08. 사전

사전은 특정 키를 주면 이와 관련된 값을 돌려주는 내용 기반으로 검색하는 자료형이다.

8.1 사전의 연산

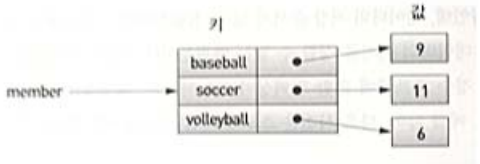
사전은 임의 객체의 집합적 자료형인데, 데이터의 저장 순서가 없다. 집합적이라는 의미에서 리스트나 튜플과 동일한. 하지만, 데이터의 순서를 정할수 없는 매핑형이다. 시퀀스 자료형은 데이터의 순서를 정할수 있어서 정수 오프셋에 의한 인덱싱이 가능하지만, 매핑형에서는 키를 이용해 값에 접근한다. 예를들어, 다음 사전은 스포츠 게임 종목에 대한 참여 선수의 수이다.

```
>>> member = {'basketball':5, 'soccer':11, 'baseball':9}
>>> member['baseball']                          # 검색
9
```

member라는 사전은 세 개의 입력값을 가지고 있다. 각각의 입력 값은 키와 값으로 구분된다. 첫 번째 입력값은 basketball이라는 키와 5라는 값이다. 사전에서 값을 꺼내려면 키를 사용한다. 사전은 변경 가능한 자료형으로 값을 저장할때도 키를 사용한다. 키가 사전에 등록되어 있지 않으면 새로운 항목이 만들어지며, 키가 이미 사전에 등록되어 있으면, 이 키에 해당하는 값이 변경된다.

```
>>> member['bolleyball'] = 7                    # 새 값을 설정한다.
>>> member['volleyball'] = 6                    # 값을 변경한다.
>>> member
{'basketball': 5, 'soccer': 11, 'baseball': 9, 'bolleyball': 7, 'volleyball': 6}
>>> len(member)                                 # 길이 정보를 확인한다.
5
>>> del member ['basketball']                   # 항목을 삭제한다.
>>> 'soccer' in member                          # 멤버를 검사한다...
True
```

사전을 출력하면 당연히 어떤 순서에 의해서 입력 값들이 표현된다. 그러나 이런 순서는 고정된 것이 아니다. 키에 의한 검색 속도를 빠르게 하기 위해 사전은 내부적으로 해시 기법을 이용하여 데이터를 저장한다. 이 기법은 데이터의 크기가 증가해도 빠른 속도로 데이터를 찾을수 없게 해준다. 사전의 구조를 그림으로 표현하면 그림 8-1과 같다.



값은 임의의 객체가 될수 있지만, 키는 해시 가능이고 변경 불가능한 자료형이어야 한다. 예를 들어, 문자열과 숫자, 튜플은 키가 될수 있지만, 리스트와 사전은 키가 될수 없다.

```
>>> d = {}
>>> d['str'] = 'abc'
```

```
>>> d[1] = 4
>>> d[(1, 2, 3)] = 'tuple'
>>> d[[1, 2, 3]] = 'list'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> d[{1:2}] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

함수를 키나 값으로 활용할 수도 있다. 다음은 함수를 값으로 활용한 예이다. action[0]은 add() 함수와 같고, action[1]은 sub()함수와 같다.

```
>>> def add(a, b):
...     return a + b
...
>>> def sub(a, b):
...     return a - b
...
>>> action = {0:add, 1:sub}
>>> action[0](4, 5)
9
>>> action[1](4, 5)
-1
```

사전의 모든 값을 순차적으로 참조하는데 사용하는 일반적인 방법은 사전의 반복자를 이용하는 것이다. 사전 자체를 for 문에 이용하면 키에 대한 반복이 실행된다.

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> for key in D:              # for key in D.keys():와 동일한 결과를 얻는다.
...     print(key, D[key])
...
a 1
b 2
c 3
```

items() 메서드를 사용하여 key 와 value항목을 함께 꺼내 올수도 있다. items() 메서드는 (키, 값)의 사전 뷰를 반환한다.

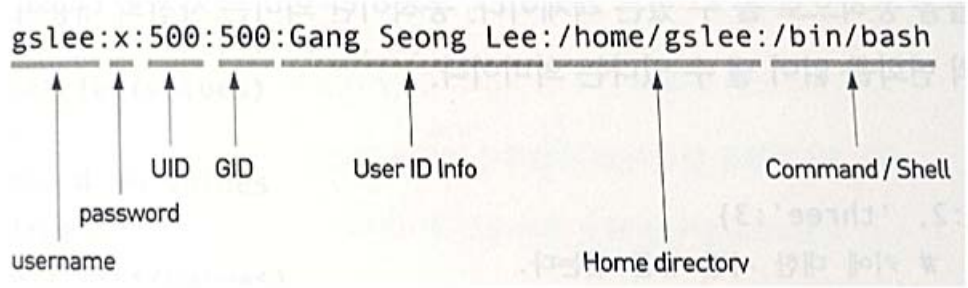
```
>>> D.items()
dict_items([('a', 1), ('b', 2), ('c', 3)])
```

```
>>> for (key, value) in D.items():
...     print(key, value)
...
a 1
b 2
c 3
```

사전을 사용하는 예로 리눅스의 passwd 파일을 사전으로 읽는 코드를 살펴보자. 리눅스의 /etc/passwd 파일은 다음과 같은 내용으로 되어 있다.

```
root:x:0:0:root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
~ 생략 ~
```

사용하는 형식은 다음과 같다.



여기서 패스워드 x의 의미는 실제 패스워드는 /etc/shadow 파일에 암호화되어 저장되어 있다는 의미이다. /etc/passwd파일을 읽어서 사전에 등록해 보자.

```
>>> pwd_dict = {}                          # 빈 사전을 정의한다.
>>> for line in open( '/etc/passwd' ):  # 각 줄에 대해서
        fields = line.rstrip().split( ':' )# 오른쪽 공백을 없애고  ':' 로 분리한다.
        pwd_dict[fields[0]] = fields[1:]   # username을 키로, 나머지를 값으로 등록한다.

>>> pwd_dict[ 'root' ]          # 루트에 관한 정보이다.
[ 'X' , '0' , '0' , 'root' , '/root' , '/bin/bash:]
>>> pwd_dict[ 'gslee' ]         # gslee에 관한 정보이다.
[ 'x' , '500' , '500' , 'Gang Seong Lee' , '/home/gslee' , '/bin/bash' ]
 '500'
>>> pwd_dict[ 'gslee' ][5]      # gslee의 shell이다.
 '/bin/bash'
```

8.2 사전의 뷰

사전의 메서드로는 여러개가 있지만, 우선 keys()와 values(), items() 메서드를 살펴보자.

- keys() 메서드 키에 대한 사전 뷰를 반환한다.
- values() 메서드 값에 대한 사전 뷰를 반환한다.
- items() 메서드 항목들의 사전 뷰를 반환한다.

여기서 뷰란 사전 항목들을 동적으로 볼수 있는 객체이다. 동적이란 의미는 사전의 내용이 바뀌어도 뷰를 통해서 내용의 변화를 읽어 낼수 있다는 의미이다.

```
>>> d = {'one':1, 'two':2, 'three':3}
>>> keys = d.keys()             # 키에 대한 사전 뷰를 얻는다.
>>> keys                        # 값을 확인한다.
dict_keys(['one', 'two', 'three'])
>>> d['four'] = 4
>>> keys                        # 뷰도 동적으로 변경된다.
dict_keys(['one', 'two', 'three', 'four'])
```

뷰객체는 인덱싱이 가능하지 않지만, 리스트로 변환하거나 멤버 검사가 가능하고 반복자를 제공한다.

```
>>> list(keys)                  # 리스트로 변환한다.
['one', 'two', 'three', 'four']
>>> len(keys)                   # 항목수를 반환한다.
4
>>> 'one' in keys               # 멤버 검사가 가능하다.
True
>>> iter(keys)                  # 반복자 객체로 사용할수 있다
<dict_keyiterator object at 0x000001470EF729F8>
>>> for k in keys:              # 반복문에서 사용한 예이다.
...     print(k)
...
one
two
three
four
```

values()와 items() 메서드에 대해서도 keys() 메서드와 같은 연산이 가능하다.

```
>>> values = d.values()
>>> values
dict_values([1, 2, 3, 4])
>>> len(values)
4
```

```
>>> 4 in values
True
>>> list(values)
[1, 2, 3, 4]
>>>
>>> items = d.items()
>>> items
dict_items([('one', 1), ('two', 2), ('three', 3), ('four', 4)])
>>> len(items)
4
```

키와 항목에 대한 사전 뷰들은 항목들이 유일하고 중복되지 않는다는 점에서 집합과 유사한 특성이 있다. 사전의 뷰 객체들은 집합 연산을 허용한다.

```
>>> keys
dict_keys(['one', 'two', 'three', 'four'])
>>> keys & {'one','two','five'}                # 교집합
{'two', 'one'}
>>> keys - {'one','two','five'}                # 차집합 등
{'four', 'three'}
```

8.3 사전의 메서드

다음은 사전의 뷰객체를 얻어내는 keys() values(), items() 이외의 메서드에 대하여 간략하게 요약한 것이다. D는 사전객체이다.

- D.clear() — 사전 D의 모든 항목을 삭제한다.
- D.copy() — 사전을 복사한다. 얕은 복사에 해당한다.
- D.get(key [,x]) — 값이 존재하면 D[key]를 반환하고 아니면 x를 반환한다.
- D.setdefault(key [, x]) — get() 메서드와 같으나 값이 존재하지 않을 때 값을 설정한다. D[key]=x
- D.update(b) — 사전 b의 모든 항목을 D에 갱신한다. for k in b.keys(): D(k)=b[k]
- D.popitem() — (키,값) 튜플을 반환하고 사전에서 제거한다.
- D.pop(key) — key 항목의 값을 반환하고 사전에서 제거한다.
- D.fromkeys(seq[,value]) — fromkeys() 메서드는 클래스 메서드다. seq와 value를 이용하여 만든 새로운 사전을 반환한다.

이들 메서드를 사용한 예이다.

```
>>> d = {'one':1, 'two':2, 'three':3}
>>> d2 = d.copy()                          # 사전을 복사한다.
>>> d['four']=4                            #  d에 새 항목을 추가한다.
>>> d2
{'one': 1, 'two': 2, 'three': 3}
>>> d3 = {'inie':9, 'ten':10}
>>> d.update(d3)                           # d3의 사전 내용을 d로 갱신한다.
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'inie': 9, 'ten': 10}
>>> d.popitem()                            # 항목 한 개를 꺼낸다.
('ten', 10)
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'inie': 9}
>>> d.pop('two')
2
>>> d
{'one': 1, 'three': 3, 'four': 4, 'inie': 9}
```

다음은 get()과 setdefault() 메서드를 사용한 예이다.

```
>>> d = {'one':1, 'two':2,'three':3}
>>> d.get('one')              # d[ 'one ']과 동일하다.
1
>>> d.get('tne')              #  'ten '이 없을 경우 None을 default 값으로 반환한다.
>>> d.get('ten',10)           # ' ten' 이 없을 경우 10을 default 값으로 반환한다.
10
```

```
>>> d
{'one': 1, 'two': 2, 'three': 3}
>>> d.setdefault('ten',10)      # ' ten' 이 없을 경우 새로운 항목이 생긴다.
10
>>> d
{'one': 1, 'two': 2, 'three': 3, 'ten': 10}
```

8.4 사전 내장

사전내장은 리스트 내장과 유사한 방식으로 동작하지만 사전을 만들어 낸다. 사전 내장은 중괄호[]를 사용하며 키:값 형식으로 항목을 표현한다.

```
>>> {w:1 for w in 'abc'}                # 키:값 = w:1
{'a': 1, 'b': 1, 'c': 1}
>>>
>>> a1 = 'abcd'
>>> a2 = (1, 2, 3, 4)
>>> { x:y for x,y in zip(a1, a2) }
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> { w:k for k, w in [(1, 'one'),(2,'two'),(3,'three')]}
{'one': 1, 'two': 2, 'three': 3}
>>> { w:k + 1 for k, w in enumerate(['one','two','three'])}
{'one': 1, 'two': 2, 'three': 3}
```

다음예는 사전내장을 이용하여 어떤 사전의 값들을 키로, 또 키들을 값으로 하는 사전을 만든다.

```
>>> d = {'one':1,'two':2,'three':3}
>>> {v:k for k,v in d.items()}
{1: 'one', 2: 'two', 3: 'three'}
```

다음은 다중 for 문을 이용한 사전 내장의 예이다.

```
>>> {(k, v):k+v for k in range(3) for v in range(3)}
{(0, 0): 0, (0, 1): 1, (0, 2): 2, (1, 0): 1, (1, 1): 2, (1, 2): 3, (2, 0): 2, (2, 1): 3, (2, 2): 4}
```

8.5 심볼테이블

심볼테이블이란 변수들이 저장되는 공간이다. 파이썬에서는 모든 심볼 테이블을 저장하는데 사전을 사용한다. 심볼 이름은 사전의 키로, 심볼값은 사전의 값으로 등록된다. 예를들어, a=1이라고 했을 때 {fa' :1}이란 항목이 만들어진다.

이름	값
a	1
b	2
pi	3.14

- 전역/지역 심볼 테이블

globals() 함수를 사용하면 전역 영역(모듈 영역)의 심볼 테이블(사전)을 얻는다.

```
>>> a=1
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class
'_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module
'builtins' (built-in)>, 'D': {'a': 1, 'b': 2, 'c': 3}, 'key': 'c', 'value': 3, 'd': {'one': 1, 'two':
2, 'three': 3}, 'keys': dict_keys(['one', 'two', 'three', 'four']), 'k': 'four', 'values':
dict_values([1, 2, 3, 4]), 'items': dict_items([('one', 1), ('two', 2), ('three', 3), ('four', 4)]),
'd2': {'one': 1, 'two': 2, 'three': 3}, 'de': {'inie': 9, 'ten': 10}, 'd3': {'inie': 9, 'ten': 10},
'a1': 'abcd', 'a2': (1, 2, 3, 4), 'a': 1}
```

globals() 함수로부터 반환된 사전은 앞에서 정의한 변수 이외에도 _doc_나 _name_과 같은 내장 이름이 들어 있다.

locals() 함수를 사용하면 지역 영역의 심볼 테이블(사전)을 얻는다.

```
>>> def f(a,b):
...     c = 10
...     print(locals())
...
>>> f(2, 3)
{'c': 10, 'b': 3, 'a': 2}
```

- 객체의 심볼 테이블

이름 공간(심볼이 저장되는 공간)을 가지는 모든 객체는 심볼 테이블을 갖는다. 모듈과 함수, 클래스, 클래스 인스턴스, 함수 모두 이름 공간을 갖는다. 어떤 객체의 심볼 테이블은 _dict_ 속성을 확인해 보면 된다. 다음 예는 모듈과 클래스, 클래스 인스턴스의 심볼 테이블을 보여준다.

```
>>> import re
>>> re.__dict__                    # 모듈의 심볼 테이블이다.
```

{'__name__': 're', '__doc__': 'Support for regular expressions (RE).₩n₩nThis module provides regular expression matching operations similar to₩nthose found in Perl. It supports both 8-bit and Unicode strings; both₩nthe pattern and the strings being processed can contain null bytes and₩ncharacters outside the US ASCII range.₩n₩nRegular expressions can contain both special and ordinary characters.₩nMost ordinary characters, like "A", "a", or "0", are the simplest₩nregular expressions; they simply match themselves. You can₩nconcatenate ordinary characters, so last matches the string ₩'last₩'.₩n₩nThe special characters are:₩n   "."
…중략…

```
>>> class C:
...     x = 10
...     y = 20
...
>>> C.__dict__          # 클래스의 심볼 테이블을 얻는다.
mappingproxy({'__module__': '__main__', 'x': 10, 'y': 20, '__dict__': <attribute '__dict__' of 'C' objects>, '__weakref__': <attribute '__weakref__' of 'C' objects>, '__doc__': None})
>>> c = C()
>>> c.a = 100
>>> c.b = 200
>>> c.__dict__          # 클래스 인스턴스의 심볼 테이블을 얻는다.
{'a': 100, 'b': 200}
```

함수도 이름 공간이다. 따라서 함수에도 속성값을 지정할수 있다. 다음은 함수의 심볼 테이블을 얻는 예이다.

```
>>> def f():
...     pass
...
>>> f.a = 1
>>> f.b = 2
>>> f.__dict__          # 함수의 심볼 테이블을 얻는다.
{'a': 1, 'b': 2}
```

이름공간에서 어떤 이름의 값을 얻어내는 방법에는 다음과 같은 것들이 있다. 다음의 세가지 방법은 모두 동일한 결과를 낸다.

```
>>> import math
>>> math.sin
<built-in function sin>
>>> math.__dict__['sin']
<built-in function sin>
>>> getattr(math,'sin'  )              # getattr(이름공간, 이름)
<built-in function sin>
```

여기서 getattr() 함수는 문자열로 주어진 이름의 객체를 반환한다. 반대로 이름 공간에 값을 설정할수도 있다. 다음 세가지 방법은 모두 동일한 결과를 낸다.

```
>>> math.mypi = 3.14                    # ①
>>> math.__dict__['mypi'] = 3.14                # ②
>>> setattr(math, 'mypi', 3.14)         # ③
>>> math.mypi
3.14
```

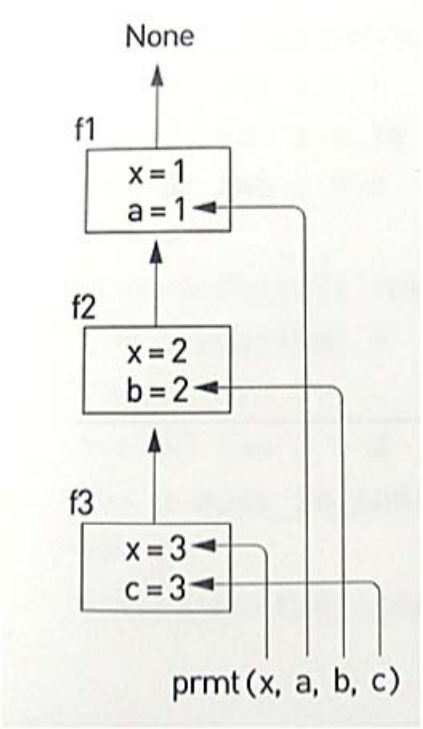
모듈 내에서 자신의 모듈을 참조하는 방법은 다음과 같다. 다음 세가지 방법도 동일한 결과를 낸다.

```
>>> import sys
>>> current_module = sys.modules[__name__]      # ①
>>> a = 10
>>> getattr(current_module, 'a')                # ②
10
>>> current_module.__dict__['a']                # ③
10
```

8.6 예제:이름 공간 구현하기

프로그래밍 언어에서 이름 공간은 서로 연계되어 수행된다. 예를들어, 다음 코드를 보자 세 개의 함수가 중첩되어 선언되어 있다. f3 함수의 이름 공간은 f2함수의 이름 공간과 연계되어 있다. 즉, f3에서 찾을수 없는 이름은 f2에서 찾으려고 시도한다. 만일 변수 a를 요구하면 f1이름공간까지 거슬러 올라갈 것이다.

```
>>> def f1():
...     x  = 1
...     a = 1
...     def f2():
...             x = 2
...             b = 2
...             def f3():
...                     x = 3
...                     c = 3
...                     print(x, a, b, c)       # 3 1 2 3
...             f3()                            # f3() 함수를 호출한다.
...     f2()                                    # f2() 함수를 호출한다.
...
>>> f1()                                        # f1() 함수를 호출한다.
3 1 2 3
```



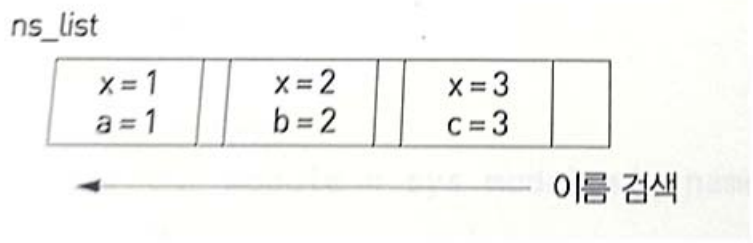
이와 유사한 동작을 하는 이름 공간을 사전을 이용하여 구현해 보자. 여러 이름 공간의 관계는 다양한 방법으로 구현할수 있지만 여기서는 스택으로 동작하는 리스트를 사용하여 구현해 보자. ns_list는 사전으로 된 이름 공간을 저장하는 리스트이다.

```
>>> ns_list = []
```

새로운 계층적인 이름 공간이 만들어질때마다 이름 공간이 리스트에 추가된다.

```
>>> ns_list.append({})# 이름 공간을 추가한다. (f1)가한다.
>>> ns_list[-1]['x']=1# 해당 이름 공간에 변수를 추가한다.
>>> ns_list[-1]['a'] = 1
>>>
>>> ns_list.append({})# 다시 새로운 이름 공간을 추가한다. (f2)
>>>
>>> ns_list[-1]['x']=2
>>> ns_list[-1]['b'] = 2
>>>
>>> ns_list.append({})# 이름 공간을 추가한다. (f3)
>>> ns_list[-1]['x'] = 3
>>> ns_list[-1]['c'] = 3
```

어떤 이름의 값을 연결된 이름 공간에서 찾는 함수는 간단히 만들어진다.




```
>>> def getValue(name):
...     for ns in reversed(ns_list):# 역순으로 이름 공간을 검색한다.
...             if name in ns:
...                     return ns[name]
...
>>> getValue('x')# f3의 x
3
>>> getValue('c')# f3의 c
3
>>> getValue('b')# f2의 b
2
>>> getValue('a')# f1의 a
1
```

8.7 순서를 유지하는 사전: OrderedDict 사전

만일 입력 순서를 기억해야 하는 사전이 필요하면 collections 모듈의 OrderedDict 사전을 사용할수 있다. 이 자료형은 사전과 동일하게 동작하지만, 데이터가 추가된 순서를 기억하며 for 문 등으로 반복할 때 입력한 순서대로 처리된다. popitem() 메서드는 맨 마지막 항목을 제거하며, orderedDict 사전만의 move_to_end(key) 메서드는 키 항목을 맨 뒤로 이동시킨다.

```
>>> from collections import OrderedDict
>>> d = OrderedDict()
>>> d['one']=1
>>> d['ten']=10
>>> d['two']=2
>>> d
OrderedDict([('one', 1), ('ten', 10), ('two', 2)])
>>> d.popitem()                          # 맨 마지막의 항목을 꺼내고 삭제한다.
('two', 2)
>>> d['two']=2                           # 다시 추가한다.
>>> d.move_to_end('ten')                 #  'ten '을 맨 마지막으로 이동시킨다.
>>> d
OrderedDict([('one', 1), ('two', 2), ('ten', 10)])
```