

05

함수

목차

1. 함수 기초
2. 함수 심화
3. 함수의 인수
4. 좋은 코드를 작성하는 방법

01. 함수 기초

■ 함수의 개념과 장점

- 함수(function) : 어떤 일을 수행하는 코드의 덩어리, 또는 코드의 묶음
- 함수의 장점
 - ① 필요할 때마다 호출 가능
 - ② 논리적인 단위로 분할 가능
 - ③ 코드의 캡슐화

01. 함수 기초

■ 함수의 선언

```
def 함수 이름 (매개변수 #1 ...):  
    수행문 1  
    수행문 2  
    return <반환값>
```

- ① **def** : 'definition'의 줄임말로, 함수를 정의하여 시작한다는 의미이다.
- ② **함수 이름** : 함수 이름은 개발자가 마음대로 지정할 수 있지만, 파이썬에서는 일반적으로 다음과 같은 규칙을 사용한다.
 - 소문자로 입력한다.
 - 띄어쓰기를 할 경우에는 _ 기호를 사용한다. ex) save_model
 - 행위를 기록하므로 동사와 명사를 함께 사용하는 경우가 많다. ex) find_number
 - 외부에 공개하는 함수일 경우, 줄임말을 사용하지 않고 짧고 명료한 이름을 정한다.

01. 함수 기초

■ 함수의 선언

```
def 함수 이름 (매개변수 #1 ...):  
    수행문 1  
    수행문 2  
    return <반환값>
```

- ③ **매개변수(parameter)** : 함수에서 입력값으로 사용하는 변수를 의미하며, 1개 이상의 값을 적을 수 있다.
- ④ **수행문** : 수행문은 반드시 들여쓰기한 후 코드를 입력해야 한다. 수행해야 하는 코드는 일반적으로 작성하는 코드와 같다. if나 for 같은 제어문을 사용할 수도 있고, 고급 프로그래밍을 하게 되면 함수 안에 함수를 사용하기도 한다.

01. 함수 기초

■ 함수의 실행 순서

코드 5-1 rectangle_area.py

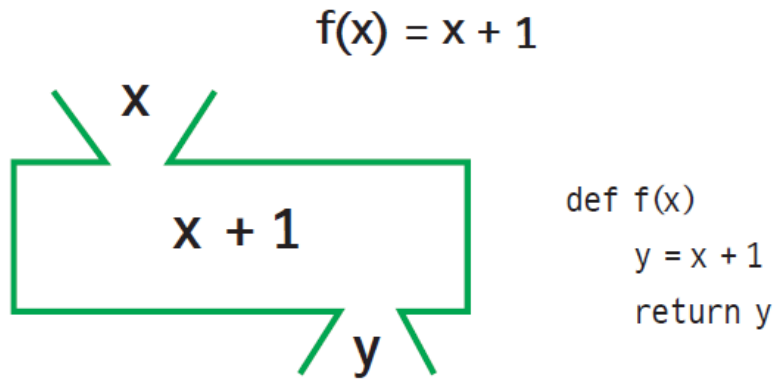
```
1 def calculate_rectangle_area(x, y):
2     return x * y
3
4 rectangle_x = 10
5 rectangle_y = 20
6 print("사각형 x의 길이:", rectangle_x)
7 print("사각형 y의 길이:", rectangle_y)
8
9 # 넓이를 구하는 함수 호출
10 print("사각형의 넓이:", calculate_rectangle_area(rectangle_x, rectangle_y))
```

```
사각형 x의 길이: 10
사각형 y의 길이: 20
사각형의 넓이: 200
```

01. 함수 기초

■ 프로그래밍의 함수와 수학의 함수

- 간단히 $f(x) = x + 1$ 을 코드로 나타낸다면, 다음과 같은 형태로 표현할 수 있다.



[수학에서의 함수 형태]

01. 함수 기초

■ 프로그래밍의 함수와 수학의 함수

- 실제로 다음과 같은 문제가 있다면 프로그래밍에서 코드의 함수로 어떻게 표현할 수 있을까?

$f(x) = 2x + 7$, $g(x) = x^2$ 이고 $x = 2$ 일 때
 $f(x) + g(x) + f(g(x)) + g(f(x))$ 의 값은?

$f(2) = 11$, $g(2) = 4$, $f(g(x)) = 15$, $g(f(x)) = 121$
 $\therefore 11 + 4 + 15 + 121 = 151$

```
1 def f(x):  
2     return 2 * x + 7  
3 def g(x):  
4     return x ** 2  
5 x = 2  
6 print(f(x) + g(x) + f(g(x)) + g(f(x)))
```


01. 함수 기초

■ 함수의 형태

매개변수 유무 반환값 유무	매개변수 없음	매개변수 있음
반환값 없음	함수 안 수행문만 수행	매개변수를 사용하여 수행문만 수행
반환값 있음	매개변수 없이 수행문을 수행한 후, 결과값 반환	매개변수를 사용하여 수행문을 수행한 후, 결과값 반환

```
1 def a_rectangle_area():          # 매개변수 × , 반환값 ×
2     print(5 * 7)
3 def b_rectangle_area(x, y):      # 매개변수 ○ , 반환값 ×
4     print(x * y)
5 def c_rectangle_area():          # 매개변수 × , 반환값 ○
6     return(5 * 7)
7 def d_rectangle_area(x , y):     # 매개변수 ○ , 반환값 ○
8     return(x * y)
9
10 a_rectangle_area()
11 b_rectangle_area(5, 7)
12 print(c_rectangle_area())
13 print(d_rectangle_area(5, 7))
```

02. 함수 심화

■ 함수의 호출 방식 :

종류	설명
값에 의한 호출 (call by value)	<ul style="list-style-type: none">• 함수에 인수를 넘길 때 값만 넘김• 함수 안의 인수값 변경 시, 호출된 변수에 영향을 주지 않음
참조 호출 (call by reference)	<ul style="list-style-type: none">• 함수에 인수를 넘길 때 메모리 주소를 넘김• 함수 안의 인수값 변경 시, 호출된 변수값도 변경됨

02. 함수 심화

■ 함수의 호출 방식(call by value)

```
1 def f(x):  
2     y = x  
3     x = 5  
4     return y * y  
5  
6 x = 3  
7 print(f(x))  
8 print(x)
```

02. 함수 심화

■ 함수의 호출 방식(call by object reference)

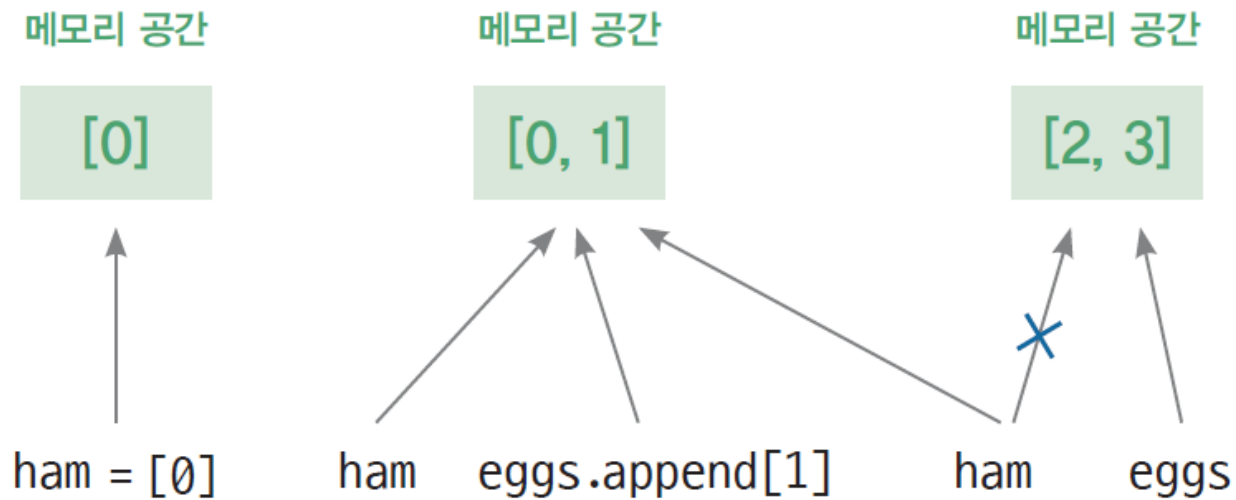
코드 5-6 call2.py

```
1 def spam(eggs):  
2     eggs.append(1)          # 기존 객체의 주소값에 [1] 추가  
  
3     eggs = [2, 3]          # 새로운 객체 생성  
4  
5 ham = [0]  
6 spam(ham)  
7 print(ham)
```

[0, 1]

02. 함수 심화

■ 함수의 호출 방식



[객체 호출 방식]

02. 함수 심화

■ 변수의 사용 범위

- 변수의 사용 범위(scoping rule) : 변수가 코드에서 사용되는 범위
- 지역 변수(local variable) : 함수 안에서만 사용
- 전역 변수(global variable) : 프로그램 전체에서 사용

02. 함수 심화

■ 변수의 사용 범위

```
1 def test(t):  
2     print(x)  
3     t = 20  
4     print("In Function:", t)  
5  
6 x = 10  
7 test(x)  
8 print("In Main:", x)  
9 print("In Main:", t)
```

```
-  
  
10  
In function: 20  
In Main: 10  
Traceback (most recent call last):  
  File "scoping_rule.py", line 9, in <module>  
    print("In Main:", t)  
NameError: name 't' is not defined
```

02. 함수 심화

■ 변수의 사용 범위

```
1 def f():  
2     s = "I love London!"  
3     print(s)  
4  
5 s = "I love Paris!"  
6 f()  
7 print(s)
```



```
I love London!  
I love Paris!
```


02. 함수 심화

■ 변수의 사용 범위

- 함수내에서 global 변수 사용

```
1 def f():  
2     global s  
3     s = "I love London!"  
4     print(s)  
5  
6 s = "I love Paris!"  
7 f()  
8 print(s)
```

```
I love London!  
I love London!
```

02. 함수 심화

■ 변수의 사용 범위

```
1 def calculate(x, y):
2     total = x + y          # 새로운 값이 할당되어 함수 안 total은 지역 변수가 됨
3     print("In Function")
4     print("a:", str(a), "b:", str(b), "a + b:", str(a + b), "total:", str(total))
5     return total
6
7 a = 5                      # a와 b는 전역 변수
8 b = 7
9 total = 0                  # 전역 변수 total
10 print("In Program - 1")
11 print("a:", str(a), "b:", str(b), "a + b:", str(a + b))
12
13 sum = calculate(a, b)
14 print("After Calculation")
15 print("Total:", str(total), " Sum:", str(sum)) # 지역 변수는 전역 변수에 영향을 주지
                                                # 않음
```

```
In Program - 1
a: 5 b: 7 a + b: 12
In Function
a: 5 b: 7 a + b: 12 total: 12
After Calculation
Total : 0 Sum: 12
```

02. 함수 심화

■ 재귀 함수

- 재귀 함수(recursive function) : 함수가 자기 자신을 다시 부르는 함수이다.

$$\begin{aligned} 1! &= 1 \\ 2! &= 2(1) = 2 \\ 3! &= 3(2)(1) = 6 \\ 4! &= 4(3)(2)(1) = 24 \\ 5! &= 5(4)(3)(2)(1) = 120 \end{aligned}$$
$$n! = n \cdot (n-1) \cdots 2 \cdot 1 = \prod_{i=1}^n i$$

[점화식]

- 위 수식이 팩토리얼(factorial) 함수이다. 정확히는 'n!'로 표시하면 $n! = n \times (n-1)!$ 로 선언할 수 있다. 자신의 숫자에서 1씩 빼면서 곱하는 형식이다. 보통은 점화식이라고 한다.

02. 함수 심화

■ 재귀 함수

- factorial() 함수는 n의 변수를 입력 매개변수로 받은 후 n == 1이 아닐 때까지 입력된 n과 n에서 1을 뺀 값을 입력값으로 하여 자신의 함수인 factorial()로 다시 호출한다.

```
1 def factorial(n):
2     if n == 1:
3         return 1
4     else:
5         return n * factorial(n - 1)
6
7 print(factorial(int(input("Input Number for Factorial Calculation: "))))
```

```
5 * factorial(5 - 1)
= 5 * 4 * factorial(4 - 1)
= 5 * 4 * 3 * factorial(3 - 1)
= 5 * 4 * 3 * 2 * factorial(2 - 1)
= 5 * 4 * 3 * 2 * 1
```

```
Input Number for Factorial Calculation: 5
120
```

← 사용자 입력
← 화면 출력

03. 함수의 인수

■ 함수의 인수 사용법

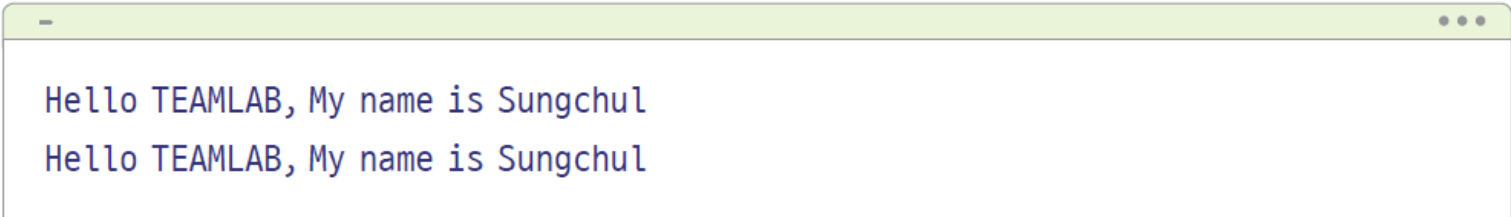
종류	내용
키워드 인수	함수의 인터페이스에 지정된 변수명을 사용하여 함수의 인수를 지정하는 방법
디폴트 인수	별도의 인수값이 입력되지 않을 때, 인터페이스 선언에서 지정한 초깃값을 사용하는 방법
가변 인수	함수의 인터페이스에 지정된 변수 이외의 추가 변수를 함수에 입력할 수 있게 지원하는 방법
키워드 가변 인수	매개변수의 이름을 따로 지정하지 않고 입력하는 방법

03. 함수의 인수

■ 키워드 인수

- 키워드 인수(keyword arguments) : 함수에 입력되는 매개변수의 변수명을 사용하여 함수의 인수를 지정하는 방법

```
1 def print_something(my_name, your_name):  
2     print("Hello {0}, My name is {1}".format(your_name, my_name))  
3  
4 print_something("Sungchul", "TEAMLAB")  
5 print_something(your_name = "TEAMLAB", my_name = "Sungchul")
```



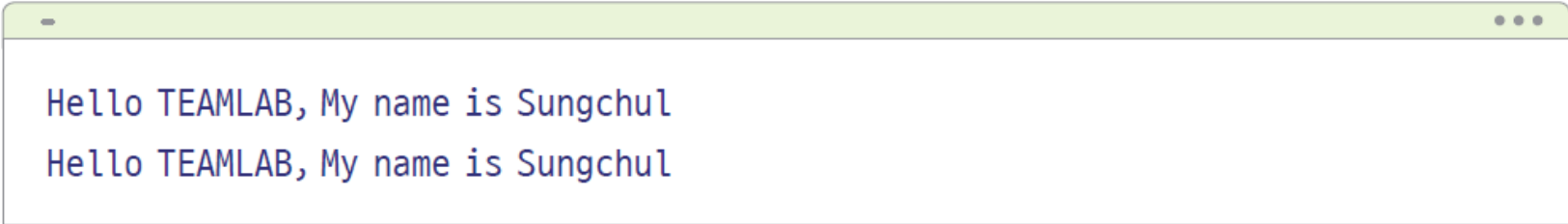
```
Hello TEAMLAB, My name is Sungchul  
Hello TEAMLAB, My name is Sungchul
```

03. 함수의 인수

■ 디폴트 인수

- 디폴트 인수(**default arguments**) : 매개변수에 기본값을 지정하여 사용하고, 아무런 값도 인수로 넘기지 않으면 지정된 기본값을 사용하는 방식

```
1 def print_something_2(my_name, your_name = "TEAMLAB"):  
2     print("Hello {0}, My name is {1}".format(your_name, my_name))  
3  
4 print_something_2("Sungchul", "TEAMLAB")  
5 print_something_2("Sungchul")
```



```
Hello TEAMLAB, My name is Sungchul  
Hello TEAMLAB, My name is Sungchul
```

03. 함수의 인수

■ 가변 인수

- 함수의 매개변수 개수가 정해지지 않고 사용하는 것이 가변 인수(variable-length arguments)
- 가변 인수는 *(asterisk라고 부름)로 표현할 수 있는데, *는 파이썬에서 기본적으로 곱셈 또는 제곱 연산 외에도 변수를 묶어 주는 가변 인수를 만든다

```
1 def asterisk_test(a, b, *args):  
2     return a + b + sum(args)  
3  
4 print(asterisk_test(1, 2, 3, 4, 5))
```

15

03. 함수의 인수

■ 가변 인수

- 앞의 코드를 다음과 같이 변경한 후 실행하면, 다음과 같은 결과를 얻을 수 있다.

```
1 def asterisk_test(a, b, *args):  
2     print(args)  
3  
4 print(asterisk_test(1, 2, 3, 4, 5))
```

```
(3, 4, 5)  
NONE
```

- args의 결과값이 괄호로 묶여 출력되는 것을 확인할 수 있다. 이렇게 괄호로 묶여 출력되는 자료형을 튜플(tuple)자료형이라고 한다.
- 가변인수 *는 반드시 일반적인 키워드 인수가 모두 끝난 후 넣어야 한다. 리스트와 비슷한 튜플 형태로 함수 안에서 사용할 수 있으므로 인덱스를 사용하여, 즉 args[0], args[1] 등으로 변수에 접근할 수 있다.

03. 함수의 인수

■ 가변 인수

- 또 다른 사용법으로 [코드 5-16]과 같이 변환할 수 있다.

코드 5-16 asterisk3.py

```
1 def asterisk_test_2(*args):  
2     x, y, *z = args  
3     return x, y, z  
4  
5 print(asterisk_test_2(3, 4, 5))
```



(3, 4, [5])

- 입력받은 가변 인수의 개수를 정확히 안다면, `x, y, z = args`처럼 언패킹을 사용할 수 있다.
- 만약 `*z`가 아닌 상태에서 `asterisk_test_2(3, 4, 5, 10, 20)`으로 변경하여 코드를 실행하면 오류가 발생한다. 왜냐하면 언패킹의 개수가 맞지 않기 때문이다.

03. 함수의 인수

■ 가변 인수

- 언패킹 코드를 `x, y, *z = args`로 변경하면 어떤 결과가 나올까?

코드 5-17 asterisk4.py

```
1 def asterisk_test_2(*args):  
2     x, y, *z = args  
3     return x, y, z  
4  
5 print(asterisk_test_2(3, 4, 5, 10, 20))
```

```
(3, 4, [5, 10, 20])
```

03. 함수의 인수

■ 키워드 가변 인수

- 키워드 가변 인수(keyword variable-length arguments)는 매개변수의 이름을 따로 지정하지 않고 입력하는 방법으로, 이전 가변 인수와는 달리 *를 2개 사용하여 함수의 매개변수를 표시한다.
- 입력된 값은 튜플 자료형이 아닌 딕셔너리 자료형(dictionary type)으로 사용할 수 있다.
- 키워드 가변 인수는 반드시 모든 매개변수의 맨 마지막, 즉 가변 인수 다음에 선언되어야 한다.

03. 함수의 인수

■ 키워드 가변 인수

코드 5-18 kwargs.py

```
1 def kwargs_test(**kwargs):  
2     print(kwargs)  
3     print("First value is {first}".format(**kwargs))  
4     print("Second value is {second}".format(**kwargs))  
5     print("Third value is {third}".format(**kwargs))  
6  
7 kwargs_test(first = 3, second = 4, third = 5)
```

```
{'first': 3, 'second': 4, 'third': 5}  
First value is 3  
Second value is 4  
Third value is 5
```

03. 함수의 인수

■ 키워드 가변 인수

- 다음 코드에서 3, 4는 각각 one, two에 할당되고, 나머지 5, 6, 7, 8, 9는 args에, first = 3, second = 4, third = 5는 딕셔너리형으로 kwargs에 저장된다.

```
>>> def kwargs_test(one, two, *args, **kwargs):  
...     print(one + two + sum(args))  
...     print(kwargs)  
...  
>>> kwargs_test(3, 4, 5, 6, 7, 8, 9, first = 3, second = 4, third = 5)  
42  
{'first': 3, 'second': 4, 'third': 5}
```

04. 좋은 코드를 작성하는 방법

■ 좋은 코드의 의미

- 프로그래밍은 팀플레이(team play)이다. 좋은 프로그래밍을 하는 규칙이 있어야 한다.



[페이스북 사무실]

04. 좋은 코드를 작성하는 방법

■ 좋은 코드의 의미

- 가독성 좋은 코드를 작성하기 위해서는 여러 가지가 필요하지만, 일단 여러 사람의 이해를 돕기 위한 규칙이 필요하다. 프로그래밍에서는 이 규칙을 일반적으로 코딩 규칙(coding convention)이라고 한다.

"컴퓨터가 이해할 수 있는 코드는 어느 바보나 다 짤 수 있다. 좋은 프로그래머는 사람이 이해할 수 있는 코드를 짤다."
- 마틴 파울러

04. 좋은 코드를 작성하는 방법

■ 코딩 규칙

- 들여쓰기는 4 스페이스
- 한 줄은 최대 79자까지
- 불필요한 공백은 피함
- 파이썬에서는 이러한 규칙 중 파이썬 개발자가 직접 정한 것이 있다. 이를 (PEP 8Python Enhance Proposal 8)이라고 하는데, 이는 파이썬 개발자들이 앞으로 필요한 파이썬의 기능이나 여러 가지 부수적인 것을 정의한 문서이다.

04. 좋은 코드를 작성하는 방법

■ PEP 8의 코딩 규칙

- = 연산자는 1칸 이상 띄우지 않는다

```
variable_example = 12      # 필요 이상으로 빈칸이 많음  
variable_example = 12      # 정상적인 띄어쓰기
```

- 주석은 항상 갱신하고, 불필요한 주석은 삭제한다.
- 소문자 l, 대문자 O, 대문자 I는 사용을 금한다.

```
lI00 = "Hard to Understand"    # 변수를 구분하기 어려움
```

- 함수명은 소문자로 구성하고, 필요하면 밑줄로 나눈다.

04. 좋은 코드를 작성하는 방법

여기서 잠깐! flake8 모듈

- 코딩을 한 후, 코딩 규칙을 제대로 지켰는지 확인하는 방법 중 하나는 flake8 모듈로 체크하는 것이다. Flake8을 설치하기 위해서는 먼저 cmd 창에 다음과 같이 입력한다.

```
conda install -c anaconda flake8
```

- Atom에 [코드 5-19]와 같이 코드를 작성한 후, 'test_flake.py'로 저장한다.

코드 5-19 test_flake.py

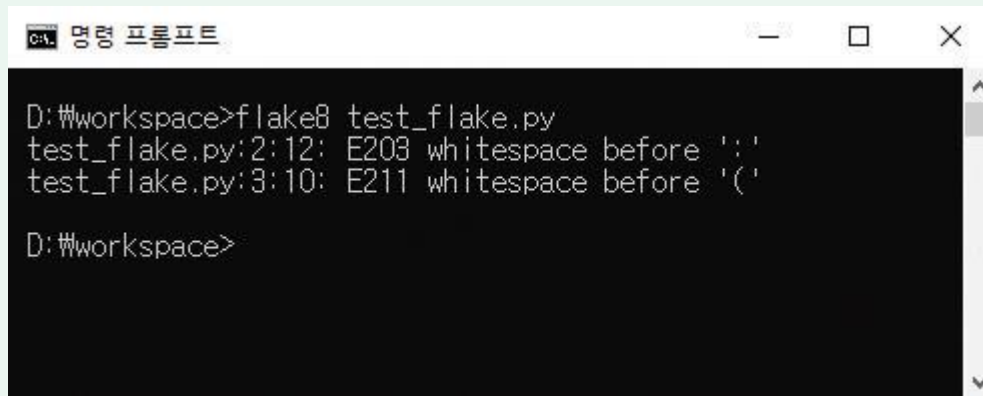
```
1 lL00 = "123"  
2 for i in 10 :  
3     print("Hello")
```

04. 좋은 코드를 작성하는 방법

여기서 잠깐! flake8 모듈

- 그리고 cmd 창에 다음과 같이 입력하면, 각 코드의 수정 방법을 알려 준다.

```
flake8 test_flake.py
```



```
C:\> 명령 프롬프트

D:\workspace>flake8 test_flake.py
test_flake.py:2:12: E203 whitespace before ':'
test_flake.py:3:10: E211 whitespace before '('

D:\workspace>
```

[flake8 모듈로 코드 수정 방법 확인]

04. 좋은 코드를 작성하는 방법

■ 함수 개발 가이드라인 : 함수 이름

- 함수는 가능하면 짧게 작성할 것(줄 수를 줄일 것)
- 함수 이름에 함수의 역할과 의도를 명확히 드러낼 것

```
def print_hello_world():  
    print("Hello, World")  
def get_hello_world():  
    return "Hello, World"
```

04. 좋은 코드를 작성하는 방법

■ 함수 개발 가이드라인 : 함수의 역할

- 하나의 함수에는 유사한 역할을 하는 코드만 포함해야 한다. 함수는 한 가지 역할을 명확히 해야 한다. 다음 코드처럼 두 변수를 더하는 함수라면 굳이 그 결과를 화면에 출력할 필요는 없다. 이름에 맞는 최소한의 역할을 할 수 있도록 작성해야 한다.

```
def add_variables(x, y):  
    return x + y
```

```
def add_variables(x, y):  
    print(x, y)  
    return x + y
```

04. 좋은 코드를 작성하는 방법

■ 함수 개발 가이드라인 : 함수를 만드는 경우

- 공통으로 사용되는 코드를 함수로 변환

코드 5-20 hello1.py

```
1 a = 5
2 if (a > 3):
3     print("Hello World")
4     print("Hello TEAMLAB")
5 if (a > 4):
6     print("Hello World")
7     print("Hello TEAMLAB")
8 if (a > 5):
9     print("Hello World")
10    print("Hello TEAMLAB")
```

```
-
Hello World
Hello TEAMLAB
Hello World
Hello TEAMLAB
```

04. 좋은 코드를 작성하는 방법

■ 함수 개발 가이드라인 : 함수를 만드는 경우

- 공통으로 사용되는 코드를 함수로 변환

코드 5-21 hello2.py

```
1 def print_hello():
2     print("Hello World")
3     print("Hello TEAMLAB")
4
5 a = 5
6 if (a > 3):
7     print_hello()
8
9 if (a > 4):
10    print_hello()
11
12 if (a > 5):
13    print_hello()
```

```
Hello World
Hello TEAMLAB
Hello World
Hello TEAMLAB
```


04. 좋은 코드를 작성하는 방법

■ 함수 개발 가이드라인 : 함수를 만드는 경우

- 복잡한 로직이 사용되었을 때, 식별 가능한 이름의 함수로 변환

코드 5-22 math1.py

```
1 import math
2 a = 1; b = -2; c = 1
3
4 print((-b + math.sqrt(b ** 2 - (4 * a * c))) / (2 * a))
5 print((-b - math.sqrt(b ** 2 - (4 * a * c))) / (2 * a))
```

1.0

1.0

04. 좋은 코드를 작성하는 방법

■ 함수 개발 가이드라인 : 함수를 만드는 경우

- 복잡한 로직이 사용되었을 때, 식별 가능한 이름의 함수로 변환

코드 5-23 math2.py

```
1 import math
2
3 def get_result_quadratic_equation(a, b, c):
4     values = []
5     values.append((-b + math.sqrt(b ** 2 - (4 * a * c))) / (2 * a))
6     values.append((-b - math.sqrt(b ** 2 - (4 * a * c))) / (2 * a))
7     return values
8
9 print(get_result_quadratic_equation(1,-2,1))
```

[1.0, 1.0]