



# 컴퓨터 프로그래밍 II : Python

3주차 : 함수

조현준(Chris)

[cho1475@korea.ac.kr](mailto:cho1475@korea.ac.kr)



# 2주차 복습 : 변수와 자료형

변수란? 데이터를 자유롭게 사용하기 위한 <메모리 공간> + 이름(혹은 별칭)

수학에서:

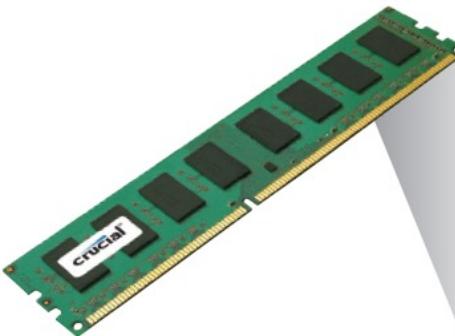
$$a = 1$$

a는 1과 같다.

프로그래밍에서:

$$a = 1$$

a에 1을 저장한다.



메모리 (memory)	주소 (address)	변수 (variable)
	0x0007	
	0x0006	
	0x0005	
	0x0004	
7	0x0003	b
1	0x0002	a
chris.cho	0x0001	name

b = 7

a = 1

name = 'chris.cho'

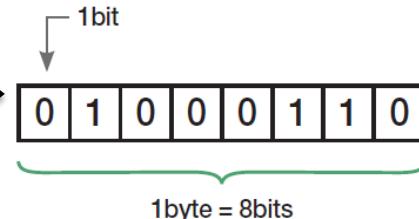
# 2주차 복습 : 변수와 자료형

자료형 == 메모리 영역의 크기

int, float, string, bool 등의 값의 종류 → **dynamic typing** : 파이썬은 자료형을 명시하지 않음

논리 타입	boolean	1byte, [False, True]
자타입	char	2byte, Unicode
정수 타입	byte	1byte, -128~127
	short	2byte, -32768~32767
	int	4byte, -2 <sup>31</sup> ~2 <sup>31</sup> -1
	long	8byte, -2 <sup>63</sup> ~2 <sup>63</sup> -1
실수 타입	float	4byte, -3.4E38~3.4E38
	double	8byte, -1.7E308~1.7E308

출처 : <https://dudri63.github.io/2019/02/13/java3/>



파이썬도 static typing concept 지원함(optional)

```
1 #type hint concept
2 b: int = 7
3 a: int = 1
4 name: str = "chris.cho"
5 print(b, a, name)
```

7 1 chris.cho

# 2주차 복습 : 자료형 변환

자료형 변환(casting) == implicit(or dynamic) vs explicit

- *implicit casting* / 불가능한 케이스 •

## 5.1 [edge case] can't implicit(or dynamic) casting

```
1 "5" + 2 #operation of string and others(int or float or bool)
```

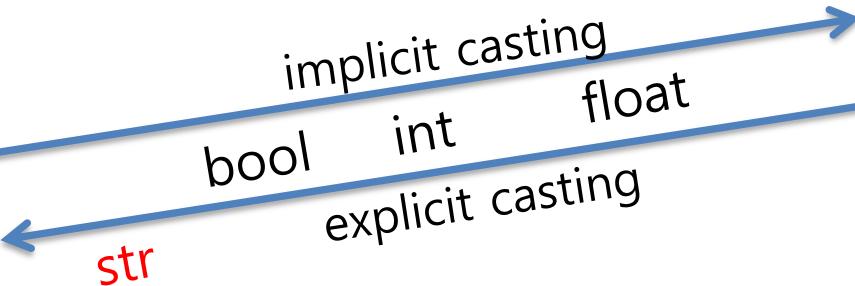
executed in 7ms, finished 12:46:30 2021-09-18

```
-----  
TypeError  
<ipython-input-15-663395cf1cb0> in <module>  
----> 1 "5" + 2 #operation of string and others(int or float or bool)
```

```
TypeError: can only concatenate str (not "int") to str
```

- (파이썬에서만) 특별히 허용하는 implicit casting

- 일반적인 casting rule



## 5.1 (only python) special implicit(or dynamic) casting

```
1 print(1/3) #int->float  
2 print(1+True) #bool-> others(int or float)
```

executed in 4ms, finished 12:43:43 2021-09-18

```
0.3333333333333333  
2
```

## 5.1 implicit(or dynamic) casting

```
1 print(1+1.3) #int -> float
```

executed in 4ms, finished 12:39:42 2021-09-18

```
2.3
```

## 5.2 Explicit casting

```
1 print(int(3.5)) #float -> int  
2 print(bool(1)) #int -> bool  
3 print(str("1")) #other -> string
```

executed in 4ms, finished 12:39:43 2021-09-18

```
3  
True  
1
```



어떻게 코드 블럭의  
복잡함을 감추고  
쉽게 코드를  
이해할 수 있게 만들까요??

# 3주차: 함수 (Function)

1. 함수 개념
2. 함수 기초
3. 함수 심화
4. 좋은 코드
5. 실습 예제

# 함수 개념



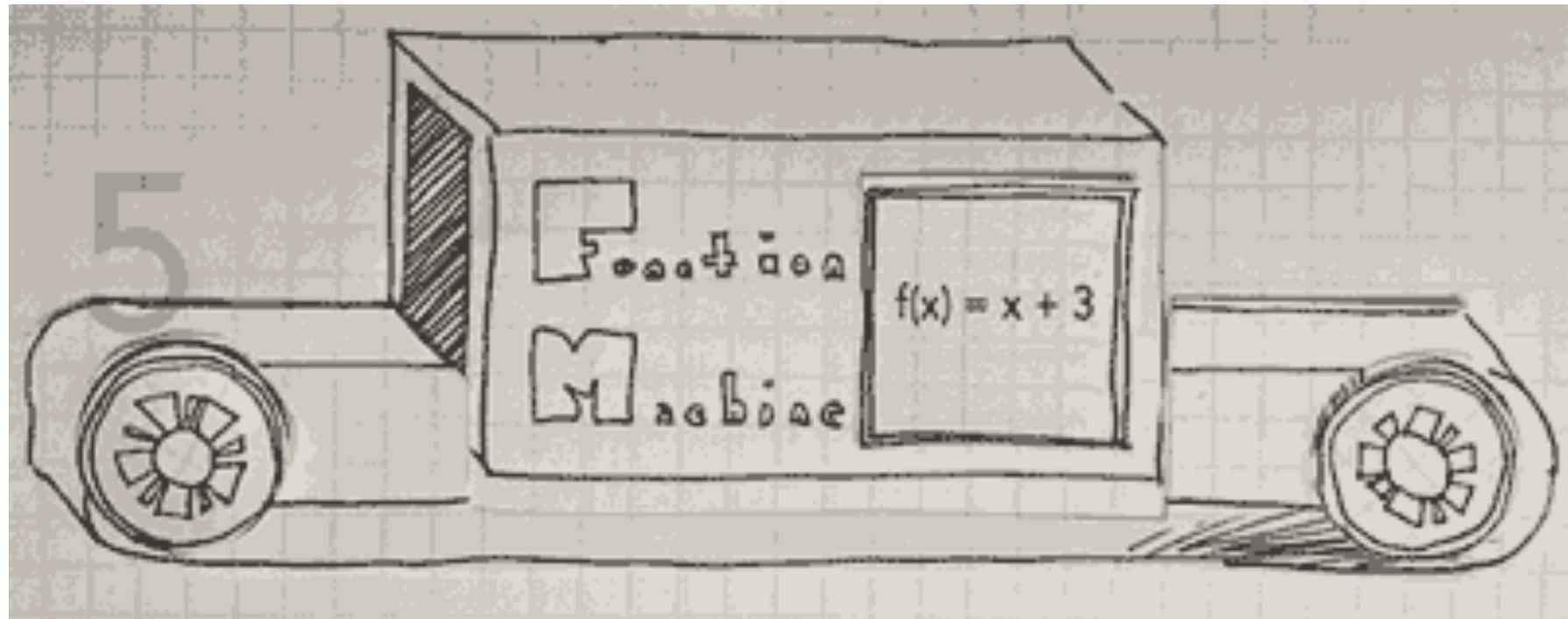


프로그래밍의 세계에서  
'함수'라는 개념은  
과연 어디서  
시작되었을까요?



# 함수의 유래

수학적 함수  $y = f(x) = x + 3$  → 입력 매개 변수(x)에 실제 인자 값을 넣고  
특정 수식을 적용하여 결과 값(y)을 반환(return)하는 것



# 함수의 개념

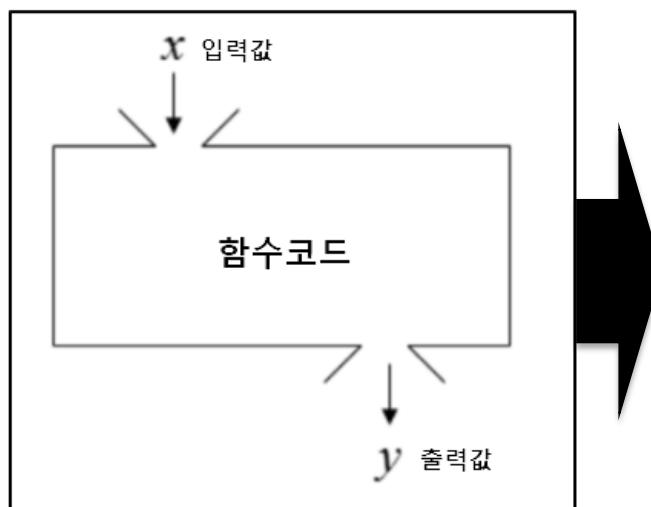
## 프로그래밍적 함수

- 특정수식 특정 작업을 수행하는 코드 블럭(혹은 코드 묶음)
- 입력 인자 값을 넣으면 특정 작업을 수행하여 원하는 결과를 얻음

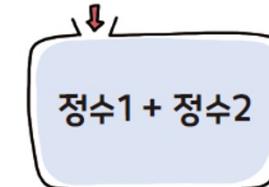
특정 작업 예시: 덧셈 같은 비교적 간단한 연산부터

네트워크 연결, 회원 인증, 메일 발송과  
같이 복잡하고 어려운 작업까지 모두 포함함

함수명

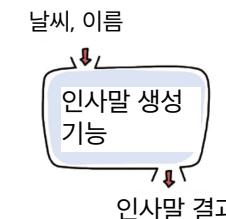


정수1, 정수2



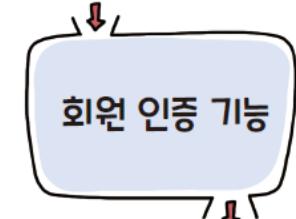
덧셈 결과

네트워크 주소



연결 결과

아이디, 패스워드



인증 결과

(a) 비교적 간단한 함수

(b) 복잡한 함수

그림 10-3 프로그램에서 함수 사용의 예

# 함수 기초



# 파이썬 함수 기초

## 함수의 정의

```
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. e.g. solution
  - 띄어쓰기를 할 경우에는 \_ 기호를 사용한다. e.g. save\_model
  - 행위를 기록하므로 동사와 명사를 함께 사용하는 경우가 많다. e.g. find\_number
  - 외부에 공개하는 함수일 경우, 줄임말을 사용하지 않고 짧고 명료한 이름을 정한다.

# 파이썬 함수 기초

## 함수의 정의

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

- ③ **매개변수(parameter)** : 함수에서 입력값으로 사용하는 변수를 의미하며, 40개 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.
- ⑤ **결과 반환(return)** : 일반적으로 함수가 외부로 결과를 전달하기 위해서는 **return** 다음에 한 칸 띄고 **반환 값(혹은 변수)**를 작성해야 한다.

# 파이썬 함수 기초

## 함수의 정의

- 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\_area이고, x와 y라는 2개의 매개변수를 사용하고 있다.
- ③ return의 의미는 값을 반환한다는 뜻으로, x와 y를 곱한 값을 반환하는 함수로 이해한다.

# 파이썬 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 반환되는 것이다. 파이썬의 함수도 같은 개념이다. 수학에서  $x$ 에 해당하는 것이 매개변수, 즉 입력값이고,  $x + 1$ 의 계산 과정이 함수 안의 코드이며, 그 결과가 출력값이다.

# 파이썬 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))
```



A terminal window showing the execution of the rectangle\_area.py script. The window has a light green background and a dark green title bar. The title bar shows the window's name and the number of tabs open. The main area of the terminal displays the following text:

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

# 파이썬 함수 기초

## 함수의 실행 순서: [코드 5-1] 해석

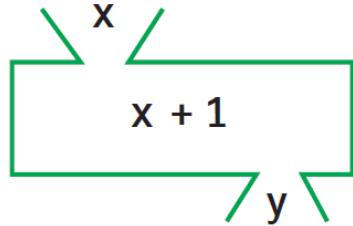
- 1행의 함수가 정의된 def 부분은 실행하지 않는다. 실행되지 않는 것처럼 보일 뿐, 해당 코드를 메모리에 업로드하여 다른 코드를 호출해 사용할 수 있도록 준비 과정을 거친다.  
만약 함수의 선언 부분을 코드의 맨 끝에 둔다면 해당 코드 호출에 오류가 발생할 것이다  
  
•  
  
• 다음으로 함수 다음의 코드를 실행한다. 정확히는 rectangle\_x = 10과 rectangle\_y = 20, 2개의 변수에 값이 할당되고 그 값이 출력된다. 다음 코드인 print("사각형의 넓이:", calculate\_rectangle\_area(rectangle\_x,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

# 파이썬 함수 기초

## 프로그래밍의 함수와 수학의 함수

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

$$f(x) = x + 1$$



```
def f(x)
    y = x + 1
    return y
```

[수학에서의 함수 형태]

# 파이썬 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?

$f(2) = 11, g(2) = 4, f(g(2)) = 15, g(f(2)) = 121$

$\therefore 11 + 4 + 15 + 121 = 151$

# 파이썬 함수 기초

## 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)))
```



- 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.

## Parameters

# Function Definition

```
def add(a, b):  
    return a + b
```

# Function Call

```
add(2, 3)
```

안에 들어갈  
이것은  
무엇일까요?

# 파이썬 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, 결 과값 반환

[함수의 형태]

# 파이썬 함수 기초

## 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
```

# 파이썬 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 것 이 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.



1부 끝  
10분 휴식





# 함수 심화

# 함수 심화

기본값 인자(default argument)

```
def add_numbers(n1 = 100, n2 = 1000):
```

```
    result = n1 + n2
```

```
    return result
```

```
result = add_numbers(5.4)
```

```
print(result)
```

```
n1 = 100, n2 = 1000
```

# 함수 심화

키워드 인자(keyword argument) vs 위치 인자(positional argument)

All positional arguments

`foo(3, 4, 5)`



Positional argument(s) followed  
by keyword argument(s)

`foo(3, b=4, c=5)`

`foo(3, 4, c=5)`



All keyword arguments

`foo(a=3, b=4, c=5)`



Keyword argument(s) followed  
by positional argument(s)

`foo(a=3, b=4, 5)`

`foo(a=3, 4, c=5)`

`foo(3, b=4, c)`



# 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

# 함수 심화

## 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)
```

```
-
```

```
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
```

# 함수 심화

## 변수의 사용 범위 : [코드 5-7] 해석

- 관심을 두어야 할 변수는 x와 t이다. 프로그램이 가장 먼저 시작되는 지점은 6행의 `x = 10`이다. 그리고 7행에서 x는 `test(x)` 함수로 변수를 넘기게 된다. 그렇다면 함수 안에서 처음 만나는 2행 `print(x)`의 x는 어떤 변수일까? 이때의 x는 함수 안에서 재정의되지 않았으므로 함수를 호출한 메인 프로그램의 `x = 10`의 x를 뜻한다. 즉, 프로그램 전체에서 사용할 수 있는 전역 변수이다.
- 함수 안의 t는 `test(x)` 함수의 x를 t로 치환하여 사용한다. 즉, 함수 안에서는 x를 따로 선언한 적은 없고, t를 선언하여 사용하는 것이다. 그리고 3행의 `t = 20`에 의해 t에 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 t가 함수 안에서만 사용할 수 있는 지역변수이기 때문이다.

# 함수 심화

## 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)
```



The screenshot shows a terminal window with a light gray background. At the top, there's a green header bar with three small icons: a minus sign, a maximize button, and three dots. The main area of the terminal contains two lines of text in a dark blue font: "I love London!" on the first line and "I love Paris!" on the second line.

# 02. 함수 심화

## 변수의 사용 범위 : [코드 5-8] 해석

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

# 함수 심화

## 변수의 사용 범위

- 그렇다면 함수 안의 변수와 함수 밖의 함수가 같은 이름을 사용하기 위해서는 어떻게 해야 할까? 함수 내에서 전역 변수로 선언된 변수를 사용하기 위해서는 global이라는 파일에서 제공하는 키워드를 사용해야 한다.

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)
```

The screenshot shows a terminal window with a light green header bar containing a minus sign and three dots. The main area of the terminal displays two lines of text: "I love London!" followed by "I love London!". This indicates that the variable 's' was modified within the function 'f()' and that this change was reflected when the function was called again at line 8.

# 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.

# 함수 심화

## 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
```



# 함수 더욱 심화

Higher-order Functions

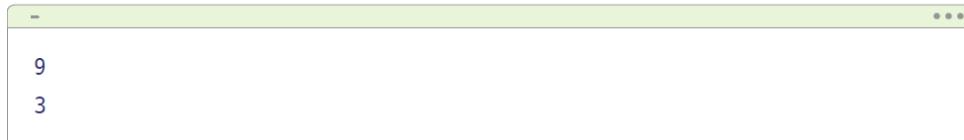
Higher-orderder fucions

# 함수 심화

## 함수의 호출 방식

코드 5-5 call1.py

```
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)
```



```
-  ...
9
3
```

# 함수 심화

## 함수의 호출 방식 : [코드 5-5] 해석

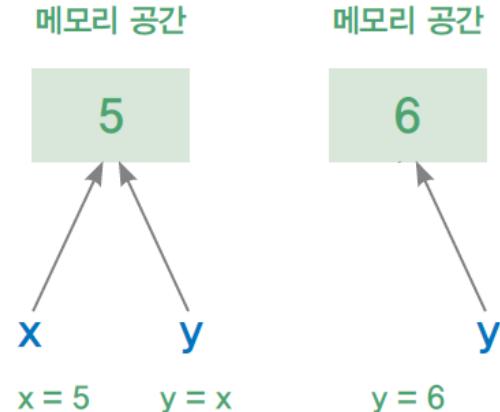
- 함수 밖에 있는 변수 x의 메모리 주소와 함수 안에 있는 변수 x의 메모리 주소가 같은지 다른지 확인할 필요가 있다.
- 함수 안에 변수가 인수로 들어가 사용될 때, 변수를 호출하는 방식을 전통적인 프로그래밍에서는 다음과 같이 크게 두 가지로 나눈다.

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>

[함수가 변수를 호출하는 방식]

# 함수 심화

파이썬 함수의 호출 방식(call by object reference)



### [파이썬에서 변수를 호출하는 방식]

- 파이썬은 객체의 주소가 함수로 넘어간다는 뜻으로, 객체 호출(call by object reference)로 명명되는 방식을 사용한다. 파이썬에서는 새로운 값을 할당하거나 해당 객체를 지울 때는 영향을 주지 않고, 단순히 해당 객체에 값을 초기화 때는 영향을 준다.

# 함수 심화

## 재귀 함수

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

$$1! = 1$$

$$2! = 2(1) = 2$$

$$n! = n \cdot (n - 1) \cdots 2 \cdot 1 = \prod_{i=1}^n i \quad 3! = 3(2)(1) = 6$$

$$4! = 4(3)(2)(1) = 24$$

$$5! = 5(4)(3)(2)(1) = 120$$

[점화식]

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

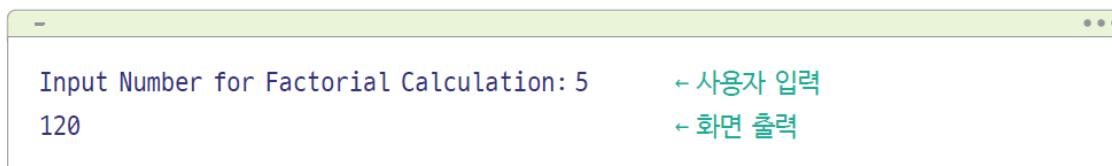
# 함수 심화

## 재귀 함수

- 아래 코드에서 factorial( ) 함수는 n의 변수를 입력 매개변수로 받은 후  $n == 1$ 이 아닐 때 까지 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: "))))
```



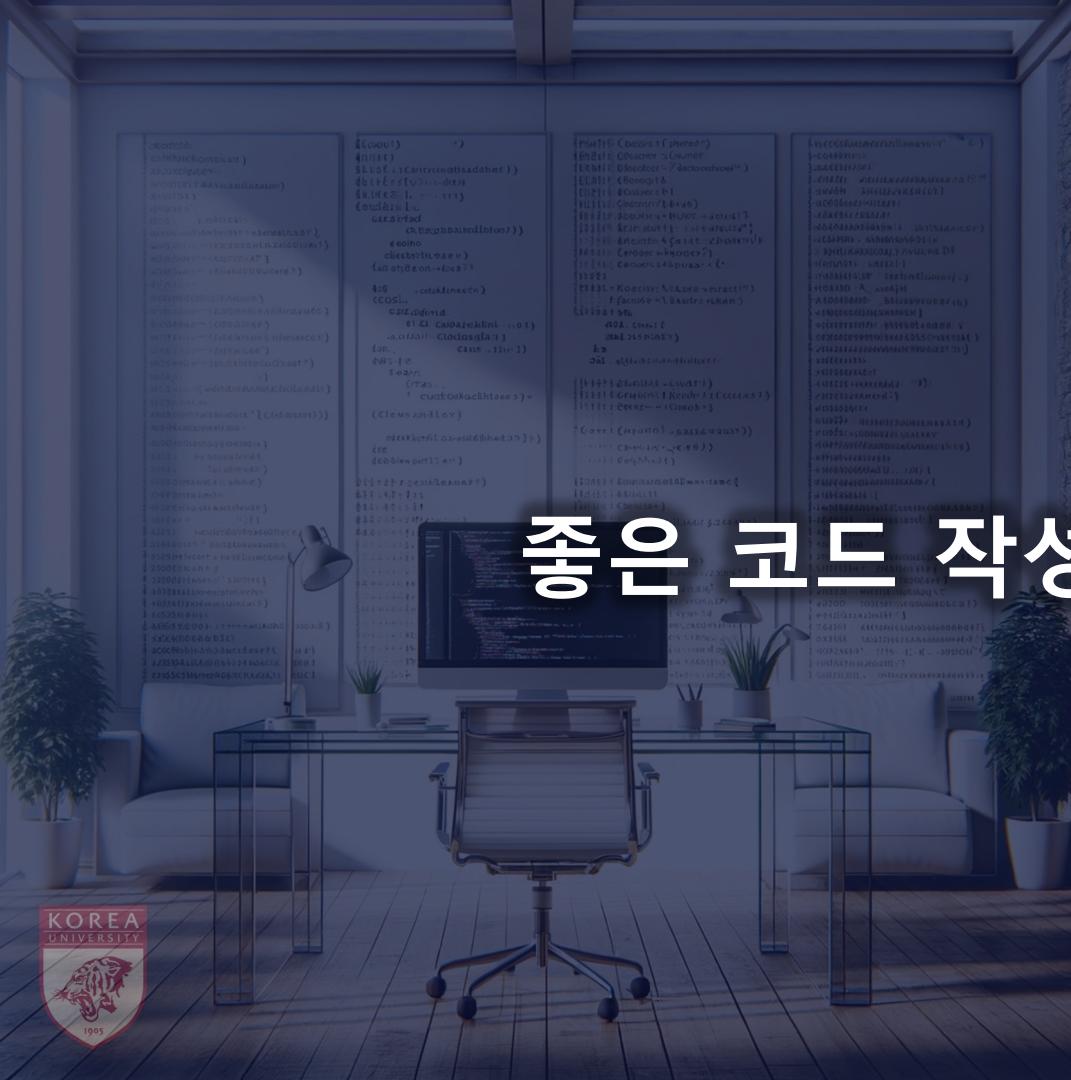
# 함수 심화

## 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
```

# 좋은 코드 작성하는 법



# 03. 좋은 코드를 작성하는 방법

## 좋은 코드의 의미

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



[페이스북 사무실]

# 03. 좋은 코드를 작성하는 방법

## 좋은 코드의 의미

- 가독성 좋은 코드를 작성하기 위해서는 여러 가지가 필요하지만, 일단 여러 사람의 이해를 돋기 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 짠다."

- 마틴 파울러

# 03. 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.

# 03. 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 간단하고, 불필요한 주석은 삭제한다.
- 소문자 l, 대문자 O, 대문자 I는 사용을 금한다.

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

- 함수명은 소문자로 구성하고, 필요하면 밑줄로 나눈다.

# 03. 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"
```

# 03. 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
```

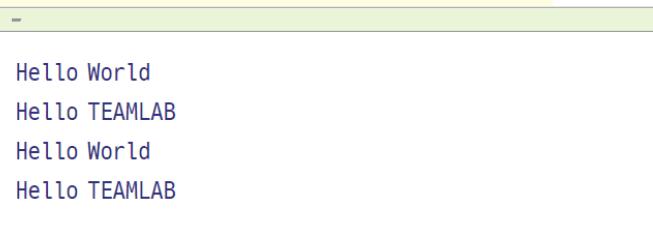
# 03. 좋은 코드를 작성하는 방법

## 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")
```



A screenshot of a terminal window showing the output of the Python code. The terminal has a light green header bar with a small minus sign icon. The main area displays four lines of text in black font: "Hello World", "Hello TEAMLAB", "Hello World", and "Hello TEAMLAB".

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

# 03. 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()
```



A terminal window showing the output of the Python code. The output consists of four lines of text: "Hello World", "Hello TEAMLAB", "Hello World", and "Hello TEAMLAB".

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

# 03. 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))
```



A screenshot of a terminal window with a light gray background. The window title bar is at the top. Inside the window, there are two lines of text output: "1.0" on the first line and "1.0" on the second line. The window has standard OS X-style scroll bars on the right side.

# 03. 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))
```



A screenshot of a terminal window showing the output of the code execution. The window has a title bar with three dots on the right. The main area contains the text [1.0, 1.0] in blue, indicating the result of the print statement.

# 실습 예제

<http://qj.rightline.kr/contest>



# Q&A



# 감사합니다.

본 강의 교안은 한빛 아카데미(주)에서 제공된  
강의 교안을 참고하여 제작되었습니다.

