Python (6)

파이썬 제어문 - if, while, for

03-1 if 문

○ if 문은 언제 사용할까?

다음과 같은 상황을 가정해 보자

```
‘돈이 있으면 택시를 타고 가고, 돈이 없으면 걸어간다.’
```

파이썬에서는 위 상황을 다음과 같이 표현할 수 있다.

```
money = True
if money:
    print("택시를 타고 간다")
else:
    print("걸어 간다")
--------------------
택시를 타고 간다
```

○ if 문의 기본 구조

if 문의 가장 기본적인 구조와 문법은 다음과 같다.

```
if 조건문:
    실행할 명령문
    실행할 명령문
    ...
```

여기서, 조건문 결과가 True이면 그 아래의 실행할 if 블록의 명령문들을 실행하고, False이면 아래의 if 블록의 명령문들을 실행하지 않는다.

다음의 예를 보자.

```
age = 18
if age >= 18:
    print("투표가 가능합니다")
```

위 예는 age가 18 이기 때문에 if 문의 조건문인 age >= 18 이 True가 된다. 따라서 if 문의 실행문인 print()가 실행된다.
만약 age가 18보다 작은 값이 되면, print()문은 실행되지 않을 것이다.

○ if...else 문의 구조

다음은 **if**와 **else**를 사용한 조건문의 기본 구조이다.

```
if 조건문:
    실행할_문장1
    실행할_문장2
    ...
else:
    실행할_문장A
    실행할_문장B
    ...
```

조건문을 테스트해서 참이면 **if** 문 바로 다음 문장(**if** 블록)들을 수행하고 조건문이 거짓이면 **else** 문 다음 문장(**else** 블록)들을 수행하게 된다.
여기서, **else** 문은 **if** 문 없이 독립적으로 사용할 수 없다.

◆ 들여쓰기 방법 알아보기

if 문을 만들 때는 **if** 조건문: 바로 다음 문장부터 **if** 문에 속하는 모든 문장에 들여쓰기(**indentation**)를 해야 한다.

다음 예를 보면 조건문이 참일 경우 ‘실행할_문장1’을 들여쓰기했고 ‘실행할_문장2’와 ‘실행할_문장3’도 들여쓰기했다.

```
if 조건문:
    실행할_문장1
    실행할_문장2
    실행할_문장3
```

다음처럼 작성하면 오류가 발생한다. ‘수행할_문장2’를 들여쓰기하지 않았기 때문이다.

```
if 조건문:
    실행할_문장1
실행할_문장2
    실행할_문장3
```

다음과 같은 경우에도 오류가 발생한다. ‘수행할_문장3’을 들여쓰기했지만, ‘수행할_문장1’이나 ‘수행할_문장2’와 들여쓰기의 깊이가 다르다. 즉, 동일한 코드 블록에서는 언제나 같은 깊이로 들여쓰기를 해야 한다.

```
if 조건문:
     실행할_문장1
     실행할_문장2
```

```
실행할_문장3
```

그렇다면 들여쓰기는 공백 문자([Spacebar])로 하는 것이 좋을까, 탭 문자([Tab])로 하는 것이 좋을까? 이에 대한 논란은 파이썬을 사용하는 사람들 사이에서 아직도 계속되고 있다.
공백 문자로 하자는 쪽과 탭 문자로 하자는 쪽 모두가 동의하는 내용은 2가지를 혼용해서 쓰지 말자는 것이다.
공백 문자로 할 것이라면 공백으로 통일하고, 탭 문자로 할 것이라면 탭으로 통일하자는 말이다.
탭이나 공백은 프로그램 소스에서 눈으로 보이는 것이 아니기 때문에 혼용해서 쓰면 오류의 원인이 되므로 주의하자.

요즘 파이썬 커뮤니티에서는 들여쓰기를 할 때 공백 문자 4개를 사용하는 것을 권장한다.
또한 파이썬 에디터는 대부분 탭 문자로 들여쓰기를 하더라도 탭 문자를 공백 문자 4개로 자동 변환하는 기능을 갖추고 있다.

◆ 조건문 다음에 콜론(:)을 잊지 말자!

if 조건문 뒤에는 반드시 콜론(:)이 붙는다.
어떤 특별한 의미가 있다기보다는 파이썬의 문법 구조이다.
앞으로 배울 while이나 for, def, class도 역시 문장의 끝에 콜론(:)이 항상 들어간다.
초보자들은 이 콜론(:)을 빠뜨리는 경우가 많으므로 특히 주의하자.

파이썬이 다른 언어보다 보기 쉽고 소스 코드가 간결한 이유는 바로 콜론(:)을 사용하여 들여쓰기를 하도록 만들었기 때문이다.
하지만 이는 숙련된 프로그래머들이 파이썬을 처음 접할 때 제일 혼란스러워하는 부분이기도 하다.
다른 언어에서는 if 문에 속한 문장들을 {}로 감싸지만, 파이썬에서는 들여쓰기로 해결한다는 점을 기억하자.

○ 조건문이란 무엇인가?

if 조건문에서 '조건문'이란 참과 거짓으로 판단되는 문장을 말한다.

앞에서 살펴본 택시 예제에서 조건문은 **money**가 된다.

```
money = True
if money:
    ...
```

money는 **True**이기 때문에 조건이 참이 되어 **if** 문 다음 문장을 실행한다.

◆ 비교 연산자

이번에는 조건문에 비교 연산자(<, >, ==, !=, >=, <=)를 쓰는 방법에 대해 알아보자.

다음 표는 비교 연산자를 잘 설명해 준다.

비교연산자	설명
x < y	x가 y보다 작다.
x > y	x가 y보다 크다.
x == y	x와 y가 같다.
x != y	x와 y가 같지 않다.
x >= y	x가 y보다 크거나 같다.
x <= y	x가 y보다 작거나 같다.

이러한 비교 연산자는 두 값을 비교하고 True 또는 False 부울 값을 반환한다.
이제 이 연산자를 어떻게 사용하는지 알아보자.

```
x = 3
y = 2
print(x > y)
-------------------
True
```

x에 3, y에 2를 대입한 후 x > y라는 조건문을 수행하면 True를 리턴한다. x > y 조건문이 참이기 때문이다.

```
print(x < y)
```

```
--------------------
False
```

위 조건문은 거짓이기 때문에 **False**를 리턴한다.

```
print(x == y)
--------------------
False
```

x와 **y**는 같지 않다. 따라서 위 조건문은 거짓이므로 **False**를 리턴한다.

```
print(x != y)
--------------------
True
```

x와 **y**는 같지 않다. 따라서 위 조건문은 참이므로 **True**를 리턴한다.

사용법을 알아봤으므로 이제 응용해 보자. 앞에서 살펴본 택시 예제를 다음처럼 바꾸려면 어떻게 해야 할까?

```
만약 5000원 이상의 돈을 가지고 있으면 택시를 타고 가고, 그렇지 않으면 걸어가라.
```

이 상황은 다음처럼 프로그래밍할 수 있다.

```
money = 2000
if money >= 5000:
    print("택시를 타고 가라")
else:
    print("걸어가라")
-------------------
걸어가라
```

money >= 5000 조건문이 거짓이 되기 때문에 else 문 다음 문장을 수행하게 된다.

◆ 논리 연산자 and, or, not

조건을 판단하기 위해 사용하는 다른 연산자로는 and, or, not이 있다. 각각의 연산자는 다음처럼 동작한다.

연산자	설명
x and y	x와 y 모두 True이어야 True이다. 그렇지 않으면 False이다
x or y	x와 y 둘 중 하나만 True이어도 True이다. 그렇지 않으면 False이다
not x	x가 True이면 False이고, x가 False이면 True이다

1) and 연산자

and 연산자는 두 조건이 동시에 **True**인지 확인한다.

```
a and b
```

두 조건이 모두 **True**이면 **True**를 반환한다. 그리고 조건 **a** 또는 **b** 중 하나가 **False**이면 **False**를 반환한다.

다음 예에서는 **and** 연산자를 사용하여 **price**를 숫자와 비교하는 두 조건을 판단하는 예이다.

```
price = 9.99
print(price > 9 and price < 10)
```

price가 **9**보다 크고 **10**보다 작기 때문에 결과는 **True**가 된다.

다음 예에서는 **price**가 **10**보다 크지 않기 때문에 **False**를 반환한다.

```
print(price > 10 and price < 20)
```

이 예에서 조건 **price > 10**은 **False**를 반환하고 두 번째 조건 **price < 20**은 **True**를 반환한다.

다음 표에서는 두 조건을 결합할 때 **and** 연산자의 결과를 보여줍니다.

a	b	a and b
True	True	True
True	False	False
False	False	False
False	True	False

표에서 볼 수 있듯이 and 연산자는 모든 조건이 모두 True로 평가되는 경우에만 True를 반환한다.

2) or 연산자

and 연산자와 마찬가지로 **or** 연산자도 여러 조건을 확인한다.
그러나 개별 조건 중 하나 이상이 **True**인 경우 **True**를 반환한다.

```
a or b
```

다음 표에서는 두 조건을 결합할 때 or 연산자의 결과를 보여준다.

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

or 연산자는 모든 조건들이 **False**인 경우에만 **False**를 반환합니다. 그 외에는 **True**를 반환한다.

다음 예에서는 **or** 연산자를 사용하는 방법을 보여준다.

```
price = 9.99
print(price > 10 or price < 20)
```

이 예에서 **price < 20**은 **True**를 반환하므로 전체 표현식이 **True**를 반환한다.

다음 예에서는 두 조건 모두 **False**로 평가되므로 **False**를 반환합니다.

```
price = 9.99
print(price > 10 or price < 5)
```

3) not 연산자

not 연산자는 하나의 조건에 적용된다.
그리고 해당 조건의 결과를 반전시켜 **True**가 **False**가 되고 **False**가 **True**가 된다.

```
not a
```

not 연산자는 조건이 **True**이면 **False**를 반환하고, 조건이 **False**이면 **not** 연산자는 **True**를 반환한다.

다음 표에서는 **not** 연산자의 결과를 보여준다.

a	not a
True	false
False	True

다음 예에서는 **not** 연산자를 사용한다. **price > 10**은 **False**를 반환하므로 **price > 10**이 아닌 경우 **True**를 반환합니다

```
price = 9.99
```

```
print(not price > 10)
```

다음은 not 및 and 연산자를 결합한 또 다른 예이다.

```
print(not (price > 5 and price < 10))
```

이 예에서 Python은 다음 순서에 따라 조건을 평가한다.

먼저 (price > 5 and price < 10)은 True로 평가된다.
둘째, Not True는 False로 평가된다.

이는 논리 연산자의 우선 순위라는 중요한 개념에 의해 이루어진다.

4) 논리 연산자의 우선순위

표현식에 논리 연산자를 혼합하면 Python은 '연산자 우선 순위'라고 하는 순서대로 이를 평가한다.

다음은 not, and, or 연산자의 우선순위를 보여준다.

Operator	우선순위
not	높음
and	중간
or	낮음

이러한 우선 순위에 따라 Python은 우선 순위가 가장 높은 연산자의 피연산자를 먼저 그룹화한 다음 우선 순위가 낮은 연산자의 피연산자를 그룹화한다.

표현식에 동일한 우선순위를 갖는 여러 논리 연산자가 있는 경우 Python은 이를 왼쪽에서 오른쪽으로 평가한다.

a or b and c	=>	a or (b and c)
a and b or c and d	=>	(a and b) or (c and d)
a and b and c or d	=>	((a and b) and c) or d
not a and b or c	=>	((not a) and b) or c

다음 예를 통해 or 연산자를 적용하는 예를 알아보자.

```
돈이 5000원 이상 있거나 카드가 있다면 택시를 타고 가고, 그렇지 않으면 걸어가라.
```

```
money = 2000
card = True
if money >= 5000 or card:
    print("택시를 타고 가라")
else:
    print("걸어가라")
--------------------
택시를 타고 가라
```

money는 2000이지만, card가 True이기 때문에 money >= 5000 or card 조건문이 참이 된다.
따라서 if 문에 속한 '택시를 타고 가라' 문장이 출력된다.

◆ in, not in

파이썬은 다른 프로그래밍 언어에서 쉽게 볼 수 없는 재미있는 조건문도 제공한다. 바로 다음과 같은 것들이다.

in	not in
x in 리스트	x not in 리스트
x in 튜플	x not in 튜플
x in 문자열	x not in 문자열

영어 단어 in의 뜻이 '~안에'라는 것을 생각해 보면 다음 예가 쉽게 이해될 것이다.

```
print(1 in [1, 2, 3])
--------------------
True
```

```
print(1 not in [1, 2, 3])
--------------------
False
```

앞에서 첫 번째 예는 '[1, 2, 3]이라는 리스트 안에 1이 있는가?'라는 조건문이다.
1은 [1, 2, 3] 안에 있으므로 참이 되어 True를 리턴한다.
두 번째 예는 '[1, 2, 3] 리스트 안에 1이 없는가?'라는 조건문이다.
1은 [1, 2, 3] 안에 있으므로 거짓이 되어 False를 리턴한다.

다음은 튜플과 문자열에 **in**과 **not in**을 적용한 예이다.
각각의 결과가 나온 이유는 쉽게 유추할 수 있다.

```
print('a' in ('a', 'b', 'c'))
--------------------
True
```

```
print('j' not in 'python')
--------------------
True
```

이번에는 우리가 계속 사용해 온 택시 예제에 **in**을 적용해 보자.

```
만약 주머니에 돈이 있으면 택시를 타고 가고, 없으면 걸어가라.
```

```
pocket = ['paper', 'cellphone', 'money']
if 'money' in pocket:
    print("택시를 타고 가라")
```

```
else:
    print("걸어가라")
--------------------
택시를 타고 가라
```

['paper', 'cellphone', 'money'] 리스트 안에 'money'가 있으므로 'money' in pocket은 참이 된다.
따라서 if 문에 속한 문장이 수행된다.

◆ 조건문에서 아무 일도 하지 않게 설정하고 싶다면?

가끔 조건문의 참, 거짓에 따라 실행할 행동을 정의할 때나 아무런 일도 하지 않도록 설정하고 싶을 때가 있다.
다음 예를 살펴 보자.

```
주머니에 돈이 있으면 가만히 있고, 주머니에 돈이 없으면 카드를 꺼내라.
```

이럴 때 사용하는 것이 바로 pass이다. 위 예를 pass를 적용해서 구현해 보자.

```
pocket = ['paper', 'money', 'cellphone']
if 'money' in pocket:
    pass
```

```
else:
    print("카드를 꺼내라")
--------------------
```

pocket 리스트 안에 **money** 문자열이 있기 때문에 **if** 문 다음 문장인 **pass**가 수행되고 아무런 결괏값도 보여 주지 않는다.

◆ 다양한 조건을 판단하는 if...elif...else

if와 **else**만으로는 다양한 조건을 판단하기 어렵다.
다음 예를 보더라도 **if**와 **else**만으로는 조건을 판단하는 데 어려움을 겪게 된다는 것을 알 수 있다.

```
주머니에 돈이 있으면 택시를 타고 가고, 주머니에 돈은 없지만 카드가 있으면 택시를 타고 가고, 돈도 없고 카드도 없으면 걸어 가라.
```

위 문장을 보면 조건을 판단하는 부분이 두 군데 있다.
먼저 주머니에 돈이 있는지를 판단해야 하고 주머니에 돈이 없으면 다시 카드가 있는지 판단해야 한다.

if와 **else**만으로 위 문장을 표현하려면 다음과 같이 할 수 있다.

```
pocket = ['paper', 'cellphone']
card = True
if 'money' in pocket:
    print("택시를 타고가라")
else:
    if card:
        print("택시를 타고가라")
    else:
        print("걸어가라")
--------------------
택시를 타고가라
```

언뜻 보기에도 이해하기 어렵고 산만한 느낌이 든다.
이런 복잡함을 해결하기 위해 파이썬에서는 다중 조건 판단을 가능하게 하는 **elif**를 사용한다.

위 예에 **elif**를 사용하면 다음과 같이 바꿀 수 있다.

```
pocket = ['paper', 'cellphone']
card = True
if 'money' in pocket:
    print("택시를 타고가라")
elif card:
```

```
    print("택시를 타고가라")
else:
    print("걸어가라")
--------------------
택시를 타고가라
```

즉, elif는 이전 조건문이 거짓일 때 수행된다. if, elif, else를 모두 사용할 때 기본 구조는 다음과 같다.

```
if 조건문:
    수행할_문장
    ...
elif 조건문:
    수행할_문장
    ...
elif 조건문:
    수행할_문장
    ...
...
else:
   수행할_문장
   ...
```

위에서 볼 수 있듯이 **elif**는 개수에 제한 없이 사용할 수 있다.

◆ if 문을 한 줄로 작성하기

앞의 **pass**를 사용한 예를 보면 **if** 문 다음에 수행할 문장이 한 줄이고 **else** 문 다음에 수행할 문장도 한 줄밖에 되지 않는다.

```
if 'money' in pocket:
    pass
else:
    print("카드를 꺼내라")
```

이렇게 수행할 문장이 한 줄일 때 좀 더 간략하게 코드를 작성하는 방법이 있다.

```
if 'money' in pocket:  pass
else: print("카드를 꺼내라")
```

if 문 다음에 수행할 문장을 콜론(:) 뒤에 바로 적었다. **else** 문 역시 마찬가지이다.

○ 조건부 표현식

다음 코드를 살펴보자. score가 60 이상일 경우 message에 문자열 "success", 아닐 경우에는 문자열 "failure"를 대입하는 코드이다.

```
if score >= 60:
    message = "success"
else:
    message = "failure"
```

파이썬의 조건부 표현식(conditional expression)을 사용하면 위 코드를 다음과 같이 간단히 표현할 수 있다.

```
message = "success" if score >= 60 else "failure"
```

조건부 표현식은 다음과 같이 정의한다.

```
변수 = 조건문이_참인_경우의_값 if 조건문 else 조건문이_거짓인_경우의_값
```

조건부 표현식은 가독성에 유리하고 한 줄로 작성할 수 있어 활용성이 좋다.

03-2 while 문

문장을 여러번 반복해서 수행해야 할 경우 while 문을 사용한다. 그래서 while 문을 '반복문'이라고도 부른다.

○ while 문의 기본 구조

다음은 while 문의 기본 구조이다.

```
while 조건문:
    실행할_문장
    실행할_문장
    실행할_문장
    ...
```

while 문은 조건문이 참인 동안 while 문에 속한 문장들이 반복해서 수행된다.

'열 번 찍어 안 넘어가는 나무 없다'라는 속담을 파이썬 프로그램으로 만들면 다음과 같다.

```
treeHit = 0
while treeHit < 10:
     treeHit = treeHit +1
     print("나무를 %d번 찍었습니다." % treeHit)
     if treeHit == 10:
           print("나무 넘어갑니다.")
--------------------
나무를 1번 찍었습니다.
나무를 2번 찍었습니다.
나무를 3번 찍었습니다.
나무를 4번 찍었습니다.
나무를 5번 찍었습니다.
나무를 6번 찍었습니다.
나무를 7번 찍었습니다.
나무를 8번 찍었습니다.
나무를 9번 찍었습니다.
나무를 10번 찍었습니다.
나무 넘어갑니다.
```

위 예에서 **while** 문의 조건문은 **treeHit < 10**이다.
즉, **treeHit**가 **10**보다 작은 동안 **while** 문에 포함된 문장들을 계속 수행한다.
whlie문 안의 문장을 보면 가장 먼저 **treeHit = treeHit + 1**로 **treeHit** 값이 계속 **1**씩 증가한다는 것을 알 수 있다.

그리고 나무를 treeHit번만큼 찍었다는 것을 알리는 문장을 출력하고 treeHit가 10이 되면 "나무 넘어갑니다."라는 문장을 출력한다.
그러고 나면 treeHit < 10 조건문이 거짓이 되므로 while 문을 빠져나가게 된다.

treeHit = treeHit + 1은 프로그래밍을 할 때 매우 자주 사용하는 기법이다.
treeHit 값을 1만큼씩 증가시킬 목적으로 사용하며 treeHit += 1처럼 작성해도 된다.

다음은 while 문이 반복되는 과정을 순서대로 정리한 것입니다.

treeHit	조건문 조건	판단	수행하는 문장	while문
0	0 < 10	참	나무를 1번 찍었습니다.	반복
1	1 < 10	참	나무를 2번 찍었습니다.	반복
2	2 < 10	참	나무를 3번 찍었습니다.	반복
3	3 < 10	참	나무를 4번 찍었습니다.	반복
4	4 < 10	참	나무를 5번 찍었습니다.	반복
5	5 < 10	참	나무를 6번 찍었습니다.	반복
6	6 < 10	참	나무를 7번 찍었습니다.	반복
7	7 < 10	참	나무를 8번 찍었습니다.	반복
8	8 < 10	참	나무를 9번 찍었습니다.	반복
9	9 < 10	참	나무를 10번 찍었습니다. 나무 넘어갑니다.	반복
10	10 < 10	거짓		종료

❍ while 문 만들기

이번에는 여러 가지 선택지 중 하나를 선택해서 입력받는 예제를 만들어 보자.
먼저 다음과 같이 여러 줄짜리 문자열을 만들자.

```
prompt = """
1. Add
2. Del
3. List
4. Quit

... Enter number: """
```

이어서 number 변수에 0을 먼저 대입한다.
이렇게 변수를 먼저 설정해 놓지 않으면 다음에 나올 while 문의 조건문인 number != 4에서 변수가 존재하지 않는다는 오류가 발생한다.

```
number = 0
while number != 4:
    print(prompt)
```

```
    number = int(input())
```

while 문을 보면 number가 4가 아닌 동안 prompt를 출력하고 사용자로부터 번호를 입력받는다. 다음 결과 화면처럼 사용자가 값 4를 입력하지 않으면 계속해서 prompt를 출력한다.

여기에서 number = int(input())는 사용자의 숫자 입력을 받아들이는 것이라고만 알아 두자.
int나 input 함수에 대한 내용은 뒤에 나오는 내장 함수 부분에서 자세하게 다룬다.

```
1. Add
2. Del
3. List
4. Quit

... Enter number:
```

4를 입력하면 조건문이 거짓이 되어 while 문을 빠져나가게 된다.

```
... Enter number:
4
```

❍ while 문 강제로 빠져나가기

while 문은 조건문이 참인 동안 계속 **while** 문 안의 내용을 반복적으로 수행한다.
하지만 강제로 **while** 문을 빠져나가고 싶을 때가 있다.

예를 들어 커피 자판기를 생각해 보자. 자판기 안에 커피가 충분히 있을 때 동전을 넣으면 커피가 나온다.
그런데 자판기가 제대로 작동하려면 커피가 얼마나 남았는지 항상 검사해야 한다.
만약 커피가 떨어졌다면 판매를 중단하고 '판매 중지' 문구를 사용자에게 보여 주어야 한다.
이렇게 판매를 강제로 멈추게 하는 것이 바로 **break** 문이다.

다음 예는 커피 자판기 이야기를 파이썬 프로그램으로 표현해 본 것이다.

```
coffee = 10
money = 300
while money:
    print("돈을 받았으니 커피를 줍니다.")
    coffee = coffee -1
    print("남은 커피의 양은 %d개입니다." % coffee)
    if coffee == 0:
        print("커피가 다 떨어졌습니다. 판매를 중지합니다.")
        break
```

money가 300으로 고정되어 있고 while money:에서 조건문인 money는 0이 아니기 때문에 항상 참이다.
따라서 무한히 반복되는 무한 루프를 돌게 된다.
그리고 while 문의 내용을 한 번 수행할 때마다 coffee = coffee - 1에 의해 coffee의 개수가 1개씩 줄어든다.
만약 coffee가 0이 되면 if coffee == 0: 문장에서 coffee == 0이 참이 되므로 if 문 다음 문장 "커피가 다 떨어졌습니다. 판매를 중지합니다."가 출력되고 break 문이 호출되어 while 문을 빠져나가게 된다.

하지만 실제 자판기는 위 예처럼 작동하지는 않을 것이다.
다음은 자판기의 실제 작동 과정과 비슷하게 만들어 본 예이다.

```
coffee = 10
while True:
    money = int(input("돈을 넣어 주세요: "))
    if money == 300:
        print("커피를 줍니다.")
        coffee = coffee - 1
    elif money > 300:
        print("거스름돈 %d를 주고 커피를 줍니다." % (money - 300))
        coffee = coffee - 1
    else:
        print("돈을 다시 돌려주고 커피를 주지 않습니다.")
        print("남은 커피의 양은 %d개 입니다." % coffee)

    if coffee == 0:
```

```
        print("커피가 다 떨어졌습니다. 판매를 중지 합니다.")
        break
```

위 프로그램 소스를 따로 설명하지는 않겠다. 여러분이 소스를 입력하면서 무슨 내용인지 이해할 수 있다면 지금까지 배운 **if** 문이나 **while** 문을 이해했다고 보면 된다.
만약 **money = int(input("돈을 넣어 주세요: "))** 문장이 이해되지 않는다면 이 문장은 사용자로부터 값을 입력받는 부분이고 입력받은 숫자를 **money** 변수에 대입하는 것이라고만 알아 두자.

다음과 같은 입력란이 나타난다.

```
돈을 넣어 주세요:
```

입력란에 여러 숫자를 입력해 보면서 결과를 확인하자.

```
돈을 넣어 주세요: 500
거스름돈 200를 주고 커피를 줍니다.
돈을 넣어 주세요: 300
커피를 줍니다.
돈을 넣어 주세요: 100
돈을 다시 돌려주고 커피를 주지 않습니다.
남은 커피의 양은 8개입니다.
돈을 넣어 주세요:
```

❍ while 문의 조건문으로 돌아가기

while 문 안의 문장을 수행할 때 입력 조건을 검사해서 조건에 맞지 않으면 while 문을 종료하게 된다.
그런데 프로그래밍을 하다 보면 while 문을 종료하지 않고 while 문의 조건문으로 다시 돌아가게 만들고 싶은 경우가 생기게 된다.
이때 사용하는 것이 바로 continue 문이다.

1부터 10까지의 숫자 중에서 홀수만 출력하는 것을 while 문을 사용해서 작성한다고 생각해 보자. 어떤 방법이 좋을까?

```
a = 0
while a < 10:
    a = a + 1
    if a % 2 == 0:
        continue
    print(a)
--------------------
1
3
5
7
9
```

위는 1부터 10까지의 숫자 중 홀수만 출력하는 예이다.
a가 10보다 작은 동안 a는 1만큼씩 계속 증가한다.
a % 2 == 0(a를 2로 나누었을 때 나머지가 0인 경우)이 참이 되는 경우는 a가 짝수일 때이다.
즉, a가 짝수이면 continue 문을 수행한다.
이 continue 문은 while 문의 맨 처음인 조건문(a < 10)으로 돌아가게 하는 명령어이다.
따라서 위 예에서 a가 짝수이면 print(a) 문장은 수행되지 않을 것이다.

○ 무한 루프

이번에는 무한 루프(endless loop)에 대해 알아보자.
무한 루프란 무한히 반복한다는 의미이다.
우리가 사용하는 일반 프로그램 중에서 무한 루프 개념을 사용하지 않는 프로그램은 거의 없다. 그만큼 자주 사용한다는 뜻이다.

파이썬에서 무한 루프는 while 문으로 구현할 수 있다. 다음은 무한 루프의 기본 형태이다.

```
while True:
    수행할_문장
    수행할_문장
    ...
```

while 문의 조건문이 **True**이므로 항상 참이 된다.
따라서 **while** 문 안에 있는 문장들은 무한히 수행될 것이다.

다음은 무한 루프의 예이다.

```
while True:
    print("Ctrl+C를 눌러야 while문을 빠져나갈 수 있습니다.")
--------------------
Ctrl+C를 눌러야 while문을 빠져나갈 수 있습니다.
Ctrl+C를 눌러야 while문을 빠져나갈 수 있습니다.
Ctrl+C를 눌러야 while문을 빠져나갈 수 있습니다.
....
```

위 문장은 영원히 출력된다.
하지만 이 예처럼 아무 의미 없이 무한 루프를 돌리는 경우는 거의 없을 것이다. **[Ctrl+C]**를 눌러 빠져나가자.

03-3 for 문

파이썬의 직관적인 특징을 가장 잘 보여 주는 것이 바로 이 for 문이다.
while 문과 비슷한 반복문인 for 문은 문장 구조가 한눈에 들어온다는 장점이 있다.

○ for 문의 기본 구조

for 문의 기본 구조는 다음과 같다.

```
for 변수 in 리스트(또는 튜플, 문자열):
    수행할_문장
    수행할_문장
    ...
```

리스트나 튜플, 문자열의 첫 번째 요소부터 마지막 요소까지 차례로 변수에 대입되어 '수행할_문장' 등이 수행된다.

❍ 예제를 통해 for 문 이해하기

for 문은 예제를 통해서 살펴보는 것이 가장 알기 쉽다. 다음 예제를 직접 입력해 보자.

1. 전형적인 for 문

```
test_list = ['one', 'two', 'three']
for i in test_list:
    print(i)
--------------------
one
two
three
```

['one', 'two', 'three'] 리스트의 첫 번째 요소인 'one'이 먼저 i 변수에 대입된 후 print(i) 문장을 수행한다.
다음에 두 번째 요소 'two'가 i 변수에 대입된 후 print(i) 문장을 수행하고 리스트의 마지막 요소까지 이것을 반복한다.

2. 다양한 for 문의 사용

```
a = [(1,2), (3,4), (5,6)]
```

first, last = (1,2) 구조분해

```
for (first, last) in a:
    print(first + last)
--------------------
3
7
11
```

위 예는 a 리스트의 요솟값이 튜플이기 때문에 각각의 요소가 자동으로 (first, last) 변수에 대입된다.

3. for 문의 응용

for 문의 쓰임새를 알기 위해 다음과 같은 문제를 생각해 보자.

```
총 5명의 학생이 시험을 보았는데 시험 점수가 60점 이상이면 합격이고 그렇지 않으면 불합격이다. 합격인지, 불합격인지 결과를 보여 주시오.
```

먼저 학생 5명의 시험 점수를 리스트로 표현해 보자.

```
marks = [90, 25, 67, 45, 80]
```

1번 학생은 90점이고 5번 학생은 80점이다.

이런 점수를 차례로 검사해서 합격했는지, 불합격했는지 통보해 주는 프로그램을 만들어 보자.

```
marks = [90, 25, 67, 45, 80]   # 학생들의 시험 점수 리스트

number = 0   # 학생에게 붙여 줄 번호
for mark in marks:   # 90, 25, 67, 45, 80을 순서대로 mark에 대입
    number = number +1
    if mark >= 60:
        print("%d번 학생은 합격입니다." % number)
    else:
        print("%d번 학생은 불합격입니다." % number)
```

각각의 학생에게 번호를 붙여 주기 위해 number 변수를 사용하였다.
점수 리스트 marks에서 차례로 점수를 꺼내어 mark라는 변수에 대입하고 for 문 안의 문장들을 수행한다.
먼저 for 문이 한 번씩 수행될 때마다 number는 1씩 증가한다.

이 프로그램을 실행하면 mark가 60 이상일 때 합격 메시지를 출력하고 60을 넘지 않을 때 불합격 메시지를 출력한다.

○ for 문과 continue 문

while 문에서 살펴본 continue 문을 for 문에서도 사용할 수 있다. 즉, for 문 안의 문장을 수행하는 도중 continue 문을 만나면 for 문의 다음 조건문으로 돌아가게 된다.

앞에서 for 문 응용 예제를 그대로 사용해서 60점 이상인 사람에게는 축하 메시지를 보내고 나머지 사람에게는 아무런 메시지도 전하지 않는 프로그램을 작성해 보자.

```
marks = [90, 25, 67, 45, 80]

number = 0
for mark in marks:
    number = number +1
    if mark < 60:
        continue
    print("%d번 학생 축하합니다. 합격입니다. " % number)
--------------------
1번 학생 축하합니다. 합격입니다.
3번 학생 축하합니다. 합격입니다.
5번 학생 축하합니다. 합격입니다.
```

점수가 60점 이하인 학생인 경우에는 mark < 60이 참이 되어 continue 문이 수행된다.
따라서 축하 메시지를 출력하는 부분인 print 문을 수행하지 않고 for 문의 처음으로 돌아가게 된다.

○ for 문과 함께 자주 사용하는 range 함수

for 문은 숫자 리스트를 자동으로 만들어 주는 range 함수와 함께 사용하는 경우가 많다.
다음은 range 함수의 간단한 사용법이다.

```
a = range(10)
print(a)
print(list(a))
--------------------
range(0, 10)    --> range object
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

range(10)은 0부터 10 미만의 숫자를 포함하는 range 객체를 만들어 준다.
list() 함수를 사용하면 range 객체를 list로 변환한다.

시작 숫자와 끝 숫자를 지정하려면 range(시작_숫자, 끝_숫자) 형태를 사용하는데, 이때 끝 숫자는 포함되지 않는다.
즉, 끝_숫자 - 1 까지의 정수 범위를 반환한다.

```
a = range(1, 11)
print(a)
```

```
print(list(a))
--------------------
range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

◆ range() 함수의 세 번째 매개변수 – step

아래의 예는 1부터 6까지, 2 간격의 값을 출력한다.

```
a = range(1, 7, 2)
print(list(a))
--------------------
[1, 3, 5]
```

◆ range() 함수의 세 번째 매개변수가 음수인 경우 – step

아래의 예는 10부터 1까지, 내림차순의 list를 생성한다.

```
a = range(10, 0, -1)
```

```
print(list(a))
--------------------
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

세 번째 매개변수인 step 값이 음수이면, 당연히 시작_숫자가 끝_숫자 보다 값이 커야 한다.
즉, 1씩 감소하는 리스트가 생성된다.

◆ for문에서의 range 함수 활용 예시 살펴보기

for와 range 함수를 사용하면 1부터 10까지 더하는 것을 다음과 같이 쉽게 구현할 수 있다.

```
add = 0

for i in range(1, 11):
    add = add + i

print(add)
--------------------
55
```

range(1, 11)은 숫자 1부터 10까지(1 이상 11 미만)의 숫자를 데이터로 가지는 객체이다.

따라서 위 예에서 i 변수에 숫자가 1부터 10까지 하나씩 차례로 대입되면서 add = add + i 문장을 반복적으로 수행하고 add는 최종적으로 55가 된다.

또한 우리가 앞에서 살펴본 합격 축하 문장을 출력하는 예제도 range 함수를 사용해서 다음과 같이 바꿀 수 있다.

```
marks = [90, 25, 67, 45, 80]
for number in range(len(marks)):
    if marks[number] < 60:
        continue
    print("%d번 학생 축하합니다. 합격입니다." % (number+1))
```

len는 리스트 안의 요소 개수를 리턴하는 함수이다.
따라서 len(marks)는 5, range(len(marks))는 range(5)가 될 것이다.
number 변수에는 차례로 0부터 4까지의 숫자가 대입되고 marks[number]는 차례대로 90, 25, 67, 45, 80 값을 가지게 된다.

◆ for와 range를 이용한 구구단

for와 range 함수를 사용하면 소스 코드 단 4줄만으로 구구단을 출력할 수 있다. 들여쓰기에 주의하면서 입력해 보자.

```
for i in range(2,10):               # 1번 for문
    for j in range(1, 10):          # 2번 for문
```

```
            print(i*j, end=" ")
        print('')
--------------------
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

위 예를 보면 for 문을 두 번 사용했다.
1번 for 문에서 2부터 9까지의 숫자(range(2, 10))가 차례대로 i에 대입된다.
i가 처음 2일 때 2번 for 문을 만나게 된다.
2번 for 문에서 1부터 9까지의 숫자(range(1, 10))가 j에 대입되고 그다음 문장인 print(i*j, end=" ")를 수행한다.
따라서 i가 2일 때 2 * 1, 2 * 2, 2 * 3, … 2 * 9까지 차례대로 수행되며 그 값을 출력하게 된다.
그다음으로 i가 3일 때 역시 2일 때와 마찬가지로 수행될 것이고 i가 9일 때까지 계속 반복된다.

print(i*j, end=" ")와 같이 print 함수에 end 파라미터를 설정한 이유는, 해당 결괏값을 출력할 때 다음 줄로 넘기지 않고 그 줄에 계속 출력하기 위해서이다.
그다음에 이어지는 print('')는 2단, 3단 등을 구분하기 위해 사용했다. 두 번째 for 문이 끝나면 결괏값을 다음 줄부터 출력하게 하는 역할을 한다.

print 문의 end 매개변수에는 줄바꿈 문자(\n)가 기본값으로 설정되어 있다.
print 문은 다음에 보다 자세히 다룬다.

○ 리스트 컴프리헨션 사용하기

리스트 안에 for 문을 포함하는 리스트 컴프리헨션(list comprehension)을 사용하면 좀 더 편리하고 직관적인 프로그램을 만들 수 있다.
다음 예제를 살펴보자.

```
a = [1,2,3,4]
result = []
for num in a:
    result.append(num*3)

print(result)
--------------------
[3, 6, 9, 12]
```

위 예제에서는 a 리스트의 각 항목에 3을 곱한 결과를 result 리스트에 담았다.

리스트 컴프리헨션을 사용하면 다음과 같이 좀 더 간단하게 작성할 수 있다.

```
a = [1,2,3,4]
result = [num * 3 for num in a]
print(result)
--------------------
[3, 6, 9, 12]
```

만약 [1, 2, 3, 4] 중에서 짝수에만 3을 곱하여 담고 싶다면 리스트 컴프리헨션 안에 'if 조건문'을 사용하면 된다.

```
a = [1,2,3,4]
result = [num * 3 for num in a if num % 2 == 0]
print(result)
--------------------
[6, 12]
```

1. range() 함수
2. 리스트 컴프리핸션

리스트 컴프리헨션의 문법은 다음과 같다. 'if 조건문' 부분은 앞의 예제에서 볼 수 있듯이 생략할 수 있다.

```
[표현식 for 항목 in 반복_가능_객체 if 조건문]
```

조금 복잡하지만, for 문을 2개 이상 사용하는 것도 가능하다.
for 문을 여러 개 사용할 때의 문법은 다음과 같다.

```
[표현식 for 항목1 in 반복_가능_객체1 if 조건문1
      for 항목2 in 반복_가능_객체2 if 조건문2
      ...
      for 항목n in 반복_가능_객체n if 조건문n]
```

만약 구구단의 모든 결과를 리스트에 담고 싶다면 리스트 컴프리헨션을 사용하여 다음과 같이 간단하게 구현할 수도 있다.

```
result = [x*y for x in range(2,10)
     for y in range(1,10)]
print(result)
--------------------
[2, 4, 6, 8, 10, 12, 14, 16, 18, 3, 6, 9, 12, 15, 18, 21, 24, 27, 4, 8, 12, 16,
20, 24, 28, 32, 36, 5, 10, 15, 20, 25, 30, 35, 40, 45, 6, 12, 18, 24, 30, 36, 42
, 48, 54, 7, 14, 21, 28, 35, 42, 49, 56, 63, 8, 16, 24, 32, 40, 48, 56, 64, 72,
9, 18, 27, 36, 45, 54, 63, 72, 81]
```

지금까지 프로그램의 흐름을 제어하는 **if** 문, **while** 문, **for** 문에 대해 학습하였다.

while 문과 **for** 문은 매우 비슷하다.
실제로 **for** 문으로 작성한 코드를 **while** 문으로 바꿀 수 있는 경우도 많고, **while** 문을 **for** 문으로 바꾸어서 사용할 수 있는 경우도 많다.

Python Date Type

1. 숫자 – integer, float

2. 문자열
   1) indexing
   2) slicing (start, end, stop)
   3) formatting – %s, %d, %f, “”, format, {}

3. list[ ]
   1) indexing
   2) slicing

4. Tuple( )
   – 한 번 정의되면 바꿀 수가 없다.

5. Dictionary
   – 순서가 중요하지 않다. (순서가 없다.)
   – a = {key : value}
   – a[key] => value

6. Set { }
   – new Set([ ])
   – 중복된 값을 가질 수가 없다.