    - 예) (x,)는 단일 객체 singleton 튜플을 반환 <— 붙은 쉼표(trailing comma)가 반드시 있어야 한다!!!
      - ◎ 괄호는 생략해도 무방하지만 객체 뒤에 쉼표(,)를 반드시 붙여야 한다
    - 예) (x, y, z)는 3개의 객체로 구성된 다중 객체 튜플을 반환
  - 쉼표(,)로 구분한 값
    - ◎ 소괄호 없이 쉼표(,)로 구분한 객체를 입력
      - 예) x,는 단일 객체 singleton 튜플을 반환 <— 붙은 쉼표(trailing comma)가 반드시 있어야 한다!!!
      - 예) x, y, z는 3개의 객체로 구성된 다중 객체 튜플을 반환
    - ◎ 괄호 없이 쉼표로 구분해서 나열된 객체는 기본적으로 튜플로 간주
  - tuple() 생성자
    - ◎ tuple() : 빈(empty) 튜플을 반환
    - ◎ tuple(순회형) : 순회형 iterable 객체를 얇은 복사(shallow copy)해서 반환
      - 형변환 클래스(str(), int(), float())처럼 다른 자료형의 객체를 튜플로 변환할 때 사용하는데 순회형만 튜플로 변환이 가능
        - ◎ 순회형 : 문자열, 리스트, 튜플, 세트, 딕셔너리, 순회자(iterator) 객체 등



## 예시 : 튜플 생성

```
t1 = (5, -3, 3.14, 'red', '드럼')
t2 = 5, -3, 3.14, 'red', '드럼'
string_tuple= tuple('가나다라마바사')
complex_tuple = False, (), t2, ['x', 3, string_tuple], None

print(type(t1), type(t2), type(string_tuple), type(complex_tuple))
print(len(t1), len(t2), len(string_tuple), len(complex_tuple))

print(t1)
print(t2)
print(string_tuple)
print(complex_tuple)
```



튜플은 불변자료형이라 생성한 후 내용의 변경이 불가능

```
tuple1
```

```
(5, -3, 3.14, 'red', '드럼')
```

```
tuple1[-1] = '기타'
```



튜플은 불변자료형이기 때문에  
튜플이 담고 있는 객체를 변경할 필요가 있다면 `list()` 생성자를 사용하여 튜플을 리스트로 형변환해야 한다  
튜플은 리스트로 형변환이 가능하고 리스트도 튜플로 형변환이 가능

```
T = 1, 2, 3
type(T)
print(T)
```

```
L = list(T)    # 튜플을 리스트로 형변환
type(L)
print(L)
```

```
T = tuple(L)  # 리스트를 튜플로 다시 형변환
type(T)
print(T)
```



튜플은 일단 생성하면 변경할 수 없기 때문에 튜플이 담은 객체에 영향을 미치지 않는 연산자와 메소드만 지원

따라서, 리스트 관련 연산자와 메소드 중 객체를 변경하는 연산자와 메소드를 제외한 나머지는 같다

구체적으로

`del, copy(), append(), insert(), extend(), remove(), pop(), reverse(), sort()` 를 제외한  
나머지 연산자와 메소드는 사용할 수 있다

책을 참고해서 반드시 따라해보세요.

시험 범위에 포함됩니다.



## ● 튜플 연산자 사용하기

- 아래 객체들을 포함하는 튜플을 생성한 후 출력
    - 1, 5, (2, 3), 'green', ['드럼', '기타']
  - 분할 연산자를 사용해 방금 만든 튜플의 마지막 두 객체만 가지고 있는 새로운 튜플을 생성한 후 출력
  - (2,3)이 처음 만든 튜플에 포함되어 있는지 확인 확인
  - 5가 두 번째 만든 튜플에 들어 있는지 확인
- 
- 실행 결과 예시

```
(1, 5, (2, 3), 'green', ['드럼', '기타'])  
('green', ['드럼', '기타'])  
True  
False
```



$$(x, y, z) = (i, j, k)$$
$$x, y, z = i, j, k$$

## ● 튜플 할당(tuple assignment)이란?

- ◎ 등호를 기준으로 우변의 값 혹은 표현식 각각을 좌변의 같은 위치에 해당하는 변수로 할당하는 것
  - 좌변 : 튜플 변수
  - 우변 : 표현식 튜플(문자열과 리스트 같은 시퀀스형 자료도 가능)
- ◎ 원리
  - 좌변의 변수에 대응하는 우변의 값을 할당
  - 우변의 모든 표현식은 할당되기 전에 평가(evaluation) 또는 계산된다
  - 좌변의 변수 개수와 우변의 표현식 값의 개수는 반드시 일치해야 한다
- ◎ 이점
  - 튜플 할당은 매우 편리하고 유용하게 사용할 수 있다
  - 예를 들어, 한 줄에서 여러 개의 변수를 한꺼번에 할당하는 것이 가능해져 코드가 간소해진다
  - 두 변수 간의 값을 서로 바꿀 때(swap)도 유용



## 예시 : 튜플 할당

```
i, j, k = '드럼', '기타', '피아노'  
print(i, j, k)
```

```
# 스왑(swap operation)  
i, j = j, i  
print(i, j, k)
```

```
x, y = 1, 2, 3
```

```
a, b, c = ['엄마', '아빠', ('나', '동생')]  
print(a)  
print(b)  
print(c)
```

```
email = 'abc@xyz.kr'  
user_name, domain = email.split('@')  
print(user_name)  
print(domain)
```



x, y = 1, 2, 3

x, \_, y, \_ = 1, 2, 3, 4

x

y

-



## 튜플 패킹(tuple packing)

`t = x, y, z` 또는 `t = (x, y, z)`

우변의 객체들을 좌변에 있는 변수 하나에 할당하는 것  
이는 마치 박스 하나에 여러 개의 물건을 집어넣는 것과 같다

```
t = 3.14, [5, -3], 'green', ('드럼', '기타')
```

```
type(t)
```

```
print(t)
```



## 튜플 언패킹(tuple unpacking)

```
x, y, z = t
```

또는

```
(x, y, z) = t
```

좌변의 변수 개수 == 우변의 튜플 길이

우변의 패킹한 튜플 변수에서 여러 개의 값을 좌변으로 꺼내온다

즉, 튜플 안의 객체들을 여러 개의 변수에 한 번에 할당

이는 마치 박스에 들어 있는 여러 개의 물건을 꺼내는 것과 같다

```
t = 3.14, [5, -3], 'green', ('드럼', '기타')  
a, b, c, d = t
```

```
print(a)
```

```
print(b)
```

```
print(c)
```

```
print(d)
```



# 시퀀스형 패킹/언패킹 연산자

\*

- 모든 시퀀스형(리스트, 튜플 등)은 패킹/언패킹 연산자 \*를 사용하여 객체의 패킹(packing) 또는 언패킹(unpacking)이 가능(언패킹은 '함수'에서 다룬다)

## 패킹

- 할당문의 좌변 변수에 위치
- 좌변의 변수(반드시 두 개 이상)에 우변의 객체를 할당할 때 우변 객체의 수가 좌변 변수의 수보다 많을 경우, 객체를 순서대로 변수에 할당하는데 나머지 객체 모두는 별 기호 표시된 변수에 리스트로 할당

$x, *y = i, j, k, \dots$

```
x, y, *z = 3.14, [5, -3], 'green', ('드럼', '기타')
```

x

y

z

```
*x, y, z = 3.14, [5, -3], 'green', ('드럼', '기타')
```

x

y

z

```
x, *y, z = 3.14, [5, -3], 'green', ('드럼', '기타')
```

x

y

z



# Lab : 튜플 패킹/언패킹



## ● 튜플 패킹/언패킹 연산자 사용하기

- 학생의 프로그래밍 과목 성적은 다음과 같은 형식의 튜플로 정의
  - 형식 : (이름, 과제1 성적, 중간고사 성적, 과제2 성적, 기말고사 성적, 성별)
  - 값 : ('라이언', 50, 40, 70, 80, 'M')
- 위 값을 가진 튜플을 생성하고, 튜플 언패킹 연산자를 사용하여
  - 변수 **name**에는 이름을, 변수 **gender**에는 성별을, 변수 **score**에는 성적을 리스트로 담는다
- 그 후 다음 실행 결과 예시와 같이 학생의 이름, 성별, 평균 성적을 차례로 출력
- **sum** 함수는 순회형 객체가 담고 있는 숫자들의 합을 반환
  - 예) `sum([1,2,3])` # 결과 : 6
- 실행 결과 예시

이름 :	라이언
성별 :	M
평균 성적 :	60.0



# 언제 튜플을 사용할까?

## 변수의 값이 변하면 안될 경우

- 즉, 시퀀스형 자료를 함수의 전달인자(argument)로 사용해야 하는 경우
- 튜플의 값은 불변성이므로 객체가 변할 가능성이 없기 때문에 가변성 값들 가진 리스트보다 **안전**

## 처리 속도 및 공간

- 튜플을 처리하는 속도가 리스트 처리 속도보다 **빠르다**
- 튜플은 리스트보다 **더 적은 공간**을 사용

## 구문상 더 간단한 경우

- 튜플을 사용하는 것이 리스트보다 구문상 더 간단할 경우
- e.g., **return**문, 튜플 할당 등

## 딕셔너리의 키

- 딕셔너리의 키(key)로 시퀀스형 자료를 사용해야 하는 경우
- 리스트는 딕셔너리의 키로 사용할 수 없다

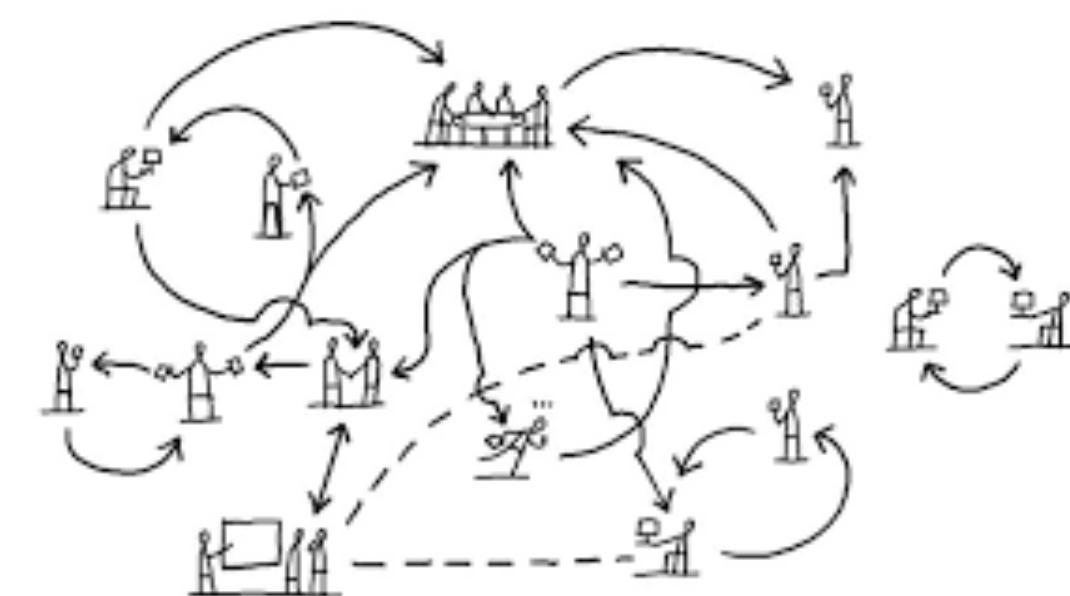
## 참고

- list()** 생성자 : 튜플을 리스트로 변환
- tuple()** 생성자 : 리스트를 튜플로 변환

# 매핑형

---

## Mapping Types





## ● 매핑형(mapping types)이란?

- 순서없이(unordered) 0개 이상의 키-매핑값(key-value) 쌍으로 된 객체를 참조하는 자료형

### ● 키(key)

- 키는 해시 가능한 객체(hashable object)만 사용 가능
- 해시 가능한 자료형 : 그 값이 불변성인 모든 내장 불변자료형(built-in immutable data types)
  - `int`, `float`, `str`, `tuple`, `frozenset` 등
- 딕셔너리의 키로 사용 할 수 없는 해시 가능하지 않는 자료형
  - `list`, `dict`, `set` 등

### ● 매핑값(value)

- 어떠한 형태의 객체든지 상관 없음(즉, 모든 종류의 자료형이 가능)

### ● 순회형(iterator)

## ● 순서없는(unordered) 매핑형

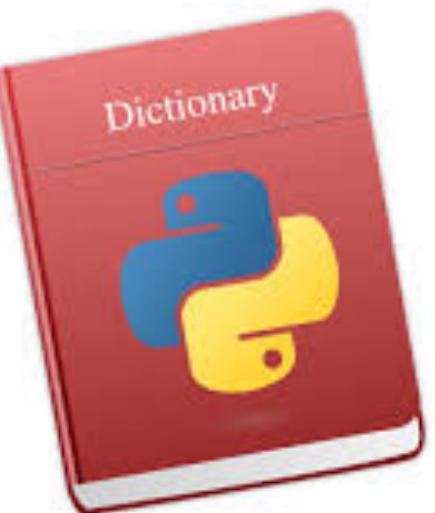
- 딕셔너리(`dict`) : 내장 자료형(built-in data types)
- 디폴트 딕셔너리(`collections.defaultdict`) : 표준 라이브러리

## ● 순서(ordered)가 있는 매핑형(Python 3.1부터 사용 가능)

- 순서형 딕셔너리(`collections.OrderedDict`) : 표준 라이브러리
- 딕셔너리(`dict`)와 동일한 메소드와 속성을 가지고 있지만 객체들을 추가하는 순서대로 저장

# 딕셔너리형

Dictionary Type





# 딕셔너리형

## ● 딕셔너리(dictionary)란?

### ◎ 가변자료형(mutable)

- 담고 있는 객체를 삽입, 변경, 삭제하는 것이 가능

### ◎ 순서없이(unordered) 0개 이상의 키-매핑값(key-value) 쌍으로 된 객체를 참조하는 맵핑형

- 각 객체는 쉼표( , )로 구분
- 키(key) : 해시가능한 객체(불변자료형)만 가능
- 매핑값(value) : 어떠한 타입의 객체도 가능

### ◎ 딕셔너리의 각 객체는 키-매핑값(key-value) 쌍으로 구성

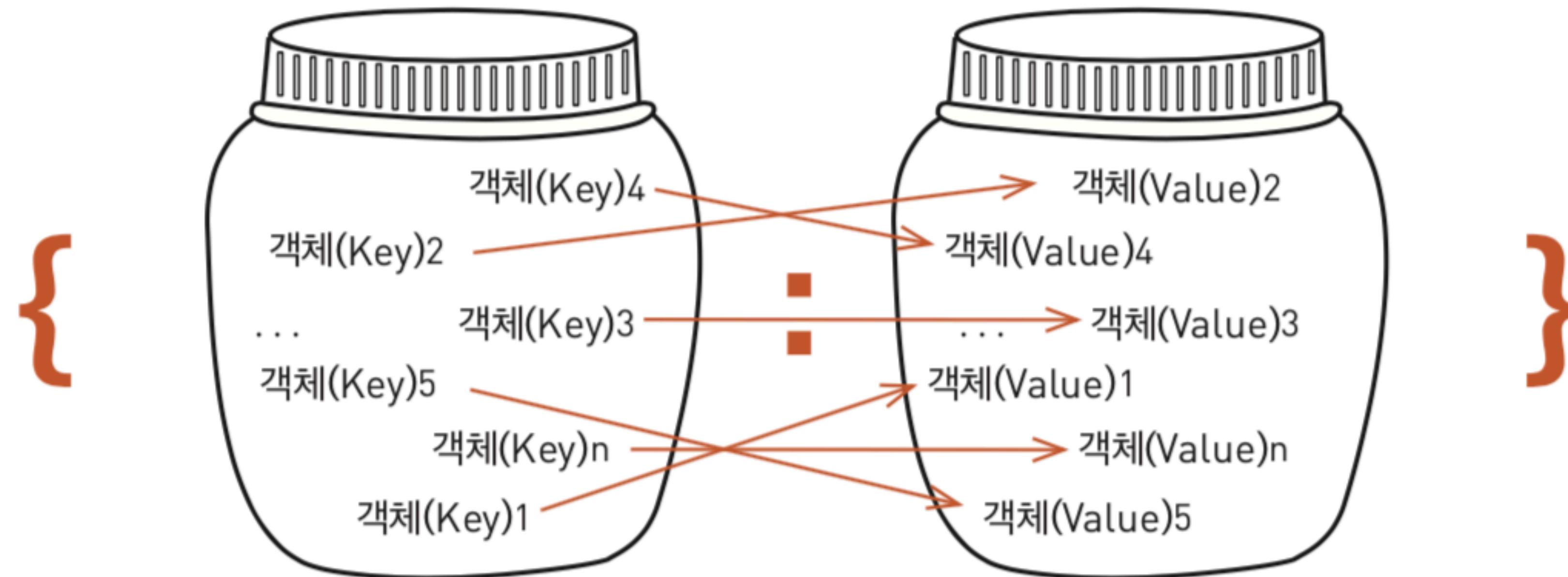
- 키와 매핑값이 1대1로 맵핑
- 예) '이름' = '라이언', '생일' = '1234년 5월 25일'

### ◎ 출력 형식

- 딕셔너리는 항상 중괄호( {}) 형태로 출력
- 세트도 중괄호( {}) 형태로 출력

## ● 리스트와의 차이

- 딕셔너리의 객체는 순서가 없고 키를 통해 맵핑값에 접근할 수 있다
- 인덱스의 개념이 없기 때문에 분할(슬라이싱)하거나 구간 이동을 할 수 없다





# 예시 : 딕셔너리

```
instruments = ['드럼', '기타', '피리']
simple_dict = {1: '엄마', 2: '아빠', 3: '동생'}
complex_dict = {
    '악기': instruments,
    '가족': simple_dict,
    (10, 9): '한글날',
    'RGB': [255, 72, 90],
    'id': (),
    0: 55,
    -307: None,
    'fruits': set()
}
```

```
simple_dict
```

```
{1: '엄마', 2: '아빠', 3: '동생'}
```

```
complex_dict
```

```
{'악기': ['드럼', '기타', '피리'],
 '가족': {1: '엄마', 2: '아빠', 3: '동생'},
 (10, 9): '한글날',
 'RGB': [255, 72, 90],
 'id': (),
 0: 55,
 -307: None,
 'fruits': set()}
```

```
print(len(simple_dict), len(complex_dict))
```

```
3 8
```

```
simple_dict[3] = '나'
```

```
simple_dict[4] = '동생'
```

```
simple_dict
```

```
{1: '엄마', 2: '아빠', 3: '나', 4: '동생'}
```

```
complex_dict['악기'][-1] = '베이스'
```

```
complex_dict
```

```
{'악기': ['드럼', '기타', '베이스'],
 '가족': {1: '엄마', 2: '아빠', 3: '나', 4: '동생'},
 (10, 9): '한글날',
 'RGB': [255, 72, 90],
 'id': (),
 0: 55,
 -307: None,
 'fruits': set()}
```

```
complex_dict[(10, 9)]
```

```
'한글날'
```



# 딕셔너리 생성

## { }

- {} : 빈(empty) 딕셔너리를 반환
- 중괄호({ }) 내에 키:매핑값 쌍을 넣고 쉼표로 구분
  - 예) {x:a}는 단일 객체singleton 딕셔너리를 반환
  - 예) {x:a, y:b, z:c}는 3개의 객체로 구성된 다중 객체 딕셔너리를 반환

## dict() 생성자

- dict() : 빈(empty) 딕셔너리를 반환
- dict(키워드) : 불특정 다수의 키워드 전달인자의 키워드를 키로, 키워드의 값을 매핑값으로 하는 딕셔너리를 반환
  - 키워드는 유효한 파이썬 식별자여야 한다
- dict(순회형) : 불특정 다수의 순회형 전달인자에 기반한 딕셔너리를 반환
  - 순회형 전달인자의 각 객체는 두 개의 객체(첫번째 객체는 키, 두번째 객체는 값)로 구성되어 있다
- dict(매핑형) : 불특정 다수의 매핑형 전달인자에 기반한 딕셔너리를 반환
  - zip() 함수 등을 사용해서 딕셔너리를 만든다

## 기타 방법 : 고급 기능

- 딕셔너리 축약(dictionary comprehension)
  - 예) {x:y for x, y in 순회형}



# 딕셔너리 생성

{ } (중괄호) 키:매핑값 쌍으로 된 각 객체는 쉼표(,)로 구분

```
family = {1: '엄마', 2: '아빠', 3: '동생'}  
print(family)
```

```
{1: '엄마', 2: '아빠', 3: '동생'}
```

```
d1 = {None: None}  
type(d1)  
len(d1)
```

```
d2 = {None: '', '': None}  
type(d2)  
len(d2)  
print(d2[None])  
print(repr(d2[None])) # 대표 형식으로 출력  
print(d2[''])
```

```
mydict = {  
    '악기': ['드럼', '기타', '베이스'],  
    '가족': family,  
    (10, 9): '한글날',  
    'RGB': [255, 72, 90],  
    'id': (),  
    0: 55,  
    -307: None,  
    'fruits': set()  
}  
print(mydict)  
len(family)  
len(mydict)
```



## 키로 맵핑값 추출

### 딕셔너리[키] 키를 통해 맵핑값을 추출

딕셔너리가 참조하는 객체들은 리스트와는 달리 순서가 없기 때문에

- 인덱스 번호로 개별 객체를 추출할 수도 없고,
- 분할 연산자를 사용해서 일부 또는 전체를 가져올 수도 없다

```
mydict = {
    '악기': ['드럼', '기타', '베이스'],
    '가족': {1: '엄마', 2: '아빠', 3: '동생'},
    (10, 9): '한글날',
    'RGB': [255, 72, 90],
    'id': (),
    0: 55,
    -307: None,
    'fruits': set()
}
mydict['악기']
mydict[(10, 9)]
```

```
d8 = {None: 'Null value', True: '참', False: 0}
d8[None] # None이 키
d8[False] # False가 키
```



## 키에 맵핑값 할당

### 딕셔너리[키] = 맵핑값

- 기존의 맵핑값을 새로운 맵핑값으로 갱신할 수도 있고,
- 새로운 객체 쌍을 딕셔너리에 추가할 수도 있다

```
family = {1: '엄마', 2: '아빠', 3: '동생'}
mydict = {
    '악기': ['드럼', '기타', '베이스'],
    '가족': family,
    (10, 9): '한글날',
    'RGB': [255, 72, 90],
    'id': (),
    0: 55,
    -307: None,
    'fruits': set()
}
mydict
```

```
{'악기': ['드럼', '기타', '베이스'],
 '가족': {1: '엄마', 2: '아빠', 3: '동생'},
 (10, 9): '한글날',
 'RGB': [255, 72, 90],
 'id': (),
 0: 55,
 -307: None,
 'fruits': set()}
```



키에 매핑값 할당

딕셔너리[키] = 매핑값

## 키의 매핑값 갱신

- 키가 이미 존재하면, 기존의 매핑값을 새로운 매핑값으로 대체
- 이는 리스트에서 인덱스를 사용해 인덱스 위치에 있는 객체를 새로운 객체로 교체하는 것과 비슷

## 객체 추가

- 키가 존재하지 않으면, 새로운 객체를 키와 매핑값 쌍으로 딕셔너리에 추가
- 이는 리스트에서 append() 메소드로 객체를 추가하는 것과 비슷

```
family = {1: '엄마', 2: '아빠', 3: '동생'}
```

```
# family 키 3의 매핑값을 '동생'에서 '나'로 갱신(교체)
family[3] = '나'
print(family)
```

```
# 키가 4이고 매핑값이 '동생'인 객체를 추가
family[4] = '동생'
print(family)
```



## 매핑값이 복합자료형일 때 딕셔너리 **추가** 및 **갱신**

**추가** : 매핑값의 자료형이 객체를 추가하는 방식을 그대로 사용

**갱신** : 매핑값의 자료형이 객체를 갱신하는 방식을 그대로 사용

## 매핑값이 **리스트**

먼저 **키**로 해당 **매핑값**을 부른 후, 리스트에 객체를 추가하거나 갱신

```
mydict
```

```
{'악기': ['드럼', '기타', '베이스'],
 '가족': {1: '엄마', 2: '아빠', 3: '나', 4: '동생'},
 (10, 9): '한글날',
 'RGB': [255, 72, 90],
 'id': (),
 0: 55,
 -307: None,
 'fruits': set()}
```

```
mydict['악기'].append('키보드')
print(mydict)
```

```
mydict['악기'][2] = '베이스 기타'
print(mydict)
```



## 매핑값이 복합자료형일 때 딕셔너리 **추가** 및 **갱신**

**추가** : 매핑값의 자료형이 객체를 추가하는 방식을 그대로 사용

**갱신** : 매핑값의 자료형이 객체를 갱신하는 방식을 그대로 사용

### 매핑값이 딕셔너리

먼저 **키**로 해당 **매핑값**을 부른 후, 딕셔너리에 객체를 추가하거나 갱신

**dict[키][새로운 키] = <새로운 키의 매핑값>**

mydict

```
{'악기': ['드럼', '기타', '베이스 기타', '키보드'],
 '가족': {1: '엄마', 2: '아빠', 3: '나', 4: '동생'},
 (10, 9): '한글날',
 'RGB': [255, 72, 90],
 'id': (),
 0: 55,
 -307: None,
 'fruits': set()}
```

```
mydict['가족'][5] = '퍼피'
print(mydict)
```

```
mydict['가족'][2] = '아버지'
print(mydict)
```



## 매핑값이 복합자료형일 때 딕셔너리 추가 및 갱신

**추가** : 매핑값의 자료형이 객체를 추가하는 방식을 그대로 사용

**갱신** : 매핑값의 자료형이 객체를 갱신하는 방식을 그대로 사용

### 매핑값이 세트

먼저 키로 해당 **매핑값**을 부른 후, **add()** 메소드로 추가

mydict

```
{'악기': ['드럼', '기타', '베이스 기타', '키보드'],
 '가족': {1: '엄마', 2: '아버지', 3: '나', 4: '동생', 5: '퍼피'},
 (10, 9): '한글날',
 'RGB': [255, 72, 90],
 'id': (),
 0: 55,
 -307: None,
 'fruits': set()}
```

```
mydict['fruits'].add('블루베리')
print(mydict)
```



# 딕셔너리 관련 연산자

## ● 삭제 연산자

### ● **del**

- 객체에 대한 참조를 삭제하는 연산자(i.e., 식별자를 삭제)

## ● 멤버십 연산자

### ● **in / not in**





# 딕셔너리 객체 삭제 연산자

## ● del 객체

- 리스트에서 사용한 삭제 연산자 **del**과 같은 원리로 사용
- 단지 차이점은
  - 리스트처럼 인덱스 번호나 분할 연산자를 사용하지 않고,
  - 키를 통해 대상 객체를 삭제한다

mydict

```
{'악기': ['드럼', '기타', '베이스 기타', '키보드'],
'가족': {1: '엄마', 2: '아버지', 3: '나', 4: '동생', 5: '퍼피'},
(10, 9): '한글날',
'RGB': [255, 72, 90],
'id': (),
0: 55,
-307: None,
'fruits': {'블루베리'}}
```

```
del mydict['id']
print(mydict)
```

```
del mydict['악기'][-1]
print(mydict)
```

```
del mydict['가족'][5]
print(mydict)
```

```
mydict['fruits'].remove('블루베리')
print(mydict)
```



멤버십 연산자 **in**과 **not in**은 딕셔너리에서도 같은 원리로 동작

mydict

```
{'악기': ['드럼', '기타', '베이스 기타'],
 '가족': {1: '엄마', 2: '아버지', 3: '나', 4: '동생'},
 (10, 9): '한글날',
 'RGB': [255, 72, 90],
 0: 55,
 -307: None,
 'fruits': set()}
```

0 **in** mydict

'한글날' **in** mydict

'한글날' **in** mydict.values()

'가족' **not in** mydict

'드럼' **in** mydict

'드럼' **in** mydict.values()

'드럼' **in** mydict['악기']



## ● 딕셔너리를 생성하고 수정하기

- 아래 키-매핑값(key-value) 쌍으로 되어있는 딕셔너리를 생성한 후 출력

- 매핑값(value)은 반드시 튜플로 되어 있어야 한다

Key	Value
'even'	(2, 4, 6, 8, 10)
'odd'	(1, 3, 5, 7, 9)
'prime'	(2, 3, 5, 7)

- 아래 키-매핑값(key-value) 쌍을 방금 만든 항목으로 추가한 후 이 딕셔너리를 다시 출력

- 키 : 'all'

- 매핑값 : (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

- 키 값이 'odd'인 항목을 삭제한 후 이 딕셔너리를 다시 출력

- 실행 결과 예시

```
{'even': (2, 4, 6, 8, 10), 'odd': (1, 3, 5, 7, 9), 'prime': (2, 3, 5, 7)}  
{'even': (2, 4, 6, 8, 10), 'odd': (1, 3, 5, 7, 9), 'prime': (2, 3, 5, 7), 'all': (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)}  
{'even': (2, 4, 6, 8, 10), 'prime': (2, 3, 5, 7), 'all': (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)}
```



# 딕셔너리 메소드

문법	설명
<code>d.copy()</code>	▶ 딕셔너리 <code>d</code> 의 얇은 복사본(shallow copy)을 반환
<code>dict.fromkeys(i[, v])</code>	▶ 순회형 <code>i</code> 의 모든 객체가 키가 되고, <code>v</code> 가 주어지지 않으면 매핑값이 <code>None</code> , <code>v</code> 가 주어지면 매핑값이 <code>v</code> 인 새로운 딕셔너리를 반환하는 클래스 메소드
<code>d.setdefault(k[, v])</code>	<ul style="list-style-type: none"><li>▶ 키 <code>k</code> 가 딕셔너리 <code>d</code> 에 없으면, 키가 <code>k</code> 이고, <code>v</code> 가 주어지지 않으면 매핑값이 <code>None</code>, <code>v</code> 가 주어지면 매핑값이 <code>v</code> 인 객체를 딕셔너리 <code>d</code> 에 <u>추가</u>하고 해당 매핑값 <code>v</code> 를 반환</li><li>▶ 키 <code>k</code> 가 딕셔너리 <code>d</code> 에 이미 있으면, 해당 매핑값을 반환</li></ul>
<code>d.update(x)</code>	<ul style="list-style-type: none"><li>▶ <code>x</code> 의 모든 <b>키:매핑값</b> 쌍 중에 딕셔너리 <code>d</code> 에 없는 <b>키:매핑값</b> 쌍을 딕셔너리 <code>d</code> 에 <u>추가</u>하고, 딕셔너리 <code>d</code> 의 키 중 <code>x</code> 와 같은 키를 가진 객체가 있으면 <code>x</code> 의 키에 담긴 매핑값으로 <u>대체</u></li><li>▶ <code>x</code> 는 딕셔너리, 키워드 전달인자 또는 (<b>키</b>, <b>매핑값</b>) 쌍의 순회형일 수도 있다</li></ul>
<code>d.popitem()</code>	<ul style="list-style-type: none"><li>▶ 딕셔너리 <code>d</code> 에서 마지막 객체를 (<b>키</b>, <b>매핑값</b>) 튜플형으로 반환하고 해당 객체를 딕셔너리 <code>d</code> 에서 삭제</li><li>▶ 딕셔너리 <code>d</code> 가 비어 있으면 <b>KeyError</b> 가 발생</li></ul>
<code>d.pop(k[, v])</code>	<ul style="list-style-type: none"><li>▶ 키 <code>k</code> 가 딕셔너리 <code>d</code> 에 있으면, 그 매핑값을 반환하고 해당 객체를 딕셔너리 <code>d</code> 에서 삭제</li><li>▶ 키 <code>k</code> 가 딕셔너리 <code>d</code> 에 없고, <code>v</code> 가 주어지지 않으면 <b>KeyError</b> 가 발생하고, <code>v</code> 가 주어지면 <code>v</code> 를 반환</li></ul>



# 딕셔너리 메소드

문법	설명
<code>d.clear()</code>	▶ 딕셔너리 <code>d</code> 의 모든 객체를 삭제
<code>d.keys()</code>	▶ 딕셔너리 <code>d</code> 의 모든 키를 담은 형식(읽기전용 순회형) <code>dict_keys</code> 를 반환
<code>d.values()</code>	▶ 딕셔너리 <code>d</code> 의 모든 매핑값을 담은 형식(읽기전용 순회형) <code>dict_values</code> 를 반환
<code>d.items()</code>	▶ 딕셔너리 <code>d</code> 의 모든 (키, 매핑값) 쌍을 담은 형식(읽기전용 순회형) <code>dict_items</code> 를 반환
<code>d.get(<i>k</i>[, <i>v</i>])</code>	<ul style="list-style-type: none"><li>▶ 키 <code>k</code> 가 딕셔너리 <code>d</code> 에 있으면, 해당 매핑값을 반환</li><li>▶ 키 <code>k</code> 가 딕셔너리 <code>d</code> 에 없고, <code>v</code> 가 주어지지 않으면 <code>None</code>, <code>v</code> 가 주어지면 <code>v</code> 를 반환</li></ul>



구체적으로

copy(), fromkeys(), setdefault(), update(), popitem(), pop(), clear(), get()  
메소드

책을 참고해서 반드시 따라해보세요.

시험 범위에 포함됩니다.



# 딕셔너리 질의 메소드

**d.keys()**

딕셔너리 *d* 의 모든 키를 담은 형식(읽기전용 순회형)  
*dict\_keys*를 반환

**d.values()**

딕셔너리 *d* 의 모든 매핑값을 담은 형식(읽기전용 순회형)  
*dict\_values*를 반환

**d.items()**

딕셔너리 *d* 의 모든 (**키**, **매핑값**) 쌍을 담은 형식(읽기전용  
순회형) *dict\_items*를 반환

```
d = {  
    '악기': ['드럼', '기타', '베이스', '키보드', '보컬'],  
    '가족': {1: '엄마', 2: '아빠', 3: '나', 4: '동생'},  
    (10, 9): '한글날',  
    'RGB': [255, 72, 90],  
    0: 55,  
    -307: None,  
    'fruits': {'블루베리'},  
    'X': '강아지',  
    1: 55,  
    2: 55  
}
```

**d.keys()**

**d.values()**

**d.items()**

`type(d.keys())`

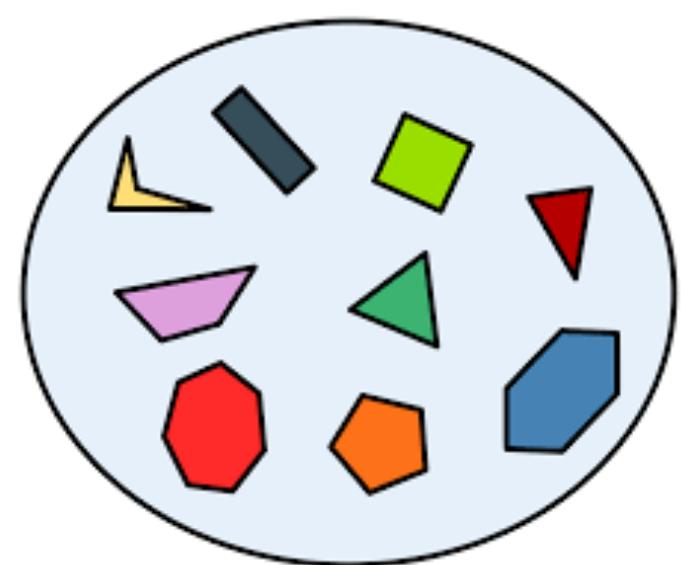
`tuple(d.keys())`

`list(d.items())`

# 집합형

---

Set Types





## ● 집합형(set types)이란?

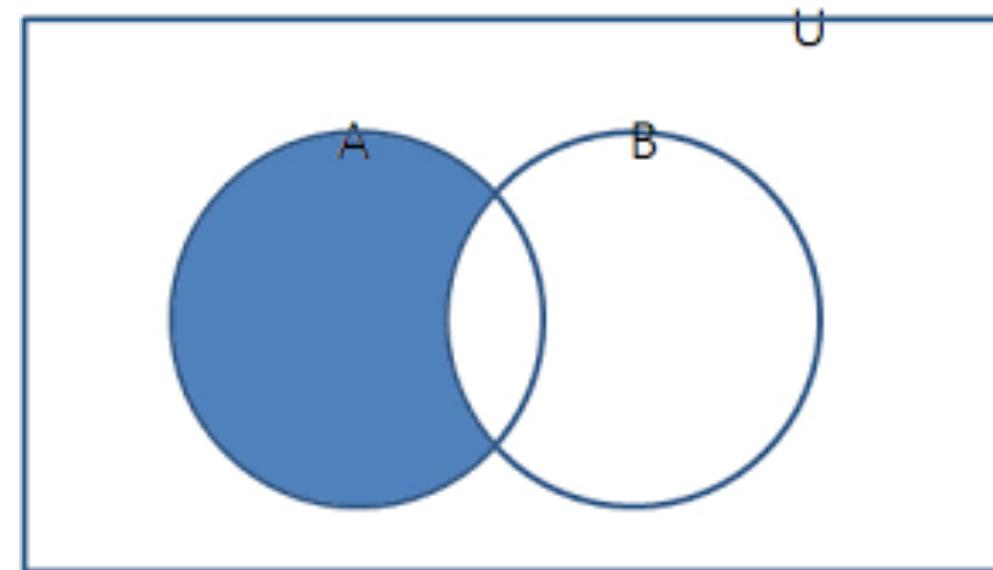
- 순서없이(unordered) 0개 이상의 객체를 참조하는 자료형
  - ✿ 순회할 경우 집합형은 임의의 순서대로 객체를 꺼내온다
  - ✿ 중복이 허용되지 않는다
  - ✿ 집합형에는 해시가능한 객체만 담을 수 있다
    - ◎ 해시가능한 객체(hashable object) : `__hash__()` 메소드를 가지고 있으며, 이 메소드는 객체가 생성되고 폐기될 때까지 항상 동일한 반환 값을 돌려주는 객체를 지칭
    - ✿ 해시가능한 자료형 : 그 값이 불변성인 모든 내장 불변자료형(buit-in immutable data types)
      - ◎ `int`, `float`, `str`, `tuple`, `frozenset` 등
    - ✿ 집합형에 담을 수 없는 해시가능하지 않는 자료형
      - ◎ `dict`, `list`, `set` 등
  - 순회형(iterable)

## ● 집합형 종류

- 세트(set) : 가변자료형(mutable)
- 프로즌 세트(frozenset, 고정집합) : 불변자료형(immutable)

# 세트형

Set Type



## ● 세트(set)란?

### ● 가변자료형(mutable)

- ▣ 담고 있는 객체를 삽입, 변경, 삭제하는 것이 가능

### ● 순서없이(unordered) 0개 이상의 해시가능한 객체를 참조하는 집합형

- ▣ 각 객체는 쉼표( , )로 구분한다

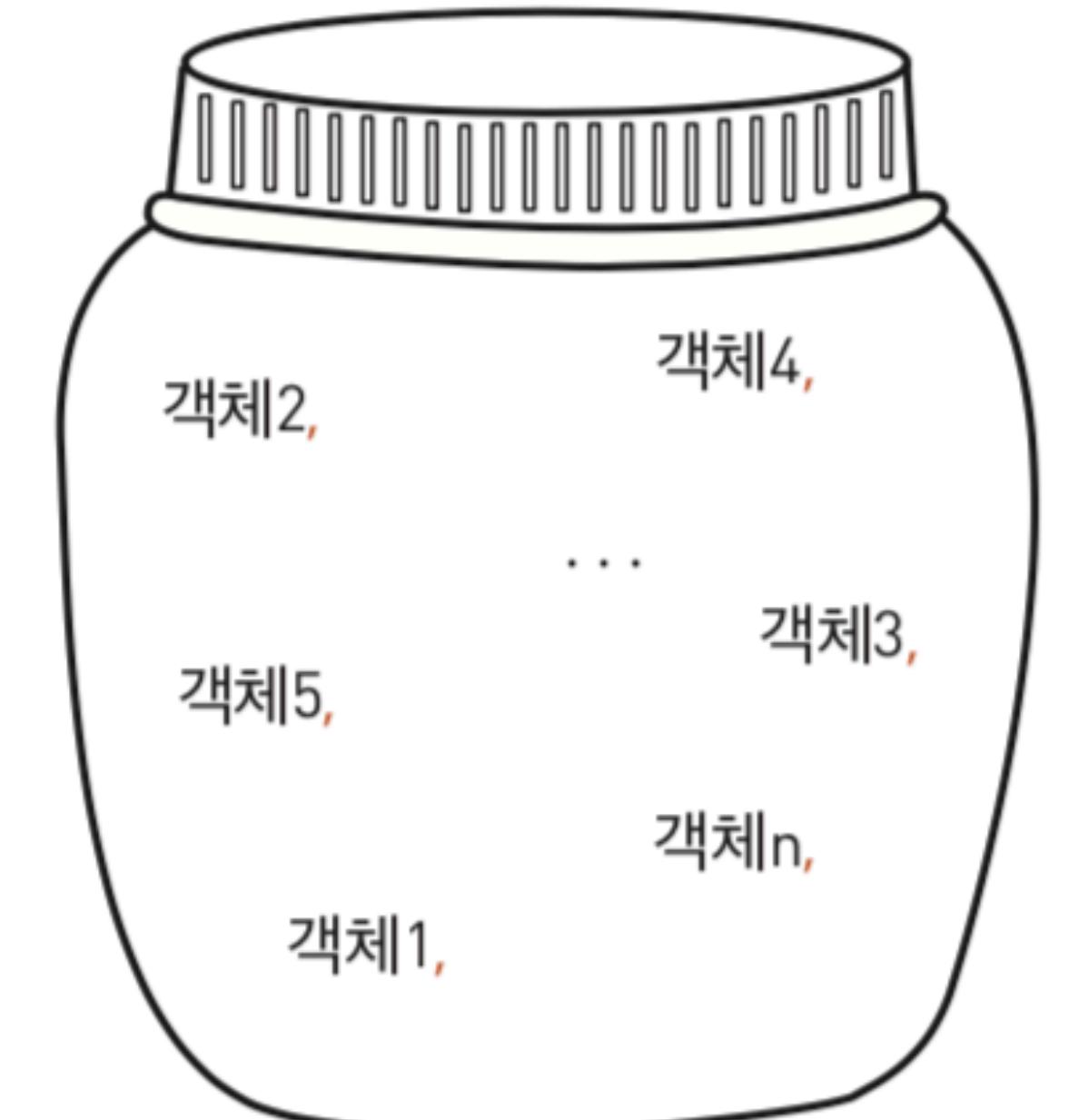
- ▣ 해시가능한 객체(불변자료형)만 담을 수 있다

### ● 출력 형식

- ▣ 세트는 항상 중괄호( { } ) 형태로 출력

### ● 인덱스의 개념이 없기 때문에 분할(슬라이싱)하거나 구간 이동을 할 수 없다

{



}



## { }

- 중괄호({ }) 내에 쉼표로 구분된 객체를 입력

- 예) {x}는 단일 객체(singleton) 세트를 반환

- 예) {x, y, z}는 3개의 객체로 구성된 다중 객체 세트를 반환

## set() 생성자

- set() : 빈(empty) 세트를 반환



- 빈 세트를 빈 중괄호({ })로 생성할 수 없다

- 빈 중괄호({ })는 빈 딕셔너리(dic)를 생성할 때 사용

- 딕셔너리가 세트보다 파이썬에서 더 중요하고 훨씬 자주 사용하는 자료형

- set(순회형) : 순회형(iterator) 객체를 얇은 복사(shallow copy)해서 반환

- 형변환 클래스(str(), int(), float())처럼 다른 자료형의 객체를 세트로 변환할 때 사용하는데 순회형만 세트로 변환이 가능

- 순회형 : 문자열, 리스트, 튜플, 세트, 딕셔너리, 순회자(iterator) 객체 등

## 기타 방법 : 고급 기능

- 세트 축약(set comprehension)

- 예) {x for x in 순회형}



{ } (중괄호) 각 객체는 쉼표(,)로 구분

```
set1 = {'a', 'b', 3}  
print(set1)  
  
set2 = {'a', 'b', 3, 'a', 3}  
print(set2)  
  
set3 = {True, 5, '드럼', ('x', 'y'), -3.14, None}  
print(set3)  
len(set3)
```

```
set4 = {True, 5, '드럼', ['x', 'y'], -3.14, None}  
set5 = {True, 5, {'drum': '드럼'}, -3.14, None}  
set6 = {True, 5, '드럼', ('x', 'y', [1, 2, 3]), -3.14, None}
```



## set() 빈 세트를 만드는 유일한 방법

- 빈 중괄호({})는 빈 딕셔너리를 생성할 때 사용하기 때문에  
빈 세트를 만들려면 set() 생성자를 사용해야만 한다

```
empty_set = set()
```

```
type(empty_set)
```

```
print(empty_set)
```

## set() 생성자(클래스)

## set(순회형)

```
set7 = set('abcde')  
print(set7)
```

```
set8 = set('바나나')  
print(set8)
```

```
set9 = set({'바나나'})  
print(set9)  
len(set9)
```

```
set10 = {'바나나'}  
print(set10)  
len(set10)
```



세트는 리스트나 튜플로 형변환이 가능하지만 딕셔너리로 형변환 할 수는 없다

리스트, 튜플, 딕셔너리는 세트로 형변환이 가능하다(단, 딕셔너리의 경우 키(key)만 세트로 형변환)

```
s1 = {1, 2, 3}
t1 = tuple(s1)
L1 = list(s1)
print(t1)
print(L1)
```

```
s2 = set(t1)
s3 = set(L1)
print(s2)
print(s3)
```

```
d1 = {(10, 9): '한글날', 0: 55, '악기': ['드럼', '기타', '베이스']}
s4 = set(d1)
print(s4)
```

```
t2 = 'a', (1, 2, [3, 4])
s5 = set(t2)
```

```
L2 = [1, 2, 3, {'color': 'green'}]
s6 = set(L2)
```



- 멤버십 연산자

- in / not in

- 세트(set) 연산자

- <, >

- 진부분 집합

- <=, >=

- 부분 집합

- |

- 세트 결합(합집합)

- &

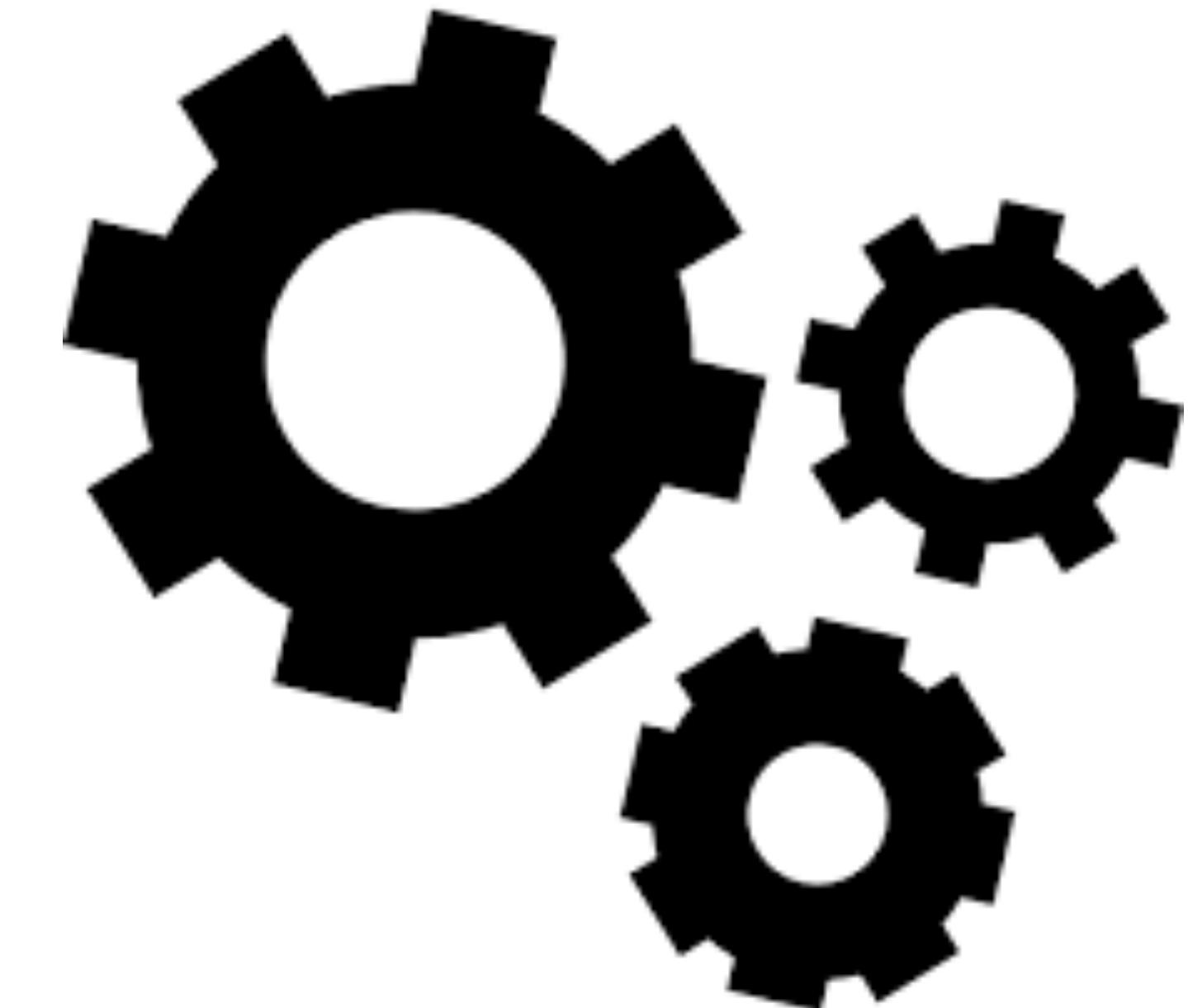
- 세트 교차(교집합)

- 

- 세트 빼기(차집합)

- ^

- 세트 대칭차(대칭 차집합)





구체적으로

**in, not in, <, <=, >, >=**

연산자

책을 참고해서 반드시 따라해보세요.

**시험 범위에 포함됩니다.**



## | 세트 결합(합집합)

```
x = {1, 2, 3, 4, 5}  
y = {3, 4, 5, 6, 7}
```

```
x | y
```

```
x & y
```

## & 세트 교차(교집합)

```
x - y
```

```
y - x
```

## - 세트 빼기(차집합)

```
x ^ y
```

```
y ^ x
```

## ^ 세트 대칭차(대칭 차집합)



# 세트 메소드

문법	설명
<code>s.copy()</code>	▶ 세트 <code>s</code> 의 얕은 복사본(shallow copy)을 반환
<code>s.issubset(t)</code> <code>s &lt;= t</code>	▶ 세트 <code>s</code> 가 세트 <code>t</code> 의 부분집합일 때 참(True)을 반환 ▶ 세트 <code>s</code> 가 세트 <code>t</code> 의 진부분집합(proper subset)인지 확인할 때는 <code>s &lt; t</code> 를 사용
<code>s.issuperset(t)</code> <code>s &gt;= t</code>	▶ 세트 <code>s</code> 가 세트 <code>t</code> 의 포함집합일 때 참(True)을 반환 ▶ 세트 <code>s</code> 가 세트 <code>t</code> 의 진포함집합(proper superset)인지 확인할 때는 <code>s &gt; t</code> 를 사용
<code>s.isdisjoint(t)</code>	▶ 세트 <code>s</code> 와 세트 <code>t</code> 가 서로 같은 객체가 하나도 없을 때 참(True)을 반환
<code>s.union(t)</code> <code>s   t</code>	▶ 세트 <code>s</code> 의 모든 객체와 세트 <code>t</code> 의 객체 중 세트 <code>s</code> 에 없는 객체들을 담은 새로운 세트를 반환(프로즌 세트(frozenset)에도 사용 가능)
<code>s.update(t)</code> <code>s  = t</code>	▶ 세트 <code>t</code> 에 있는 객체 중 세트 <code>s</code> 에 없는 객체들을 세트 <code>s</code> 에 추가
<code>s.intersection(t)</code> <code>s &amp; t</code>	▶ 세트 <code>s</code> 와 세트 <code>t</code> 에 공통으로 존재하는 객체들(교집합)을 담은 새로운 세트를 반환 (프로즌 세트(frozenset)에도 사용 가능)
<code>s.intersection_update(t)</code> <code>s &amp;= t</code>	▶ 세트 <code>s</code> 자신과 세트 <code>t</code> 에 공통으로 존재하는 객체들만 세트 <code>s</code> 에 담는다



# 세트 메소드

문법	설명
<code>s.difference(t)</code> <code>s - t</code>	▶ 세트 <code>s</code> 에는 있지만 세트 <code>t</code> 에 없는 객체들을 담은 새로운 세트를 반환(프로즌 세트( <b>frozenset</b> )에도 사용 가능)
<code>s.difference_update(t)</code> <code>s -= t</code>	▶ 세트 <code>s</code> 의 객체 중 세트 <code>t</code> 에 있는 모든 객체를 삭제
<code>s.symmetric_difference(t)</code> <code>s ^ t</code>	▶ 세트 <code>s</code> 의 객체와 세트 <code>t</code> 의 객체 중 두 세트에 모두 속한 객체를 제외한 나머지를 담은 새로운 세트를 반환(프로즌 세트( <b>frozenset</b> )에도 사용 가능)
<code>s.symmetric_difference_update(t)</code> <code>s ^= t</code>	▶ 세트 <code>s</code> 에 자신과 세트 <code>t</code> 의 객체 중 두 세트에 모두 속한 객체를 제외한 나머지 객체들을 담는다
<code>s.add(x)</code>	▶ 세트 <code>s</code> 에 객체 <code>x</code> 가 없다면 세트 <code>s</code> 에 <code>x</code> 를 추가
<code>s.pop()</code>	▶ 세트 <code>s</code> 에서 임의(random)의 객체를 반환하고 해당 객체를 세트 <code>s</code> 에서 삭제 - 세트 <code>s</code> 가 비어있는 경우 <b>KeyError</b> 가 발생
<code>s.remove(x)</code>	▶ 세트 <code>s</code> 에서 객체 <code>x</code> 를 삭제(☞ <code>set.discard(s)</code> 참조) - 세트 <code>s</code> 안에 객체 <code>x</code> 가 없으면 <b>KeyError</b> 가 발생
<code>s.discard(x)</code>	▶ 세트 <code>s</code> 에서 객체 <code>x</code> 를 삭제(☞ <code>set.remove(s)</code> 참조) - 세트 <code>s</code> 안에 객체 <code>x</code> 가 없어도 오류 메시지를 출력하지 않는다
<code>s.clear()</code>	▶ 세트 <code>s</code> 의 모든 객체를 삭제



# 세트 추가/삭제 메소드

## `s.add(x)`

- 세트 `s`에 객체 `x`가 없다면 세트 `s`에 `x`를 추가

```
x = {1, 2, 3, 4, 5}
```

```
x.add(5)  
print(x)
```

## `s.pop()`

- 세트 `s`에서 임의(random)의 객체를 반환하고 해당 객체를 세트 `s`에서 삭제
- 세트 `s`가 비어있는 경우 `KeyError` 가 발생

```
x.add('a')  
print(x)
```

## `s.remove(x)`

- 세트 `s`에서 객체 `x`를 삭제
- 세트 `s` 안에 객체 `x`가 없으면 `KeyError` 가 발생

```
x.pop()  
print(x)
```

## `s.discard(x)`

- 세트 `s`에서 객체 `x`를 삭제
- 세트 `s` 안에 객체 `x`가 없어도 오류 메시지를 출력하지 않는다

```
x.remove(4)  
print(x)
```

## `s.clear()` 세트 `s`의 모든 객체를 삭제

```
x.discard('a')  
print(x)
```

```
x.discard('a')
```

```
x.remove('a')
```

```
x.clear()  
print(x)
```



## ● 세트 연산자 또는 메소드 사용하기

- 다음은 영화 리뷰 사이트 중 하나인 IMDb(<https://www.imdb.com>)에 올라온 영화 리뷰 중 일부
  - there is almost as much music as a musical but it doesn't feel much like a musical
  - what wonderful music they used in this film
  - the original music was also quite good!the original music was also quite good!
- 위 3개 리뷰에서 공통적으로 사용한 단어를 출력



● 각 리뷰 문자열을 분할하여 단어들의 리스트를 생성

● 리스트를 세트로 변환

● 세트들의 교집합을 구하여 모든 리뷰에서 사용한 단어들의 집합을 계산

## ● 실행 결과 예시

```
{'music'}
```