2
함수 만들기
Function Abstraction

프로그램 작성 작업을 코딩(coding)이라고 하는데, 일반적으로 기능코딩(functional coding)과 안전코딩(secure coding)의 순서로 진행한다. 기능코딩 단계에서는 요구사항에 명시한대로 프로그램이 잘 작동하는데 집중하고, 기능이 완벽하게 작동하면 다음 단계인 안전코딩으로 넘어간다. 기능코딩과 안전코딩을 사례를 통해서 알아보면서, 조건문과 반복문, 함수 정의/호출의 개념을 익혀보자.

1. 기능코딩

사례 : 날짜-분 변환

다음은 사용자로부터 날짜(day) 수를 입력 받아서 그 날짜에 해당하는 분(minute) 수를 계산하는 프로그램이다. 프로그램을 읽고 이해한 후, 편집창에 입력하고 실행해보자. 입력받은 정수는 정수 문자열이므로 계산하기 전 정수로 바꾸어야 한다.

```
1 print("날짜를 분으로 환산해드립니다.")
2 days = int(input("며칠?"))
3 hours = days * 24 * 60
4 print(days, "일을 분으로 따지면", hours, "분이다.")
```

이 프로그램에 정수를 입력하면 기대한 대로 맞는 답을 주면서 프로그램이 정상적으로 종료한다. 즉, 이 프로그램은 기능적으로 편집기로 이 프로그램을 `d2m.py` 이름의 파일에 저장한 후, (음수를 포함한) 다양한 정수를 입력하여 여러번 실행하여 잘 작동하는지 확인해보자. 입력 사례: `5`, `-7`, `+3`, `0`
요구한 대로 기능코딩을 완성하였다.

사례 : 날짜-분 변환 (요구사항 변화에 따른 기능 개선)

음수를 입력하면 어떤 결과가 나오나? 음수 계산이 굳이 필요할까? 이 프로그램 기획자가 음수는 별 의미가 없으니 0 이상의 자연수 입력만 처리하도록 요구사항을 변경했다고 하자. 그리고 음수 입력은 계산을 하지 말고 "음수는 취급하지 않습니다."라는 메시지를 보여주자고 한다. 이 새로운 요구사항에 맞추어 프로그램을 다음과 같이 수정할 수 있다.

```
1 print("날짜를 분으로 환산해드립니다.")
2 days = int(input("며칠?"))
3 if days >= 0 :
4     hours = days * 24 * 60
5     print(days,"일을 분으로 따지면", hours, "분이다.")
6 else :
7     print("음수는 취급하지 않습니다.")
```

수정한 프로그램을 위와 같은 입력사례로 실행하여 어떤 결과가 나오는지 관찰해보자. 0 이상의 정수 입력에 대해서는 이전 프로그램과 동일하게 작동한다. 그리고 음의 정수를 입력하면 계산을 생략하고 정해둔 메시지를 프린트한다. 정수 입력에 대해서 요구사항에 충실하게 맞추어 기능코딩을 완성하였다.

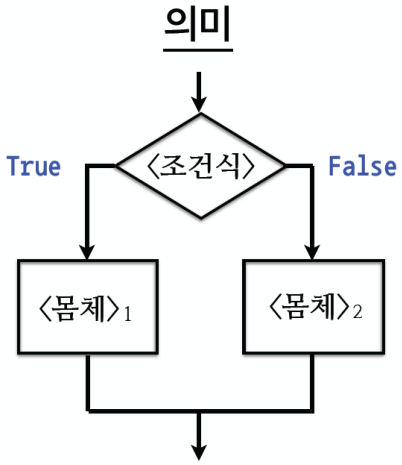
어떻게 이런 결과가 나왔을지 이 개선한 프로그램을 읽어보고 실행의미를 추측해보자. 여기서 `if ... else ...` 는 조건문이라고 하는데, 조건문의 문법과 의미를 공부해보자.

조건문

조건문(conditional command)의 **문법(syntax, grammar)**은 다음과 같다.

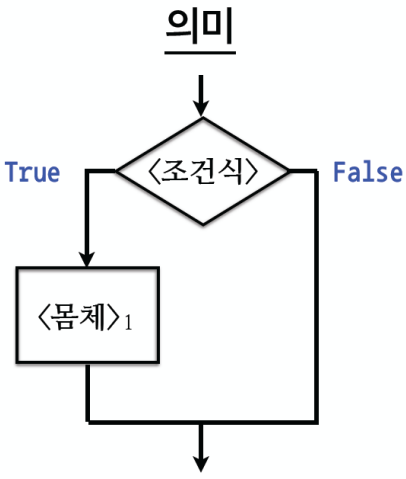
```
if 〈조건식〉 :
    〈몸체〉1
else :
    〈몸체〉2
```

〈조건식〉은 논리식이며, 〈몸체〉에는 한 줄 이상의 명령문(command)을 나열할 수 있다. 지금까지 배운 명령문으로는 지정문과 프린트문, 조건문이 있다.



조건문의 **의미(semantics, meaning)**는 바로 위의 그림과 같이 흐름도(flow chart)로 표현할 수도 있다. 〈조건식〉를 계산하여 결과가 `True`이면 〈몸체〉1을 실행하고, `False`이면 〈몸체〉2을 실행한다.

조건문의 〈몸체〉는 반드시 일정하게 **들여쓰기(indentation)**를 해야하는데, 같은 간격으로 띄어져있는 명령문 덩어리 전체를 블록(block)이라 한다. 일반적으로 Python은 4칸씩 들여쓰기를 권장하므로 우리도 그렇게 하기로 하자. 들여쓰기 일정하게 잘 맞추어야 한다. 즉, 〈몸체〉1과 〈몸체〉2는 같은 간격으로 들여쓰기를 해야한다. 들여쓰기가 일정하지 않으면 구문오류로 처리된다. 바로 위의 사례 코드에서 조건문 내의 두 블록이 각각 4칸씩 들여쓰기가 되어있음을 확인해보자. (Python 전용 편집기는 자동으로 들여쓰기를 해주기도 한다.)



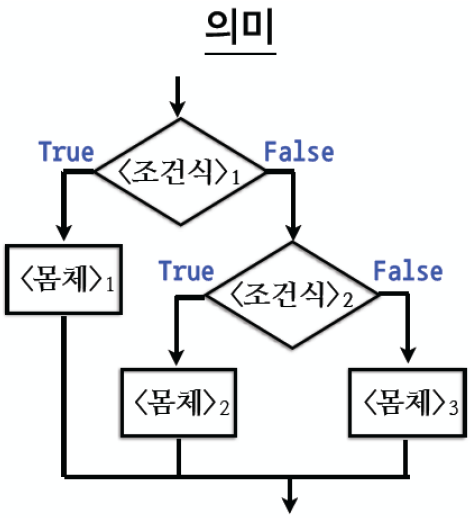
여기서 `else` 이하는 다음과 같이 생략해도 된다.

```
if 〈조건식〉 :
    〈몸체〉1
```

의미는 오른쪽의 흐름도와 같다.

조건이 둘 이상이고 조건에 따라서 다르게 계산해야하는 경우, 아래와 같이 elif 절을 추가하면 된다. elif 절은 여러개 있을 수 있다. 여기서도 마지막 else 절은 생략할 수 있다.

```
if 〈조건식〉1 :
    〈몸체〉1
elif 〈조건식〉2 :
    〈몸체〉2
else :
    〈몸체〉3
```



〈조건식〉1의 계산결과가 True이면 〈몸체〉1을 실행하고, 〈조건식〉1의 계산결과가 False이고 〈조건식〉2의 계산결과가 True이면 〈몸체〉2을 실행하고, 〈조건식〉2의 계산결과도 False이면 〈몸체〉3을 실행한다.

이제 조건문을 이해했는지 확인해보자. 다음 프로그램을 먼저 눈으로 읽어서 어떤 프로그램인지 이해하고, 직접 실행하여 제대로 이해했는지 확인해보자.

```
1  score = input('Enter your score : ')
2  score = int(score)
3  if 90 <= score <= 100:
4      print('A')
5  elif 80 <= score <= 89:
6      print('B')
7  elif 70 <= score <= 79:
8      print('C')
9  elif 60 <= score <= 69:
10     print('D')
11 elif 0 <= score <= 59:
12     print('F')
13 else:
14     print('Invalid input!')
```

2. 안전코딩 - 입력확인

날짜를 분으로 환산하는 프로그램을 다시 한번 음미해보자. 이 프로그램은 정수 입력에 대해서 완벽히 작동하도록 작성하였다. 즉, 0 이상의 정수에 대해서는 날짜를 분으로 환산을 해주고, 0 미만의 정수에 대해서는 음수는 취급하지 않는다는 메시지를 출력해주도록 하였다. 즉, 입력이 정수로 주어지는 한 정상적으로 작동하는 프로그램이다. 그러나 정수가 아닌 입력에 대해서는 ValueError 오류메시지와 함께 비정상적으로 종료한다. 실행창에서 직접 확인해보자.

프로그램 사용자에게 아무리 친절히 안내해도 사용자가 안내하는대로 착하게 입력하리라 단정하는 건 금물이다. 따라서 사용자 입력은 모두 **입력확인**(input validation)을 반드시 해야 한다. 즉, 사용자 입

력이 프로그램에서 기대하고 있는 부류인지 반드시 확인하여 그렇지 않은 경우 걸러내거나 재입력받도록 프로그램을 짜야한다. 입력확인을 제대로 하지 않아 잘못된 사용자 입력이 프로그램 실행과정에 투입된다면 의도하지 않았던 결과를 초래할 수 있다. 사실 악의적인 해킹이나 소프트웨어의 치명적인 오류는 사용자 입력을 제대로 확인하지 않아 발생하는 경우가 대부분이다. 따라서 입력을 반드시 확인하여 어떤 기형입력에 대해서도 프로그램이 비정상 종료하지 않고 버틸 수 있도록 코딩해야 한다. 이를 **안전코딩**(secure coding)이라고 한다.

사례 : 날짜-분 변환 (입력확인)

위 프로그램에서 0에서 9까지의 숫자가 아닌 다른 문자가 포함된 입력이 들어오는 경우, 무시하고 사용자에게 재입력을 요청하도록 프로그램을 다음과 같이 수정할 수 있다. 사용자 입력을 받아서 정수로 바꾸기 전 입력을 확인하도록 했는데 진하게 표시된 부분이다.

```
1  print("날짜를 분으로 환산해드립니다.")
2  days = input("며칠?")
3  while not days.isdigit():
4      print("숫자가 아닌 문자가 들어있습니다.")
5      days = input("며칠?")
6  days = int(days)
7  if days >= 0 :
8      hours = days * 24 * 60
9      print(days, "일을 분으로 따지면", hours, "분이다.")
10 else :
11     print("음수는 취급하지 않습니다.")
```

while 문을 처음 만났으니 문법과 의미를 먼저 공부한 뒤에 돌아와서 이 프로그램을 이해해보도록 하자.

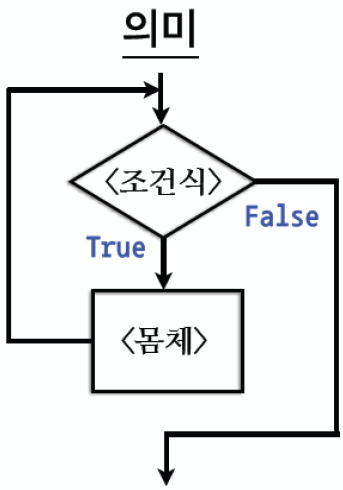
반복문 while

while 반복문은 지정한 조건이 만족하는 동안 동일 블록을 반복 실행하도록 고안한 프로그램 구조이다.

반복문(iteration command)의 문법은 다음과 같다.

```
while 〈조건식〉 :
    〈몸체〉
```

〈조건식〉은 논리식이며, 〈몸체〉는 한 줄 이상의 명령문을 나열할 수 있다. 〈몸체〉에는 지금까지 배운 명령문인 지정문과 프린트문, 조건문 뿐만 아니라 반복문까지도 포함할 수 있다. 〈몸체〉 부분은 조건문과 마찬가지로 네 칸 들여쓰기를 하여 하나의 블록을 이루게 해야 한다.



이 반복문의 의미는 오른쪽 흐름도로 이해할 수 있듯이 〈조건식〉의 계산 결과가 True인 동안 〈몸체〉을 반복 실행하다가, 〈조건식〉의 계산 결과가 False가 되면 바로 실행을 멈춘다.

사례 : 날짜-분 변환 (입력확인 계속)

이제 while 문을 이해했으니 이 프로그램의 의미를 이해해보자.

```
1  print("날짜를 분으로 환산해드립니다.")
2  days = input("며칠?")
3  while not days.isdigit():
4      print("숫자가 아닌 문자가 들어있습니다.")
5      days = input("며칠?")
6  days = int(days)
7  if days >= 0 :
8      hours = days * 24 * 60
9      print(days, "일을 분으로 따지면", hours, "분이다.")
10 else :
11     print("음수는 취급하지 않습니다.")
```

줄 2에서 입력이 정수 문자열이라는 보장이 없으므로 int 함수로 타입변환을 하지 않았다. (정수가 아닌 입력 문자열로 정수로 타입변환을 시도하면 오류가 날 것이기 때문이다.) 그러면 days 변수에는 입력받은 문자열이 지정된다. 줄 3~5는 반복문이다. 조건식인 not days.isdigit() 의 계산 결과가 True가 되는 한될 몸체인 줄 4~5를 반복 실행하여 사용자로부터 재입력을 받는다. 다시 말해, not days.isdigit() 이 False가 될 때까지, 또 다른 말로 days.isdigit() 이 True가 될 때까지, 계속 재입력을 받는다. days.isdigit() 이 생소하므로 따로 공부하자.

문자열 메소드 : isdigit()

Python 언어는 실행기와 함께 표준 라이브러리(The Python Standard Library: 문서[1])를 제공한다. 표준 라이브러리 문서 4.7.1 String Methods 를 뒤져보면 문자열을 처리할 수 있는 유용한 기능의 **메소드(method)**를 많이 찾을 수 있다. (메소드는 문자열이 수행할 수 있는 기능으로 이해하면 된다.) 이 중 하나가 isdigit() 이다.

문자열 메소드는 다음과 같은 형식으로 호출한다.

〈문자열〉.isdigit()

의미는 〈문자열〉을 계산한 문자열이 모두 숫자이면 True, 그렇지 않고 하나라도 숫자가 아닌 문자가 있으면 False를 내준다. 다음 예를 하나씩 실행창에서 확인하면서 의미를 이해해보자.

```
"345".isdigit()
"-45".isdigit()
"4.5".isdigit()
"4 7".isdigit()
```

[1] The Python Standard Library 문서는 https://docs.python.org/2/library/index.html 에서 참고

사례 : 날짜-분 변환 (입력확인 계속)

이제 이 프로그램의 의미를 이해해보자. (위와 동일)

```
1  print("날짜를 분으로 환산해드립니다.")
2  days = input("며칠?")
3  while not days.isdigit():
4      print("숫자가 아닌 문자가 들어있습니다.")
5      days = input("며칠?")
6  days = int(days)
7  if days >= 0 :
8      hours = days * 24 * 60
9      print(days, "일을 분으로 따지면", hours, "분이다.")
10 else :
11     print("음수는 취급하지 않습니다.")
```

이제 `not days.isdigit()`의 의미를 이해할 수 있다. 입력 문자열이 숫자로만 구성되어 있으면 이 조건식이 `False`가 되어 몸체(줄 3~4)를 실행하지 않고 넘어가지만, 하나라도 숫자가 아닌 문자가 포함되어 있으면 조건식이 `True`가 되어 몸체를 실행하여 다시 입력을 받는다. 입력 문자열의 문자가 모두 숫자일 때까지 반복문의 몸체를 재실행한다.

`while` 문을 빠져나와 줄 6에 도달하면 `days` 변수의 값은 숫자로만 구성된 문자열이 확실하다. 이제는 `int`로 타입변환을 해도 오류가 절대 발생하지 않을테므로 안전하다. 줄 6의 지정문을 실행하면 `days` 변수의 값은 정수로 변환하여 재지정된다.

그런데 이제는 입력확인을 거친 입력은 모두 숫자로만 구성되어 있으므로, 음이 될 수 없다. 따라서 `days` 변수이 값이 `0` 이상인지를 검사할 필요가 없어졌다. 따라서 이 조건문을 지우면 다음과 같이 프로그램이 완성된다.

```
1  print("날짜를 분으로 환산해드립니다.")
2  days = input("며칠?")
3  while not days.isdigit() :
4      print("숫자가 아닌 문자가 들어있습니다.")
5      days = input("며칠?")
6  days = int(days)
7  hours = days * 24 * 60
8  print(days, "일을 분으로 따지면", hours, "분이다.")
```

이 프로그램을 다음의 입력으로 각각 실행하고 결과를 관찰하여, 절대 오류가 발생하지 않는 안전한 프로그램임을 확인해보자.

```
nine
-9
+3
0
5
```

사례 : 날짜-분 변환 (서비스의 반복)

위 프로그램은 한번 사용하면 프로그램이 끝나버린다. while 반복문을 활용하여 필요한 만큼 여러번 사용할 수 있도록 할 수 있다. 반복하려는 코드 〈블록〉을 다음과 같이 while True: 로 감싸면 블록 몸체를 무한정 반복하여 실행한다.

```
while True:
    〈블록〉
```

반복을 그만하고 바깥으로 나가고 싶으면 break 명령을 사용할 수 있다. 〈블록〉 안에서 break 명령을 실행하면 break 명령을 포함하고 있는 블록을 감싸고 있는 가장 안쪽 while 문의 바깥으로 실행 흐름이 빠져나간다.

다음 프로그램은 날짜를 분으로 환산하는 계산을 사용자가 원하는 한 계속하다, 사용자가 그만하기를 원하는 경우 프로그램을 멈추도록 작성하였다. 바깥의 while 문(줄 2~14)의 몸체 블록은 줄 3~14이며 블록 내부에 있는 break 명령이 실행되지 않는 한 이 몸체 블록은 반복 실행된다.

몸체 블록 끝 부분의 진하게 표시한 추가 코드의 의미를 잘 새겨보자.

```
1  print("날짜를 분으로 환산해드립니다.")
2  while True:
3      days = input("며칠?")
4      while not days.isdigit() :
5          print("숫자가 아닌 문자가 들어있습니다.")
6          days = input("며칠?")
7      days = int(days)
8      hours = days * 24 * 60
9      print(days, "일을 분으로 따지면", hours, "분이다.")
10     cont = input("계속하시겠습니까?(예/아니오) ")
11     while not (cont == '예' or cont == '아니오'):
12         cont = input("계속하시겠습니까?(예/아니오) ")
13     if (cont == '아니오'):
14         break
```

몸체 블록의 끝부분(줄 10~14)은 사용자로 부터 '예' 또는 '아니오' 입력을 받고, '예'인 경우 계속 반복하지만, '아니오'인 경우 break 문을 실행하게 되어 while 문을 빠져나와 실행을 마친다. 여기서 break 문이 속한 while 문은 2번이지 11번이 아님을 확인하고 넘어가자.

사용자가 원하는 동안 특정한 일을 반복하고 싶은 경우 사용하는 전형적인 패턴이므로 익혀두면 좋다.

3. 함수

컴퓨터를 분해해보면 여러 종류의 작은 부품들이 결합되어 만들어졌음을 알 수 있다. 이 뿐만 아니라 우리 주위에 있는 대부분의 물건들은 작은 부품들을 적절히 잘 골라서 조립하여 만들어진 완성품이다. 많이 쓰이는 부품은 표준을 만들어 규격화하여 대량생산 해놓고 필요에 맞게 골라 쓸 수 있게 하기도 하고, 특정 용도로만 사용되는 부품은 자가 제작하여 쓰기도 한다.

소프트웨어를 구성하고 있는 프로그램도 마찬가지다. 특정한 일을 하는 프로그램을 부품화하여 이름을 붙여놓고 필요할 때마다 불러서 가져다 쓸 수 있으면 좋다. 소프트웨어가 복잡해지면 더욱 더 부품화가 절실해진다. 자주 쓰일만한 부품들은 미리 만들어 모아놓고 필요할 때 가져다 쓰면 편리할 것이다. 이미 공부한 붙박이 함수가 그런 것이다. 어떤 경우에는 특정 용도로 맞춤형 부품을 별도로 자가제작하여 써야할 때도 있다. 소프트웨어 부품 중 하나로 가장 간단하면서 널리 알려진 것이 **함수**(function)이다.

프로그램 안에 같은 코드가 여러번 중복 나타나면서 길어지면 프로그램을 이해하거나 유지관리하기 어려워진다. 코드의 일부분을 수정하는 경우 중복코드를 모두 찾아서 일관성 있게 수정해야 할 수도 있기 때문이다. 만약에 한 군데라도 빠트리게 되면 프로그램 오류를 야기하는 원인이 될 수 있다. 따라서 중복되는 부분은 함수로 만들어두고 필요할 때마다 불러쓰도록 하면 프로그램의 이해가 쉬워지고, 수정이 필요한 경우에 한번만 고치면되므로 유지관리가 쉬워진다.

함수 정의와 호출: 문법과 의미

함수의 작성 행위를 **함수 정의**(function definition)라고 하고, 불러쓰는 행위를 **함수 호출**(function call)이라고 한다.

함수를 정의하는 문법은 다음과 같다.

```
def 〈함수이름〉 ( 〈변수〉1, 〈변수〉2, … , 〈변수〉n ) :
    〈몸체〉
```

〈함수이름〉은 변수 이름을 만드는 규칙과 동일하다. 괄호 안에 들어가는 변수들은 필요한 대로 0개 이상 나열할 수 있는데 〈몸체〉에서 사용할 변수 이름을 지정하는 것으로 **형식파라미터**(formal parameter)라고 한다. 〈몸체〉는 1줄 이상의 명령문으로 구성된 블록으로 다른 블록과 마찬가지로 4칸 들여쓰기를 해야한다. 함수를 정의할 때 몸체는 실행하지 않는다.

함수를 호출하는 문법은 다음과 같다.

```
〈함수이름〉 ( 〈표현식〉1, 〈표현식〉2, … , 〈표현식〉n )
```

이미 정의되어 있는 함수만 호출이 가능하며, 괄호 안에 들어가는 표현식은 **실제파라미터**(actual parameter) 또는 **인수**(argument)라고 하며, 호출을 하면 인수(실제파라미터)를 계산한 결과 값을 각각 짝을 맞추어 형식파라미터의 변수에 지정을 하고 함수의 〈몸체〉를 실행한다.

함수에서 만들어 낸 정보를 결과로 내주려면 다음과 같은 형식으로 〈몸체〉의 끝부분에 `return` 문을 달아두어야 한다.

```
return 〈표현식〉
```

〈표현식〉을 계산한 결과를 함수 호출의 결과로 내준다. return 문이 없으면 함수 호출은 아무런 값도 내주지 않으며, return 문이 없는 함수를 **프로시저**(procedure)라고 한다.

연습문제

1. 반지름을 입력받아 원의 면적을 구하여 내주는 함수 area_circle을 정의해보자. 반지름이 r 이면 원의 면적은

$$\pi r^2$$

π 값은 Python의 math 모듈에 pi라는 이름으로 정해 놓았으므로 가져다 쓰면 된다. 모듈을 가져다 쓰려면 다음과 같이 쓰기전에 수입(import)해야 한다.

```
>>> import math
>>> math.pi
3.141592653589793
```

원의 면적을 구하는 Python 표현식은 `math.pi * r ** 2` 이다.

이제 다음의 빈 공간을 채워서 area_circle 함수를 작성해보자. 계산 결과에서 소수점 둘째자리 미만은 반올림하도록 하자.

```
def area_circle(r):
    import math
    return
```

작성한 함수가 잘 작동하는지 다음 호출로 시험해보자.

```
area_circle(5)
area_circle(9)
area_circle(23)
```

2. 이번엔 area_circle의 파라미터 k를 추가하여 반올림할 자리수를 인수로 전달할 수 있도록 프로그램을 수정해보자.

```
def area_circle(r,k):
    import math
    return
```

작성한 함수가 잘 작동하는지 다음 호출로 시험해보자.

```
area_circle(9,2)
area_circle(9,3)
area_circle(9,4)
```

실습 문제 1. 수강과목 점수 평균 구하기

표준입력창에서 점수를 하나씩 차례로 입력받아 평균을 계산하여 내주는 프로시저 score_average()를 만들어보자. score_average() 프로시저를 호출하면 표준입출력창을 통하여 사용자와 대화식으로 프로그램이 아래와 같이 작동한다. 보통 폰트로 된 부분은 컴퓨터가 사용자에게 보여주는 부분이고, 파란색으로 볼드체로 진하게 보이는 부분이 사용자가 키보드로 입력한 부분이다.

```
Enter your scores. I will calculate your average score.
score : -15
score again: 120
score again: 92
score : 만점
score again: 78
score : 54
score : 88
score : 64
score : 95
score : 0
Your average score of 6 subject(s) is 78.5
```

허용하는 키보드 입력은 0 과 100 사이의 정수이다. 위의 -15, 120, 만점은 이 요구사항을 만족하지 못하므로 재입력을 요구하였다. 실제 점수는 1과 100사이의 정수만 허용하고, 0은 점수 입력이 끝났다는 표시로만 사용한다. 위의 사례에서 92, 78, 54, 88, 64, 95 는 모두 유효한 입력이다. 0을 입력받으면 더 이상 입력을 받지 않고 그때까지 입력받은 점수의 평균을 계산하여, 몇 과목이며 평균은 몇점임을 표준창에 위와 같이 프린트한다. 평균 점수는 소수점 첫째 미만은 반올림한다.

만약 다음과 같이 유효한 점수가 하나도 없으면 평균을 계산하는 대신 점수가 없다는 메시지를 프린트한다.

```
Enter your scores. I will calculate your average score.
score : 110
score again: 120
score again: 200
score again: 0
There is no score.
```

사실 평균을 구하기 위해서 0으로 나누면 ZeroDivisionError 오류가 발생하므로 어짜피 별도로 처리해 주어야 한다. 즉, 0으로 나누는 대신 메시지를 프린트한 것이다.

알고리즘

평균을 구하려면 입력받은 점수를 모두 더한 다음, 과목의 개수로 나누면 된다.

1. Enter your scores. I will calculate your average score. 메시지를 프린트한다.
2. count 변수를 0으로 초기화한다. 이 변수는 수강과목 개수가 몇 개인지 기억하는 변수이다.
3. total 변수를 0으로 초기화한다. 이 변수는 수강과목 점수의 합을 기억하는 변수이다.
4. 수강과목 점수를 차례로 하나씩 입력받는다. (각 점수 입력확인 필수 – 0이상 100이하 정수면 통과)
5. 입력의 종료는 0을 입력하여 표시한다. 즉, 입력이 0이면 입력 받는 작업을 중지한다.

6. 입력이 0이 아니면 정수로 변환한 뒤 total 변수의 값에 합하여 누적한다.
7. count 변수의 값을 1 증가한다.
8. count 변수의 값이 0이 아닌 경우에만 평균을 계산하여 소수점 첫째리 미만은 반올림하고, Your average score of _ subject(s) is _ 형식으로 프린트 한다. 앞 부분은 과목의 수가 채워지고, 뒷 부분은 평균 점수가 채워진다.
9. count 변수의 값이 0인 경우에는 나눗셈을 하면 ZeroDivisionError 오류가 발생하므로 나누지 않고, There is no score. 라는 메시지를 프린트한다.

완성해야 할 함수

```
def score_average():
    print("Enter your scores. I will calculate your average score.")
    count = 0
    total = 0
    while True:
        
        
        
        
        
        
        
    if
        
        
        
```

실습 2. 수강과목 점수 평균 구하기 (과락은 빼고 평균 계산)

앞에서 완성한 프로시저 score_average()를 다음 추가 요구사항을 적용, 수정한 score_average2() 프로시저를 만들어보자.

추가 요구사항

- 입력한 과목점수가 60점 미만인 점수를 제외하고 평균을 구하도록 수정한다.
- 그리고 60점 미만이어서 과락한 과목이 몇 과목인지 알려준다.

score_average2() 프로시저를 호출하면 표준입출력창을 통하여 사용자와 대화식으로 프로그램이 아래와 같이 작동한다. 보통 폰트로 된 부분은 컴퓨터가 사용자에게 보여주는 부분이고, 파란색으로 볼드체로 진하게 보이는 부분이 사용자가 키보드로 입력한 부분이다.

```
Enter your scores. I will calculate your average score.
score : -15
score again: 120
score again: 92
score : 만점
score again: 78
score : 54
score : 88
score : 64
score : 95
score : 0
Your average score of 5 subject(s) is 83.4
You failed in 1 subject(s).
```

score_average() 프로시저와 동일하지만 1~59 점 사이의 키보드 입력은 과락이므로 평균계산에 포함하지 말아야 한다. 위의 사례에서는 54점이 유일하게 과락에 해당하는 점수이므로 평균 계산에 포함시키지 않았다. 과락한 점수는 몇 개인지 기억해 두었다가 몇 과목이 과락했다고 알려주어야 한다. 위에서는 1과목이므로 마지막 줄에서 그렇게 알려주었다.

평균을 구할 점수가 없을 때에도, 과락한 과목이 있으면 과락 과목의 개수를 알려주어야 한다.

```
Enter your scores. I will calculate your average score.
score : 43
score : 59
score : 0
There is no score.
You failed in 2 subject(s).
```

실습 3. 윤년인지 아닌지 판단해주는 함수 만들기

2월은 평년인지 윤년인지에 따라 각각 28일 또는 29일이 있다. 기본적으로 윤년은 4년마다 한번씩 돌아오는데 오차를 줄이기 위해서 예외를 둔다. 윤년을 결정하는 규칙은 "**4의 배수이면 윤년인데, 그 중에서 400의 배수를 제외한 100의 배수는 윤년이 아니다**"이다. 따라서 2008, 2012, 2016, 1600, 2000, 2400년은 모두 윤년이고, 2013, 2014, 2015, 2100, 2200, 2300은 모두 평년이다. 0 또는 양의 정수를 받아서 윤년이면 `True`, 평년이면 `False`를 내주는 함수 `isleapyear`를 작성하시오. 밑줄 친 부분에 논리식 하나를 만들어 채우면 된다. (음수 인수의 경우에는 0을 내주어야 한다.)

```
def isleapyear(year):



# test case
for y in range(5):
    print(y,isleapyear(y))
for y in range(2010,2017):
    print(y,isleapyear(y))
for y in range(1900, 2600, 100):
    print(y,isleapyear(y))
```