실전코딩

03. 파이썬 프로그래밍 리뷰
- 자료구조와 알고리즘 -

강원대학교 컴퓨터공학과 박치현

Outline

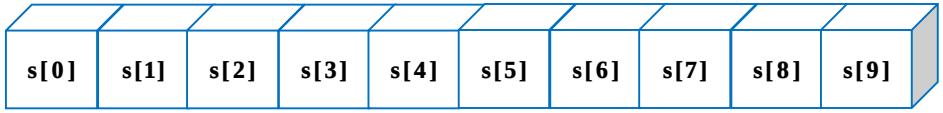
- 자료구조와 알고리즘
- 자료구조
- 리스트
- 튜플
- 딕셔너리
- 집합 **(set)**

자료구조

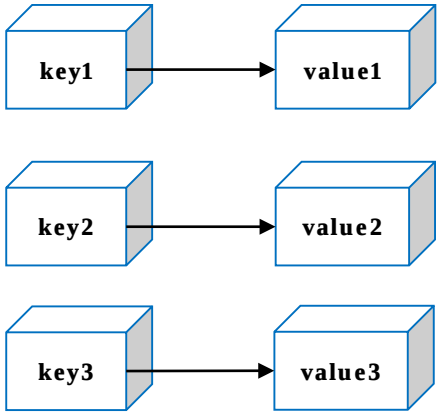
- **자료구조 (Data Structure)**

  데이터를 효율적으로 관리하는 방법을 정의

- List



- Dictionary



자료구조와 알고리즘

- **자료구조 (Data Structure)**

  데이터를 효율적으로 관리하는 방법을 정의

- **알고리즘 (Algorithm)**

  문제해결을 위해 정해진 절차, 방법 혹은 과정을 나타내는 것,

  계산 실행을 위한 단계적 절차

* 문제 해결 절차에 맞게 데이터를 효율적으로 관리

- 문제 해결을 위한 알고리즘 설계
- 각 절차에 따라 적절한 자료구조 활용
- 필요에 따라 새로운 형태의 자료구조 설계

자료구조

- **자료구조 (Data Structure)**

  데이터를 효율적으로 관리하는 방법을 정의

리터럴(literal)

- 값 자체를 의미
  - 숫자 (십진법, 2진법, 8진법, 16진법, 소수점과 e, 복소수 등)
  - 문자 (“” 혹은 ‘’ 안에 있는 문자열)
  - 논리값 (True, False)
  - 특수값 (None)
  - 컬렉션 **(list, tuple, dictionary, set)**

컬렉션은 여러가지 요소를 하나로 묶어 사용하는 data type임

컬렉션은 파이썬의 리터럴 중 하나이고, list, tuple, dictionary, set 같은 것이 있음

파이썬에는 배열이라는 참조타입은 없지만 컬렉션 데이터 타입으로 데이터를 다룰 수 있다.

리스트

- **리스트 (list)**

  자료들의 모임으로, 순서가 있는 수정가능한 객체의 집합

  - list 선언 방식

  리스트명 = [원소1, 원소2, 원소3, ...]

```
list1 = [1, 2, 3, 4, 5]
```

  - list 의 활용

```
l1 = []
l2 = [1, 2, 3]
l3 = ['T', 'E', 'S', 'T']
l4 = [1, 2, 'T', 'E']
l5 = [1, 2, ['T', 'E']]
l6 = [[1, 2,], ['T', 'E']]
```

리스트

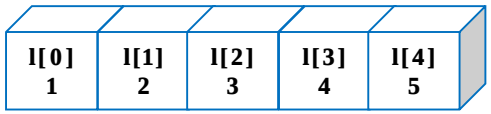
- **리스트 (list)**

  자료들의 모임으로, 순서가 있는 수정가능한 객체의 집합

  - list 선언 방식

  리스트명 = [원소1, 원소2, 원소3, ...]

```
l = [1, 2, 3, 4, 5]
```



index는 0부터 시작

index는 size-1까지

리스트

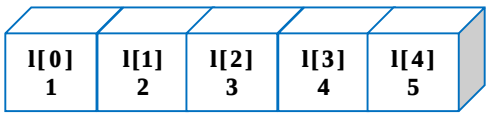
- **리스트 (list)**

  자료들의 모임으로, 순서가 있는 수정가능한 객체의 집합

  - list 값 접근 방식 : 인덱싱, 슬라이싱

  리스트명 = [원소1, 원소2, 원소3, ...]

```
l = [1, 2, 3, 4, 5]
```



***** 인덱싱 **:** 리스트 안의 특정한 값에 접근하는 기법

- **print(l[0]) #l[0]**의 값인 **1**이 출력
- **print(l[0] + l[1]) #1+2**의 결과 **3**이 출력
- **l[0] = 10 #l[0]** 값이 **10**으로 변경
- **print(l[-1]) #l**의 마지막 요소값 의미

리스트

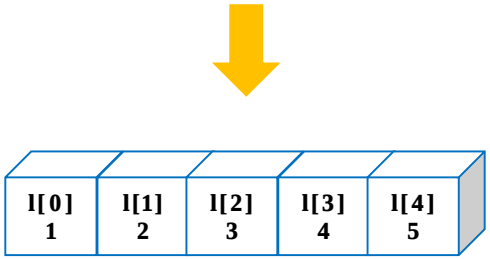
- **리스트 (list)**

  자료들의 모임으로, 순서가 있는 수정가능한 객체의 집합

- list 값 접근 방식 : 인덱싱, 슬라이싱

리스트명 = [원소1, 원소2, 원소3, ...]

```
l = [1, 2, 3, 4, 5]
```



***** 슬라이싱 : 리스트 안의 특정 범위의 값에 접근하는 기법

- **l[0:2] #index 0**부터 **2**미만의 **(0≤index<2)** 값을 가져옴, **[1, 2]**
- **l[:2] #index 0**부터 **2**미만의 **(0≤index<2)** 값을 가져옴, **[1, 2]**
- **l[2:] #index 2**부터 리스트 마지막 값까지 가져옴, **[3, 4, 5]**

리스트

- **리스트 (list)**

  자료들의 모임으로, 순서가 있는 수정가능한 객체의 집합

```
1   l = [1, 2, 3, 4, 5]
2
3   print("type(l): ", type(l))
4   print("len(l): ", len(l))
5
6   print("for loop 1, ")
7   for i in l:
8       print("v: ", i)
9
10  print("for loop 2, ")
11  for i in range(0, len(l)):
12      print("v: ", l[i])
```

실행결과

```
type(l):  <class 'list'>
len(l):  5
for loop 1,
v:  1
v:  2
v:  3
v:  4
v:  5
for loop 2,
v:  1
v:  2
v:  3
v:  4
v:  5
```

리스트

- **리스트 (list)**

  자료들의 모임으로, 순서가 있는 수정가능한 객체의 집합

```
1   l = [1, 2, 3, 4, 5]
2
3   #indexing
4   print("before l[0]: ", l[0])
5   print("l[0] + l[1]: ", l[0] + l[1])
6   print("l[-1]: ", l[-1])
7
8   l[0] = 10
9   print("after l[0]: ", l[0])
10
11  #slicing
12  print("l[0:2]", l[0:2])
13  print("l[:2]", l[:2])
14  print("l[2:]", l[2:])
```

실행결과

```
before l[0]:  1
l[0] + l[1]:  3
l[-1]:  5
after l[0]:  10
l[0:2] [10, 2]
l[:2] [10, 2]
l[2:] [3, 4, 5]
```

리스트

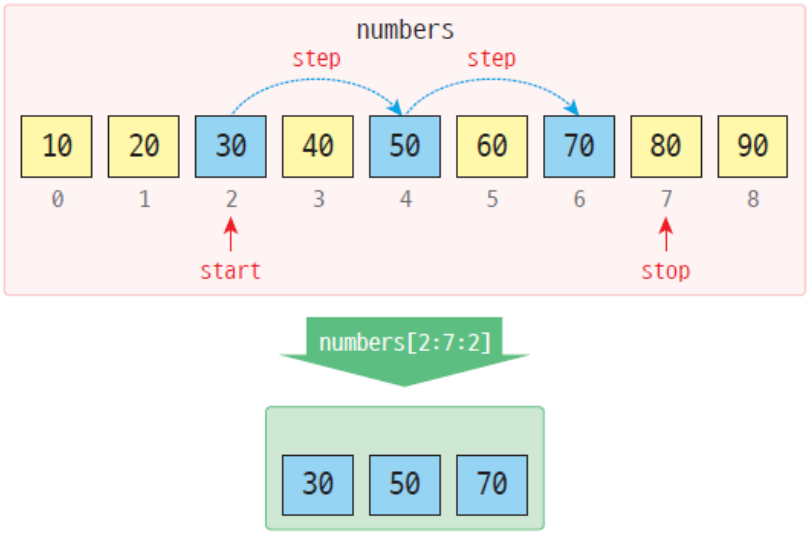
- **리스트 (list)**

  고급 슬라이싱

Syntax: 슬라이싱 #2

형식 리스트[start : stop : step]

예
```
numbers = [ 10, 20, 30, 40, 50, 60, 70, 80, 90 ]
sublist = numbers[2:7:2]
```



참고: 파이썬 익스프레스, 천인국 저

리스트

- **리스트 (list)**

  자료들의 모임으로, 순서가 있는 수정가능한 객체의 집합

```
1   l1 = []
2   l2 = [1, 2, 3]
3   l3 = ['T', 'E', 'S', 'T']
4   l4 = [1, 2, 'T', 'E']
5   l5 = [1, 2, ['T', 'E']]
6   l6 = [[1, 2], ['T', 'E']]
7
8   print("l5, ")
9   for i in l5:
10      print(i)
11
12  print("l6 print 1: ")
13  for i in l6:
14      print(i)
15
16  print("l6 print 2: ")
17  for i in l6:
18      print("i: ", i)
19      for j in i:
20          print("j: ", j)
```

실행결과

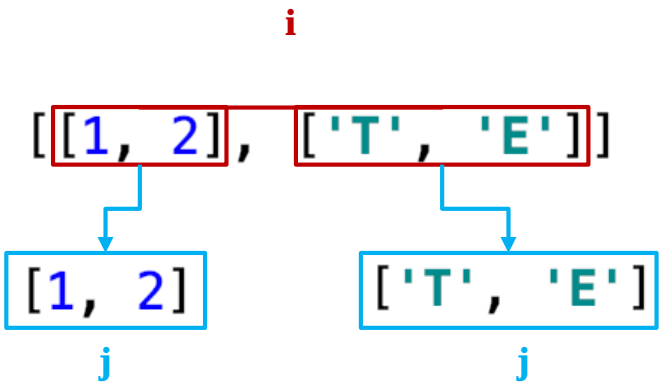
```
l5,
1
2
['T', 'E']
l6 print 1:
[1, 2]
['T', 'E']
l6 print 2:
i:  [1, 2]
j:  1
j:  2
i:  ['T', 'E']
j:  T
j:  E
```

리스트

- **리스트 (list)**

  자료들의 모임으로, 순서가 있는 수정가능한 객체의 집합

```
1   l1 = []
2   l2 = [1, 2, 3]
3   l3 = ['T', 'E', 'S', 'T']
4   l4 = [1, 2, 'T', 'E']
5   l5 = [1, 2, ['T', 'E']]
6   l6 = [[1, 2], ['T', 'E']]
7
8   print("l5, ")
9   for i in l5:
10      print(i)
11
12  print("l6 print 1: ")
13  for i in l6:
14      print(i)
15
16  print("l6 print 2: ")
17  for i in l6:
18      print("i: ", i)
19      for j in i:
20          print("j: ", j)
```



리스트

- **리스트 (list)**

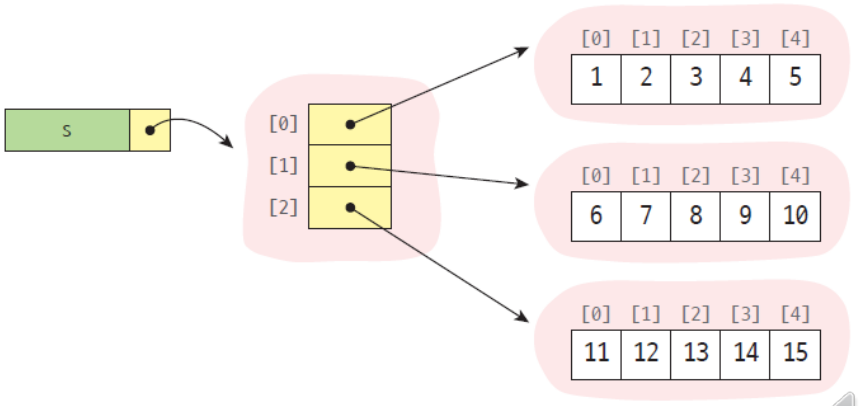
  2차원 리스트 == 2차원 테이블 ← 리스트의 리스트로 구현

```
s = [
    [1,2,3,4,5],
    [6,7,8,9,10],
    [11,12,13,14,15]
]

print(s)
```

출력:
[[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15]]


```
# 동적으로 리스트를 생성할 수 도 있음
rows = 3
cols = 5

s1 = [ ]
for row in range(rows):
    s1 += [[0]*cols] # 2차원 리스트끼리 합쳐진다.

s2 = [([0] * cols) for row in range(rows)]
print(s1)
print(s2)
```

출력: s1, s2 동일함
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]



참고: 파이썬 익스프레스, 천인국 저

리스트

- 리스트 연산

- list 더하기 (+)

```
1    l1 = [1, 2, 3]
2    l2 = ['T', 'E', 'S', 'T']
3
4    l3 = l1 + l2
5    print(l3)
```

실행결과

```
[1, 2, 3, 'T', 'E', 'S', 'T']
```

- list 반복 (*)

```
1    l1 = [1, 2, 3]
2
3    l2 = l1 * 3
4    print(l2)
```

실행결과

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

리스트

- 리스트 수정, 추가, 삭제

- list 수정

```
1    l1 = [1, 2, 3]
2    print("before l1: ", l1)
3
4    l1[0] = 10
5    print("after l1: ", l1)
```

실행결과

```
before l1:  [1, 2, 3]
after l1:  [10, 2, 3]
```

- list 추가

```
1    l1 = [1, 2, 3]
2    print("before l1: ", l1)
3
4    l1.append(5)
5    print("after l1: ", l1)
```

실행결과

```
before l1:  [1, 2, 3]
after l1:  [1, 2, 3, 5]
```

- list 삭제

```
1    l1 = [1, 2, 3]
2    print("before l1: ", l1)
3
4    del l1[0]
5    print("after l1: ", l1)
```

실행결과

```
before l1:  [1, 2, 3]
after l1:  [2, 3]
```

리스트

- 리스트 추가, 삭제

- List 추가

```
l1 = [1, 2, 3]
l1.insert(3, 4) #insert(index, value)
print(l1)

l1 = [1, 2, 3, 4]
l2 = [5, 6, 7]
l1.append(l2)
print(l1)

l1 = [1, 2, 3, 4]
l2 = [5, 6, 7]
l1.extend(l2)
print(l1)
```

[1, 2, 3, 4]
[1, 2, 3, 4, [5, 6, 7]]
[1, 2, 3, 4, 5, 6, 7]

- List 삭제

```
l1 = [1, 2, 3, 4, 5]
l1.pop(2) # pop 안에는 지우고 싶은 값의 index
print(l1)

l1 = [1, 2, 3, 4, 5]
l1.remove(2) # remove 안에는 지우고 싶은 값
print(l1)
```

실행결과

[1, 2, 4, 5]
[1, 3, 4, 5]

리스트

- 리스트 메소드(함수)

```
1   l1 = [1, 3, 2, 5, 4]
2   print("before sort: ", l1)
3
4   #sort()
5   l1.sort()
6   print("after sort: ", l1)
7
8   #reverse()
9   l1.reverse()
10  print("after reverse: ", l1)
11
12  #index(value)
13  print("l1.index(1): ", l1.index(1))
14  print("l1.index(5): ", l1.index(5))
15
16  #insert(pos, value)
17  l1.insert(0, 1)
18  print("after insert: ", l1)
19
20  #count(value)
21  print("l1.count(1): ", l1.count(1))
```

실행결과

```
before sort:  [1, 3, 2, 5, 4]
after sort:  [1, 2, 3, 4, 5]
after reverse:  [5, 4, 3, 2, 1]
l1.index(1):  4
l1.index(5):  0
after insert:  [1, 5, 4, 3, 2, 1]
l1.count(1):  2
```

리스트

- **zip 함수**

  - 여러 개의 순회 가능한 객체를 인자로 받고, 각 객체가 담고 있는 원소를 튜플(tuple)의 형태로 차례로 접근할 수 있는 반복자(iterator)를 반환

  예) 두 개의 list를 받아서 항목 두 개를 묶어서 제공함

```
l1 = [1, 2, 3, 4, 5]
l2 = ['one', 'two', 'three', 'four', 'five']

for pair in zip(l1, l2):
    print(pair)

for a, b in zip(l1, l2):
    print("number ", a, " is ", b)
```

결과

```
(1, 'one')
(2, 'two')
(3, 'three')
(4, 'four')
(5, 'five')
number 1 is one
number 2 is two
number 3 is three
number 4 is four
number 5 is five
```

리스트

- **zip 함수**

  - 여러 개의 순회 가능한 객체를 인자로 받고, 각 객체가 담고 있는 원소를 튜플(tuple)의 형태로 차례로 접근할 수 있는 반복자(iterator)를 반환

  예) 두 개의 list를 받아서 항목 두 개를 묶어서 제공함

```
l3 = ['하나', '둘', '셋', "넷", "다섯"]

for pair in zip(l1, l2, l3):
    print(pair)

for a, b, c in zip(l1, l2, l3):
    print("number ", a, " is ", b, " ", c)
```

결과

```
(1, 'one', '하나')
(2, 'two', '둘')
(3, 'three', '셋')
(4, 'four', '넷')
(5, 'five', '다섯')
number 1 is one 하나
number 2 is two 둘
number 3 is three 셋
number 4 is four 넷
number 5 is five 다섯
```

```
t4 = ('일', '이', '삼', "사", "오")
for pair in zip(l1, l2, t4):
    print(pair)
```

```
(1, 'one', '일')
(2, 'two', '이')
(3, 'three', '삼')
(4, 'four', '사')
(5, 'five', '오')
```

리스트

- 리스트 메소드 정리

메소드	설명
append()	요소를 리스트의 끝에 추가한다.
extend()	리스트의 모든 요소를 다른 리스트에 추가한다.
insert()	지정된 위치에 항목을 삽입한다.
remove()	리스트에서 항목을 삭제한다.
pop()	지정된 위치에서 요소를 삭제하여 반환한다.
clear()	리스트로부터 모든 항목을 삭제한다.
index()	일치되는 항목의 인덱스를 반환한다.
count()	인수로 전달된 항목의 개수를 반환한다.
sort()	오름차순으로 리스트 안의 항목을 정렬한다.
reverse()	리스트 안의 항목의 순서를 반대로 한다.
copy()	리스트의 복사본을 반환한다.

참고: 파이썬 익스프레스, 천인국 저

리스트

- 리스트에서 사용할 수 있는 내장 함수

함수	설명
round()	주어진 자리수대로 반올림한 값을 반환한다.
reduce()	특정한 함수를 리스트 안의 모든 요소에 적용하여 결과값을 저장하고 최종 합계값만을 반환한다.
sum()	리스트 안의 숫자들을 모두 더한다.
ord()	유니코드 문자의 코드값을 반환한다.
cmp()	첫 번째 리스트가 두 번째 보다 크면 1을 반환한다.
max()	리스트의 최대값을 반환한다.
min()	리스트의 최소값을 반환한다.
all()	리스트의 모든 요소가 참이면 참을 반환한다.
any()	리스트 안의 한 요소라도 참이면 참을 반환한다.
len()	리스트의 길이를 반환한다.
enumerate()	리스트의 요소들을 하나씩 반환하는 객체를 생성한다.
accumulate()	특정한 함수를 리스트의 요소에 적용한 결과를 저장하는 리스트를 반환한다.
filter()	리스트의 각 요소가 참인지 아닌지를 검사한다.
map()	특정한 함수를 리스트의 각 요소에 적용하고 결과를 담은 리스트를 반환한다.

참고: 파이썬 익스프레스, 천인국 저

```python
l1 = [1, 2, 3, 4, 5]
input = 6
if any(input == i for i in l1):
    print("6 exists in L1")
else:
    print("6 does not exist in L1")

input = 1
if any(input == i for i in l1):
    print("1 exists in L1")
else:
    print("1 does not exist in L1")

l1 = [2, 2, 2, 2, 2]
input = 2
if all(input == i for i in l1):
    print("2 is the only element in L1")
else:
    print("2 is not the only element in L1")
```

결과

```
6 does not exist in L1
1 exists in L1
2 is the only element in L1
```

튜플

- **튜플 (tuple)**

  순서가 있는 객체의 집합, 값을 변경할 수 없음
  (리스트와 유사하지만 값을 변경할 수 없는 것이 특징임)

  - tuple 선언 방식

  튜플명 = (원소1, 원소2, 원소3, ...)

```
t1 = (1, 2, 3, 4, 5)
```

  - tuple 활용

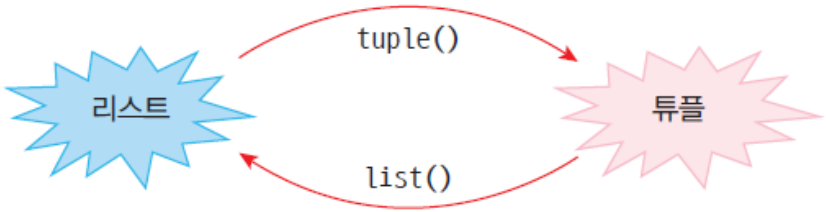
```
t1 = (1, 2, 3, 4, 5)
t2 = ('t', 'u', 'p', 'l', 'e')
t3 = 1, 2, 't', 'u'
t4 = (1, 2, ('t', 'u'))
t5 = ((1, 2), ['t', 'u'])
```

← **()없이도 선언 가능**

튜플

- **튜플 (tuple)**

  순서가 있는 객체의 집합, 값을 변경할 수 없음
  (리스트와 유사하지만 값을 변경할 수 없는것이 특징임)




```
L1 = [1,2,3,4]
t1 = tuple(l1)

print(type(l1))
print(l1)
print(type(t1))
print(t1)


t2 = (5,6,7,8)
l2 = list(t2)

print(type(l2))
print(l2)
print(type(t2))
print(t2)
```

결과

```
<class 'list'>
[1, 2, 3, 4]
<class 'tuple'>
(1, 2, 3, 4)

<class 'list'>
[5, 6, 7, 8]
<class 'tuple'>
(5, 6, 7, 8)
```

튜플

- **튜플 (tuple)**

  순서가 있는 객체의 집합, 값을 변경할 수 없음

```
1   t1 = (1, 2, 3, 4, 5)
2   t2 = 1, 2, 't', 'u'
3   t3 = ((1, 2), ['t', 'u'])
4
5   print("type(t1): ", type(t1),"| type(t2): ", type(t2),
6         "| type(t3): ", type(t3))
7
8   for i in t1:
9       print(i)
10
11  print("print tuple t3: ")
12
13  for i in t3:
14      print("type(i): ", type(i))
15      for j in i:
16          print("type(j): ", type(j), "| value: ", j)
```

실행결과

```
type(t1):  <class 'tuple'> | type(t2):  <class 'tuple'> | type(t3):  <class 'tuple'>
1
2
3
4
5
print tuple t3:
type(i):  <class 'tuple'>
type(j):  <class 'int'> | value:  1
type(j):  <class 'int'> | value:  2
type(i):  <class 'list'>
type(j):  <class 'str'> | value:  t
type(j):  <class 'str'> | value:  u
```

튜플

- **튜플 (tuple)**

  순서가 있는 객체의 집합, 값을 변경할 수 없음

  - list 와 마찬가지로 인덱싱, 슬라이싱을 통해 값 접근 가능 **(값의 변경을 불가능)**

```
1   t1 = (1, 2, 3, 4, 5)
2   t2 = 1, 2, 't', 'u'
3   t3 = ((1, 2), ['t', 'u'])
4
5   print("t1[2]:    ", t1[2])
6   print("t2[-1]:   ", t2[-1])
7   print("t3[1][0]: ", t3[1][0])
8
9   print("t1[0:2]", t1[0:2])
10  print("t1[2:]", t1[2:])
```

실행결과

```
t1[2]:     3
t2[-1]:    u
t3[1][0]:  t
t1[0:2] (1, 2)
t1[2:] (3, 4, 5)
```

튜플

- 튜플 연산

- tuple 더하기 **(+)** 와 반복 **(*)**

```
1    t1 = (1, 2, 3, 4, 5)
2    t2 = t1 + (7, 'A')
3
4    print("t2: ", t2)
5
6    t3 = 1, 2, 3
7    t4 = t3 * 3
8
9    print("t4: ", t4)
```

실행결과

```
t2:  (1, 2, 3, 4, 5, 7, 'A')
t4:  (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

튜플

- 튜플의 활용

```
1   # tuple의 선언
2   t1 = 1, 2, 3, 4, 5
3   print("type(t1): ", type(t1))
4
5   t2 = (8)
6   t3 = (8, )
7   print("type(t2): ", type(t2), "| type(t3): ", type(t3))
8
9   # tuple의 할당
10  (a, (b, (c, d, e))) = (1, (2, (3, 4, 5)))
11
12  print("a: ", a, "| b: ", b, "| c: ", c,
13        "| d: ", d, "| e: ", e)
14
15  a, b = b, a
16  print("a: ", a, "| b: ", b, "| c: ", c,
17        "| d: ", d, "| e: ", e)
18
19  # tuple 변환
20  l1 = [1, 2, 3, 4, 5]
21  s1 = "tuple_test"
22
23  print("type(l1): ", type(l1), "| type(s1): ", type(s1))
24  print("l1: ", l1)
25  print("s1: ", s1)
26
27  t4 = tuple(l1)
28  t5 = tuple(s1)
29
30  print("type(t4): ", type(t4), "| type(t5): ", type(t5))
31  print("l1: ", t4)
32  print("s1: ", t5)
```

실행결과

```
type(t1):  <class 'tuple'>
type(t2):  <class 'int'> | type(t3):  <class 'tuple'>
a:  1 | b:  2 | c:  3 | d:  4 | e:  5
a:  2 | b:  1 | c:  3 | d:  4 | e:  5
type(l1):  <class 'list'> | type(s1):  <class 'str'>
l1:  [1, 2, 3, 4, 5]
s1:  tuple_test
type(t4):  <class 'tuple'> | type(t5):  <class 'tuple'>
l1:  (1, 2, 3, 4, 5)
s1:  ('t', 'u', 'p', 'l', 'e', '_', 't', 'e', 's', 't')
```

튜플

- 튜플의 활용

```
1   # tuple의 선언
2   t1 = 1, 2, 3, 4, 5
3   print("type(t1): ", type(t1))
4
5   t2 = (8)
6   t3 = (8, )
7   print("type(t2): ", type(t2), "| type(t3): ", type(t3))
8
```

- **( )** 없이도 **tuple** 선언이 가능
- **( )** 가 있어도 값이 하나인 경우, **tuple**로 인지하지 않음
- **(,)** 를 활용하여 **tuple** 선언 가능

실행결과

```
type(t1):  <class 'tuple'>
type(t2):  <class 'int'> | type(t3):  <class 'tuple'>
```

튜플

- 튜플의 활용

```
# tuple의 선언
t1 = 1, 2, 3, 4, 5
print("type(t1): ", type(t1))

t2 = (8)
t3 = (8, )
print("type(t2): ", type(t2), "| type(t3): ", type(t3))

# tuple의 할당
(a, (b, (c, d, e))) = (1, (2, (3, 4, 5)))

print("a: ", a, "| b: ", b, "| c: ", c,
      "| d: ", d, "| e: ", e)

a, b = b, a
print("a: ", a, "| b: ", b, "| c: ", c,
      "| d: ", d, "| e: ", e)

# tuple 변환
l1 = [1, 2, 3, 4, 5]
s1 = 'tuple_test'

print("type(l1): ", type(l1), "| type(s1): ", type(s1))
print("l1: ", l1)
print("s1: ", s1)

t4 = tuple(l1)
t5 = tuple(s1)

print("type(t4): ", type(t4), "| type(t5): ", type(t5))
print("l1: ", t4)
print("s1: ", t5)
```

- **tuple** 내 **tuple** 구조를 활용하여 할당 가능
- **tuple** 을 활용하여 추가 변수 없이 **swap**이 가능

실행결과

```
type(t1):  <class 'tuple'>
type(t2):  <class 'int'> | type(t3):  <class 'tuple'>
a:  1 | b:  2 | c:  3 | d:  4 | e:  5
a:  2 | b:  1 | c:  3 | d:  4 | e:  5
type(l1):  <class 'list'> | type(s1):  <class 'str'>
l1:  [1, 2, 3, 4, 5]
s1:  tuple_test
type(t4):  <class 'tuple'> | type(t5):  <class 'tuple'>
l1:  (1, 2, 3, 4, 5)
s1:  ('t', 'u', 'p', 'l', 'e', '_', 't', 'e', 's', 't')
```

튜플

- 튜플의 활용

```
# tuple의 선언
t1 = 1, 2, 3, 4, 5
print("type(t1): ", type(t1))

t2 = (8)
t3 = (8, )
print("type(t2): ", type(t2), "| type(t3): ", type(t3))

# tuple의 할당
(a, (b, (c, d, e))) = (1, (2, (3, 4, 5)))

print("a: ", a, "| b: ", b, "| c: ", c,
      "| d: ", d, "| e: ", e)

a, b = b, a
print("a: ", a, "| b: ", b, "| c: ", c,
      "| d: ", d, "| e: ", e)
```

```
# tuple 변환
l1 = [1, 2, 3, 4, 5]
s1 = "tuple_test"

print("type(l1): ", type(l1), "| type(s1): ", type(s1))
print("l1: ", l1)
print("s1: ", s1)

t4 = tuple(l1)
t5 = tuple(s1)

print("type(t4): ", type(t4), "| type(t5): ", type(t5))
print("l1: ", t4)
print("s1: ", t5)
```

실행결과

```
type(t1):  <class 'tuple'>
type(t2):  <class 'int'> | type(t3):  <class 'tuple'>
a:  1 | b:  2 | c:  3 | d:  4 | e:  5
a:  2 | b:  1 | c:  3 | d:  4 | e:  5
type(l1):  <class 'list'> | type(s1):  <class 'str'>
l1:  [1, 2, 3, 4, 5]
s1:  tuple_test
type(t4):  <class 'tuple'> | type(t5):  <class 'tuple'>
l1:  (1, 2, 3, 4, 5)
s1:  ('t', 'u', 'p', 'l', 'e', '_', 't', 'e', 's', 't')
```

- **tuple()**을 이용하여 변환

튜플

- 튜플과 함수

  - **tuple**을 이용하여 함수에서 여러 값의 **return**이 가능

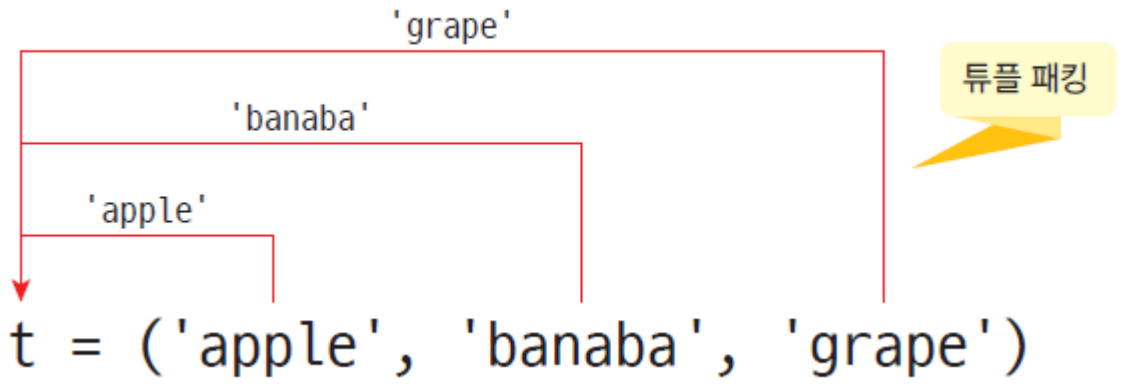
```
1   def sum_mul(t1):
2       sum = 0
3       mul = 1
4       for i in t1:
5           sum += i
6           mul *= i
7
8       return sum, mul
9
10  print("call sum_mul, ")
11  t1 = (1, 2, 3, 4, 5)
12  print("sum_mul(t1): ", sum_mul(t1))
13  print("type(sum_mul(t1)): ", type(sum_mul(t1)))
14
15  a, b = sum_mul(t1)
16  print("a: ", a, "| b: ", b)
17  print("type(a): ", type(a), "| type(b): ", type(b))
```

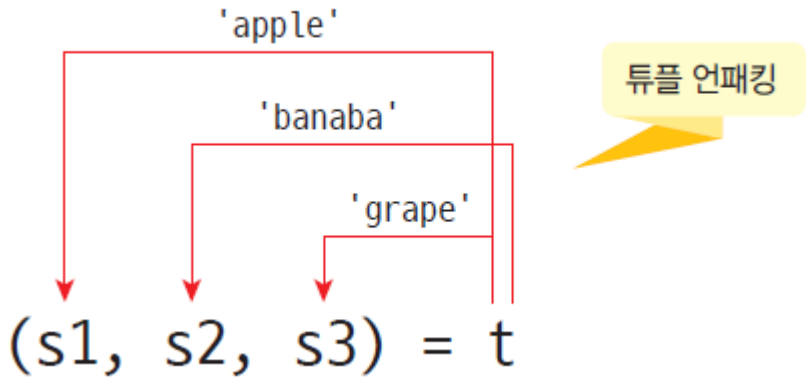
- **tuple** 을 이용한 여러 값 **return**
- **return** 값을 각각의 변수에 할당 가능

실행결과

```
call sum_mul,
sum_mul(t1):  (15, 120)
type(sum_mul(t1)):  <class 'tuple'>
a:  15 | b:  120
type(a):  <class 'int'> | type(b):  <class 'int'>
```

튜플





참고: 파이썬 익스프레스, 천인국 저

튜플

- 튜플과 리스트에서 **enumerate()** 함수를 사용할 수 있음
  - enumerate() 함수는 순서와 리스트(튜플)의 값을 전달하는 기능

```
t1 = ('a','b','c','d')
for index, value in enumerate(t1):
    print(index, value)

print()

l1 = ['a','b','c','d']
for index, value in enumerate(t1):
    print(index, value)
```

결과

```
0 a
1 b
2 c
3 d

0 a
1 b
2 c
3 d
```

딕셔너리

- 딕셔너리 **(dictionary)**

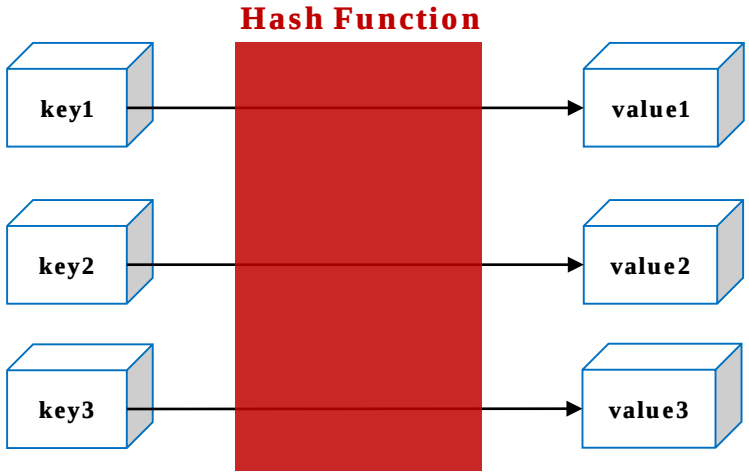
  key와 **value**로 맵핑되어 있는 순서가 없는 집합

  - dictionary 선언 방식

  딕셔너리명 = **{'key1' : 'value1' , 'key2' : 'value2' , ...}**

```
d1 = {"a" : 1, "b" : 2, "c" : 3}
```

  - dictionary 구조



딕셔너리

- 딕셔너리 **(dictionary)**

  key와 **value**로 맵핑되어 있는 순서가 없는 집합

  - dictionary 선언 방식

  딕셔너리명 = **{‘key1’ : ‘value1’ , ‘key2’ : ‘value2’ , ...}**

```
d1 = {"a" : 1, "b" : 2, "c" : 3}
```

  - 딕셔너리명**[key]** 를 통해 **value**에 접근가능
  - **d1[“a”]**을 통해 **1** 접근가능
  - **d1[“b”]**을 통해 **2** 접근가능
  - **d1[“c”]**을 통해 **3** 접근가능

딕셔너리

- 딕셔너리 **(dictionary)**

  key와 **value**로 맵핑되어 있는 순서가 없는 집합

```
1   d1 = {"a" : 1, "b" : 2, "c" : 3}
2
3   print("d1: ", d1)
4   print("type(d1): ", type(d1))
5
6   # 데이터 접근
7   print("d1[\"a\"]: ", d1["a"], "| d1[\"b\"]: ", d1["b"])
8
9   print("d1.keys(): ", d1.keys())
10  for key in d1.keys():
11      print("d1[", key, "]: ", d1[key])
12
13  d1["a"] = 10
14
15  print("d1: ", d1)
```

실행결과

```
d1:  {'a': 1, 'b': 2, 'c': 3}
type(d1):  <class 'dict'>
d1["a"]:  1 | d1["b"]:  2
d1.keys():  dict_keys(['a', 'b', 'c'])
d1[ a ]:  1
d1[ b ]:  2
d1[ c ]:  3
d1:  {'a': 10, 'b': 2, 'c': 3}
```

- 순서가 없기 때문에 **index** 가 아닌 **key**로 접근 가능
- 딕셔너리명**.keys()**를 통해 **key** 접근 가능

딕셔너리

- 딕셔너리 **(dictionary)**

key와 **value**로 맵핑되어 있는 순서가 없는 집합

```
1   d1 = {"a" : 1, "b" : 2, "c" : 3, "a" : 5, "b" : 8}
2
3   # key가 중복되는 경우
4   print("d1: ", d1)
5
6   # 데이터 삽입, 삭제
7   d1["d"] = 10
8   d1["e"] = 20
9   print("추가 후 d1: ", d1)
10
11  del d1["d"]
12  del d1["c"]
13  print("삭제 후 d1: ", d1)
14  print()
15
16  # dict 변환
17  l1 = [['a', 1], ['b', 2], ['c', 3]]
18  print("type(l1): " , type(l1))
19  d2 = dict(l1)
20  print("type(d2): " , type(d2))
21  print(d2)
22  print()
23  t1 = (('a', 1), ('b', 2), ('c', 3))
24  print("type(t1): " , type(t1))
25  d3 = dict(t1)
26  print("type(d3): " , type(d3))
27  print(d3)
```

- **key**가 중복되는 경우, 마지막 **key**의 값을 활용
- **list** 내 **list, tuple** 내 **tuple** 구조를 활용하여
- **dictionary**로 변환 가능 **(dict( )**함수 사용**)**

실행결과

```
d1:  {'a': 5, 'b': 8, 'c': 3}
추가 후 d1:  {'a': 5, 'b': 8, 'c': 3, 'd': 10, 'e': 20}
삭제 후 d1:  {'a': 5, 'b': 8, 'e': 20}

type(l1):  <class 'list'>
type(d2):  <class 'dict'>
{'a': 1, 'b': 2, 'c': 3}

type(t1):  <class 'tuple'>
type(d3):  <class 'dict'>
{'a': 1, 'b': 2, 'c': 3}
```

딕셔너리

- 딕셔너리 **(dictionary)**

  key와 **value**로 맵핑되어 있는 순서가 없는 집합

```
1   d1 = {"a" : 1, "b" : 2, "c" : 3}
2
3   # key의 자료형
4   s1 = "this is string"
5   l1 = [1, 2, 3, 4, 5]
6   t1 = (1, 2, 3)
7   i1 = 8
8
9   d2 = {s1:1, l1:2, t1:3, i1:4}
10  print("d2: ", d2)
```

- **list, set, dictionary**는 **key**로 사용할 수 없음

```
TypeError: unhashable type: 'list'
```

```
1   d1 = {"a" : 1, "b" : 2, "c" : 3}
2
3   # key의 자료형
4   s1 = "this is string"
5   l1 = [1, 2, 3, 4, 5]
6   t1 = (1, 2, 3)
7   i1 = 8
8
9   d2 = {s1:1, t1:3, i1:4}
10  print("d2: ", d2)
```

- **String, tuple, integer** 모두 **key**로 사용 가능

Tuple (순서가 있는 객체의 집합, 값을 변경할 수 없음)의 경우 불변 객체의 참조만 담고 있는 경우는 **key**로 사용 가능
예를 들어 **t1=([1], 2, 3)**인 경우 **0**번째 **element [1]**는 **list**이기 때문에 값의 추가가 가능하고(불변 객체 아님), 이럴 경우 **key**로 사용 불가

실행결과

```
d2:  {'this is string': 1, (1, 2, 3): 3, 8: 4}
```

딕셔너리

- 딕셔너리 **(dictionary)**

  key와 **value**로 맵핑되어 있는 순서가 없는 집합

```
1   d1 = {"a" : [10, 20, 30], "b" : 2, "c" : 3, "D":{"A":65}, "e":(100, 200)}
2
3   print("d1: ", d1)
4   print()
5
6   # for loop
7   for i in d1:
8       print("i:    ", i)
9   print()
10
11  print("type(d1.keys())", type(d1.keys()))
12  for key in d1.keys():
13      print("key: ", key)
14  print()
15
16  print("type(d1.values())", type(d1.values()))
17  for value in d1.values():
18      print("value: ", value)
19  print()
20
21  # update()
22  print("before update(): ", d1)
23  d1.update({"a":1, "b":2, "c":3, "D":4, "e":5, "d":6})
24  print("after update():  ", d1)
```

- 딕셔너리명**.keys( ): key**값들을 **return**
- 딕셔너리명**.values( ): value**값들을 **return**
- 여러 값 수정을 위한 **update( )**

실행결과

```
d1:  {'a': [10, 20, 30], 'b': 2, 'c': 3, 'D': {'A': 65}, 'e': (100, 200)}

i:    a
i:    b
i:    c
i:    D
i:    e

type(d1.keys()) <class 'dict_keys'>
key:  a
key:  b
key:  c
key:  D
key:  e

type(d1.values()) <class 'dict_values'>
value:  [10, 20, 30]
value:  2
value:  3
value:  {'A': 65}
value:  (100, 200)

before update():  {'a': [10, 20, 30], 'b': 2, 'c': 3, 'D': {'A': 65}, 'e': (100, 200)}
after update():   {'a': 1, 'b': 2, 'c': 3, 'D': 4, 'e': 5, 'd': 6}
```

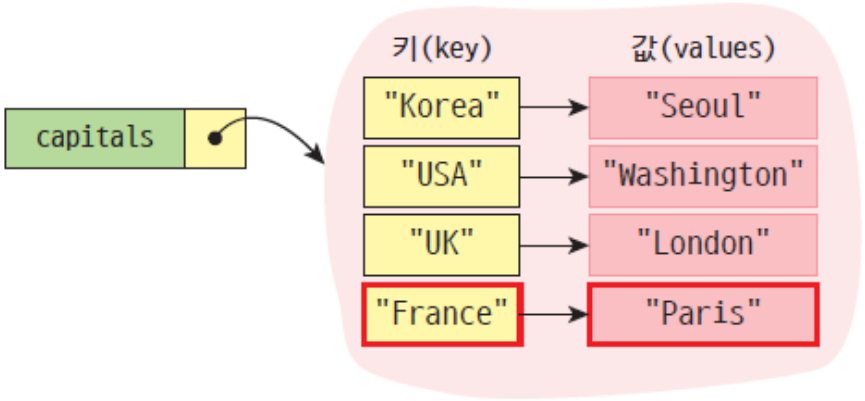
딕셔너리

- 딕셔너리 **(dictionary)**

  항목 추가 시 **[]** 연산자 이용 가능

```
capitals ={}
capitals["Korea"]="Seoul"
capitals["USA"]="Washington"
capitals["UK"]="London"
capitals["France"]="Paris"
print(capitals)
```

```
{'Korea': 'Seoul', 'USA': 'Washington', 'UK': 'London', 'France': 'Paris'}
```



참고: 파이썬 익스프레스, 천인국 저

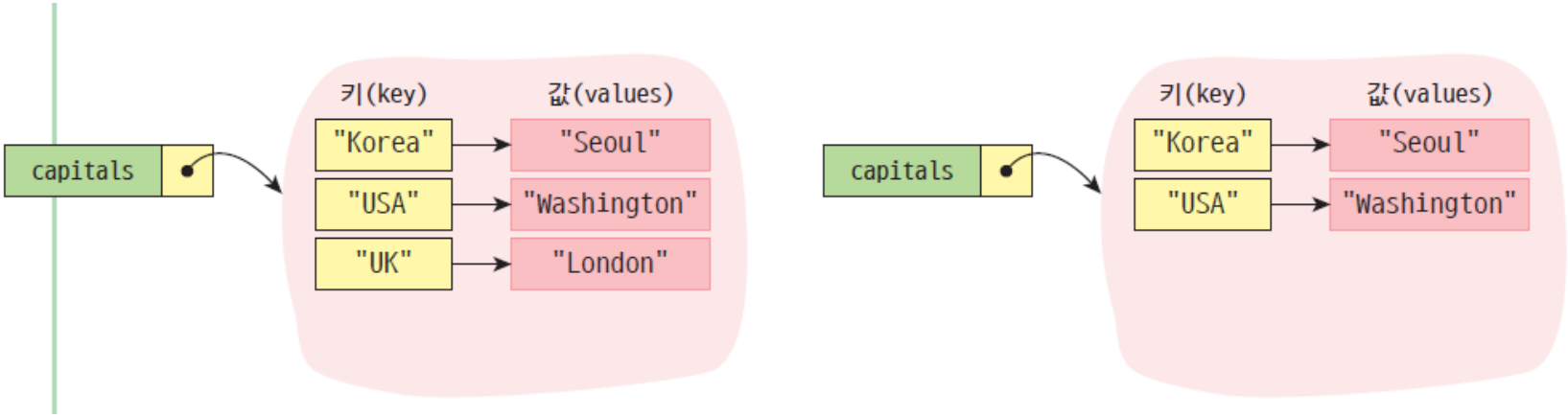
딕셔너리

- 딕셔너리 **(dictionary)**

  항목 삭제 시 **pop()** 사용

```
city = capitals.pop("UK")
```

만약 주어진 키를 가진 항목이 없으면
KeyError 예외가 발생한다




```
if "UK" in capitals :
        capitals.pop("UK")
```

딕셔너리

- 딕셔너리 **(dictionary)**

  항목 방문 **(retrieve)**

```
capitals ={"Korea":"Seoul","USA":"Washington","UK":"London"}
for key in capitals :
    print( key,":", capitals[key])
```

```
Korea : Seoul
USA : Washington
UK : London
```

```
capitals ={"Korea":"Seoul","USA":"Washington","UK":"London"}
for key, value in capitals.items():
        print( key,":", value )
```

```
Korea : Seoul
USA : Washington
UK : London
```

참고: 파이썬 익스프레스, 천인국 저

딕셔너리

- 딕셔너리 **(dictionary)**

  항목 방문 **(retrieve)**

```
capitals ={"Korea":"Seoul","USA":"Washington","UK":"London"}
print( capitals.keys())
print( capitals.values())
```

```
dict_keys(['Korea', 'USA', 'UK'])
dict_values(['Seoul', 'Washington', 'London'])
```

```
for key in sorted( capitals.keys()):
        print(key, end=" ")
```

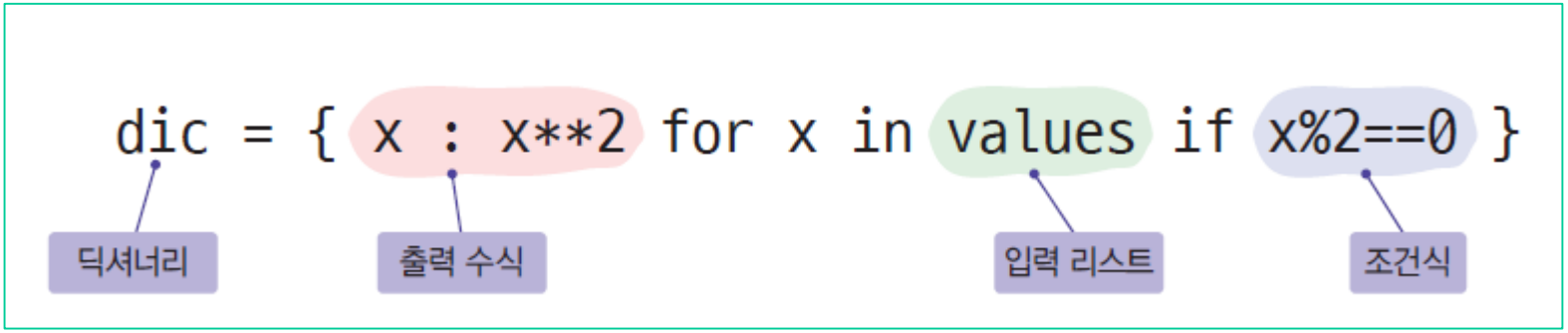
```
Korea UK USA
```

참고: 파이썬 익스프레스, 천인국 저

딕셔너리

- 딕셔너리 **(dictionary)**

  딕셔너리 함축




```
values =[1,2,3,4,5,6]

dic ={ x : x**2 for x in values if x%2==0 }
print(dic)
```

```
{2: 4, 4: 16, 6: 36}
```

참고: 파이썬 익스프레스, 천인국 저

딕셔너리

- 딕셔너리 **(dictionary)**

  딕셔너리 함축

```
dic ={ i:str(i) for i in [1,2,3,4,5]}
print( dic )
```

```
{1: "1", 2: "2", 3: "3", 4: "4", 5: "5"}
```

참고: 파이썬 익스프레스, 천인국 저

딕셔너리

- 딕셔너리 **(dictionary)** 메소드

연산	설명
`d = dict()`	공백 딕셔너리를 생성한다.
$d=\{k_1:v_1,k_2:v_2, \ldots,k_n:v_n\}$	초기값으로 딕셔너리를 생성한다.
`len(d)`	딕셔너리에 저장된 항목의 개수를 반환한다.
`k in d`	k가 딕셔너리 d 안에 있는지 여부를 반환한다.
`k not in d`	k가 딕셔너리 d 안에 없으면 True를 반환한다.
`d[key] = value`	딕셔너리에 키와 값을 저장한다.
`v = d[key]`	딕셔너리에서 key에 해당되는 값을 반환한다.
`d.get(key, default)`	주어진 키를 가지고 값을 찾는다. 만약 없으면 default 값이 반환된다.
`d.pop(key)`	항목을 삭제한다.
`d.values()`	딕셔너리 안의 모든 값의 시퀀스를 반환한다.
`d.keys()`	딕셔너리 안의 모든 키의 시퀀스를 반환한다.
`d.items()`	딕셔너리 안의 모든 (키, 값)을 반환한다.

참고: 파이썬 익스프레스, 천인국 저

집합 (set)

- **집합 (set)**

  순서가 없고 **unique** 값을 갖는 집합 (수학에서의 집합과 유사)

  - set 선언 방식

  set명 = {value1, value2, value3, ...}

  set명 = set( )

```
s1 = {1, 2, 3, 4, 5}
s2 = set()
```

  - set: 중복값 제거

```
s3 = {1, 1, 1, 2, 3, 4, 5}
```

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

집합 (set)

- **집합 (set)**

  순서가 없고 **unique** 값을 갖는 집합 (수학에서의 집합과 유사)

  - set 원소 추가 : add(), update()

  set명.add(원소)

  set명.update([원소1, 원소2, 원소3, ...])

  `s1 = {1, 2, 3, 4, 5}` → **s1.add(8)** → `{1, 2, 3, 4, 5, 8}`

  `s1 = {1, 2, 3, 4, 5}` → **s1.update([10, 20, 30])** → `{1, 2, 3, 4, 5, 10, 20, 30}`

  - set 원소 제거 : remove(), discard()

  `s1 = {1, 2, 3, 4, 5}` → **s1.remove(5)** → `{1, 2, 3, 4}`

  `s1 = {1, 2, 3, 4, 5}` → **s1.discard(5)** → `{1, 2, 3, 4}`

  `s1 = {1, 2, 3, 4, 5}` → **s1.remove(1000)** → `KeyError: 1000`

  `s1 = {1, 2, 3, 4, 5}` → **s1.discard(1000)** → 정상 실행

집합 (set)

- 집합 (set)

```python
s1 = {1, 2, 3, 4, 5}
s2 = set()

print("type(s1): ", type(s1))
print("type(s2): ", type(s2))

# set, 중복제거
s3 = {1, 1, 1, 2, 3, 4, 5}
print("s3: ", s3)
print()

# in 활용
for i in s3:
    print(i)

print("2 in s3: ", 2 in s3)

# add(), 원소 추가
s1.add(10)
s2.add(1)
print("after add() s1: ", s1)
print("after add() s2: ", s2)

# update(), 여러 값 추가
s2.update([20, 30, 40, 50])
print("after update() s2: ", s2)

# remove(), 원소 제거
s1.remove(10)
s2.remove(20)
print("after remove() s1: ", s1)
print("after remove() s2: ",s2)

# discard(), 원소 제거, 없는 경우 오류발생 없음
s1.discard(1000)
print("after update() s1: ", s1)
```

```python
s1 = {2,3,1,5,4}
print(type(s1))
print(s1)

출력결과
<class 'set'>
{1, 2, 3, 4, 5}
```

실행결과

```
type(s1):  <class 'set'>
type(s2):  <class 'set'>
s3:  {1, 2, 3, 4, 5}

1
2
3
4
5
2 in s3:  True
after add() s1:  {1, 2, 3, 4, 5, 10}
after add() s2:  {1}
after update() s2:  {1, 40, 50, 20, 30}
after remove() s1:  {1, 2, 3, 4, 5}
after remove() s2:  {1, 40, 50, 30}
after update() s1:  {1, 2, 3, 4, 5}
```

집합 (set)

- 집합 (set)
  - set 연산자, 메소드

```
s1 = {1, 2, 3, 4, 5}        s3 = {1, 2, 3}
s2 = {3, 4, 5, 6, 7}        s4 = {4, 5, 6}
```

연산자	의미	적용 예	결과
|	합집합	s1 | s2	{1, 2, 3, 4, 5, 6, 7}
&	교집합	s1 & s2	{3, 4, 5}
-	차집합	s1 – s2	{1, 2}
^	합집합-교집합 (대칭차)	s1 ^ s2	{1, 2, 6, 7}

메소드	의미	적용 예	결과
union	합집합	s1.union(s2)	{1, 2, 3, 4, 5, 6, 7}
intersection	교집합	s1.intersection(s2)	{3, 4, 5}
difference	차집합	s1.difference(s2)	{1, 2}
symmetric_difference	합집합-교집합	s1.symmetric_difference(s2)	{1, 2, 6, 7}
issubset	subset 확인	s3.issubset(s1)	True
issuperset	superset 확인	s1.issuperset(s3)	True
isdisjoint	공통 원소를 갖지 않는지 확인	s3.isdisjoint(s4)	True

집합 (set)

- 집합 (set)

```
1   s1 = {1, 2, 3, 4, 5}
2   s2 = {3, 4, 5, 6, 7}
3   s3 = {1, 2, 3}
4   s4 = {4, 5, 6}
5
6   print("s1 | s2: ", s1 | s2)
7   print("s1 & s2: ", s1 & s2)
8   print("s1 - s2: ", s1 - s2)
9   print("s1 ^ s2: ", s1 ^ s2)
10  print()
11  print("before s1: ", s1)
12  s1 |= s2
13  print("after s1: ", s1)
14  print()
15  s1 = {1, 2, 3, 4, 5}
16  print("s1.union(s2): ", s1.union(s2))
17  print("s1.intersection(s2): ", s1.intersection(s2))
18  print("s1.difference(s2): ", s1.difference(s2))
19  print("s1.symmetric_difference(s2): ",
20        s1.symmetric_difference(s2))
21  print()
22  print("s1.issubset(s3): ", s1.issubset(s3))
23  print("s3.issubset(s1): ", s3.issubset(s1))
24  print("s1.issuperset(s3): ", s1.issuperset(s3))
25  print("s3.issuperset(s1): ", s3.issuperset(s1))
26  print()
27  print("s1.isdisjoint(s3): ", s1.isdisjoint(s3))
28  print("s3.isdisjoint(s4): ", s3.isdisjoint(s4))
```

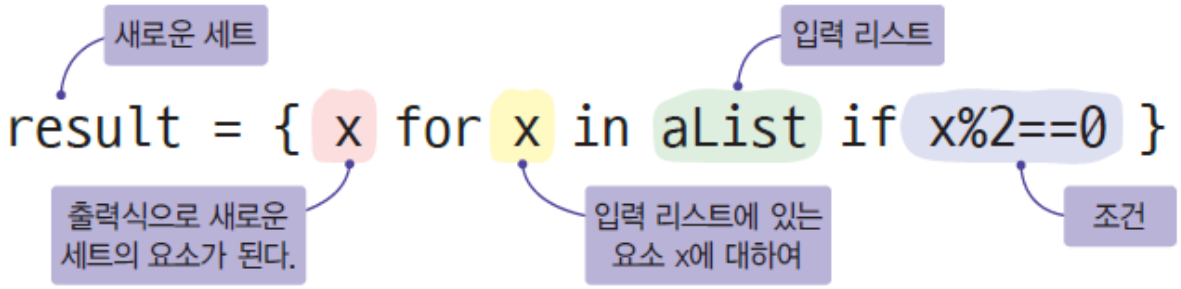
실행결과

```
s1 | s2:  {1, 2, 3, 4, 5, 6, 7}
s1 & s2:  {3, 4, 5}
s1 - s2:  {1, 2}
s1 ^ s2:  {1, 2, 6, 7}

before s1:  {1, 2, 3, 4, 5}
after s1:  {1, 2, 3, 4, 5, 6, 7}

s1.union(s2):  {1, 2, 3, 4, 5, 6, 7}
s1.intersection(s2):  {3, 4, 5}
s1.difference(s2):  {1, 2}
s1.symmetric_difference(s2):  {1, 2, 6, 7}

s1.issubset(s3):  False
s3.issubset(s1):  True
s1.issuperset(s3):  True
s3.issuperset(s1):  False

s1.isdisjoint(s3):  False
s3.isdisjoint(s4):  True
```

집합 (set)

- **집합 (set)**

  함축 연산 가능




```
aList  =[1,2,3,4,5,1,2 ]
result ={ x for x in aList if x%2==0 }
print(result)
```

실행결과

```
{2, 4}
```

참고: 파이썬 익스프레스, 천인국 저

집합 (set)

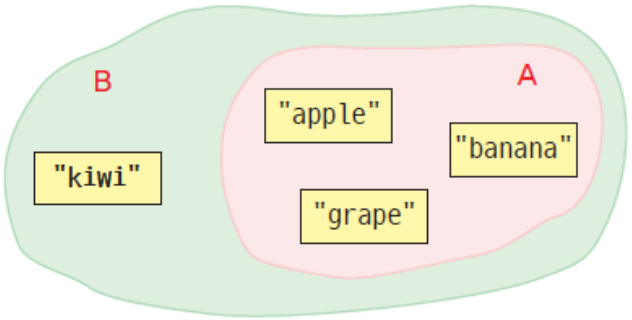
- **집합 (set)**

  부분 집합 연산

```
A ={"apple","banana","grape"}
B ={"apple","banana","grape","kiwi"}

if A < B :                    # 또는 A.issubset(B) :
   print("A는 B의 부분 집합입니다.")
```

실행결과

```
A는 B의 부분 집합입니다.
```



참고: 파이썬 익스프레스, 천인국 저

집합 (set)

- 집합 (set)

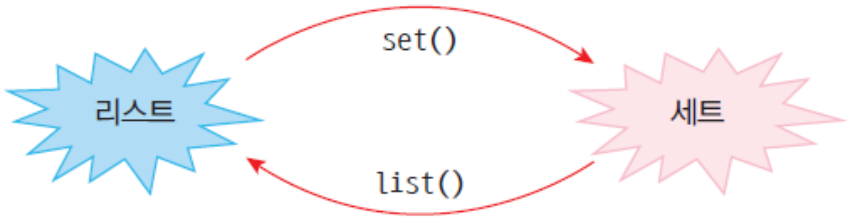
  ==, != 연산

```
A ={"apple","banana","grape"}
B ={"apple","banana","grape","kiwi"}

if A == B :
        print("A와 B는 같습니다.")
else :
        print("A와 B는 같지 않습니다.")
```

```
A와 B는 같지 않습니다.
```


집합 (set)

- 집합 (set) ← → 리스트(list)




```
>>> list1 =[1,2,3,4,5,1,2,4 ]
>>> len(set(list1))
5
```

서로 다른 정수는 몇 개나 있을까?

```
>>> list1 =[1,2,3,4,5 ]
>>> list2 =[3,4,5,6,7 ]
>>> set(list1)&set(list2)
{3, 4, 5}
```

공통적인 정수는무엇일까?

참고: 파이썬 익스프레스, 천인국 저

퀴즈 (o/x)

1. 튜플은 리스트와 유사하고, 생성 후 값을 변경할 수 있다.

2. Set는 중복된 요소(원소)를 허용한다.