

데이터과학을 위한

# 파이썬 프로그래밍



## 05. 함수

# 목차

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

01

함수 기초

# 01. 함수 기초

여러 명이 프로그램을 함께 개발할 때, 코드를 어떻게 작성하면 좋을까?

- ① 다 같이 모여 토론하며 한 줄 한 줄 작성하기
- ② 가장 잘하는 사람이 혼자 작성하기
- ③ 필요한 부분을 나누어 작성한 후 합치기 → 가장 일반적이고 많이 사용하는 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>
```

### 참고 : 파이썬의 이름 명명 규칙

- 모듈의 이름은 `소문자`
- 클래스 이름은 `카멜 케이스` (camelCase)
- 함수의 이름은 `스네이크 케이스` (snake\_case)

- ① **def** : 'definition'의 줄임 말로서 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, 줄임 말을 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) 혹은 function arguments** : 함수에서 입력 값으로 사용하는 변수를 의미하며, 1개 이상의 값을 적을 수 있다.
- ④ **수행문** : 수행문은 반드시 **들여쓰기** 한 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 선언

- 함수 선언 작성 예시를 간단한 코드로 살펴보자

```
def calculate_rectangle_area(x, y)
    return x * y
```

- ① 먼저 선언된 함수를 확인할 수 있다.
- ② 함수 이름은 `calculate_rectangle_area`이고, `x`와 `y`라는 2개의 매개변수를 사용하고 있다.
- ③ `return`의 의미는 값을 반환한다는 뜻으로, `x`와 `y`를 곱한 값을 반환하는 함수로 이해한다.



# 01. 함수 기초

## 여기서 잠깐! 반환

- 약간 어렵게 느껴질 수 있는 부분이 바로 '반환'이라는 개념이다. 이는 수학에서의 함수와 같은 개념이라고 생각하면 된다. 예를 들어, 수학에서  $f(x) = x + 1$ 이라고 한다면  $f(1)$ 의 값은 얼마일까? 중학교 정도의 수학을 이해하고 있다면  $f(1) = 2$ 라는 것을 알 것이다. 즉, 함수  $f(x)$ 에서  $x$ 에 1이 들어가면 2가 반환 되는 것이다. 파이썬의 함수도 같은 개념이다. 수학에서  $x$ 에 해당하는 것이 매개변수, 즉 입력 값이고,  $x + 1$ 의 계산 과정이 함수 안의 코드이며, 그 결과가 출력 값이다.

# 01. 함수 기초

예제: rectangle\_area.py

## ■ 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))
```

\* 참고

함수는 여러번 중복 선언가능하다.  
맨 나중에 선언한 함수가 유효하다.  
예제 소스 프로그램 참조...

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

# 01. 함수 기초

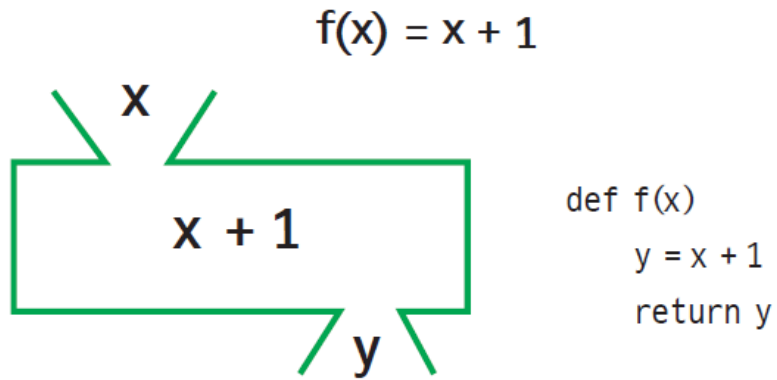
## ■ 함수의 실행 순서: [코드 5-1] 해석

- 1행의 함수가 정의된 def 부분은 실행하지 않는다. 실행되지 않는 것처럼 보일 뿐, 해당 코드를 메모리에 업로드하여 다른 코드를 호출해 사용할 수 있도록 준비 과정을 거친다. 만약 함수의 선언 부분을 코드의 맨 끝에 둔다면 해당 코드 호출에 오류가 발생할 것이다.
- 다음으로 함수 다음의 코드를 실행한다. 정확히는 `rectangle_x = 10`과 `rectangle_y = 20`, 2개의 변수에 값이 할당되고 그 값이 출력된다. 다음 코드인 `print("사각형의 넓이:", calculate_rectangle_area(rectangle_x, rectangle_y))`를 호출한다. 해당 함수를 호출하고, `rectangle_x`와 `rectangle_y` 변수에 할당된 값이 `calculate_rectangle_area`에 입력된다. 그러면 함수 코드 `return x * y`에 의해 반환값 200이 반환된다.

# 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?  
→ [코드 5-2] 확인

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

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

## ■ 함수의 실행 순서

코드 5-2 function.py

```
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)))
```

151

- ➡ 6행의 print( ) 함수의 코드인 f(x), g(x), f(g(x)), g(f(x))가 각각 11, 4, 15, 121로 치환되어 결과가 나오는 것을 확인할 수 있다.

## 여기서 잠깐! 매개변수와 인수

- 매개변수는 함수의 인터페이스 정의에 있어 어떤 변수를 사용하는지를 정의하는 것이다. 그에 반해 인수는 실제 매개변수에 대입되는 값을 뜻한다.

### 코드 5-3 parameter.py

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



11

- ➡ [코드 5-3]에서 'def f(x):'의 x를 매개변수라고 한다. 일반적으로 함수의 입력값에 대한 정의를 함수 사용에 있어 인터페이스를 정의한다고 한다. 매개변수는 함수의 인터페이스 정의에 있어 어떤 변수를 사용하는지를 정의하는 것이다. 즉, 위 함수에서는 x가 해당 함수의 매개변수이다. 그에 반해, 인수는 실제 매개변수에 대입되는 값을 뜻한다. 매개변수가 설계도라면 인수는 그 설계도로 지은 건물 같은 것이다. 위 코드에서는 f(2)에서 2가 인수에 해당한다.

# 01. 함수 기초

## ■ 함수의 형태

매개변수 유무 반환값 유무	매개변수 없음		매개변수 있음	
	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	

[함수의 형태]



## ■ 함수의 형태

코드 5-4 function\_type.py

```
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))
```

```
-
35
35
35
35
```

# 01. 함수 기초

## ■ 함수의 형태 : [코드 5-4] 해석

- 첫 번째 함수는 매개변수와 반환값이 모두 없는 경우이다. 입력값도 없고 반환되는 변수도 없지만, `print(5 * 7)`로 인해 35가 출력된다. 이 경우에는 `a_rectangle_area()`가 35로 치환되는 것이 아니고, 반환값이 없기 때문에 함수 자체는 `none` 값을 가진다. 대신 함수 안에 있는 `print()` 함수로 인해 35만 출력하는 것이다.
- 두 번째 함수는 `b_rectangle_area()`가 매개변수로 `x, y`를 받고, 그 값을 계산하여 화면에 출력하는 함수이다. 역시 반환값이 없으므로 11행에서 `b_rectangleArea(5, 7)`을 실행하면 35가 출력되지만, `b_rectangleArea(5, 7)` 자체가 35로 치환되지는 않는다. 반환이 없으면 해당 함수는 `none`으로 치환된다.
- 세 번째, 네 번째 함수는 반환값이 있는 경우이다. `c_rectangle_area()`와 `d_rectangle_area()` 함수 모두 함수 안에 `print()` 함수가 있는 것이 아니라, 함수를 호출한 곳에 `print()` 함수가 있는 것을 확인할 수 있다. 이는 두 함수 모두 `return` 구문으로 인해 35로 치환되기 때문이다.  
이렇게 `return`이 있는 경우, 즉 함수의 출력값이 있는 경우에는 그 함수를 호출한 곳에서 함수의 반환값을 변수에 할당하여 사용할 수 있다.

02

함수 심화

## 02. 함수 심화

예제: call1.py  
실습 0

### ■ 함수의 호출 방식 - 지역 변수

코드 5-5 call1.py

*전역변수 x가 전역 변수의 지위를 잃어 버리고 지역변수로 바뀐 사례*

```
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)
```

x는 파라미터 변수. 메인으로 부터 넘어온다.

x는 지역 변수로 새롭게 정의된다. 만약 x가 list 자료라면 원소를 바꾸는 수준의 변경이라면 전역변수 자격을 잃지 않는다. => call2.py에서 검토

메인에서 선언된 변수 x는 원칙적으로 전역변수이다. 함수에서 바꾸지만 않으면....

9

3

## 02. 함수 심화

### ■ 함수의 호출 방식 : [코드 5-5] 해석

- 함수 밖에 있는 변수 x의 메모리 주소와 함수 안에 있는 변수 x의 메모리 주소가 같은지 다른지 확인할 필요가 있다.
- 함수 안에 변수가 인수로 들어가 사용될 때, 변수를 호출하는 방식을 전통적인 프로그래밍에서는 다음과 같이 크게 두 가지로 나눈다.

[함수가 변수를 호출하는 방식]

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

# 참고(1): 전역 변수가 성공한 사례

예제: call1.py  
실습 1

```
def f( ):
```

```
    y = x    # 전역변수라서 참조(읽기 동작) 가능하다. 수정하지 않으면 전역변수 자격 유지..
```

```
# x = 5 # 함수 선언단계에서 허락되지 않음
```

```
    return y * y
```

```
x = 3    # 메인 루틴에서 선언한 변수는 전역변수
```

```
print(f())
```


```
print(x)
```

 9  
출력결과 3

- 원천적으로 메인 루틴에서 선언한 변수 x는 전역변수이다. 함수 내에서 그 값을 읽어 사용할 수 있다.
- 본 사례의 경우 함수 내에서 변수 x에 값을 assign 하는 동작은 함수 선언 단계에서 부터 허락되지 않는다. => 메인 루틴에서만 값을 assign하면서 새로운 변수를 선언할 수 있다.

## 참고(2): 전역 변수가 성공한 사례

예제: call1.py  
실습 2

```
def f():  
    y = x.copy()    # y는 다른 메모리에 공간에 데이터를 복사하여 새로운 메모리 공간을 확보한 변수이다.  
    z = x           # z는 x와 같은 데이터 공간을 공유한 변수이다.  
    x[0] = 5        # 값을 바꾸었지만 mutable data sequence 자료형은 계속 전역변수 지위를 유지한다.  
    z[1] = 'q'      # x와 데이터 영역을 공유하기 때문에 x도 같이 바뀐다.  
    return x, y, z  # 3개의 파라미터를 tuple로 묶어서 반환한다.  
x = [1, 2, 3]      # 원천적으로 메인 루틴에서 선언한 변수는 전역변수이다. 함수 내에서도 그 값을 읽어 사용할 수 있다.  
print(x)            [1, 2, 3]  
print(f())         ([5, 'q', 3], [1, 2, 3], [5, 'q', 3])
```

출력결과

- 함수 내에서 전역변수에 write가 가능한 경우가 있다.
  - 리스트 자료는 함수 내에서 값을 assign 하는 동작도 허용된다.
- mutable(비정적) & data sequence(나열형) 자료형은 함수 내에서 수정을 해도 계속 전역변수 지위를 유지한다.
  - mutable data sequence 자료형 - 연속적인 데이터 나열형 자료. 내부 값을 변경할 수 있다.
  - list 자료는 대표적인 mutable data sequence 자료형이다.

## 02. 함수 심화

### ■ 함수의 호출 방식

[코드 5-6] call2.py

```
1      def      spam(eggs):      # 주의!! 반환 값이 없음.
2          eggs.append(1)      # 기존 객체의 주소 값에 [1] 추가
3          eggs = [2, 3]      # 새로운 객체 생성. 함수 안의 local 변수
4
5      ham = [0]      # 1개 원소의 리스트 선언
6      spam(ham)      # 함수 호출
7      print(ham)
8
9      #print(eggs) # NameError: name 'eggs' is not defined
10     # eggs는 함수 안에서만 쓰이는 local 변수이다.
```

[0, 1] # 출력 결과

\* 본래 리스트와 같은 *mutable data sequence* 자료형은 함수 내에서 수정을 해도 계속 전역변수 지위를 유지한다. 다만, 위 사례의 3번 줄에서 *eggs*는 새로운 *local* 변수 객체를 생성한 경우가 되기 때문에 메인 루틴에서 액세스가 불가능하다.



## 02. 함수 심화

### ■ 함수의 호출 방식 : [코드 5-6] 해석

- [코드 5-6]에서 먼저 ham이라는 리스트를 만들고, 함수 spam에 ham을 인수로 넣었다. 이때 함수 안에서는 이름이 ham에서 eggs로 바뀐다. ham과 eggs는 함수의 호출 방식 객체 호출이므로 같은 주소를 공유한다. 따라서 2행의 eggs.append(1)에 의해 해당 리스트에 1이 추가되면, ham과 eggs 모두의 영향을 받는다.
- 3행의 eggs = [2, 3]은 새로운 리스트를 만드는 코드이다. 그래서 이 경우, 더는 ham과 eggs와 같은 메모리 주소를 가리키지 않고 eggs는 자기만의 메모리 주소를 가지게 된다. 그리고 함수를 빠져나가 7행의 print(ham)이 실행되면 2행의 eggs.append(1)에 의해 [0, 1]이 화면에 출력된다. 이것이 바로 객체 호출(call by object reference)이라는 파이썬의 함수 안 변수 호출 방식이다.
- 새로운 값을 할당하기 전까지는 기존에 넘어온 인수 객체의 주소값을 쓰는 방식이라고 이해하면 된다. 이 내용을 알아야 하는 가장 큰 이유는 다른 사람의 코드를 이해하기 위함이다.

## 02. 함수 심화

- 정리: mutable type data sequence는 함수 안에서 수정 가능하고, 전역적 특성을 가진다.

1. 파이썬의 function arguments는 call by reference 방식으로 전달된다.
2. 즉, 값을 담고 있는 객체를 함수에게 전달하는 것이 아니라, 그 객체의 주소만을 전달한다.
3. 이 때문에 그 객체가 list와 같이 data sequence 이면서 mutable type이면 함수 안에서 그 객체를 변경함으로써 function arguments의 값을 바꿀 수도 있다.
4. mutable type 객체가 함수 안에서 변경 가능한 것은 원소 값을 바꾸는 경우에 한정된다. 객체를 새로 선언하는 수준의 변경은 함수 안에서 새로 정의하는 것과 같아서 전역변수가 될 수 없다.
5. 정수나 스트링 같은 immutable type의 객체는 함수 안에서 값을 수정할 수 없다.

## 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

예제: `scoping_rule.py`

### ■ 변수의 사용 범위

코드 5-7 `scoping_rule.py`

```
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) ←
```

메인에서 선언된 변수 `x`는 원천적으로 전역변수이다. 함수 안에서 assignment로 바꾸지 않고 읽기 동작만 하면 전역변수 지우는 계속 유지된다.

변수 `t`는 함수 안에서만 유효하다. 즉, 지역 변수이다.

```
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 : [코드 5-7] 해석

- 관심을 두어야 할 변수는  $x$ 와  $t$ 이다. 프로그램이 가장 먼저 시작되는 지점은 6행의  $x = 10$ 이다. 그리고 7행에서  $x$ 는 `test(x)` 함수로 변수를 넘기게 된다. 그렇다면 함수 안에서 처음 만나는 2행 `print(x)`의  $x$ 는 어떤 변수일까? 이때의  $x$ 는 함수 안에서 재정의되지 않았으므로 함수를 호출한 메인 프로그램의  $x = 10$ 의  $x$ 를 뜻한다. 즉, 프로그램 전체에서 사용할 수 있는 전역 변수이다.
- 함수 안의  $t$ 는 `test(x)` 함수의  $x$ 를  $t$ 로 치환하여 사용한다. 즉, 함수 안에서는  $x$ 를 따로 선언한 적은 없고,  $t$ 를 선언하여 사용하는 것이다. 그리고 3행의  $t = 20$ 에 의해  $t$ 에 20이 할당되고, 실제로 4행 `print("In Function:", t)`문의 결과에 의해 In Function: 20이 화면에 출력되는 것으로 예상할 것이다. 하지만 함수가 종료되고 코드에 9행의 `print("InMain:", t)`가 실행되면 오류가 출력된다. 왜냐하면  $t$ 가 함수 안에서만 사용할 수 있는 지역변수이기 때문이다.

## 02. 함수 심화

예제: local\_variable.py

### ■ 변수의 사용 범위

코드 5-8 local\_variable.py

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

s의 요소를 바꾸는 것이 아니라 새로 변수를 선언하는 것과 같다. 따라서 지역변수를 선언하는 것이다.

만약 2번 줄을 아래와 같이 시도한다면  
s[0] = 'y' # 오류 발생. s는 스트링이라 immutable data sequence라서 시도조차 할 수가 없다.  
만약 mutable data sequence라면, 전역변수를 유지할 수 있다.

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

## 02. 함수 심화

### ■ 변수의 사용 범위 : [코드 5-8] 해석

- [코드 5-8]에서 변수 `s`는 함수 `f()`의 안에서도 사용되고 밖에서도 사용된다. `s`의 값은 어떻게 바뀔까? 프로그램이 시작되자마자 `s`에는 'I love Paris!'라는 값이 할당된다. 하지만 그 후, 함수 안으로 코드의 실행이 옮겨가 다시 `s`에 'I love London!' 값이 저장되고, 그 값이 먼저 출력된다.
- 그렇다면 함수 밖 변수 `s`의 값은 변경되었을까? 함수가 종료된 후 7행 `print(s)`의 실행 결과는 'I love Paris!'이다. 왜 이런 일이 발생했을까? 함수 안과 밖의 `s`는 같은 이름을 가졌지만, 사실 다른 메모리 주소를 가진 전혀 다른 변수이기 때문이다. 따라서 함수 안의 `s`는 해당 함수가 실행되는 동안에만 메모리에 있다가 함수가 종료되는 순간 사라진다. 당연히 함수 밖 `s`와는 메모리 주소가 달라 서로 영향을 주지 않는다. 변수의 이름이 같다고 다 같은 함수가 아니다.

## 02. 함수 심화

예제: global\_variable.py

### ■ 변수의 사용 범위

- 그렇다면 함수 안의 변수와 함수 밖의 함수가 같은 이름을 사용하기 위해서는 어떻게 해야 할까? **함수 내에서 전역 변수를 선언하기 위해서는 global 키워드를 사용한다.**
- *함수 밖에서 global 선언을 하는 것이 아니다. 메인에서 선언된 변수는 처음부터 global이다.*

코드 5-9 global\_variable.py

```
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)
```

함수 내에서 global 변수를 선언  
하는 방법

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



## 02. 함수 심화

### ■ 변수의 사용 범위 : [코드 5-9] 해석

- 기존 코드에서 변경된 것은 2행의 함수 내 global s 코드 하나이다. 그러나 결과는 이전과 다르게 출력되는 것을 확인할 수 있다. 그 이유를 알아보자. 기존 s에는 'I love Paris!'가 저장되어 있는데, f() 함수가 들어가는 순간 global s가 선언되어 함수 밖 s, 즉 전역 변수 s의 메모리 주소를 사용한다. 그래서 이전과 달리 함수 안과 함수 밖 s는 같은 메모리 주소를 사용하게 되고, 해당 메모리 주소에 새로운 값인 'I love London!'이 할당되면 함수 밖 s에도 해당값이 할당되어 [코드 5-9]와 같은 결과가 출력되는 것이다.

## 02. 함수 심화

예제: `scoping_rule_final.py`

### ■ 변수의 사용 범위

코드 5-10 `scoping_rule_final.py`

```
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.

### ■ 재귀 함수

- 아래 코드에서 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.

코드 5-11 factorial.py

```
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: "))))
```

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

← 사용자 입력  
← 화면 출력

## 02. 함수 심화

### ■ 재귀 함수

- 만약 처음 사용자가 5를 입력했다면, 다음과 같은 순서대로 계산될 것이다.

```
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
```

03

함수의 인수

## 03. 함수의 인수

- 파이썬에서 인수를 사용하는 방법에 대해 알아보자

종류	내용
키워드 인수	함수의 인터페이스에 지정된 변수명을 사용하여 함수의 인수를 지정하는 방법
디폴트 인수	별도의 인수값이 입력되지 않을 때, 인터페이스 선언에서 지정한 초기값을 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

[파이썬에서 인수를 사용하는 방법]

### ■ 1) 키워드 인수

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

코드 5-12 keyword.py

```
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

### ■ 1) 키워드 인수 : [코드 5-12] 해석

- [코드 5-12]에서 `print_something( )` 함수는 (`my_name`, `your_name`) 입력 인터페이스를 가진다. 일반적으로 함수를 호출할 때 인수가 순서대로 들어가도록 코드를 작성한다.
- 4행 : `print_something("Sungchul", "TEAMLAB")`에서 'Sungchul'은 `my_name`에, 'TEAMLAB'은 `your_name`에 할당된다. 하지만 함수의 입력 변수명만 정확히 기재된다면, 순서에 상관없이 해당 함수에 인수를 넣을 수 있다.
- 5행 : `print_something(your_name= "TEAMLAB", my_name = "Sungchul")`에서 각각의 함수에서 사용되는 변수명을 명시함으로써, 해당 변수에 값이 할당될 수 있도록 처리하였다. 그래서 입력되는 순서에 상관없이 'Sungchul'은 `my_name`으로, 'TEAMLAB'은 `your_name`으로 할당되었다. 따라서 두 함수 호출 코드의 실행 결과가 동일하게 출력되는 것이다.

### ■ 2) 디폴트 인수

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

코드 5-13 default.py

```
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

### ■ 2) 디폴트 인수 : [코드 5-13] 해석

- 1행 : `def print_something_2(my_name, your_name = "TEAMLAB")`에서 `your_name` 매개 변수에는 기본값으로 'TEAMLAB'이 지정된 것을 확인할 수 있다. 이 경우, 함수를 호출할 때 `your_name` 매개변수에는 별도의 값을 할당하지 않아도 함수가 작동한다.
- 5행 : `print_something_2("Sungchul")`에서 함수의 인터페이스의 매개변수가 2개임에도, 인수를 하나만 입력하였다. 이 경우, 입력된 값은 첫 번째 매개변수인 `my_name`에 할당되고, 두 번째 매개변수인 `your_name`에는 디폴트 인수로 지정된 'TEAMLAB'이 할당된다. 4행과 5행의 코드를 모두 실행해도 두 코드의 결과는 같다.
- 이러한 디폴트 인수는 보통 함수를 사용하는 사람이 초깃값을 입력해 주지 않을 때, 예를 들어 초깃값을 0으로 할당하면 `'init=0'`과 같은 형태로 입력된다. 가끔은 그 변수에 특정한 값이 입력되지 않으면 사용되지 않을 때도 있는데, 이 경우에는 `'init = None'`과 같은 형식으로 초깃값을 지정한다.

## 03. 함수의 인수

### ■ 3) 가변 인수

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

## 03. 함수의 인수

예제: asterisk1.py

### ■ 3) 가변 인수

코드 5-14 asterisk1.py

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

```
7 # *를 사용한 파라미터는 나머지 인수들을 대표한다.  
8 def asterisk_test(a, b, *aaa):  
9     print(type(aaa), aaa)  
10    return a + b + sum(aaa)  
11  
12  
13 print(asterisk_test(1, 2, 3, 4, 5))
```

```
<class 'tuple'> (3, 4, 5)  
15
```

```
15
```

➡ [코드 5-14]의 asterisk\_test( ) 함수는 변수 a, b를 받고, 나머지 변수는 \*args로 받고 있다. 여기서 \*args를 가변 인수라고 한다. asterisk\_test(1, 2, 3, 4, 5)에서 1과 2는 각각 a와 b에 할당되고, 나머지 인수인 3, 4, 5가 모두 \*args에 할당된다.

### ■ 3) 가변 인수

- [코드 5-14]를 [코드 5-15]와 같이 변경한 후 실행하면, 다음과 같은 결과를 얻을 수 있다.

코드 5-15 asterisk2.py

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

가변 인수는 키워드 인수 뒷부분에 정의할 수 있다. (a, \*args, b)는 안됨

주의 !! 함수의 return value가 없음.

```
(3, 4, 5)  
None
```

- ➔ [코드 5-15]의 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 놓아야 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

예제: asterisk3.py

### ■ 3) 가변 인수

코드 5-16 asterisk3.py

개수가 정의되어 있지 않은 입력 파라미터를 일단 모두 한개의 tuple형 변수로 받은 후 이를 unpacking 해 본다.  
앞의 2개는 변수 x, y로 받고 갯수가 정해지지 않은 나머지는 z 변수(list)로 받는다.

```
9 def asterisk_test_2(*args): # 입력 파라미터가 몇 개인지 정의하지 않는다.
10     print('\nin function: ', type(args), args)
11     x, y, *z = args        # 3번째부터는 z이름으로 list로 unpacking한다.
12     print('in function: ', type(z), z)
13     return x, y, z        # 여러개를 반환할 때는 tuple로 반환한다.
14
15
16 print('in main: ', asterisk_test_2(3, 4, 5))          # (3, 4, [5])
17 print('in main: ', asterisk_test_2(3, 4, 5, 10, 20)) # (3, 4, [5, 10, 20])
```

```
in function: <class 'tuple'> (3, 4, 5)
in function: <class 'list'> [5]
in main: (3, 4, [5])

in function: <class 'tuple'> (3, 4, 5, 10, 20)
in function: <class 'list'> [5, 10, 20]
in main: (3, 4, [5, 10, 20])
```

- 입력받은 가변 인수의 개수를 정확히 안다면, x, y, z = args처럼 언패킹을 사용할 수 있다. 그러나 asterisk\_test\_2(3, 4, 5, 10, 20)으로 변경하면 오류가 발생한다. 왜냐하면 언패킹의 개수가 맞지 않기 때문이다. => [소스 프로그램의 실습 1](#)에 기술하였음.

## 03. 함수의 인수

예제: asterisk4.py

*\* asterisk3.py에 중복 설명하였으므로 생략함....*

### ■ 3) 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

### ■ 4) 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.
- 함수의 파라미터 변수를 **\*\***로 지정하여 선언하면 함수를 입력 파라미터를 '키워드=값'으로 지정하여 호출하면 함수에게는 이 파라미터는 사전 자료형 (dictionary type) 전달된다. ; {키워드:값}
- **\*\***로 지정하는 키워드 가변 인수는 반드시 모든 매개변수의 맨 마지막, 즉 가변 인수 다음에 선언되어야 한다.

- 사전 자료는 { }로 표현되며 내부 구성요소는 key:value로 구성된다.
  - `dic = {Key1:Value1, Key2:Value2, Key3:Value3, ...}`
  - `student = {1: 'Tom', 2: 'Nancy', 3:'Jane' }`
  - `food = {'carrot': 2000, 'egg': 500 }`
- 사전 자료는 immutable한 키(key)와 mutable한 값(value)으로 구성된 원소가 순서가 없이 나열된 집합이다.
  - 순서가 없기 때문에 인덱스로는 접근할 수 없고, 키로 접근한다.
  - `print(student[2])`                    *# => Nancy*
  - `student[2] = 'JH'`                    *# value 요소는 mutable*
  - `print(student[2])`                    *# JH*
  - `print(food['egg'])`                    *# => 500*

# 가변길이 인수 연습

실습2: @variable\_length\_arguments.py

```
29 def make_dictionary(max_length=10, **entries):
30     print(f'\ntype(entries)={type(entries)}, max_length={max_length}, len(entries)={len(entries)}')
31     return entries
32
33
34 a = make_dictionary(max_length=2, key1=5, key2=7, key3=9, key4=3, key5=9)
35 print('type(a)=', type(a), '| a=', a)
36 # type(entries)=<class 'dict'>, max_length=2, len(entries)=5
37 # type(a)= <class 'dict'> | a= {'key1': 5, 'key2': 7, 'key3': 9, 'key4': 3, 'key5': 9}
38
39 a = make_dictionary(1, key1=5, key2=7, key3=9)
40 print('type(a)=', type(a), '| a=', a)
41 # type(entries)=<class 'dict'>, max_length=1, len(entries)=3
42 # type(a)= <class 'dict'> | a= {'key1': 5, 'key2': 7, 'key3': 9}
43
44 a = make_dictionary(3, key1=5, key2=7, key3=9)
45 print('type(a)=', type(a), '| a=', a)
46 # type(entries)=<class 'dict'>, max_length=3, len(entries)=3
47 # type(a)= <class 'dict'> | a= {'key1': 5, 'key2': 7, 'key3': 9}
```

- \*\*를 이용하여 유입된 파라미터는 "키워드=값"의 형태로 호출되는 데 함수 안에서는 "key:value"로 구성된 사전형 자료로 취급된다.
- 가변길이의 인수의 최대수를 제한하는 기법은 본 예제의 실습 3를 참고하기 바람.

## 03. 함수의 인수

예제: kwargs.py

### ■ 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)
```

*\* key=value 들로 구성된 사전자료로 함수가 호출된다.*

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

## 03. 함수의 인수

### ■ 키워드 가변 인수 : [코드 5-18] 해석

- 키워드 가변 인수는 변수명으로 kwargs를 사용한다. 변수명 자체는 중요하지 않지만, \*는 반드시 \*\* 이렇게 2개를 붙여야 한다.
- 먼저 print( ) 함수에는 3개의 키워드 인수를 넣으므로 그 인수들이 {'first': 3, 'second': 4, 'third': 5} 형태로 출력되는 것을 확인할 수 있다. 이러한 형태를 딕셔너리 자료형이라고 하며, 실행 결과 변수명과 값이 쌍으로 저장된 것을 확인할 수 있다.
- 4행의 print("Secondvalue is {second}".format(\*\*kwargs))와 같이 개별 변수명을 따로 불러내 사용할 수도 있는데, 이 코드는 print( ) 함수에서 사용하는 출력 기능을 사용하여 변수 kwargs에 있는 second 변수를 print( ) 함수에서 사용할 수 있도록 하였다. 이미 키워드 가변 인수를 사용하여 'first = 3, second = 4, third = 5' 변수를 함수 안에 넣었기 때문에 second라는 변수를 사용할 수 있다.
- 이전 가변 인수에서 보았듯이 딕셔너리 자료형 변수에 \*를 2개 붙이면, 개별 변수로 풀려 함수에 들어갈 수 있다.

## 03. 함수의 인수

### ■ 키워드 가변 인수

- 인터프리터는 다음과 같이 해석한다.

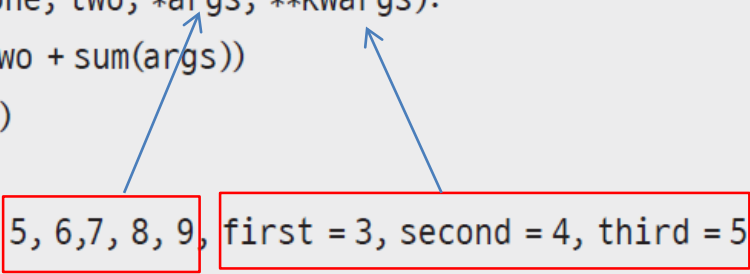
```
>>> kwargs = {'first': 3, 'second': 4, 'third': 5}
>>> print("Second value is {second}".format(**kwargs))
Second Value is 4
>>> print("Second value is {second}".format(first = 3, second = 4, third = 5))
Second Value is 4
```

## 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



## 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 8, Python 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

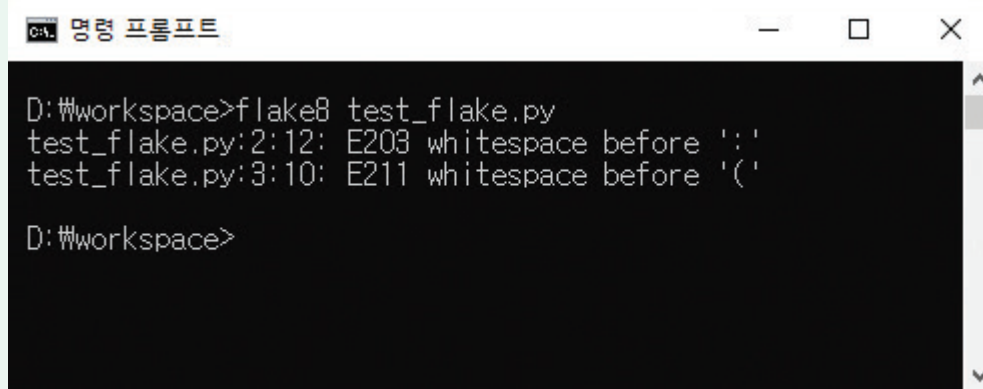
```
1 lL00 = "123"  
2 for i in 10 :  
3     print("Hello")
```

## 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

예제: hello1.py

### ■ 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")
```

함수 하나로..

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

## 04. 좋은 코드를 작성하는 방법

예제: hello2.py

### ■ 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

예제: math1.py

### ■ 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

예제: math2.py

### ■ 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]

# Thank You !