파이썬 프로그래밍

리스트(배열, sequentially stored in continued memory)

- 여러 개의 데이터를 연속적으로 담아 처리하기 위해 사용하는 자료형
- 리스트 대신에 배열이라고 부르기도 함
- 리스트는 대괄호([])안에 원소를 넣어 초기화하며, 쉼표(,)로 원소를 구분
- 비어 있는 리스트를 선언하고자 할 때는 list() 혹은 간단히 []를 이용할
- 리스트의 원소에 접근할 때는 인덱스(Index) 값을 괄호에 넣는다.
- 인덱스는 0부터 시작
- 인덱스 값을 입력하여 리스트의 특정한 원소에 접근하는 것을 인덱싱(Indexing)이라고 한다.
- 파이썬의 인덱스 값은 양의 정수와 음의 정수를 모두 사용
- 음의 정수를 넣으면 원소를 거꾸로 탐색
- 리스트에서 연속적인 위치를 갖는 원소들을 가져와야 할 때는 슬라이싱(Slicing)을 이용
- 대괄호 안에 콜론(:)을 넣어서 시작 인덱스와 끝 인덱스를 설정
- 끝 인덱스는 실제 인덱스보다 1을 더 크게 설정

리스트 슬라이싱

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

a[0:2]  # [1, 2]

a = [1, 2, 3, ['a', 'b', 'c']]

a[0]    #1


a[-1]    #['a', 'b', 'c']


a[3]    #['a', 'b', 'c']
```

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

a[2:5]    #[3, ['a', 'b', 'c'], 4]

a[3][:2]   #['a', 'b']

len(a)     #6
```

리스트 메서드

append(x) Method:

```
people = ["Buffy", "Faith"]
people.append("Giles")
# people[len(people):] = ["Giles“]
print(people)
#['Buffy', 'Faith', 'Giles']
people[len(people):] = ["Xander", "Xander"]
people
#['Buffy', 'Faith', 'Giles', 'Xander‘, "Xander"]]
```

extend(c) Method:

```
people = ["Buffy", "Faith"]
people.extend("Giles")
people #['Buffy', 'Faith', 'G', 'i', 'l', 'e', 's']

people += "Willow"
People #['Buffy', 'Faith', 'G', 'i', 'l', 'e', 's', 'W', 'i', 'l', 'l','o', 'w']

people += ["Xander"]
People #['Buffy', 'Faith', 'G', 'i', 'l', 'e', 's', 'W', 'i', 'l', 'l','o', 'w', 'Xander‘]

people[len(people):] = "Jim“
#['Buffy', 'Faith', 'G', 'i', 'l', 'e', 's', 'W', 'i', 'l', 'l','o', 'w', ‘J 'i', ‘m']
```

리스트 메서드

insert(i, x) Method:

```
people = ["Buffy", "Faith"]
people.insert(1, "Xander")
print(people)
['Buffy', 'Xander', 'Faith']
```

remove() Method:

```
people = ["Buffy", "Faith"]
>>> people.remove("Buffy")
>>> people
['Faith']
>>> people.remove("Buffy")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

리스트 메서드

pop() Method:

```
people = ["Buffy", "Faith"]
people.pop() #'Faith'
people #['Buffy']
```

del:

```
a = [-1, 4, 5, 7, 10]
del a[0]
print(a)     #[4, 5, 7, 10]
del a[2:3]
print(a)     #[4, 5, 10]
del a
# also used to delete entire variable
```

리스트 메서드

index(x) Method:

```
people = ["Buffy", "Faith"]
people.index("Buffy")
```

count(x) Method:

```
people = ["Buffy", "Faith", "Buffy"]
people.count("Buffy")
```

sort() Method:

```
people = ["Xander", "Faith", "Buffy"]
people.sort()
print(people)
#['Buffy', 'Faith', 'Xander']
```

리스트 메서드

reverse() Method:

```
people = ["Xander", "Faith", "Buffy"]
people.reverse() #=people[::-1]
```

List Unpacking

```
first, *rest = [1,2,3,4,5]
print(first)
print(rest) #[2, 3, 4, 5]
```

List Comprehensions

- [item for item in iterable]
- [expression for item in iterable]
- [expression for item in iterable if condition]

```
a = [y for y in range(1900, 1940) if y%4 == 0]
print(a)
#[1900, 1904, 1908, 1912, 1916, 1920, 1924, 1928, 1932, 1936]
b = [2**i for i in range(13)]
print(b)
#[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]
c = [x for x in a if x%2==0]
print(c)
#[1900, 1904, 1908, 1912, 1916, 1920, 1924, 1928, 1932, 1936]
d = [str(round(355/113.0,i)) for i in range(1,6)]
print(d)
#['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

List Comprehensions

[Bad]

```
result = [(x, y) for x in range(10) for y in range(5) if x * y >10]
```

[Good]

```
result = []
for x in range(10):
    for y in range(5):
        if x * y > 10:
            result.append((x, y))
```

튜플(,): immutable

튜플은 처음 정의할 때를 제외하고는 데이터 변경이나 추가, 삭제 등이 불가합니다.
대신 리스트보다는 속도가 빠름

```
t1 = 1234, 'hello!'

t1[0]
#1234

print(t1) #(12345, 'hello!')

t2 = t1, (1, 2, 3, 4, 5) # nested(중첩)
print(t2)#((1234, 'hello!'), (1, 2, 3, 4, 5))
```

```
empty = ()
t1 = 'hello',

len(empty)

len(t1)
print(t1) #('hello',)
```

튜플(,)

튜플메서드: A.count(x), A.index(x)

```
t = 1, 5, 7, 8, 9, 4, 1, 4
t.count(4)

t = 1, 5, 7
t.index(5)
```

튜플 언패킹

모든 iterable 객체는 *를 사용하여 언패킹 가능

```
x, *y = (1, 2, 3, 4)
print(x) #1
print(y) #[2, 3, 4]

*x, y = (1, 2, 3, 4)
print(x) #[1, 2, 3]
```

튜플 형태로 한 줄에 여러 변수의 값을 선언

```
name = "김종국"
age = 20
hobby = "코딩"
print(name, age, hobby) # 김종국 20 코딩

(name, age, hobby) = ("김종국", 20, "코딩")
print(name, age, hobby) # 김종국 20 코딩
```

세트(set, 집합) :중복을 허용하지 않으며 또한 데이터의 순서도 보장하지 않는 컬렉션

```
my_set = {1, 2, 3, 3, 3} # 중복을 허용하지 않으므로 3은 1번만 들어감
print(my_set) # {1, 2, 3}
java = {"유재석", "김태호", "양세형"} # 자바 개발자 집합
python = set(["유재석", "박명수"]) # 파이썬 개발자 집합

# 교집합 (java 와 python 을 모두 할 수 있는 개발자)
print(java & python) # {'유재석'}
# {'유재석'}
print(java.intersection(python))
# 합집합 (java 또는 python 을 할 수 있는 개발자)
print(java | python) # {'박명수', '유재석', '김태호', '양세형'}
print(java.union(python)) # {'박명수', '유재석', '김태호', '양세형'}
```

셋 메서드

- A.add(x)
- A.update(B) 혹은 A|=B는 B를 A에 추가한다.(합집합)
- A.union(B) 와 A|B는 update()메서드와 같지만 연산 결과를 복사본으로 반환한다.
- A.intersection(B)와 A&B는 A와 B의 교집합의 복사본을 반환한다.
- A.difference(B)와 A-B는 A와 B의 차집합의 복사본을 반환한다.

- A.clear()는 A의 모든 항목을 제거한다.
- A.discard(x)는 A의 항목 x를 제거하며 반환 값은 없다.
- A.remove(x)는 discard()와 같지만 x가 없을 경우 KeyError
- A.pop()은 한 항목을 무작위로 제거하고 반환한다. 셋이 비어있을 경우 KeyError

세트(set, 집합)

차집합 (java 는 할 수 있지만 python 은 할 줄 모르는 개발자)

print(java - python) # {'양세형', '김태호'} print(java.difference(python)) # {'양세형', '김태호'}

python 개발자 추가 (기존 개발자 : 박명수, 유재석)

python.add("김태호")

print(python) # {'박명수', '유재석', '김태호'}

java 개발자 삭제 (기존 개발자 : 유재석, 김태호, 양세형)

java.remove("김태호")

print(java) # {'유재석', '양세형'}

사전(dictionary)

- 사전 자료형은 키(Key)와 값(Value)의 쌍을 데이터로 가지는 자료형
- 앞서 다루었던 리스트나 튜플이 값을 순차적으로 저장하는 것과는 대비됨
- 사전 자료형은 키와 값의 쌍을 데이터로 가지며, 원하는 '변경 불가능한 (Immutable) 자료형'을 키로 사용할 수 있음
- 파이썬의 사전 자료형은 해시 테이블(Hash Table)을 이용하므로 데이터의 조회 및 수정에 있어서 O(1)의 시간에 처리할 수 있음
- 사전 자료형에서는 키와 값을 별도로 뽑아내기 위한 메서드를 지원
- 키 데이터만 뽑아서 리스트로 이용할 때는 keys() 함수를 이용
- 값 데이터만을 뽑아서 리스트로 이용할 때는 values() 함수를 이용

사전(dictionary)

- "이름" = "홍길동", "생일" = "몇 월 며칠" 과 같은 대응 관계를 나타낼 수 있는 자료형
- {Key1:Value1, Key2:Value2, Key3:Value3, ...}

```
dic = {'name':'pey', 'phone':'0119993323', 'birth': '1118'}
```

key	value
name	pey
phone	01199993323
birth	1118

사전(dictionary) 추가 삭제

```
a = {1: 'a'}
a[2] = 'b'
print( a)  #{1: 'a', 2: 'b'}'}

a['name'] = 'pey'
print(a) #{1: 'a', 2: 'b', 'name': 'pey'}

a[3] = [1,2,3]
print( a)
#{1: 'a', 2: 'b', 'name': 'pey', 3: [1, 2, 3]}
```

```
del a[1]
print(a)
#{2: 'b', 'name': 'pey', 3: [1, 2, 3]}
```

사전(dictionary) 예제로 접근하기(사물함)

```
cabinet = {3: "유재석", 100: "김태호"}
print(cabinet[3]) # 유재석 -> key 3에 해당하는
value print(cabinet[100]) # 김태호 -> key 100 에 해당하는 value
print(cabinet.get(3)) # 유재석 -> key 3에 해당하는 value
print(cabinet[5]) # key가 5 인 값이 없을 땐 에러 발생 즉, 스크립트종료
print(cabinet.get(5)) # key가 5 인 값이 없을 땐 None 반환 후 에러 없이 스크립트 계속
# 사전 자료형에 값이 있는지 여부 확인
print(3 in cabinet)  # True
print(5 in cabinet)  # False
```

사전(dictionary) 예제로 접근하기(사물함)

```
# key 는 정수형이 아닌 문자열도 가능
cabinet = {"A-3": "유재석", "B-100": "김태호"}
print(cabinet["A-3"]) # 유재석
print(cabinet["B-100"]) # 김태호
# 업데이트 또는 추가
print(cabinet) # {'A-3': '유재석', 'B-100': '김태호'}
cabinet["A-3"] = "김종국" # key 에 해당하는 값이 있는 경우 업데이트
cabinet["C-20"] = "조세호" # key 에 해당하는 값이 없는 경우 신규 추가
print(cabinet) # {'A-3': '김종국', 'B-100': '김태호', 'C-20': '조세호'}
# 삭제
del cabinet["A-3"] # key "A-3" 에 해당하는 데이터 삭제
print(cabinet) # {'B-100': '김태호', 'C-20': '조세호'}
```

사전(dictionary) 예제로 접근하기(사물함)

```
# key 들만 출력
print(cabinet.keys()) # dict_keys(['B-100', 'C-20'])
# value 들만 출력
print(cabinet.values()) # dict_values(['김태호', '조세호'])
# key, value 쌍으로 출력
print(cabinet.items()) # dict_items([('B-100', '김태호'), ('C-20', '조세호')])
# 전체 삭제
cabinet.clear()
print(cabinet) # {}
```

딕셔너리 메서드

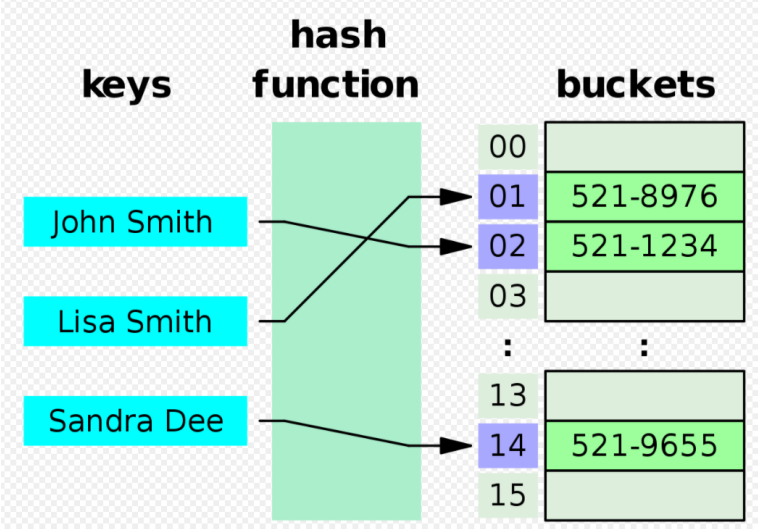
- A.update(B)는 딕셔너리 A에 딕셔너리 B의 키가 존재하면 A의 (키,값)을 B의 (키,값)으로 갱신한다. 존재하지 않으면 (키,값)을 추가한다.
- A.get(key)는 A의 key 값을 반환한다. key가 없으면 아무것도 반환하지 않는다.
- items(), values(), keys()는 딕셔너리 view이다.
- A.pop(key)는 A의 key를 제거한 후 그 값을 반환한다.
- A.popitem()은 A에서 (키,값)을 제거한 후 반환한다.
- A.clear()는 딕셔너리의 모든 항목을 제거한다.

딕셔너리 성능

연산	시간복잡도
복사	O(N)
조회	O(1)
할당	O(1)
삭제	O(N)
멤버십	O(1)
반복	O(N)

사전(dictionary)

딕셔너리는 해시 테이블로 구현되어 있다. 해시 함수는 특정 객체에 해당하는 정수 값을 상수 시간에 계산한다. 정소는 연관 배열의 인덱스로 사용된다.



컬렉션 매핑 타입인 딕셔너리는 iterable하다. in과 len()도 지원한다. 정렬되지 않은 매핑 타입은 임의 순서로 iterate한다. 항목에 접근하는 시간 복잡도 O(1)이며 mutable하다. 또한 삽입 순서를 기억하지 않으며 indexing할 수 없다.

Counter

```
seq1 = [1, 2, 3, 5, 1, 2, 5, 5, 2, 5, 1, 4] #항목 발새우횟수를 매핑
seq_counts = Counter(seq1)
print(seq_counts) # Counter({5: 4, 1: 3, 2: 3, 3: 1, 4: 1})
eq2 = [1, 2, 3]
seq_counts.update(seq2)
print(seq_counts) #Counter({1: 4, 2: 4, 5: 4, 3: 2, 4: 1})

seq3 = [1, 4, 3]
for key in seq3:
    seq_counts[key] += 1
print(seq_counts) #Counter({1: 5, 2: 4, 5: 4, 3: 3, 4: 2})
    '

seq_counts_2 = Counter(seq3)
print(seq_counts_2) #Counter({1: 1, 4: 1, 3: 1})
print(seq_counts + seq_counts_2) #Counter({1: 6, 2: 4, 3: 4, 5: 4, 4: 3})
print(seq_counts - seq_counts_2) #Counter({1: 4, 2: 4, 5: 4, 3: 2, 4: 1})
```

자료구조 변경

```
menu = {"커피", "우유", "주스"}
print(menu, type(menu)) # menu 의 type 정보 : set

menu = list(menu) # 리스트 형태로 변환
print(menu, type(menu)) # menu 의 type 정보 : list

menu = tuple(menu) # 튜플 형태로 변환
print(menu, type(menu)) # menu 의 type 정보 : tuple

menu = set(menu) # 세트 형태로 변환
print(menu, type(menu)) # menu 의 type 정보 : set

info=[("name", "Willow"), ("age",15), ("hobby","nerding")]
info=dict(info)
```

Quiz

- (1,2,3) 튜플에 값 4를 추가하여 (1,2,3,4)를 만들어 출력해 봅시다.

- 딕셔너리 a = dict()가 있다.
  다음 중 오류가 발생하는 경우를 고르고, 그 이유를 설명해 보자.
  1) a['name'] = 'python'
  2) a[('a',)] = 'python'
  3) a[[1]] = 'python'
  4) a[250] = 'python'

- 리스트 a = [1, 1, 1, 2, 2, 3, 3, 3, 4, 4, 5] 에서 중복 숫자를 제거해 보자.

Quiz

- 파이썬은 다음처럼 동일한 값에 여러 개의 변수를 선언할 수 있습니다. 다음과 같이 a, b 변수를 선언한 후 a의 두 번째 요소 값을 변경하면 b 값은 어떻게 될까요? 그리고 이런 결과가 오는 이유에 대해 설명해 보세요.

```
a = b = [1, 2, 3]
a[1] = 4
print(b)
```

해결법: 슬라이싱

```
>>> a = [1,2,3]
>>> b = a[:]
>>> id(a)
4396179528
>>> id(b)
4393788808
>>> a == b
True
>>> a is b
False
>>> b[0] = 5
>>> a
[1, 2, 3]
>>> b
[5, 2, 3]
```

해결법: 슬라이싱?

```
>>> a = [[1,2], [3,4]]
>>> b = a[:]
>>> id(a)
4395624328
>>> id(b)
4396179592
>>> id(a[0])
4396116040
>>> id(b[0])
4396116040
```

```
>>> a[1].append(5)
>>> a
[[8, 9], [3, 4, 5]]
>>> b
[[1, 2], [3, 4, 5]]
>>> id(a[1])
4396389896
>>> id(b[1])
4396389896
```

Copy module 사용

```
>>> import copy
>>> a = [[1,2],[3,4]]
>>> b = copy.copy(a)
>>> a[1].append(5)
>>> a
[[1, 2], [3, 4, 5]]
>>> b
[[1, 2], [3, 4, 5]]
```

```
>>> import copy
>>> a = [[1,2],[3,4]]
>>> b = copy.deepcopy(a)
>>> a[1].append(5)
>>> a
[[1, 2], [3, 4, 5]]
>>> b
[[1, 2], [3, 4]]
```

실습

- 두 단어가 철자의 순서를 바꾸어 놓은 것인지 확인하는 코드블록을 작성하시오
  - 입력: s1 = "marina“, s2 = "aniram"
  - 결과: True

  - 입력: s1 = "google“, s2 = "gouglo"
  - 결과: False

실습

- 파이썬 코딩 대회에서 참석률을 높이기 위해 댓글 이벤트를 진행하기로 하였습니다. 댓글 작성자들 중에 추첨을 통해 1명은 치킨, 3명은 커피 쿠폰을 받게 됩니다. 추첨 프로그램을 작성하시오.
  - 조건1 : 편의상 댓글은 20명이 작성하였고 아이디는 1~20이라고 가정
  - 조건2 : 댓글 내용과 상관없이 무작위로 추첨하되 중복은 불가
  - 조건3 : random 모듈의 shuffle 과 sample 을 활용
  - (출력 예제)

  -- 당첨자 발표 --

  치킨 당첨자 : 1

  커피 당첨자 : [2, 3, 4]

  -- 축하합니다 --

실습(주사위 합의 모든 경로)

- 주사위 2개를 던져서 나오는 모든 경우의 수를 구하려고 합니다.

각각의 주사위의 합이 나오는 경우 수를 구하여 사전형으로 결과를 주는 코드 블록을 작성하시오.

  - 결과 : {2:1,3:2,4:3,5:2,.....}

- 위 예제에서 모든 합의 경로를 사전형으로 주는 코드 블록을 작성하시오.
  - 결과: {2:[1,1],3:[[1,2],[2,1]],4:[1,3],[2,2],[3,1].....}

참고자료(defaultdict)

defaultdict를 활용해 기본값을 'int' 로 선언해주고,

기존에 없던 key를 호출하면 다음과 같이 해당 key가 0으로 자동 초기화된다.

```
>>> from collections import defaultdict
>>> d_dict = defaultdict(int)
>>> d_dict["a"]
0
>>> d_dict
defaultdict(<class 'int'>, {'a': 0})
```

참고자료(defaultdict)

미리 선언하지 않은 key에 값을 더해주는 것도 가능하다.

```
>>> d_dict = defaultdict(int)
>>> d_dict["a"] += 10
10
>>> d_dict
defaultdict(<class 'int'>, {'a': 10})
```

다음과 같이 lambda 식을 사용해 원하는 초기값을 지정할수도 있다.

```
>>> d_dict = defaultdict(lambda: -1)
>>> d_dict["a"]
-1
>>> L_dict = defaultdict(list)
>>> L_dict["a"]
 []
```