

IT 개론

6장. 함수

목차

1. 추상화 개념
2. 함수 소개
3. 파이썬 내장 함수
4. 사용자 정의 함수
5. 함수의 인수
6. 전역 변수와 지역 변수
7. 람다 함수 (lambda)
8. 반복 함수와 재귀 함수


함수

함수를 정의한다(define)

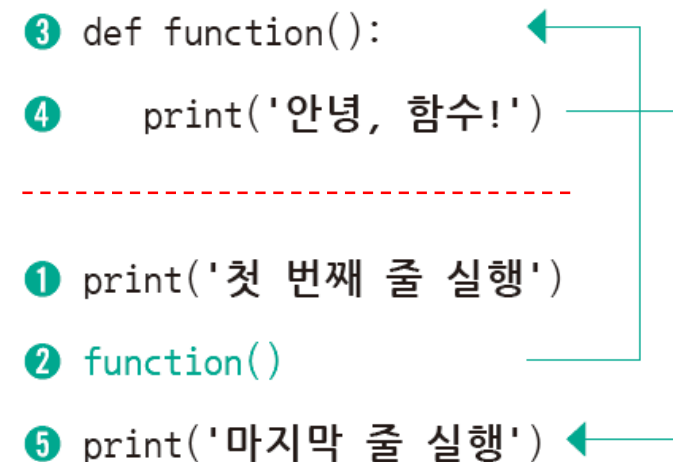
함수 이름

 **def**  **function**():

print('안녕, 함수!')

 해당 함수가 실행할 코드

함수

if 조건문	함수
<pre>❶ print('첫 번째 줄 실행') ❷ if 조건: ❸ print('안녕, if!') ❹ print('마지막 줄 실행')</pre>	<pre>❸ def function(): ❹ print('안녕, 함수!') ----- ❶ print('첫 번째 줄 실행') ❷ function() ❺ print('마지막 줄 실행')</pre> 

함수

코드 function2.py

```
def function():  
    print('안녕, 함수!')
```

```
PS C:\Users\User\Documents> python function2.py  
PS C:\Users\User\Documents>
```

아무 일도 일어나지 않는다

```
def function():  
    print('안녕, 함수!')
```



```
function()
```

안녕, 함수!

함수 function() 실행됨

함수

코드

```
def print_root(a, b, c):    # 소괄호 안을 a, b, c로 채웁니다.  
    r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)  
    r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)  
  
    print('해는 {} 또는 {}'.format(r1, r2))
```

(코드 줄임)

함수

```
x = 1                                # 변수명을 바꿨습니다.  
y = 2  
z = -8  
  
#  $a * x^2 + b * x + c = 0$ ,  $a \neq 0$  인  $x$ 에 관한 이차방정식에서  
# 근의 공식은  
  
print_root(x, y, z)  
  
x = 2  
y = -6  
z = -8  
  
# 한 번 더 구하려면  
print_root(x, y, z)
```

해는 2.0 또는 -4.0

해는 4.0 또는 -1.0

함수

```
def print_root(a, b, c):
```

(함수 내용)

...

```
print_root(x, y, z)
```

매개변수

실행인자

매개변수 a, b, c 가 추가됨으로서 이전에는 함수 밖의 a, b, c 를 참조했지만 지금은 괄호 안에 주어진 세 개의 실행인자 x, y, z 에 의해 a, b, c 가 결정된다.

함수

코드

```
def print_round(number):  # 함수를 정의합니다.  
    rounded = round(number)  
    print(rounded)  
  
print_round(4.6)          # 바로 숫자를 넣어서 함수를 실행합니다(함수 호출).  
print_round(2.2)  
print_round(3.6)
```

5
2
4

실행인자로 꼭 변수를 넘기지 않고 필요한 값을 직접 넣어 사용할 수도 있다.

함수

	매개변수	실행인자
영문 이름	parameter	argument
역할	함수를 정의할 때 사용하는 이름	함수를 실행할 때 넘기는 변수 또는 값
예시	<code>def print_root(a, b, c):</code> 또는 <code>def print_round(number):</code>	<code>print_root(x, y, z)</code> 또는 <code>print_round(4.6)</code>
주의점	<ul style="list-style-type: none">매개변수와 실행인자가 여러 개면 쉼표(,)로 구분합니다.매개변수와 실행인자의 개수는 같아야 합니다(개수가 다르면 오류가 발생합니다).	

함수

코드 function4.py

```
def add_10(value):  
    '''value에 10을 더한 값을 돌려주는 함수'''  
    result = value + 10  
    return result  
  
add_10(42)
```

```
PS C:\Users\User\Documents> python function4.py
```

```
PS C:\Users\User\Documents>
```

아무 것도 출력되지 않는다.

함수

코드 function4.py

```
def add_10(value):  
    '''value에 10을 더한 값을 돌려주는 함수'''  
    result = value + 10  
    return result  
  
n = add_10(42)  
print(n)
```

52

`add_10(42)`는 $42 + 10$, 즉 52를 return한다.
`n`에 `add_10(42)`가 할당되어 있으므로, `n`에 52가
return된다.

함수

코드

```
def add_10(value):  
    '''value에 10을 더한 값을 돌려주는 함수'''  
    return 10  
  
    result = value + 10  
    return result
```

10

return은 실행된 즉시 함수 실행을 끝낸다.

따라서 return 10에서 함수 add_10은 종료된다.

함수

코드

```
def add_10(value):  
    '''value에 10을 더한 값을 돌려주는 함수'''  
    if value < 10:  
        return 10  
    result = value + 10  
    return result  
  
n = add_10(5)  
print(n)  
  
n = add_10(42)  
print(n)
```

10

52

함수

코드 function5.py

```
def root(a, b, c):    # 출력하는 기능이 없으니 함수 이름을 print_root에서 root로 수정
    r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
    r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
    return r1

r = root(1, 2, -8)
print('근은 {}'.format(r))
```

근은 2.0

이차방정식의 근을 return해야 하는데 한 개의 값만 return이 되고 있다.

함수

코드

```
def root(a, b, c):  
    r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)  
    r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)  
    return r1, r2          # 값 두개를 return한다  
  
r1, r2 = root(1, 2, -8)    # r1, r2에 순서대로 들어간다.  
print('근은 {}와/과 {}'.format(r1, r2))
```

근은 2.0와/과 -4.0

위 방식으로 여러 개의 값을 return 하는 것이 가능하다.

1. 추상화 개념

◆ 추상화(abstraction)

- 추상화는 무언가를 숨긴다는 것으로 현실 세계의 사물을 개념화하고 단순화시키는 것이다.
- 현실 세계의 사물을 구성하는 데이터와 그 데이터에 적용하는 기능으로 나누어서 추상화하는 것이 일반적이다.

◆ 두 가지 추상화

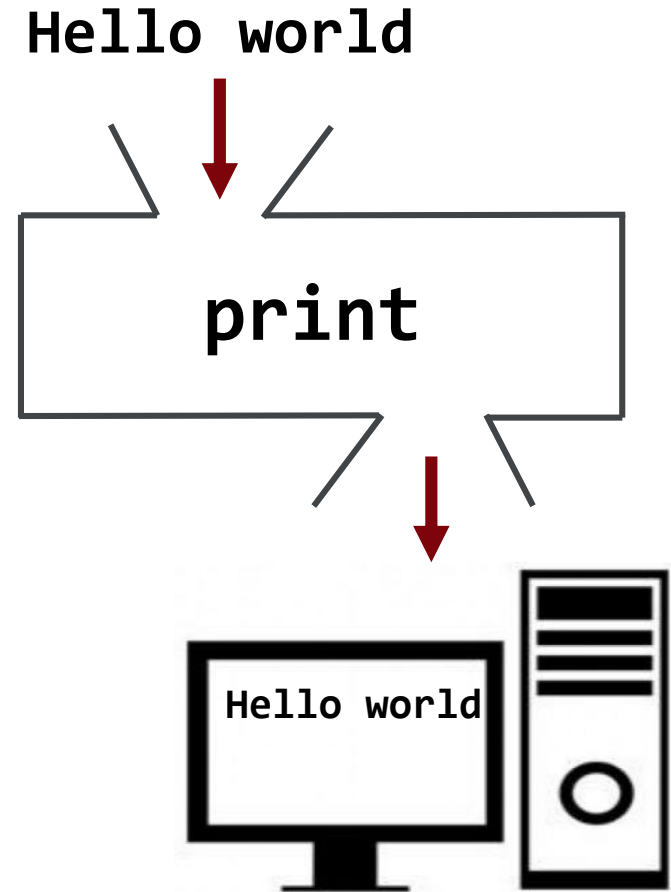
- Control abstraction (기능 추상화) → 함수
- Data abstraction (데이터 추상화) → 클래스 (8장)

2. 함수

◆ 함수는 블랙 박스 (black box)이다

- 함수는 입력과 출력을 갖는 black box이다.
- 주어진 입력에 대해서 어떤 과정을 거쳐 출력이 나오는지 가 숨겨져 있다.

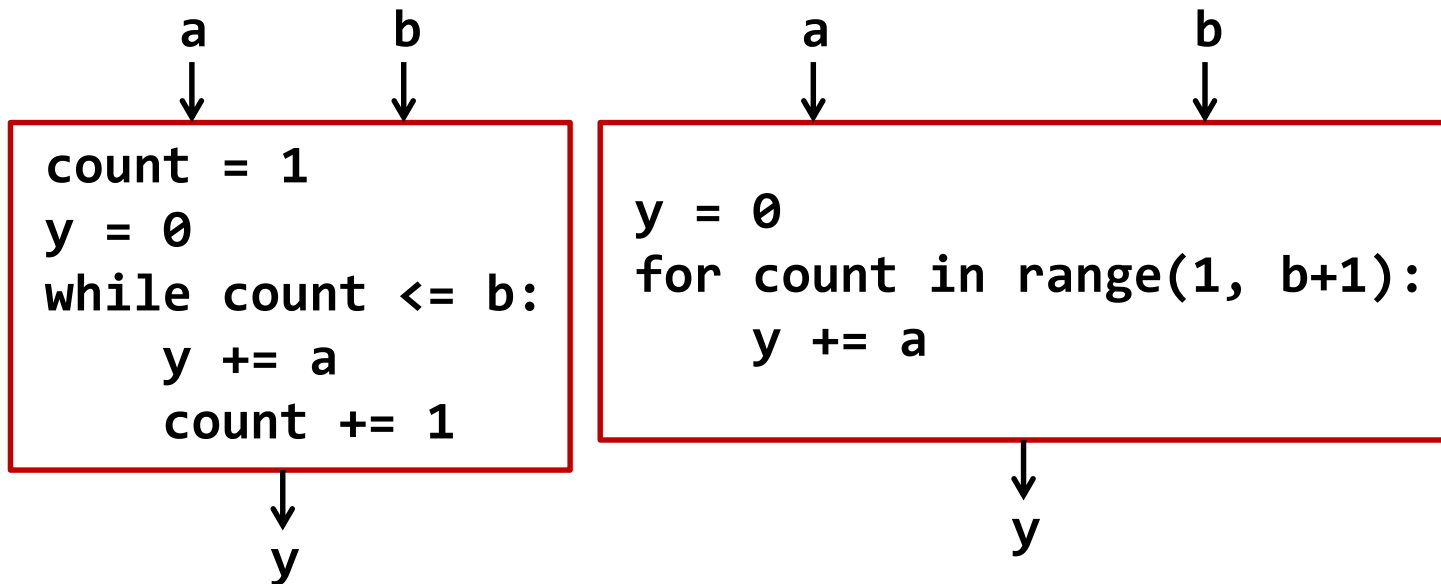
```
>>> print('Hello world')
```



2. 함수

◆ 함수는 블랙 박스 (black box)이다

- 예제) 두 수를 입력받아서 두 수의 곱을 구하고자 한다.
 - 두 박스 부분이 다르지만 (즉, 구현이 다르지만) 입력 a , b 가 같은 값이 주어지면 동일한 결과 y 값을 출력한다.
 - 함수 사용자는 박스 안의 구현은 신경쓰지 않고 주어진 입력에 대해서 어떤 결과가 출력되는지만 알면 된다.



2. 함수

◆ 함수를 사용하는 이유

- 프로그램 구성을 용이하게 한다.
 - 큰 문제를 작은 부분 문제로 나누어서(분할) 다룰 수 있도록 함으로써 프로그램 구성을 융통성있게 할 수 있다.
- 한 번 작성한 함수는 여러 곳에서 재사용이 가능하다. 즉, 필요할 때마다 호출하여 사용할 수 있다.

◆ 함수 작성시에 중요한 부분

- **함수 이름** - 함수가 하는 일을 적절하게 표현하는 이름
- **입력과 출력**을 분명하게 명시

3. 파이썬 내장 함수

◆ 내장 함수 (built-in functions)

- 파이썬 언어에는 이미 만들어서 제공하는 함수들
- IDLE에서 `dir(__builtins__)`라고 입력하면 파이썬에서 제공하는 함수 리스트를 볼 수 있다.
- 내장 함수에 어떤 것들이 있는지 학습하고 적절히 사용할 줄 아는 것이 중요하다.

◆ 사용자 정의 함수 (user-defined functions)

- 사용자가 직접 만드는 함수
- 함수 작성 문법을 익히고 직접 작성해 보는 것이 중요하다.

3. 파이썬 내장 함수

```
>>> dir(__builtins__)
['ArithmeticError', 'ZeroDivisionError', '_',
 '__build_class__', '__debug__', '__doc__', '__import__',
 '__loader__', '__name__', '__package__', '__spec__', 'abs',
 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes',
 'callable', 'chr', 'classmethod', 'compile', 'complex',
 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float',
 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance',
 'issubclass', 'iter', 'len', 'license', 'list', 'locals',
 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct',
 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range',
 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple',
 'type', 'vars', 'zip']
```

3. 파이썬 내장 함수

◆ iterable 과 iterator

- 파이썬 언어에는 이미 만들어서 제공하는 함수들
- IDLE에서 `dir(__builtins__)`라고 입력하면 파이썬에서 제공하는 함수 리스트를 볼 수 있다.
- 내장 함수에 어떤 것들이 있는지 학습하고 적절히 사용할 줄 아는 것이 중요하다.
- iterable : str, list, tuple, set, dict
(__iter__ 메소드를 갖는 객체들)

3. 파이썬 내장 함수

함수	설명	
abs	<pre>>>> abs(-4) 4 >>> abs(-5.3) 5.3 >>> abs(2) 2</pre>	
all(iterable)	iterable이 모두 True이면 True를 반환	
	<pre>>>> all([1,2,3]) True >>> all([0,1,2,3]) False >>> all({'a','b'}) True >>> all({'a','b',' '}) True >>> all({'a','b',' '}) False</pre>	<pre>>>> all((1,2,3)) True >>> all('hello') True >>> all({1:'one', 2:'two'}) True >>> all({0:'zero'}) False</pre>

3. 파이썬 내장 함수

함수	설명
any(iterable)	iterable에서 적어도 하나가 True이면 True를 반환
	<pre>>>> any([1,2,3]) True >>> any([0,1,2,3]) True >>> any({'a', 'b'}) True</pre>

- 다음은 모두 False이다.

```
>>> bool(0) # 0 외의 다른 수는 True
False
>>> bool([]) # 빈 리스트는 False
False
>>> bool({}) # 빈 사전은 False
False
>>> bool('') # 빈 문자열은 False
False
```

```
>>> bool(False)
False
>>> bool(()) # 빈 튜플은 False
False
>>> bool(set()) # 빈 집합은 False
False
```

3. 파이썬 내장 함수

함수	설명	
bin oct hex	<pre>>>> bin(10) # 10을 2진수로 변환 >>> oct(10) # 10을 8진수로 변환 '0o12' >>> hex(10) # 10을 16진수로 변환 '0xa'</pre>	
chr ord	chr 함수 : 아스키코드 → 문자로 변환 ord 함수는 : 문자 → 아스키코드로 변환	
	<pre>>>> print(ord('a')) 97 >>> print(chr(65)) A</pre>	
bool int float complex	<pre>>>> x = int(5.3) >>> print(x) 5 >>> x = int(5.8) >>> print(x) 5</pre>	<pre>>>> y = float(2) >>> z = float(0) >>> print(y,z) 2.0 0.0 >>> x = complex(5) >>> print(x) (5+0j)</pre>

3. 파이썬 내장 함수

함수	설명
str	<pre>>>> L = [1,2,3] ; T = (1,3,5) ; S = {5,6,7} >>> D = {1:'one', 2:'two'} >>> w = str(L) ; x = str(T) ; y = str(S) ; z = str(D) >>> w '[1, 2, 3]' >>> x '(1, 3, 5)' >>> y '{5, 6, 7}' >>> z '{1: 'one', 2: 'two'}'</pre>
list	<pre>>>> string = "python" >>> a = list(string); b = list(T); c = list(S); d = list(D) >>> print(a, b, c, d) ['p', 'y', 't', 'h', 'o', 'n'] [1, 3, 5] [5, 6, 7] [1, 2]</pre>

3. 파이썬 내장 함수

함수	설명
tuple	<pre>>>> string = "python" >>> L = [1,2,3] ; S = {5,6,7} ; D = {1: 'one', 2: 'two'} >>> w = tuple(string) ; x = tuple(L) ; y = tuple(S) >>> z = tuple(D) >>> w ('p', 'y', 't', 'h', 'o', 'n') >>> print(x,y,z) (1, 2, 3) (5, 6, 7) (1, 2)</pre>
set	<pre>>>> T = (1,3,4,6,3,2,5,6) >>> w = set(string) >>> x = set(L) >>> y = set(T) >>> z = set(D) >>> w {'n', 'p', 'h', 't', 'o', 'y'} >>> print(x,y,z) {1, 2, 3} {1, 2, 3, 4, 5, 6} {1, 2}</pre>

3. 파이썬 내장 함수

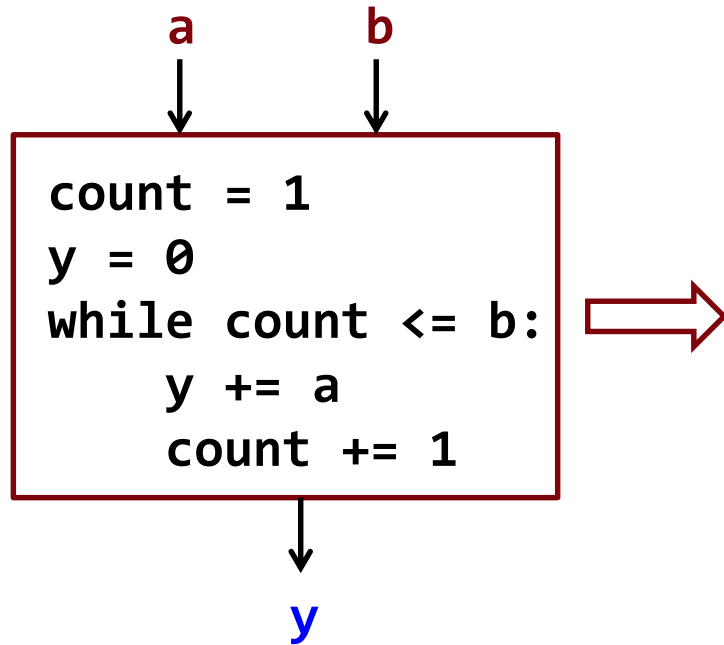
함수	설명
enumerate	<pre>>>> L = ['red', 'yellow', 'blue'] >>> E = enumerate(L) # 문자열, 튜플, 문자열 >>> EL = list(E) >>> print(EL) [(0, 'red'), (1, 'yellow'), (2, 'blue')] >>> for i in enumerate(L): print(i) (0, 'red') (1, 'yellow') (2, 'blue')</pre>
id	<pre>>>> a = 10 >>> print(id(a)) 504091232 >>> x = 'python' >>> print(id(x)) 4807136</pre>

3. 파이썬 내장 함수

함수	설명
<code>isinstance</code>	<pre>>>> isinstance(a, int) True >>> isinstance(3.5, float) True >>> isinstance([1], list) True</pre>
<code>zip</code>	<pre>>>> list(zip([1,3,5],[2,4,6])) [(1, 2), (3, 4), (5, 6)] >>> list(zip([1,2,3], "abc")) [(1, 'a'), (2, 'b'), (3, 'c')] >>> list(zip([1,3,5],{7,8,9}, (10,20,30))) [(1, 8, 10), (3, 9, 20), (5, 7, 30)]</pre>

4. 사용자 정의 함수

◆ 함수 구조



a와 **b**를 매개변수
(parameter)라고 한다.

함수명 : multiply

```
def multiply (a,b):
    count = 1
    y = 0
    while count <= b:
        y += a
        count += 1
    return y
```

y를 반환값(return)이라고
한다.

두 함수는 같은 기능을 하는 함수이다. 즉, 두 수를 입력 받아 $a*b$ 를 반환한다. 추상화를 통하여 함수의 기능을 숨김을 보여준다.

4. 사용자 정의 함수

- ◆ 가장 간단한 함수 형태 (매개변수, 반환값 없는 경우)
 - 함수에 입력이 없을 수도 있다 (빈 괄호로 둔다)
 - 함수에 출력이 없을 수도 있다 (return 구문이 없다)

```
def hello():  
    print('hello world')  
    print('hello python~')
```

main - 여기에서 프로그램 수행 시작

```
print('start of the program')  
hello()  
print('middle of the program')  
hello()  
print('end of the program')
```

```
start of the program  
hello world  
hello python~  
middle of the program  
hello world  
hello python~  
end of the program
```


4. 사용자 정의 함수

◆ 함수 매개변수 (입력)

- 함수에 입력이 없으면 빈 괄호로 둔다.

```
def add_to_ten():  
    y = 0  
    for a in range(11):  
        y += a  
    return y
```

- 함수에 입력이 여러 개일 수도 있다.

예제 - 국어, 영어, 수학 세 과목 성적을 입력으로 받아서 평균을 구한다.

```
def average(kor, eng, math):  
    sum = kor + eng + math  
    avg = sum / 3  
    return avg
```

4. 사용자 정의 함수

◆ 함수의 반환값 (return 문)

- 내장 함수와 마찬가지로 함수를 호출한 자리에는 함수의 반환값이 대체된다.
- 예) 양의 정수 n 을 입력받아 1부터 n 까지의 합을 구하는 프로그램을 함수로 작성하시오.

```
def add_to_n(n):  
    sum = 0  
    for a in range(n+1):  
        sum += a  
    return sum
```

main 여기에서부터 프로그램이 수행됨.

```
x = int(input('Enter n : '))
```

```
y = add_to_n(x)      # add_to_n 함수 호출. add_to_n 함수 수행됨.  
print(y)
```

4. 사용자 정의 함수

◆ 함수의 반환값 (return 문)

- 함수에 반환값이 2개 이상 있을 수도 있다. (튜플 처리)
- 반환하고자 하는 값들을 콤마로 분리하여 반환한다.
- 반환하고자 하는 값들을 튜플로 묶어서 반환한다.
- 예) 두 수를 입력받아서 두 수의 합과 두 수의 곱을 반환하는 함수

```
def add_multiply(a,b):  
    x = a + b  
    y = a * b  
    return x,y      # return (x,y) 라고 해도 된다.
```

파이썬에서는 데이터를 콤마로 분리하면 튜플로 인식한다.

4. 사용자 정의 함수

◆ 함수의 반환값 (return 문)

- 예제 - main에서 두 수를 입력받아 함수로 넘겨서 두 수의 합과 곱을 반환한다.

```
def add_multiply(x,y):  
    sum = x + y  
    mult = x * y  
    return sum, mult    # 튜플로 반환한다.  
  
if __name__ == '__main__':    # main임을 이렇게 표현하기도 한다.  
    a = int(input('Enter a : '))  
    b = int(input('Enter b : '))  
    m,n = add_multiply(a,b)  
    print(m,n)
```

5. 함수의 인수

◆ 인수의 기본값

- 함수를 호출할 때 인수를 넘겨주지 않아도 인수가 자신의 기본값을 취하도록 하는 기능.

```
>>> def inc(a, step=1):  
    return a + step
```

```
>>> b = inc(10)
```

```
>>> print(b)
```

```
11
```

```
>>> c = inc(10, 50)
```

```
>>> print(c)
```

```
60
```

- 첫 번째 매개 변수 a 에는 반드시 인자값을 넘겨야 한다.
- 두 번째 매개 변수인 step에는 값을 넘기면 넘기는 값 step이 되고, 값을 넘기지 않으면 1이 default로 이용된다.

5. 함수의 인수

◆ 인수의 기본값

- 기본값이 정의된 인수 다음에 기본값이 없는 인수가 올 수 없다.

```
>>> def inc(step=1, a):  
    return a + step
```

SyntaxError: non-default argument follows default argument

- 인수가 여러 개인 경우

```
>>> def foo(a, b=1, c=2):    # OK  
    return a + b + c
```

```
>>> def foo2(a, b, c=10):    # OK  
    return a + b + c
```

```
>>> def foo3(a, b=1, c):     # error  
    return a + b + c
```

SyntaxError: non-default argument follows default argument

5. 함수의 인수

◆ 키워드 인수

- 인수 이름으로 값을 전달하는 방식.

```
>>> def area(x, y):  
    return x * y
```

```
>>> area(10,5)  
50
```

```
>>> area(y=5, x=10) # 매개변수와 값을 같이 적어 준다  
50
```

```
>>> area(10, y=5)    # OK  
50
```

```
>>> area(x=10, 5)    # error  
SyntaxError: non-keyword arg after keyword arg
```

5. 함수의 인수

◆ 키워드 인수

- 키워드 인수의 위치는 보통의 인수 이후이다.

```
>>> def volume(x,y,z):  
        return x * y * z
```

```
>>> volume(1,3,5)
```

```
15
```

```
>>> volume(y=7,z=5,x=2)
```

```
70
```

```
>>> volume(z=2,x=4,y=5)
```

```
40
```

```
>>> volume(5, z=10, y=2)
```

```
100
```

```
>>> volume(5, x=2, z=20)    # error
```


5. 함수의 인수

◆ 가변 인수 리스트

- 고정되지 않은 수의 인수를 함수에 전달하는 방법이 있다.
- 함수를 정의할 때 인수 목록에 반드시 넘겨야 하는 고정 인수를 우선 나열하고, 나머지를 **튜플** 형식으로 한꺼번에 받는다.

```
>>> def foo(a, *b):  
    print(a, b)  
>>> foo()  
.....  
TypeError: .....  
>>> foo(5)  
5 ()  
>>> foo(5,6)  
5 (6,)  
>>> foo(1,2,3,4,5)  
1 (2, 3, 4, 5)
```

```
>>> def avg(first, *rest):  
    return (first+sum(rest))/(1+len(rest))  
  
>>> avg(10,20)  
15.0  
>>> avg(10,20,30,40,50)  
30.0
```

5. 함수의 인수

◆ 가변 인수 리스트 예

```
def test_args(arg, *args):  
    print("arg :", arg)  
    for x in args:  
        print("in args :", x)
```

```
>>> test_args(1, 2, 3, 4)  
arg : 1  
in args : 2  
in args : 3  
in args : 4
```

```
>>> test_args(4, 'red', 'blue', 5)  
arg : 4  
in args : red  
in args : blue  
in args : 5
```

5. 함수의 인수

◆ 정의되지 않은 키워드 인수 처리하기

- 만일 미리 정의되어 있지 않은 키워드 인수를 받으려면, 함수를 정의할 때 마지막에 ****kw** 형식으로 기술한다.
- 전달받는 형식은 **사전**이다. 즉, 키는 키워드(변수명)가 되고, 값은 키워드 인수로 전달되는 값이 된다.

```
>>> def foo(x, y, **kw):  
    print(x,y)  
    print(kw)
```

```
>>> foo(x=5, y=6, z=7)
```

```
5 6
```

```
{'z': 7}
```

```
>>> foo(x=10, y=20, a=1, b=2, c=3)
```

```
10 20
```

```
{'c': 3, 'b': 2, 'a': 1}
```

5. 함수의 인수

◆ 정의되지 않은 키워드 인수 처리하기 예

```
def test_kwargs(arg, **kwargs):  
    print("arg :", arg)  
    for key in kwargs:  
        print(kwargs[key])
```

```
>>> test_kwargs(10, a="one", b="two", c="three")  
arg : 10  
one  
three  
two
```

5. 함수의 인수

◆ 가변 인수 리스트와 정의되지 않은 키워드 인수

- 가변 인수 리스트는 튜플로 처리된다.
- 정의되지 않은 키워드 인수는 사전으로 처리된다.

```
>>> def foo(a, b, *args, **kw):  
    print(a,b)  
    print(args)  
    print(kw)
```

```
>>> foo(1,2,3,4,c=5,d=10)  
1 2  
(3, 4)  
{ 'c': 5, 'd': 10}
```

6. 전역 변수와 지역 변수

- ◆ 전역 변수 : 프로그램 전체에서 사용 가능
- ◆ 지역 변수 : 함수 내에서만 사용 가능

```
>>> a = 10
>>> def foo():
>>>     print(a)

>>> foo()
10
>>> print(a)
10
```

전역 변수만 있음

```
>>> def foo():
>>>     a = 100
>>>     print(a)

>>> foo()
100
>>> print(a)
Traceback (most recent call last):
  File "<pyshell#94>", line 1, in <module>
    print(a)
NameError: name 'a' is not defined
```

지역 변수만 있음

6. 전역 변수와 지역 변수

```
>>> a = 100
>>> def foo():
    a = 200
    print(a)

>>> print(a)
100
>>> foo()
200
>>> print(a)
100
```

이름은 같지만 **전역 변수 a**와 **지역 변수 a**는 다른 변수임.

```
>>> a = 100
>>> def foo():
    print(a)
    a = 200
    print(a)

>>> print(a)
100
>>> foo()
.....
UnboundLocalError: local variable 'a'
referenced before assignment
>>>
```

a가 함수 foo 내에 지역 변수로 있기 때문에 전역 변수를 참조하지 않는다. 하지만 지역 변수가 사용하려고 할 때, 만들어지지 않았기 때문에 에러이다.

6. 전역 변수와 지역 변수

◆ global 키워드

```
>>> a = 100
>>> def foo():
    global a
    print(a)
    a = 200
    print(a)
```

```
>>> print(a)
100
>>> foo()
100
200
>>> print(a)
200
```

```
>>> def foo():
    global a
    a = 50
    print(a)

>>> print(a)
Traceback (most recent call last):
  File "<pyshell#142>", line 1, in <module>
    print(a)
NameError: name 'a' is not defined

>>> foo()
50
>>> print(a)
50
```


6. 전역 변수와 지역 변수

```
def foo():  
    global s  
    print('2 :', s)  
    s = "I learn Java"  
    print('3 :', s)
```

```
# main  
s = "I learn Python"  
print('1 :', s)  
foo()  
print('4 :', s)
```

```
1 : I learn Python  
2 : I learn Python  
3 : I learn Java  
4 : I learn Java
```

```
def foo(x,y):  
    global a  
    a = 10  
    x,y = y,x  
    b = 20  
    c = 30  
    print(a,b,x,y)
```

```
# main
```

```
a,b,x,y = 11,22,33,44  
foo(100,200)  
print(a,b,x,y)
```


7. 람다 함수 (lambda)

◆ 람다 함수의 정의

- 이름없는 한 줄짜리 함수.
- 람다 함수는 return 문을 사용하지 않는다.
- 람다 함수의 몸체는 문이 아닌 하나의 식이다.

lambda <인수들> : <반환할 식>


lambda : 1



입력 인수는 없음.

항상 1을 반환.

lambda x, y : x + y



함수 인수 반환할 값

7. 람다 함수 (lambda)

◆ 람다 함수 사용하기

```
>>> def add(x,y):  
    return x+y  
  
>>> add(2,3)  
5  
>>> add('hello', 'world')  
'helloworld'
```



add : 함수명



```
>>> add = lambda x, y: x + y  
>>> add(2,3)  
5  
>>> add('hello', 'world')  
'helloworld'
```

```
>>> import math  
>>> square_root = lambda x: math.sqrt(x)  
>>> square_root(100)  
10.0
```

7. 람다 함수 (lambda)

◆ 람다 함수는 함수를 인자로 넘길 때 유용하다

```
>>> def f1(x):  
    return x*x + 3*x -10  
>>> def f2(x):  
    return x * x * x  
>>> def g(func):  
    return [func(x) for x in range(-5, 5)]  
>>> g(f1)  
[0, -6, -10, -12, -12, -10, -6, 0, 8, 18]  
>>> g(f2)  
[-125, -64, -27, -8, -1, 0, 1, 8, 27, 64]
```

```
>>> def g(func):  
    return [func(x) for x in range(-5, 5)]
```

```
>>> g(lambda x:x * x + 3 * x - 10)  
[0, -6, -10, -12, -12, -10, -6, 0, 8, 18]  
>>> g(lambda x:x * x * x)  
[-125, -64, -27, -8, -1, 0, 1, 8, 27, 64]
```

위의 코드를 람다함수
로 바꾼 예

7. 람다 함수 (lambda)

◆ 람다 함수에서 기본 인자, 키워드 인자, 가변 인자 사용하기

```
>>> incr = lambda x, inc = 1: x + inc
>>> incr(10)
11
>>> incr(10, 5)
15
>>> vargs = lambda x, *args: args
>>> vargs(1, 2, 3, 4, 5)
(2, 3, 4, 5)
>>>
>>> kwords = lambda x, *args, **kw: kw
>>> kwords(1, 2, 3, a=4, b=6)
{'b': 6, 'a': 4}
```

7. 람다 함수 (lambda)

◆ map 내장 함수

- 입력 집합(X)과 사상 함수(f)가 주어져 있을 때, $Y = f(X)$ 를 구한다.
- 두 개 이상의 인수를 받는다.
- 첫 인수는 함수(f)이며 두 번째부터는 입력 집합(X)인 시퀀스 자료형 (문자열, 리스트, 튜플 등)이어야 한다.
- 첫 번째 인수인 함수는 입력 집합 수만큼의 인수를 받는다.

7. 람다 함수 (lambda)

◆ map 내장 함수

```
>>> def f(x):  
    return x * x  
  
>>> X = [1, 2, 3, 4, 5]  
>>> list(map(f, X))  
[1, 4, 9, 16, 25]  
>>> names = ['Alice', 'Paul', 'Bob', 'Robert']  
>>> length = map(len, names)  
>>> list(length)  
[5, 4, 3, 6]
```

7. 람다 함수 (lambda)

◆ map 내장 함수

```
>>> X = [1, 2, 3, 4, 5]
>>> Y = map(lambda a:a * a, X)
>>> list(Y)
[1, 4, 9, 16, 25]

>>> X = range(10)
>>> Y = map(lambda x: x * x + 4 * x + 5, X)
>>> list(Y)
[5, 10, 17, 26, 37, 50, 65, 82, 101, 122]

>>> X = [1, 2, 3, 4, 5]
>>> Y = [6, 7, 8, 9, 10]
>>> Z = map(lambda x, y:x + y, X, Y) # 인자가 2개
>>> list(Z)
[7, 9, 11, 13, 15]
```


7. 람다 함수 (lambda)

◆ filter 내장 함수

- 주어진 시퀀스 데이터 중에서 필터링하여 참인 요소만 모아 출력.
- 두 개의 인수를 가지며 첫 인수는 (map() 함수와 같이) 함수이고, 두 번째 인수는 시퀀스 자료형이다.

```
>>> X = [1,3,5,7,9]
>>> result = filter(lambda x:x>5, X)
>>> list(result)
[7, 9]
>>> list(filter(lambda x:x%2==1, range(11)))
[1, 3, 5, 7, 9]
>>> dash = '-'
>>> dash.join(filter(lambda x:x<'a', 'abcABCdefDEF'))
'A-B-C-D-E-F'
```