# [ 1-4. 기본 문법 및 데이터 타입 소개2 ]

## 4. 리스트(List)

리스트(List)는 여러 값을 순서대로 저장하는 데 사용되는 자료형입니다. 리스트는 다양한 데이터 타입의 값들을 포함할 수 있으며, 변경 가능(mutable)하여 리스트 내의 값을 추가, 삭제, 또는 변경할 수 있습니다.

### 4-1. 리스트 기본 사용방법

리스트는 대괄호 "[]" 안에 값들을 쉼표로 구분하여 나열함으로써 생성할 수 있습니다.

In [ ]:
```
my_list = [1, 2, 3]
names = ["Alice", "Bob", "Charlie"]
mixed = [1, "Alice", True, 2.34]
```

리스트의 값에 접근하기 위해서는 인덱스를 사용합니다. 인덱스는 0부터 시작합니다.

In [ ]:
```
print(names[0])
```

```
Alice
```

In [ ]:
```
print(mixed[3])
```

```
2.34
```

In [ ]:
```
print(names[-1])
```

```
Charlie
```

append() 메서드 : 리스트 맨 뒤에 값 추가

In [ ]:
```
names.append("Diana")
print(names)
```

```
['Alice', 'Bob', 'Charlie', 'Diana']
```

insert() 메서드 : 특정 위치에 값 삽입

In [ ]:
```
names.insert(1, "Eve")
print(names)
```

```
['Alice', 'Eve', 'Bob', 'Charlie', 'Diana']
```

extend() 메서드 : 튜플이나 리스트 등의 여러 값을 한번에 삽입

In [ ]:
```
names.extend(["Alice", "Bob"])
print(names)
```

```
['Alice', 'Eve', 'Bob', 'Charlie', 'Diana', 'Alice', 'Bob']
```

index(찾고자 하는 값 [, 시작점] [, 종결점]) 메서드 : 값의 위치 확인(처음 찾은 결과만 반환)

In [ ]:
```
names.index("Alice")
```

Out[ ]:
```
0
```

In [ ]:
```
names.index("Alice", 1)
```

Out[ ]:
```
5
```

In [ ]:
```
names.index("Bob", 3, 7)
```

Out[ ]:
```
6
```

count() 메서드 : 해당 값의 개수 반환

In [ ]:
```
names.count("Alice")
```

Out[ ]:
```
2
```

pop([위치]]) 메서드 : 인자를 생략하면 가장 뒤의 값을 추출하며, 위치 인자를 입력하면 해당 위치의 값을 추출합니다.

In [ ]:
```
print(names)
```

```
['Alice', 'Eve', 'Bob', 'Charlie', 'Diana', 'Alice', 'Bob']
```

In [ ]:
```
poppedval = names.pop()
print(names)
print(poppedval)
```

```
['Alice', 'Eve', 'Bob', 'Charlie', 'Diana', 'Alice']
Bob
```

In [ ]:
```
poppedval = names.pop(3)
print(names)
print(poppedval)
```

```
['Alice', 'Eve', 'Bob', 'Diana', 'Alice']
Charlie
```

sort() 메서드 : 리스트 값들을 정렬

In [ ]:
```
names.sort()
print(names)
```

```
['Alice', 'Alice', 'Bob', 'Diana', 'Eve']
```

In [ ]:
```
# reverse 인자 설정을 통해 역순으로 정렬할 수 있음.
names.sort(reverse=True)
print(names)
```

```
['Eve', 'Diana', 'Bob', 'Alice', 'Alice']
```

[참고] 메서드(method)와 속성(attribute)

- 메서드(method)는 객체에 속한 함수로, 객체의 데이터를 조작하거나 특정 작업을 수행하는 데 사용됩니다.
- 속성은 객체에 속한 변수로, 객체의 상태를 나타내는 데이터입니다.

=> 메서드는 객체의 동작을 정의하며, '객체.메서드()' 형식으로 호출됩니다. 속성에는 직접 접근할 수 있으며, '객체.속성' 형식으로 사용됩니다.

## 4-2. 리스트 슬라이싱

리스트 슬라이싱은 파이썬에서 리스트의 일부분을 선택하여 새로운 리스트를 생성하는 기능입니다. 슬라이싱은 "리스트[시작 인덱스 : 끝 인덱스 : 간격]" 형태로 사용됩니다.

- 시작 인덱스 : 슬라이싱할 범위의 시작점입니다(포함). 생략하면 리스트의 시작부터 슬라이싱합니다.
- 끝 인덱스 : 슬라이싱할 범위의 끝점입니다(미포함). 생략하면 리스트의 끝까지 슬라이싱합니다.
- 간격 : 선택적으로, 슬라이싱할 요소들 사이의 간격을 지정합니다. 생략하면 기본값은 "1"입니다.

In [ ]:
```
my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# 전체 리스트 슬라이싱
print(my_list[:])  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# 시작 인덱스 2부터 끝 인덱스 5까지 슬라이싱 (5는 포함되지 않음)
print(my_list[2:5])  # [2, 3, 4]

# 리스트의 시작부터 끝 인덱스 3까지 슬라이싱
print(my_list[:4])  # [0, 1, 2, 3]

# 시작 인덱스 3부터 리스트 끝까지 슬라이싱
print(my_list[3:])  # [3, 4, 5, 6, 7, 8, 9]

# 간격을 2로 설정하여 슬라이싱
print(my_list[::2])  # [0, 2, 4, 6, 8]

# 리스트를 거꾸로 슬라이싱
print(my_list[::-1])  # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 3, 4]
[0, 1, 2, 3]
[3, 4, 5, 6, 7, 8, 9]
[0, 2, 4, 6, 8]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

## 4-3. 리스트 컴프리헨션(List Comprehension)

리스트 컴프리헨션(List Comprehension)은 파이썬에서 리스트를 생성하는 간결하고 효율적인 방법입니다. 이 방식은 기존 리스트를 사용해 새로운 리스트를 만들 때, 반복문과 조건문을 한 줄의 코드로 표현할 수 있게 해줍니다. 리스트 컴프리헨션은 코드의 가독성을 높이고, 길이가 긴 반복문 코드를 단축시켜 줍니다.

### [기본 내용]

리스트 컴프리헨션의 기본 구조는 다음과 같습니다.

**[표현식 for 항목 in 반복가능객체 if 조건문]**

- 표현식 : 새로운 리스트의 각 요소에 대한 연산이나 처리를 정의합니다.
- for 항목 in 반복가능객체 : 기존 리스트(또는 반복 가능한 다른 객체)를 순회하는 반복문입니다.
- if조건문 : (선택적)조건에 맞는 요소에 대해서만 표현식을 적용합니다.

In [ ]:
```
# 0부터 9까지의 숫자 중에 대해서 제곱을 구하여 새로운 리스트 생성
squares = [x**2 for x in range(10)]
print(squares)  # 출력: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In [ ]:
```
# 0부터 9까지의 숫자 중에서 짝수의 제곱을 구하여 새로운 리스트 생성
squares = [x**2 for x in range(10) if x % 2 == 0]
print(squares)  # 출력: [0, 4, 16, 36, 64]
```

```
[0, 4, 16, 36, 64]
```

In [ ]:
```
# 두 리스트의 모든 조합을 포함하는 새로운 리스트 생성
pairs = [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
print(pairs)  # 출력: [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3,
```

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

[장점]

- 간결성 : 여러줄의 코드를 한 줄로 줄일 수 있어 코드가 더 간결해 집니다.
- 표현력 : 리스트를 생성하는 로직을 명확하게 표현할 수 있습니다.
- 성능 : 일반적인 for 반복문에 비해 리스트 컴프리헨션의 실행속도가 더 빠를 수 있습니다.

[for문과의 비교]

In [ ]:
```
# for문을 사용하여 제곱값 구하기
squares = []  # 빈 리스트를 생성
for x in range(10):  # 0부터 9까지 숫자에 대해서 반복
    squares.append(x**2)  # 현재 숫자의 제곱을 squares 리스트에 추가
print(squares)  # 출력: [0, 4, 16, 36, 64, 81]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

for문을 사용하면 리스트 컴프리헨션에 비해 코드가 조금 더 길어지지만, 코드의 작동 방식을 좀 더 명확하게 이해할 수 있습니다. 또한 복잡한 로직을 처리할 때는 for문을 사용하는 것이 더 적합할 수 있습니다.

[참고] range함수

range() 함수는 파이썬에서 숫자 시퀀스를 생성하는데 사용됩니다.

- range(종료값) : 0부터 시작하여 종료값 바로 앞까지의 숫자를 포함하는 시퀀스를 생성합니다.(종료값 미포함)
- range(시작값, 종료값) : 시작값에서 시작하여 종료값 바로 앞까지의 숫자를 포함하는 시퀀스를 생성합니다.(종료값 미포함)
- range(시작값, 종료값, 증가값) : 시작값에서 시작하여 증가값만큼 숫자를 증가시키면서, 종료값 바로 앞까지의 숫자를 포함하는 시퀀스를 생성합니다. 증가값으로 음수를 지정하면 숫자가 감소합니다.

=> 주의사항 : range() 함수는 실제로 리스트를 생성하지 않습니다. 대신, 지정된 범위에 따라 숫자를 순차적으로 생성하는 range 객체를 반환합니다. 따라서 메모리를 효율적으로 사용할 수 있습니다. 실제 리스트가 필요한 경우 list()함수를 사용하여 결과를 리스트로 변환할 수 있습니다.

# 5. 튜플(Tuple)

튜플(Tuple)은 파이썬의 기본 자료형 중 하나로, 여러 값을 순서대로 저장하는데 사용되는 자료형 입니다. 리스트와 유사하게 데이터를 순서대로 저장하지만, 튜플은 한 번 생성되면 그 안에 저장된 데이터를 변경할 수 없습니다. 이러한 특성 때문에 튜플은 프로그램 실행 도중 값이 변경되지 않아야 하는 데이터를 저장하는 데 적합합니다.

## 5-1. 튜플 특징

- 불변성(Immutability) : 튜플에 저장된 데이터는 변경할 수 없습니다. 이는 튜플의 값을 추가, 삭제, 변경할 수 없다는 의미입니다. 이는 데이터가 의도치 않게 변경되는 것을 방지하고 프로그램의 안정성을 높이는 데 도움을 줍니다.
- 순서가 있음(Ordered) : 튜플의 값들은 순서를 가지고 있어, 인덱싱이나 슬라이싱을 통해 접근할 수 있습니다.
- 중복 허용 : 튜플 내에서 같은 값을 여러개 가질 수 있습니다.
- 읽기 전용 : 읽기 전용으로 제공되는 메서드는 count(), index() 정도뿐으로 적습니다. 하지만 기능이 적은만큼 리스트에 비해 속도가 빠르고 적은 메모리를 사용합니다.

## 5-2. 튜플의 사용

In [ ]:
```
my_tuple = (1, 2, 3)
single_element_tuple = (42,) # 값이 하나만 있는 튜플으르 생성하는 경우 뒤에 쉼표를 붙이
```

In [ ]:
```
print(my_tuple, single_element_tuple)
```

```
(1, 2, 3) (42,)
```

## 5-3. 파이썬 내부에서 튜플 사용

파이썬 내부에서는 알게 모르게 튜플이 사용되는 경우가 많습니다.

### 5-3-1. 함수의 반환 값

파이썬에서 함수는 여러 값을 튜플로 묶어 반환할 수 있습니다. 이는 하나의 함수 호출로 여러 값을 반환받고 싶을 때 사용합니다.

In [ ]:
```
def min_max(values):
    return min(values), max(values)
minimum, maximum = min_max([1, 2, 3, 4, 5])
print(minimum, maximum)
```

```
1 5
```

### 5-3-2. 패킹 및 언패킹

여러 변수에 동시에 값을 할당하는 과정에서 튜플이 내부적으로 사용됩니다. 이를 튜플 패킹 및 언패킹이라고 합니다.

In [ ]:
```
a, b = 1, 2  # 튜플 패킹 및 언패킹
print(f"value a : {a}")
print(f"value b : {b}")
```

```
value a : 1
value b : 2
```

### 5-3-3. for 루프와 함께 사용

'enumerate' 함수나 'dict.items()' 메서드와 같이 튜플을 반환하는 함수와 함께 for 루프에서 튜플 언패킹을 사용할 수 있습니다.

In [ ]:
```
for index, value in enumerate(['a', 'b', 'c']):
    print(index, value)
```

```
0 a
1 b
2 c
```

# 6. 세트(Set)

세트(Set)는 순서가 없고 중복된 요소를 갖지 않는 값의 나열입니다. 세트는 수학의 집합 개념을 컴퓨터 과학에 도입한 것으로, 고유한 요소의 모음으로 생각할 수 있습니다. 세트는 중괄호 "{}" 또는 "set()" 함수를 사용하여 생성할 수 있습니다.

## 6-1. 세트 특징

- 중복 없음 : 세트에는 같은 값이 두 번 포함될 수 없습니다. 이 특성은 데이터의 중복을 제거하는 데 유용합니다.
- 순서 없음 : 세트는 요소 간 순서를 유지하지 않습니다. 따라서, 인덱스로 개별 요소에 접근할 수 없습니다.
- 가변성 : 세트는 요소를 추가하거나 제거할 수 있으며, 이를 통해 동적으로 세트의 내용을 변경할 수 있습니다.

## 6-2. 세트의 사용

In [ ]:
```
# 중괄호를 사용한 세트 생성
my_set = {1, 2, 3, 4, 5}

# set() 함수를 사용한 세트 생성
second_set = set([1, 2, 3, 4, 5])

print(my_set)
print(second_set)
```

```
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
```

In [ ]:
```python
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}

# 합집합
print(a | b)  # {1, 2, 3, 4, 5, 6}

# 교집합
print(a & b)  # {3, 4}

# 차집합
print(a - b)  # {1, 2}

# 대칭 차집합 (합집합 - 교집합)
print(a ^ b)  # {1, 2, 5, 6}
```

```
{1, 2, 3, 4, 5, 6}
{3, 4}
{1, 2}
{1, 2, 5, 6}
```

## 6-3. 세트의 메서드

- add() 메서드 : 세트에 값을 추가합니다.

In [ ]:
```python
fruits = {"apple", "banana", "cherry"}
fruits.add("orange")
print(fruits)  # {'cherry', 'banana', 'orange', 'apple'}
```

```
{'orange', 'banana', 'apple', 'cherry'}
```

- remove() 메서드 : 세트에서 값을 제거합니다. 값이 세트에 없는 경우 'KeyError'를 발생시킵니다.

In [ ]:
```python
fruits.remove("banana")
print(fruits)  # {'cherry', 'orange', 'apple'}
```

```
{'orange', 'apple', 'cherry'}
```

- discard() 메서드 : 세트에서 값을 제거합니다. 값이 세트에 없어도 에러를 발생시키지 않습니다.

In [ ]:
```python
fruits.discard("banana")
print(fruits)  # {'cherry', 'orange', 'apple'}
```

```
{'orange', 'apple', 'cherry'}
```

- clear() 메서드 : 세트의 모든 요소를 제거합니다.

In [ ]:
```
fruits.clear()
print(fruits)  # set()
```

```
set()
```

- pop() 메서드 : 세트에서 하나의 요소를 무작위로 제거하고 그 요소를 반환합니다. 세트가 비어 있을 경우 'KeyError'가 발생합니다.

In [ ]:
```
fruits = {"apple", "banana", "cherry"}
removed_fruit = fruits.pop()
print(removed_fruit)  # 'cherry', 'orange', 'apple' 중 하나
print(fruits)  # 제거된 요소를 제외한 나머지
```

```
banana
{'apple', 'cherry'}
```

- union() 메서드 : 두 세트의 합집합을 반환합니다.
- update() 메서드 : 기존 세트를 두 세트의 합집합으로 갱신합니다.

In [ ]:
```
a = {1, 2, 3}
b = {3, 4, 5}
c = a.union(b)
print(c)  # {1, 2, 3, 4, 5}
a.update(b)
print(a)  # {1, 2, 3, 4, 5}
```

```
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
```

# 7. 딕셔너리(Dictionary)

딕셔너리(Dictionary), 또는 딕셔너리는 파이썬의 핵심 데이터 구조 중 하나로, 키(Key)와 값(Value)의 쌍으로 데이터를 저장합니다. 딕셔너리는 "{}"를 사용하여 생성하며, 각 키와 값은 "Key: Value" 형태로 표현됩니다.

## 7-1. 딕셔너리의 특징

- 변경 가능 : 딕셔너리는 가변 객체로, 저장된 데이터를 변경할 수 있습니다.
- 순서 없음 : 딕셔너리는 기본적으로 순서에 의존하지 않는 데이터 구조입니다. (파이썬 3.7 이전까지는 요소의 순서를 보장하지 않았습니다. 파이썬 3.7 이후부터는 삽입 순서를 유지합니다.)
- 키의 유일성 : 딕셔너리에서 각 키는 유일해야 하며, 한 키에 하나의 값만 매핑됩니다. 같은 키에 다른 값을 할당하면 이전 값이 덮어씌워집니다.
- 다양한 데이터 타입 : 딕셔너리의 키로는 변경 불가능(immutable)한 데이터 타입만 사용할 수 있습니다(문자열, 숫자, 튜플 등). 값으로는 어떤 데이터 타입도 사용할 수 있습니다.

## 7-2. 딕셔너리 사용 예시

In [ ]:
```python
# 딕셔너리 생성
my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}

# 값 접근
print(my_dict['name'])  # Alice

# 값 추가 및 변경
my_dict['email'] = 'alice@example.com'  # 새로운 키-값 쌍 추가
my_dict['age'] = 31  # 기존 키에 대한 값 변경

# 값 제거
del my_dict['city']  # 키와 그 값을 삭제
my_dict.pop('age')  # 'age' 키와 값을 삭제하고 값을 반환

# 키 존재 확인
'city' in my_dict  # False

# 모든 키와 값 순회
for key, value in my_dict.items():
    print(key, value)
```

```
Alice
name Alice
email alice@example.com
```

## 7-3. 딕셔너리의 메서드

- keys() 메서드 : 딕셔너리의 모든 키를 반환합니다. 반환된 키 값들은 리스트와 비슷하지만, 리스트의 메서드를 사용할 수는 없습니다.

In [ ]:
```python
person = {'name': 'Alice', 'age': 30, 'city': 'New York'}
keys = person.keys()
print(keys)  # dict_keys(['name', 'age', 'city'])
```

```
dict_keys(['name', 'age', 'city'])
```

- values() 메서드 : 딕셔너리의 모든 값을 반환합니다.

In [ ]:
```
values = person.values()
print(values)  # dict_values(['Alice', 30, 'New York'])
```

```
dict_values(['Alice', 30, 'New York'])
```

- items() 메서드 : 딕셔너리의 모든 키-값 쌍을 튜플로 묶어 반환합니다. 이 메서드는 딕셔너리를 순회할 때 매우 유용합니다.

In [ ]:
```
items = person.items()
print(items)  # dict_items([('name', 'Alice'), ('age', 30), ('city', 'New
```

```
dict_items([('name', 'Alice'), ('age', 30), ('city', 'New York')])
```

- get(key[, default]) 메서드 : 지정한 키에 대한 값을 반환합니다. 키가 딕셔너리에 없는 경우 'None'이나, 'default' 인자로 전달된 기본값을 반환합니다.

In [ ]:
```
email = person.get('email', 'Not Available')
print(email)  # Not Available
```

```
Not Available
```

- pop(key[, default]) 메서드 : 지정한 키에 대한 값을 반환하고, 해당 키-값 쌍을 딕셔너리에서 제거합니다. 키가 없는 경우 'default'를 반환하며, 'default'가 제공되지 않고 키가 없는 경우 'KeyError'를 발생시킵니다.

In [ ]:
```
age = person.pop('age')
print(age)  # 30
print(person)  # {'name': 'Alice', 'city': 'New York'}
```

```
30
{'name': 'Alice', 'city': 'New York'}
```

- update([other]) 메서드 : 다른 딕셔너리 'other'의 키-값 쌍으로 현재 딕셔너리를 갱신합니다. 'other'에 같은 키가 있으면 해당 값으로 덮어씁니다.

In [ ]:
```
person.update({'age': 31, 'email': 'alice@example.com'})
print(person)  # {'name': 'Alice', 'city': 'New York', 'age': 31, 'email'
```

```
{'name': 'Alice', 'city': 'New York', 'age': 31, 'email': 'alice@example.
com'}
```

# 8. 부울(Bool)

파이썬에서 'bool' 타입은 정수의 하위 유형으로, 논리적 값인 'True' 또는 'False'를 나타냅니다. 이 값들은 조건문과 반복문에서 표현식의 참 또는 거짓 값을 나타내는 데 사용되어, 논리 연산을 수행하는 데 쓰입니다.

파이선에서 'bool'은 내장 타입이며, 모든 객체는 참 거짓 값을 가집니다. 'if'나 'while' 조건문 또는 논리 연산의 피연산자로 사용될 수 있습니다.

## 8-1. 'bool'에 대한 주요 내용

- 'True'와 'False' : 'bool' 타입의 유일한 두 가지 값입니다.
- 참 거짓 값 판단 : 파이썬은 값이 참인지 거짓인지를 자동으로 평가합니다. 예를 들어, '0', 'None', '[]', '{}', '""'은 'False'로 간주되며, 0이 아닌 숫자, 비어 있지 않은 문자열, 리스트, 튜플, 세트, 딕셔너리는 'True'로 간주됩니다.
- 정수에서 파생됨 : 'True'는 '1'과 동등하고, 'False'는 '0'과 동등합니다.

## 8-2. 'bool' 사용 예시

In [ ]:
```
boolval = False
print(type(boolval))
```

```
<class 'bool'>
```

In [ ]:
```
# 비교연산의 결과로 'bool' 값이 표시됨.
print(1 < 2)
print(1 > 2)
print(1 != 2)
print(1 == 2)
```

```
True
False
True
False
```

In [ ]:
```
# 논리연산의 결과로 'bool' 값이 표시됨.
print(True and False)
print(True & False)
print(True or False)
print(True | False)
print(not True)
print(not False)
```

```
False
False
True
True
False
True
```