디지털 교재

파이썬으로 시작하는

자료구조와 알고리즘

<파이썬 사전지식>

NAVER CONNECT

파이썬 사전 지식

객체 (Object)

파이썬에서 객체는 **상태(특성)**와 **정의된 행위(메서드)**를 갖고 있는 모든 데이터를 의미합니다. 파이썬에서 존재하는 모든 데이터는 객체로 이루어져 있습니다. 따라서 앞으로 소개될 자료구조 혹은 데이터는 객체라고 간주하시면 됩니다.

Container vs Sequence

Container

컨테이너(Container)란 서로 다른 데이터에 대해서 조직하고, 수정할 수 있는 내부 구현된 데이터 구조를 의미합니다. 가장 대표적인 컨테이너로는 **리스트(list), 딕셔너리(dictionary), 튜플(tuple), 그리고 집합(set)**이 있습니다. 추가로 collections 모듈에서 더 다양한 컨테이너 데이터타입을 지원합니다. 각 데이터타입에 대한 설명은 3강 해시테이블에 있으니 참고 바랍니다.

(자주 사용하는) collections 모듈에서 지원하는 컨테이너 데이터타입

- deque
- OrderedDict
- DefaultDict
- Counter

Sequence

Sequence 객체는 데이터를 **나열**된 상태로 저장하고 있는 객체를 의미합니다. 대표적으로 **리스트(list), 문자열(string), 튜플(tuple)**이 존재합니다. **인덱싱(indexing), 슬라이싱(slicing)**이 가능하다는 것이 가장 큰 특징입니다.

Iterable vs Iterator

Iterable

Iterable은 한 번에 하나의 멤버[(1)]를 반환할 수 있는 객체, 즉 **반복(iterate)**할 수 있는 모든 객체를 일컫는 용어입니다. 리스트(list), 문자열(string), 튜플(tuple)과 같은 **sequence 타입**뿐만 아니라 딕셔너리(dictionary), 파일 객체(file object)와 같은 **non-sequence 타입들**도 iterable 객체에 속합니다. 임의로 직접 iterable 객체를 만들 수도 있습니다.[(2)]

Iterable 객체 주요 쓰임

1. for 문

```
l = [1, 2, 3, 4, 5]

for i in l:
    print(i)
```

▼ 결과

```
1
2
3
```

```
4
5
```

for문에서 iterable 객체를 사용하게 되면, 자동으로 임시 변수를 선언해서 iterator 객체에 iterable 객체를 인자로 넣어서 반복을 수행합니다.

2. 나열이 필요한 경우(sequence) : `zip()` 이 대표적

```
a = ['a', 'b', 'c', 'd', 'e']
b = [1, 2, 3, 4, 5]

# zip 이용 예시
for s, n in zip(a, b):
    print(s, n)
```

▼ 결과

```
a 1
b 2
c 3
d 4
e 5
```

`zip()` 메서드 작동 방식에 대해 궁금하신 분은 **Packing vs Unpacking** 부분을 참고하시면 됩니다. 그렇다면 만약 리스트 a와 b의 길이가 다르다면 어떨까요?

```
a = ['a', 'b', 'c', 'd', 'e']
b = [1, 2, 3]

# zip 이용 예시
for s, n in zip(a, b):
    print(s, n)
```

▼ 결과

```
a 1
b 2
c 3
```

더 짧은 길이의 iterable 객체의 길이만큼만 반복을 수행합니다.

Iterator

Iterator는 일련의 데이터를 담고 있는 객체를 의미합니다. 내부 메서드인 `__next()__` 혹은 내부 구현 메서드인 `next()` 를 계속 호출해서 연속적으로 데이터를 반환합니다. 그러다 모든 데이터를 접근했다면 `StopIteration` 예외가 발생해서, 반복을 종료합니다. 보통 메서드 `iter()` 를 사용해서 iterator를 구현할 수 있습니다.[3]

차이점

- **용도**: iterable은 **반복 가능**한 객체를 나타내며, iterator는 iterable의 요소를 하나씩 순차적으로 접근하는 데 사용됩니다.
- **메소드**: iterable은 `__iter__()` 메서드를, iterator는 `__iter__()` 와 `__next__()` 메서드를 모두 구현합니다.
- **재사용 여부**: iterable은 반복적으로 사용될 수 있지만, iterator는 한 번만 사용됩니다.

예시

```
# iterable 예시
my_list = [1, 2, 3]  # 리스트는 iterable입니다.
for item in my_list:
    print(item)  # 리스트의 각 요소에 접근합니다.

# iterator 생성
my_iter = iter(my_list)  # 이터레이터 생성

# iterator 사용
print(next(my_iter))  # 출력: 1
print(next(my_iter))  # 출력: 2
print(next(my_iter))  # 출력: 3

# iterator는 여기서 끝에 도달하며, 더 이상 요소가 없습니다.
```

💡 **결론**
앞으로 나올 설명에서 어떤 자료구조 혹은 객체가 iterable하다고 설명하면,
반복을 수행할 수 있는 객체라고 이해하시면 됩니다.

Mutable vs Immutable

Mutable

Mutable은 id를 유지한 채로, **수정할 수 있는** 객체를 의미합니다. 이러한 객체들은 보통 여러 데이터를 담고 있는 객체입니다. 대표적으로는 리스트(list), 딕셔너리(dictionary), 집합(set) 등이 있습니다.

Immutable

Immutable은 **고정**된 값으로 이루어진 객체, 즉 **수정 불가능**한 객체를 의미합니다. 대표적으로 숫자, 문자열, 튜플(tuple) 등이 있습니다. 따라서 새로운 값을 저장하기 위해서는 또 다른 객체를 생성해야 합니다. 객체 내부적인 수정이 불가능하다는 특성은 딕셔너리(dictionary)의 키(key)와 같은 역할을 할 때 큰 힘을 발휘합니다.

얕은 복사 vs 깊은 복사, 그리고 함수 인자 전달 방식

id

`id()` 메서드는 객체의 **"정체성"**을 반환합니다. 그 정체성은 객체가 살아있는 동안 가지는 고유한 정수형 값으로 표현됩니다. 당연히 서로 다른 객체는 명확하게 구분되어야 하므로 다른 id 값을 부여받습니다. id 값에 대한 개념은 매우 중요하기 때문에 잘 알고 있어야 합니다. 다음 간단한 코드로 id가 서로 다른 객체마다 다르게 부여된다는 것을 확인해 봅시다.

```
a = 3
b = 4

id(a)
id(b)
```

▼ 결과

```
# 부여되는 id는 환경에 따라 다르게 부여되기 때문에 이 예시와는 다른 정수값이 반환될 수도 있습니다.

4353497784
4353497816
```

얕은 복사 vs 깊은 복사

얕은 복사 (Shallow copy)

얕은 복사란 객체의 레퍼런스(reference)[4] 값만을 복사하는 것을 의미합니다. 말 그대로 온전히 객체의 모든 것을 복사하는 것이 아니라, 레퍼런스를 기존 객체와 공유하게 됩니다. 파이썬의 할당(assignment)은 기본적으로 얕은 복사 방식으로 이루어지기 때문에 많은 주의가 필요합니다. 다음 코드를 보며 설명을 이어가겠습니다.

```
a = [1, 2, 3, 4, 5]
b = a # 얕은 복사

b.append(10)
print("a :", a)
print("b :", b)
print("a id :", id(a))
print("b id :", id(b))
```

▼ 결과

```
a : [1, 2, 3, 4, 5, 10]
b : [1, 2, 3, 4, 5, 10]
a id : 4370369216
b id : 4370369216
```

객체 `b`는 리스트형 객체인 `a`를 할당받았습니다. 얕은 복사에 의해 객체 `b`는 객체 `a`의 내용을 **공유**하게 됩니다. 즉, **같은 레퍼런스**를 가리키기 때문에 한 객체에 변화가 발생하면, 다른 객체에도 자동으로 그 변화가 똑같이 적용됩니다.

인덱스 슬라이싱을 통해 복사한다면 객체 `c`와 객체 `d`가 서로 다른 레퍼런스를 갖기에 한 객체의 내부 원소가 변하더라도 다른 객체에는 영향을 미치지 않음을 확인할 수 있습니다.

```
c = [1, 2, 3, 4, 5]
d = c[:] # 깊은 복사일까요??
d.pop()

print("c :", c)
print("d :", d)
print("c id :", id(c))
print("d id :", id(d))
```

▼ 결과

```
c : [1, 2, 3, 4, 5]
d : [1, 2, 3, 4]
c id : 1960037332224
d id : 1960037285632
```

그렇다면 이것도 깊은 복사일까요?

```
c = [[10, 20, 30], 2, 3, 4, 5]
d = c[:] # 얕은 복사입니다.
```

```
d[0].pop()

print("c :", c)
print("d :", d)
```

▼ 결과

```
c : [[10, 20], 2, 3, 4, 5]
d : [[10, 20], 2, 3, 4, 5]
```

위와 같이 다른 레퍼런스를 갖더라도 내부 원소가 **mutable**인 경우 원소에 대해서는 같은 레퍼런스를 갖기에 얕은 복사입니다.

- `obj.copy()` : 리스트의 경우, `list[:]`와 동일한 방식입니다.

깊은 복사 (Deep copy)

깊은 복사는 객체의 모든 것을 재귀적으로 복사해서 새로운 객체를 만드는 방식을 말합니다. 얕은 복사와 다르게 원본 데이터 내용을 복사하고, 원본 데이터와 다른 id값을 가져서 서로 다른 레퍼런스를 갖게 됩니다.

```
import copy

e = [[10, 20, 30], 2, 3, 4, 5]
f = copy.deepcopy(e) # 깊은 복사
f[0].pop()

print("e :", e)
print("f :", f)
```

▼ 결과

```
e : [[10, 20, 30], 2, 3, 4, 5]
f : [[10, 20], 2, 3, 4, 5]
```

copy.deepcopy() vs copy.copy()

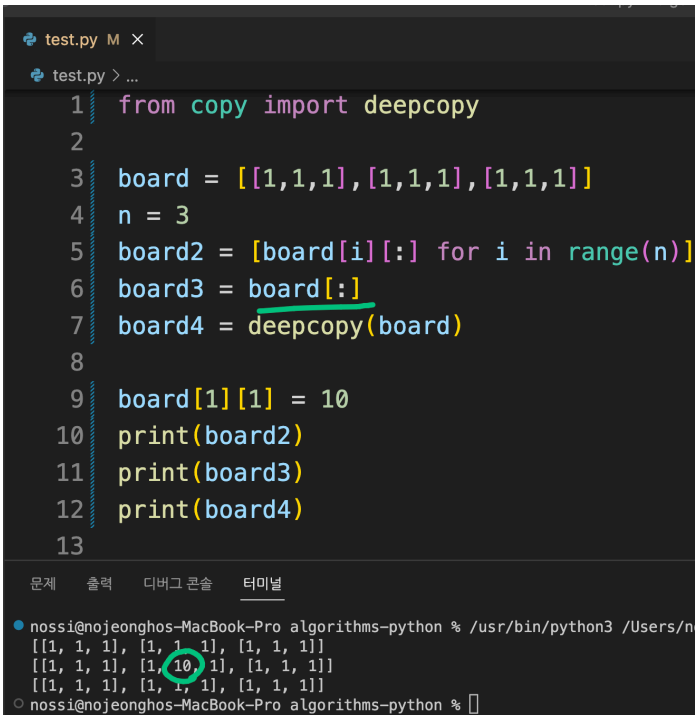
copy 모듈은 `copy.copy()` 메서드와 `copy.deepcopy()` 메서드를 지원합니다. 보여드리지 않은 `copy.copy()` 메서드는 **얕은 복사**를 하기 때문에 사용하실 때 주의를 하셔야 합니다. 참고로 `copy.deepcopy()` 는 실행속도가 매우 느리지만, 2차원 리스트를 복사할 때에는 사용할 수 있습니다.

💡 **그렇다면 어떤 방법을 선택해야 하는가??**

문제에서 주어진 데이터의 형태를 먼저 확인합니다. 만약 리스트의 원소가 모두 immutable인 경우 `[:]`, `obj.copy()`가 **깊은 복사와 유사**한 기능을 지원하기에 해당 방식을 사용하는 것이 더욱 빠릅니다. 만일 고차원 리스트인 경우

```
g = [[1, 2, 3], [2, 3, 4], [3, 4, 5]]
h = [g[i][:] for i in range(len(g)]
```

와 같은 방식으로 깊은 복사를 흉내낼 수 있습니다.



Call by assignment (Call by object reference)

함수 인자 전달 방식은 Call by Reference와 Call by Value로 나뉩니다.

💡 Call by Value와 Call by Reference

Call by Value : 변수의 **복사값**을 전달하는 방식으로, 함수 내에서 인자에 대해 수정하더라도 원본 데이터는 유지됩니다.

Call by Reference : 변수의 **주솟값**을 전달하는 방식으로, 함수 내에서 인자에 대해 수정한다면 원본 데이터도 같이 수정됩니다.

하지만, 파이썬은 특이하게도 Call by assignment, 혹은 Call by object reference 라는 방식으로 함수 인자 전달이 이루어집니다. 쉽게 말해 이 방식은 주어지는 객체에 따라 전달 방식에 차이가 존재한다는 것으로, **Call by Value와 Call by Reference를 객체의 데이터형에 따라 다르게 적용**하는 것입니다.

위에서 파이썬의 객체는 Mutable과 Immutable로 나뉜다는 것을 확인했습니다. 이것이 함수 인자 전달 방식에도 적용되어, 수정이 가능한 **Mutable 객체는 Call by Reference** 방식을, 수정이 안되는 **Immutable 객체는 Call by Value** 방식을 사용합니다. 두 가지 상황을 아래의 예시 코드를 통해 확인해 보겠습니다.

```
# Call by Value 적용 (Immutable 객체 : int, string, tuple, ...)
a = 3

def foo(a):
    a = a + 3
    print("Inner Function :", a)

foo(a)
print("Outer Function :", a)
```

▼ 결과

```
Inner Function : 6
Outer Function : 3
```

```
# Call by Reference 적용 (Mutable 객체 : list, dictionary, ...)
li = [1, 2, 3, 4, 5]

def foo(l):
    l.pop()
    print("Inner Function :", l)

foo(li)
print("Outer Function :", li)
```

▼ 결과

```
Inner Function : [1, 2, 3, 4]
Outer Function : [1, 2, 3, 4]
```

1번째 예시 코드에서는 함수 내부와 외부에서의 출력 결과가 다르게 나왔지만, 2번째 예시 코드의 두 출력문은 모두 동일하게 나왔습니다. 이처럼 객체 데이터형에 따라 적용되는 방식이 다르기 때문에 특히, **mutable 객체**를 함수의 인자로 넘길 때 많은 주의가 필요합니다.

Mutable 객체와 깊은 복사(Deep copy)

위에서 보여드린 것처럼 mutable 객체는 얕은 복사로 인자가 전달되기 때문에 무턱대고 함수로 처리해서 새로운 값으로 반환받게 되면, 원본 데이터를 잃게 되어 원하는 논리를 구현하지 못하게 될 수 있습니다. 이 경우,
깊은 복사로 새로운 객체를 생성해서 처리하는 것이 바람직합니다.

Packing vs Unpacking

Tuple

패킹(packing)과 언패킹(unpacking)에 알아보기 전에, 튜플(tuple)에 대해 먼저 보겠습니다. 튜플은 **immutable**, **iterable**, 그리고 **sequence** 객체에 해당합니다. 튜플은 소괄호 `()`로 묶여서 표현되는데, **소괄호를 안 하더라도** 콤마 `,`로 묶으면 튜플이 생성됩니다. 즉, 튜플을 결정하는 핵심 요소는 소괄호가 아닌 콤마 `,`입니다.

```
a = tuple([1, 4, 9])
b = ('t', 'b', [1, 2])
c = 1, 2, 3, 'a'
d = 0,

print("a :", a)
print("b :", b)
print("c :", c)
print("d :", d)
```

▼ 결과

```
a : (1, 4, 9)
b : ('t', 'b', [1, 2])
c : (1, 2, 3, 'a')
d : (0,)
```

위의 코드를 통해 모든 객체가 튜플로 출력되는 것을 알 수 있습니다. 간혹 튜플이 별로 안 쓰인다고 생각하시는 경우가 있지만, 직접적으로도, 간접적으로도 많이 쓰입니다. 가장 대표적인 것이 Packing과 Unpacking을 사용하는 경우입니다.

Packing vs Unpacking

```
# Packing 예시
team = "Tom", "Bob", "Sam", "Michael"
print(team)

# Unpacking 예시
employee_info = [("Lee", 24), ("Park", 20), ("Kim", 25), ("Jin", 21)]
name, age = employee_info.pop()
print(name)
print(age)
```

▼ 결과

```
('Tom', 'Bob', 'Sam', 'Michael')
Jin
21
```

파이썬의 강력한 기능 중 하나가 바로 Packing과 Unpacking입니다.[5] Unpacking을 통해 `name, age` 변수에 묶여있는 객체들을 한 번에 할당할 수 있어 효율적인 코드 작성을 할 수 있습니다.

이렇게 여러 객체를 한꺼번에 담는 것은 매우 좋은 기능이지만, 만약 일부 객체만을 이용하고 싶을 경우 어떻게 해야 할까요? 이럴 때 `*`를 사용해서 효율적으로 packing과 unpacking을 수행할 수 있습니다.

```
prof_list = [
    ("Kim", 52, "010-1111-1222", "soccer", 5),
    ("Park", 48, "010-3311-6622", "football", 4),
    ("Oh", 57, "010-3271-6929", "tennis", 1),
    ("Lee", 42, "010-6274-8809", "soccer", 2)
    ]

for name, age, *_ in prof_list:   # 3개의 원소를 1개의 원소로 패킹(Packing)하기
    print("prof_name :", name)
    print("prof_age :", age)
    print("Others :", _)
```

▼ 결과

```
prof_name : Kim
prof_age : 52
Others : ['010-1111-1222', 'soccer', 5]
prof_name : Park
prof_age : 48
Others : ['010-3311-6622', 'football', 4]
prof_name : Oh
prof_age : 57
Others : ['010-3271-6929', 'tennis', 1]
prof_name : Lee
prof_age : 42
Others : ['010-6274-8809', 'soccer', 2]
```

리스트 `prof_list`의 모든 데이터는 이름, 나이, 전화번호, 취미, 평가 점수 순으로 이루어진 튜플입니다. 우선 for문을 통해 `prof_list`를 돌면서 하나의 튜플에서 5가지 데이터를 unpacking합니다. 만약 바로 for문을 사용했다면 5개의 변수를 통해 각각 접근했어야 합니다. 하지만, `*`을 이용하면, 남은 인자들을 하나의 **리스트 객체로 packing**하게 됩니다. 따라서 위의 코드와 같이 만약 이름과 나이만 필요할 때, `name` 그리고 `age`로 설정하고 나머지는 `*_`로 묶어주면, 3개의 변수로 하나의 튜플을 이루고 있는 모든 변수를 담을 수 있습니다.

`*` 를 사용한 Packing은 for문뿐만 아니라, 함수의 인자를 전달받을 때도 사용할 수 있습니다.

```
def get_mult_result(*li):
    result = 1
    for i in li:
        result *= i
    return result

print(get_mult_result(3, 5, 7, 2, 4))
```

▼ 결과

```
840  # 입력된 전체 숫자들의 곱
```

위의 코드에서 `*`를 통해 나열된 숫자들이 하나의 튜플(tuple)로 packing되어 전달됩니다. 또 다른 특수 인자 전달 형태로 `**`도 존재합니다. 흔히 `**kwargs`로 쓰이는데, **keyword arguments**의 약자로 쓰입니다. 딕셔너리 형태로 인자를 넘기는 방식인데, key값으로 인자 이름, value값으로 인자로 넘기는 데이터가 담깁니다. 이러한 key-value 쌍들을 한꺼번에 packing해서 `**kwargs`를 통해 함수 내부에 전달됩니다. 간단한 예시를 들자면, 아래와 같습니다.

```
def greeting(**kwargs):
    for time, name in kwargs.items():
        if (time == "morning") or (time == "afternoon") or (time == "evening"):
            print(f"Good {time}! Mr./Ms. {name}!!")
```

```
        else:
            print("Hello!")

greeting(evening="park", morning="kim", afternoon="lee")
greeting(new="Kim")
```

▼ 결과

```
Good evening! Mr./Ms. park!!
Good morning! Mr./Ms. kim!!
Good afternoon! Mr./Ms. lee!!
Hello!
```

가령 print 함수에서 `end` 인자를 따로 설정해서 출력문 형식을 임의로 바꿔 사용하듯이, 흔히 파이썬에서 특정 함수 혹은 메서드를 실행할 때 위와 같은 방식으로 다양한 인자에 대해서 선택적으로 설정해서 전달하는 것을 볼 수 있는데, 모두 `**` 에 의한 것입니다.

zip 함수 (Packing)

zip 함수는 여러 iterable 객체를 인자로 받아서 병렬로 반복을 수행하며, 각 객체에서 같은 반복 횟수에 접근하는 데이터를 한 번에 튜플(tuple)로 묶어서 반환해 줍니다. 코딩 테스트에서 자주 이용하는 함수로, 독립적인 iterable 객체들을 묶고 싶을 때 편리하게 사용할 수 있습니다. 예시 코드로 알아보겠습니다.

```
friends = ["Bob", "Sally", "Jessica", "Mike", "David"]
friend_nums = [10, 8, 39, 21, 2]

for f, n in zip(friends, friend_nums):
    print(f"My friend {f} has {n} friends")
```

▼ 결과

```
My friend Bob has 10 friends
My friend Sally has 8 friends
My friend Jessica has 39 friends
My friend Mike has 21 friends
My friend David has 2 friends
```

위의 코드에서도 각 친구 별로 친구 숫자를 출력하기 위해 두 개의 리스트를 zip 함수로 packing해서 각 반복 횟수에 대응되는 데이터가 변수 `f`, `n`에 할당되었습니다. zip 함수의 인자로 2개 이상의 iterable 객체를 전달할 수 있습니다. 그렇게 되면 for문을 돌면서 전달된 iterable 객체 수로 이루어진 튜플을 리스트가 종료할 때까지 반환합니다.

💡 **인자로 넘기는 iterable 객체의 길이가 서로 다를 경우**

주어진 여러 iterable 객체들 중 **가장 짧은 길이**의 객체에 맞춰서 생성되는 튜플의 개수가 정해집니다.

itertools

itertools는 파이썬의 대표 라이브러리 중 하나로, 완전 탐색 문제 중 순열, 조합 문제에서 매우 활용도가 높은 `permutations()` 와 `combinations()`를 포함하고 있습니다. 해당 라이브러리에 존재하는 함수들은 나열되는 결괏값을 담고 있는 iterator를 반환합니다.

우선 itertools 라이브러리를 불러옵니다.

```
import itertools
# 바로 함수를 사용하고 싶다면, 다음과 같이 쓰시면 됩니다.
# from itertools import permutations, combinations
```

permutations()

주어진 iterable 객체에 대해서 **순서를 고려**한 **순열**의 모든 가짓수를 포함하는 iterator를 반환하는 함수입니다. 1번째 인자로 여러 데이터 값으로 나열된 iterable 객체가 주어지고, 2번째 인자로 하나의 묶음에 들어갈 데이터의 개수를 전달합니다. iterator가 반환되기 때문에 반환 객체 자체를 사용하기보다는 보통 `list()`를 사용해서 순열의 모든 가짓수를 리스트에 담아서 사용합니다.

- 사용 방식 : `permutations([iterable], [number])`

```
import itertools

p = itertools.permutations("AMSJ", 2)
p_list = list(p)
print(p_list)
```

▼ 결과

```
[('A', 'M'), ('A', 'S'), ('A', 'J'), ('M', 'A'), ('M', 'S'), ('M', 'J'), ('S', 'A'), ('S', 'M'), ('S', 'J'), ('J', 'A'), ('J', 'M'
```

combinations()

`permutations()` 와 비슷하지만, 이 함수는 **조합**의 모든 가짓수를 포함하는 iterator를 반환합니다. 조합은 **순서를 고려하지 않은** 고유한 데이터의 묶음을 의미합니다. 따라서 순서가 달라도 같은 종류의 데이터가 존재하면, 중복 처리가 됩니다. `permutations()`와 동일하게, 1번째 인자로 여러 데이터 값이 존재하는 iterable 객체가 주어지고, 2번째 인자로 하나의

조합에 들어갈 데이터의 개수를 입력합니다. iterator가 반환되기 때문에 객체 자체로 쓰이기보다는 리스트로 변환한 다음에 활용됩니다.

- 사용 방식 : `combinations([iterable], [number])`

```
import itertools

c = itertools.combinations([3, 7, 2, 1, 4], 3)
comb_list = list(c)
print(comb_list)
```

▼ 결과

```
[(3, 7, 2), (3, 7, 1), (3, 7, 4), (3, 2, 1), (3, 2, 4), (3, 1, 4), (7, 2, 1), (7, 2, 4), (7, 1, 4), (2, 1, 4)]
```

permutations()와 combinations() 사용 시 주의 사항

각 함수는 주어진 iterable 객체 내의 데이터들의
고유한 위칫값(인덱스)으로 순열 혹은 조합의 가짓수를 만들어내는 것이기 때문에, 만약 중복되는 데이터가 존재하더라도 다른 인덱스를 갖고 있다면 서로 다른 데이터로 취급합니다. 아래 예시 코드처럼 만약 중복되는 데이터가 주어져도, 반환되는 순열 혹은 조합의 가짓수는 모두 잘 생성된 것이라고 할 수 있습니다. 따라서, 올바른 문제 풀이에서 벗어나지 않는 선에서, 고유한 값들로만 이루어진 iterable 객체를 가지고 순열 혹은 조합을 계산해야 혼란과 실수를 줄일 수 있을 것입니다.

```
# 중복되는 다수의 데이터가 존재하는 iterable 객체에 대한 permutations와 combinations
import itertools

nums = [1, 2, 2]
chars = "AABCCC"

num_list = list(itertools.permutations(nums, 3))
print(f"num_list : {num_list}")

char_list = list(itertools.combinations(chars, 3))
print(f"char_list : {char_list}")
```

▼ 결과

```
num_list : [(1, 2, 2), (1, 2, 2), (2, 1, 2), (2, 2, 1), (2, 1, 2), (2, 2, 1)]
char_list : [('A', 'A', 'B'), ('A', 'A', 'C'), ('A', 'A', 'C'), ('A', 'A', 'C'), ('A', 'B', 'C'), ('A', 'B', 'C'), ('A', 'B', 'C')]
```

기타 - 알아두면 좋은 방법

필요없는 반환값은 '_'로 처리하기

함수에서 사용하기

함수를 사용하다가 필요 없는 반환값을 정의된 함수 형태에 의해 어쩔 수 없이 반환해야 한다면, '_' 문자를 이용해서 가시적으로 필요 없는 데이터임을 보일 수 있습니다.

```
import math

n = 5

def get_xy(n, x):
    if x > n:
        return n, 0
    else:
        return x, math.sqrt((n**2)-(x**2))

_, y = get_xy(n, 3)
print(_, y)
```

▼ 결과

```
3 4.0
```

물론 당연히 '_'도 변수이기 때문에 함수 반환값을 저장합니다. 그래서 실제로 코드 구현에 사용할 수 있지만, 암묵적으로 이런 방법을 통해 선택적으로 반환값을 받는다는 것을 보여줄 수 있다는 점에서 편리합니다. 이 방법은 반복문에서도 종종 사용합니다.

반복문에서 사용하기

for문에서 이렇게 여러 변수에 값을 담아서 사용하는 경우, 일부 데이터만 사용한다는 것을 표현할 때, '_'를 활용하면 좋습니다.

```
info = [('a', 12), ('b', 93), ('c', 102), ('d', 1)]

for _, password in info:
    print(password)
```

▼ 결과

```
12
93
102
1
```

문자와 아스키(ASCII)코드

코딩 테스트 문제를 보면, 영문자를 활용해서 특정 문제 속 개념을 지칭하는 경우가 많습니다. 예를 들어, 최단 거리 문제에서 각 건물을 A, B, ... , E, F 등과 같이 표시하는 경우, 인물 간의 관계를 표시한 문제에서 각 인물을 A, B, ..., E, F로 표시하는 경우 등이 있습니다. 무조건은 아니지만 코드를 구현하다 보면 이런 영문 대소문자를 아스키코드로 변환해서 값으로 비교하고 싶은 경우가 있습니다. 파이썬은 하나의 영문자를 길이 1인 문자열로 생각하기 때문에 따로 변환하는 함수를 사용해야 하는데, 그것이 바로 파이썬 내장함수 `ord()` 입니다.

```
# 숫자 0 ~ 9 아스키 코드 범위 : 48 ~ 57
print(f"0 ~ 9 : {ord('0')} ~ {ord('9')}")
# 영문 대문자 아스키 코드 범위 : 65 ~ 90
print(f"A ~ Z : {ord('A')} ~ {ord('Z')}")
# 영문 소문자 아스키 코드 범위 : 97 ~ 122
print(f"a ~ z : {ord('a')} ~ {ord('z')}")

word = "MissIssiPpi"
for s in word:
    if (65 <= ord(s) <= 96):
        print(s)
```

▼ 결과

```
0 ~ 9 : 48 ~ 57
A ~ Z : 65 ~ 90
a ~ z : 97 ~ 122
M
I
P
```

내장 함수 `ord()` 와 반대로, 아스키코드에서 문자로 변환해 주는 내장 함수 `chr()` 도 있습니다.

```
print(f"ASCII code of 'A' : {ord('A')}")
print(f"The character of ascii code 65 : {chr(65)}")
```

▼ 결과

```
ASCII code of 'A' : 65
The character of ascii code 65 : A
```

/ 와 //의 차이

파이썬에서 나눗셈 연산자는 `/` 와 `//` 가 있습니다. 두 연산자 모두 나눗셈을 수행하지만, 반환되는 자료형에 차이가 있습니다. `/` 는 정수형끼리 나눴어도 **부동소수점** 값이 반환되고, `//` 는 정수형끼리 나누면 **정수형** 값인 **몫**을 반환합니다. 나누어떨어지는 것과는 상관없이 반환 값의 자료형이 결정되어 있기 때문에 코딩테스트에서 사용할 때 주의해야 합니다.

```
print(5/2)
print(5//2)
```

▼ 결과

```
2.5
2
```

결론적으로는 `//`를 더 자주 사용합니다. 예를 들어, 투 포인터(two pointer) 문제와 같이 인덱스를 다룰 경우, 이진 탐색(Binary Search), 진법 계산을 하는 경우, 실제로 문제에서 몫과 나머지는 다루는 경우 등이 사용 예시라고 할 수 있습니다.

enumerate()

내장함수 `enumerate()` 은 iterable 객체를 for문에 사용할 때, **인덱스**를 동시에 쓸 수 있게 해줍니다. 따라서 for문을 사용할 때 매우 유용합니다.

`enumerate()` 는 iterable 객체를 인자로 받아 열거 객체를 반환하는데, 이때 **count값과 데이터**를 **튜플(tuple)**로 묶습니다. 따라서 unzip을 통해 각 요소에 접근하여 이용할 수 있습니다. 기본 count 초기값은 0으로, 인자 start를 지정해서 원하는 초기값을 설정할 수 있습니다.

```
player_list = ["Sam", "Tom", "Alice", "Nancy", "Bob", "Michael"]

# 기본적인 enumerate() 사용
for i, p in enumerate(player_list):
    print(i, p)

print()

# 인자 start=1로 지정한 상황
```

```
for j, player in enumerate(player_list, start=1):
    print(f"Player {j} : {player}")
```

▼ 결과

```
0 Sam
1 Tom
2 Alice
3 Nancy
4 Bob
5 Michael

Player 1 : Sam
Player 2 : Tom
Player 3 : Alice
Player 4 : Nancy
Player 5 : Bob
Player 6 : Michael
```

print 함수의 end 인자 활용하기 (디버깅에 활용하기)

코딩 테스트를 풀면서 원하는 답이 나오지 않을 때 구현된 코드 논리가 어떻게 되는지를 확인할 때, 출력문을 잘 활용해서 확인하는 것이 매우 중요합니다. 파이썬의 `print` 함수는 기본적으로 마지막 글자로 개행문자 `\n`을 넣기 때문에, `end` 인자 설정 없이 여러 줄에 걸쳐서 `print` 함수를 연속적으로 사용하면, 쓰인 `print` 함수 개수 만큼의 출력문이 나오게 됩니다. 하지만 임의로 마지막 글자를 원하는 글자로 바꿀 수가 있는데, 그때 쓰이는 인자가 `end` 입니다. 다음 활용 예시를 보겠습니다.

```
n = [9, 2, 5, 3, 1, 23, 24, 5, 431, 543]
n.sort()

for i in n:
    print(i, end=" ") # 출력문의 마지막을 공백 문자로 변경하기
print()
```

▼ 결과

```
1 2 3 5 5 9 23 24 431 543
```

보통 위의 코드처럼 '공백 문자', '콤마' 등이 자주 사용됩니다. 이 기능을 활용해서 깔끔하게 출력되게끔 할 수 있다면, 잘못 구현된 지점을 더 빠르게 찾을 수 있을 것입니다.

for-else / while-else 구문 활용하기

반복문을 실행하다 보면 특정 조건에 의해 반복문을 빠져나오도록 코드를 작성하는 경우가 매우 많습니다. 파이썬에서는 **반복문을 모두 실행했을 때**와 **반복문 도중 빠져나올 때**를 구분하는 문법인 **for-else 구문, while-else 구문**이 존재합니다. 다음은 활용 예시입니다.

for-else 구문을 사용하지 않고 내부 for문에만 break를 설정하면 다음과 같은 결과를 얻게 됩니다.

```
# arr1과 arr2에 중복되는 숫자가 존재하는지 확인하기
arr1 = [1,2,3,4]
arr2 = [5,6,3,7]

for i in arr1:
    for j in arr2:
        print(i, j)
        if i == j:
            print("탈출")
            break
```

▼ 결과

```
1 5
1 6
1 3
1 7
2 5
2 6
2 3
2 7
3 5
3 6
3 3
탈출
4 5
4 6
4 3
4 7
```

이처럼 외부 for문이 종료되지 않아 의도하지 않은 결과가 나옴을 알 수 있습니다. 이를 for-else 구문을 통해 해결할 수 있습니다.

```
# arr1과 arr2에 중복되는 숫자가 존재하는지 확인하기
arr1 = [1,2,3,4]
arr2 = [5,6,3,7]

for i in arr1:
```

```
    for j in arr2:
        print(i, j)
        if i == j:
            print("탈출")
            break
    else:
        continue
    break
```

▼ 결과

```
1 5
1 6
1 3
1 7
2 5
2 6
2 3
2 7
3 5
3 6
3 3
탈출
```

이중 for문을 사용해서 `arr1`과 `arr2`에 접근합니다. 그렇다면 내부 for문에 의해서 2가지 상황이 나올 것입니다.

- `i == j` ⇒ "탈출" 출력 ⇒ else문 건너뛰기 ⇒ 외부 for문의 break 만남 ⇒ 외부 for문 빠져나가기
- `i != j` ⇒ if문 들어가지 않음 ⇒ `arr2` 의 다음 요소 접근 ⇒ 모든 요소 접근했다면, 외부 for문의 else문으로 들어감 ⇒ continue에 의해 `arr1` 의 다음 요소 접근 ⇒ 내부 for문 진입해서 `arr2` 요소 접근 ⇒ 반복

for-else문의 장점은 그냥 내부 for문에서 if-break 구문을 사용했을 때와 다르게, 반복문 종료의 요인을 명확하게 구분할 수 있다는 것입니다. 이런 구분으로 위와 같이 중복 데이터 찾기, 데이터 존재 여부 확인하기 등의 경우로 다양하게 활용할 수 있습니다. while-else문도 다음과 같이 사용할 수 있습니다.

```
n = 0
while (n != 4):
    if n > 5:
        print("무한 반복문 탈출!!")
        break
    n += 1
else:
    print("아무튼 탈출!!")
```

▼ 결과

```
아무튼 탈출!!
```

최대, 최소값 구하기

큰 값 설정하는 방법

1. `sys.maxsize` (int형)

```
import sys

positive = sys.maxsize
negative = -sys.maxsize
```

2. 비트연산을 이용 4byte, 8 byte (int형)

```
# 4byte
positive = 1<<31 - 1
negative = -1<<31
# 8byte
positive = 1<<31 - 1
negative = -1<<31
```

2. `float("inf")` (float 형)

```
positive = float("inf")
negative = float("-inf")
```

3. `math.inf` (float 형) - **Python 3.5 이상**

```
import math
positive = math.inf
negative = -math.inf
```

5. 문제에서 정해주는 최대값을 사용

---

1. 멤버의 의미

객체 내부에 여러 가지 데이터를 담긴 경우가 많습니다. 이런 객체를 구성하고 있는 데이터를 멤버라고 부릅니다.

2. 임의 iterable 객체 만들기

클래스에서 `__iter__()` 혹은 `__getitem__()` 메서드를 정의하면, 해당 클래스로 생성된 객체는 iterable 객체의 특성을 지니게 됩니다. 즉, 기존 iterable 객체들처럼 사용할 수 있게 됩니다.

3. Iterable과 Iterator의 관계

결론적으로는 iterator 또한 iterable 객체라고 정의할 수 있습니다. 하지만, 반대로, 모든 Iterable 객체가 Iterator가 될 수는 없다는 것을 의미합니다.

4. 레퍼런스 (Reference)

파이썬에서 레퍼런스는 특정 객체의 메모리 주소를 가리키는 말입니다.

5. Packing & Unpacking

공식 문서에서 정의하는 이름은 **Tuple Packing**, 그리고 **Sequence Unpacking**이라고 합니다.