문제 풀 때 알아두면 유용한 파이썬3 Idiom과 Tip

1. 코드의 구성

- 모든 테스트케이스를 입력받은 다음에 처리하는 형태가 아니라 각 테스트케이스를 입력받고 출력하는 형태로 해결해야 한다. 모든 테스트케이스를 다 입력받아 유지하면 메모리 공간이 너무 많이 필요하고, 코딩하기도 불편하다.
- 여러 함수를 작성할 때 전역변수와 지역변수 간의 충돌 등 문제가 발생할 수 있으며, 파이썬은 전역변수보다 지역변수의 처리가 효과적이기 때문에 다음과 같이 **main** 함수를 만들어 사용하는 것이 효과적이다.

```
from sys import stdin

def main():
    T = int(stdin.readline())
    for _ in range(T):
        # read testcase
        # solve testcase
        # print solution

main()
```

이와 같이 작성하면 혹시 이 모듈을 **import**하면 모듈 수준의 코드인 **main()**이 바로 실행될 수 있는 문제점이 있다. 이 때문에 위와 같이 하지 않고 다음과 같이 하는 경우도 많다. 하지만 judge 사이트에 제출할 때는 아래와 같은 형태를 꼭 사용할 필요는 없다.

```
if __name__ == '__main__':
    main()
```

2. 입력의 처리

- Python은 줄단위로 대부분 입력을 처리하기 때문에 이 부분에 대해 특별히 다르게 처리할 필요는 없다.
- Python은 입력을 조금 더 빠르게 처리하기 위해 **sys.stdin.readline()**을 사용할 수 있다. 이때 결과는 문자열이며, 끝에 줄바꿈 문자가 포함되어 있다. 줄바꿈 문자를 제거할 필요가 있으면 **rstrip**을 추가로 사용할 수 있다. 하지만 공백단위로 나누어 데이터를 처리하거나 한 줄에 정수 하나만 있는 경우에는 **rstrip**을 사용할 필요는 없다.
- 입력을 처리하기 위해 BIF(Build-In-Function)인 **eval**을 사용하는 경우가 있지만 타입에 따라 **int**나 **float**를 사용하는 것이 바람직하다.
  T = eval(stdin.readline()) $\Longrightarrow$ **T = int(stdin.readline())**
- Python은 줄단위로 입력을 처리하기 때문에 다른 언어와 달리 데이터의 개수를 주는 정보 등을 활용할 필요가 없다.

```
N =  int(stdin.readline())
nums = list(map(int, stdin.readline().split()));
```

위 예에서 **nums**을 만들기 위해 **N**을 사용할 필요가 없다.

- **split**를 사용할 때 **split('')**을 사용하지 말고 **split()**를 사용해야 한다. **split('')**을 사용하면 줄바꿈 문자와 공백 수에 따라 기대한 것과 다르게 분리될 수 있다.

3. 출력의 처리

- 원칙은 출력의 공백의 수, 줄바꿈 위치가 중요합니다. 현재 judge는 정확하게 일치하지 않아도 통과되지만 다른 사이트는 출력 형태가 정확하게 요구사항을 만족하지 않으면 답이 틀린 것으로 판단할 수 있다. 배열에 있는 일련의 정수를 출력해야 하면 다음과 같이 출력하는 것이 가장 올바른 방법이다.

```
print(' '.join(map(str, nums)))
```

4. 입력 데이터의 표현

- 문제에 따라 입력 데이터가 단순 숫자가 아니라 어떤 의미를 지닐 수 있다. Python은 클래스를 정의하여 사용하는 것이 코드 가독성 측면에서 효과적일 수 있다.
- 예를 들어 x, y 두 개의 값으로 구성된 평면 좌표 데이터를 처리해야 하는 문제이면 Python은 간단하게 튜플을 통해 나타낼 수 있지만 Python도 클래스를 정의하여 사용하면 가독성에 좋아진다.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

for _ in range(T):
    N = int(stdin.readline())
    pointInfo = list(map(int, stdin.readline().split()))
    points = []
    for i in range(0, len(pointInfo), 2):
        points.append(Point(pointInfo[i], pointInfo[i+1])
    # solve testcase
    # print solution
```

위 예에서 **zip**을 활용하면 다음과 같이 더 간결하게 처리할 수 있다.

```
points = [Point(x, y) for x, y in zip(pointInfo[0::2], pointInfo[1::2])]
```

- $m \times n$ 2차원 배열 만들기

```
matrix = [[0]*n for _ in range(m)]
matrix = [[0 for _in range(n)] for _ in range(m)]
```

후자는 동일 값으로 모든 요소를 초기화해야 할 때보다 식을 이용하여 초기화해야 할 때 많이 사용한다.

```
matrix = [[i+1 for i in range(n)] for _ in range(m)]
```

5. 최댓값, 최솟값

- 방법 1. **math.inf**을 사용함

```
import math

maxvalue = math.inf
```

- 방법 2. **float('inf')**를 사용함

```
maxvalue = float('inf')
```

6. 데이터를 정렬해야 하는 경우

- 정렬은 $O(n \log n)$에 할 수 있기 때문에 구현하는 알고리즘 $O(n \log n)$보다 느린 알고리즘이고 정렬이 문제 해결에 도움이 되면 필요한 만큼 활용할 필요가 있다.
- 정렬할 때 주의할 점은 구조체 데이터를 정렬할 경우 첫 번째 기준뿐만 아니라 첫 번째 기준 값이 같을 경우 어떻게 정렬해야 하는지 추가 고민이 필요하다. 예를 들어 좌표를 정렬하는데, x 좌표를 기준으로 정렬하였을 때 x 좌표가 같은 경우 어떤 순서로 정렬해야 하는지 추가로 지정하지 않으면 테스트데이터를 생성한 기준과 다른 기준으로 정렬하여 답이 틀릴 수 있다.

```
points.sort(key=lambda p: (p.x, p.y))
```

7. 알아두면 유용한 idiom

- swap

```
x, y = y, x
```

- 조건부 연산자

```
print('true' if x>y else 'false')
```

8. 자료구조

8.1 스택

- 방법 1. **list**를 이용하는 방법. 시간복잡도. amortized $O(1)$

```
stack = []
stack.append(3)
stack.append(5)
print(stack[-1])    # peek
print(stack.pop())
print(stack.pop())
```

- 방법 2. **deque**를 이용하는 방법. 시간복잡도. $O(1)$. 내부적으로 이중 연결구조

```
from collections import deque

stack = deque()
stack.append(3)
stack.append(5)
print(stack[-1])    # peek
print(stack.pop())
print(stack.pop())
```

- 방법 3. **LifoQueue**를 이용하는 방법. 동기화 기능이 포함되어 있기 때문에 일반적인 경우에는 이 방법을 사용할 필요가 없다.

```
from queue import LifoQueue

stack = LifoQueue()
stack.put(3)
stack.put(5)
print(stack.queue[-1])      # peek
print(stack.get())
print(stack.get())
```

8.2 FIFO 큐

- 방법 1. **deque**를 이용하는 방법. 시간복잡도. $O(1)$. 내부적으로 이중 연결구조

```
from collections import deque

Q = deque()
Q.append(3)
Q.append(5)
print(Q[0])     # peek
print(Q.popleft())
print(Q.popleft())
```

- 방법 2. **Queue**를 이용하는 방법. 동기화 기능이 포함되어 있기 때문에 일반적인 경우에는 이 방법을 사용할 필요가 없다.

```
from queue import Queue

Q = Queue()
Q.put(3)
Q.put(5)
print(Q.queue[0])   # peek
print(Q.get())
print(Q.get())
```

8.3 우선순위 큐

- 방법 1. **heapq**를 이용하는 방법.

```
import heapq

heap = []
heapq.heappush(heap, 'apple')
heapq.heappush(heap, 'banana')
print(heap[0])  # peek
while heap:
    print(heapq.heappop(heap))
```

비교자는 튜플의 첫 번째 요소를 이용한다. 따라서 **(key, data)** 형태로 push하는 방법을 이용하거나 클래스를 정의하고 **__lt__** 메소드를 정의하면 된다.

- 방법 2. **PriorityQueue**를 이용하는 방법. 동기화 기능이 포함되어 있기 때문에 일반적인 경우에는 이 방법을 사용할 필요가 없다.

```
from queue import PriorityQueue

Q = PriorityQueue()
Q.put(3)
Q.put(5)
print(Q.queue[0])   # peek
while not Q.empty():
    print(Q.get())
```

8.4 집합

- 예제)

```
S = set()
S.add(5)
S.add(3)
if 3 in S: print('3 is in the set')
S.remove(3)
S.clear()
```

- **set()**은 내부적으로 해시 테이블을 이용한다. 다른 언어와 달리 초기 해시 테이블 용량을 지정하는 방법이 없다.
- **add**, **remove**, **in** 연산의 비용은 모두 평균 $O(1)$이다.
- 제공 연산: **isdisjoint**, **issubset**, **issuperset**, **union**, **intersection**, **difference**, **symmetric_difference**, **update**, ...
- **issubset** 대신에 두 집합 간 비교 연산자를 통해 부분 집합 여부를 확인할 수 있다.
- 한 변수가 특정 값 중 하나인지 확인하고 싶으면 여러 개 조건문을 결합하여 사용하지 않고 다음과 같이 집합을 이용할 수 있고, **any**를 이용할 수 있다.

```
if a == 0 || a == 2 || a == 5:
if a in {0, 2, 5}:
if any((a == 0, a == 2, a == 5))
```

8.5 맵

- 예제)

```
map = {} # map = dict()
map['apple'] = 5
map['banana'] = 3
if 'apple' in map: print(map.get('apple'))
if 'banana' in map: del map['banana']
```

- **dict()**은 내부적으로 해시 테이블을 이용한다. 다른 언어와 달리 초기 해시 테이블 용량을 지정하는 방법이 없다.
- **map['banana']**와 **map.get('banana')**의 차이점: 전자는 없을 경우 **KeyError** 예외를 발생하지만 후자는 **None**을 반환한다. 더욱이 **get** 메소드는 두 번째 인자를 통해 없을 경우 **None** 대신에 다른 값을 반환하도록 할 수 있다. 다음은 주어진 문자열의 문자 빈도수를 맵을 통해 구하는 코드이다.

```
freq = {}
for c in S:
    freq[c] = freq.get(c, 0)+1
```

- **KeyError** 문제는 **collections.defaultdict**을 이용할 수 있다.

```
freq = collections.defaultdict(int)
for c in word:
    freq[c] += 1
```

  문자열에서 문자 등장 빈도를 알고 싶으면 **Counter**를 이용하는 것이 더 간편하다.

```
freq = collections.Counter(word)
```

- 제공 연산: **keys()**, **pop(key)**, **popitem()**, **values()**, ...
- 파이썬은 **switch** 문이 없다. 많은 경우 **switch** 문이 필요할 때 맵을 활용할 수 있다.

```
if operator = '+': return left + right
elif operator = '-': return left - right
elif operator = '*': return left * right
else: return left / right
```

위와 같은 다중 if-else 문은 다음과 같이 맵을 이용할 수 있다.

```
return {'+': left + right, '-': left - right,
     '*': left * right, '/': keft / right}.get(operator)
```

9. 프로그래밍 Tip

- 파이썬에서 모든 변수는 참조 변수이다. 심지어 원시 타입도 모두 참조 변수이다. 또 불변 타입과 그렇지 않은 타입을 구분하고 용도에 맞게 사용해야 한다.
- slicing은 꼭 필요한 경우에만 사용한다. 리스트를 slicing하면 새로운 리스트가 만들어지므로 생성 비용도 소요되며, 공간 복잡도도 증가한다.
- 파이썬은 연쇄 비교 연산을 지원한다.

```
if index >= 0 and index < size:
if a == 0 and b == 0 and c==0:
```

와 같은 비교는 파이썬에서는 다음과 같이 작성할 수 있다.

```
if 0 <= index < size:
if a == b == c == 0:
```

- Python에서 난수 발생
  - **random()**보다 **randint(a, b)**가 훨씬 느리다. 따라서 **randInt(a,b)** 대신에 다음을 사용하는 것이 더 효과적이다.

```
n = a+int((b-a+1)*random.random())
```

- 리스트는 반복문을 통해 **append**하여 구축하는 것보다 list comprehension을 이용하는 것이 효과적이다.

```
nums = []
for i in range(N):
    nums.append(i+1)
```

보다는 다음이 효과적이다.

```
nums = [i+1 for i in range(N)]
```

list comprehension 외에 set comprehension, dict comprehension도 단순 반복문보다 효과적이다. 또 list comprehension과 **all** 함수를 이용하여 다중 조건이 모두 **True**인지 검사할 수 있다. 참고로 빈 리스트를 **all**에 전달하면 결과는 **True**이다.

- 반복문을 작성할 때 꼭 알아두면 편리한 내장 함수는 **range**, **reversed**, **enumerate**이다.
  - **enumerate**: container를 반복할 때 색인이 포함된 튜플을 만들어 줌

```
for i, item in enumerate(items):
```

- dot 연산자를 사용하지 않기

```
import sys
T = int(sys.stdin.readline())
```

보다는 다음이 효과적이다.

```
from sys import stdin
T = int(stdin.readline())
```

- 문자열을 연속적으로 계속 결합해야 하면 **+** 연산자를 사용하지 말고, 결합해야 할 문자열을 리스트로 모은 후, 최종적으로 **join** 연산을 하는 것이 더 효과적이다.

```
def funcFilter(lines):
    for line in lines:
        if 'def' in line:
            yield line
```

와 같은 generator는 다음과 같이 간단하게 작성하여 사용할 수 있다.

```
funcFilter = (line for line in lines if 'def' in line)
```

참고. **yield**는 **return**처럼 함수를 종료해 주지만 이 함수가 다시 호출되면 이전 위치부터 실행된다.

10. 주의사항

- list comprehension 대신에 괄호를 사용하면 tuple을 만드는 것으로 착각할 수 있다. 실제는 generator를 만들어 주며, generator는 한번에 하나씩 처리해 준다.
- **is**와 == 연산자의 사용
  - 변수가 **None**인지 여부를 검사할 때는 반드시 **is**를 활용해야 한다.
  - == 연산자는 클래스가 **__eq__**를 통해 재정의할 수 있는 연산자이다.

이 문서에 오류가 있거나 추가되면 좋을 내용이 있으면 *sangjin@koreatech.ac.kr*로 연락주세요.