Python (7)

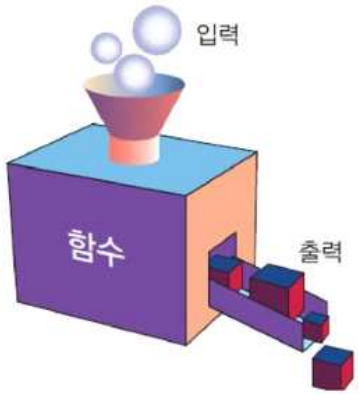
함수 (Function)

04-1 함수(Function)

○ 함수란 무엇인가?

함수를 설명하기 전에 아래와 같은 제조기를 생각해 보자.

- 제조기에 재료를 넣는다.
- 그리고 제조기를 작동한다.
- 그러면, 재료에 따라 제품이 나온다



여기서 우리가 제조기에 넣는 재료는 '입력값', 결과로 생성된 제품은 '출력(결과값)'이 된다.
이와 같이 제조기는 바로 함수라고 할 수 있다.
입력값을 가지고 어떤 일을 수행한 후 그 결과값을 내어 놓는 것이 바로 함수가 하는 일이다.

함수는 모든 프로그래밍에서 매우 중요하다.
모든 프로그램은 함수의 집합이라해도 과언이 아니다.
따라서, 기본 개념부터 집중하여 차근히 학습해 나가야 한다.

○ 함수를 사용하는 이유

프로그래밍을 하다 보면 똑같은 내용을 반복해서 작성하게되는 경우가 많다.
이때가 바로 함수가 필요한 때이다.

즉, 코딩에서 반복되는 부분이 있을 경우, '반복적으로 사용되는 가치 있는 부분'을 한 코드 블록으로 묶어,
'어떤 입력값을 주었을 때 어떤 결과값을 리턴해 준다'라는 형식의 함수를 작성한다.

함수를 사용하는 또 다른 이유는,
코딩 시 프로그램을 기능 단위의 함수로 분리해 놓으면 전체 프로그램 흐름을 일목요연하게 볼 수 있다.

마치 공장에서 원재료가 여러 공정을 거쳐 하나의 완제품이 되는 것처럼,
프로그램에서도 입력한 값이 여러 함수를 거치면서 원하는 결과값을 생성하게 된다.

이렇게 되면 프로그램 흐름도 잘 파악할 수 있고, 오류가 어디에서 나는지도 쉽게 파악할 수 있다.

○ 파이썬 함수의 구조

파이썬 함수의 구조는 다음과 같다.

```
def 함수_이름(매개변수):     def = define
    수행할_문장
    수행할_문장
    ...
```

def는 함수를 만들 때 사용하는 예약어이며, 함수 이름은 함수를 만드는 사람이 임의로 만들 수 있다.
함수 이름 뒤 괄호 안의 매개변수는 이 함수에 입력으로 전달되는 값을 받는 변수이다.
이렇게 함수를 정의한 후 **if**, **while**, **for** 문 등과 마찬가지로 코드 블록에 함수에서 수행할 문장을 입력한다.

간단하지만 많은 것을 설명해 주는 다음 예를 살펴보자.

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

```
    return a + b
```

위 함수는 다음과 같이 해석된다.

    이 함수의 이름은 add이고,
    입력으로 2개의 값을 받으며,
    2개의 입력값을 더한 값을 반환(return)한다.

이제 직접 add 함수를 사용해 보자.

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

a = 3
b = 4
c = add(a, b)          # add(3, 4)의 리턴값을 c에 대입
print(c)
--------------------
7
```

변수 a에 3, b에 4를 대입한 후 앞에서 만든 add 함수에 a와 b를 입력값으로 넣어 준다.
그리고 변수 c에 add 함수의 리턴값을 대입하면 print(c)로 c의 값을 확인할 수 있다.

여기서 한가지 주의할 것은 함수를 사용하기 전에 반드시 함수를 먼저 정의(선언)해야 한다는 것이다.
그렇지 않으면 오류가 발생한다.

○ 매개변수와 인수

매개변수(parameter)와 인수(arguments)는 혼용해서 사용하는 용어이므로 잘 기억해 두자.
매개변수는 함수에 입력으로 전달된 값을 받는 변수, 인수는 함수를 호출할 때 전달하는 입력값을 의미한다.

```
def add(a, b):  # a, b는 매개변수
    return a+b

print(add(3, 4))  # 3, 4는 인수
```

○ 입력값과 리턴값에 따른 함수의 형태

함수는 들어온 입력값을 받은 후 어떤 처리를 하여 적절한 값을 리턴해 준다.

함수의 형태는 입력값과 리턴값의 존재 유무에 따라 4가지 유형으로 나뉜다. 자세히 알아보자.

1. 일반적인 함수

입력값이 있고 리턴값이 있는 함수가 일반적인 함수이다.
앞으로 프로그래밍을 할 때 만들 함수는 대부분 다음과 비슷한 형태일 것이다.

```
def 함수_이름(매개변수):
    수행할_문장
    ...
    return 리턴값
```

다음은 일반적인 함수의 전형적인 예이다.

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

add 함수는 2개의 입력값을 받아 서로 더한 결과값을 리턴한다.

이 함수를 사용하는 방법은 다음과 같다. 입력값으로 3과 4를 주고 리턴값을 받아 보자.

```
a = add(3, 4)
print(a)
--------------------
7
```

이처럼 입력값과 리턴값이 있는 함수의 사용법을 정리하면 다음과 같다.

```
리턴값을_받을_변수 = 함수_이름(입력_인수1, 입력_인수2, ...)
```

2. 입력값이 없는 함수

입력값이 없는 함수가 존재할까? 당연히 존재한다. 다음과 같이 작성해 보자.

```
def say():
    return 'Hi'
```

say라는 이름의 함수를 만들었다.
그런데 매개변수 부분을 나타내는 함수 이름 뒤의 괄호 안이 비어 있다.

다음을 직접 입력해 보자.

```
a = say()
print(a)
--------------------
Hi
```

위 함수를 쓰기 위해서는 say()처럼 괄호 안에 아무런 값도 넣지 않아야 한다.
이 함수는 입력값은 없지만, 리턴값으로 "Hi"라는 문자열을 리턴한다.
즉, a = say()처럼 작성하면 a에 "Hi"라는 문자열이 대입되는 것이다.

이처럼 입력값이 없고 리턴값만 있는 함수는 다음과 같이 사용한다.

```
리턴값을_받을_변수 = 함수_이름()
```

3. 리턴값이 없는 함수

리턴값이 없는 함수 역시 존재한다. 다음 예를 살펴보자.

```
def add(a, b):
    print("%d, %d의 합은 %d입니다." % (a, b, a+b))
```

리턴값이 없는 함수는 호출해도 리턴되는 값이 없기 때문에 다음과 같이 사용한다.

```
add(3, 4)
--------------------
3, 4의 합은 7입니다.
```

즉, 리턴값이 없는 함수는 다음과 같이 사용한다.

```
함수_이름(입력_인수1, 입력_인수2, ...)
```

여기서 ‘3, 4의 합은 7입니다.’라는 문장을 출력했는데 왜 리턴값이 없다는 것인지 의아하게 생각할 수도 있다.
이 부분을 초보자들이 혼란스러워하는데, print 문은 함수의 구성 요소 중 하나인 ‘수행할_문장’에 해당하는 부분일 뿐이다.
리턴값은 당연히 없다.
리턴값은 오직 return 명령어로만 돌려받을 수 있다.

이를 확인해 보자. 리턴받을 값을 a 변수에 대입하고 a 값을 출력해 보면 리턴값이 있는지, 없는지 알 수 있다.

```
a = add(3, 4)
print(a)
--------------------
3, 4의 합은 7입니다.
None
```

위 예에서 a = add(3, 4)를 실행하는 순간, 바로 '3, 4의 합은 7입니다.'가 먼저 출력된다.
그리고, 변수 a에 어떤 값이 리턴된다.
변수 a를 print 해보면, None이 출력된다.
위의 add 함수처럼 리턴값이 없을 때 a = add(3, 4)처럼 쓰면 함수 add는 리턴값으로 a 변수에 None을 리턴한다.
None이란 ‘값이 없는 자료형’이다.

4. 입력값도, 리턴값도 없는 함수

입력값도, 리턴값도 없는 함수 역시 존재한다. 다음 예를 살펴보자.

```
def say():
    print('Hi')
```

입력 인수를 받는 매개변수도 없고 return 문도 없으니 입력값도, 리턴값도 없는 함수이다.

이 함수를 사용하는 방법은 단 1가지이다.

```
say()
--------------------
Hi
```

즉, 입력값도, 리턴값도 없는 함수는 다음과 같이 사용한다.

```
함수_이름()
```

○ 매개변수를 지정하여 함수 호출하기

함수 실행을 하기 위해 인수를 지정할 때에는, 인수를 지정하는 순서가 매우 중요하다.
다시 말하면, 인수의 순서대로 함수의 매개 변수에 할당 된다.

다음 예를 살펴보자.

```
def sub(a, b):
    return a - b

result = sub(6, 3)
print(result)
--------------------
3
```

위 예에서 인수 6은 매개변수 a에, 인수 3은 매개변수 b에 순서대로 각각 할당 된다.
만약, 아래와 같이 함수를 호출하면, 결과는 달라진다.

```
result = sub(3, 6)
print(result)
--------------------
-3
```

위 예에서 인수 6은 매개변수 a에, 인수 6은 매개변수 b에 순서대로 각각 할당 되어 결과는 -3이 리턴된다.
이와 같이 함수를 호출할 경우에는 인수의 순서가 매우 중요하다.

그런데, 함수를 호출할 때 다음과 같이 함수의 각 매개변수에 대해 인수를 지정하여 사용할 수도 있다.

```
result = sub(a=7, b=3)       # a에 7, b에 3을 전달
print(result)
--------------------
4
```

매개변수를 지정하면 다음과 같이 순서에 상관없이 사용할 수 있다는 장점이 있다.

```
result = sub(b=5, a=3)       # b에 5, a에 3을 전달
print(result)
--------------------
-2
```

그러나, 위 방법은 다른 사람이 코드를 읽을 때 혼동을 줄 가능성이 크기 때문에, 좋은 방법이 아니다.
가급적 사용하지 않는 것이 좋다.

○ 함수의 매개변수가 몇 개가 될지 모를 경우

매개변수가 여러 개일 때 그 입력값을 모두 더해 주는 함수를 생각해 보자.
하지만 몇 개가 입력될지 모를 때는 어떻게 해야 할까?
파이썬은 이런 문제를 해결하기 위해 다음과 같은 방법을 제공한다.

```
def 함수_이름(*매개변수):
    수행할_문장
    ...
```

일반적으로 볼 수 있는 함수 형태에서 괄호 안의 매개변수 부분이 '*매개변수'로 바뀌었다.

◆ 여러 개의 입력값을 받는 함수 만들기

다음 예를 통해 여러 개의 입력값을 모두 더하는 함수를 작성해보자.

예를 들어 add_many(1, 2)이면 3, add_many(1, 2, 3)이면 6, add_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)이면 55를 리턴하는 함수를 만들어 보자.

```
def add_many(*args):
    result = 0
    for i in args:
        result = result + i      # *args에 입력받은 모든 값을 더한다.
    return result
```

위에서 만든 add_many 함수는 입력값이 몇 개이든 상관없다.
*args처럼 매개변수 이름 앞에 *을 붙이면 입력값을 전부 모아 튜플로 만들어 주기 때문이다.
만약 add_many(1, 2, 3)처럼 이 함수를 쓰면 args는 (1, 2, 3)이 되고 add_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)처럼 쓰면 args는 (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)이 된다.
여기에서 *args는 임의로 정한 변수 이름이다. *pey, *python처럼 아무 이름이나 써도 된다.

args는 인수를 뜻하는 영어 단어 arguments의 약자이며 관례적으로 자주 사용한다.

작성한 add_many 함수를 다음과 같이 사용해 보자.

```
result = add_many(1,2,3)
print(result)
```

```
result = add_many(1,2,3,4,5,6,7,8,9,10)
print(result)
--------------------
6
55
```

add_many(1, 2, 3)으로 함수를 호출하면 6을 리턴하고,
add_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)으로 함수를 호출하면 55를 리턴하는 것을 확인할 수 있다.

여러 개의 입력을 처리할 때 def add_many(*args)처럼 함수의 매개변수로 *args 하나만 사용할 수 있는 것은 아니다.
다음 예를 살펴보자.

```
def add_mul(choice, *args):
    if choice == "add":   # 매개변수 choice에 "add"를 입력받았을 때
          result = 0
          for i in args:
                result = result + i
    elif choice == "mul":   # 매개변수 choice에 "mul"을 입력받았을 때
          result = 1
          for i in args:
                result = result * i
```

```
    return result
```

add_mul 함수는 여러 개의 입력값을 의미하는 ***args** 매개변수 앞에 **choice** 매개변수가 추가되어 있다.

이 함수는 다음과 같이 사용할 수 있다.

```
result = add_mul('add', 1,2,3,4,5)
print(result)

result = add_mul('mul', 1,2,3,4,5)
print(result)
--------------------
15
120
```

매개변수 **choice**에 **'add'**가 입력된 경우 ***args**에 입력되는 모든 값을 더해서 **15**를 리턴하고,
'mul'이 입력된 경우 ***args**에 입력되는 모든 값을 곱해 **120**을 리턴한다.

여기서 한가지 중요한 것은, ***args** 매개변수는 다른 매개변수의 뒤에 와야 한다는 것이다.

◆ 키워드 매개변수, kwargs

이번에는 키워드 매개변수에 대해 알아보자.
키워드 매개변수를 사용할 때는 매개변수 앞에 별 2개(**)를 붙인다.

역시 이것도 예제로 알아보자. 먼저 다음과 같은 함수를 작성해 보자.

```
def print_kwargs(**kwargs):
    print(kwargs)
```

print_kwargs는 입력받은 매개변수 kwargs를 출력하는 단순한 함수이다.
이제 이 함수를 다음과 같이 사용해 보자.

def add_mul(choice, * args, **kwargs)
순서를 꼭 지켜야한다

```
print_kwargs(a=1)
print_kwargs(name='foo', age=3)
--------------------
{'a': 1}
{'age': 3, 'name': 'foo'}
```

함수의 입력값으로 a=1이 사용되면 kwargs는 {'a': 1}이라는 딕셔너리가 되고,
입력값으로 name='foo', age=3이 사용되면 kwargs는 {'age': 3, 'name': 'foo'}라는 딕셔너리가 된다.

즉, **kwargs처럼 매개변수 이름 앞에 **을 붙이면 매개변수 kwargs는 딕셔너리가 되고 모든 Key=Value 형태의 입력값이 그 딕셔

너리에 저장된다는 것을 알 수 있다.

kwargs는 **‘keyword arguments’**의 약자이며 **args**와 마찬가지로 관례적으로 사용한다.

여기서 한가지 주의할 것은, **kwargs**는 함수의 매개변수 중 맨 마지막에 정의되어야 한다.

○ 함수의 리턴값은 언제나 하나이다

먼저 다음 함수를 만들어 보자.

```
def add_and_mul(a,b):
    return a+b, a*b
```

add_and_mul은 **2**개의 입력 인수를 받아 더한 값과 곱한 값을 리턴하는 함수이다.

이 함수를 다음과 같이 호출하면 어떻게 될까?

```
result = add_and_mul(3,4)
```

리턴값은 a+b와 a*b인데, 리턴값을 받아들이는 변수는 result 하나만 쓰였으므로 오류가 발생하지 않을까?
당연한 의문이다.
하지만 오류는 발생하지 않는다.
그 이유는 함수의 리턴값은 2개가 아니라 언제나 1개라는 데 있다.
add_and_mul 함수의 리턴값 a+b와 a*b는 튜플값인 (a+b, a*b)로 리턴된다.

따라서 result 변수는 다음과 같은 값을 가지게 된다.

```
result = (7, 12)
```

즉, 리턴값으로 (7, 12)라는 튜플 값을 가지게 되는 것이다.

만약 이 하나의 튜플 값을 2개의 값으로 분리하여 받고 싶다면 함수를 다음과 같이 호출하면 된다.

```
result1, result2 = add_and_mul(3, 4)
```

이렇게 호출하면 result1, result2 = (7, 12)가 되어 result1은 7, result2는 12가 된다.

또 다음과 같은 의문이 생길 수도 있다.

```
def add_and_mul(a,b):
```

```
    return a+b
    return a*b
```

위와 같이 return 문을 2번 사용하면 2개의 리턴값을 돌려 주지 않을까? 하지만 기대하는 결과는 나오지 않는다.

그 이유는 add_and_mul 함수를 호출해 보면 알 수 있다.

```
result = add_and_mul(2, 3)
print(result)
--------------------
5
```

add_and_mul(2, 3)의 리턴값은 5 하나뿐이다.
두 번째 return 문인 return a * b는 실행되지 않았다는 뜻이다.
즉, 함수는 return 문을 만나는 순간, 리턴값을 돌려 준 다음 함수를 빠져나가게 된다.

따라서 이 함수는 다음과 완전히 동일하다.

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

즉 함수는 return문을 만나는 순간 결괏값을 돌려준 다음 함수를 빠져나가게 된다.

◆ return의 또 다른 쓰임새

특별한 상황일 때 함수를 빠져나가고 싶다면 return을 단독으로 써서 함수를 즉시 빠져나갈 수 있다.
다음 예를 살펴보자.

```
def say_nick(nick):
    if nick == "바보":
        return
    print("나의 별명은 %s 입니다." % nick)
```

위는 매개변수 nick으로 별명을 입력받아 출력하는 함수이다.
이 함수 역시 리턴값은 없다.
이때 문자열을 출력한다는 것과 리턴값이 있다는 것은 전혀 다른 말이므로 혼동하지 말자.
함수의 리턴값은 오로지 return 문에 의해서만 생성된다.

만약 입력값으로 '바보'라는 값이 들어오면 문자열을 출력하지 않고 함수를 즉시 빠져나간다.

```
say_nick('야호')
--------------------
```

```
나의 별명은 야호입니다.
```

```
say_nick('바보')
--------------------
```

이처럼 리턴값이 없는 함수에서 **return**으로 함수를 빠져나가는 방법은 실제 프로그래밍에서 자주 사용한다.

○ 매개변수에 초기값 미리 설정하기

이번에는 조금 다른 형태로 함수의 인수를 전달하는 방법에 대해서 알아보자.
다음은 매개변수에 초기값(**Default value**)을 미리 설정해 주는 경우이다.

```
def say_myself(name, age, man=True):
    print("나의 이름은 %s 입니다." % name)
    print("나이는 %d살입니다." % age)
    if man:
        print("남자입니다.")
    else:
        print("여자입니다.")
```

위 함수를 보면 매개변수가 name, age, man=True이다. 그런데 낯선 것이 나왔다. man=True처럼 매개변수에 미리 값을 넣어 준 것이다.
이것이 바로 함수의 매개변수에 초깃값을 설정하는 방법이다.

say_myself 함수는 다음처럼 2가지 방법으로 사용할 수 있다.

```
say_myself("홍길동", 27)
```

```
say_myself("홍길동", 27, True)
```

입력값으로 ("홍길동", 27)처럼 2개를 주면 name에는 "홍길동", age에는 27이 대입된다.
그리고 man이라는 변수에는 입력값을 주지 않았지만, man은 초깃값 True를 갖게 된다.

따라서 위 예에서 say_myself 함수를 사용한 2가지 방법은 모두 다음처럼 동일한 결과를 출력한다.

```
나의 이름은 박응용입니다.
나이는 27살입니다.
남자입니다.
```

이제 초깃값이 설정된 부분을 False로 바꿔 호출해 보자.

```
say_myself("춘향이", 21, False)
```

man 변수에 False 값이 대입되어 다음과 같은 결과가 출력된다.

```
나의 이름은 춘향이입니다.
나이는 21살입니다.
여자입니다.
```

함수의 매개변수에 초깃값을 설정할 때 주의할 것이 하나 있다. 만약 위에서 살펴본 say_myself 함수를 다음과 같이 만들면 어떻게 될까?

```
def say_myself(name, man=True, age):
    print("나의 이름은 %s 입니다." % name)
    print("나이는 %d살입니다." % age)
    if man:
        print("남자입니다.")
    else:
        print("여자입니다.")
```

이전 함수와 바뀐 부분은 초깃값을 설정한 매개변수의 위치이다. 결론을 미리 말하면 이것은 함수를 실행할 때 오류가 발생한다.

얼핏 생각하기에 위 함수를 호출하려면 다음과 같이 하면 될 것 같다.

```
say_myself("박응용", 27)
```

위와 같이 함수를 호출한다면 name 변수에는 "박응용"이 들어갈 것이다. 하지만 파이썬 인터프리터는 27을 man 매개변수와 age 매개변수 중 어느 곳에 대입해야 할지 판단하기 어려우므로 이러한 상황에서는 오류가 발생한다.

오류 메시지는 다음과 같다.

```
SyntaxError: non-default argument follows default argument
```

위 오류 메시지는 '초깃값이 없는 매개변수(age)는 초깃값이 있는 매개변수(man) 뒤에 사용할 수 없다'라는 뜻이다.
즉, 매개변수로 (name, age, man=True)는 되지만, (name, man=True, age)는 안 된다는 것이다.
초기화하고 싶은 매개변수는 항상 뒤쪽에 놓아야 한다는 것을 잊지 말자.

○ 함수 안에서 선언한 변수의 효력 범위

함수 안에서 사용할 변수의 이름을 함수 밖에서도 동일하게 사용한다면 어떻게 될까?

변수
  전역변수(Globla variable)
  지역변수(Local variable)

다음 예를 살펴보자.

```
a = 1
def vartest(a):
    a = a +1

vartest(a)
print(a)
--------------------
1
```

먼저 a라는 변수를 생성하고 1을 대입했다.
그리고 입력으로 들어온 값에 1을 더해 주고 결괏값은 리턴하지 않는 vartest 함수를 선언했다.
그리고 vartest 함수에 입력값으로 a를 주었다.
마지막으로 a의 값을 print(a)로 출력했다.
과연 어떤 값이 출력될까?

vartest 함수에서 매개변수 a의 값에 1을 더했으므로 2가 출력될 것 같지만, 프로그램 소스를 작성해서 실행해 보면 결괏값은 1이 나온다.
그 이유는 함수 안에서 사용하는 매개변수는 함수 안에서만 사용하는 '함수만의 변수'이기 때문이다.
즉, def vartest(a)에서 입력값을 전달받는 매개변수 a는 함수 안에서만 사용하는 변수일 뿐, 함수 밖의 변수 a와는 전혀 상관없다는 뜻이다.

따라서 vartest 함수는 다음처럼 매개변수 이름을 hello로 바꾸어도 이전의 vartest 함수와 완전히 동일하게 동작한다.

```
def vartest(hello):
    hello = hello + 1
```

즉, 함수 안에서 사용하는 매개변수는 함수 밖의 변수 이름과는 전혀 상관없다는 뜻이다.

다음 예를 보면 더욱 분명하게 이해할 수 있을 것이다.

```
def vartest(a):
    a = a + 1

vartest(3)
print(a)
```

위 프로그램 소스를 에디터로 작성해서 실행하면 어떻게 될까?
오류가 발생한다.

이유는 vartest(3)을 수행하면 vartest 함수 안에서 a는 4가 되지만,
함수를 호출하고 난 후 print(a) 문장은 오류가 발생하게 된다.
그 이유는 print(a)에서 사용한 a 변수는 어디에도 선언되지 않았기 때문이다.
다시 한번 말하지만, 함수 안에서 선언한 매개변수는 함수 안에서만 사용될 뿐, 함수 밖에서는 사용되지 않는다.
이것을 이해하는 것은 매우 중요하다.

○ 함수 안에서 함수 밖의 변수를 변경하는 방법

그렇다면 vartest라는 함수를 사용해서 함수 밖의 변수 a를 1만큼 증가할 수 있는 방법은 없을까?
이 질문에는 2가지 해결 방법이 있다.

1. return 사용하기

```
a = 1
def vartest(a):
    a = a +1
    return a
```

```
a = vartest(a)
print(a)
--------------------
2
```

첫 번째 방법은 return을 사용하는 방법이다.
vartest 함수는 입력으로 들어온 값에 1을 더한 값을 리턴하도록 변경했다.
따라서 a = vartest(a)라고 작성하면 a에는 vartest 함수의 리턴값이 대입된다.

여기에서도 물론 vartest 함수 안의 a 매개변수는 함수 밖의 a와는 다른 것이다.

2. global 명령어 사용하기

```
a = 1
def vartest():
    global a
    a = a + 1

vartest()
print(a)
--------------------
2
```

두 번째 방법은 global 명령어를 사용하는 방법이다.
위 예에서 볼 수 있듯이 vartest 함수 안의 global a 문장은 함수 안에서 함수 밖의 a 변수를 직접 사용하겠다는 뜻이다.
하지만 프로그래밍을 할 때 global 명령어는 가급적 사용하지 않는 것이 좋다.
함수는 독립적으로 존재하는 것이 좋기 때문이다.
외부 변수에 종속적인 함수는 그다지 좋은 함수가 아니다.
따라서 되도록 global 명령어를 사용하는 이 방법은 피하고 첫 번째 방법을 사용하기를 권한다.

◆ lambda 예약어

lambda는 함수를 생성할 때 사용하는 예약어로, def와 동일한 역할을 한다.
보통 함수를 한 줄로 간결하게 만들 때 사용한다.
우리말로는 '람다'라고 읽고 def를 사용해야 할 정도로 복잡하지 않거나 def를 사용할 수 없는 곳에 주로 쓰인다.

사용법은 다음과 같다.

```
함수_이름 = lambda 매개변수1, 매개변수2, ... : 매개변수를_이용한_표현식
```

한번 직접 만들어 보자.

```
add = lambda a, b: a+b
result = add(3, 4)
print(result)
------------------------------
7
```

lambda로 만든 함수는 return 명령어가 없어도 표현식의 결괏값을 리턴한다.

add는 2개의 인수를 받아 서로 더한 값을 리턴하는 lambda 함수이다. 위 예제는 def를 사용한 다음 함수와 하는 일이 완전히 동일하다.

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

result = add(3, 4)
print(result)
--------------------------
7
```

Python
1. Data Type |
2. if, while, for | 50%
3. 함수 |

add는 변수가 아니고 함수이다.

add = lambda. a,b : a+b
함수이름. 함수를 선언하는 키워드 매개변수 표현식