Python 문법

처리 시간

정수형

실수형

/ 나누기

% 나머지

// 몫

리스트 자료형 []

인덱싱 음수 가능

슬라이싱

컴프리헨션

문자열 자료형

문자열 연산

인덱싱

슬라이싱

변경 불가능

튜플 자료형

- 리스트와 유사하지만 **한 번 선언되면 값 변경 불가능**
- 리스트는 []을 이용하지만, 튜플은 소괄호 ()를 이용한다.
- 리스트에 비해 상대적으로 **공간 효율적**이다.

장점

- **서로 다른 성질**의 데이터를 묶어서 관리해야 할 때

    - 최단 경로 알고리즘에서는 (비용, 노드 번호)의 형태로 튜플 자료형을 자주 사용한다.
- 데이터의 나열을 **해싱(Hashing)의 키 값**으로 사용해야 할 때
    - 튜플은 변경이 불가능하므로 리스트와 다르게 키 값으로 사용될 수 있다.
- 리스트보다 **메모리를 효율적**으로 사용해야 할 때

사전 자료형

- **키(Key)와 값(Value)의 쌍으로 데이터를 가지는 자료형**
    - 리스트나 튜플이 값을 순차적으로 저장하는 것과는 대비된다.
- 원하면 **'변경 불가능한(Immutable) 자료형'을 키로 사용**할 수 있다.
- 해시 테이블(Hash Table)을 이용하므로 **데이터의 조회 및 수정에 있어서 O(1)의 시간에 처리**할 수 있다.

집합 자료형

- 특징
    - 중복을 허용하지 않는다.
    - 순서가 없다.
- 집합은 **리스트 혹은 문자열을 이용하여 초기화** 할 수 있다.
    - set() 함수 이용
- 혹은 **중괄호({})** 안에 각 원소를 콤마(,)를 기준으로 구분하여 삽입함으로써 초기화 할 수 있다.
- **데이터의 조회 및 수정에 있어서 O(1)의 시간에 처리**할 수 있다.
- 집합 자료형의 연산

합집합	|
교집합	&
차집합	-

사전 자료형과 집합 자료형의 특징

- 리스트나 튜플은 순서가 있기 때문에 인덱싱을 통해 자료형의 값을 얻을 수 있다.
- 사전 자료형과 집합 자료형은 **순서가 없기 때문에** 인덱싱으로 값을 얻을 수 없다.
    - 사전의 키(Key) 혹은 집합의 원소(Element)를 이용해 **O(1)**의 시간 복잡도로 조회한다.

기본 입출력

- input() 한 줄의 문자열을 입력 받는 함수
- map() 리스트의 모든 원소에 각각 특정한 함수를 적용할 때 사용

```
n = int(input())
data = list(map(int, input().split()))
print(data)
```

빠르게 입력 받기

- 사용자로부터 **입력을 최대한 빠르게 받아야하는 경우**가 있다.
- sys 라이브러리에 정의되어 있는 sys.stdin.readlint() 메서드를 이요한다.
    - 단, 입력 후 엔터(Enter)가 줄 바꿈 기호로 입력되므로 이를 제거하기 위해 rstrip() 메서드를 함께 사용한다.

```
import sys
data = sys.stdin.readlint().rstrip()
print(data)
```

표준 출력 방식

- 기본 출력은 print() 함수를 이용
    - 각 변수를 콤마(,)를 이용하여 띄어쓰기로 구분하여 출력할 수 있다.
- print()는 **기본적으로 출력 이후에 줄 바꿈을 수행**한다.
    - 줄 바꿈을 원치 않는 경우 ‘end’ 속성을 이용할 수 있다.

```
a = 1
b = 2
```

```
print(a, b) # 자동 줄 바꿈
print(7, end=" ")
print(8, end=" ")
answer = 7
print("정답은 " + str(answer) + "입니다.")

'''
[실행 결과]
1 2
7 8 정답은 7입니다.
'''
```

f-string 예제

- 문자열 앞에 접두사 **‘f’**를 붙여 사용
- 중괄호 안에 변수명을 기입하여 간단히 문자열과 정수를 함께 넣을 수 있다.

```
answer = 7
print(f"정답은 {answer}입니다.")

'''
[실행 결과]
정답은 7입니다.
'''
```

들여쓰기

- 파이썬에서는 **코드의 블록(Block)을 들여쓰기(Input)로 지정**한다.
- 탭을 사용하는 쪽과 공백 문자(Space)를 여러 번 사용하는 쪽으로 두 진영이 있다.
  - 이에 대한 논쟁은 지금까지도 활발하다.
- 파이썬 스타일 가이드라인에서는 **4개의 공백 문자를 사용하는 것을 표준으로 설정**하고 있다.

조건문

- 기본적인 형태는 **if ~ elif ~ else**이다.
  - 조건문을 사용할 때 elif 혹은 else 부분은 경우에 따라서 사용하지 않아도 된다.

비교 연산자

- **특정한 두 값을 비교할 때 이용**할 수 있다.
  - 대입 연산자(=)와 같음 연산자(==)의 차이점에 유의하자

비교 연산자	설명
X == Y	X와 Y가 서로 같을 때 참(True)이다.
X != Y	X와 Y가 서로 다를 때 참(True)이다.
X > Y	X가 Y보다 클 때 참(True)이다.
X < Y	X가 Y보다 작을 때 참(True)이다.
X >= Y	X가 Y보다 크거나 같을 때 참(True)이다.
X <= Y	X가 Y보다 작거나 같을 때 참(True)이다.

논리 연산자

- 논리 값 (True/False) 사이의 연산을 수행 할 때 사용한다.

논리 연산자	설명
X and Y	X와 Y가 모두 참(True)일 때 참(True)이다.
X or Y	X와 Y중에 하나만 참(True)이어도 참(True)이다.
not X	X가 거짓(False)일 때 참(True)이다.

파이썬의 기타 연산자

- 다수의 데이터를 담는 자료형을 위해 **in 연산자**와 **not in 연산자**가 제공된다.
  - 리스트, 튜플, 문자열, 딕셔너리 모두에서 사용 가능하다.

in 연산자와 not in 연산자	설명
x in 리스트	리스트 안에 x가 들어가 있을 때 참(True)이다.
x not in 문자열	문자열 안에 x가 들어가 있지 않을 때 참(True)이다.

파이썬의 pass 키워드

- 아무것도 처리하고 싶지 않을 때 pass 키워드를 사용한다.

- 디버깅 과정에서 일단 조건문의 형태만 만들어 놓고 조건문을 처리하는 부분을 비워놓고 싶은 경우

```
score = 85
if score >= 80:
    pass #나중에 작성할 소스코드
else:
    print('성적이 80점 미만입니다.')

print('프로그램을 종료합니다.')

'''
[실행 결과]
프로그램을 종료합니다.
'''
```

조건문의 간소화

- 5줄

```
score = 85
if score >= 80:
    result = "Success"
else:
    result = "Fail"
print(result)
```

- 4줄

```
score = 85
if score >= 80: result = "Success"
else: result = "Fail"
print(result)
```

- 3줄

```
score = 85
result = "Success" if score >= 80 else "Fail"
print(result)
```

파이썬 조건문 내에서의 부등식

- 다른 언어와 다르게 파이썬은 조건문 안에서 수학의 부등식을 그대로 사용할 수 있다.
  - x > 0 and x < 20 와 0 < x < 20 둘 다 사용 가능

반복문

- for / while
  - for문이 더 간결한 경우가 많다.
- while

```
i = 1
result = 0
while i <= 9:
    if i % 2 == 1: result += i
    i += 1
print(result)

'''
[실행결과]
25
'''
```

- 무한루프
  - 코딩 테스트에서 무한 루프를 구현할 일은 거의 없으니 유의한다.
  - 반복문을 작성한 뒤에는 항상 반복문을 탈출할 수 있는지 확인한다.

```
x = 10
while x > 5: print(x)

'''
[실행결과]
10
10
10
10
10
...
'''
```

- for 문
  - 특정한 변수를 이용하여 'in' 뒤에 오는 **데이터(리스트, 튜플 등)에 포함되어 있는 원소를 첫 번째 인덱스부터 차례대로 하나씩 방문**한다.

```
array = [9, 8, 7, 6, 5]
for x in array: print(x)

'''
[실행결과]
9
8
7
6
5
'''
```

- 연속적인 값을 차례대로 순회할 때에는 **range()**를 주로 사용한다.
  - range(시작 값, 끝 값+1) 형태로 사용한다.
  - 인자를 하나만 넣으면 자동으로 시작 값은 0이 된다.

```
result = 0
for i in range(1, 10): result += i
print(result)

'''
[실행결과]
45
'''
```

파이썬의 continue 키워드

- 반복문에서 남은 코드의 실행을 건너뛰고, 다음 반복을 진행하고자 할 때 **continue**를 사용한다.

```
result = 0
for i in range(1, 10):
    if i % 2 == 0: continue
    result += i
print(result)

'''
[실행결과]
25
'''
```

파이썬의 break 키워드

- 반복문을 즉시 탈출하고자 할 때 **break**를 사용한다.

```
i = 1
while True: print("현재 i의 값:", i)
    if i == 5: break
    i += 1

'''
[실행결과]
현재 i의 값: 1
현재 i의 값: 2
현재 i의 값: 3
현재 i의 값: 4
현재 i의 값: 5
'''
```

예제

- 점수가 80점만 넘으면 합격

```
scores = [90, 85, 77, 65, 97]
for i in range(5):
    if scores[i] >= 80: print(i+1, "번 학생은 합격입니다.")

'''
[실행결과]
1 번 학생은 합격입니다.
2 번 학생은 합격입니다.
5 번 학생은 합격입니다.
'''
```

- 특정 번호의 학생은 제외하기

```
scores = [90, 85, 77, 65, 97]
cheating_student_list = {2, 4}
for i in range(5):
    if i+1 in cheating_student_list: continue
    if scores[i] >= 80: print(i+1, "번 학생은 합격입니다.")

'''
[실행결과]
1 번 학생은 합격입니다.
5 번 학생은 합격입니다.
'''
```

- 구구단

```
for i in range(2, 10):
    for j in range(1, 10): print(i, "X", j, "=", i*j)
    print()

'''
[실행결과]
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
2 X 4 = 8
2 X 5 = 10
2 X 6 = 12
2 X 7 = 14
2 X 8 = 16
2 X 9 = 18

3 X 1 = 3
3 X 2 = 6
...
'''
```

함수

- 함수(Function)란 특정한 작업을 하나의 단위로 묶어 놓은 것을 의미한다.
- 함수를 사용하면 불필요한 소스코드의 반복을 줄일 수 있다.
- 종류
  - 내장 함수: 파이썬이 기본으로 제공하는 함수
  - 사용자 정의 함수: 개발자가 직접 정의하여 사용할 수 있는 함수

함수 정의하기

- 매개변수: 함수 내부에서 사용할 변수
- 반환 값: 함수에서 처리 된 결과를 반환

```
def 함수명(매개변수):
    실행할 소스코드
    return 반환 값
```

파라미터 지정하기

- 파라미터의 변수를 직접 지정할 수 있다.
    - 이 경우 매개변수의 순서가 달라도 상관 없다.

```
def add(a, b): print('함수의 결과:', a + b)
add(b = 3, a = 7)

'''
[실행결과]
함수의 결과: 10
'''
```

global 키워드

- global 키워드로 변수를 지정하면 해당 함수에서는 지역 변수를 만들지 않고, **함수 바깥에 선언된 변수를 바로 참조**하게 된다.

```
a = 0
def func():
    global a
    a += 1
for i in range(10): func()
print(a)

'''
[실행결과]
10
'''
```

    - 예외로 배열은 함수 내에서 global 키워드 없이도 전역변수인 배열을 가져와 사용 가능하다.

여러개의 반환 값

람다 표현식 → 이름 없는 함수

- 함수를 간단하게 작성할 수 있다.
    - 특정 기능을 수행하는 함수를 한 줄에 작성할 수 있다는 점이 특징이다.

- 내장 함수에서 자주 사용되는 예시

```
array = [('홍길동', 50), ('이순신', 32), ('아무개', 74)]

def my_key(x): return x[1]
print(sorted(array, key=my_key))

'''
[실행결과]
[('이순신', 32), ('홍길동', 50), ('아무개', 74)]
'''
```

```
array = [('홍길동', 50), ('이순신', 32), ('아무개', 74)]

print(sorted(array, key=lambda x: x[1])

'''
[실행결과]
[('이순신', 32), ('홍길동', 50), ('아무개', 74)]
'''
```

- 여러 개의 리스트에 적용

```
list1 = [1, 2, 3, 4, 5]
list2 = [6, 7, 8, 9, 10]

result = map(lambda a, b: a + b, list1, list2)

print(list(result))

'''
[실행결과]
[7, 9, 11, 13, 15]
'''
```

실전에서 유용한 표준 라이브러리

- 내장 함수: 기본 입출력 함수부터 정렬 함수까지
- itertools
- heapq
- bisect
- collections

- math