위키독스

원본 URL: https://wikidocs.net/19

03장 프로그램의 구조를 쌓는다! 제어문

이번 장에서는 if, while, for 등의 제어문에 대해서 배우고자 한다. 제어문을 배우기 전에 집을 짓는 일을 생각해 보자.

집을 지을 때 나무, 돌, 시멘트 등은 재료가 되고, 철근은 집의 뼈대가 된다. 프로그램을 만드는 것도 집 짓기와 매우 비슷한 면이 있다. 나무, 돌, 시멘트와 같은 재료는 자료형이 되고, 집의 뼈대를 이루는 철근은 이번 장에서 알아볼 제어문에 해당한다. 이번 장에서는 자료형을 바탕으로 제어문을 이용하여 프로그램의 구조를 만들어 보자.

마지막 편집일시 : 2017년 5월 1일 2:38 오후

이전글 : 02-8 자료형의 값을 저장하는 공간, 변수

다음글 : 03-1 if문

위키독스

원본 URL: https://wikidocs.net/20

03-1 if문

if문은 왜 필요할까?

다음과 같은 상상을 해보자.

> "돈이 있으면 택시를 타고, 돈이 없으면 걸어 간다."

우리 모두에게 언제든지 일어날 수 있는 상황 중 하나이다. 프로그래밍도 사람이 하는 것이므로 위의 문장처럼 주어진 조건을 판단한 후 그 상황에 맞게 처리해야 할 경우가 생긴다. 이렇듯 프로그래밍에서 조건을 판단하여 해당 조건에 맞는 상황을 수행하는 데 쓰이는 것이 바로 if 문이다.

위와 같은 상황을 파이썬에서는 다음과 같이 표현할 수 있다.

```
>>> money = True
>>> if money:
...     print("택시를 타고 가라")
... else:
...     print("걸어 가라")
...
택시를 타고 가라
```

money에 입력된 True는 참이다. 따라서 if문 다음의 문장이 수행되어 '택시를 타고 가라'가 출력된다.

if문의 기본 구조

다음은 if와 else를 이용한 조건문의 기본 구조이다.

```
if 조건문:
    수행할 문장1
    수행할 문장2
    ...
else:
    수행할 문장A
```

```
    수행할 문장B
    ...
```

조건문을 테스트해서 참이면 if문 바로 다음의 문장(if 블록)들을 수행하고, 조건문이 거짓이면 else문 다음 의 문장(else 블록)들을 수행하게 된다. 그러므로 else문은 if문 없이 독립적으로 사용할 수 없다.

들여쓰기

if문을 만들 때는 if 조건문: 바로 아래 문장부터 if문에 속하는 모든 문장에 들여쓰기(indentation)를 해주어야 한다. 다음과 같이 조건문이 참일 경우 "수행할 문장1"을 들여쓰기 했고 "수행할 문장2"와 "수행할 문장3"도 들여쓰기를 해주었다. 다른 프로그래밍 언어를 사용했던 사람들은 파이썬에서 "수행할 문장"들을 들여쓰기 하는 것을 무시하는 경우가 많으니 더 주의해야 한다.

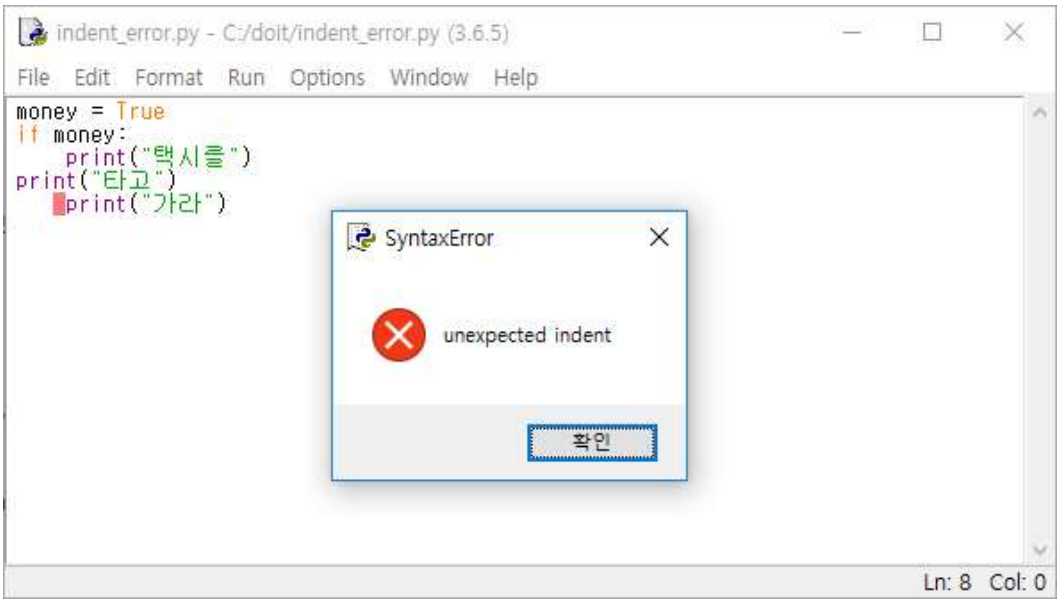
```
if 조건문:
    수행할 문장1
    수행할 문장2
    수행할 문장3
```

다음과 같이 작성하면 오류가 발생한다. "수행할 문장2"를 들여쓰기 하지 않았기 때문이다.

```
if 조건문:
    수행할 문장1
수행할 문장2
    수행할 문장3
```

IDLE 에디터에서 다음과 같이 작성하여 실행해 보자.

```
money = True
if money:
    print("택시를")
print("타고")
    print("가라")
```



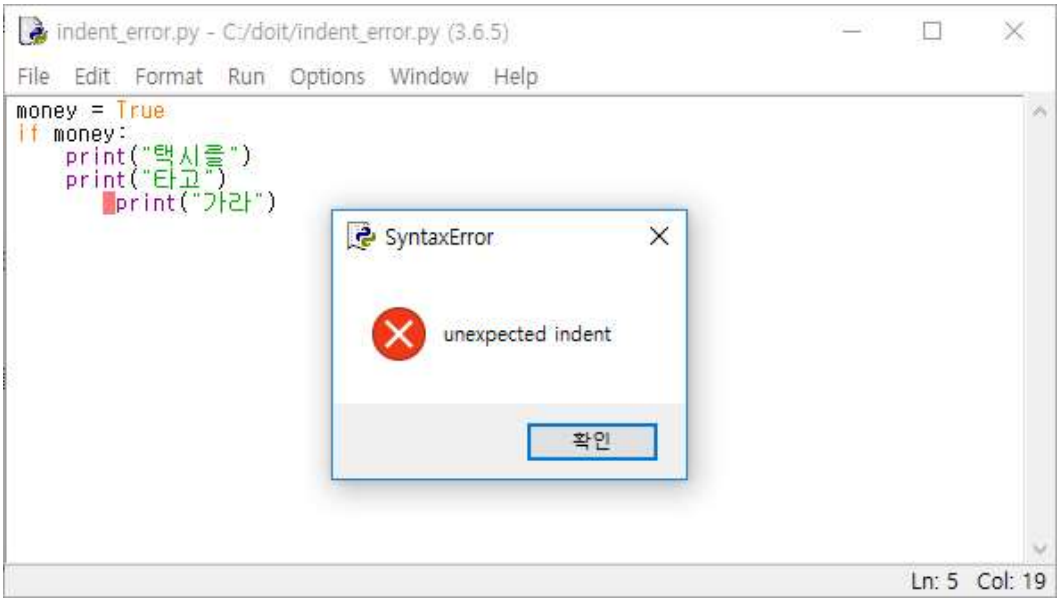
들여쓰기 오류가 발생하는 것을 확인할 수 있다.

다음과 같은 경우에도 에러가 발생한다. "수행할 문장3"을 들여쓰기 했지만 "수행할 문장1"이나 "수행할 문장2"와 들여 쓰기의 깊이가 다르다. 즉, 들여쓰기는 언제나 같은 깊이로 해야 한다.

```
if 조건문:
    수행할 문장1
    수행할 문장2
        수행할 문장3
```

IDLE 에디터에서 다음과 같이 작성하여 실행해 보자.

```
money = True
if money:
    print("택시를")
    print("타고")
        print("가라")
```



마찬가지로 들여쓰기 오류가 발생하는 것을 확인할 수 있다.

그렇다면 들여쓰기는 공백[`Spacebar`]으로 하는 것이 좋을까? 아니면 탭[`Tab`]으로 하는 것이 좋을까? 이에 대한 논란은 파이썬을 사용하는 사람들 사이에서 아직도 계속되고 있다. 탭으로 하자는 쪽과 공백으로 하자는 쪽 모두가 동의하는 내용은 단 하나, 2가지를 혼용해서 쓰지는 말자는 것이다. 공백으로 할 거면 항상 공백으로 통일하고, 탭으로 할 거면 항상 탭으로 통일해서 사용하자는 말이다. 탭이나 공백은 프로그램 소스에서 눈으로 보이는 것이 아니기 때문에 혼용해서 쓰면 에러의 원인이 되기도 한다. 주의하도록 하자.

※ 요즘 파이썬 커뮤니티에서는 들여쓰기를 할 때 공백(`Spacebar`) 4개를 사용하는 것을 권장한다.

[조건문 다음에 콜론(:)을 잊지 말자!]

if 조건문 뒤에는 반드시 콜론(:)이 붙는다. 어떤 특별한 의미가 있다기보다는 파이썬의 문법 구조이다. 왜 하필 콜론(:)인지 궁금하다면 파이썬을 만든 귀도에게 직접 물어보아야 할 것이다. 앞으로 배우게 될 while이나 for, def, class문

에도 역시 문장의 끝에 콜론(:)이 항상 들어간다. 초보자들은 이 콜론(:)을 빠뜨리는 경우가 많으니 특히 주의하자.

파이썬이 다른 언어보다 보기 쉽고 소스 코드가 간결한 이유는 바로 콜론(:)을 사용하여 들여쓰기(indentation)를 하도록 만들었기 때문이다. 하지만 이는 숙련된 프로그래머들이 파이썬을 처음 접할 때 제일 혼란스러워하는 부분이기도 하다. 다른 언어에서는 if문을 { } 기호로 감싸지만 파이썬에서는 들여쓰기로 해결한다는 점을 기억하자.

조건문이란 무엇인가?

if 조건문에서 "조건문"이란 참과 거짓을 판단하는 문장을 말한다.

이전에 살펴보았던 택시 예제에서 조건문에 해당되는 문장은 money이다.

```
>>> money = True
>>> if money:
```

money는 True이기 때문에 참이 되어 if문 다음의 문장을 수행하게 된다.

비교연산자

이번에는 조건문에 비교연산자(`<`, `>`, `==`, `!=`, `>=`, `<=`)를 쓰는 방법에 대해서 알아보자.

다음 표는 비교연산자를 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

이제 위의 연산자들을 어떻게 사용하는지 알아보자.

```
>>> x = 3
>>> y = 2
>>> x > y
True
>>>
```

x에 3을, y에 2를 대입한 다음에 `x > y` 라는 조건문을 수행하면 True를 리턴한다. `x > y` 라는 조건문이 참이기 때문이다.

```
>>> x < y
False
```

위의 조건문은 거짓이기 때문에 False를 리턴한다.

```
>>> x == y
False
```

x와 y는 같지 않다. 따라서 위의 조건문은 거짓이다.

```
>>> x != y
True
```

x와 y는 같지 않다. 따라서 위의 조건문은 참이다.

앞에서 살펴본 택시 예제를 다음처럼 바꾸려면 어떻게 해야 할까?

> "만약 3000원 이상의 돈을 가지고 있으면 택시를 타고 그렇지 않으면 걸어 가라"

위의 상황은 다음처럼 프로그래밍할 수 있다.

```
>>> money = 2000
>>> if money >= 3000:
...     print("택시를 타고 가라")
... else:
...     print("걸어가라")
...
걸어가라
>>>
```

`money >= 3000` 이라는 조건문이 거짓이 되기 때문에 else문 다음의 문장을 수행하게 된다.

and, or, not

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

연산자	설명
x or y	x와 y 둘중에 하나만 참이면 참이다
x and y	x와 y 모두 참이어야 참이다
not x	x가 거짓이면 참이다

다음의 예를 통해 or 연산자의 사용법을 알아보자.

> "돈이 3000원 이상 있거나 카드가 있다면 택시를 타고 그렇지 않으면 걸어 가라"

```
>>> money = 2000
>>> card = True
```

```
>>> if money >= 3000 or card:
...     print("택시를 타고 가라")
... else:
...     print("걸어가라")
...
택시를 타고 가라
>>>
```

money는 2000이지만 card가 True이기 때문에 `money >= 3000 or card` 라는 조건문이 참이 된다. 따라서 if문 다음의 "택시를 타고 가라"라는 문장이 수행된다.

x in s, x not in s

더 나아가 파이썬은 다른 프로그래밍 언어에서 쉽게 볼 수 없는 재미있는 조건문들을 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

in이라는 영어 단어의 뜻이 "~안에"라는 것을 생각해 보면 다음 예들이 쉽게 이해될 것이다.

```
>>> 1 in [1, 2, 3]
True
>>> 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]이라는 리스트 안에 1이 없는가?"라는 조건문이다. 1은 [1, 2, 3] 안에 있으므로 거짓이 되어 False를 리턴한다.

다음은 튜플과 문자열에 적용한 예이다. 각각의 결과가 나온 이유는 쉽게 유추할 수 있다.

```
>>> 'a' in ('a', 'b', 'c')
True
>>> 'j' not in 'python'
True
```

이번에는 우리가 계속 사용해 온 택시 예제에 in을 적용해 보자.

"만약 주머니에 돈이 있으면 택시를 타고, 없으면 걸어 가라"

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

```
...     print("걸어가라")
...
택시를 타고 가라
>>>
```

['paper', 'cellphone', 'money']라는 리스트 안에 'money'가 있으므로 'money' in pocket은 참이 된다. 따라서 if문 다음의 문장이 수행된다.

[조건문에서 아무 일도 하지 않게 설정하고 싶다면?]

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

> "주머니에 돈이 있으면 가만히 있고 주머니에 돈이 없으면 카드를 꺼내라"

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

```
>>> pocket = ['paper', 'money', 'cellphone']
>>> if 'money' in pocket:
...     pass
... else:
...     print("카드를 꺼내라")
...
```

pocket이라는 리스트 안에 money라는 문자열이 있기 때문에 if문 다음 문장인 pass가 수행되고 아무런 결과값도 보여 주지 않는다.

다양한 조건을 판단하는 elif

if와 else만으로는 다양한 조건을 판단하기 어렵다. 다음과 같은 예를 보더라도 if와 else만으로는 조건을 판단하는 데 어려움을 겪게 된다.

> "주머니에 돈이 있으면 택시를 타고, 주머니에 돈은 없지만 카드가 있으면 택시를 타고, 돈도 없고 카드도 없으면 걸어 가라"

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

if와 else만으로 위의 문장을 표현하려면 다음과 같이 할 수 있다.

```
>>> pocket = ['paper', 'handphone']
>>> 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 elif라는 것을 사용한다.

위의 예를 elif를 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 문장1>
    <수행할 문장2>
    ...
elif <조건문>:
    <수행할 문장1>
    <수행할 문장2>
    ...
elif <조건문>:
    <수행할 문장1>
    <수행할 문장2>
    ...
...
else:
   <수행할 문장1>
   <수행할 문장2>
   ...
```

위에서 볼 수 있듯이 elif는 개수에 제한 없이 사용할 수 있다.

[if문을 한 줄로 작성하기]

앞의 pass를 사용한 예를 보면 if문 다음에 수행할 문장이 한 줄이고, else문 다음에 수행할 문장도 한 줄밖에 되지 않는다.

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

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

```
>>> pocket = ['paper', 'money', 'cellphone']
>>> if 'money' in pocket: pass
... else: print("카드를 꺼내라")
...
```

if문 다음의 수행할 문장을 콜론(:) 뒤에 바로 적어 주었다. else문 역시 마찬가지이다.

조건부 표현식

다음과 같은 코드를 보자.

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

위 코드는 score가 60 이상일 경우 message에 "success"를 아닐 경우에는 "failure"를 대입하는 코드이다.

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

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

조건부 표현식은 다음과 같이 정의된다.

```
조건문이_참인_경우 if 조건문 else 조건문이_거짓인_경우
```

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

연습문제

(연습문제 풀이 : https://wikidocs.net/17090#03-1-if)

[문제1] 조건문

다음 코드의 결과값은 무엇일까?

```
>>> a = "Life is too short, you need python"
>>> if 'wife' in a:
...     print('wife')
... elif 'python' in a and 'you' not in a:
...     print('python')
... elif 'shirt' not in a:
...     print('shirt')
... elif 'need' in a:
...     print('need')
```

```
... else:
...     print('none')
```

마지막 편집일시 : 2018년 12월 30일 8:50 오후

댓글 38 피드백

이전글 : 03장 프로그램의 구조를 쌓는다! 제어문

다음글 : 03-2 while문

03-2 while문

while문의 기본 구조

반복해서 문장을 수행해야 할 경우 while문을 사용한다. 그래서 while문을 반복문이라고도 부른다.

다음은 while문의 기본 구조이다.

```
while <조건문>:
    <수행할 문장1>
    <수행할 문장2>
    <수행할 문장3>
    ...
```

while문은 조건문이 참인 동안에 while문 아래에 속하는 문장들이 반복해서 수행된다.

"열 번 찍어 안 넘어 가는 나무 없다" 라는 속담을 파이썬 프로그램으로 만든다면 다음과 같이 될 것이다.

```
>>> 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.
```

위의 예에서 while문의 조건문은 treeHit < 10 이다. 즉, treeHit가 10보다 작은 동안에 while문 안의 문장들을 계속 수행한다. whlie문 안의 문장을 보면 제일 먼저 treeHit = treeHit + 1로 treeHit 값이 계속 1씩 증가한다. 그리고 나무를 treeHit번만큼 찍었음을 알리는 문장을 출력하고 treeHit가 10이 되면 "나무 넘어갑니다."라는 문장을 출력한다. 그러고 나면 treeHit < 10 이라는 조건문이 거짓이 되므로 while문을 빠져나가게 된다.

※ treeHit = treeHit + 1 은 프로그래밍을 할 때 매우 자주 사용하는 기법이다. treeHit의 값을 1만큼씩 증가시킬 목적으로 사용되며, treeHit +=1 처럼 사용되기도 한다.

다음은 while문이 반복되는 과정을 순서대로 정리한 표이다. 이렇게 긴 과정을 소스코드 단 5줄로 만들 수 있다니 놀랍지 않은가?

treeHit	조건문	조건판단	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

treeHit	조건문	조건판단	수행하는 문장	while문
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

while문 만들기

다음을 직접 입력해 보자. 여러 가지 선택지 중 하나를 선택해서 입력받는 예제이다. 먼저 다음과 같이 여러 줄짜리 문자열을 만들어 보자.

```
>>> prompt = """
... 1. Add
... 2. Del
... 3. List
... 4. Quit
...
... Enter number: """
>>>
```

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

```
>>> number = 0
>>> while number != 4:
...     print(prompt)
...     number = int(input())
...

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

Enter number:
```

while문을 보면 number가 4가 아닌 동안 prompt를 출력하고 사용자로부터 번호를 입력받는다. 다음의 결과 화면처럼 사용자가 4라는 값을 입력하지 않으면 계속해서 prompt를 출력한다.

※ 여기서 number = int(input())는 사용자의 숫자 입력을 받아들이는 것이라고만 알아두자. int나 input 함수에 대한 내용은 뒤의 내장 함수 부분에서 자세하게 다룬다.

```
Enter number:
1

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

4를 입력하면 조건문이 거짓이 되어 while문을 빠져나가게 된다.

```
Enter number:
4
>>>
```

while문 강제로 빠져나가기

while문은 조건문이 참인 동안 계속해서 while문 안의 내용을 반복적으로 수행한다. 하지만 강제로 while문을 빠져나가고 싶을 때가 있다. 예를 들어 커피 자판기를 생각해 보자. 자판기 안에 커피가 충분히 있을 때에는 동전을 넣으면 커피가 나온다. 그런데 자판기가 제대로 작동하려면 커피가 얼마나 남았는지 항상 검사해야 한다. 만약 커피가 떨어졌다면 판매를 중단하고 "판매 중지"라는 문구를 사용자에게 보여주어야 한다. 이렇게 판매를 강제로 멈추게 하는 것이 바로 break문이다.

다음의 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 있으므로 while money:에서 조건문인 money는 0이 아니기 때문에 항상 참이다. 따라서 무한히 반복되는 무한 루프를 돌게 된다. 그리고 while문의 내용을 한 번 수행할 때마다 coffee = coffee - 1에 의해서 coffee의 개수가 1개씩 줄어든다. 만약 coffee가 0이 되면 `if coffee == 0:` 이라는 문장에서 `coffee == 0` 이 참이 되므로 if문 다음의 문장인 "커피가 다 떨어졌습니다. 판매를 중지합니다."가 수행되고 break문이 호출되어 while문을 빠져나가게 된다.

하지만 실제 자판기는 위의 예처럼 작동하지는 않을 것이다. 다음은 자판기의 실제 작동 과정과 비슷하게 만들어 본 예이다. 이해가 안 되더라도 걱정하지 말자. 아래의 예는 조금 복잡하니까 대화형 인터프리터를 이용하지 말고 IDLE 에디터를 이용해서 작성해 보자.

```
# coffee.py

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
```

위의 프로그램 소스를 따로 설명하지는 않겠다. 여러분이 소스를 입력하면서 무슨 내용인지 이해할 수 있다면 지금껏 배운 if문이나 while문을 이해했다고 보면 된다. 만약 `money = int(input("돈을 넣어 주세요:"))` 라는 문장이 이해되지 않는다면 이 문장은 사용자로부터 입력을 받는 부분이고 입력받은 숫자를 money라는 변수에 대입하는 것이라고만 알아두자.

이제 coffee.py 파일을 저장한 후 명령 프롬프트 창에서 다음과 같이 실행해 보자. 아래와 같은 입력란이 나타난다.

```
C:\doit>python coffee.py
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:
```

while문의 맨 처음으로 돌아가기

while문 안의 문장을 수행할 때 입력된 조건을 검사해서 조건에 맞지 않으면 while문을 빠져나간다. 그런데 프로그래밍을 하다 보면 while문을 빠져나가지 않고 while문의 맨 처음(조건문)으로 다시 돌아가게 만들고 싶은 경우가 생기게 된다. 이때 사용하는 것이 바로 continue문이다.

만약 1부터 10까지의 숫자 중에서 홀수만 출력하는 것을 while문을 이용해서 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
```

위의 예는 1부터 10까지의 숫자 중 홀수만을 출력하는 예이다. a가 10보다 작은 동안 a는 1만큼씩 계속 증가한다. `if a % 2 == 0` (a를 2로 나누었을 때 나머지가 0인 경우)이 참이 되는 경우는 a가 짝수일 때이다. 즉, a가 짝수이면 continue 문장을 수행한다. 이 continue문은 while문의 맨 처음(조건문: `a<10`)으로 돌아가게 하는 명령어이다. 따라서 위의 예에서 a가 짝수이면 print(a)는 수행되지 않을 것이다.

무한 루프

이번에는 무한 루프(Loop)에 대해서 알아보자. 무한 루프란 무한히 반복한다는 의미이다. 우리가 사용하는 일반적인 프로그램 중에서 무한 루프의 개념을 사용하지 않는 프로그램은 거의 없다. 그만큼 자주 사용된다는 뜻이다.

파이썬에서 무한 루프는 while문으로 구현할 수 있다. 다음은 무한 루프의 기본적인 형태이다.

```
while True:
    수행할 문장1
    수행할 문장2
    ...
```

while문의 조건문이 True이므로 항상 참이 된다. 따라서 while문 안에 있는 문장들은 무한하게 수행될 것이다.

다음의 무한 루프 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.
....
```

위의 문장은 영원히 출력된다. 하지만 이 예처럼 아무 의미 없이 무한 루프를 돌리는 경우는 거의 없을 것이다. `[Ctrl+C]` 를 눌러 빠져나가도록 하자.

연습문제

(연습문제 풀이 : https://wikidocs.net/17090#03-2-while)

[문제1] 3의 배수의 합

1부터 1000까지의 자연수 중 3의 배수의 합을 구하시오.

[문제2] 50점 이상의 총합

다음은 A학급 학생의 점수를 나타내는 리스트이다. 다음 리스트에서 50점 이상의 점수들의 총합을 구하시오.

```
A = [20, 55, 67, 82, 45, 33, 90, 87, 100, 25]
```

[문제3] 별 표시하기1

while문을 이용하여 아래와 같이 별(`*`)을 표시하는 프로그램을 작성해 보자.

```
*
**
***
****
```

마지막 편집일시 : 2018년 12월 30일 9:23 오후

이전글 : 03-1 if문

다음글 : 03-3 for문

03-3 for문

파이썬의 직관적인 특징을 가장 잘 대변해 주는 것이 바로 이 for문이다. while문과 비슷한 반복문인 for문은 매우 유용하고 문장 구조가 한눈에 쏙 들어온다는 장점이 있다. for문을 잘 사용하면 프로그래밍이 때때로 즐거워질 것이다.

for문의 기본 구조

for문의 기본적인 구조는 다음과 같다.

```
for 변수 in 리스트(또는 튜플, 문자열):
    수행할 문장1
    수행할 문장2
    ...
```

리스트나 튜플, 문자열의 첫 번째 요소부터 마지막 요소까지 차례로 변수에 대입되어 "수행할 문장1", "수행할 문장2" 등이 수행된다.

예제로 for문 이해하기

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

1. 전형적인 for문

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

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

2. 다양한 for문의 사용

```
>>> a = [(1,2), (3,4), (5,6)]
>>> for (first, last) in a:
...     print(first + last)
...
3
7
11
```

위의 예는 a 리스트의 요소값이 튜플이기 때문에 각각의 요소들이 자동으로 (first, last)라는 변수에 대입된다.

※ 이 예는 02장에서 살펴보았던 튜플을 이용한 변수값 대입 방법과 매우 비슷한 경우이다.

```
>>> (first, last) = (1, 2)
```

3. for문의 응용

for문의 쓰임새를 알기 위해 다음을 가정해 보자.

"총 5명의 학생이 시험을 보았는데 시험 점수가 60점이 넘으면 합격이고 그렇지 않으면 불합격이다. 합격인지 불합격인지 결과를 보여주시오."

우선 학생 5명의 시험 점수를 리스트로 표현해 보았다.

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

1번 학생은 90점이고 5번 학생은 80점이다.

이런 점수를 차례로 검사해서 합격했는지 불합격했는지 통보해 주는 프로그램을 만들어 보자. 역시 IDLE 에디터로 작성한다.

```
# marks1.py
marks = [90, 25, 67, 45, 80]

number = 0
for mark in marks:
    number = number +1
    if mark >= 60:
        print("%d번 학생은 합격입니다." % number)
    else:
        print("%d번 학생은 불합격입니다." % number)
```

각각의 학생에게 번호를 붙여 주기 위해 number라는 변수를 이용하였다. 점수 리스트인 marks에서 차례로 점수를 꺼내어 mark라는 변수에 대입하고 for문 안의 문장들을 수행하게 된다. 우선 for문이 한 번씩 수행될 때마다 number는 1씩 증가한다.

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

```
C:\doit>python marks1.py
1번 학생은 합격입니다.
2번 학생은 불합격입니다.
3번 학생은 합격입니다.
4번 학생은 불합격입니다.
5번 학생은 합격입니다.
```

for문과 continue

while문에서 살펴보았던 continue를 for문에서도 사용할 수 있다. 즉, for문 안의 문장을 수행하는 도중에 continue문을 만나면 for문의 처음으로 돌아가게 된다.

앞서 for문 응용 예제를 그대로 이용해서 60점 이상인 사람에게는 축하 메시지를 보내고 나머지 사람에게는 아무런 메시지도 전하지 않는 프로그램을 에디터를 이용해 작성해 보자.

```
# marks2.py
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)
```

점수가 60점 이하인 학생일 경우에는 mark < 60이 참이 되어 continue문이 수행된다. 따라서 축하 메시지를 출력하는 부분인 print문을 수행하지 않고 for문의 처음으로 돌아가게 된다.

```
C:\doit>python marks2.py
1번 학생 축하합니다. 합격입니다.
3번 학생 축하합니다. 합격입니다.
5번 학생 축하합니다. 합격입니다.
```

for문과 range함수

for문은 숫자 리스트를 자동으로 만들어 주는 range라는 함수와 함께 사용되는 경우가 많다. 다음은 range 함수의 간단한 사용법이다.

```
>>> a = range(10)
>>> a
range(0, 10)
```

range(10)은 0부터 10 미만의 숫자를 포함하는 range 객체를 만들어 준다.

시작 숫자와 끝 숫자를 지정하려면 range(시작 숫자, 끝 숫자) 형태를 사용하는데, 이때 끝 숫자는 포함되지 않는다.

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

range 함수의 예시 살펴보기

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

```
>>> sum = 0
>>> for i in range(1, 11):
...     sum = sum + i
...
>>> print(sum)
55
```

range(1, 11)은 숫자 1부터 10까지(1 이상 11 미만)의 숫자를 데이터로 갖는 객체이다. 따라서 위의 예에서 i 변수에 리스트의 숫자들이 1부터 10까지 하나씩 차례로 대입되면서 sum = sum + i라는 문장을 반복적으로 수행하고 sum은 최종적으로 55가 된다.

또한 우리가 앞서 살펴보았던 60점 이상이면 합격이라는 문장을 출력하는 예제도 range 함수를 이용해서 바꿀 수 있다. 다음을 보자.

```
#marks3.py
marks = [90, 25, 67, 45, 80]
for number in range(len(marks)):
    if marks[number] < 60:
        continue
    print("%d번 학생 축하합니다. 합격입니다." % (number+1))
```

len 함수는 리스트 내 요소의 개수를 돌려주는 함수이다. 따라서 len(marks)는 5가 될 것이고 range(len(marks))는 range(5)가 될 것이다. number 변수에는 차례로 0부터 4까지의 숫자가 대입될 것이고, marks[number]는 차례대로 90, 25, 67, 45, 80이라는 값을 갖게 된다. 결과는 marks2.py 예제와 동일하다.

for와 range를 이용한 구구단

for와 range 함수를 이용하면 소스 코드 단 4줄만으로 구구단을 출력할 수 있다. 들여쓰기에 주의하며 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
```

위의 예를 보면 for문이 두 번 사용되었다. ①번 for문에서 2부터 9까지의 숫자(range(2, 10))가 차례로 i에 대입된다. i가 처음 2일 때 ②번 for문을 만나게 된다. ②번 for문에서 1부터 9까지의 숫자(range(1, 10))가 j에 대입되고 그 다음 문장인 `print(i*j)` 를 수행한다.

따라서 i가 2일 때 `2*1, 2*2, 2*3, … 2*9` 까지 차례로 수행되며 그 값을 출력하게 된다. 그 다음으로 i가 3일 때 역시 2일 때와 마찬가지로 수행될 것이고 i가 9일 때까지 계속 반복된다.

[입력 인수 end를 넣어 준 이유는 무엇일까?]

앞의 예제에서 `print(i*j, end=" ")` 와 같이 입력 인수 end를 넣어 준 이유는 해당 결과값을 출력할 때 다음줄로 넘기지 않고 그 줄에 계속해서 출력하기

위해서이다. 그 다음에 이어지는 print(' ')는 2단, 3단 등을 구분하기 위해 두 번째 for문이 끝나면 결과값을 다음 줄부터 출력하게 해주는 문장이다.

리스트 안에 for문 포함하기

리스트 안에 for문을 포함하는 리스트 내포(List comprehension)를 이용하면 좀 더 편리하고 직관적인 프로그램을 만들 수 있다. 다음의 예제를 보자.

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

위 예제는 a라는 리스트의 각 항목에 3을 곱한 결과를 result라는 리스트에 담는 예제이다.

이것을 리스트 내포를 이용하면 아래와 같이 간단히 해결할 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 짝수인 2와 4에만 3을 곱하여 담고 싶다면 다음과 같이 리스트 내포 안에 "if 조건문"을 사용할 수도 있다.

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

리스트 내포의 일반적인 문법은 다음과 같다. "if 조건문" 부분은 앞의 예제에서 볼 수 있듯이 생략할 수 있다.

```
[표현식 for 항목 in 반복가능객체 if 조건문]
```

조금 복잡하지만 for문을 2개 이상 사용하는 것도 가능하다. for문을 여러 개 사용할 때의 문법은 다음과 같다.

```
[표현식 for 항목1 in 반복가능객체1 if 조건문1
        for 항목2 in 반복가능객체2 if 조건문2
        ...
        for 항목n in 반복가능객체n if 조건문n]
```

만약 구구단의 모든 결과를 리스트에 담고 싶다면 리스트 내포를 이용하여 아래와 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]
```

지금껏 우리는 프로그램의 흐름을 제어하는 if문, while문, for문에 대해서 알아보았다. 아마도 여러분은 while문과 for문을 보면서 2가지가 아주 비슷하다는 느낌을 받았을 것이다. 실제로 for문을 사용한 부분을 while문으로 바꿀 수 있는 경우도 많고, while문을 for문으로 바꾸어서 사용할 수 있는 경우도 많다.

연습문제

(연습문제 풀이 : https://wikidocs.net/17090#03-3-for)

[문제1] 1부터 100까지 출력

1부터 100까지의 숫자를 for문을 이용하여 출력하시오.

[문제2] 학급의 평균 점수

for문을 이용하여 A 학급의 평균 점수를 구해 보자.

```
A = [70, 60, 55, 75, 95, 90, 80, 80, 85, 100]
```

[문제3] 리스트 내포

리스트 중에서 홀수에만 2를 곱하여 저장하는 다음과 같은 코드가 있다.

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

result = []
for n in numbers:
    if n % 2 == 1:
        result.append(n*2)
```

위 코드를 리스트 내포(list comprehension)를 이용하여 표현하시오.

마지막 편집일시 : 2018년 12월 30일 9:02 오후

이전글 : 03-2 while문

다음글 : 04장 프로그램의 입력과 출력은 어떻게 해야 할까?