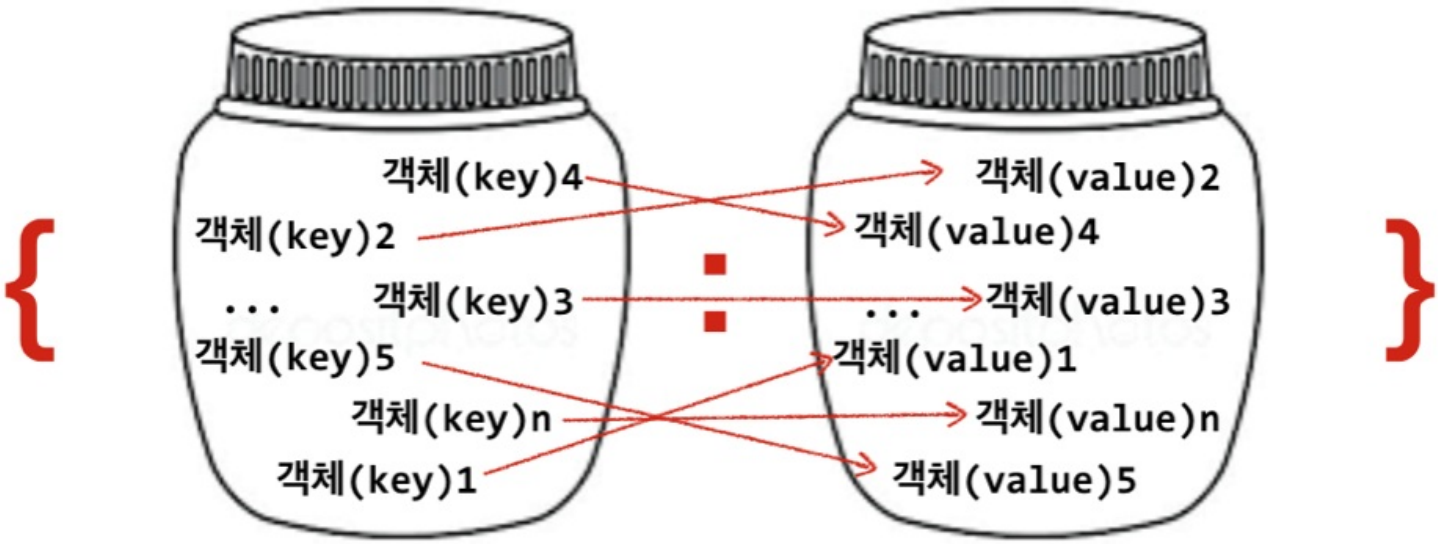
매핑 자료형

- **순서 없이**(unordered) 0개 이상의 키-매핑값(key-value) 쌍으로 된 객체를 참조하는 자료형을 말한다.
- **키**(key)
  - 해시가능한 객체만 키(key)로 사용할 수 있다.
- 해시가능한(hashable) 객체란?
  - 그 값이 불변성(immutable)인 자료형으로 **int**, **float**, **str**, **tuple** 등이 있다.
- 해시가능하지 않는 자료형은 딕셔너리의 키로 사용 할 수 없는데 **list**, **dict**, **set**이 여기에 속한다.
- **매핑값**(value)
  - 매핑값으로는 모든 종류의 자료형을 사용할 수 있다.

딕셔너리(dict)

- **가변자료형**(mutable)이다.
  - 생성한 후 내용의 변경이 가능하다. 즉, 담고 있는 객체를 삭제, 변경, 삽입하는 것이 가능하다.
- **순서 없이**(unordered) 0개 이상의 키-매핑값(key-value) 쌍으로 된 객체를 참조하는 **매핑형**이다.
- 각 객체 쌍은 쉼표(,)로 구분한다.
- 객체 하나가 **키:매핑값** 형태로 되어 있으며, 키와 매핑값도 객체다.
- **키**(key) : 해시가능한 객체(불변자료형)만 가능하다.
- **매핑값**(value) : 어떠한 자료형의 객체도 가능하다
- 출력 형식
  - 딕셔너리는 항상 중괄호(**{ }**) 형태로 출력한다.



딕셔너리와 리스트의 차이

리스트와 달리 **딕셔너리**가 담고 있는 객체는 순서가 없다.

리스트는 같은 값을 가진 객체를 포함할 수 있지만,

딕셔너리는 키의 중복을 허락하지 않기 때문에 같은 키를 가진 객체는 포함할 수 없다.

단, 매핑값은 중복이 가능하다. 즉, 키만 다르면 매핑값은 같아도 상관없다.

리스트와 달리 **딕셔너리**는 인덱스 개념이 없기 때문에,

분할 연산자를 사용해 객체 전체 또는 일부를 추출할 수 없다.

리스트는 인덱스 번호로 객체를 추출하지만,

딕셔너리는 키로 객체를 추출한다.

딕셔너리 생성

딕셔너리를 만드는 방법으로는 다음 세 가지가 있다.

- **{ }**(중괄호)
- **dict()** 생성자(클래스)
- 딕셔너리 축약(dictionary comprehension)

{ }(중괄호)

- 키:매핑값 쌍으로 된 각 객체는 쉼표(,)로 구분한다.

In [ ]:

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

In [ ]:

```
print(family)
```

모든 복합자료형은 어떠한 자료형도 담을 수 있다.

- 따라서 딕셔너리는 매핑값으로 딕셔너리 자신을 포함해 어떠한 자료형도 담을 수 있다.
- 단, 키는 해시가능한(hashable) 불변자료형(immutable)만 가능하다.

In [ ]:

```
mydict = {
    '악기': ['드럼', '기타', '베이스'],  # 매핑값이 리스트다.
    '가족': family,                    # 매핑값이 딕셔너리다.
    (10, 9): '한글날',                 # 키가 튜플(불변자료형)이다.
    'RGB': [255, 72, 90],
    'id': (),                          # 매핑값이 빈 튜플이다.
    0: 55,
    -307: None,                        # 매핑값 뿐만 아니라 키에도 None이 올 수 있다
    'fruits': set()                    # 매핑값이 세트다.
}
print(mydict)
```

dict() 생성자(클래스)

In [ ]:

```
d1 = dict()
print(d1)
```

In [ ]:

```
d2 = dict(drum = '드럼', guitar = '기타')
print(d2)
```

딕셔너리 축약(dictionary comprehension)

In [ ]:

```
numbers = [1, 2, 3, 4, 5]
d3 = {x: x**2 for x in numbers}
print(d3)
```

딕셔너리 인덱스

키를 통해 매핑값 추출하기

딕셔너리가 참조하는 객체들은 리스트와는 달리 순서가 없기 때문에

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

다음 형식으로 키를 통해 매핑값을 추출할 수 있다.

    딕셔너리[키]

In [ ]:

```
d8 = {None: 'Null value', True: '참', False: 0}
```

In [ ]:

```
# --- {None: 'Null value', True: '참', False: 0}
# None이 키다.
d8[None]
```

In [ ]:

```
# --- {None: 'Null value', True: '참', False: 0}
# False가 키다.
d8[False]
```

그러면 지금부터는 앞에서 사용한 딕셔너리로 개별 객체를 추출하는 방법을 알아보자.

In [ ]:

```
# --- {'악기': ['드럼', '기타', '베이스'], ...}
mydict['악기']
```

In [ ]:

```
# --- {(10, 9): '한글날', ...}
mydict[(10, 9)]
```

키에 매핑값을 할당하기

이번에는 키에 매핑값을 할당해보자. 다음과 같은 형식으로 코드를 작성한다.

    딕셔너리[키] = 매핑값

이 형식으로

- 새로운 객체 쌍을 딕셔너리에 추가할 수도 있고,
- 기존의 매핑값을 새로운 매핑값으로 갱신할 수도 있다.

키의 매핑값 갱신하기

키가 이미 존재하는 경우에는

- 기존의 매핑값을 새로운 매핑값으로 대체하게 된다.

이는 마치 리스트에서 인덱스를 사용해 인덱스 위치에 있는 객체를 새로운 객체로 **교체**하는 것과 비슷하다.

그럼 ***family***의 키 3의 매핑값을 '동생'에서 '나'로 바꿔보자.

In [ ]:

```
# --- {1: '엄마', 2: '아빠', 3: '동생'}
family[3] = '나'
print(family)
```

객체 추가하기

만약 키가 존재하지 않으면,

- 새로운 객체를 키와 매핑값 쌍으로 딕셔너리에 추가한다.

이는 리스트에서 **append()** 메소드로 객체를 **추가**하는 것과 비슷하다.

그럼 ***family***에 없는 키를 추가해보자.

In [ ]:

```
# --- {1: '엄마', 2: '아빠', 3: '나'}
# 키가 4이고 매핑값이 '동생'인 객체를 추가한다.
family[4] = '동생'
print(family)
```

만약 매핑값의 객체가 복합자료형이면?

매핑값이 복합자료형이면, 주의할 점은

- **매핑값의 자료형이 객체를 추가하는 방법**을 사용해야 한다는 것이다.

마찬가지로 매핑값을 수정하는 방법도,

- **매핑값의 자료형이 객체를 수정하는 방법**을 사용해야 한다.

매핑값이 리스트인 경우

매핑값이 리스트면 여기에 객체를 추가할 때

- **append()** 등 리스트 자료형이 객체를 추가하는 메소드를 사용해야 한다.

키가 '악기'인 객체의 매핑값(리스트)으로 '보컬'을 추가해보자.

In [ ]:

```
# --- {'악기': ['드럼', '기타', '베이스'], ...}
# '키보드'를 '악기'에 추가한다.
mydict['악기'].append('키보드')
print(mydict)
```

이번에는 키가 '악기'인 매핑값 리스트의 값을 수정해보자.

In [ ]:

```
# --- {'악기': ['드럼', '기타', '베이스', '키보드'], ...}
# '악기'의 세 번째 객체를 '베이스 기타'로 교체한다.
mydict['악기'][2] = '베이스 기타'
print(mydict)
```

매핑값이 딕셔너리인 경우

매핑값이 딕셔너리면 앞서 설명했듯이

- 키가 없을 때 객체를 추가하거나
- 키가 있을 때 매핑값을 갱신하는 방식인
  dict[키] = 매핑값 형식을 사용해야 한다.

먼저 키로 해당 매핑값을 부른 후, 딕셔너리에 객체를 추가하거나 매핑값을 갱신해야 한다.

따라서 다음과 같은 형식이 될 것이다.

`dict[키][새로운-키] = 새로운-키의-매핑값`

키가 '가족'이고 매핑값이 딕셔너리인 객체에 키가 5고 매핑값이 '퍼피'인 객체를 추가해보자

In [ ]:

```
# --- {'가족': {1: '엄마', 2: '아빠', 3: '나', 4: '동생'}, ...}
# '가족'에 키가 5이고 매핑값이 '퍼피'인 객체를 추가한다.
mydict['가족'][5] = '퍼피'
print(mydict)
```

이번에는 딕셔너리 매핑값을 수정해보자.

In [ ]:

```
# --- '가족': {1: '엄마', 2: '아빠', 3: '나', 4: '동생', 5: '퍼피'}, ...}
# '가족' 중 키가 2인 객체의 매핑값을 '아버지'로 대체한다.
mydict['가족'][2] = '아버지'
print(mydict)
```

In [ ]:

```
print(mydict)
```

딕셔너리 관련 연산자

딕셔너리와 관련한 대표적인 연산자로는 다음과 같은 것이 있다.

삭제 연산자 : **del**
멤버십 연산자 : **in**/**not in**

삭제 연산자

리스트에서 사용한 삭제 연산자 **del**은 딕셔너리에도 같은 원리로 사용할 수 있다.

단지 차이점은

- 리스트처럼 인덱스 번호나 분할 연산자를 사용하지 않고,
- 키를 통해 대상 객체를 삭제한다는 점이다.

해당 키를 가진 객체를 가지고 있지 않다면 KeyError를 발생시킨다

In [ ]:

```
# --- {'id': (), ...}
# 키가 'id'인 객체를 삭제한다.
del mydict['id']
print(mydict)
```

In [ ]:

```
# --- {'id': (), ...}
# 키가 'id'인 객체를 삭제한다.
del mydict['id']
print(mydict)
```

멤버십 연산자

멤버십 연산자 **in**과 **not in**은 딕셔너리에서도 같은 원리로 동작한다.

즉, 딕셔너리 안에 객체가 존재하는지 확인할 때 사용한다.

In [ ]:

```
# --- 키가 0인 객체가 딕셔너리에 존재하는지 확인한다.
# {0: 55, ...}
0 in mydict
```

In [ ]:

```
# --- '한글날'이 딕셔너리에 존재하는지 확인한다.
# {(10, 9): '한글날', ...}
'한글날' in mydict
```

'한글날'은 존재하지 않는다고 한다.

딕셔너리에서 **in**과 **not in**은 기본적으로 키(key)가 존재하는지를 확인하기 때문이다.

그럼 키 대신 매핑값이 존재하는지 확인하려면 어떻게 해야 할까?

딕셔너리 메소드 중 하나인 **values()**를 사용하면 된다.

In [ ]:

```
# --- {(10, 9): '한글날', ...}
# '한글날'이 딕셔너리의 매핑값에 존재하는지 확인한다.
'한글날' in mydict.values()
```

In [ ]:

```
# --- '가족': {1: '엄마', 2: '아버지', 3: '나', 4: '동생'}, ...}
# '가족'이 딕셔너리에 존재하지 않는지 확인한다.
'가족' not in mydict
```

딕셔너리 관련 메소드

키 생성 메소드 : **fromkeys()**
추가 메소드 : **setdefault()**, **update()**
삭제 메소드 : **popitem()**, **pop()**, **clear()**
질의 메소드 : **keys()**, **values()**, **items()**, **get()**

키 생성 메소드

dict.fromkeys(*s[,v]*)

- 시퀀스형 *s* 의 모든 객체가 키가 된다.
  - *v* 가 주어지지 않으면 매핑값이 None
  - *v* 가 주어지면 모든 매핑값은 *v* 가 된다.

In [ ]:

```
keys = ['a', 'b', 'c', 'd']
val = True

d1 = dict.fromkeys(keys)
d2 = dict.fromkeys(keys, val)

print(d1)
print(d2)
```

추가 메소드

d.setdefault(*k[,v]*)

- 키 *k* 가 딕셔너리 *d* 에 없는 경우
  - 키가 *k* 인 객체를 삽입한다.
  - *v* 가 주어지면 매핑값으로 사용, 주어지지 않으면 None을 매핑 값으로 사용
- 키 *k* 가 딕셔너리 *d* 에 있는 경우
  - 딕셔너리를 업데이트 하지 않고 해당 매핑 값을 반환한다.

In [ ]:

```
d = {'a': True, 'b':False}

print(d.setdefault('a', False))
print(d)

print(d.setdefault('c', False))
print(d)
```

추가 메소드

d.update(*x*)

- x의 모든 키:매핑값 쌍 중에서
  - 딕셔너리 *d* 에 없는 키:매핑값 쌍을 딕셔너리 *d* 에 추가
  - 딕셔너리 *d* 에 있는 키는 새로운 매핑값으로 갱신

In [ ]:

```
d = {'a': True, 'b':False}
x = {'b':True, 'c':False}
d.update(x)

print(d)
```

삭제 메소드

d.popitem( )

- 딕셔너리 *d* 에서 임의의 객체를 (키, 매핑값) 튜플형으로 반환
- 해당 객체를 d에서 삭제

In [ ]:

```
d = {'a': True, 'b':False, 'c':False, 'd':True}
print(d.popitem())
print(d)
```

삭제 메소드

d.pop(*k[,v]*)

- 키 *k* 가 딕셔너리 *d* 에 있는 경우
  - 그 매핑값을 반환하고 해당 객체를 딕셔너리 *d* 에서 삭제
- 키 *k* 가 딕셔너리 *d* 에 없는 경우
  - *v* 가 주어지면 _v_를 반환
  - *v* 가 주어지지 않으면 KeyError 발생

In [ ]:

```
d = {'a': True, 'b':False, 'c':False, 'd':True}

print(d.pop('a'))
print(d)

print(d.pop('a', 'key does not exist'))
print(d.pop('a'))
```

삭제 메소드

d.clear( )

- 딕셔너리 *d* 의 모든 객체를 삭제

In [ ]:

```
d = {'a': True, 'b':False, 'c':False, 'd':True}
print(d)
d.clear()
print(d)
```

질의 메소드

- *d*.**keys()**
  - 딕셔너리 *d*의 모든 키를 담은 형식(읽기 전용 리스트) **dict_keys**를 반환한다.
- *d*.**values()**
  - 딕셔너리 *d*의 모든 매핑값을 담은 형식(읽기 전용 리스트) **dict_values**를 반환한다.
- *d*.**items()**
  - 딕셔너리 *d*의 모든 (키, 매핑값) 쌍을 담은 형식(읽기 전용 리스트) **dict_items**를 반 환한다.

In [ ]:

```
d = {'a': True, 'b':False, 'c':False, 'd':True}

print(d.keys())
print(d.values())
print(d.items())
```

keys(), **values()**, **items()**는 **읽기 전용 리스트**를 반환하기 때문에,

반환한 값을 받아서 직접 사용할 수는 없다.

In [ ]:

```
# 자료형이 리스트가 아니다.
type(d.keys())
```

In [ ]:

```
print(d.keys())
```

따라서 직접 사용하려면...

읽기 전용 리스트를 일반 리스트나 튜플로 **형변환**해야 한다.

In [ ]:

```
# 딕셔너리의 모든 키를 튜플로 형변환한다.
tuple(d.keys())
```

In [ ]:

```
# 딕셔너리의 모든 키:매핑값 쌍을
# (키, 매핑값)의 튜플을 객체로 담고 있는 리스트로 형변환한다.
list(d.items())
```

keys(), values(), items() 이 세 메소드는 왜 처음부터 바로 처리가 가능한 리스트 등의 자료형이 아닌 '읽기 전용' 리스트를 반환할 것일까?

컴퓨터의 자원을 더 효율적으로 사용하기 위해서다.

- 새로운 객체의 생성을 하지 않고, 딕셔너리의 키-매핑값들을 직접 참조하여 사용

질의 메소드

d.**get**(***k***[, ***v***])

- 키 *k* 가 딕셔너리 *d* 에 있는 경우, 해당 매핑값을 반환
- 키 *k* 가 딕셔너리 *d* 에 없는 경우
  - *v* 가 주어지지 않으면 None을 반환
  - *v* 가 주어지면 *v* 를 반환

In [ ]:

```
d = {'a': True, 'b':False, 'c':False, 'd':True}

print(d.get('a'))
print(d.get('e'))
print(d.get('e', 'key does not exist'))
```