튜플 (Tuple)

- 시퀀스(sequence)는 순서가 있는 자료 구조를 말함
- 파이썬에서 시퀀스 형에 속하는 자료구조는 string, list, tuple, range 등이 있으며, 이들 자료형에는 동일한 연산을 지원함

  : 인덱싱(indexing), 슬라이싱(slicing), +, *, in, not in 연산자
- 튜플은 리스트와 유사한 자료형이지만, 튜플의 원소는 추가/삭제/수정이 불가능 함

  : 원소를 변경하면 에러 발생
- 값이 변하지 않는 iterable한 데이터 저장에 효율적인 자료구조
  - 튜플을 사용하여 데이터를 실수로 추가, 삭제, 변경하는 것을 막을 수 있음
  - 프로그램 수행 동안 변경하지 않는 데이터는 튜플로 저장하는 것이 효율적

튜플 (Tuple)

- 리스트처럼 여러 데이터형을 저장할 수 있는 자료형
- 형식(syntax)

  튜플명 = (원소1, 원소2, 원소3, ...)

  - ( ) 소괄호를 사용하여 표시, 괄호는 생략 가능
  - 콤마를 사용하면 괄호가 없어도 튜플로 인식
  - 원소로 정수, 문자열, 튜플, 리스트 등 가능
- 튜플의 원소는 추가/삭제/수정할 수 없음, **immutable 자료형**
- 리스트 자료형에서 언급한 indexing, slicing, operator, 내장 함수 등을 사용할 수 있음
  - 다만, slicing으로 원소를 수정할 수는 없음
  - del 명령어로 튜플 자체 삭제만 가능 (원소 삭제에는 사용 못함)

튜플 (Tuple)

- 빈 튜플 생성

```
t = ()
t = tuple()
```

- 튜플 생성
  - 콤마를 사용하면 괄호가 없어도 튜플로 인식
  - 원소가 하나인 튜플을 만들 때도 반드시 콤마가 필요
  - 여러 가지 데이터를 튜플로 묶는 것을 튜플 packing이라 함
  - 반대로, 튜플의 각 원소를 여러 개의 변수에 할당하는 것을 튜플 unpacking이라 함 (튜플 원소의 수와 변수의 수가 일치해야 함)
    : 튜플의 값들을 여러 변수에 동시 할당하는 것이 가능

튜플 (Tuple)

```
>>> T = (1,)                    # 원소가 한 개인 튜플
>>> type(T)
<class 'tuple'>
>>> S = (1)                     # S = 1 과 같은 명령어
>>> type(S)
<class 'int'>
>>> C = 10,                     # 원소가 한 개인 튜플
>>> type(C)
<class 'tuple'>
>>> D = 1,2,3                   # 원소가 세 개인 튜플, D = (1, 2, 3)
>>> type(D)
<class 'tuple'>
>>> student = ("철수", 19, "CS“)          # packing
>>> student  ('
철수', 19, 'CS')
>>> (name, age, major) = student          # unpacking
>>> name
'철수‘
>>>
```

튜플 (Tuple)

```
>>> t1 = tuple("ab")      # t1 = ('a','b')
>>> t2 = ("ab",)          # 단일 원소 튜플 (콤마 필요)
>>> t3 = ("ab")           # t3 = "ab"  (문자열 할당. 튜플 아님!!)
>>> print(t1, t2, t3)
('a', 'b') ('ab',) ab
>>> t = t1 + (5,6)        # 새로운 튜플 t 생성
>>> print(t)
('a', 'b', 5, 6)
>>> del t                 # 튜플 자체는 삭제 가능
>>>
```

```
del t1[1];   t1[2] = 9;    # 오류 (원소 삭제/수정 불가)
```

튜플 (Tuple)

- 튜플을 이용한 변수들의 값 교환(swap)

```
>>> x=10; y=20
>>> print(x,y)
10 20
>>> x,y = y,x     # (x,y) = (y,x) 와 동일
>>> print(x,y)
20 10
>>>
```

- 문자열, 리스트와 동일하게 인덱싱, 슬라이싱 사용
  : 다만, 데이터 변경은 불가능
- + 연결 연산, * 반복연산, in 연산, len() 함수 사용가능

튜플 (Tuple)

- 튜플 객체의 메소드

```
>>> dir(tuple)
['__add__', '__class__', ……'__subclasshook_', 'count','index']
```

Method	Description (T : 튜플, x : 임의의 object)
T.index(x)	튜플에서 데이터 x의 인덱스를 반환
T.count(x)	튜플에서 데이터 x의 개수를 반환

튜플 (Tuple)

- 튜플의 원소가 리스트인 경우는 예외적으로 원소 값 수정이 가능

```
>>> t = (1,[1,2],[],"hi")   # t= 1, [1,2], [], "hi“
>>> t[1][1] = 4             # 원소가 리스트인 경우 수정 가능
>>> t[2][0:0]=[10,20]       # 빈 리스트에 원소 추가
>>> print(t)
(1, [1, 4], [10, 20], 'hi')
>>>
```

```
>>> t = (1,[1,2],[],"hi")   # t= 1, [1,2], [], "hi“
>>> t[2]=[10,20]            # 원소를 다른 리스트로 수정
TypeError: 'tuple' object does not support item assignment
>>> print(t)
(1, [1, 2], [], 'hi')
>>>
```

튜플 (Tuple) 예제(example)

- 다양한 Tuple 사용 예제

```
data1 = (123)                                   # 단일 원소를 가진 tuple을 선언할 때는
data2 = (123,)                                  # 단일 원소 뒤에 ,를 붙여줘야 한다
print("type of data1 : ",type(data1))           # ,를 넣지 않은 경우의 자료형(int) 출력
Print( " type of data2 :  " ,type(data2))       # ,를 넣어서 선언한 경우의 자료형(tuple) 출력

Tuple_1 = (1, 2, 3); tuple_2 = (4, 5, 6)                #두개의 tuple 선언
tuple_3 = tuple_1 + tuple_2                     #tuple의 병합
Tuple_1 = tuple_1 + data2                       #tuple과 단일 원소 tuple의 병합
#tuple_1 = tuple_1 + data1                      #tuple과 정수형의 병합 error(tuple 원소 변경 불가)
#tuple_1[1] = data1                             #tuple index를 통한 원소 변경 불가
Print(tuple_3, tuple_1)

tuple_4 = tuple_1[0:1] + tuple_2[0:1]           #tuple slicin을 이용하면 새로운 tuple로 병합
tuple_5 = tuple_1[0] + tuple_2[0]               #tuple index를 이용하면 덧셈 연산
print(tuple_4)
print(tuple_5)
```

- 출력 결과

```
type of data1 :  <class 'int'>
type of data2 :  <class 'tuple'>
(1, 2, 3, 4, 5, 6) (1, 2, 3, 123)
(1, 4)
5
```

집합 (Set)

- 데이터들을 순서 없이 모아둔 자료형
- 형식(syntax)

  집합명 = {원소1, 원소2, 원소3, ...}

  - 중괄호 { }를 사용하여 표시, 공집합도 가능
  - 값을 바꿀 수 없는 객체만 원소로 올 수 있음
  - 리스트처럼 내용 변경이 가능한 object는 원소로 올 수 없음
- 집합 자료형의 특징
  - 중복 불가능
    - 집합에는 동일한 원소가 두 개 이상 있을 수 없음
    - 자료의 중복을 제거하기 위한 필터 역할로 활용될 수 있음
  - 순서가 없음(Unordered)

집합 (Set)

- 빈 집합 생성(공집합)

```
s = set()
# s = {} 명령어는 빈 사전(dictionary) 생성
```

- 집합은 순서가 없기(unordered) 때문에 출력하면 원소들이 임의의 순서로 출력되며, 인덱싱이 지원되지 않음
  - 집합의 원소를 인덱싱으로 참조하려면 리스트나 튜플 자료형으로 변환한 후 인덱싱 사용
- 집합은 원소의 추가 / 삭제 가능
- membership 연산자(in, not in), loop를 통한 원소 검색 가능
- set() 함수 이용 집합 생성
  - set() 함수의 인수로 리스트(또는 문자열, 튜플) 데이터를 입력하면 해당 원소들을 자신의 원소로 하는 집합을 만듬

집합 (Set)

```
>>> S1 = {3, "cat", False}          # 값을 바꿀 수 없는 자료형만 원소로 옴
>>> S2 = set([1,2,3,2])             # 리스트의 원소를 집합으로 변환
>>> print(S2)                       # 중복 값은 제거
{1, 2, 3}
>>> S3 = set("Hi Hong!")            # 문자열의 문자를 집합으로 변환
>>> print(S3)
{' ', 'H', 'g', '!', 'i', 'n', 'o'}
>>> S4 = { 1, 2, [3, 4] }           # 오류 : 리스트는 원소로 올 수 없음
TypeError: unhashable type: 'list'
```

집합 (Set)

- 집합 객체의 메소드

```
>>> dir(set)
['__and__', '__class__', ..... '__xor_', 'add', 'clear', 'copy',  'differen
ce', 'difference_update', 'discard', 'intersection',  'intersection_up
date', 'isdisjoint', 'issubset', 'issuperset', 'pop',  'remove', 'symme
tric_difference', 'symmetric_difference_update',  'union', 'update']
```

집합 (Set)

- 집합 객체의 메소드

Method	Description (A, B : 집합, x : 임의의 object)
A.add(x)	데이터 x를 A의 원소로 추가
A.clear( )	A의 모든 원소를 제거. A를 공집합으로 만듬
B = A.copy()	A를 B로 복사(shallow copy)
A.discard(x)	집합 A에서 원소 x를 제거. 없는 원소를 삭제하려고 할 때에도 에러 발생하지 않음(x ∉ A이면 무시)
A.remove(x)	집합 A에서 원소 x를 제거. 없는 원소를 삭제하려고 하면 KeyError가 발생(x ∉ A이면 KeyError)
A.pop( )	집합 A에서 임의의 원소를 하나 지우고 그 값을 반환(A가 공집합이면 KeyError). 순서가 없기 때문에 임의의 원소 가져옴
A.union(B)	A ∪ B를 반환 (A | B 와 동일)
A.intersection(B)	A ∩ B를 반환 (A & B와 동일)
A.difference(B)	B에는 없고 A에만 있는 원소의 집합 반환 (A – B와 동일)

집합 (Set)

- 집합 객체의 메소드(계속)

Method	Description(A, B : 집합)
A.isdisjoint(B)	A ∩ B = ∅이면 True
A.issubset(B)	A ⊆ B 이면 True
A.issuperset(B)	A ⊇ B 이면 True
A.update(B)	A를 A ∪ B로 갱신
A.intersection_update(B)	A를 A ∩ B로 갱신
A.difference_update(B)	A를 A - B로 갱신
A.symmetric_difference(B)	A ∪ B에는 있으나 A ∩B에는 없는 원소의 집합 반환(A ^ B와 동일)
A.symmetric_difference_update(B)	A를 (A ∪ B) – (A ∩B)로 갱신

집합 (Set)

```
>>> s1 = set([1, 2, 3])
>>> s1.add(4)              # 존재하는 데이터를 추가하면 변동이 없음
>>> s1
{1, 2, 3, 4}
>>> s1.update([4, 5, 6]) # 4,5,6를 원소로 추가(4는 이미 존재)
>>> s1
{1, 2, 3, 4, 5, 6}
>>> s1.remove(6)           # 없는 원소를 remove할 경우는 KeyError 발생
>>> s1
{1, 2, 3, 4, 5}
>>> s1.discard(6)          # 없는 원소를 삭제하면 무시.(Error 발생 없음)
>>> s1.pop()               # 집합에서 임의의 원소를 반환하면서 삭제
1
>>> s1.clear()             # 집합 s1을 공집합으로 만듬
>>> s1
set()
>>>
```

집합 (Set)

```
>>> L = [1,1,2,2]; B = {3,4,5,6}
>>> A = set(L)
>>> A = A.union({3,4})   # A = {1,2,3,4} B = {3,4,5,6}
>>> C = A | B            # A ∪ B (합집합, union)
>>> D = A & B            # A ∩ B (교집합, intersection)
>>> E = A.difference(B) # E = A – B 와 동일  (차집합, difference)
>>> print(C, D, E)
{1, 2, 3, 4, 5, 6} {3, 4} {1, 2}
>>>
```

집합 (set) 예제(example)

- 다양한 set 사용 예제

```
s1 = set([1,2,3,4,5,6,7,8,9])                    #다양한 형태의 set 선언
s2 = {1,3,5,7,9}
s3 = {2,4,6,8}
s4 = { ' a ' , ' b ' , ' c ' , ' d ' , ' e ' , ' f ' , ' g ' , ' h ' }
s5 =  set( " hello, world " )
print(type(s1),type(s2),type(s3),type(s4))       #set 자료형 확인

if s2.isdisjoint(s3):                            #s2 ∩ s3 = ⌀이면 True 반환
    print("s2 and s3 have no intersection.")     #교집합이 없다 출력
s2.update(s3)                                    #s2와 s3의 합집합으로 s2를 갱신
If s2.issubset(s1) :                             #s2가 s1의 부분 집합이라면 True 반환
    print( " s2 and s3 are in s1. " )            #s2와 s3는 s1의 부분집합임을 출력
print(s4.symmetric_difference(s5))               #s4 ∪ s5에는 있으나 s4 ∩ s5에는 없는 원소의 집합  반환
temp=s4                                          #s4의 데이터 temp에 임시저장(s4가 변경될 것이기 때문)
s4.difference_update(s5)                         #s4를 s4 – s5로 갱신(갱신 메소드는 반환값이 없다)
print(s4,"₩n",temp)                              #변경된 s4(차집합)과 메소드를 이용한 차집합 반환의 비교
```

- 출력 결과

```
<class 'set'> <class 'set'> <class 'set'> <class 'set'>
s2 and s3 have no intersection.
s2 and s3 are in s1.
{'o', 'b', ',', 'c', 'f', 'w', 'l', 'r', 'a', ' ', 'g'}
{'b', 'c', 'f', 'a', 'g'}
 {'b', 'c', 'f', 'a', 'g'}
```

실습

사전 (Dictionary)

- 사전은 키(key)와 값(value)의 쌍을 저장하는 순서가 없는 테이터 형

키(key)	값(value)
"AI lab"	922
"CAD lab"	905
"DB lab"	1003

- 사전은 중괄호 { }로 표시하며, 원소들은 콤마로 분리

사전명 = {키1:값1, 키2:값2, ...}

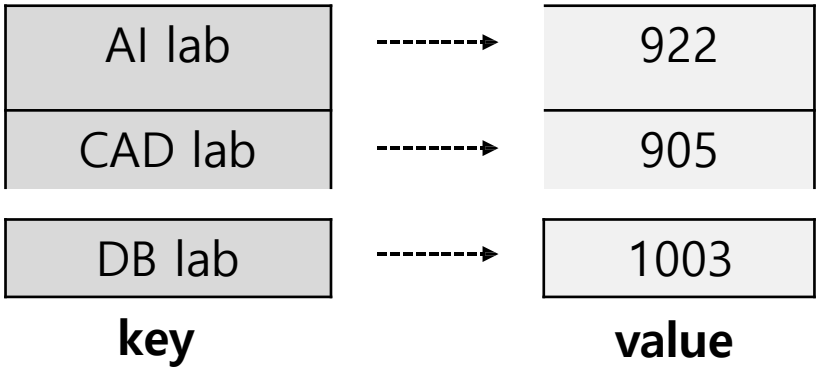
- key에는 정수/실수/문자열/튜플 같은 immutable 자료형만 허용되며, mutable한 자료형 (리스트, 집합, 사전)은 올 수 없음
- value에는 어떤 객체도 가능
- key 값은 고유한 값만 올 수 있으며, 중복되는 key 값을 설정하면 하나를 제외한 나머지 것들이 모두 무시됨

사전 (Dictionary)

- 빈 사전 생성

```
d = dict()
d = {}  # 빈 사전(dictionary) 생성
```

- 사전은 key 값으로 이에 대응되는 value 값을 구하는 방식
- 인덱싱에서의 기호 [ ] 안에 인덱스 번호가 아닌 key 값을 주어 해당 value 값을 참조
- in, not in 연산자 , len() 함수는 사용 가능
- +, * 연산자는 사용할 수 없음



사전 (Dictionary)

- 사전 생성

```
>>> d = dict()                  # 빈 사전(dictionary) 생성
>>> s = {}                      # 빈 사전(dictionary) 생성
>>> number = {1:25, 2:30, 3:27}
>>> type(number)
<class 'dict'>
>> number[2]                   #key값이 2인 원소의 value 값을 가져옴
30
>>> D1 = dict( [(1,'a'), (2,'b')] )  # 쌍으로 구성된 원소를 갖는
>>> D2= dict( ({1,'a'}, {2,'b'}) )   # list, tuple, set 등의 iterable한
>>> D3= dict( {(1, 'a'), (2,'b')} )  # 데이터로 사전 생성.
>>> print(D1, D2, D3)                # D1,D2,D3는 동일한 사전
{1: 'a', 2: 'b'} {1: 'a', 'b': 2} {1: 'a', 2: 'b'}
>>>
```

set 자료형은 순서가 없기 때문에 임의의 원소가 key로 설정됨

사전 (Dictionary)

- 사전에 원소(쌍) 추가하기

```
d[key] = value    # d: 사전형 변수
```

key : value 쌍이 원소로 사전 d에 추가

- 사전에서 원소 삭제하기

```
del d[key]    # d: 사전형 변수
```

key에 해당하는 {key: value} 쌍이 사전에서 삭제

```
>>> d = { }
>>> d[1] = 'a'    # key 1, 값 'a' 인 원소 추가
>>> d[2] = 'b'    # key 2, 값 'b' 인 원소 추가
>>> print(d)
{1: 'a', 2: 'b'}
>>> d[2] = 'c'    # key 2의 값을 'c' 로 수정
>>> print(d)
{1: 'a', 2: 'c'}
>>> del d[1]      # key 1에 해당하는 원소 쌍 삭제
>>> print(d)
{2: 'c'}
```

사전 (Dictionary)

- 아래 내장 함수들은 리스트 등에서 이미 언급한 것들임

function	Description (인수 D : 사전)
len(D)	D의 원소 개수를 반환
sorted(D)	D의 key를 정렬한 리스트 반환. D는 불변이고 오름차순 또는 내림차순으로 정렬 가능. 예) sorted(D, reverse=True)
list(D)	D의 key들을 리스트로 변환하여 반환
set(D)	D의 key들을 집합으로 변환하여 반환
tuple(D)	D의 key들을 튜플로 변환하여 반환

사전 (Dictionary)

사전의 Method	Description （ D : 사전 ）
D.clear( )	D의 모든 원소를 삭제. D = { }가 됨
D.items( )	key와 value의 쌍을 튜플로 묶은 값을 dict_items 객체로 반환. 튜플(key, value)를 원소로 하는 리스트를 만들거나 (key, value)가 필요한 반복문에서 사용
D.keys()	딕셔너리 D의 Key만을 모아서 dict_keys객체로 반환. 필요한 반복문에서 사용
D.values( )	value로 구성된 dict_values객체를 반환
D.get(key[,d])	key에 대한 value 값 반환. 존재하지 않는 key이면 d 반환, d가 주어지지 않았으면 None 반환
D.pop(key[,d])	key에 대한 value 반환하고 해당 원소를 삭제. 존재하지 않는 key이면 d 반환, d가 주어지지 않았으면 KeyError
D.copy( )	D를 복사하여 반환 (shallow copy).
D.update(other)	존재하는 key이면 value를 갱신, 없으면 쌍을 추가. 예) D.update([('a',2),('b',3)]) #D에 없으면 ('a',2)와 ('b',3)를 추가.

반환값 dict_keys 객체는 list(a.keys())와 같은 명령어로 리스트로 변환 가능. 리스트로 변환하지 않더라도 기본적인 반복문(예: for문)에 적용할 수 있음(dict_values, dict_items 객체도 동일)

사전 (Dictionary)

- 사전에서 key로 value 얻기 (d.get() 메소드, d는 사전 변수라고 가정)
  - d.get(x) : 사전 d에서 key 값 x에 대응되는 value 반환
  - d[x]와 동일한 결과값을 반환
  - x가 존재하지 않는 key 값일 경우, d.get(x)는 None 반환. d[x]는 key 값 오류 발생
- 사전에서 항목 삭제하기
  - d.pop(x): 사전 d에서 key 값 x에 대응되는 value를 반환하고 해당 원소 쌍 삭제

```
>>> grade = {'Kim' : 89,  'Park' : 45, 'Lee':78}
>>> num = grade.get('Kim')      # grade['Kim'] 와 동일
>>> num
89
>>> num = grade.pop('Kim')
>>> num
89
>>> grade, type(grade)
({'Park': 45, 'Lee': 78}, <class 'dict'>)
```

사전 (Dictionary)

- 사전에서 key 값들 가져오기(d는 사전 변수라고 가정)
  - d.keys() : 사전 d의 모든 key 값들을 dict_keys 객체로 반환
  - dict_keys 객체로 받은 값을 리스트와 같은 자료형으로 변환하여 사용(리스트의 메소드를 적용하는 것이 코딩에 유용하기 때문)
- 사전에서 key 값, value 값들 동시에 가져오기
  - d.items() : 사전 d에서 key와 value의 쌍을 튜플로 묶은 값들을 dict_items 객체로 반환

```
>>> grade = {'Kim' : 89,  'Park' : 45, 'Lee':78}
>>> L = grade.keys()
>>> L, type(L)
(dict_keys(['Kim', 'Park', 'Lee']), <class 'dict_keys'>)
>>> L = list(L)
>>> L, type(L)
(['Kim', 'Park', 'Lee'], <class 'list'>)
>>> item = list(grade.items())
>>> item, type(item)
([('Kim', 89), ('Park', 45), ('Lee', 78)], <class 'list'>)
```

사전 (Dictionary)

- 사전에서 value 값들 가져오기 (d는 사전 변수라고 가정)
  - d.values() : 사전 d에서 value 값들을 dict_values 객체로 반환
- 사전에서 해당 key가 존재하는지 검사
  - in 연산자 : membership 검사는 key 값만 가능
- 사전에서 모든 원소들 제거
  - d.clear() : 원소들만 제거하고 빈 사전만 남음

```
>>> grade = {'Kim' : 89,  'Park' : 45, 'Lee':78}
>>> values = list(grade.values())
>>> values, type(values)
([89, 45, 78], <class 'list'>)
>>> 'Kim' in grade
True
>>> 89 in grade          # 사전의 key 값에서만 검사
False
>>> grade.clear()
>>> grade
{}
```

사전 (Dictionary) 예제

```
D = {'a' : 1, 'b' : 2, 'c' : 3, 'd' : 4}
print(len(D))                    # 4 (D의 원소 수 계산)
keyList = sorted(D)              # D의 key 값을 기준으로 정렬한 리스트
print(keyList)                   # ['a', 'b', 'c', 'd']

print(D['b'])                    # 2 (key가 'b'인 원소의 value를 반환)
print(D.get('b'))                # 2 (D['b']와 동일)
print(D.get('e'))                # None (사전에 없어 None 반환)
print(D.get('e',0))              # 0 (사전에 없어 0 반환(다른 값 가능))
#print(D['e'])                   # KeyError (사전에 없어 오류-> 주석처리)

print(D.pop('c'))                # 3 ('c'의 value 반환, ('c':3) 제거)
print(D)                         # {'a': 1, 'b': 2, 'd': 4}
print(D.pop('c', 1))             # 1 ('c'가 없음. 1 반환)
#print(D.pop('c'))               # KeyError (사전에 없어 오류->주석처리)
```

- 출력 결과

```
4
['a', 'b', 'c', 'd']
2
2
None
0
3
{'d': 4, 'a': 1, 'b': 2}
1
```

반복(iterable) 객체 관련 내장 함수

- 반복 객체 : 문자열, 리스트, 딕셔너리, 집합

```
>>> max("abcdkhslow"), min("abcdkhslow")
('w', 'a')
>>> sorted("abcdkhslow")
['a', 'b', 'c', 'd', 'h', 'k', 'l', 'o', 's', 'w']
>>> a = zip(("a","b","c"),[10,20,30])
>>> a
<zip object at 0x03392A80>
>>> a = list(a)
>>> a
[('a', 10), ('b', 20), ('c', 30)]
>>> a = ["10","20","30","40"]
>>> a = map(int, a)
>>> a
<map object at 0x033C4690>
>>> a = list(a)
>>> a
[10, 20, 30, 40]
```

반복 가능한 데이터를 쌍으로 묶어주는 함수

두번째 인수인 반복 객체의 모든 값들에게 첫번째 인수인 함수를 적용

실습