[복합/군집 자료형]

복합/군집 자료형

- 컨테이너/군집/Collection 자료형: str: "문자열" , list:[리스트], tuple:(튜플), dict:{사전}, set:{집합}

- list (순서O,중복O,변경O), tuple(순서O,중복O,변경X),dict(순서X,중복X,변경O), set(순서X,중복X,변경O),

srt(순서O,중복O,변경X)

여러 개의 자료를 한 변수에 할당

- 파이썬에서는 하나의 변수에 여러 개의 자료를 한꺼번에 할당하고, 필요한 때 꺼내 쓸 수도 있다.

- Data Types(리스트, 튜플, 집합, 딕셔너리) : 리스트 = [ ] , 튜플 = ( ) , 셋 = { } , 딕셔너리 { : }

>>> a = 10
>>> type(a)
<class 'int'>
	>>> d = True
	>>> type(d)
	<class 'bool'>
		>>> g = (1,3,5)
		>>> type(g)
		<class 'tuple'>

>>> b = 3.5
>>> type(b)
<class 'float'>
	>>> e = 'hello'
	>>> type(e)
	<class 'str'>
		>>> h = {2,4,6,8}
		>>> type(h)
		<class 'set'>

>>> c = 3+7j
>>> type(c)
<class 'complex'>
	>>> f = [1,2,3,4]
	>>> type(f)
	<class 'list'>
		>>> i = {'baseball':9, 'soccer':11}
		>>> type(i)
		<class 'dict'>

[참고] 데이터 타입에 따라, 각각 사용되는 연산자와 메소드(함수)가 다르다.

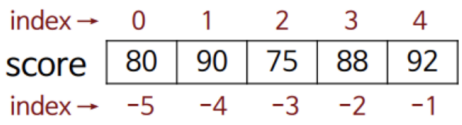
- int, float, str, bool, list, tuple, dict 은 built-in Class들이다
- int( ),float( ), list( ),..등등 는 객체를 만드는 Class라고 말 할 수 있다.

In [ ]:
```
# [참고] 비교: i = int(), f = float(), s = str()
# int, float, str, bool, list, tuple, dict .등등 은 builtin으로 설정된 Class들이다
l = list([1,2,3])
i = int(3)
f = float(3.1)
s = str("hi")
print(type(l))
print(type(i))
print(type(f))
print(type(s))
```

[리스트(list) 자료형 (순서O,중복O,변경O)] : [ ]

복합자료 모임을 하나의 변수에 담아서 처리(순서가 있음)

- 다른 언어에서는 `**배열**` 이라고 함
- `list는 자료들 모임 /입력된 순서 유지`
- **어떤 자료형도 할당 할수 있다 (파이썬의 어떤 객체도 원소로 넣을 수 있음(다른 컬렉션도 가능))**
- `같은 목적의 유사한 원소` 를 묶어서 리스트형 변수로 선언 (예: score = [80, 9-, 75, 88, 92])
- 각 데이터를 `원소(요소)` 라 함
- 각 원소는 순서에 따라 인덱스가 부여됨, 각원소 변경가능
- **(비교) 튜플은 리스트와 비슷하나 변경 할 수 없음(읽기전용)**



In [ ]:
```
# 여러개 정수형 자료형 와 리스트 비교
score0 = 87; score1 = 84; score2 = 95; score3 = 67; score4 = 88; score5 = 94
score_list = [87, 84, 95, 67, 88, 94, 63]
```

리스트 생성

- 순서가 있음, 인덱스를 이용하여 데이터에 접근 가능
- 리스트을 이용하면 1, 3, 5, 7, 9라는 숫자 모음 전체를 한개의 이름(리스트명) 으로 표현 (예: 홀수)

- 리스트명 = [값, 값, 값] 또는 list()

- 대괄호([ ])로 감싸 주고 각 요소값들은 쉼표(,)로 구분해 준다.
- 변수에 값을 저장할 때 로 묶어주면 리스트가 되며 각 값은 ,(콤마)로 구분해줍니다.

In [ ]:
```
# 빈요소
a = []
a
```

In [ ]:
```
# 숫자
scores = [95, 75, 85] # list((95, 75, 85))
print(scores)
type(scores) # 자료형
```

In [ ]:
```
# 문자
city = ['서울', '부산', '인천', '대전']
city
```

In [ ]:
```
# 숫자, 문자 혼합
person = ['james', 17, 175.3, True]
person
```

In [ ]:
```
['james', 17, 175.3, True, [95, 75, 85]] # 리스트 안에 리스트
```

In [ ]:
```
# 직관적 (2차원 배열)
x = [[85,  90,  20,  50],
     [70, 100,  70,  70],
```

```
     [25,  65,  15,  25],
     [80,  45,  80,  40],
     [35,  50,  75,  25]]
x
```

- 형변환(Casting) 다른 자료형의 데이터를 리스트로 변환하기 : 리스트명 = list( )

In [ ]:
```
a1 = []              # 빈 리스트 만들기 1
a2 = list()          # 빈 리스트 만들기 2
a3 = list((3, 5, 9)) # 튜플을 리스트로 만들기 --> [3, 5, 9]
a4 = list("Apple")   # 문자열을 리스트로 만들기 --> ['A', 'p', 'p', 'l', 'e']
a1, a2, a3, a4
```

In [ ]:
```
# 비교: i = int(), f = float(), s = str()
l = list()
i = int()
f = float()
s = str()
l, i, f, s
```

- 내장함수 range를 사용하여 리스트 만들기 : list (range(start, end, step))사용

- range(횟수)함수: 연속된 숫자(정수)를 만들어 줌
- 리스트 = list(range(횟수))
- 리스트 = list(range(시작, 끝))
- 리스트 = list(range(시작, 끝, 증가폭))

In [ ]:
```
list(range(10))# 연속된 숫자(정수)를 만들어주는 range() 함수
```

In [ ]:
```
# range를 사용하여 리스트 만들기 : list (range(start, end, step) )사용

ar = list(range(10)) # 0에서 1씩증가 10미만 : list(range(0,10))
br = list(range(3, 10)) # 3부터 1씩증가 10미만
cr = list(range(3, 10, 2)) # 3부터 2씩증가 10미만

ar, br, cr
```

리스트의 수정과삭제(del)

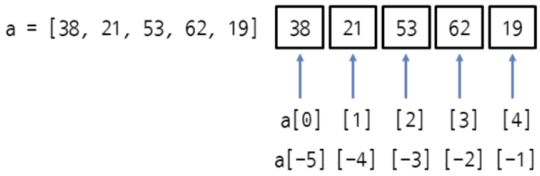
In [ ]:
```
a = [1, 2, 3]
a[2] = 4
a
```

In [ ]:
```
a = [1, 2, 3]
del a[1]
a
```

In [ ]:
```
a = [1, 2, 3, 4, 5]
del a[2:]
a
```

리스트 indexing 과 Slicing

- 리스트도 문자열처럼 인덱싱과 슬라이싱이 가능



In [ ]:
```
# 인덱싱
a = [38, 21, 53, 62, 19]
print(a[0])
print(a[2])
print(a[-1]) # 리스트의 뒤에서 첫 번째(인덱스 -1) 요소 출력
print(a[-5]) # 리스트의 뒤에서 다섯 번째(인덱스 -5) 요소 출력

# 슬라이싱
print(a[0:3])
print(a[2:])
print(a[-3:-1])
print(a[-3:])
```

- 리스트 인덱싱 (indexing)

In [ ]:
```
a = [1, 2, 3]
a
```

In [ ]:
```
a[0]
```

In [ ]:
```
a[0] + a[2]
```

In [ ]:
```
a = [1, 2, 3, ['a', 'b', 'c']]
a
```

In [ ]:
```
a[0], a[-1], a[3]
```

In [ ]:
```
a[-1][0] # 이중인덱싱
```

In [ ]:
```
a[-1][1], a[-1][2]
```

- 리스트 슬라이싱(slicing)

- 문자열과 마찬가지로 리스트에서도 슬라이싱 기법을 적용할 수 있다.
- 슬라이싱은 자른다,나눈다 뜻
- 리스트 슬라이싱 위치는 아래와 같은 구조 --> `그부분을 자른다`

In [ ]:
```
alpha = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

In [ ]:
```
alpha[2:6]    # 2 이상 6 미만 위치의 범위 선택
```

In [ ]:
```
alpha[:3]     # 3 미만 위치의 범위 선택 (시작 위치 생략)
```

In [ ]:
```
alpha[5:]     # 5 이상 위치의 범위 선택 (종료 위치 생략)
```

In [ ]:
```
alpha[:]      # 전체 범위 선택 (시작, 종료 위치 모두 생략)
```

In [ ]:
```
alpha[::2]    # 전체 범위에서 두 요소마다 하나씩 선택
```

In [ ]:
```
alpha[1::2]  # 1 이상의 범위에서 두 요소마다 하나씩 선택
```

In [ ]:
```
alpha[::-1]  # 전체 범위에서 뒤에서부터 한 요소마다 하나씩 선택
```

리스트 연산

- 리스트에 대한 +(합병) , *(반복) 연산

- 리스트 합병(+)/반복하기(*)

- 문자열에서 "abc" + "def" = "abcdef"가 되는 것과 같은 이치
- 문자열에서 "abc" * 3 = "abcabcabc" 가 되는 것과 같은 이치

In [ ]:
```
a = [1, 2, 3]
b = [4, 5, 6]
a + b
```

In [ ]:
```
# a * b # 에러
```

In [ ]:
```
a = [1, 2, 3]
a * 3
```

리스트 기본 함수 / 메소드 (Built-in Functions & Methods)

- `built-in 함수와 메소드의 차이` : built-in 함수는 파이썬에서 별다른 조작없이 사용할 수 있는 함수이며, `특정 객체 등에 속해있지 않음`
- 메소드는 다른 객체와 연관되어 있음. 즉, 메소드는 특정 객체에 대하여 정의되어 있는 함수
- 따라서 해당하는 object.method_name()과 같이 객체 이후 점(.) 아래 레벨에서 사용

내장함수 (Built-in Fuction)

- 파이썬에서 범용적으로 자주 사용되는 70여개 함수 --> `내장 함수 (Built-in Functions)`
- 그중 리스트 관련 built-in functions : list(), sorted(), len(), sum(), max(), min() 등

In [ ]:
```
# 리스트로 변환
list((1,2,3)), list("Hello"), list({1,2,3}), list(range(10))
```

In [ ]:
```
# 자료의 개수 구하기 (리스트 길이)
a = [1, 3, 5, 2, 4]
len(a)
```

리스트 관련 메소드 (변수/객체.메소드())

- 문자열과 마찬가지로 리스트 변수명 뒤에 '.'를 붙여서 사용
- 객체/변수.메소드()

메소드	하는 일
index(x)	원소 x를 이용하여 위치를 찾는 기능을 한다.
append(x)	원소 x를 리스트의 끝에 추가한다.
count(x)	리스트 내에서 object 원소의 개수를 반환한다.
extend([x1, x2])	[x1, x2] 리스트를 리스트에 삽입한다.
insert(index, x)	원하는 index 위치에 x를 추가한다.
remove(x)	x 원소를 리스트에서 삭제한다.
pop(index)	index 위치의 원소를 삭제한 후 반환한다. 이때 index는 생략될 수 있으며 이 경우 리스트의 마지막 원소를 삭제하고 이를 반환한다.
sort()	값을 오름차순 순서대로 정렬한다. reverse 인자의 값이 True 이면 내림차순으로 정렬한다.
reverse()	리스트를 원래 원소들의 역순으로 만들어 준다.

- 리스트 요소(원소) 추가 : +, extend(seq), append(obj), insert(obj, index)

- +와 extend()는 이어붙이기(concatenation) 연산
- append()와 insert()는 원소 한 개를 리스트에 추가하는 메소드

In [ ]:
```
# 이어붙이기(concatenation) 연산
a = [1, 2, 3]
b = [4, 5, 6]
a + b
```

In [ ]:
```
# 리스트 확장(extend)
# extend(): 리스트를 연결하여 확장 (a.extend(x)에서 x에는 리스트만 올 수 있으며, 원래의 a
a = [1,2,3]
a.extend([4,5])
a
```

In [ ]:
```
x = [6,7]
a.extend(x)
a
```

In [ ]:
```
# append()는 리스트의 마지막에 추가하고 insert(obj, index)는 특정 위치에(index 바로 직전
a = [1, 2, 3]
a.append(4)
a
```

In [ ]:
```
# append() , extend() 차이점
a.append([5,6]) # 비교 a.extend([5,6])
a
```

In [ ]:
```
# 리스트에 특정위치에 요소 삽입(insert)
# insert(a, b)는 리스트의 a번째 위치에 b를 삽입하는 함수
a = [1, 2, 3]
a.insert(0, 4)
a
```

In [ ]:
```
a.insert(3, 5)
a
```

- 리스트 요소 제거 : remove(), pop(), clear()

In [ ]:
```
# remove(x)는 리스트에서 첫 번째로 나오는 x를 삭제하는 함수이다.
# remove(value)는 리스트에 있는 첫 번째 value값을 가지는 원소를 제거. 만약 value값이 없다면
a = [1, 2, 3, 1, 2, 3, 4, 5]
a.remove(3)
a
```

In [ ]:
```
a.remove(3)
a
```

In [ ]:
```
# pop()은 가장 마지막 원소를 지운 후 그 원소를 리턴하고, pop(index)는 index 위치의 원소를 지
a.pop(), a
```

In [ ]:
```
# clear()는 리스트의 모든 원소를 제거하여 리스트를 빈리스트로 만듬
a.clear()
a
```

기타 : sort(), reverse(), index(), count()

- 리스트 정렬(sort)

In [ ]:
```
a = [1, 4, 3, 2]
a.sort()
a
```

In [ ]:
```
a = ['a', 'c', 'b']
a.sort()
a
```

- 리스트 뒤집기(reverse)

In [ ]:
```
a = ['a', 'c', 'b']
a.reverse()
a
```

- 위치 반환(index)

- index(x) 함수는 리스트에 x라는 값이 있으면 x의 위치값을 리턴한다.

In [ ]:
```
a = [1,2,3]
a.index(3)

#a.index(1)
```

In [ ]:
```
 # a.index(0)
```

- 리스트에 포함된 요소 x의 개수 세기(count)

In [ ]:
```
a = [1,2,3,1]
a.count(1)
```

리스트 존재 유무 (요소 in 리스트)

- 리스트에 요소와 동일한 데이터가 있는지 확인

In [ ]:
```
mylist = [1, 2, 'Life', 'is']
'Life' in mylist
```

In [ ]:
```
'Life' not in mylist
```

[튜플(tuple) 자료형 (순서O,중복O,변경X)] : ( )

- 튜플은 본질적으로 list와 매우 유사한 구조 (순서있음), **읽기전용**
- 읽기 전용이라, 관련 메서드 적고, 속도는 빠름
- 튜플은 원소의 추가, 삭제, 수정이 불가능함
- 원소들 변경, 누락이 되어서는 안되는 경우 사용 (예:요일, 년, 등등)

In [ ]:
```
dir(tuple)
```

튜플 생성

- 튜플(tuple)은 몇 가지 점을 제외하곤 리스트와 거의 비슷하며 리스트와 다른 점은 다음과 같다.
- 리스트는 [ ]으로 둘러싸지만 튜플은 ( )으로 둘러싼다.
- 리스트는 그 값의 생성, 삭제, 수정이 가능하지만 튜플은 그 값을 바꿀 수 없다.
- tuple의 원소는 어떤 타입도 가능하다.
- tuple 내부는 ","로 원소를 구분한다.
- 길이 1개짜리 tuple을 만들려면 반드시 뒤에 `콤마(,)를 붙여야함 예) (3,)`
- 빈 튜플: ()
- 요소가 하나인 튜플: (1,) 또는 1,
- 요소가 둘 이상인 튜플: (1, 2) 또는 1, 2

빈 튜플 만들기	tuple0 = ()
하나의 요소를 가진 튜플 만들기	tuple1 = (1,) # 반드시 쉼표를 사용해야 함에 유의
기본적인 튜플 만들기	tuple2 = (1, 2, 3, 4)
간단한 방식의 튜플 만들기	tuple3 = 1, 2, 3, 4
리스트로부터 튜플 만들기	n_list = [1, 2, 3, 4] tuple4 = tuple(n_list)

In [ ]:
```
# 다양한 형태의 튜플 만들기
t0 = ()
t1 = (95,) # 비교 t = (95)
t11 = (95) # 튜플 이 아니다

t2 = (95, 75, 85)
t22 = 1,2,3,4  # 튜플만들기 가능

t3 = ('서울', '부산', '인천', '대전')
t4 = ('james', 17, 175.3, True)

t5 = ('james', 17, 175.3, True, [95, 75, 85], ('서울', '화성고'), {7, 8}, {1:'I
```

```
print(t0, t1, t11, t2, t22, t3, t4, t5)
```

In [ ]:
```
type(t0)
```

튜플 연산

- 튜플의 인덱싱과 슬라이싱, 더하기(+)와 곱하기(*)

- 튜플은 값을 변경시킬 수 없다는 점만 제외하면 리스트와 완전히 동일

- 튜플 더하기

In [ ]:
```
t1 = (1, 2, 'a', 'b')
t2 = (3, 4)
t1 + t2
```

- 튜플 곱하기

In [ ]:
```
t2 = (3, 4)
t2 * 3
```

튜플 indexing / Slicing

- 인덱싱하기

In [ ]:
```
t1 = (1, 2, 'a', 'b')
t1[0], t1[3]
```

- 슬라이싱하기

In [ ]:
```
t1 = (1, 2, 'a', 'b')
t1[1:]
```

관련 내장함수 (Built-in Fuction)

- 튜플 길이 구하기

In [ ]:
```
t1 = (1, 2, 'a', 'b')
len(t1)
```

- 튜플 요소값 삭제 시 오류

In [ ]:
```
t1 = (1, 2, 'a', 'b')
# del t1[0] # 오류 발생
```

In [ ]:
```
dir(tuple)
```

- 리스트에 특정 값이 있는지 체크하기

In [ ]:
```
# 만약 주머니에 돈이 있으면 택시를 타고, 없으면 걸기
pocket = ['paper', 'cellphone', 'money']
if 'money' in pocket:
    print("택시")
else:
    print("도보")
```

In [ ]:
```
pocket = ['paper', 'cellphone', 'money']
if 'money' in pocket:print("택시")
else:print("도보")
```

- 리스트에 특정 값이 없는지 체크하기

In [ ]:
```
# 만약 주머니에 돈이 있으면 택시를 타고, 없으면 걸기
pocket = ['paper', 'cellphone', 'money']
if 'money' not in pocket:
    print("택시")
else:
    print("도보")
```

[딕셔너리/사전(dictionary) 자료형 (순서X,중복X,변경O)]: { }

- 딕셔너리는 데이터들을 <키(Key) : 값(Value)>의 쌍으로 저장하는 순서가 없는 자료구조

- 키(Key)는 중복될 수 없으며 값(Value)은 중복이 가능, 동일한 값이 두 개의 다른 키에 대응 가능

- 단어 뜻 그대로 사전과 비슷한 형식의 자료형
- 하나의 요소가 '키(key)'와 '값(value)'의 쌍(pair)으로 구성 : {apple:사과}
- **key가 될수 있은 자료형은 변경할수 없는 자료형만 가능(list불가)**

In [ ]:
```
a = {"one": "하나", "two": "둘", "three": "셋"}
print(a["one"]) # index대신에 key를 호출하여 value를 출력
```

딕셔너리 생성

In [ ]:
```
d = {'떡볶이':'3000원', '순대':'3000원','튀김':'4000원', '라면':'4000원'}
d
```

In [ ]:
```
type(d)
```

In [ ]:
```
# 오류 발생
# de = { ['떡볶이','순대']:'3000원',['튀김','라면']:'4000원'} # 오류 발생 key의 리스트
```

In [ ]:
```
# 오류 해결
dc = { ('떡볶이','순대'):'3000원',('튀김','라면'):'4000원'} # 오류 발생 key의 리스트형
dc['떡볶이','순대']
```

딕셔너리 추가

In [ ]:
```
a = {1: 'a'}
a[2] = 'b'
a
```

In [ ]:
```
a['name'] = '김길동'
a
```

In [ ]:
```
a[3] = [1,2,3]
a
```

딕셔너리 요소 삭제

In [ ]:
```
d = {'떡볶이':'3000원', '순대':'3000원','튀김':'4000원', '라면':'4000원'}
del d['떡볶이']
d
```

indexing : 딕셔너리에서 Key 사용해 Value 얻기

In [ ]:
```
price = {'떡볶이':'3000원', '순대':'3000원', '라면':'4000원'}
price['튀김'] = '4000원'
price['튀김']
```

In [ ]:
```
price2 = { ('떡볶이','순대'):'3000원',('튀김','라면'):'4000원'} # 오류 발생 key의 리스
price2['떡볶이','순대']
```

In [ ]:
```
dic = {'name':'김길동', 'phone':'0119993323', 'birth': '1118'}
print(dic['name'], dic['phone'], dic['birth'])
```

딕셔너리 관련 메소드

- keys(): 키를 모두 가져옴 d.keys()
- values(): 값을 모두 가져옴 d.values()
- items(): 키-값 쌍을 모두 가져옴 : d.items()
- clear() : dict의 모든 요소 제거 d.clear()
- get(key): key에 저장된 값 반환 d.get(key)

메소드	하는 일
keys()	딕셔너리 내의 모든 키를 반환한다.
values()	딕셔너리 내의 모든 값을 반환한다.
items()	딕셔너리 내의 모든 항목을 [키]:[값] 쌍으로 반환한다.
get(key)	키에 대한 값을 반환한다. 키가 없으면 None을 반환한다.
pop(key)	키에 대한 값을 반환하고, 그 항목을 삭제한다. 키가 없으면 KeyError 예외를 발생시킨다.
popitem()	랜덤하게 선택된 항목을 반환하고 그 항목을 삭제한다.
clear()	딕셔너리 내의 모든 항목을 삭제한다.

In [ ]:
```
dir(dict)
```

- Key 리스트 만들기(keys())

- a.keys()는 딕셔너리 a의 Key만을 모아서 `dict_keys 객체` 를 돌려준다.

In [ ]:
```
# keys(): 키를 모두 가져옴
dic = {'name': '김길동', 'phone': '0119993323', 'birth': '1118'}
dic.keys()
```

In [ ]:
```
list(dic.keys()) # keys 리스트 만들기(values)
```

- Value 리스트 만들기(values())

In [ ]:
```
# values(): 값을 모두 가져옴
dic.values()
```

In [ ]:
```
list(dic.values()) # Values 리스트 만들기
```

- Key , Value 쌍 리스트 만들기(items())

In [ ]:
```
dic.items() # Key, Value 쌍 얻기(items)
```

In [ ]:
```
list(dic.items())
```

- get (key, default값)

- 딕셔너리 안에 찾으려는 Key 값이 없을 경우 미리 정해 둔 디폴트 값을 대신 가져오게 하고 싶을 때 사용

In [ ]:
```
a = {'name':'김길동', 'phone':'0119993323', 'birth': '1118'}
a.get('name') , a.get('phone') # Key로 Value얻기(get)
```

In [ ]:
```
a = {'name':'pey', 'phone':'0119993323', 'birth': '1118'}
print(a.get('address'))
```

In [ ]:
```
a.get('address','없는 키 값') # a 에 'foo'에 해당하는 값이 없음 --> 디폴트 값인 'bar'를
```

- 삭제 : clear() --> 비교 del

In [ ]:
```
dic.clear()  # Key: Value 쌍 모두 지우기(clear)
dic
```

- key in 사전 : 사전에 key 값이 있는지 확인 (key not in 사전)

In [ ]:
```
a = {'name':'김길동', 'phone':'0119993323', 'birth': '1118'}
'name' in a  , 'email' in a
```

[집합(set) 자료형 (순서X,중복X,변경O)] : { }

- 단순한 데이터의 그룹
- 원소간의 순서 없음(인덱스 번호와 같은 것 없음)
- 동일한 요소를 중복하여 가질 수 없음

- 집합을 이용하면 많은 자료를 분리하고 활용하는데 편리

집합(set) 자료형 생성

- Variable_name = {element_1, element_2...}
- 집합에 관련된 것들을 쉽게 처리하기 위해 만들어진 자료형이다.
- set()의 괄호 안에 리스트 또는 문자열을 입력하여 만듬

In [ ]:
```
# 다양한 형태의 집합
s1 = { }
s2 = {1, 2, 3}
s3 = {'Life', 'is', 'too', 'short'}
s4 = {1, 2, 'Life', 'is'}
s5 = {1, 2, (5,6), ('too', 'short')}
print(s1, s2, s3, s4, s5)
```

In [ ]:
```
# s6 = {1, 2, [3,4]} # 에러 발생
s6 = {1, 2, (3,4)} # 에러 해결
print(s6)
```

In [ ]:
```
a1 = [1,2,3]
s1 = set(a1) #리스트형을 set 형으로 변환
s1
```

In [ ]:
```
s2 = set("Hello")
s2
```

집합 자료형의 특징

- 중복을 허용하지 않는다.
- 순서가 없다(Unordered).
- 딕셔너리 역시 순서가 없는 자료형이라 인덱싱을 지원하지 않는다.
  ※ 중복을 허용하지 않는 set의 특징은 자료형의 중복을 제거하기 위한 필터 역할로 종종 사용되기도 한다.

In [ ]:
```
s1 = set([5,1,3,2,1,2,3,'이름',7]) # 중복제거
aa = list(s1)
aa
```

In [ ]:
```
aa[0]
```

In [ ]:
```
t1 = tuple(s1)
t1
```

In [ ]:
```
t1[0]
```

---------- End ----------