제10장
자료구조[스택, 큐, 튜플, 세트]

1. 자료구조의 이해

1) 자료구조의 개념

- 프로그래밍을 하다 보면 다양한 형태의 데이터를 저장하여 처리해야 할 때가 있다. 실생활에서 찾아볼 수 있는 대표적인 데이터 저장 사례로 전화번호부가 있다. 요즘은 전화번호부에서 전화번호를 찾는 일이 거의 없지만, 과거에는 'Yellow Page'라는 두꺼운 전화번호부에서 전화번호를 검색하였다. 전화번호를 효율적으로 찾기 위해서는 이름을 가나다 순서대로 저장하는 것이 좋다. 휴대전화의 연락처에 전화번호를 저장하고 보여 주는 방식이기도 하다. 이러한 방식으로 데이터의 특징을 고려하여 데이터를 저장하는 방법을 자료구조(data structure)라고 한다.

(a) 전화번호부 (b) 휴대전화의 연락처

[실생활 속 자료구조]

1. 자료구조의 이해

1) 자료구조의 개념

- 전화번호부 이외에도 실생활에서 데이터의 특징을 반영하여 저장해야 할 정보는 많다. 예를 들어, 은행의 번호표 처리 방식을 생각해 보자. 은행의 번호표는 번호표 단말기에서 사용자가 번호표를 하나씩 뽑으면 대기 인원이 1씩 증가하고, 해당 사용자가 은행 서비스 이용을 종료하면 1씩 감소한다. 이 경우, 사용되는 번호표의 번호 정보와 현재 대기 인원을 모두 관리한다면 효율적으로 데이터를 관리할 수 있을 것이다. 또 다른 예로, 택배 수화물을 트럭에 쌓을 때 위치 정보를 어떻게 저장할지에 대한 것이다. 나중에 배달하는 수화물일수록 트럭 안쪽에 쌓고, 먼저 배달하는 수화물일수록 바깥에 쌓는 것이 좋다. 이러한 특징을 고려한 데이터 저장 방식은 매우 유용할 것이다.
- 종합하면, 자료구조는 특징이 있는 정보를 메모리에 효율적으로 저장 및 반환하는 방법으로, 데이터를 관리하는 방식이다. 특히 대용량일수록 메모리에 빨리 저장하고 빠르게 검색하여, 메모리를 효율적으로 사용하고 실행 시간을 줄일 수 있게 해 준다.

1. 자료구조의 이해

2) 파이썬에서의 자료구조

- 파이썬에서는 어떤 종류의 자료구조가 있을까? 사실 이미 자료구조의 기본 저장 방식인 리스트에 대해 앞서서 배웠으며, 이외에 자료구조는 아래 표와 같다. 아래에 열거된 다양한 자료구조를 하나하나 배우며 실제 사용할 수 있는 방법에 대해 알아보자.

자료구조명	특징
스택(stack)	나중에 들어온 값을 먼저 나갈 수 있도록 해 주는 자료구조(last in first out)
큐(queue)	먼저 들어온 값을 먼저 나갈 수 있도록 해 주는 자료구조(first in first out)
튜플(tuple)	리스트와 같지만, 데이터의 변경을 허용하지 않는 자료구조
세트(set)	데이터의 중복을 허용하지 않고, 수학의 집합 연산을 지원하는 자료구조
딕셔너리 (dictionary)	전화번호부와 같이 키(key)와 값(value) 형태의 데이터를 저장하는 자료구조, 여기서 키값은 다른 데이터와 중복을 허용하지 않음
collections 모듈	위에 열거된 여러 자료구조를 효율적으로 사용할 수 있도록 지원하는 파이썬 내장(built-in) 모듈

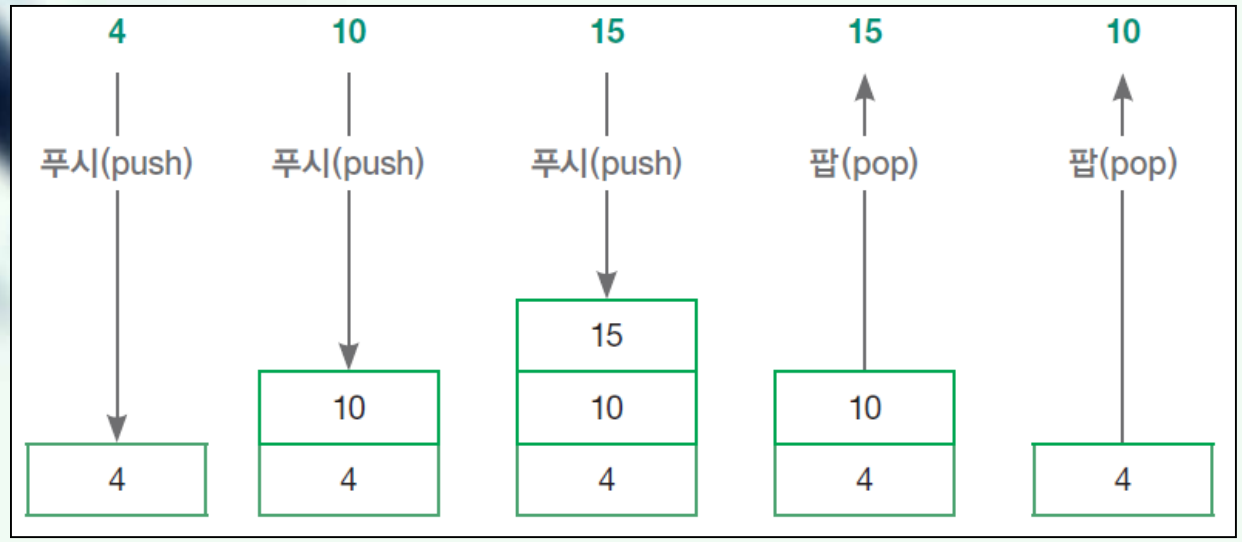
[파이썬에서 제공하는 자료구조]

(참고)이 장에서 다루는 자료구조의 내용이 아주 어렵지는 않지만, 실제 자료구조는 컴퓨터 공학과에서 한 학기 수업으로 가르칠 정도로 조금은 어려운 과목이다. 여기서는 간단한 개요 수준과 파이썬에서 다루는 방식만 배우지만, 컴퓨터 공학과에서는 실제 각 자료구조를 구현하는 방식을 배우기 때문이다. 공부하다가 궁금증이 생기면 자료구조와 알고리즘을 다룬 전문 서적을 참고하는 것을 권장한다.

2. 스택과 큐

1) 스택(stack)

- 자료구조에서 첫 번째로 다룰 주제는 스택이다. 스택 stack은 컴퓨터 공학과 학생들도 전공을 배우면서 처음 배우는 자료구조로, 자료구조의 핵심 개념 중 하나이다. 스택을 간단히 표현하면 LIFO(Last In First Out)으로 정의할 수 있다. 즉, 마지막에 들어간 데이터가 가장 먼저 나오는 형태로, 데이터의 저장 공간을 구현하는 것이다.
- 일반적으로 스택이라고 하면 그림에서 보이는 사각형의 저장 공간을 뜻한다. 즉, 4, 10 과 같은 데이터를 저장하는 공간으로, 리스트와 비슷하지만 저장 순서가 바뀌는 형태를 스택 자료구조(stack data structure)라고 한다. 스택에서 데이터를 저장하는 것을 푸시(push), 데이터를 추출하는 것을 팝(pop) 이라고 한다.



[스택]

2. 스택과 큐

2) 스택의 사용 용도

- 스택은 어떤 상황에서 사용할 수 있을까? 앞서 언급한 사례 중 택배 수화물을 저장하는 방식을 보면, 먼저 배달해야 하는 수화물은 트럭의 바깥쪽에, 나중에 배달해야 하는 수화물은 트럭의 안쪽에 넣는다. 수화물을 하나의 데이터로 본다면 먼저 들어간 수화물보다 나중에 들어간 수화물이 먼저 나와야 하는 경우와 같다. 수화물에 대한 데이터도 이러한 방식으로 저장한다면, 좀 더 쉽게 데이터를 추출할 수 있다.
- 파이선에서는 리스트를 사용하여 스택을 구현할 수 있다. 리스트라는 저장 공간을 만든 후, append() 함수로 데이터를 저장(push)하고 추출(pop)한다. 다음 코드를 확인해 보자.

```
a = [1, 2, 3, 4, 5]
a.append(10)
print(a)
a.append(20)
print(a)
a.pop()
a.pop()
```

```
[1, 2, 3, 4, 5, 10]
[1, 2, 3, 4, 5, 10, 20]
20
10
```

사실 파이썬에서는 훨씬 더 효율적이고 강력한 스택 라이브러리를 제공한다. 추후에 배우겠지만, collections라는 모듈이 있는데 이 모듈에서 deque는 자료구조를 제공하여 조금 더 빠르게 스택을 구현할 수 있다. 이후에 다시 배운다.

2. 스택과 큐

3) 스택으로 만들 수 있는 프로그램

- 스택으로 만들 수 있는 프로그램에는 어떤 것이 있을까? 다양한 프로그램을 만들 수 있지만, 앞서 만들었던 십진수를 이진수로 변환하는 '이진수 변환기' 프로그램이 스택을 사용하여 만들 수 있는 프로그램 중 하나이다. 2로 나눈 나머지 값을 스택에 푸시한 후, 마지막으로 들어온 값부터 팝으로 추출하고 출력한다면 원하는 결과를 얻을 수 있다.
- 또 다른 예로는 입력한 텍스트의 역순을 추출하는 프로그램을 작성하는 것이다. 아래 코드를 보자.

```
word = input("Input a word: ")
world_list = list(word)
print(world_list)

result = []
for _ in range(len(world_list)):
        result.append(world_list.pop())
print(result)
print (word[::-1])
```

```
Input a word: PYTHON  # 사용자 입력(PYTHON)
['P', 'Y', 'T', 'H', '0', 'N']
['N', '0', 'H', 'T', 'Y', 'P']
NOHTYP
```

먼저 입력한 텍스트는 변수 word에 저장되고, 그 값을 리스트형으로 변환한다. 그 후 값을 차례대로 추출하면, 입력한 텍스트의 역순값이 출력된다. 코드에서 확인할 코드가 있다. 바로 _ 기호이다. 이 기호는 파이썬에서 굉장히 많이 쓰이는 코드 중 하나이다. 일반적으로 for문에서 많이 쓰이는데, for문에 기호가 있으면 해당 반복문에서 생성되는 값은 코드에서 사용하지 않는다는 뜻이다. 코드에서는 6행의 range(len(world_list))에서 생성되는 값이 반복문 내에서 사용되지 않으므로 _ 로 할당받은 것이다.

2. 스택과 큐

4) 큐(Queue)

- 이번에는 스택의 반대 개념인 큐에 대해 알아보자. 큐(queue)는 스택과 다르게 먼저 들어간 데이터가 먼저 나오는 FIFO(First in First Out)의 메모리 구조를 가지는 저장 체계이다. 그림은 큐에서 일반적으로 사용할 수 있는 저장 체계이다. 기존에 삽입시 offer(), 추출시 poll()을 사용한다.

PUT(A)	PUT(B)	PUT(C)	GET( )	PUT(D)	GET( )
				D	D
		C	C	C	C
	B	B	B	B	
A	A	A			

[큐]

- 큐는 어떤 상황에서 사용할 수 있을까? 대표적인 예로, 앞서 언급한 사례 중 은행에서 대기 번호표를 뽑을 때 번호를 저장하는 방식이다. 먼저 온 사람이 앞의 번호표를 뽑고, 번호가 빠른 사람이 먼저 서비스를 받는 구조이다.
- 메모리 개념으로 볼 큐는 스택보다 구현이 조금 더 복잡하다. 사실 이 부분은 컴퓨터 공학과 전공 수업에서 배우는 내용이다. 스택은 메모리가 시작하는 지점이 고정되어 있지만, 큐는 처음 값이 저장되는 메모리 주소가 값이 사용됨에 따라 계속 바뀌게 되어 구현에 좀 더 신경을 써야 한다. 파이썬에서는 이러한 부분이 스스로 구현되어 있어서 어렵지 않게 사용할 수 있다.

2. 스택과 큐

4) 큐(Queue)

- 파이썬에서 큐를 구현하는 것은 매우 간단하다. 기본적으로 스택의 구현과 같은데, pop() 함수를 사용할 때 인덱스가 0번째인 값을 쓴다는 의미로 pop(0)을 사용하면 된다. 즉, pop() 함수가 리스트의 마지막 값을 가져온다고 하면, pop(0)은 맨 처음 값을 가져온다는 뜻이다.

```
a = [1, 2, 3, 4, 5]
a.append(10)
print(a)
a.append(20)
print(a)
a.pop(0)
a.pop(0)
```

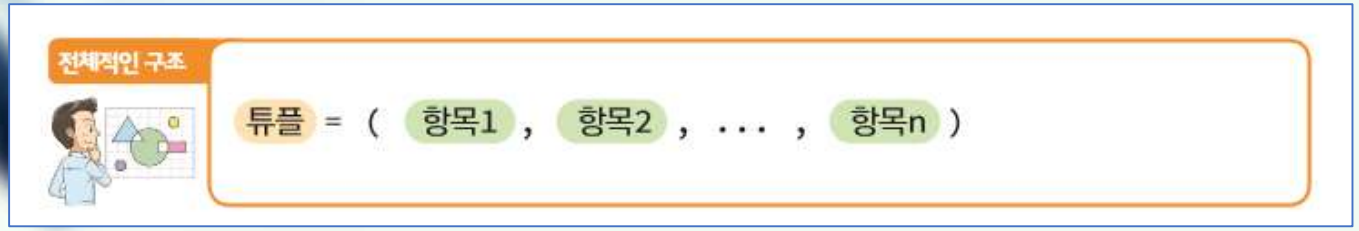
```
[1, 2, 3, 4, 5, 10]
[1, 2, 3, 4, 5, 10, 20]
1
2
```

간단하게 스택과 큐의 개념에 대해 배웠다. 스택과 큐라는 개념 자체는 어렵지 않지만, 실제로 구현하기 쉽지 않다. 하지만 파이썬에서는 그렇게 어렵지 않으니 개념으로 이해하도록 한다.
나중에 메모리 구조를 이해할 정도로 실력이 쌓이면, 직접 구현체를 클래스로 만들어 볼 것이다.

3. 튜플

1) 튜플(tuple)의 개념

- 튜플(tuple)은 리스트와 아주 유사하다. 하지만 튜플의 내용은 변경될 수 없다. 어떻게 보면 리스트보다 튜플이 불편하다. 그렇다면 왜 튜플을 사용하는 것일까? 어떤 경우에는 리스트가 한번 만들어지면 내용을 변경할 수 없게 하는 것이 도움이 된다. 이유는 리스트는 변경될 수 있는 객체이기 때문에 실수로 요소가 추가되거나 삭제, 변경될 수 있기 때문이다. 또 튜플은 리스트에 비하여 접근 속도가 빠르다. 아울러 튜플도 시퀀스의 일종이다.
- 시퀀스에 속하는 자료 구조들은 동일한 연산을 지원한다. 인덱싱(indexing), 슬라이싱(slicing), 덧셈 연산(adding), 곱셈 연산(multiplying) 이 지원된다.



- 예를 들어서 색상을 저장하는 튜플을 생성하면 다음과 같다.

```
colors = ("red", "green", "blue")
print(colors)
numbers = (1, 2, 3, 4, 5 )
print(numbers)
```

```
('red', 'green', 'blue')
(1, 2, 3, 4, 5)
```

3. 튜플

1) 튜플(tuple)의 개념

- 물론 정수를 저장하는 튜플도 생성할 수 있다.

```
numbers = (1, 2, 3, 4, 5)
print(numbers)
```

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

- 튜플도 리스트와 마찬가지로 여러 가지 자료형의 값을 섞어서 생성할 수 있다.

```
t = (1,2, 'hello!')
print(t)
```

```
(1, 2, 'hello!')
```

- 공백 튜플은 단순히 소괄호만 적어주면 된다.
- 하나의 값 만을 가진 튜플을 생성할 때는 반드시 값 다음에 쉼표를 붙여야 한다. 쉼표가 없으면 단순한 수식으로 처리된다. 즉 (10)은 정수 10이나 마찬가지이다.

```
t = (10, )
```

- 또 리스트로부터 튜플을 생성할 수도 있다.

```
t = tuple ([1, 2, 3, 4, 5])
```

3. 튜플

1) 튜플(tuple)의 개념

- 튜플도 리스트와 마찬가지로 내부에 다른 튜플을 가질 수 있다.

```
t = (1, 2, 'hello!')
u = t, (1, 2, 3, 4, 5)
print(u)
```

```
((1, 2, 'hello!'), (1, 2, 3, 4, 5))
```

- 튜플은 시퀀스의 일종이기 때문에 모든 시퀀스 연산이 적용된다. 또 len(), min(), max()와 같은 함수들을 사용할 수 있다.

```
numbers = (1, 2, 3, 4, 5)
print(len(numbers))
```

```
5
```

- 튜플은 변경될 수 없는 객체이므로 튜플의 요소는 변경될 수 없다.

```
t1 = (1, 2, 3, 4, 5)
t1[0] = 100
Traceback (most recent call last):
File "<pyshell#11)", line 1, in <module>
t1[0]=100
TypeError: tuple' object does not support item assignment
```

3. 튜플

1) 튜플(tuple)의 개념

- 하지만 2개의 튜플을 합하여 새로운 튜플을 만들 수는 있다. 새로운 튜플을 만드는 것은 허용되지만 한번 만들어진 튜플은 수정할 수 없는 것이다.

```
numbers = ( 1, 2, 3, 4, 5 )
colors = ("red", "green", "blue")
t = numbers + colors
print(t)
```

```
(1, 2, 3, 4, 5, 'red', 'green', 'blue')
```

2) 기본적인 튜플 연산들

- 튜플은 +와 와 같은 연산자에 반응한다. 리스트와 동일하게 +는 접합을 의미하고 *는 반복을 의미한다. 물론 튜플은 변경이 불가능하므로 연산의 결과는 새로운 튜플이 된다. 실제로 튜플은 시퀀스가 제공하는 모든 일반 연산을 사용할 수 있다.

파이썬 수식	결과	설명
`len((1, 2, 3))`	3	튜플의 길이
`(1, 2, 3) + (4, 5, 6)`	(1, 2, 3, 4, 5, 6)	접합
`('Hi!',) * 4`	('Hi!', 'Hi!', 'Hi!', 'Hi!')	반복
`3 in (1, 2, 3)`	True	멤버쉽
`for x in (1, 2, 3): print x,`	1 2 3	반복

3. 튜플

3) 인덱싱, 슬라이싱, 매트릭스

- 튜플도 시퀀스의 일종이기 때문에 인덱싱과 슬라이싱은 문자열이나 리스트와 동일하게 동작한다. 다음과 같은 튜플을 가정하자.

```
t = ('apple', 'banana', 'strawberry')
```

- 인덱싱과 슬라이싱의 결과는 다음과 같다.
  - t[1] – 'banana' (인덱스는 0부터 시작한다.)
  - t[-2]--'banana' (음수 인덱스는 오른쪽부터 왼쪽으로 진행된다.)
  - t[1:]-- ['banana', 'strawberry'] (슬라이싱은 튜플의 한 부분을 추출한다.)

4) 괄호가 없는 튜플

- 튜플은 (...)을 사용하여 감싸는 것이 원칙이다. 하지만 만약 괄호 없이 나열된 객체들은 기본적으로 튜플로 간주된다.

```
t1 = 'physics', 'chemistry', 'c language'
t2 = 1, 2, 3, 4, 5
t3 = "a", "b", "c", "d"
```

3. 튜플

5) 튜플 내장 함수

- 튜플은 다음과 같은 내장 함수를 지원하고 시퀀스가 제공하는 모든 함수는 사용이 가능하다.

함수	설명
len(t)	튜플의 길이를 반환한다.
max(t)	튜플에 저장된 최대값을 반환한다.
min(t)	튜플에 저장된 최소값을 반환한다.
tuple(seq)	리스트를 튜플로 변환한다.

6) 튜플 대입 연산

- 파이썬은 튜플 대입 연산(tuple assignment)이라는 기능을 가지고 있다. 이 기능은 튜플에서 여러 개의 변수로 한 번에 값을 대입하는 강력한 기능이다.

```
student1 = ("철수", 19, "CS")
(name, age, major) = student1
print(name)
print(age)
print(major)
```

출력 결과
"철수"
19
"CS"

3. 튜플

6) 튜플 대입 연산

- 튜플에 값을 저장하는 과정을 튜플 패킹(tuple packing)이라고도 한다. 반대로 튜플에서 값을 꺼내서 변수에 대입하는 과정을 튜플 언패킹(tuple unpacking)이라고도 생각할 수 있다. 튜플 대입 연산을 가장 효과적으로 사용하는 예가 변수의 값을 교환하는 경우이다.

  일반적인 프로그래밍 언어에서 변수 x와 변수 y의 값을 교환하려면 다음과 같은 문장을 작성하여야 한다.

```
temp = x
x = y
y = temp
```

- 하지만 튜플에서는 다음과 같이 튜플 대입 연산을 사용하여 한 문장으로 작성하는 것이 가능하다.

```
(x, y) = (y, x)
```

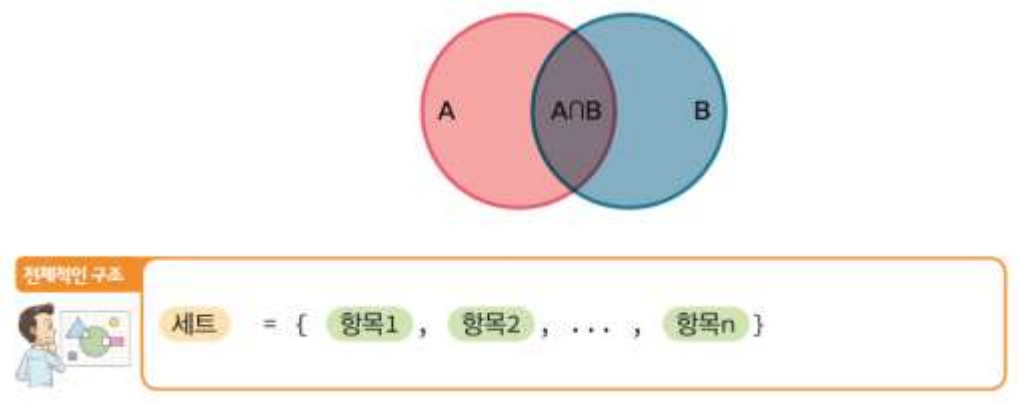
- 왼쪽은 변수들이 모인 튜플이고 오른쪽은 값들이 모인 튜플이 된다. 값은 해당되는 변수에 대입된다. 대입되기 전에 오른쪽의 수식들이 먼저 계산되므로 혼동이 발생할 소지는 없다. 단 이 경우에 변수의 개수와 값의 개수는 일치하여야 한다.

```
(x, y, z) = (1, 2)
...
ValueError: not enough values to unpack (expected 3, got 2)
```

4. 세트(Set)

1) 세트(set)

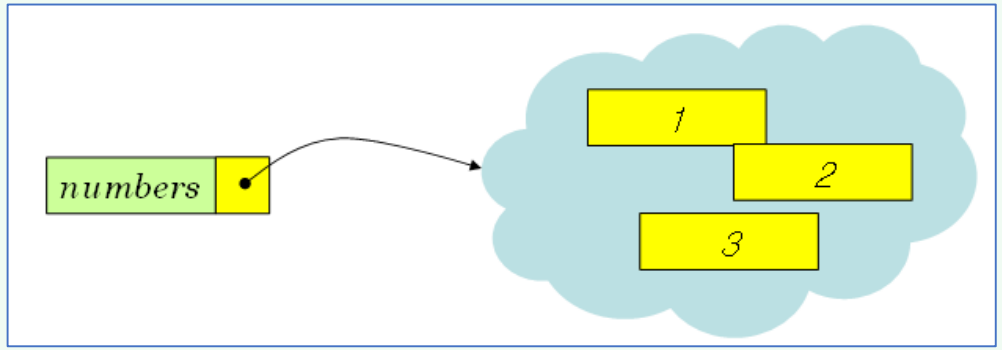
- 세트(set)는 우리가 수학에서 배웠던 집합이다. 세트는 중복되지 않은 항목들이 모인 것이다. 세트의 항목 간에는 순서가 없다. 만약 응용 프로그램에서 순서 없는 항목들의 집합을 원한다면 세트가 최선의 선택이 된다. 하지만 중복된 항목은 없어야 한다. 파이썬에서 세트를 생성하려면 요소들을 중괄호 기호 {...}로 감싸면 된다.



중복을 불허하는 특징 때문에 프로그래밍에서 매우 유용하다. 대표적으로 문서 하나에 들어가 있는 단어 종류의 개수를 셀 때 모든 단어를 추출한 후 세트로 변환하면, 단어 종류의 개수를 쉽게 파악할 수 있다.

- 세트는 중괄호 기호 안에 항목들을 쉼표로 분리하여 놓으면 된다.

```
numbers = {2, 1, 3}
print(numbers)
출력결과 : {1, 2, 3}
```



- 세트의 크기는 len() 함수로 알 수 있다

```
len(numbers)
출력결과 : 3
```

4. 세트(Set)

1) 세트(set)

- 세트를 만들 때, 문자열로 이루어진 세트도 생성할 수 있으며 여러 가지 자료형을 섞어도 된다.

```
fruits = { "Apple", "Banana", "Pineapple" }
mySet = { 1.0, 2.0, "Hello World", (1, 2, 3) }
```

- 세트는 집합이기 때문에 요소가 중복되면 자동으로 중복된 요소를 제거한다.

```
cities = { "Paris", "Seoul", "London", "Berlin", "Paris", "Seoul" }
print(cities)
출력결과 : {'Seoul', 'London', 'Berlin', 'Paris'}
```

- 비어 있는 세트를 생성하려면 set() 함수를 사용한다.

```
numbers = set()
```

- 어떤 항목이 세트 안에 있는지를 검사하려면 in 연산자를 사용하면 된다.

```
numbers = {2, 1, 3}
if 1 in numbers :
        print("집합 안에 1이 있습니다.")
출력결과 : 집합 안에 1이 있습니다.
```

4. 세트(Set)

1) 세트(set)

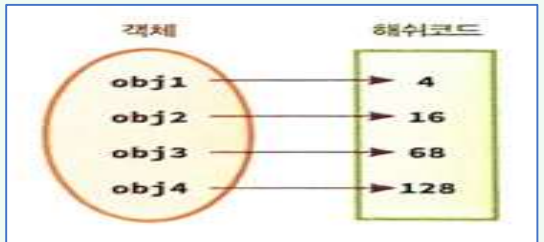
- 세트의 항목은 순서가 없기 때문에 위치를 가지고 세트의 항목에 접근할 수는 없다. 하지만 for 반복문을 이용하여 각 항목들에 접근할 수 있다.

```
numbers = {2, 1, 3}
for min numbers:
        print(x, end=" ")
출력결과 : 1 2 3
```

- 여기서 주의할 점은 항목들이 출력되는 순서는 입력된 순서와 다를 수도 있다는 점이다. 우리의 예제에서도 입력된 순서는 2, 1, 3이지만 출력되는 순서는 1, 2, 3이다.
- 만약 정렬된 순서로 항목을 출력하기를 원한다면 다음과 같이 sorted() 함수를 사용하면 된다.

```
for x in sorted (numbers):
        print(x, end=" ")
출력결과 : 1 2 3
```

- 세트는 모든 요소들을 해싱(hashing)을 이용하여 저장하고 관리한다. 따라서 요소들은 해싱가능(hashable)하여야 한다. 해싱이란 아주 간단히 설명하자면 각각의 객체에 식별할 수 있는 숫자 코드를 부여하여 객체를 테이블에 저장하는 것이다.



4. 세트(Set)

1) 세트(set)

- 파이썬에서 요소가 해싱 가능하려면 해쉬 코드를 가져야 하고 그 값이 변경되면 안된다. 따라서 세트는 변경 가능한 항목을 가지면 안 된다. 예를 들어서 세트 안에 리스트를 넣으면 안 된다.

```
numbers = {1, 2, [3, 4, 5]}
...
TypeError : unhashable type: list'
```

- 하지만 리스트로부터 세트를 생성하는 것은 가능하다.

```
print(set([1, 2, 3, 1, 2, 3]))
출력결과 : {1, 2, 3}        # 중복 제거
```

- 문자열로부터 세트를 생성하는 것도 가능하다.

```
print(set("abcdefa"))
출력결과 : {'f', 'a', 'b', 'e', 'c', 'd'}   # 중복 제거, 출력결과도 매번 다를 수 있다.
```

- 집합의 요소에 대하여 반복하려면 다음과 같은 문장을 사용하면 된다.

```
for char in set ("banana"):
        print(char, end = " ")
출력결과 : a b c
```

4. 세트(Set)

2) 요소 추가하고 삭제하기

- 세트는 변경 가능한 객체이다. 따라서 세트에 요소를 추가하거나 삭제할 수 있다. 그러나 세트의 요소에는 인덱스가 없기 때문에 인덱싱이나 슬라이싱 연산은 의미가 없다.

```
numbers = { 2, 1, 3 }
print(numbers[0])
TypeError: 'set' object does not support indexing
```

- add() 메소드를 이용하여서 하나의 요소를 추가할 수 있다. 예를 들어서 다음과 같은 문장이 가능하다.

```
numbers.add(4)
print(numbers)
출력결과 : {1, 2, 3, 4}
```

- 여러 개의 요소는 update() 메소드로 추가할 수 있다. 물론 중복된 요소는 추가되지 않는다.

```
numbers.update([2, 3, 4, 5])
print(numbers)
출력결과 : {1, 2, 3, 4, 5}
```

- 요소를 삭제할 때는 discard() 메소드를 사용할 수 있다.

```
numbers.discard(5)
print(numbers)
출력결과 : {1, 2, 3, 4}
```

4. 세트(Set)

2) 요소 추가하고 삭제하기

- remove() 메소드도 사용할 수 있다. 만약 세트에 없는 요소를 삭제하려고 하면 remove()는 예외를 발생시킨다.

```
numbers.remove(6)          # 세트에 6이 없으므로 예외가 발생된다.
KeyError: 6
```

- clear() 메소드는 세트의 전체 요소를 지운다.

```
numbers.clear()            # 세트의 크기가 0이 된다.
print(len(numbers))
출력결과 : 0
```

4. 세트(Set)

3) 부분 집합 연산

- 2개의 세트가 같은지도 검사할 수 있다. 이것은 ==와 =! 연산자를 사용하는 것이 가장 쉽다.

```
A = {1, 2, 3}
B = {1, 2, 3}
print(A == B)
출력결과 : True
```

- < 연산자와 <= 연산자를 사용하면 세트가 진부분 집합인지, 부분 집합인지를 검사할 수 있다. >와 >= 연산자를 사용하면 진상위 집합(A ⊃ B이고 A ≠ B인 경우 진상위 집합이라고 한다.즉 상위 집합이지만 두 집합이 같지 않다.) 상위 집합도 검사할 수 있다.

```
A = {1, 2, 3, 4, 5}
B = {1, 2, 3}
print(B < A)
출력결과 : True
```

- 부분집합인지를 검사하는 메소드는 issubset()이다.

```
A = {1, 2, 3, 4, 5}
B = {1, 2, 3}
print(B.issubset(A))
출력결과 : True
```

4. 세트(Set)

3) 부분 집합 연산

- 상위집합인지를 검사하는 메소드는 issuperset()이다.

```
A = {1, 2, 3, 4, 5}
B = {1, 2, 3}
print(A.issuperset(B))
출력결과 : True
```

- 요소가 집합에 포함되어 있는지는 in 키워드를 이용하여 검사할 수 있다.

```
mySet = set ("banana")
print('a' in mySet)
출력결과 : True
print('p' not in mySet)
출력결과 : True
```

4) 집합 연산

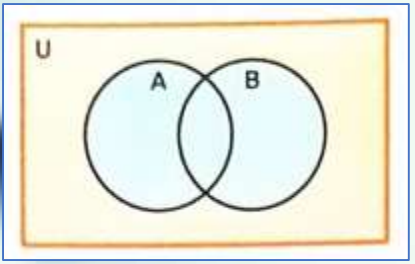
- 세트가 유용한 이유는 교집합이나 합집합과 같은 여러 가지 집합 연산을 지원하기 때문이다. 이것은 연산자나 메소드로 수행할 수 있다. 일단 다음과 같은 2개의 집합이 세트로 정의되어 있다고 가정하자.

```
A = {1, 2, 3}
B = {3, 4, 5}
```

4. 세트(Set)

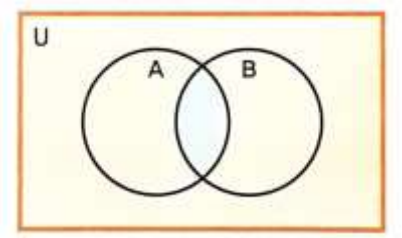
4) 집합 연산

- 합집합은 2개의 집합을 합하는 연산이다. 물론 중복되는 요소는 제외된다. 합집합은 | 연산자나 union() 메소드를 사용한다.




```
print(A | B)
출력결과 : {1, 2, 3, 4, 5}
print(A.union(B))
출력결과 : {1, 2, 3, 4, 5}
print(B.union(A))
출력결과 : {1, 2, 3, 4, 5}
```

- 교집합은 2개의 집합에서 겹치는 요소를 구하는 연산이다. 교집합은 & 연산자나 intersection() 메소드를 사용한다.

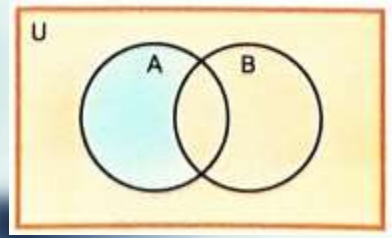



```
print(A & B)
출력결과 : {3}
A.intersection(B)
출력결과 : {3}
```

4. 세트(Set)

4) 집합 연산

- 차집합은 하나의 집합에서 다른 집합의 요소를 빼는 것이다. 차집합은 – 연산자나 difference() 메소드를 사용한다.




```
print(A – B)
출력결과 : {1, 2}
print(A.difference(B))
출력결과 : {1, 2}
```

- 집합에 대해서도 all(), any(), enumerate(), len(), max(), min(), sorted(), sum() 등의 메소드는 사용할 수 있다.

  all()은 세트의 모든 요소가 True인 경우에 세트가 True가 된다. any()는 하나의 요소라도 True이면 True를 반환한다.


감사합니다.