[1-6. 제어문(if, for 등)]

1. if 문

'if'문은 조건문이며, 특정 조건이 충족될 때 코드 블록을 실행하기 위해 사용됩니다. 'if'문은 주어진 조건이 참('True')이면 해당 코드 블록을 실행하고, 거짓('False')이면 코드 블록을 건너뜁니다.

선택적으로 'else' 절을 포함하여 조건이 거짓일 때 실행할 코드를 정의할 수 있으며, 여러 조건을 체크하려면 'elif'를 사용합니다.

'if'문의 선언 마지막에 콜론(:)을 표기하며, 코드 블록은 들여쓰기하여 구분합니다.

```
if 조건문:
    실행할 코드 블록
elif 조건문:
    실행할 코드 블록
else:
    실행할 코드 블록
```

- 'if'문 사용 예시

In [ ]:
```
number = 10
if number > 0:
    print("number는 0보다 큽니다.")
print("이 문장은 항상 출력됩니다.")
```

```
number는 0보다 큽니다.
이 문장은 항상 출력됩니다.
```

- 'else' 절 사용

In [ ]:
```
number = -10
if number > 0:
    print("number는 0보다 큽니다.")
else:
    print("number는 0보다 크지 않습니다.")
```

```
number는 0보다 크지 않습니다.
```

- 'elif'절 사용

In [ ]:
```
number = 0
if number > 0:
    print("number는 0보다 큽니다.")
elif number < 0:
    print("number는 0보다 작습니다.")
else:
    print("number는 0보다 크거나 작지 않습니다.")
```

```
number는 0보다 크거나 작지 않습니다.
```

2. while문

'while'문은 조건이 참(True)인 동안 반복적으로 코드 블록을 실행하는 제어 구문입니다. 조건이 거짓(False)이 되면 반복이 중지됩니다.

이를 통해 특정 조건을 만족할 때까지 반복 작업을 수행할 수 있습니다.

```
while 조건문:
    실행할 코드 블록
```

In [ ]:
```
count = 0
while count < 5:
    print("Inside loop")
    count += 1
print("Outside loop")
```

```
Inside loop
Inside loop
Inside loop
Inside loop
Inside loop
Outside loop
```

3. for문

'for'문은 반복 작업을 수행할 때 사용되는 제어문입니다. 'for'문은 주어진 시퀀스(리스트, 튜플, 문자열 등)의 각 요소를 반복적으로 실행하면서 코드 블록을 실행합니다.

3-1. 'for'문 구조

일반적인 'for'문의 구조는 다음과 같습니다.

```
for 요소 in 시퀀스:
    실행할 코드 블록
```

- 시퀀스 : 반복할 요소들을 담고 있는 데이터 구조
- 요소 : 각 반복마다 현재 처리 중인 항목
- 코드 블록 : 반복 작업을 수행하는 코드

3-2. 사용 예시

In [ ]:
```python
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

```
apple
banana
cherry
```

In [ ]:
```python
for i in range(1, 6):
    print(i)
```

```
1
2
3
4
5
```

In [ ]:
```python
student_scores = {
    "John": 85,
    "Emma": 92,
    "David": 78,
    "Sophia": 95
}

# 딕셔너리의 각 키-값 쌍을 출력하는 예시
for name, score in student_scores.items():
    print(f"{name}: {score}")
```

```
John: 85
Emma: 92
David: 78
Sophia: 95
```

- 딕셔너리.items() 메서드는 딕셔너리의 키-값 쌍을 포함하는 'dict_items' 객체를 반환합니다. 이 객체는 'for' 문과 함께 사용하여 딕셔너리의 각 키-값 쌍에 대해 반복할 수 있습니다.

4. break, continue, else

반복문에서 제어 흐름을 조절하는데 사용되는 특수한 키워드입니다.

4-1. break

'break'문은 반복문을 중단시키는 데 사용됩니다. 반복문 안에서 'break'문을 만나면 반복문 전체가 즉시 종료되고 해당 반복문 뒤에 오는 코드가 실행됩니다.

일반적으로 조건에 따라 반복문을 종료하는데 사용됩니다.

In [ ]:
```
for i in range(10):
    print(i)
    if i == 5:
        break
```

```
0
1
2
3
4
5
```

4-2. continue

'continue'문은 반복문의 현재 반복을 중단하고 다음 반복으르 시작하는데 사용됩니다. 즉, 'continue'문을 만나면 반복문의 나머지 부분은 실행되지 않고 다음 반복이 시작됩니다.

In [ ]:
```
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
```

```
1
3
5
7
9
```

4-3. else

반복문의 끝에 'else'문을 사용할 수 있습니다. 이 'else' 블록은 반복문이 정상적으로 종료된 경우에 실행됩니다. 즉, 반복문이 'break'문으로 인해 중단되지 않고 정상적으로 실행이 완료되면 'else' 블록이 실행됩니다.

예를 들어, 다음 코드는 반복문이 0부터 4까지의 숫자를 출력하고, 반복이 정상적으로 종료되면 "반복문이 종료되었습니다."를 출력합니다. 그러나 5가 출력되면 'break'문에 의해 반복문이 종료되고, 'else' 블록은 실행되지 않습니다.

In [ ]:
```
for i in range(5):
    print(i)
else:
    print("반복문이 종료되었습니다.")
```

```
0
1
2
3
4
반복문이 종료되었습니다.
```

5. lambda 함수

람다(lambda) 함수는 파이썬에서 간단한 함수를 한 줄로 작성할 때 사용되는 익명 함수입니다. 람다 함수는 이름이 없고 일회성으로 사용되며, 주로 간단한 연산이나 함수를 정의할 때 사용됩니다.

람다 함수의 일반적인 구문구조는 다음과 같습니다.

```
lambda 매개변수들: 표현식
```

- lambda : 람다 함수를 정의하는 키워드입니다.
- 매개변수들 : 람다 함수의 입력 매개변수들을 나타냅니다.
- 표현식 : 람다 함수가 실행될 때 반환될 값입니다.

In [ ]:
```
add = lambda x, y: x + y
print(add(3, 5))  # 출력: 8
```

6. enumerate 함수

'enumerate()' 함수는 반복문에서 빈번하게 사용되는 함수로 반복 가능한(iterable) 객체(리스트, 튜플, 문자열 등)를 받아 인덱스(index)와 해당 요소(element)를 순회하는 데 사용됩니다.

이 함수는 각 요소와 해당 인덱스를 튜플 형태로 반환합니다.

'enumerate()'함수의 일반적인 구문구조는 다음과 같습니다.

```
enumerate(iterable, start=0)
```

- iterable : 인덱스와 요소를 순회할 반복 가능한 객체입니다.
- start : 선택적 매개변수로, 인덱스를 시작할 값입니다. 기본값은 0입니다.

6-1. 예시 1

In [ ]:
```
fruits = ['apple', 'banana', 'cherry']

for index, fruit in enumerate(fruits):
    print(index, fruit)
```

```
0 apple
1 banana
2 cherry
```

- 이 예시에서 'enumerate(fruits)'는 각 요소와 해당 인덱스를 반환하는 'enumerate' 객체를 생성합니다. 'for'문은 이 객체를 순회하면서 각 요소와 해당 인덱스를 튜플 형태로 'index'와 'fruit' 변수에 할당하고 출력합니다.

6-2. 예시 2

In [ ]:
```
fruits = ['apple', 'banana', 'cherry']

for index, fruit in enumerate(fruits, start=1):
    print(index, fruit)
```

```
1 apple
2 banana
3 cherry
```

- 'enumerate()' 함수는 기본적으로 인덱스를 0부터 시작합니다. 그러나 선택적으로 'start' 매개변수를 사용하여 시작 인덱스를 변경할 수 있습니다.

7. zip(), map(), filter()

7-1. zip() 함수

'zip()' 함수는 여러개의 반복 가능한 객체를 받아서 각 객체의 같은 인덱스 요소들을 묶어서 튜플 형태로 반환합니다. 반환되는 튜플은 각 객체의 해당 인덱스 요소들로 구성됩니다.

만약 입력된 객체의 길이가 다르다면 가장 짧은 길이에 맞춰서 처리됩니다.

'zip()' 함수의 일반적인 구문구조는 다음과 같습니다.

```
zip(iterable1, iterable2, ...)
```

In [ ]:
```
fruits = ['apple', 'banana', 'cherry']
prices = [1.5, 2.0, 0.75]

for fruit, price in zip(fruits, prices):
    print(fruit, price)
```

```
apple 1.5
banana 2.0
cherry 0.75
```

7-2. map() 함수

'map()' 함수는 지정된 함수를 반복 가능한(iterable) 객체의 모든 요소에 적용하여 새로운 반복 가능한 객체를 반환합니다. 반환되는 객체는 입력된 반복 가능한 객체의 요소들을 해당 한수에 적용한 결과입니다.

'map()' 함수의 일반적인 구문구조는 다음과 같습니다.

```
map(function, iterable)
```

In [ ]:
```
numbers = [1, 2, 3, 4, 5]

squared = map(lambda x: x**2, numbers)
print(list(squared))
```

```
[1, 4, 9, 16, 25]
```

7-3. filter() 함수

'filter()' 함수는 주어진 함수로부터 반환값이 참(True)인 요소들로 이루어진 새로운 반복 가능한 (iterable) 객체를 생성합니다. 주로 반복 가능한 객체(리스트, 튜플 등)의 요소를 필터링하는 데 사용됩니다.

'filter()' 함수의 일반적인 구문구조는 다음과 같습니다.

```
filter(function, iterable)
```

- function : 각 요소를 평가하는 함수입니다. 이 함수는 'True' 또는 'False'를 반환해야 합니다. function은 필수적으로 하나의 인자를 받아야 합니다.
- iterable : 요소를 필터링할 반복 가능한 객체입니다.

In [ ]:
```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)
```

```
[2, 4, 6, 8, 10]
```

In [ ]:
```
words = ["apple", "banana", "cat", "dog", "elephant"]

long_words = list(filter(lambda word: len(word) >= 4, words))
print(long_words)
```

```
['apple', 'banana', 'elephant']
```

In [ ]:
```

```