

# 단원 05

## 딕셔너리와 집합

인공지능소프트웨어학과

강환수 교수



## Section 1. 키와 값인 쌍의 나열인 딕셔너리



## 딕셔너리의 개념

- 키와 값의 쌍을 항목으로 관리하는 딕셔너리
  - 딕셔너리는 말 그대로 '사전'을 생각하면 쉬움
    - 여기서 낱말을 키(key)라고 생각하고 해설을 값(value)이라고 생각



KOREAN-ENGLISH		gwan-jung-seok	
gwai-li-bi 관리비 n maintenance cost	gwang-hwal-ha-da 광활하다 adj vast, extensive	gwai-li-ja 관리자 n manager	gwang-jang 광장 n square, plaza
gwai-li-so 관리소 n management office	gwang-san 광산 n mine	gwai-lye 관례 n custom, convention	gwang-seon 광선 n ray, beam
gwai-lyeon 관련 n relation, connection ~hada -하다 v be related	gwang-taek 광택 n gloss, luster	gwai-lyeon-doe-da 관련되다 v be related (to)	gwan-gwang 관광 n tour, sightseeing ~hada -하다 v go sightseeing
gwa-mok 과목 n subject	gwan-gwang-beo-seu 관광버스 n sightseeing bus	gwa-mu-ka-da 과묵하다 adj taciturn	gwan-gwang-gaek 관광객 n tourist, sightseer

## 딕셔너리 생성

딕셔너리는 "키-값" 쌍의 모음으로, 마치 사전처럼 특정 키에 해당하는 값을 찾아볼 수 있도록 설계

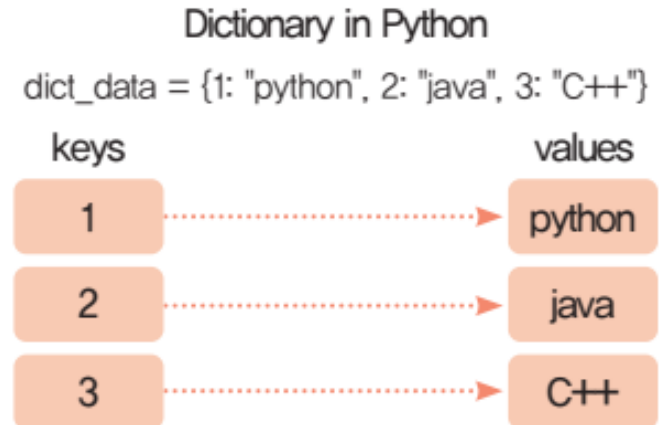
- 중괄호 { }를 사용하여 딕셔너리를 정의
  - 딕셔너리 항목의 순서는 중요하지 않으며, 키는 중복될 수 없음

dictionary 형태	
<code>{ key1: value1, key2: value2, ..., keyn: valuen }</code>	딕셔너리는 외부로 중괄호({})로 묶는다.
<ul style="list-style-type: none"> <li>• 키: 값 형태로 키와 값은 콜론으로 구분하며 항목들은 콤마로 구분</li> <li>• 메서드 <code>items()</code>로 (키, 값) 항목의 목록 형태인 <code>dict_items</code>를 반환</li> </ul>	
<code>key1, key2, ..., keyn</code>	
<ul style="list-style-type: none"> <li>• 키는 수정 불가능(또는 불변, immutable, hashable)한 자료형만 가능</li> <li>• 메서드 <code>keys()</code>로 키의 목록 형태인 <code>dict_keys</code>를 반환</li> </ul>	
<code>value1, value2, ..., valuen</code>	
<ul style="list-style-type: none"> <li>• 값은 모든 자료형이 가능</li> <li>• 메서드 <code>values()</code>로 값의 목록 형태인 <code>dict_values</code>를 반환</li> </ul>	

## 딕셔너리 자료형

- 중괄호 { }를 사용하여 딕셔너리를 정의
  - 키와 값은 콜론 : 으로 구분하고, 항목들은 콤마 , 로 구분
    - { key1: value1, key2: value2, ..., keyn: valuen }
- 키는 정수, 값은 문자열로 구성된 딕셔너리의 정의
  - 딕셔너리의 자료형은 <class 'dict'>

```
>>> dict_data = {1: 'python', 2: 'java', 3: 'C++'}
>>> dict_data
{1: 'python', 2: 'java', 3: 'C++'}
>>>
>>> type(dict_data)
<class 'dict'>
```



▲ 그림 2 딕셔너리의 이해

## 내장 함수 dict()

- 내장된 dict( ) 함수를 사용하여 딕셔너리를 생성
  - 함수 dict( ) 인자는 키, 값 쌍을 항목으로 구성되는 반복 가능(iterable)한 객체

### 내장 함수 dict()로 딕셔너리 생성

```
dict( [(key1, value1), (key2, value2), ..., (keyn, valuen)] )
dict( [[key1, value1], [key2, value2], ..., [keyn, valuen]] )
```

• 키, 값 형태로 리스트나 튜플 항목의 리스트

```
dict( ((key1, value1), (key2, value2), ..., (keyn, valuen)) )
dict( ([key1, value1], [key2, value2], ..., [keyn, valuen]) )
```

• 키, 값 형태로 리스트나 튜플 항목의 튜플

```
dict( key1=value1, key2=value2, ..., keyn=valuen )
```

• 키가 문자열이라면 키와 값을 키워드 인자 방식으로 지정 가능

## 내장 함수 dict( key1=value1, key2=value2, ..., keyn=valuen )

- dict() 함수를 사용하여 딕셔너리를 생성

- 리스트나 튜플 형태의 키-값 쌍을 인자로 받거나, 키워드 인자(key=value) 형태로도 생성 가능
  - dict( [(key1, value1), (key2, value2), ..., (keyn, valuen)] )
  - dict( key1=value1, key2=value2, ..., keyn=valuen )

```
>>> capitals = dict(미국="Washington D.C.", 이태리="Rome", 영국="London")
>>> capitals
{'미국': 'Washington D.C.', '이태리': 'Rome', '영국': 'London'}
>>> capitals['미국']
'Washington D.C.'
```

키가 문자열로 따옴표 없이 기술할 수 있다.

콜론을 사용한 [키: 값] 또는 (키: 값)의 형태가 아니라는 것에 주의하자.

## 빈 딕셔너리의 생성과 항목 추가

- 빈 중괄호 {}로 빈(empty) 딕셔너리 생성

```
>>> lect = {} # 빈 딕셔너리
>>> lect
{}

```

빈 딕셔너리를 만들어 변수 lect에 저장하는 문장이다.

- 딕셔너리에 새로운 항목을 넣으려면 '딕셔너리[새키] = 값'의 문장

```
>>> lect['강좌명'] = '인공지능 개론';

>>> lect['개설년도'] = [2026, 1];
>>> lect['학점시수'] = (3, 3);
>>> lect['교수'] = '김민국';
>>> lect
{'강좌명': '인공지능 개론', '개설년도': [2026, 1], '학점시수': (3, 3), '교수': '김민국'}
```



## 두 문자의 문자열로 딕셔너리 생성

- 함수 dict(인자)

- 각각의 한 문자가 키와 값으로 구성된 딕셔너리
- 인자: 두 문자로 구성된 문자열의 리스트나 튜플

```
>>> a = ('낮해', '밤별', '비눈')  
>>> dict(a)  
{'낮': '해', '밤': '별', '비': '눈'}
```

A diagram with two red dashed arrows. One arrow starts from the '비' character in the tuple '비눈' and points down to the '비' key in the dictionary. The other arrow starts from the '눈' character in the tuple '비눈' and points down to the '눈' value in the dictionary.

```
>>> list('낮해')  
['낮', '해']
```

## 리스트 등의 수정 가능한 객체는 딕셔너리의 키로 수정 불가능

- 키는 수정 불가능(immutable)한 자료형만 가능

- Int, float, bool, str, tuple, frozenset만 가능
  - list, set, dict는 키가 될 수 없음
- 값은 모든 자료형이 가능

```
>>> data[[3, 4]] = ('MAR', "APR")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

unhashable과 같이 수정 가능한 객체는 사용할 수 없다.

- 목록과 딕셔너리는 hash( ) 호출에서 예외가 발생

- 해시 가능과 불변은 동의어라고 생각해도 무방

```
>>> hash([1, 2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> hash({1:1, 2:4})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

## dict.fromkeys(keys, value)로 딕셔너리 생성

- 메서드 dict.fromkeys(keys, value)

- 여러 키와 동일한 값으로 구성되는 딕셔너리를 쉽게 생성
- 두 번째 인자를 10으로 지정하면 모든 값이 10으로 지정되는 딕셔너리가 생성

```
>>> foods = dict.fromkeys(['milk', 'eggs'])
>>> foods
{'milk': None, 'eggs': None}
```

```
>>> foods = dict.fromkeys(['milk', 'eggs'], 10)
>>> foods
{'milk': 10, 'eggs': 10}
```

## 딕셔너리 메서드 keys( )

- 딕셔너리 city는 우리나라의 도청 소재지 자료

```
>>> city = dict(경기='수원', 전북='전주')
>>> city
{'경기': '수원', '전북': '전주'}
```

- 딕셔너리 메서드 keys( )

- 키로만 구성된 리스트 형태의 dict\_keys 자료형을 반환

```
>>> city.keys()
dict_keys(['경기', '전북'])

>>> for key in city.keys():
...     print(f'{key}: {city[key]}')
...
경기: 수원
전북: 전주
```

- 딕셔너리 자체를 for 루프에서 사용하면 키를 순회

```
>>> for key in city:
...     print(f'{key}: {city[key]}')
...
경기: 수원
전북: 전주
```

## 딕셔너리 메소드 values() items()

- values() 메서드

- 딕셔너리의 값들을 모아 dict\_values 객체를 반환
- for 루프에서 사용하여 값들을 순회

```
>>> city = dict(경기='수원', 전북='전주')
>>> city.values()
dict_values(['수원', '전주'])
```

- items() 메서드

- 딕셔너리의 (키, 값) 쌍을 튜플 형태로 모아 dict\_items 객체를 반환
- for 루프에서 사용하여 키와 값을 동시에 순회

```
>>> city.items()
dict_items([('경기', '수원'), ('전북', '전주')])

>>> for item in city.items():
...     print(f'{item}')
...
('경기', '수원')
('전북', '전주')
```

## 딕셔너리 항목 조회

- `get(key[, default])` 메서드를 사용하여 키에 해당하는 값을 조회
  - 키가 없을 경우 `None` 을 반환하거나, 두 번째 인자로 지정된 기본값을 반환
    - `cities.get('일본')` # `None` 반환
    - `cities.get('일본', '없어요!')` # `'없어요!'` 반환

```
>>> cities = {'대한민국': ['부산', '인천'], '중국': ['상하이', '북경'], '캐나다': ['몬트리올', '오타와']}
```

```
>>> cities.get('일본')
```

```
>>> cities.get('일본', '없어요!')
```

```
'없어요!'
```

키 '일본'이 없어서 '없어요!'가 반환된다.

## 키가 없는 기본값을 지정하는 딕셔너리 메서드

- `setdefault(key[, default])` 메서드

- 키가 딕셔너리에 없는 경우
  - 키와 기본값(두 번째 인자)을 딕셔너리에 추가하고 그 값을 반환
- 키가 이미 있는 경우
  - 해당 키의 값을 반환

- `cities.setdefault('일본', ['동경', '나고야'])`

- 키 '일본' 추가하고 ['동경', '나고야'] 반환

```
>>> cities = {'대한민국': ['부산', '인천'], '캐나다': ['몬트리올', '오타와']}
>>> cities.setdefault('일본')
>>> cities
{'대한민국': ['부산', '인천'], '캐나다': ['몬트리올', '오타와'], '일본': None}
>>> cities['일본']

>>> cities = {'대한민국': ['부산', '인천'], '캐나다': ['몬트리올', '오타와']}
>>> cities.setdefault('일본', ['동경', '나고야'])
['동경', '나고야']
>>> cities
{'대한민국': ['부산', '인천'], '캐나다': ['몬트리올', '오타와'], '일본': ['동경', '나고야']}
```

## 딕셔너리 항목 삭제

- **pop() 메서드**
  - `cities.pop('중국')` # '중국' 항목 삭제하고 값 반환
  - `cities.pop('일본', '없어요!')` # '일본' 키가 없어 '없어요' 반환
- **popitem() 메서드**
  - 딕셔너리에서 마지막에 추가된 (키, 값) 쌍을 삭제하고 튜플 형태로 반환
  - 딕셔너리가 비어있으면 `KeyError` 발생
- **clear() 메서드**
  - 딕셔너리의 모든 항목을 삭제
- **del dict\_name 문장**
  - 딕셔너리 변수를 메모리에서 삭제
- **del dict\_name[key] 문장**
  - 특정 키의 항목을 삭제



## 딕셔너리 결합

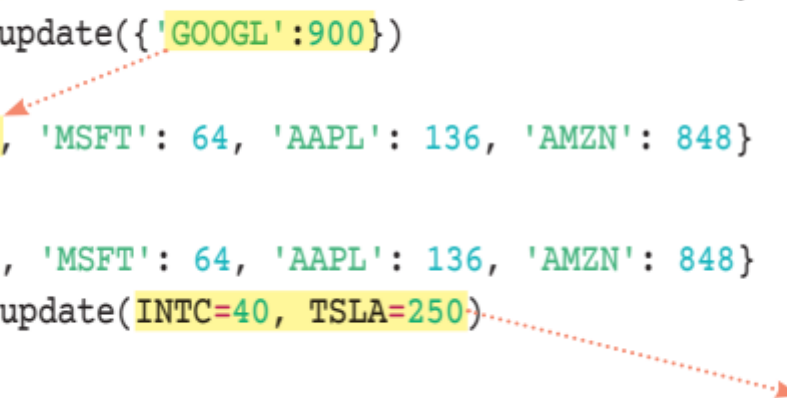
- **update(other\_dict) 메서드**

- 다른 딕셔너리의 항목을 현재 딕셔너리에 추가하거나 갱신
- 동일한 키가 있을 경우, other\_dict의 값으로 덮어 씌

```
>>> us_stock = {'GOOGL':849, 'MSFT':64}
>>> other_us_stock = {'AAPL':136, 'AMZN':848}
>>> us_stock.update(other_us_stock)
>>> us_stock
{'GOOGL': 849, 'MSFT': 64, 'AAPL': 136, 'AMZN': 848}

>>> us_stock
{'GOOGL': 849, 'MSFT': 64, 'AAPL': 136, 'AMZN': 848}
>>> us_stock.update({'GOOGL':900})
>>> us_stock
{'GOOGL': 900, 'MSFT': 64, 'AAPL': 136, 'AMZN': 848}

>>> us_stock
{'GOOGL': 900, 'MSFT': 64, 'AAPL': 136, 'AMZN': 848}
>>> us_stock.update(INTC=40, TSLA=250)
>>> us_stock
{'GOOGL': 900, 'MSFT': 64, 'AAPL': 136, 'AMZN': 848, 'INTC': 40, 'TSLA': 250}
```



## 여러 딕셔너리를 합치기와 항목 수 함수 len( )

- **\*\*dict1, \*\*dict2}와 같은 방식**

- 두 딕셔너리를 합쳐 새로운 딕셔너리를 생성

```
>>> stock1 = {'GOOGL':849, 'MSFT':64}
>>> stock2 = {'AAPL':136, 'AMZN':848}
>>> stock = {**stock1, **stock2}
>>> stock
{'GOOGL': 849, 'MSFT': 64, 'AAPL': 136, 'AMZN': 848}
```

- **len(dict) 함수**

- 딕셔너리의 항목 수를 반환

```
>>> stock
{'GOOGL': 849, 'MSFT': 64, 'AAPL': 136, 'AMZN': 848}
>>> len(stock)
4
```

## 내장 함수 min() max() 활용

- 사전 객체 subj는 세 교과목의 점수를 저장

```
>>> subj = dict(영어=80, 수학=98, 화학=65)
>>> subj
{'영어': 80, '수학': 98, '화학': 65}
```

- 내장 함수 max( )의 인자

- 딕셔너리 객체를 인자로 max(subj) 호출
  - 결과는 키의 최댓값을 반환
    - 문자열에서 문자 순서가 뒤인 문자열이 반환

```
>>> max(subj)
'화학'
```

- 함수 max(subj, key=subj.get) 호출

- 인자 key에 최댓값을 찾을 기준을 지정
- key=subj.get을 사용
  - 가장 최대인 98점의 키 '수학' 반환

```
>>> max(subj, key=subj.get)
'수학'
```

## 멤버십 검사 in

- in 연산자

- 딕셔너리에 특정 키가 존재하는지 확인, 값의 존재 여부는 확인할 수 없음
- not in 연산자
  - 딕셔너리에 특정 키가 존재하지 않는지 확인

```
>>> cp = {'한국': '서울', '중국': '북경', '독일': '베를린'}
```

```
>>> '한국' in cp
```

```
True
```

```
>>> '미국' in cp
```

```
False
```

```
>>> if '미국' not in cp:
```

```
...     cp['미국'] = '워싱턴 DC'
```

```
...
```

```
>>> cp
```

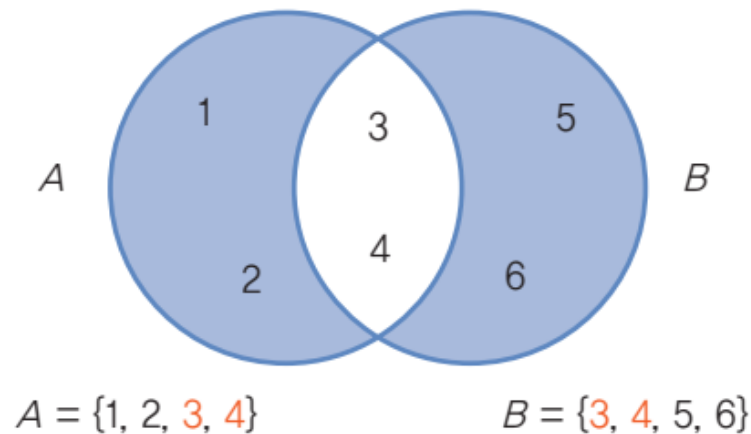
```
{'한국': '서울', '중국': '북경', '독일': '베를린', '미국': '워싱턴 DC'}
```

## Section 2. 중복과 순서가 없는 집합



## 원소는 유일하고 순서는 의미 없는 집합

- 집합은 중복되는 요소도 없고 순서도 없는 원소(elements)의 모임(collections)
  - 집합 A, B의 관계를 이해하기 쉽도록 표현한 벤 다이어그램



▲ 그림 3 수학의 집합과 벤 다이어그램

## 집합의 기본 개념 및 특징

- 집합은 {} 중괄호를 사용하여 생성하며, 각 원소는 쉼표로 구분
  - {element0, element1, element2, ..., elementn}
    - set 자료형의 이름은 set
- 특징
  - 집합의 원소는 정수, 실수, 문자열, 튜플 등 수정 불가능(immutable) 객체여야 함
    - 리스트, 딕셔너리 등 가변적인 객체는 집합의 원소가 될 수 없음
    - 원소는 int, float, str, tuple 등 수정 불가능(immutable)한 것이어야 함
  - 중첩된 집합(집합 안에 또 다른 집합을 포함)은 허용되지 않음

### 집합 set 형태

```
{ element0, element1, element2, ..., elementn }
```

중괄호 내부에서 단순 항목의 나열은 집합이다.

- 원소인 element<sub>i</sub>는 고유한(unique) 값으로 중복을 허용하지 않으며
- 원소의 순서는 의미가 없으며
- 원소는 int, float, str, tuple 등 수정 불가능(immutable)한 것이어야 함

## 집합을 만드는 내장 함수 set( )

- 집합은 내장 함수인 set(iterable)으로도 생성 가능
  - 인자는 한 개의 반복 가능한 객체

집합 set 형태
<pre>set( iterable )</pre>
<ul style="list-style-type: none"> <li>• 인자 iterable이 있으면 한 개이며 리스트와 튜플, 문자열, range 등의 반복 가능한 객체</li> <li>• 인자가 없는 set( )으로 빈 집합을 생성</li> </ul>

```
>>> set([1, 2, 3, 1])
{1, 2, 3}
```

인자의 튜플 항목은 네 개였지만, 중복을 제거하여 세 개가 되었다.

```
>>> set((1.1, 2.2, 3.3))
{1.1, 2.2, 3.3}
```

인자인 리스트나 튜플의 항목으로 구성되어 있는 집합을 생성한다.

```
>>> set(['b', 'a'])
{'a', 'b'}
```

순서는 의미 없다.

```
>>> s = set('java')
>>> s
{'a', 'v', 'j'}
```



## 함수 len( )과 소속 연산 in

- 집합 원소 수 함수 len( )

```
>>> p1 = {'fortran', 'basic', 'cobol', 'c'}
>>> p2 = {'python', 'kotlin'}
>>> len(p1)
4
>>> len(p2)
2
```

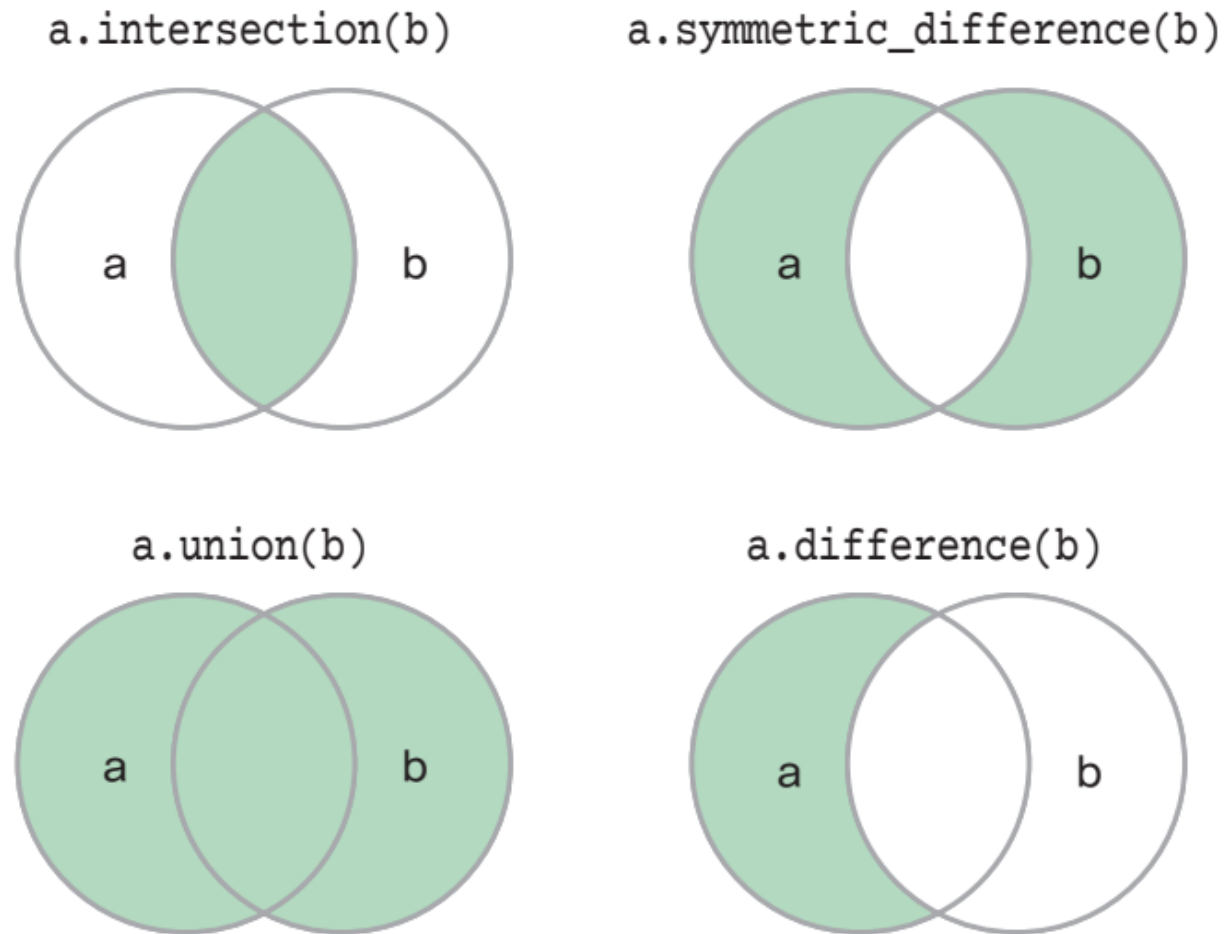
- 멤버십 연산자 in

```
>>> 'fortran' in p1
True
>>> 'fortran' in p2
False
>>> 'python' not in p1
True
```

## 메소드 `add(element)` `remove()` `copy()`

- `add(element)`: 집합에 원소를 추가
- `remove(element)`: 집합에서 특정 원소를 삭제, 원소가 없으면 `KeyError`가 발생
  - 집합 메서드 `remove(원소)`와 `discard(원소)`, `pop( )`으로 항목 삭제"
  - `discard(element)`
    - 집합에서 특정 원소를 삭제, 원소가 없어도 오류가 발생하지 않음
  - `pop()`
    - 집합에서 임의의 원소를 삭제하고 반환, 빈 집합에서 호출하면 `KeyError`가 발생
  - `clear()`
    - 집합의 모든 원소를 삭제
- `copy()`
  - 집합의 복사본을 생성

## 집합 연산과 벤 다이어그램



▲ 그림 5 집합의 벤 다이어그램

## 집합 합집합

- 합집합
  - `|`, `union()`, `update()`
    - 두 집합의 모든 원소를 포함하는 집합을 생성
- `a | b`
  - a와 b의 합집합
- `a.union(b)`
  - a와 b의 합집합을 반환하며 a는 수정하지 않음
- `a.update(b)`
  - a에 b의 합집합 결과 반영하여 a를 수정

## 교집합

- `&`, `intersection()`, `intersection_update()`
  - 두 집합에 공통으로 포함된 원소들로 구성된 집합
- `a & b`
  - a와 b의 교집합
- `a.intersection(b)`
  - a와 b의 교집합을 반환하며 a는 수정하지 않음
- `a.intersection_update(b)`
  - a에 b와의 교집합 결과를 반영하여 a를 수정

## 차집합

- `-, difference(), difference_update()`
  - 첫 번째 집합에는 포함되지만 두 번째 집합에는 포함되지 않은 원소들로 구성된 집합을 생성
- `a - b`
  - a에는 있지만 b에는 없는 원소
- `a.difference(b)`
  - a에는 있지만 b에는 없는 원소를 반환하며 a는 수정하지 않음
- `a.difference_update(b)`
  - a에 b와의 차집합 결과를 반영하여 a를 수정

## 대칭 차집합

- $\wedge$ , `symmetric_difference()`, `symmetric_difference_update()`
  - 두 집합 중 한 곳에만 포함된 원소들로 구성된 집합을 생성
- $a \wedge b$ 
  - a와 b에 각각 있지만 공통되지는 않는 원소
- `a.symmetric_difference(b)`
  - a와 b에 각각 있지만 공통되지는 않는 원소를 반환하며 a는 수정하지 않음
- `a.symmetric_difference_update(b)`
  - a에 b와의 대칭 차집합 결과를 반영하여 a를 수정

## frozenset

frozenset은 set과 유사하지만, 한 번 생성되면 내용을 변경할 수 없는 불변(immutable) 집합

- 집합의 원소나 딕셔너리의 키로 사용 가능
  - add(), remove(), clear() 등의 메서드를 지원하지 않음
  - 자료형 set에서 unhashable 자료형인 set는 원소로 사용될 수 없음
  - 집합 내부에 집합을 원소로 사용하려면 바로 frozenset를 사용
    - 자료형 frozenset는 hashable하기 때문
- 자료형 dict에서 unhashable 자료형인 set는 키로 사용될 수 없음
  - 만일, 딕셔너리의 키로 집합을 사용하려면, frozenset를 사용 가능

```
>>> fs = frozenset([1, 2, 2, 3])
>>> fs
frozenset({1, 2, 3})
>>> fs.add(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```



## Section 3. 내장 함수 zip()과 enumerate()



## 내장 함수 zip( )

- 리스트나 튜플 항목으로 조합된 항목을 생성하는 내장 함수 zip( )
  - 여러 개의 반복 가능한 객체의 항목으로 조합된 zip을 생성
  - zip(\*iterables ) 호출
    - iterables 수만큼의 튜플 항목 시퀀스를 항목으로 하는 반복자(iterator)를 반환
      - 결과의 항목인 모든 튜플의 i번째 요소는 i번째 인자로 만들어지는데,
      - 가장 적은 수의 인자에서 모두 소진될 때까지 계속

### 내장 함수 zip

```
zip( *iterables, strict=False )
```

- 항목이 각 인자인 반복 가능 항목의 i 번째 요소를 갖는 튜플로 구성되는 튜플 반복자(iterator)인 zip을 반환
- 리스트와 튜플, range, 문자열, 사전, 집합처럼 반복 가능(iterable)한 객체가 여러 개 가능
- 인자 strict는 기본이 False로 iterable의 길이가 달라도 가능
- 인자 strict=True로 지정하면 iterable의 길이가 다르면 예외 발생

## zip() 함수 호출 결과

zip 객체

- 함수 zip( )의 결과는 자료형 zip
  - 자료형 zip은 간단하게 리스트나 튜플로 변환

```
>>> data = zip('abcdefg', range(3), [1, 2, 3, 4])
>>> data, type(data)
(<zip object at 0x0000025C87F6E980>, <class 'zip'>)
>>> list(data)
[('a', 0, 1), ('b', 1, 2), ('c', 2, 3)]

>>> for t in zip(a, b):
...     print(f'서비스: {t[0]} -> 포트: {t[1]}')
...
서비스: FTP -> 포트: 20
서비스: telnet -> 포트: 23
서비스: SMTP -> 포트: 25
서비스: DNS -> 포트: 53

>>> tuple(zip('abcd', 'XY'))
(('a', 'X'), ('b', 'Y'))

>>> tuple(zip('abcd', 'XY', strict=True))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: zip() argument 2 is shorter than argument 1
```

## zip() 함수와 dict() 생성자 함께 사용

- 키-값 쌍으로 이루어진 딕셔너리를 생성

- 내장 함수 dict( )에서 zip( )의 인자 두 개를 사용하면
  - 앞은 '키', 뒤는 '값의 조합'이 되어 딕셔너리가 생성

```
>>> fields = ['name', 'age', 'job']
>>> values = ['홍길동', '25', 'Python Developer']

>>> he = dict(zip(fields, values))
>>> he
{'name': '홍길동', 'age': '25', 'job': 'Python Developer'}
```

- 주의사항

- zip() 함수는 반환값이 zip 객체(iterator)이므로
  - 결과를 사용하려면 list() 또는 tuple() 등으로 형변환 필요
- 입력되는 iterable 객체에 집합(set)을 사용할 경우 튜플 항목 순서가 입력 순서와 다를 수 있음

## enumerate() 함수

내장 함수 enumerate(시퀀스)는 0부터 시작하는 첨자와 항목 값의 튜플 목록을 생성

### • 반복 가능한 객체를 입력받아

- 각 요소와 해당 요소의 인덱스(첨자)를 튜플 형태로 묶어 새로운 반복자(iterator)를 생성
  - 기본적으로 인덱스는 0부터 시작
    - (0, seq[0]), (1, seq[1]), (2, seq[2])...
- 인자 start는 기본이 0으로, 0부터 시작해서 1씩 증가
  - 인자 start=n을 지정하면 시작이 n부터 1씩 증가

```
>>> pl = ['pyhon', 'C', 'java']
>>> enumerate(pl)
<enumerate object at 0x0000025C87F73330>
```

```
>>> list(enumerate(pl))
[(0, 'pyhon'), (1, 'C'), (2, 'java')]
```

0부터 시작하는 첨자가 자동으로 삽입된 튜플 항목이 생성된다.

```
>>> list(enumerate(pl, start = 1))
[(1, 'pyhon'), (2, 'C'), (3, 'java')]
```

첨자가 1부터 시작되도록 지정할 수 있다.

## 반복에서 사용하는 내장 함수 enumerate( )

내장 함수 enumerate(시퀀스)는 0부터 시작하는 첨자와 항목 값의 튜플 목록을 생성

### • format( ) 메서드

- \*tp로 지정하면 자동으로 나뉘어 앞부분의 { }에는 첨자, 뒤의 { }에는 값이 출력된

```
>>> seasons = ("Spring", "Summer", "Fall", "Winter")
>>> for season in enumerate(seasons, start=1):
...     print(f'{season[0]}: {season[1]}')
...     print('{ }: {}'.format(*season))
...
1: Spring
1: Spring
2: Summer
2: Summer
3: Fall
3: Fall
4: Winter
4: Winter
```

앞부분은 첨자인 season[0], 뒷부분에는 season[1]인 값이 출력된다.