데이터 구조

객체 지향 ↔ 절차 지향

- 객체 지향은 class가 특정되면 객체지향이라 한다.
- class 없이 절차로만 진행되는 코드를 절차 지향이라 한다.

class

- 행위 : 함수(메서드)
- 값 : 변수(속성)

변수 이름 지정

- 자바는 카멜 케이스 (ex : aTest, startsWith)
- 파이썬은 a_test, starts_with

문자열(String)

.find(x)

- x를 찾고 x가 나오는 가장 첫번째 인덱스를 반환합니다.
- `find`(*sub*[, *start*[, *end*]])
    - start, end 로 범위를 지정할 수 있다.

.index(x)

- x의 첫 번째 위치를 반환합니다. x가 없으면 에러가 발생합니다.

.startswith(x), /.endswith(x)

- .startswith(x) : 문자열이 x로 시작하면 True를 반환하고 아니면 False를 반환합니다.
- .endswith(x) : 문자열이 x로 끝나면 True를 반환하고 아니면 False를 반환합니다.
- **화이트 스페이스나 인코딩 문제를 피하기 위해 문자열 분할보다 startswith, endswith를 권장합니다!**

- 한글은 유니코드(utf-8)를 이용하기 때문에 호환성 문제가 발생할 가능성이 있음.

.isalpha(), .isspace(), .isupper(), .istitle(), .islower()

- `.isalpha()` : 문자열이 (숫자가 아닌)글자로 이루어져 있는가?
- `.isspace()` : 문자열이 공백으로 이루어져 있는가?, 화이트 스페이스도 공백으로 판정한다.
- `.isupper()` : 문자열이 대문자로 이루어져 있는가?
- `.istitle()` : 문자열이 타이틀 형식으로 이루어져 있는가?
- `.islower()` : 문자열이 소문자로 이루어져 있는가?

.isdecimal(), .isdigit(), .isnumeric()

- 문자열(string) 내의 숫자를 말함
- `.isdecimal()` : 문자열이 0~9까지의 수로 이루어져 있는가?
- `.isdigit()` : 문자열이 숫자로 이루어져 있는가?
- `.isnumeric()` : 문자열을 수로 볼 수 있는가?

.replace(old, new[, count])

- **비파괴 메서드**
- old를 new로 대체합니다.
- count만큼 반복합니다.

.strip([chars])

- **비파괴 메서드**
- 특정한 문자들을 지정하면 문자열의 모든 조합을 제거합니다. 인자가 없을 경우 공백을 제거합니다.
- 연속된 문자열 기준으로 제거하는 것이 아니라, 포함된 문자 하나하나를 모두 제거합니다.
- 양 사이드로부터 제거해나가다가 해당 사항이 없으면 멈춥니다.
- .lstrip([chars]), .rstrip([chars]) 로 왼쪽 공백, 오른쪽 공백을 각각 제거할 수 있습니다.

.split([chars])

- 문자열을 특정한 단위로 나누어 리스트로 반환합니다.

'separator'.join(iterable)

- **비파괴 메서드**
- iterable의 문자열들을 separator(구분자)로 이어 붙인(join()) 문자열을 반환합니다.
- 다른 메서드들과 달리, 구분자가 join 메서드를 제공하는 문자열입니다.

.capitalize(), .title(), .upper()

- **비파괴 메서드**
- `.capitalize()` : 앞글자를 대문자로 만들어 반환합니다.
- `.title()` : 어포스트로피(')나 공백 이후를 대문자로 만들어 반환합니다.
- `.upper()` : 모두 대문자로 만들어 반환합니다.

.lower(), .swapcase()

- **비파괴 메서드**
- `lower()` : 모두 소문자로 만들어 반환합니다.
- `swapcase()` : 대 ↔ 소문자로 변경하여 반환합니다.

리스트(List)

- **스택에 있어서 원소를 추가하거나 제거할 때, 마지막 인덱스에서 멀어질수록 속도가 느려집니다!**
    - 감안하여 코드를 구성할 것을 권장합니다.
- 링크드 리스트 (연결 리스트)
    - head(시작점)에서 다른 주소를 가리키는 리스트를 참조, 그 리스트는 또 다른 주소를 가리키기 때문에 줄줄이 많은 리스트의 값을 참조할 수 있습니다. 중간에 리스트가 추가되면 전후의 리스트가 가리키는 주소를 바꾸면 그대로 연결이 유지할 수 있습니다.
    - 파이썬은 더블 엔디드 큐, 덱으로 구현이 되어있습니다.

.append()

- 리스트에 x를 형태 그대로 추가합니다.

(arr = [0] + list(map(int, input().split())) + [0] 처럼 리스트 중간에 대입해줄 수 있다.)

.extend(iterable)

- 리스트에 iterable(list, range, tuple, string) 값을 붙일 수가 있습니다.
- iterable 형식의 괄호들을 모두 없애고 대입됩니다.
- 문자열을 대입하면 한글자씩 잘라서 대입합니다.
- a[len(a):] = iterable 과 동일합니다.

.insert(i, x)

- 정해진 위치 i에 x를 추가합니다.
- i가 인덱스를 넘어서도 마지막 인덱스에 추가됩니다. **(에러 안남)**
- **비효율적이기 때문에 속도가 저하될 수 있음. (리스트 뒤에 추가하는 것이 가장 좋음)**

.remove(x)

- 리스트에서 값이 x인 첫번째 항목을 삭제합니다.
- 값을 찾을 수 없으면 에러가 발생합니다.

.pop([i])

- 정해진 위치 i에 있는 값을 삭제하며, 그 항목을 반환합니다.
- i가 지정되지 않으면 **마지막 항목**을 삭제하고 되돌려줍니다.

.clear()

- 리스트의 모든 항목을 삭제합니다.

.index(x)

- x값을 찾아 해당 index 값을 반환합니다.
- 리스트에서는 find 메서드를 사용할 수 없습니다.
- 마찬가지로 x가 없으면 에러 발생합니다.

.count(x)

- 원하는 값의 개수를 반환합니다.
- ```
  # 원하는 값을 모두 삭제하려면 다음과 같이 할 수 있습니다.
  a = [1, 2, 1, 3, 4]
  target_value = 1
  for i in range(a.count(target_value)):
      a.remove(target_value)
  print(a)
  ```

.sort()

- **파괴 메서드**
  - **리턴 값은 없으며**, sort 메서드가 사용된 변수에 바로 반영됩니다.
- 리스트를 정렬합니다.
- 리스트의 모든 원소의 값을 참조하기 때문에 속도가 느려집니다.
- sort(reverse=true) 를 사용하면 거꾸로 정렬해줍니다.
  ```

- 참고 : key 파라미터는 람다 함수를 이용할 때 사용합니다.

.reverse()

- **파괴 메서드** (내장함수 reversed(x)가 비파괴 함수, sort() sorted()와 같은 맥락)
    - **리턴 값은 없으며**, sort 메서드가 사용된 변수에 바로 반영됩니다.
- 리스트의 원소들을 반대로 뒤집습니다.
- sort와 마찬가지로 key와 reverse가 있습니다.

튜플(Tuple)

- 값을 변경할 수 없기 때문에 값에 영향을 미치지 않는 메서드만을 지원합니다.

.index(x[, start[, end]])

- 튜플에서 x와 같은 원소의 첫 번째 인덱스를 반환합니다.
- 해당하는 값이 없으면 에러 발생

.count(x)

- x의 개수를 반환합니다.

셋(Set)

- dir(set)으로 관련 메서드를 모두 확인할 수 있습니다.

.add(elem)

- 리스트의 append() 메서드와 같은 격
- elem을 셋에 추가합니다.

.update(*others)

- 여러 값을 한번에 추가할 수 있습니다.
- 반드시 iterable 데이터 구조를 전달해야 합니다.

.remove(elem)

- elem을 셋에서 삭제합니다.
- 없으면 에러 발생

.discard(elem)

- elem을 셋에서 삭제합니다.
- remove와 다르게 값이 없어도 에러가 발생하지 않습니다.

딕셔너리(Dictionary)

.get(key[, default])

- key를 통해 value를 가져옵니다.
- key가 존재하지 않을 경우 None을 반환합니다. (에러가 발생하지 않습니다.)

.setdefault(key[, default])

- get과 다르게 key가 딕셔너리에 없을 경우, default값을 갖는 key를 삽입한 후 default를 반환합니다.
- default가 주어지지 않을 경우 None을 반환합니다.
- default에는 value값만 작성하면 됩니다.

.pop(key[, default])

- key가 딕셔너리에 있으면 제거하고 그 값을 반환합니다.
- 없으면 default를 반환합니다.
- default가 없을때 key가 없으면 에러가 발생합니다.

.update([other])

- other가 제공하는 key,value 쌍으로 딕셔너리를 덮어씁니다.
- .update(apple='사과') 의 형식으로 작성합니다.
- 다른 딕셔너리나 key/value 쌍으로 되어있는 모든 iterable을 사용 가능합니다.
- ```
  my_dict = {'apple': '사과', 'banana': '바나나', 'melon': '멜론'}
  d = {'mango' : '망고', 'watermelon' : '수박'}
  my_dict.update(d)
  print(my_dict)
  ```

얕은 복사와 깊은 복사

변경 불가능한(immutable) 데이터

- 리터럴(literal) → value, 기본자료형
  - 숫자(Number)
  - 글자(String)
  - 참/거짓(Bool)
- `range()`
- `tuple()`
- `frozenset()`

변경 가능한(mutable) 데이터

- reference, 레퍼런스형, 객체
  - `list`
  - `dict`
  - `set`
- 레퍼런스 타입은 변수가 주소만 가지고 있음. (참조형)
  - 그 주소에 들어있는 것이 실제 데이터
  - 그러므로 다른 변수에 레퍼런스형 변수를 대입해도 주소만 복사되는 것이기에 얕은 복사가 되는 것

- code → data(정적 변수, 파이썬에는 정적 변수가 없고 전역 변수가 여기 저장됨) → heap(리스트 등과 같은 객체가 저장되는 곳) → static(지역 변수)

얕은 복사 (Shallow copy)
  ```

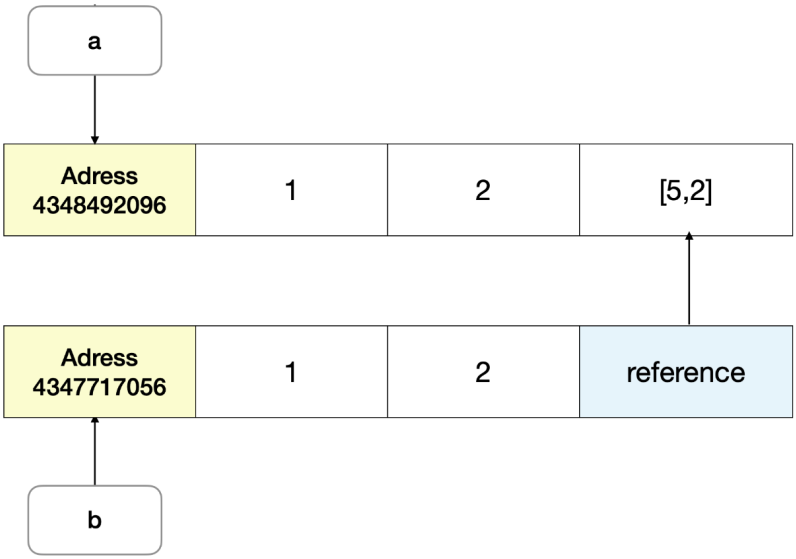
slice 연산자 사용 [:]

- mutable 데이터 중 하나인 리스트를 슬라이싱하여 할당 시, 새로운 id가 부여되며 서로 영향을 받지 않습니다.
- 하지만 2차원 mutable 속 mutable이 또 있는 경우 inner mutable은 복사되지 않으므로 여전히 얕은 복사입니다.

list() 활용

- 슬라이싱과 마찬가지로 얕은 복사입니다.

a와 b의 id는 다르다는 것을 확인하였지만, 내부 값은 영향을 받게 되었습니다.



내부의 객체 `id(a[2])` 과 `id(b[2])` 은 같은 주소를 바라보고 있기 때문입니다.

```
arr = [[0] * 5] * 5
arr[0][0] = 1000
print(arr)

# 결과 : [[1000, 0, 0, 0, 0], [1000, 0, 0, 0, 0], [1000, 0, 0, 0, 0], [1000, 0, 0, 0, 0],
[1000, 0, 0, 0, 0]]

# 얕은 복사에 해당한다.
```

```
arr = [[0] * 5 for _ in range(5)]
arr[0][0] = 1000
print(arr)

# 결과 : [[1000, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0,
0, 0, 0]]

# 깊은 복사에 해당된다.
```

깊은 복사 (Deep copy)

- 레퍼런스형 변수를 완전히 복사하려면 깊은 복사(deep copy)를 해야합니다.
- 깊은 복사는 새로운 객체를 만들고 원본 객체 내에 있는 객체에 대한 복사를 재귀적으로 삽입합니다.
- 즉, 내부에 있는 모든 객체까지 새롭게 값이 변경되게 됩니다.
- copy를 import하고 copy.deepcopy(x)를 사용합니다.
- ```
  import copy

  a = [1, 2, [1, 2]]
  b = copy.deepcopy(a)

  b[2][0] = 3
  print(a)
  ```
  ```