List, 리스트

원광대학교의 정교수님은 이번 코딩시험에 대해 10명의 학생의 성적을 기입해야합니다.

정교수님은 프로그래밍을 통해 성적을 기입하라는 학교의 방침에 따라 아래와 같이 작성하였습니다.

```
student_name1, student_score1 = '이익준', 'A'
student_name2, student_score2 = '채송화', 'A+'
student_name3, student_score3 = '안정원', 'B+'
student_name4, student_score4 = '김준완', 'B+'
student_name5, student_score5 = '양석형', 'B'
student_name6, student_score6 = '도재학', 'C'
student_name7, student_score7 = '용석민', 'D'
student_name8, student_score8 = '장겨울', 'A+'
student_name9, student_score9 = '안치홍', 'A+'
student_name10, student_score10 = '추민하', 'F'
```

학생명 하나하나에 이름과 성적을 넣어서 기입하였습니다.

이러고보니 다음 학기에는 100명의 수강생들을 기입해야하는데 변수를 이름과 성적을 100개씩을 기입하려고 하니 관리하기가 어려움을 깨달았습니다.

이러한 경우에 우리는 배열을 사용할 수 있습니다.

C언어와 Java의 경우에는 배열을 사용하지만, 파이썬에는 배열이 없고 이 역할을 리스트와 이후에 배울 튜플이 할 수 있습니다.

리스트와 튜플의 차이점은 수정이 가능한가, 불가능한가에 나뉩니다.

리스트는 수정이 가능(mutable)합니다.

반대로, 튜플은 수정이 불가능(immutable)합니다.

list의 사용법은 아래와 같습니다.

1. [값1, 값2, ...]
2. list()

```
# 1. [값1, 값2, ...]의 방법
list_data = [1,2,3,4,5]

# 2. list()의 방법
# 2-1. 빈 리스트
list_data = list()
list_data.append(1)
list_data.append(2)
list_data.append(3)
list_data.append(4)
list_data.append(5)
print(list_data) # [1, 2, 3, 4, 5]

# 2-2. list('iterable객체')
```

```
list_data = list('12345')
print(list_data) # ['1', '2', '3', '4', '5']
```

엄밀히 말하자면 2-1과 2-2의 생성 결과는 다릅니다.

2-1의 경우에는 배열에 생성된 각 인덱스의 값이 숫자형태입니다.

그러나, 2-2의 경우에는 각 값의 형태가 문자열로 나옵니다.

이를 해결하기 위해서는 map()함수를 써서 사용할 수 있습니다.

```
list_data = list(map(int, '12345'))
print(list_data) # [1, 2, 3, 4, 5]
```

심화1]

> **iterable 객체**
> 값을 차례대로 꺼낼 수 있는 객체입니다.
> 추후 배울 반복문에서 하나하나씩 값을 꺼내 쓸 수 있다면, iterable 객체라고 보시면 됩니다.
> 대표적인 iterable한 자료형은 str, list, tuple, dict, set, bytes, range가 있습니다.

심화2]

> **map함수**
> map(타입, iterable객체)를 넣어서 값을 할당할 수 있습니다.
> map만을 사용해서 값을 받으면 map타입으로 나오게 됩니다.
> 해당 결과는 반복문을 통해 나타낼 수 있지만, 주로 사용하고 하는 값에 넣어 사용합니다.

```
# map 객체 출력
print(map(int, [1,2,3,4,5])) # <map object at 0x0000016C39ECA9C8>

# 추후 배울 반복문을 통해 해당 값 출력
for i in map(int, [1,2,3,4,5]):
    print(i) # 1 2 3 4 5

# List를 이용하여 map 객체를 변환
print(list(map(int, [1,2,3,4,5]))) # [1, 2, 3, 4, 5]
```

그러면 정교수님의 고충을 우리가 해결해드리도록 하겠습니다.

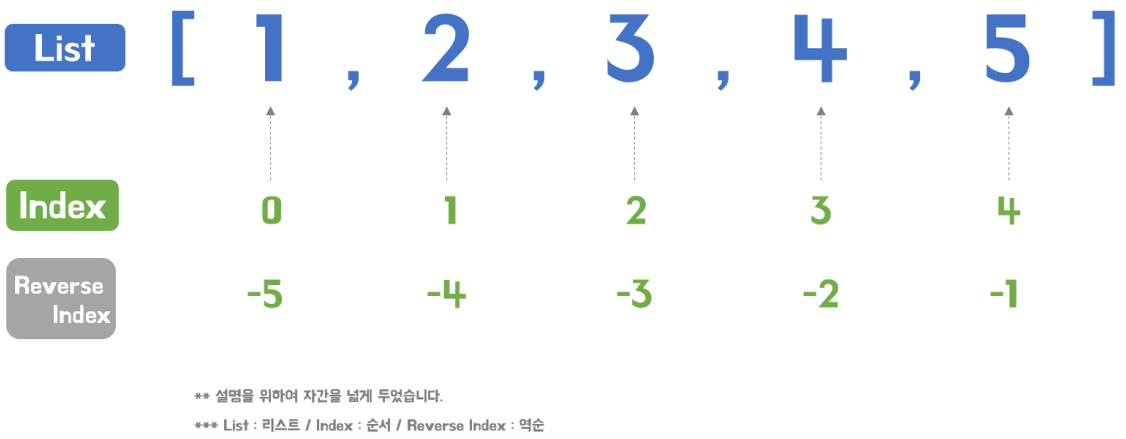
`student_name`이라는 변수에는 학생들의 이름을 담고, `student_score`는 학생들의 성적을 담도록 하겠습니다.

```
student_name = [
    '이익준', '채송화', '안정원', '김준완', '양석형',
    '도재학', '용석민', '장겨울', '안치홍', '추민하'
]

student_score = [
    'A', 'A+', 'B+', 'B+', 'B',
    'C', 'D', 'A+', 'A+', 'F'
]

print(student_name) # ['이익준', '채송화', '안정원', '김준완', '양석형', '도재학', '용석민', '장겨울', '안치홍', '추민하']
print(student_score) # ['A', 'A+', 'B+', 'B+', 'B', 'C', 'D', 'A+', 'A+', 'F']
```

우선 값은 리스트에 담아보았습니다. 해당 값을 출력하는 방법에 대해서는 이후 반복문에서 더 자세히 살펴보도록 하겠습니다.



1. 리스트 연산하기

리스트도 문자열에서 하였듯, `+` 연산자와 `*` 연산자를 사용할 수 있습니다.

```
# 리스트 더하기(+)
list_a = [1, 2, 3]
list_b = [4, 5, 6]

list_c = list_a + list_b
print(list_c) # [1, 2, 3, 4, 5, 6]
```

```
# 리스트 반복하기(*)
list_a = [1, 2, 3].

print(list_a * 3) # [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

2. 리스트 길이 구하기

리스트 길이를 구하기 위해서 `len`함수를 사용하여 길이를 구할 수 있습니다.

len함수는 문자열, 리스트, 튜플, 딕셔너리 등 다양하게 사용될 수 있습니다.

```
list_data = [1, 2, 3]

print(len(list_data)) # 3
```

```
list_data = [1, 2, 3]
str_data = '이렇게는 더할 수 없어요'

print(list_data + str_data) # TypeError: can only concatenate list (not "str")
to list
```

3. 리스트 관련 함수들

1) 리스트 수정

해당 부분은 함수는 아니지만, 함께 다루도록 하겠습니다.

리스트는 수정이 가능(mutable)하기에 인덱스로 접근하여 해당 값만을 변경할 수 있습니다.

```
list_data = [1, 4, 3]
print(list_data) # [1, 4, 3]
list_data[1] = 2
print(list_data) # [1, 2, 3]
```

2) 리스트 삭제

- del 함수

del함수를 통하여 해당 리스트 요소나 객체를 삭제할 수 있습니다.

`del list_data[n]`는 list_data에 n번째 요소값을 삭제합니다.

```
list_data = [1, 2, 3]

del list_data[1]

print(list_data) # [1,3]

del list_data

print(list_data) # NameError: name 'list_data' is not defined
```

- remove 함수

remove함수를 통하여 해당 값을 삭제할 수 있습니다.

`list_data.remove(값)`을 사용하면 가장 앞에 나오는 해당 값을 지웁니다.

```
list_data = [1, 2, 3, 2, 3]
list_data.remove(3)

print(list_data) # [1, 2, 2, 3]
```

- pop 함수

다른 방식으로는 정확하게 삭제는 아니지만, 삭제의 효과를 내는 방법이 있습니다.

리스트의 값을 꺼내옴으로서, 해당 값을 지우는 효과를 낼수도 있습니다.

`list_data.pop(index값)`을 사용하면 해당 인덱스의 값을 꺼내올 수 있습니다.

```
list_data = [1, 2, 3, 2, 3]
value = list_data.pop(2) # 2번째 인덱스의 값을 꺼내옴

print(list_data) # [1, 2, 2, 3]
print('꺼내온 값 :', value) # 꺼내온 값 : 3
```

3) 리스트 추가

- append

리스트에 값을 추가하는 방식은 `+` 연산자를 사용하는 방식도 있었지만,

정식적인 방법은 `append`함수를 사용하는 것입니다.

`list_data.append(해당값)`을 사용하여 해당 리스트에 값을 넣어줄 수 있습니다.

넣어진 값은 가장 뒤에 추가됩니다.

```
list_data = [1, 2, 3]
list_data.append(4)
list_data.append(5)

print(list_data) # [1, 2, 3, 4, 5]
```

현재까지 배운 리스트는 1차 리스트입니다. 이중 리스트와 n차 리스트도 사용가능합니다.

리스트 안에 리스트를 넣어서 사용하는 방법입니다.

```
# 이중 리스트 예시
double_list = [[1,2,3], [4,5,6], [7,8,9]]
print(double_list) # [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# 이중 리스트 접근
```

```
print(double_list[0][2]) # 3

# 위와 같은 방식으로 3x3의 이중 리스트를 생성할 수도 있고,

list_data = [1, 2, 3, [1, 2, 3], 6]
print(list_data) # [1, 2, 3, [1, 2, 3], 6]
# 이러한 방식으로 1차 리스트와 2차 리스트를 함께 사용할 수도 있습니다.

# 2차 리스트를 추가하는 방법
list_data = [1, 2, 3]
list_data.append([1, 2, 3])
print(list_data) # [1, 2, 3, [1, 2, 3]]
```

- insert

append는 리스트의 가장 뒤에 값이 추가됩니다.

우리가 값을 추가할 때, 뒤에만이 아닌 중간에 추가해야하는 경우도 있습니다.

이러한 경우에 `insert`를 사용할 수 있습니다.

`list_data.insert(index, 값)`으로 사용할 수 있습니다.

index 위치를 적어주고, 해당 값을 적어서 해당 위치에 값을 추가할 수 있습니다. 그 위치 이후에 있던 값들은 한 인덱스씩 +1이 됩니다.

```
list_data = [1,2,4]
list_data.insert(2, 3) # 2번 인덱스에 3의 값 추가

print(list_data) # [1, 2, 3, 4]
```

4) 리스트 정렬

- sort, 정렬

리스트의 요소를 정렬해 줍니다.

```
list_data = [1, 5, 2, 3, 4]
list_data.sort()

print(list_data) # [1, 2, 3, 4, 5]

list_data.sort(reverse=True)
print(list_data) # [5, 4, 3, 2, 1]
```

- reverse, 리스트 뒤집기

리스트를 역순으로 뒤집어 줍니다.

```
list_data = [1, 5, 2, 3, 4]
list_data.reverse()

print(list_data) # [4, 3, 2, 5, 1]
```

5) 위치 반환

- index, 인덱스 반환

`list_data.index(x)`는 리스트 list_data에서 x의 값의 인덱스를 반환해줍니다.

리스트에서 x의 값이 여러개일 경우에는 가장 앞에 있는 x의 값의 인덱스가 반환됩니다.

```
list_data = [1, 3, 3, 7, 6]
idx = list_data.index(7)
print(idx) # 3

print(list_data[idx]) # 7
```

6) Count

- count, 리스트에 포함된 x의 총 개수 반환

`list_data.count(x)`는 리스트 list_data에서 x가 나온 총 갯수를 반환해줍니다.

```
list_data = [1, 3, 3, 7, 6]
value = list_data.count(3)
print(value) # 2

value = list_data.count(7)
print(value) # 1
```

7) extend, 리스트 확장

- extend

extend는 리스트를 확장해주는 역할을 합니다.

list + list의 결과를 나타낸다고 생각하시면 됩니다.

`list_data1.extend(list_data2)`로 사용합니다.

```
list_data1 = [1, 2, 3]
list_data2 = [4, 5, 6]
list_data1.extend(list_data2)
print(list_data1) # [1, 2, 3, 4, 5, 6]

# 이중리스트일 경우에도 되는지 확인
list_data1 = [1, 2, 3, [1, 2, 3]]
list_data2 = [4, 5, 6, [4, 5, 6]]
```

```
list_data1.extend(list_data2)
print(list_data1) # [1, 2, 3, [1, 2, 3], 4, 5, 6, [4, 5, 6]]

# `+`연산 사용
list_data1 = [1, 2, 3]
list_data2 = [4, 5, 6]
list_data = list_data1 + list_data2
print(list_data) # [1, 2, 3, 4, 5, 6]

# `+`연산 이중리스트 경우 확인
list_data1 = [1, 2, 3, [1, 2, 3]]
list_data2 = [4, 5, 6, [4, 5, 6]]
list_data = list_data1 + list_data2
print(list_data) # [1, 2, 3, [1, 2, 3], 4, 5, 6, [4, 5, 6]]
```

`extend`함수를 사용한 결과에는 값이 반환되지 않고 바로 적용되는 모습을 보이고,

`+`연산자의 경우에는 값을 다른 변수에 넣어서 출력하는 모습을 확인하실 수 있습니다.

추후 배울 **함수**에 대한 내용이지만, 조금 알고 넘어가자면

함수에는 `return`을 통해 값을 전달하는 방식과 넘겨받은 값을 수정하여 반환하진 않는 방식으로 나뉩니다.

자세한 내용은 함수에서 다루도록 하겠습니다.