

함수로 코드 간추리기

이 단원을 시작하기에 앞서 알아둬야 하는 용어 - 정의

❖ 정의

- 정의(Definition)란, 어떤 이름을 가진 코드가 구체적으로 어떻게 동작하는지를 "구체적으로 기술"하는 것
- 파이썬에서는 함수나 메소드를 정의할 때 `definition(정의)`를 줄인 키워드인 `def`를 사용

❖ 실습 1 (def 키워드를 이용한 함수 정의)

```
>>> def hello():  
    print("hello world!")
```

```
>>> hello()  
hello world!
```

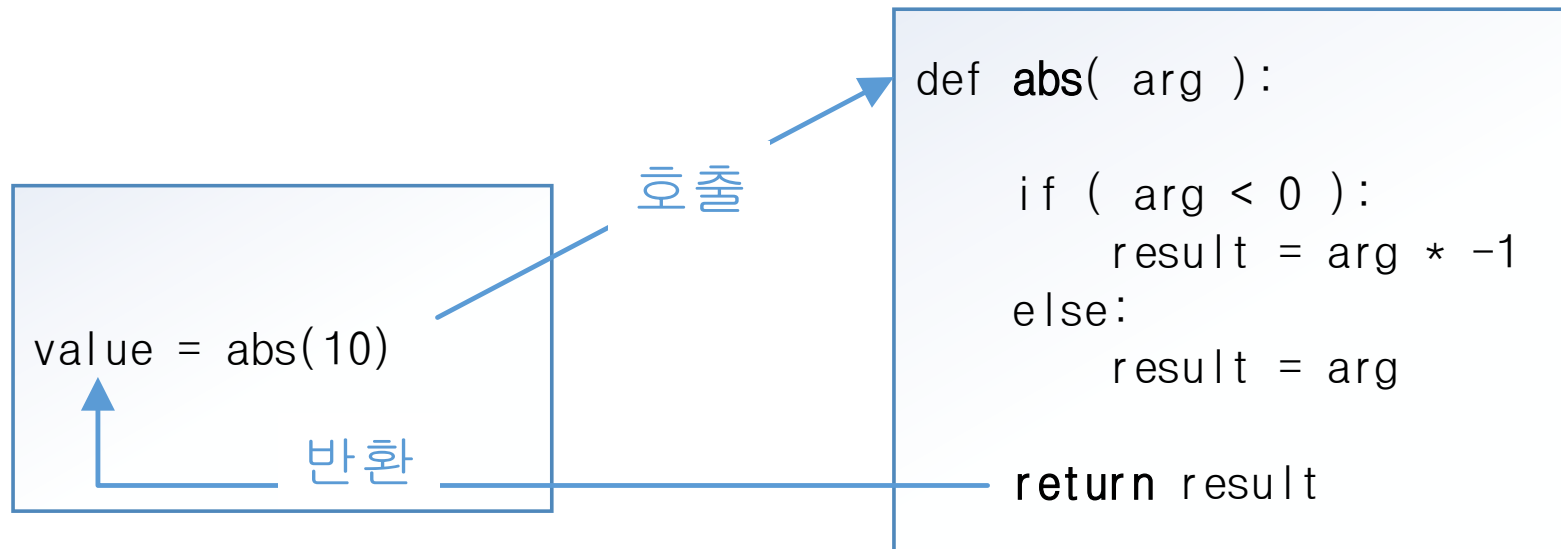
이 단원을 시작하기에 앞서 알아둬야 하는 용어 - 호출과 반환

❖ 호출(Call)

- 모든 함수는 이름을 갖고 있으며, 이 이름을 불러주면 파이썬은 그 이름 아래 정의되어 있는 코드를 실행.

❖ 반환(Return)

- 함수가 자신의 코드를 실행하고 나면 결과가 나오는데, 그 결과를 자신의 이름을 부른 코드에게 돌려줌.



함수 정의하기

❖ 함수는 def 키워드를 이용해서 코드블록에 이름을 붙인 형태

```
def 함수이름( 매개변수 목록 ):  
    # 코드블록  
    return 결과
```


❖ 실습 1 (함수 정의)

```
>>> def my_abs( arg ):  
    if ( arg < 0 ):  
        result = arg * -1  
    else:  
        result = arg  
  
    return result
```

매개변수를 입력 받는 여러 가지 방법

- ❖ 매개(媒介)는 중간에서 둘 사이의 관계를 맺어주는 것을 뜻하는 말
- ❖ 매개변수는 호출자와 함수 사이의 관계를 맺어주는 변수를 뜻함

```
def my_abs(arg):  
    if ( arg < 0 ):  
        result = arg * -1  
    else:  
        result = arg  
  
    return result
```



매개변수

- ❖ 실습 1 (잘못된 매개변수를 이용하여 함수를 호출하는 경우)

```
>>> my_abs()  
Traceback (most recent call last):  
  File "<pyshell#3>", line 1, in <module>  
    my_abs()  
TypeError: my_abs() missing 1 required positional argument: 'arg'
```

매개변수를 입력 받는 여러 가지 방법

❖ 매개변수의 이름은 보통의 변수처럼 문자와 숫자, 그리고 _ 로 만들어짐

- 숫자로 매개변수의 이름을 시작할 수는 없음.
- 변수의 역할과 의미가 잘 나타나는 이름을 붙일 것.

```
def print_name1( 123abc ) :  
    ...
```

사용불가. 123abc는 숫자가 앞에 오므로 사용할 수 없는 이름입니다.

```
def print_name2( aaa, bbb ) :  
    ...
```

나쁨. aaa, bbb를 보서는 변수의 역할을 유추할 수 없습니다.

```
def print_name3( first_name, last_name ) :  
    ...
```

좋음. 의미를 분명하게 전달하는 이름입니다.

매개변수를 입력 받는 여러 가지 방법

❖ 실습 2 (입력받은 매개변수에 따라 문자열을 반복출력)

```
>>> def print_string(text, count):  
    for i in range(count):  
        print(text)
```

```
>>> print_string('안녕하세요', 3)  
안녕하세요  
안녕하세요  
안녕하세요
```

매개변수를 입력 받는 여러 가지 방법 - 기본값 매개변수와 키워드 매개변수

❖ 기본값 매개변수(Default Argument Value)

- “이 매개변수를 입력할지 말지는 호출자 당신의 자유야. 단, 입력하지 않으면 내가 갖고 있는 기본값으로 할당할 거야.”

❖ 실습 1 (기본값 매개변수 정의와 사용)

```
>>> def print_string(text, count=1):  
    for i in range(count):  
        print(text)
```

매개변수를 정의할 때 값을 할당해놓으면 기본값 매개변수가 됩니다.

```
>>> print_string('안녕하세요')  
안녕하세요  
>>> print_string('안녕하세요', 2)  
안녕하세요  
안녕하세요
```

호출할 때 두 번째 매개변수를 생략하면 기본값 1이 사용됩니다.

매개변수를 입력 받는 여러 가지 방법

- 기본값 매개변수와 키워드 매개변수

❖ 키워드 매개변수(Keyword Argument)

- 매개변수가 많은 경우에는 호출자가 매개변수의 이름을 일일이 지정하여 데이터를 입력

매개변수를 입력 받는 여러 가지 방법

- 기본값 매개변수와 키워드 매개변수

❖ 실습 1 (기본값 매개변수 정의와 사용)

```
>>> def print_personnel(name, position='staff', nationality='Korea'):
    print('name = {0}'.format(name))
    print('position = {0}'.format(position))
    print('nationality = {0}'.format(nationality))
```

```
>>> print_personnel(name='박상현')
name = 박상현
position = staff
nationality = Korea
```

position과 nationality는 기본값이 사용됩니다.

```
>>> print_personnel(name='박상현', nationality='ROK')
name = 박상현
position = staff
nationality = ROK
```

position만이 기본값을 사용합니다.

```
>>> print_personnel(name='박상현', position='인턴')
name = 박상현
position = 인턴
nationality = Korea
```

nationality만이 기본값을 사용합니다.

매개변수를 입력 받는 여러 가지 방법 - 가변매개변수

❖ 가변 매개변수(Arbitrary Argument List)

- 입력 개수가 달라질 수 있는 매개변수
- *를 이용하여 정의된 가변 매개변수는 튜플

```
def 함수이름(*매개변수):  
    코드블록
```

매개변수 앞에 *를 붙이면 해당매개변수는 가변으로 지정됩니다.

❖ 실습 1 (가변 매개변수)

```
>>> def merge_string(*text_list):  
    result = ''  
    for s in text_list:  
        result = result + s  
    return result
```

```
>>> merge_string('아버지가', '방에', '들어가신다.')  
'아버지가방에들어가신다.'
```

매개변수를 입력 받는 여러 가지 방법 - 가변매개변수

❖ 실습 2 (딕셔너리 형식 가변 매개변수)

```
>>> def print_team(**players):  
    for k in players.keys():  
        print('{0} = {1}'.format(k, players[k]))
```

매개변수 앞에 **를 붙이면 딕셔너리 가변 매개변수가 됩니다.

```
>>> print_team(카시야스='GK', 호날두='FW', 알론소='MF', 페페='DF')  
카시야스 = GK  
페페 = DF  
알론소 = MF  
호날두 = FW
```

매개변수를 입력 받는 여러 가지 방법 - 가변매개변수

❖ 실습 3 (일반 매개변수와 함께 사용하는 가변매개변수)

```
>>> def print_args(argc, *argv):  
    for i in range(argc):  
        print(argv[i])
```

```
>>> print_args(3, "argv1", "argv2", "argv3")  
argv1  
argv2  
argv3
```

```
>>> print_args(argc=3, "argv1", "argv2", "argv3")  
SyntaxError: non-keyword arg after keyword arg
```

가변 매개변수 앞에 정의된
일반 매개변수는 키워드 매개
변수로 호출할 수 없습니다.

매개변수를 입력 받는 여러 가지 방법 - 가변매개변수

❖ 실습 4 (가변 매개변수와 함께 사용하는 일반 매개변수)

```
>>> def print_args(*argv, argc):  
    for i in range(argc):  
        print(argv[i])
```

```
>>> print_args("argv1", "argv2", "argv3", argc=3)  
argv1  
argv2  
argv3
```

가변 매개변수 뒤에 정의된 일반 매개변수는 반드시 키워드 매개변수로 호출해야 합니다.

```
>>> print_args("argv1", "argv2", "argv3", 3)  
Traceback (most recent call last):  
  File "<pyshell#15>", line 1, in <module>  
    print_args("argv1", "argv2", "argv3", 3)  
TypeError: print_args() missing 1 required keyword-only argument:  
'argc'
```

매개변수를 입력 받는 여러 가지 방법 - 가변매개변수

❖ 함수가 호출자에게 값을 반환할 때에는 return문을 이용

- return문을 이용하는 세 가지 방법
- return문에 결과 데이터를 담아 실행하기 → 함수가 즉시 종료되고 호출자에게 결과가 전달됨.
- return문에 아무 결과도 넣지 않고 실행하기 → 함수가 즉시 종료됨.
- return문 생략하기 → 함수의 모든 코드가 실행되면 종료됨.

❖ 실습 1

```
>>> def multiply(a, b):  
    return a*b  
  
>>> result = multiply(2, 3)  
>>> result  
6
```

return문은 함수의 실행을 종료시키고 자신에게 넘겨진 데이터를 호출자에게 전달합니다.

호출자에게 반환하기

❖ 실습 2 (여러 개의 return)

```
>>> def my_abs(arg):  
    if arg < 0:  
        return arg * -1  
    else:  
        return arg  
  
>>> result = my_abs(-1)  
>>> result  
1  
>>> result = my_abs(1)  
>>> result  
1
```


호출자에게 반환하기

❖ 실습 3 (None을 반환하는 경우)

```
>>> def my_abs(arg):  
    if arg < 0:  
        return arg * -1  
    elif arg > 0:  
        return arg
```

```
>>> result = my_abs(-1)
```

```
>>> result
```

```
1
```

```
>>> result = my_abs(1)
```

```
>>> result
```

```
1
```

```
>>> result = my_abs(0)
```

```
>>> result
```

```
>>> result == None
```

```
True
```

```
>>> type(result)
```

```
<class 'NoneType'>
```

return을 실행하지 못하고 함수가 종료되면 함수는 호출자에게 None을 반환합니다.

호출자에게 반환하기

❖ 실습 4 (결과 없는 return)

```
>>> def ogamdo(num):  
    for i in range(1, num+1):  
        print('제 {0}의 아해'.format(i))  
        if i == 5:  
            return
```

```
>>> ogamdo(3)
```

제 1의 아해

제 2의 아해

제 3의 아해

```
>>> ogamdo(5)
```

제 1의 아해

제 2의 아해

제 3의 아해

제 4의 아해

제 5의 아해

```
>>> ogamdo(8)
```

제 1의 아해

제 2의 아해

제 3의 아해

제 4의 아해

제 5의 아해

반환할 데이터 없이 실행하는 return문은 "반환"의 의미보다는 "함수 종료"의 의미로 사용됩니다.

8을 입력하면 for 반복문은 8번 반복을 수행하려고 준비하겠지만 실행되는 return문 때문에 다섯 번 수행하면 함수가 종료되고 맙니다.

호출자에게 반환하기

❖ 실습 5 (return없는 함수)

```
>>> def print_something(*args):  
    for s in args:  
        print(s)
```

반환할 결과도 없고 함수를 중간에 종료시킬 일도 없다면 return문은 생략해도 됩니다.

```
>>> print_something(1, 2, 3, 4, 5)  
1  
2  
3  
4  
5
```

함수 밖의 변수, 함수 안의 변수

❖ “함수 밖에서 변수 `a`를 정의하여 0을 대입하고, 함수 안에서 변수 `a`를 또 정의하여 1을 대입했다. 이 함수를 실행(호출)하고 나면 함수 밖에서 정의된 변수 `a`의 값은 얼마일까?”

- 답 : 0
- 함수 밖에 있는 `a`와 안에 있는 `a`는 이름은 같지만 사실은 완전히 별개의 변수

❖ 실습 1

```
>>> def scope_test():  
    a = 1  
    print('a:{0}'.format(a))
```

함수를 정의하는 시점에서는 `a`가 메모리에 생성되지 않습니다. 함수를 호출하면 그제서야 함수의 코드가 실행되면서 `a`가 메모리에 생성됩니다.

함수 밖에서 `a`를 정의하고 0으로 초기화 합니다.

```
>>> a = 0  
>>> scope_test()  
a:1  
>>> print('a:{0}'.format(a))  
a:0
```

`scope_test()`가 호출되면 함수 내부에서 `a`를 정의하고 1로 초기화합니다.

하지만 함수 밖에서 정의한 `a`를 출력해보면 여전히 0을 갖고 있음을 확인할 수 있습니다.

함수 밖의 변수, 함수 안의 변수

- ❖ 변수는 자신이 생성된 범위(코드블록) 안에서만 유효
- ❖ 함수 안에서 만든 변수는 함수 안에서만 살아있다가 함수 코드의 실행이 종료되면 그 생명이 다함 → 이것을 지역변수(Local Variable)라고 함
- ❖ 이와는 반대로 함수 외부에서 만든 변수는 프로그램이 살아있는 동안에는 함께 살아있다가 프로그램이 종료될 때 같이 소멸됨.
→ 이렇게 프로그램 전체를 유효 범위로 가지는 변수를 전역 변수(Global Variable) 라 함.
- ❖ 파이썬은 함수 안에서 사용되는 모든 변수를 지역변수로 간주
- ❖ 전역 변수를 사용하기 위해서는 global 키워드를 이용

함수 밖의 변수, 함수 안의 변수

❖ 실습 2

```
>>> def scope_test():  
    global a  
    a = 1  
    print('a:{0}'.format(a))
```

global 키워드는 지정한 변수의 유효범위가 전역임을 알리고, 지역변수의 생성을 막습니다.
이 a는 scope_test() 함수 안에서 전역 변수로 사용됩니다.

```
>>> a = 0  
>>> scope_test()  
a:1  
>>> print('a:{0}'.format(a))  
a:1
```

scope_test()는 0으로 초기화 되어 있는 a에 접근하여 1로 값을 변경합니다.

a를 출력해보면 scope_test() 함수 안에서 변경한 값 1이 들어있음을 확인할 수 있습니다.

자기 스스로를 호출하는 함수 : 재귀함수

- ❖ 재귀함수(Recursive Function)는 자기 스스로를 호출하는 함수
- ❖ 함수가 자기 자신을 부르는 것을 재귀호출(Recursive Call)이라 함.

❖ 재귀 함수의 예

```
def some_func(count):  
    if count > 0:  
        some_func(count-1)  
    print(count)
```

함수 밖의 변수, 함수 안의 변수

❖ 실습 1 (팩토리얼을 재귀 함수로 구현)

- 다음 재귀 관계식(Recurrence relation)을 파이썬 코드로 옮기는 예제
- $$n! = \begin{cases} 1, & n = 0 \\ (n-1)! \times n, & n > 0 \end{cases}$$

```
>>> def factorial(n):  
    if n == 0:  
        return 1  
    elif n > 0:  
        return factorial(n-1)*n
```

```
>>> factorial(5)
```

```
120
```

```
>>> factorial(10)
```

```
3628800
```

```
>>> factorial(100)
```

```
93326215443944152681699238856266700490715968264381621468592963895217599
```

```
99322991560894146397615651828625369792082722375825118521091686400000000
```

```
0000000000000000
```


함수 밖의 변수, 함수 안의 변수

- ❖ 재귀 호출의 단계가 깊어질 수록 메모리를 추가적으로 사용하기 때문에 재귀 함수가 종료될 조건을 분명하게 만들어야 함.
- ❖ 실습 2 (재귀함수를 사용할 때 주의할 점)

```
>>> def no_idea():  
    print("나는 아무 생각이 없다.")  
    print("왜냐하면")  
    no_idea()
```

종료할 조건도 지정해주지 않은 채 무조건 재귀호출을 수행하면 스택 오버플로우가 발생합니다.

```
>>> no_idea()  
나는 아무 생각이 없다.  
왜냐하면  
나는 아무 생각이 없다.  
왜냐하면...
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#10>", line 1, in <module>  
    no_idea()  
  File "<pyshell#8>", line 4, in no_idea  
    no_idea()
```

```
...
```

```
File "<pyshell#8>", line 2, in no_idea  
    print("나는 아무 생각이 없다.")
```

```
File "C:\Python34\lib\idlelib\PyShell.py", line 1342, in write  
    return self.shell.write(s, self.tags)
```

스택 오버 플로우가 발생하면 파이썬에서 지정해놓은 최대 재귀 단계를 초과했다는 에러가 출력됩니다.

함수를 변수에 담아 사용하기

❖ 실습 1

```
>>> def print_something(a):  
    print(a)
```

() 없이 함수의 이름만을 변수에 저장합니다.

```
>>> p = print_something
```

```
>>> p(123)
```

```
123
```

```
>>> p('abc')
```

```
abc
```

변수의 이름 뒤에 ()를 붙여 함수처럼 호출하면 됩니다.

함수를 변수에 담아 사용하기

❖ 실습 2

```
>>> def plus(a, b):  
    return a+b
```

```
>>> def minus(a, b):  
    return a-b
```

```
>>> flist = [plus, minus]
```

```
>>> flist[0](1, 2)  
3
```

```
>>> flist[1](1, 2)  
-1
```

plus() 함수와 minus() 함수를 리스트의 요소로 집어넣습니다.

flist[0]은 plus() 함수를 담고 있습니다. 따라서 이 요소 뒤에 괄호를 열고 매개변수를 입력하여 호출하면 plus() 함수가 호출됩니다.

flist[1]는 minus()를 담고 있으므로 이 코드는 minus(1, 2)와 같습니다.

함수를 변수에 담아 사용하기

❖ 함수를 변수에 담을 수 있는 이유?

- 파이썬이 함수를 일급 객체(First Class Object)로 다루고 있기 때문
- 일급 객체란 프로그래밍 언어 설계에서 매개변수로 넘길 수 있고 함수가 반환할 수도 있으며 변수에 할당이 가능한 개체를 가리키는 용어
- 파이썬에서는 함수를 "매개변수"로도 사용할 수 있고 함수의 결과로 "반환"하는 것도 가능

❖ 실습 3(함수를 매개변수로 사용하기)

```
>>> def hello_korean():  
    print('안녕하세요.')
```

```
>>> def hello_english():  
    print('Hello.')
```

```
>>> def greet(hello):  
    hello()
```

```
>>> greet(hello_korean)  
안녕하세요.  
>>> greet(hello_english)  
Hello.
```

함수를 변수에 담아 사용하기

❖ 실습 4(함수를 결과로써 반환하기)

```
>>> def hello_korean():  
    print('안녕하세요.')
```



```
>>> def hello_english():  
    print('Hello.')
```



```
>>> def get_greeting(where):  
    if where == 'K':  
        return hello_korean  
    else:  
        return hello_english
```



```
>>> hello = get_greeting('K')  
>>> hello()  
안녕하세요.
```



```
>>> hello = get_greeting('E')  
>>> hello()  
Hello.
```

매개변수 where가 'K'인 경우 hello_korean() 함수를 반환합니다.

그 외의 경우 hello_english() 함수를 반환합니다.

get_greeting() 함수가 반환하는 결과를 변수 hello 에 담아 "호출"합니다.

함수 안의 함수 : 중첩 함수

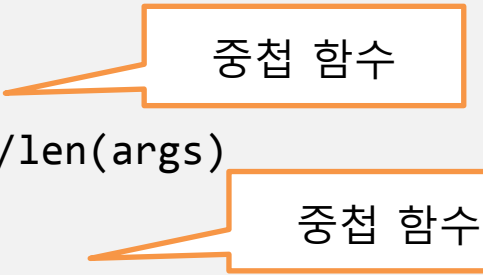
❖ 중첩 함수(Nested Function) : 함수 안에 정의된 함수

- 중첩 함수는 자신이 소속되어 있는 함수의 매개변수에 접근할 수 있음.

❖ 실습 1

```
>>> import math
>>> def stddev(*args):
    def mean():
        return sum(args)/len(args)
    def variance(m):
        total = 0
        for arg in args:
            total += (arg - m ) ** 2
        return total/(len(args)-1)
    v = variance(mean())
    return math.sqrt(v)

>>> stddev(2.3, 1.7, 1.4, 0.7, 1.9)
0.6
```



함수 안의 함수 : 중첩 함수

- ❖ 중첩함수의 자신이 소속되어 있는 함수 외부에서는 보이지 않음.
- ❖ 실습 2

```
>>> mean()  
Traceback (most recent call last):  
  File "<pyshell#2>", line 1, in <module>  
    mean()  
NameError: name 'mean' is not defined
```

pass : 구현을 잠시 미뤄두셔도 좋습니다.

❖ **pass 키워드는 함수나 클래스의 구현을 미룰 때 사용**

```
def empty_function()  
    pass
```

```
class empty_class:  
    pass
```