변수와 자료형 (4)

PDF	비어 있음
비고	집합, 딕셔너리
숫자	6
실습문제	비어 있음
실습문제답안	비어 있음

06. 셋 (Set)

06-01. 셋 개요

06-01-01. 셋이란

💡 중복된 요소를 허용하지 않으며 순서 없이 요소를 저장하는 컬렉션이다. 따라서 중복 제거가 필요할 때 유용하게 사용할 수 있다.

06-01-02. 셋 표현식

- 중괄호{}를 사용해서 집합을 생성한다.

```python
# 문자열로 집합 생성 safari_set = {"Bear", "Koara", "Deer", "Squirrel"} # 숫자 집합 nu
mbers = {1, 2, 3, 4, 5} # 혼합된 타입 집합 mixed_set = {1, "Bear", (1, 2, 3)}
```

- set() 을 사용해 형 변환도 가능하다.

```python
# 리스트로 집합 생성 another_safari_set = set(["Monkey", "Tiger", "Wolf"])
```

▶ *중복 제거를 위한 set() 함수 활용*

06-01-03. 셋의 특징

1. 순서가 없는 자료형이며, 순서가 없으므로 인덱스를 사용한 접근이 불가능하다.

```python
ohgiraffers = {'Pig', 'Squirrel', 'Bear', 'Deer'} print(ohgiraffers) # {'Bear', 'Squi
rrel', 'Deer', 'Pig'} # print(ohgiraffers[0]) # Error 발생
```

2. 중복을 허용하지 않으므로 동일한 값은 하나만 저장된다.

```python
ohgiraffers = {'Pig', 'Squirrel', 'Bear', 'Deer', 'Deer'} print(ohgiraffers) # {'Bea
r', 'Squirrel', 'Deer', 'Pig'}
```

3. 가변성을 가진 자료형으로 데이터, 즉 요소를 추가하거나 제거할 수 있다.
   - 순서가 없는 자료형으로 특정 요소에 접근할 수 없으므로 직접 변경은 할 수 없다.

```
ohgiraffers = {'Pig', 'Squirrel', 'Bear', 'korilla'} ohgiraffers.remove('korilla') # set의 데이터 중 remove의 값 제거 ohgiraffers.add("Deer") # set에 add('값')으로 값을 추가 print(ohgiraffers) # {'Bear', 'Squirrel', 'Deer', 'Pig'}
```

4. 다양한 데이터 타입을 지원한다.

```
ohgiraffers = {1, "Squirrel", 2, 'Bear', 3.0, 'pig', tuple({'Deer'}), True} print(ohgiraffers) # {1, 2, 3.0, 'Squirrel', 'pig', 'Bear', ('Deer',)}
```

5. 수학적 집합 연산(합집합, 교집합, 차집합 등)을 지원한다.

6. in 키워드를 통해 포함 여부를 반환받을 수 있다.

```
javaTeam = {'Squirrel','Deer', 'Tiger', 'Sheep', 'monkey', 'wolf'} print('Deer' in javaTeam) # True
```

06-02. 셋의 메서드

06-02-01. 요소를 추가하는 메서드

1. add(값) : set의 list에 인자로 전달받은 값을 추가한다.

```
ohgiraffers = {'Pig', 'Squirrel', 'Bear', 'Deer'} ohgiraffers.add("Elephant") print(ohgiraffers) # {'Deer', 'Squirrel', 'Elephant', 'Pig', 'Bear'}
```

2. update(값) : 요소 값을 추가한다.
   - 배열을 이용해 한번에 여러 개의 값을 추가하는 것도 가능하나, 이때도 중복된 값은 추가되지 않는다.

```
ohgiraffers = set(["Monkey", "Tiger", "Wolf"]) print(ohgiraffers) # {'Tiger', 'Monkey', 'Wolf'} ohgiraffers.update(["Monkey", "Wolf", "Tiger", "Squirrel"]) print(ohgiraffers) # {'Tiger', 'Squirrel', 'Monkey', 'Wolf'}
```

06-02-02. 요소를 제거하는 메서드

1. remove(값) : 특정 요소를 제거하며, 값이 존재하지 않으면 Error를 발생시킨다.

```
ohgiraffers = {'Pig', 'Squirrel', 'Bear', 'Deer'} # ohgiraffers.remove('Elephant') # Error 발생 ohgiraffers.remove('Pig') print(ohgiraffers) # {'Bear', 'Squirrel', 'Deer'}
```

2. discard(값) : 특정 요소를 제거하며, 값이 존재하지 않아도 Error가 발생하지 않는다.

```
ohgiraffers.discard("Elephant") print(ohgiraffers) # {'Bear', 'Squirrel', 'Deer'}
```

3. pop() : 임의의 값을 제거한다.
   - 집합은 순서를 보장하지 않으므로, 어떤 값이 제거될지 예측할 수 없다.

```
ohgiraffers.pop() print(ohgiraffers) # {'Squirrel', 'Deer'}
```

4. clear()는 모든 값을 제거한다.

```
ohgiraffers.clear() print(ohgiraffers) # set()
```

06-02-03. 집합 연산 메서드

1. union() : 두 set 자료형을 합친다. (= 합집합)

```
javaTeam = {'Squirrel','Deer', 'Tiger', 'Sheep', 'monkey', 'wolf'} pythonTeam = {'Pi
g', 'Squirrel', 'Bear', 'Deer'} ohgiraffers = javaTeam.union(pythonTeam) print(javaTe
am | pythonTeam) # {'wolf', 'Sheep', 'Tiger', 'Deer', 'monkey', 'Squirrel', 'Pig', 'B
ear'} print(ohgiraffers) # {'wolf', 'Sheep', 'Tiger', 'Deer', 'monkey', 'Squirrel',
'Pig', 'Bear'}
```

2. intersection() : 두 set 자료형의 교집합을 반환한다.

```
print(javaTeam & pythonTeam) # {'Squirrel', 'Deer'} print(javaTeam.intersection(pytho
nTeam)) # {'Squirrel', 'Deer'}
```

3. difference() : 두 set 자료형의 차집합을 반환한다. (좌항을 기준으로 우항의 차집합을 반환한다.)

```
print(javaTeam - pythonTeam) # {'monkey', 'wolf', 'Sheep', 'Tiger'} print(javaTeam.di
fference(pythonTeam)) # {'monkey', 'wolf', 'Sheep', 'Tiger'}
```

4. symmetric_difference() : 대칭 차집합을 반환한다. (양쪽 모두의 차집합을 반환한다.)

```
print(javaTeam ^ pythonTeam) # {'wolf', 'Sheep', 'Tiger', 'monkey', 'Pig', 'Bear'} pr
int(javaTeam.symmetric_difference(pythonTeam)) # {'wolf', 'Sheep', 'Tiger', 'monkey',
'Pig', 'Bear'}
```

06-02-04. 복사 메서드

1. copy() : 대상 set을 복사하여 반환한다.

```
ohgiraffers = {'Squirrel','Deer', 'Tiger', 'Sheep', 'monkey', 'wolf'} zzap_ohgiraffer
s = ohgiraffers.copy() print(ohgiraffers) # {'wolf', 'Sheep', 'Tiger', 'Deer', 'monke
y', 'Squirrel'} print(zzap_ohgiraffers) # {'wolf', 'Sheep', 'Tiger', 'Deer', 'monke
y', 'Squirrel'} zzap_ohgiraffers.remove('Tiger') print(ohgiraffers) # {'wolf', 'Shee
p', 'Tiger', 'Deer', 'monkey', 'Squirrel'} print(zzap_ohgiraffers) # {'wolf', 'Shee
p', 'Deer', 'monkey', 'Squirrel'}
```

---

07. 딕셔너리 (Dictionaries)

07-01. 딕셔너리 개요

07-01-01. 딕셔너리란

💡 키(key)와 값(value)의 쌍으로 구성된 자료형(데이터 구조)로, 키를 통해 값을 찾을 수 있으므로 매우 빠른 조회 성능을 보여준다. 연관 배열(associative array) 또는 해시(hash)라고도 한다.

07-01-02. 딕셔너리 표현식

```
딕셔너리명 = { 키1: 값1, 키2: 값2, ... }
```

```
teacher = {'name': 'deer', 'team': 'ohgiraffers'} print(type(teacher)) # <class 'dict'> #
key 값을 활용한 value 탐색 print(teacher['name']) # deer print(teacher['team']) # ohgiraf
fers
```

07-01-03. 딕셔너리의 특징

1. 키(key)와 값(value)의 쌍으로 구성된다.
   - 사전에서 단어를 찾아 뜻을 확인하는 것처럼, Dictionaries에서 키를 사용하여 해당 값을 손쉽게 찾고 활용할 수 있다.

```
ohgiraffers = { 'tiger' : 'langChain', 'Squirrel' :'JPA', 'Bear':'Python', 'Deer':'Sp
ring' } print(ohgiraffers) # {'tiger': 'langChain', 'Squirrel': 'JPA', 'Bear': 'Pytho
n', 'Deer': 'Spring'} print(ohgiraffers['tiger']) # langChain print(ohgiraffers['Squi
rrel']) # JPA
```

2. 키는 고유해야 하며 불변(immutable)한 데이터 타입이어야 한다.
   - 각 키는 유일하게 존재해야 하므로, 동일한 키로 여러 값을 저장할 수 없다.
     - 만약 중복 키로 값을 입력하면, 기존 값을 덮어쓴다.
     - 이러한 특징은 데이터 검색 및 관리의 효율성을 높여준다.
   - key로 가변 자료형(리스트, 딕셔너리)는 사용할 수 없고, 기본 자료형이나 불변 자료형(튜플)은 사용할 수 있다.

```
ohgiraffers = { 'tiger':'langChain', 'Squirrel':'JPA', 'Bear':'Python', 'Deer':'Sprin
g', "Squirrel":"MyBatis" } print(ohgiraffers['Squirrel']) # MyBatis # invalid_key_tea
cher = {["name", "team"] : 'bear.ohgiraffer'} # Error 발생 valid_key_dic = {("name",
"team") : 'bear.ohgriaffers'} print(valid_key_dic[("name", "team")]) # bear.ohgriaffe
rs
```

3. Dictionaries는 가변적인 자료구조이다.
   - 딕셔너리는 가변적인 자료구조로, Dictionaries 생성 후에도 키-값 쌍을 추가/삭제/수정할 수 있다.
   - 이는 상황에 따라 데이터를 동적으로 변경해야 하는 경우 유용하게 활용될 수 있다.

```
ohgiraffers = { 'tiger' : 'langChain', 'Squirrel' :'JPA', 'Bear':'Python', 'Deer':'Sp
ring' } print(ohgiraffers['tiger']) # langChain ohgiraffers['tiger'] = 'java' print(o
hgiraffers['tiger']) # java
```

4. 빠른 조회와 수정이 가능하다.
   - 딕셔너리는 해시 테이블을 사용하여 구현되므로, 평균적으로 O(1)의 시간 복잡도로 항목을 조회하고 수정할 수 있다.

5. del 키워드를 이용해 딕셔너리를 삭제할 수 있다.

```
del ohgiraffers # print(ohgiraffers) # Error 발생
```

6. in 키워드를 사용하여 key 값의 존재 여부를 확인할 수 있다.

```
ohgiraffers = { 'tiger' : 'langChain', 'Squirrel' :'JPA', 'Bear':'Python', 'Deer':'Sp
ring' } print('tiger' in ohgiraffers) # True
```

▶ *[심화] 딕셔너리의 함수 저장*

07-02. 딕셔너리의 메서드

07-02-01. get()

- get(키)는 매개변수로 전달받은 키에 해당하는 값을 반환한다.
- 만약 해당 키가 존재하지 않으면 기본값(None)을 반환한다.

```
ohgiraffers = { 'tiger' : 'langChain', 'Squirrel' :'JPA', 'Bear':'Python', 'Deer':'Sp
ring' } print(ohgiraffers['tiger']) # langChain print(ohgiraffers.get('tiger')) # lan
gChain print(ohgiraffers.get('test')) # None
```

07-02-02. keys()

- 딕셔너리의 모든 키를 반환한다.

```
ohgiraffers = { 'tiger' : 'langChain', 'Squirrel' :'JPA', 'Bear':'Python', 'Deer':'Sp
ring' } print(ohgiraffers.keys()) # dict_keys(['tiger', 'Squirrel', 'Bear', 'Deer'])
```

07-02-03. values()

- 딕셔너리의 모든 값을 반환한다.

```
ohgiraffers = { 'tiger' : 'langChain', 'Squirrel' :'JPA', 'Bear':'Python', 'Deer':'Sp
ring' } print(ohgiraffers.values()) # dict_values(['langChain', 'JPA', 'Python', 'Spr
ing'])
```

07-02-04. items()

- 딕셔너리의 모든 항목(키-값 쌍)을 반환한다.

```
ohgiraffers = { 'tiger' : 'langChain', 'Squirrel' :'JPA', 'Bear':'Python', 'Deer':'Sp
ring' } print(ohgiraffers.items()) # dict_items([('tiger', 'langChain'), ('Squirrel',
'JPA'), ('Bear', 'Python'), ('Deer', 'Spring')])
```

07-02-05. pop()

- pop(키)는 매개변수의 값을 제거하고 반환한다.
- 키가 존재하지 않으면 기본값을 반환하고, 만약 기본값이 지정되지 않으면 'KeyError'가 발생한다.

```
ohgiraffers = { 'tiger' : 'langChain', 'Squirrel' :'JPA', 'Bear':'Python', 'Deer':'Sp
ring' } print(ohgiraffers.pop('tiger')) # langChain # print(ohgiraffers.pop('test'))
# Error 발생 print(ohgiraffers) # {'Squirrel': 'JPA', 'Bear': 'Python', 'Deer': 'Spri
ng'}
```

- del 키워드를 이용해 특정 키의 키-값 아이템만 제거할 수도 있다.

```
del ohgiraffers['Squirrel'] print(ohgiraffers) # {'Bear': 'Python', 'Deer': 'Spring'}
```

07-02-06. popitem()

- 마지막 요소의 key-value 값을 제거하고 튜플 형태로 반환한다.

```
ohgiraffers = { 'tiger' : 'langChain', 'Squirrel' :'JPA', 'Bear':'Python',
'Deer':'Spring' } poped_item = ohgiraffers.popitem() print(poped_item) # ('Deer',
'Spring') print(type(poped_item)) # <class 'tuple'> print(ohgiraffers) # {'tiger':
'langChain', 'Squirrel': 'JPA', 'Bear': 'Python'}
```

07-02-07. update()

- 다른 딕셔너리나 키-값 쌍의 iterable을 사용하여 딕셔너리 요소를 추가한다.

```
ohgiraffers = { 'tiger' : 'langChain', 'Squirrel' :'JPA', 'Bear':'Python', 'Deer':'Sp
ring' } ohgiraffers.update({'test':'value'}) print(ohgiraffers) # {'tiger': 'langChai
n', 'Squirrel': 'JPA', 'Bear': 'Python', 'Deer': 'Spring', 'test': 'value'}
```

07-02-08. clear()

- 모든 항목을 제거한다.

```
ohgiraffers = { 'tiger' : 'langChain', 'Squirrel' :'JPA', 'Bear':'Python', 'Deer':'Sp
ring' } ohgiraffers.clear() print(ohgiraffers) # {}
```

[참고] zip()

```
k = ['one', 'two', 'three'] v = [1,2,3] # 두개의 자료를 순서대로 쌍으로 묶은 zip객체를 반환
t = zip(k,v) print(t) #<zip object at 0x034ABE90> print(type(t)) #<class 'zip'> for k,v
in t: print(k,v) # one 1 # two 2 # three 3
```

[참고] 딕셔너리의 복사

- 얕은 복사

```
# 1.shallow copy a = {'a': 1, 'b': 2, 'c': 3} b = a # 사전 레퍼런스의 복사 (공유됨)
print(a) print(b) a['a'] = 100 print(a) print(b)
```

- 깊은 복사

```
# 2.deep copy c = {'a': 1, 'b': 2, 'c': 3} d = c.copy() print(c) print(d) c['a'] =
300 print(c) print(d) # 변경되지 않음.
```

- 복사 시 알아둘 것

```
# value에 list가 있다면 copy메소드를 사용해도 list는 얕은 복사되어 변수 간 공유된다. a =
{'a': [1,2,3], 'b':100} b = a.copy() print(a) #{'a': [1, 2, 3], 'b': 100} print(b) #
{'a': [1, 2, 3], 'b': 100} a['b'] = 200 a['a'][0] = 999 print(a) #{'a': [999, 2, 3],
'b': 200} print(b) #{'a': [999, 2, 3], 'b': 100} # value에 dict가 있다면 a = {'user':
{'name':'홍길동','age':20},'amount':123} b = a.copy(); print(a) print(b) a['user']
['name'] = '홍길은' print(a) print(b)
```

[참고] 딕셔너리의 병합

```
# update 메소드를 통한 두 dict의 병합 # 동일한 키값이 있다면, 인수로 주어진 dict의 아이템으로
덮어쓰기 된다. dic1 = {'a':10, 'b':20} dic2 = {'c':100, 'd':300} dic1.update(dic2)
print(dic1) #{'a': 10, 'b': 20, 'c': 100, 'd': 300} print(dic2) #{'c': 100, 'd': 300}
```

[참고] 딕셔너리의 정렬

```
# dict의 정렬 : sorted numbers = {'first': 1, 'second': 2, 'third': 3, 'Fourth': 4}
numbers['hundred'] = 100 numbers['fifth'] = 5 print(list(numbers)) #['first', 'second',
'third', 'Fourth', 'hundred', 'fifth'] # key값 정렬 print(sorted(numbers)) #['Fourth',
'fifth', 'first', 'hundred', 'second', 'third'] # value값 정렬
print(sorted(numbers.values())) #[1, 2, 3, 4, 5, 100] # key정렬 (value값 정렬에 따라)
print(sorted(numbers, key=numbers.__getitem__))
```