



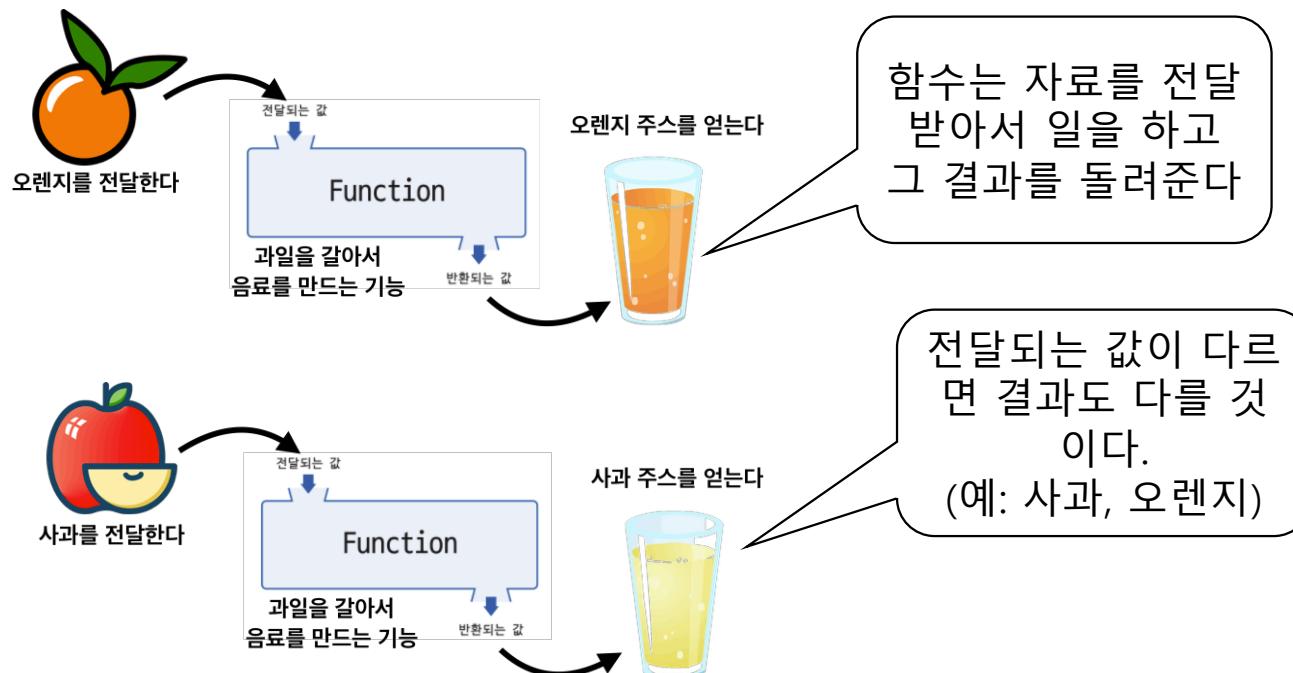
6장
함수로 일처리를 짜임새
있게 하자

이 장에서 배울 것들

- 함수의 개념을 학습해 보아요.
- 함수를 작성하는 방법을 익혀서 활용해 볼 것이에요.
- 함수를 호출하여 사용하여 방법을 학습해 보아요.
- 함수가 그 결과를 반환하는 return 문에 대하여 알아 보아요.
- 함수에 일을 시키기 위해서 넘겨주기 인자와 매개변수의 차이점을 알아 보아요.
- 효율적인 프로그래밍을 위한 모듈의 개념을 학습해 보아요.
- 모듈을 만들고 활용하는 연습도 함께 해 보아요.

6.1 짜임새 있는 기능을 만드는 멋진 기능 : 함수

- 파이썬에서는 프로그램을 쪼개는 3가지의 방법이 있다.
 - **함수****function**는 우리가 반복적으로 사용하는 코드를 묶은 것으로 코드의 덩어리와 같다.
 - **객체****object**는 코드 중에서 독립적인 단위로 분리할 수 있는 조각이다.
 - **모듈****module**은 프로그램의 일부를 가지고 있는 독립적인 파일이다.
- **함수**는 일을 수행하는 코드의 덩어리이며, 더 큰 프로그램을 구축하는 데 사용할 수 있는 작은 조각이다.



6.2 def 예약어를 이용하여 함수를 작성하고 호출하기

The diagram shows a Python function definition:

```
def print_address():
    print('경상북도')
    print('울릉군 울릉읍')
    print('독도리 산 1-96번지')
```

Annotations in red Korean text explain the code:

- 함수를 정의를 시작하는 예약어
- 함수 이름을 짓는다
- 함수에 전달할 입력을 정함: 이 함수는 인자 입력을 받지 않음
- 하위 블록의 시작을 알리는 기호
- 블록
- 하나의 블록을 이루는 코드 뭉치는 같은 길이로 들여쓰어야 함.
- 함수의 내용을 담은 블록은 함수 이름을 정의하는 문장보다 한 단계 길이 들여 썼음

- 첫째 줄 `def print_address():`에서 `def` 키워드를 이용하여 함수를 정의하고 있다. 이어서 함수의 이름을 적은 후에 소괄호 `()`와 콜론 `:`을 붙인다. 여기서 콜론 `:`은 코드 블록이 이어서 등장한다는 것을 의미한다. 이것은 이전 장에서 배운 `if`, `for`, `while`의 경우와 마찬가지이다.
- 두 번째 줄부터 네 번째 줄까지는 함수를 구성하는 문장들이 들어간다. 함수를 구성하는 문장들은 반드시 들여쓰기를 해야한다. 이 덩어리를 블록이라고 하고, 함수는 이 블록을 실행한다.

6.2 def 예약어를 이용하여 함수를 작성하고 호출하기

- 함수 안에 있는 코드들은 자동으로 실행되지 않으며, 함수가 호출되어야 함수 안의 코드가 실행된다.
- 함수 호출** function call이란 다음과 같이 함수 이름을 적어주는 것이다. 위의 코드에 다음과 같은 1줄만 추가하면 print_address() 함수가 실행되고 아래와 같이 실행 결과가 표시된다.

```
def print_address():
    print('경상북도')
    print('울릉군 울릉읍')
    print('독도리 산 1-96번지')
```

```
print_address() # 정의한 함수를 호출
```

경상북도
울릉군 울릉읍
독도리 산 1-96번지

함수 호출로 인한
수행결과.

반드시 함수 호출
문이 있어야만 함
수가 수행된다.



도전문제 6.1

- (1) 자신의 학교 혹은 직장 주소와 대표 전화번호를 출력하는 함수 `print_company_address()` 함수를 정의하여라 그리고 이 함수를 호출한 후 그 결과를 출력하여라.
- (2) 20개의 별표기호(*)를 한 줄에 출력하는 `print_star()` 함수를 생성하여라. 다음과 같이 이 함수를 호출하고 그 결과를 출력하여라.

```
print_star()  
print_star()  
print_star()  
print_star()  
print_star()
```

```
*****  
*****  
*****  
*****  
*****
```

6.3 함수를 만들고 불러서 일을 시켜보자

- 우리가 함수를 정의하더라도 **함수를 호출하지 않으면 함수 안의 코드는 실행되지 않는다.**
- 함수를 호출하려면 함수 이름과 소괄호를 이용한다. 아래는 수행 결과

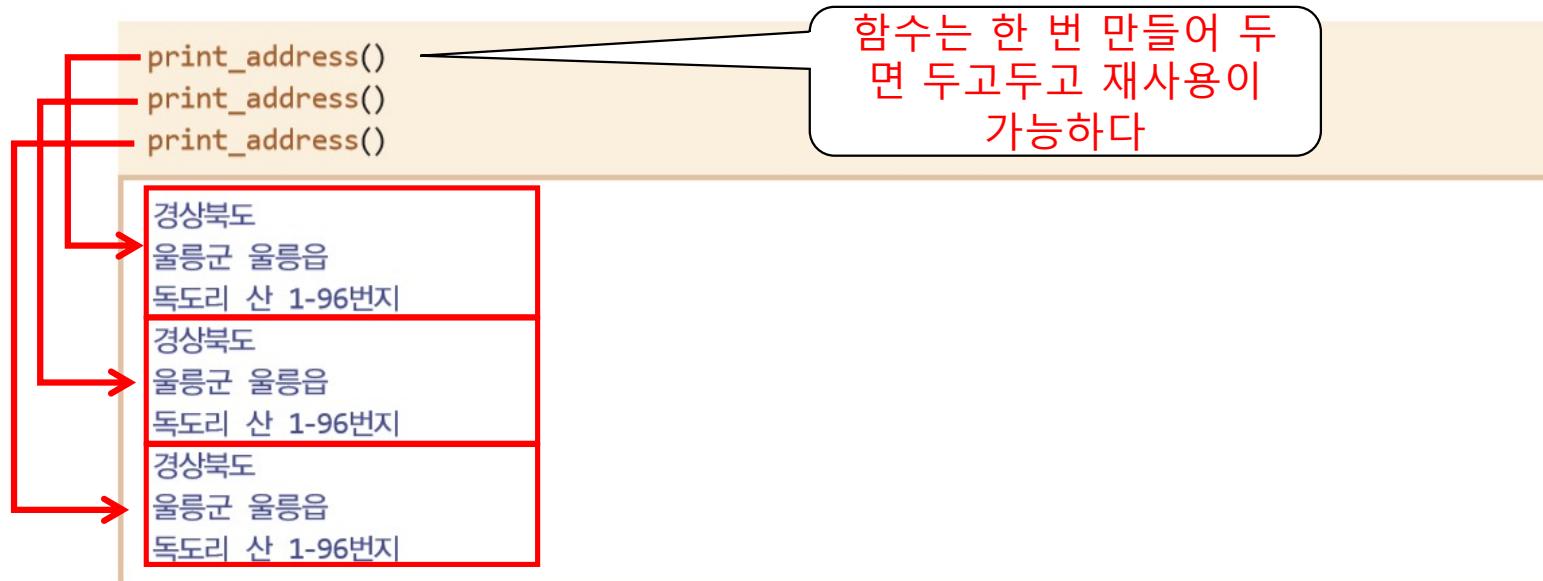
```
경상북도  
울릉군 울릉읍  
독도리 산 1-96번지
```

- 그런데 생각해보면 위와 같은 출력은 왼쪽과 같은 문장으로도 얼마든지 출력할 수 있다.

```
print('경상북도')  
print('울릉군 울릉읍')  
print('독도리 산 1-96번지')
```

6.3 함수를 만들고 불러서 일을 시켜보자

- 함수를 사용하는 주된 이유는 우리가 한 번만 함수를 정의하면 언제든지 필요할 때면 함수를 불러서 일을 시킬 수 있기 때문이다.
- 이 주소를 3번 인쇄하여 출력하고자 하는 경우를 생각해보자. 함수가 정의된 상태라면 우리는 다음과 같이 이 작업을 수행할 수 있다.



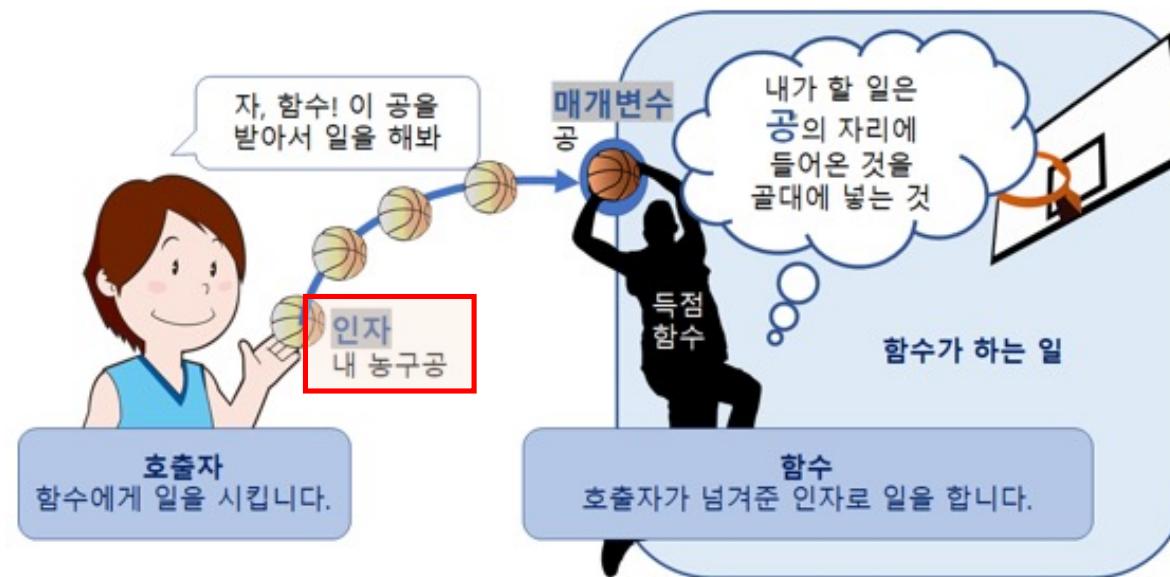


잠깐 – 함수와 루프^{loop}는 뭐가 다른가요?

위의 출력을 보면 흡사 반복 루프를 사용한 것 같다. 그렇다면 반복 루프와 함수는 어떻게 다를까? 만약 주소를 3번 출력하는 작업이 한 곳에서만 일어난다면 반복 루프와 함수는 흡사하다. 하지만 함수를 사용하면 프로그램의 여러 곳에서 동일한 작업을 시킬 수 있다. 반복 루프는 이것이 불가능하다. 그리고 함수에는 입력을 전달하여서 호출할 때마다 서로 다르게 실행하게 할 수 있다.

6.4 함수에 하나의 값을 넘겨주고 일을 시키자

- 우리는 함수 외부에서 함수 안으로 값(정보)을 전달할 수 있다. 이 값을 인자^{argument}라고 한다.



- 예를 들어서 앞의 주소를 인쇄하는 `print_address()` 함수에서 수신자의 이름은 외부에서 전달 받는 것으로 해보자. 수신자의 이름을 변경할 수 있으면 같은 회사에서 근무하는 사람들도 사용할 수 있을 것이다.

6.4 함수에 하나의 값을 넘겨주고 일을 시키자

```
def print_address(name):  
    print("서울 특별시 종로구 1번지")  
    print("파이썬 빌딩 7층")  
    print(name)
```

```
print_address("홍길동")  
print_address("널널한 교수")
```

함수의 매개변수 name을
통해서 그 결과를 다르게
만들 수 있다

"홍길동" 대신 여러분
의 이름도 가능하다

- 함수 정의부를 살펴보면 이전과 달리 함수 이름 뒤의 소괄호 안에 변수 name이 있다. 이 변수 name을 통하여 함수로 값이 전달된다. 메인 프로그램에서 print_address()를 호출할 때 "홍길동"이라는 문자열을 괄호 안에 넣어 주었는데. 이것이 함수 내부의 name 변수로 전달되는 것이다.
- 함수 호출시 전달되는 실제 값을 **인자argument**라고 하고, 함수 내부에서 전달받는 변수를 **매개변수parameter**라고 한다.

함수 내부에서 전달 받는 변수 : 매개변수

```
def print_address(name):
    print("서울 특별시 종로구 1번지")
    print("파이썬 빌딩 7층")
    print(name)
```

```
print_address("홍길동")
print_address("널널한 교수")
```

함수에 넘겨지는 값 : 인자



도전문제 6.2

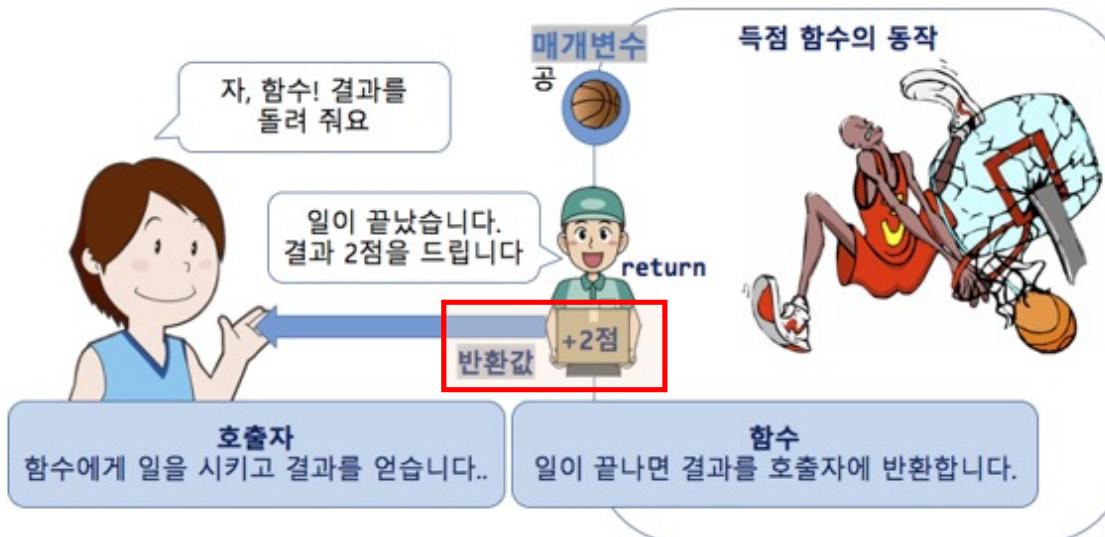
`print_address2(name, address)`와 같이 2개의 매개변수를 가지는 함수를 정의하여라. 이 함수를 호출할 때 `print_address("홍길순", "부산광역시 남구 광안로 10번길 1-1")`라고 호출하면 다음과 같은 출력이 나타나도록 함수 몸체를 작성하여라.

이름 : 홍길순

주소: 부산광역시 남구 광안로 10번길 1-1

6.5 함수에 일을 시키고 그 값을 받아오도록 하자

- 지금까지, 함수는 우리가 주는 값을 받아서 작업을 처리하였다. 그러나 함수가 매우 유용한 것은 함수가 주어진 일을 하고 우리에게 뭔가를 보낼 수 있다는 점이다. 이와 같이 함수로부터 되돌아오는 값을 **반환 값**^{return value}이라고 한다.



- 함수가 함수 내부의 값을 외부에 보낼 때는 **return 키워드**를 사용하면 된다. 원의 반지름을 보내면 원의 면적을 계산해서 반환하는 함수를 작성해보자.

- 원의 반지름을 보내면 원의 면적을 계산해서 반환하는 함수를 작성해보자.

```
def calculate_area(radius):
    area = 3.14 * radius**2
    return area          # 이전 줄에서 구한 area 값을 호출문에 돌려준다
```

- 이 함수의 호출문에 다음과 같은 할당 문을 넣어보자.

```
c_area = calculate_area(5.0)  # calculate_are() 함수가 계산한 값을 c_area에 저장
```

- c_area 에는 78.5 값이 반환되어 저장된다. 다음과 같이 출력해 보자.

```
>>> print(calculate_area(5.0))
78.5
>>> area_sum = calculate_area(5.0) + calculate_area(10.0)
>>> print(area_sum)
```

- 다음과 같이 하면 함수 호출만 하고 그 반환값을 사용하지 않게 된다

```
>>> calculate_area(10.0)  # 함수를 호출만 하고 그 반환값을 사용안함
```

6.6 함수에 여러 개의 값을 넘겨주는 고급 기능

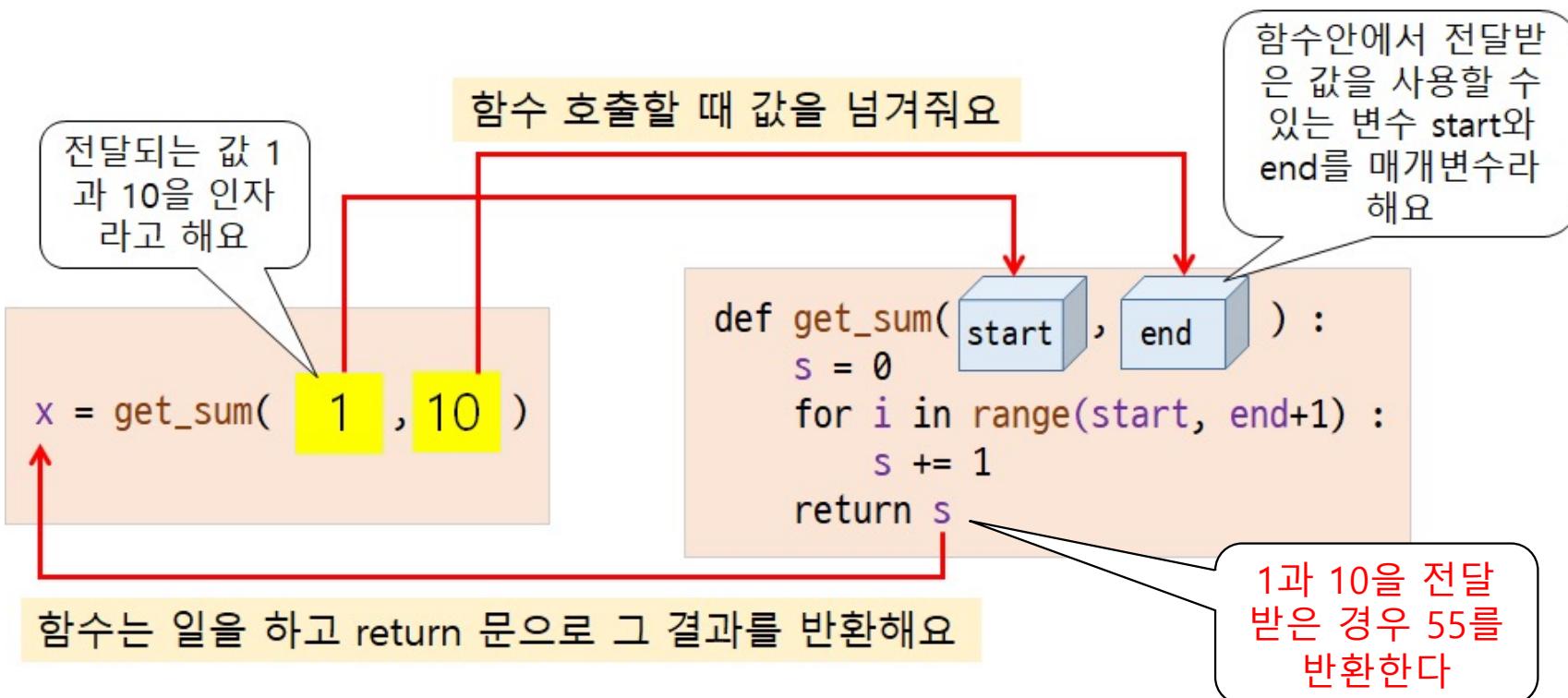
- 2개의 정수 start와 end를 받아서 start에서 end까지의 합을 계산하는 함수 get_sum()을 작성하자.

```
def get_sum(start, end):          # start, end를 매개변수로하여 인자를 받는다
    s = 0
    for i in range(start, end+1):   # start부터 end까지 정수의 합을 구함
        s += i
    return s                        # start부터 end까지 수의 합을 반환한다

print(get_sum(1, 10))            # 1에서 10까지 정수의 합 55를 출력한다
```

55

- 인자와 매개변수는 함수 호출 시에 데이터를 주고받기 위하여 필요하다. 다시 정리하면, **인자argument**는 호출 프로그램에 의하여 함수에 실제로 전달되는 값이다. **매개변수parameter**는 이 값을 전달받는 변수이다.

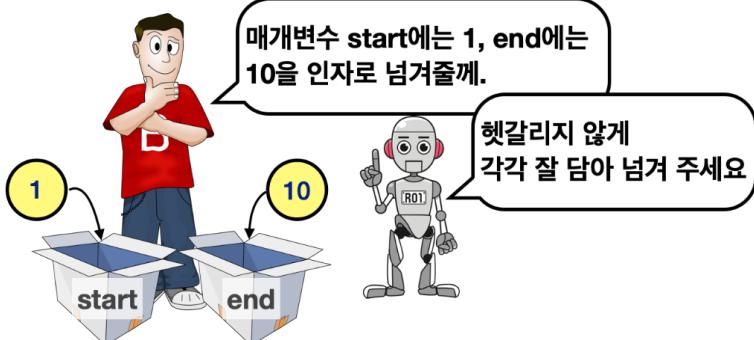


6.6 함수에 여러 개의 값을 넘겨주는 고급 기능

```
x = get_sum(1, 10) # 1과 10이 인자가 된다.  
print('x =', x)
```

```
y = get_sum(1, 20) # 1과 20이 인자가 된다.  
print('y =', y)
```

```
x = 55  
y = 210
```



- 여기서 주의할 점은 매개변수의 개수와 넘어가는 인자의 개수가 정확히 일치하여야 한다는 점이다.
- 즉 매개 변수가 두 개이면 인자도 두 개를 전달하여야 한다. 매개변수의 개수와 인자의 개수가 일치하지 않으면 오류가 발생하게 된다.

6.7 여러 개의 값을 넘겨주고 여러 개의 값을 돌려받자

- 지금까지 함수는 호출될 때 맡은 일을 수행하고 수행을 마치거나, 수행을 마칠 때 **하나의 값을 반환**하는 일을 하는 것을 살펴보았다. 파이썬의 큰 특징 중 하나는 함수가 **여러 개의 값을 반환**할 수 있다는 것으로, 다음과 같은 형식을 사용한다. 가장 큰 이유는 생산성이 뛰어나기 때문이다.
- 파이썬을 이용하면 간결하면서도 효율적인 프로그램을 빠르게 작성할 수 있다.

```
def sort_num(n1, n2):          # 2개의 값을 받아오는 함수
    if n1 < n2:
        return n1, n2           # n1이 더 작으면 n1, n2 순서로 반환
    else:
        return n2, n1           # n2가 더 작으면 n2, n1 순서로 반환

print(sort_num(110, 210))      # 110과 210을 함수의 인자로 전달하고 반환되는 값을 출력
print(sort_num(2100, 80))
```

(110, 210)
(80, 2100)

항상 작은 수, 큰 수
쌍이 반환된다

- 2개의 수를 인자로 받아서 사칙연산을 수행하고 그 값을 출력하는 프로그램

```

def calc(n1, n2):
    return n1 + n2, n1 - n2, n1 * n2, n1 / n2 # 덧셈, 뺄셈, 곱셈, 나눗셈 결과를 반환

n1, n2 = 200, 100
t1, t2, t3, t4 = calc(n1, n2) # 네 개의 값을 반환받기 위해 4개의 변수를 사용함
print(n1, '+', n2, '=', t1)
print(n1, '-', n2, '=', t2)
print(n1, '*', n2, '=', t3)
print(n1, '/', n2, '=', t4)

```

200 + 100 = 300
 200 - 100 = 100
 200 * 100 = 20000
 200 / 100 = 2.0

사칙연산의 결과를
반환받는다.



도전문제 6.3

a, b, c의 세 인자를 받아서 이 세 수의 제곱을 반환하는 함수 `get_square(a, b, c)`를 작성하여라. 그리고 이 함수가 반환하는 세 개의 결과 `a_sq, b_sq, c_sq`를 출력하도록 하여라.

```

a, b, c = 1, 2, 3
a_sq, b_sq, c_sq = get_square(a, b, c)
print(a, '제곱 :', a_sq, ', ', b, '제곱 :', b_sq, ', ', c, '제곱 :', c_sq)

```

1 제곱 : 1 , 2 제곱 : 4 , 3 제곱 : 9

6.8 변수의 범위는 어디까지인가

- 이제 변수의 사용 범위를 함수와 관련해 깊이 있게 다루어 보려 한다. 다음 코드를 작성하고 수행해 보라.

```
def print_counter():
    print('counter =', counter) # 함수 내부의 counter 값

counter = 100
print_counter()
print('counter =', counter) # 함수 외부의 counter 값

counter = 100
counter = 100
```

- counter라는 변수를 함수 외부에서 생성한 후 100이라는 값을 할당하자. 그리고 함수 내부에서 이 변수를 출력하면 100이 출력된다. 당연히 함수 외부에서도 이 출력값은 100이 될 것이다.

6.8 변수의 범위는 어디까지인가

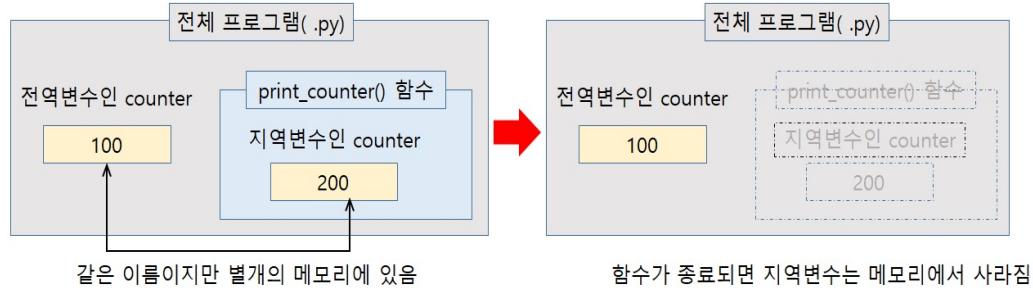
- 이제 다음과 같이 함수 내부에서 counter = 200과 같이 counter에 새로운 값을 할당해 보도록 하자. 함수 외부와 내부 모두 200이 출력될까?

```
def print_counter():
    counter = 200
    print('counter =', counter) # 함수 내부의 counter 값

counter = 100
print_counter()
print('counter =', counter) # 함수 외부의 counter 값

counter = 200
counter = 100
```

- counter = 200과 같은 할당이 이루어질 때 이 변수는 함수 안에서 새로 선언되고 생성된다.
- 이와 같이 함수 안에서 생성되는 변수를 **지역변수 local variable**라고 한다. 이 지역변수 counter가 참조하는 값 200은 전역변수가 참조하는 값 100과 별개의 메모리에 있으며, 지역변수는 함수가 종료되면 사라진다. 따라서 counter 변수를 함수의 외부에서 출력할 경우 200이 아닌 100이 출력되게 되는 것이다.

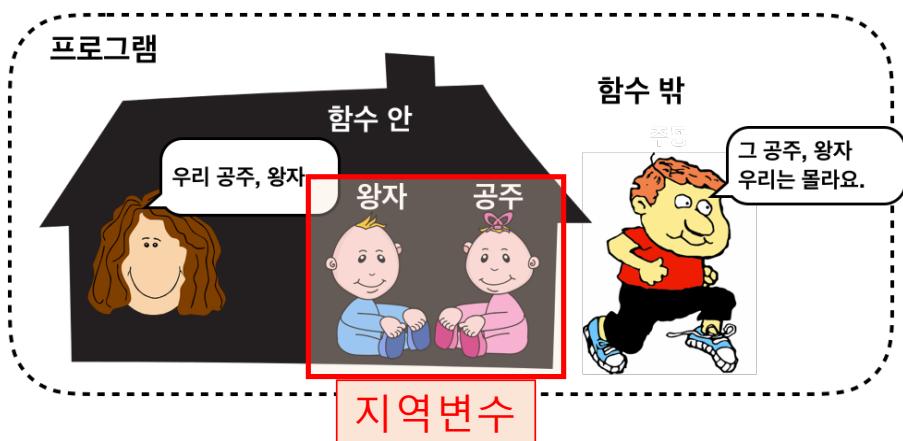


함수가 호출될 때 생성되어 호출이 완료되면 메모리에서 사라지는 지역변수

```
def print_counter():
    counter = 200
    print('counter =', counter)
```

```
counter = 100
print_counter()
print('counter =', counter)
```

print_counter() 함수의 호출에 관계없이 메모리에서 유지되는 전역변수



6.9 함수 안에서 전역변수 사용하기 : global 키워드

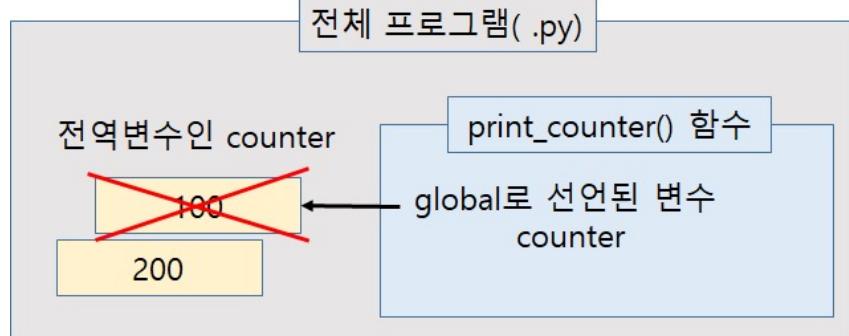
- 만일 함수 내부에서 새로 변수를 만들지 않고 전역변수인 counter를 불러서 사용하려면 어떻게 해야할까? 아래와 같이 global이라는 키워드를 사용하는 것이 바람직하다. global counter는 함수 외부의 전역변수 counter를 사용하겠다는 선언이다.

```
def print_counter():
    global counter      # 함수 외부의 전역변수 counter를 사용하겠다는 선언
    counter = 200
    print('counter =', counter) # 함수 내부의 counter 값

counter = 100
print_counter()
print('counter =', counter) # 함수 외부의 counter 값

counter = 200
counter = 200
```

- 이와 같은 선언을 할 경우 아래의 그림과 같이 print_counter()는 외부의 전역변수 counter를 사용하므로 counter = 200을 적용하면 전역변수의 값이 200으로 변경된다.



```
def print_counter():
```

```
    global counter  
    counter = 200
```

```
    print('counter =', counter)
```

지역변수가 아닌 전역변수
counter를 사용하게 된다.
함수 내부에서 값을 바꾸면
외부에서도 그 값이 바뀌게 된다

```
counter = 100
```

```
print_counter()
```

```
print('counter =', counter)
```

```
def calculate_area(radius):  
    global area  
    area = 3.14 * radius**2  
    return  
  
area = 0  
r = float(input("원의 반지름: "))  
calculate_area(r)  
print(area)
```

area 값을 반환하지 않아도
외부에서 그 값이 0에서
78.5로 바뀐다

원의 반지름: 5.0

78.5

주의 : 전역변수를 너무 많이
사용하면 코드가 길어질 경우
에러발생의 원인이 되니 꼭
필요한 경우에 한해서만 사용
합시다.

6.10 함수에 쉽게 일을 시키는 디폴트 인자

- 파이썬에서는 함수의 매개변수가 기본값을 가질 수 있다. 이것을 **디폴트 인자** **default argument**라고 한다. 예를 들어 보자. 다음과 같이 햄버거 주문 함수 `order()`가 있다고 하자. `greet()`는 항상 3개의 인자를 받아야 한다. 첫 인자는 햄버거의 개수, 두 번째는 피클 포함여부, 세 번째는 양파 포함 여부이다.

```
def order(num, pickle, onion) :  
    print('햄버거 {0} 개 - 피클 {1}, 양파 {2}'.format(num, pickle, onion))  
  
order(1, False, True)
```

햄버거 1 개 - 피클 False, 양파 True

- 만약 우리가 `order()` 함수에 3개의 인자를 전달하지 않으면 오류가 발생한다.

```
>>> order(1)      # 인자를 1개만 전달하여 order()을 호출함  
...  
TypeError: order1() missing 2 required positional arguments: 'pickle' and 'onion'
```

6.10 함수에 쉽게 일을 시키는 디폴트 인자

- 이와 같이 인자가 부족한 경우에 기본값을 넣어주는 메커니즘이 있다면 편리할 것이다. 바로 이러한 목적으로 사용하는 것이 디폴트 인자이다.

```
def order(num, pickle = True, onion = True) :  
    print('햄버거 {0} 개 - 피클 {1}, 양파 {2}'.format(num, pickle, onion))  
  
order(1, pickle = False, onion = True)  
order(2)      # 햄버거 2개를 주문, 디폴트로 pickle, onion 값은 True임
```

햄버거 1 개 - 피클 False, 양파 True

햄버거 2 개 - 피클 True, 양파 True

- 위의 코드를 살펴보면 pickle과 onion 매개변수에 True라는 디폴트 인자를 넣어 주었다. 따라서 order(2)와 같이 호출해도 프로그램은 오류없이 잘 수행된다. 반면 이 프로그램에서는 num에 대한 디폴트 값을 주지 않았으며 따라서 order()과 같은 방식으로 함수를 호출하면 프로그램은 오류를 출력할 것이다.

6.10 함수에 쉽게 일을 시키는 디폴트 인자

order(3) 만하면 자동으로 order(3, True, True) 가 된다



- 디폴트 값이 있는 매개 변수는 다음과 같이 순서를 바꾸어 인자를 넣을 수도 있다.

```
>>> order(3, onion = False, pickle = True)      # 인자의 순서를 바꾸었다.  
햄버거 3 개 - 피클 True, 양파 False
```

6.11 키워드 인자로 더욱더 고급지게 함수 활용하기

- 파이썬에서 대부분의 인자들은 함수 호출 시에 위치에 의하여 구별된다. 예를 들어서 `power(2, 10)`은 `power(10, 2)`과는 다르다. 함수 호출 `power(2, 10)`은 밑`base`이 2이고 지수`exponent`가 10인 2^{10} 을 계산할 것이고 `power(10, 2)`은 10^2 을 계산할 것이다.
- 혼동을 줄이고 안전한 프로그램을 위해 **키워드 인자**`keyword argument`는 인자들 앞에 키워드를 두어서 인자들을 구분한다

```
def power(base, exponent):  
    return base**exponent # base가 밑이고, exponent가 지수값이다.
```

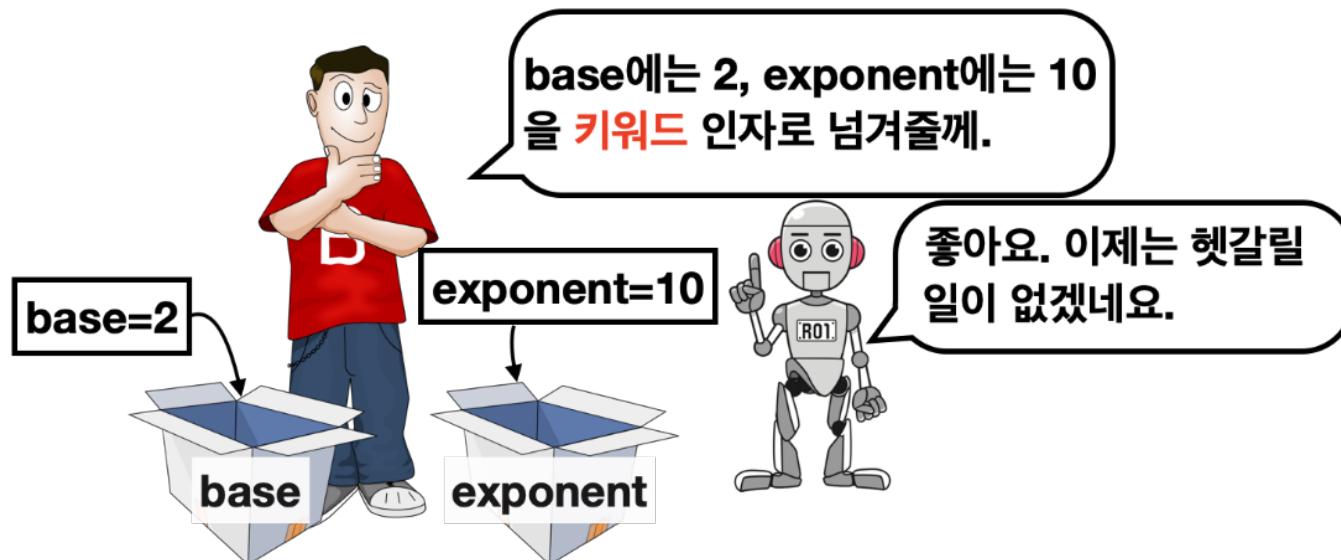
```
>>> power(2, 10) # 2의 10승을 반환한다  
1024  
>>> power(10, 2) # 10의 2승을 반환한다  
100
```

어떤 값이 밑이고 지수인지
헷갈릴 수 있다 -> 오류의
원인이 된다

6.11 키워드 인자로 더욱더 고급지게 함수 활용하기

```
>>> power(base=2, exponent=10) # 2의 10승을 반환한다  
1024  
>>> power(exponent=10, base=2) # 2의 10승을 반환한다  
1024
```

밑(base)과 지수(exponent)
를 명시를 해주므로 헷갈리
지 않는다.



- 하지만 다음과 같이 첫 인자에 키워드 인자를 넣고 두번째로 위치 인자로 넣게 되면 오류가 발생한다. 인자의 순서는 언제나 위치 인자가 먼저 나오고 키워드 인자가 나온 뒤에는 위치 인자가 나올 수 없다.

```
>>> power(base = 10, 2)    # 문법 오류  
SyntaxError: positional argument follows keyword argument
```

LAB⁶⁻³ 주급 계산 프로그램

주단위로 봉급을 받는 아르바이트생이 있다고 하자. 현재 시급과 일한 시간을 입력하면 주급을 계산해주는 함수 `weeklyPay(rate, hour)`를 만들고 이 함수를 호출하여 주급을 출력하는 프로그램을 작성해보자. 30시간이 넘는 근무 시간에 대해서는 시급의 1.5배를 지급한다고 하자. 이 함수를 구현하고 입력받은 값을 `r`와 `h` 변수에 넣어서 키워드 인자로 함수 `weeklyPay()`를 호출하는 프로그램을 작성하여라.

원하는 결과

시급을 입력하시오:10000

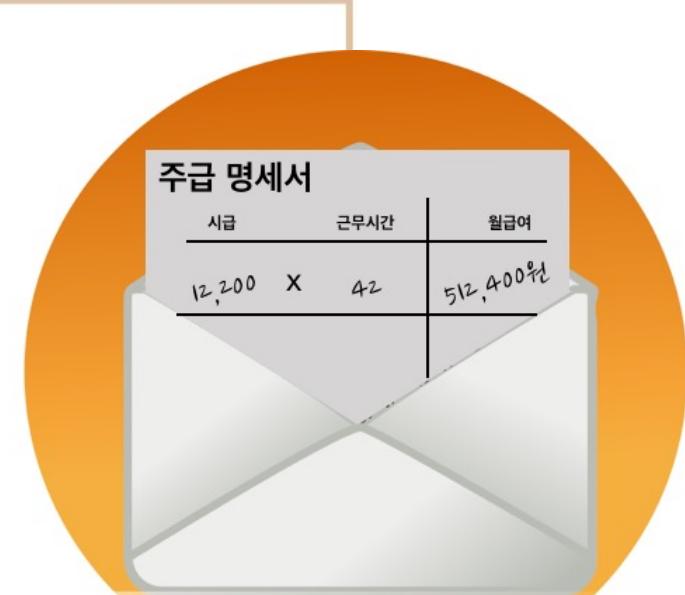
근무 시간을 입력하시오:38

주급은 420000.0

LAB⁶⁻³ 주급 계산 프로그램

```
def weeklyPay(rate, hour):
    if (hour > 30):
        money = rate*30 + 1.5*rate*(hour-30)
    else:
        money = rate*hour
    return money

r = int(input("시급을 입력하시오: "))          # 시급입력받기
h = int(input("근무 시간을 입력하시오: "))      # 근무시간 입력받기
print("주급은 " + str(weeklyPay(rate = r, hour = h)))
```



LAB⁶⁻⁴ 리스트에서 최대값/최소값을 편리하게 찾는 함수

[27, 90, 30, 87, 56]와 같은 리스트에서 최댓값이나 최솟값을 찾는 `getMinMax()` 함수를 만드려고 한다. 이 함수는 리스트를 첫 번째 매개변수로 하고, 두 번째 매개변수는 `method`라는 이름을 가지는 키워드 인자를 받는다고 하자. 이 `method`에 주어지는 인자가 '`max`'이면 최댓값, '`min`'이면 최솟값을 찾아 출력하도록 하는 것이다.

원하는 결과

[27, 90, 30, 87, 56]

최댓값을 원하면 `max`, 최솟값을 원하면 `min`을 입력하시오: `max`

90

[27, 90, 30, 87, 56]

최댓값을 원하면 `max`, 최솟값을 원하면 `min`을 입력하시오: `min`

27

[27, 90, 30, 87, 56]

최댓값을 원하면 `max`, 최솟값을 원하면 `min`을 입력하시오: `maximum`

`illegal method`

LAB⁶⁻⁴ 리스트에서 최대값/최소값을 편리하게 찾는 함수

```
def getMinMax(mylist, method = 'max'):
    minValue = 9999999999999999999999999999999999999999
    maxValue = -minValue

    if method == 'max' :
        for value in mylist:
            if value > maxValue:
                maxValue = value;
        return maxValue
    elif method == 'min' :
        for value in mylist:
            if value < minValue:
                minValue = value;
        return minValue
    else :
        print('illegal method')

list_data = [27, 90, 30, 87, 56]
while(True) :
    print(list_data)
    s = input('최댓값을 원하면 max, 최솟값을 원하면 min을 입력하시오: ')
    print(getMinMax(list_data, s))
```

6.12 자신을 호출하는 재귀 함수

- **재귀함수** recursion 란 함수 내부에서 자기 자신을 호출하는 함수를 말한다. 재귀는 매우 유용한 문제해결 기법으로 절차적 기법으로 해결하기 어려운 문제를 직관적이고 간단하게 해결할 수 있다.
- 팩토리얼 식과 정의 : $n! = n * (n - 1)!$

재귀적 정의

```
1! = 1
2! = 2 * 1
3! = 3 * 2 * 1
...
n! = n * (n-1) * (n-2) * ... * 2 * 1
```

6.12 자신을 호출하는 재귀 함수

- 이 정의를 바탕으로 팩토리얼은 $1! = 1$ 일 때 $n! = n \cdot (n-1)!$ 로 정의할 수 있다. 이를 코드로 구현하면 다음과 같이 나타낼 수 있다.

```
def factorial(n):          # n!의 재귀적 구현
    if n <= 1 :            # 종료 조건이 반드시 필요하다
        return 1
    else :
        return n * factorial(n-1)    # n * (n-1)! 정의에 따른 구현

print('4! = ', factorial(4))      # 인자로 4를 넣어 호출
```

- 재귀함수에서 중요한 점은 이 프로그램이 언제 종료되는가인데 정의에 의하여 $1! = 1$ 인 경우는 그 값이 정의되지만 1이하의 값에 대해서는 팩토리얼을 정의할 필요가 없으므로 if 문에서 n이 1이하가 되면 1을 반환하면 된다.



잠깐 – 재귀호출과 분할 정복

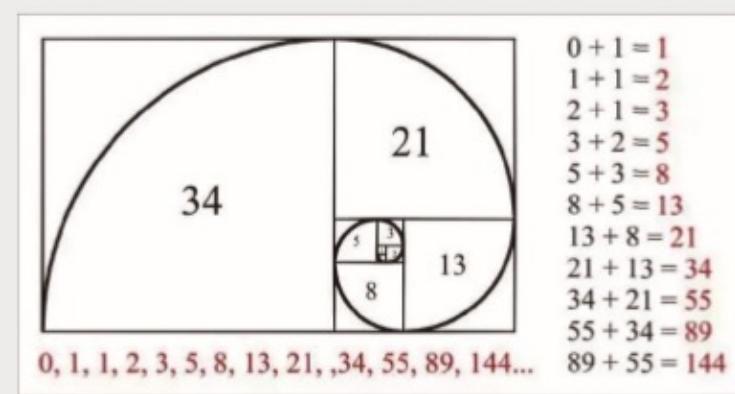
재귀호출은 함수가 자신을 호출하는데 문제의 규모가 n 일 때, n 보다 작은 규모의 문제로 쪼개어 호출하는 것이 일반적이다. 이러한 풀이 방식은 **분할 정복divide and conquer** 알고리즘과 관련이 깊다.

분할 정복 알고리즘은 **여러 갈래 재귀호출multi-branch recursion**이라고 할 수 있는데, 앞에서 본 팩토리얼 예제는 재귀호출이 한 번만 이루어지는 한 갈래 재귀호출이라 분할 정복 알고리즘이라고 할 수 없다. 문제의 크기가 n 일 때, 문제를 $n/2$ 크기로 둘로 나누어 각각을 재귀호출의 인자로 넘겨주면 전형적인 분할 정복 알고리즘이 될 것이다. 이렇게 반으로 쪼개는 것 뿐만 아니고 두 개보다 더 많은 수의 작은 문제로 쪼개어 재귀호출을 적용하는 방법들도 모두 분할 정복 알고리즘이다. 쪼개어진 문제를 모두 합했을 때 원래 문제보다 많이 커지면 안 된다.

이 알고리즘의 이름은 고대부터 존재한 정치적 사회적 전략에 뿌리를 두고 있다. 라틴어로 divide et impera라고도 한다. 이 전략은 자신을 제외한 누구에게도 권력이 집중되지 않도록 전체 권력을 잘게 쪼개어 나누어 주는 것이다. 국가들 사이의 지배관계에도 흔히 사용되었던 전략이다.

LAB⁶⁻⁷ 피보나치 함수 계산하기

피보나치 수열이란 앞의 두 항을 더해 다음 항을 만드는 수열이다. 피보나치 수열의 첫 번째 항과 두 번째 항은 0, 1로 설정한다. 이 수열에 속한 수를 피보나치 수라고 한다. 피보나치 수열의 n 번째 항을 계산하여 반환하는 함수 `fibonacci(n)`를 작성해보자.



원하는 결과

몇 번째 항: 9

21

LAB⁶⁻⁷ 피보나치 함수 계산하기

```
def fibonacci(n):
    if n < 0:                                # 입력 오류 검사
        print("잘못된 입력입니다.")
    elif n == 1:                               # 재귀 호출 중단 조건
        return 0
    elif n == 2:                               # 재귀 호출 중단 조건
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2) # 재귀 호출

i = int(input("몇 번째 항: "))
print(fibonacci(i))
```



잠깐 – 피보나치 수열 계산을 재귀호출로 하는 것은 분할 정복 divide and conquer이 아니다

우리가 구현한 피보나치 수열 계산을 위한 재귀호출 알고리즘은 분명 여러 갈래 재귀호출이다. 하지만 이 방법은 분할 정복 기법이 아니며, 효율적이지도 않다. 그 이유는 n 크기의 문제를 $n - 1$ 과 $n - 2$ 크기의 문제로 쪼개면 문제 크기의 합이 $2n - 3$ 이 되어 원래 문제보다 훨씬 더 크기 때문이다.



도전문제 6.4

재귀호출을 이용한 피보나치 수열 계산은 n 이 큰 값이 주어지면 너무 많은 계산이 필요해 제대로 값을 내지 못 한다. 하지만 $f_1 = 0, f_2 = 1$ 을 기록해 두면 $f_3 = 1$ 이라는 것을 쉽게 알 수 있고, 이것을 추가로 기록하면, f_4 는 마지막에 기록된 f_2 와 f_3 을 가져와 쉽게 2임을 알 수 있다. 이렇게 답을 늘려가는 방식을 쓰면 매우 큰 n 에 대해서도 f_n 을 구할 수 있을 것이다. 이러한 방식으로 코드를 개선해 보라.

6.13 모듈을 이용해서 함수를 두고두고 재사용하자

- **모듈**module이란, 파이썬 함수나 변수 또는 클래스들을 모아놓은 스크립트 파일을 말한다. 파이썬은 여러 기관과 개발자들에 의해서 만들어진 많은 모듈이 있으며 우리는 이 모듈을 이용하여 효과적으로 소프트웨어를 개발할 수 있다.
- 다음은 현재 스크립트나 대화창에서 모듈을 불러오는 명령이다.

```
import 모듈이름1 [, 모듈이름2, ... ]
```

6.13 모듈을 이용해서 함수를 두고두고 재사용하자

- 파이썬 설치와 함께 제공되는 모듈을 **파이썬 표준 라이브러리**python standard library라고도 하며 문자열과 텍스트 처리, 이진 데이터 처리, 날짜와 시간 처리, 배열 등의 자료형 처리, 수치 연산과 수학 함수, 파일과 디렉토리 접근, 유닉스 시스템의 데이터베이스 접근, 데이터 압축, 그래픽 등을 위한 100여 가지 이상의 모듈이 표준 라이브러리들로 제공된다.
- 우선 날짜와 시간을 다루는 데에 편리한 `datetime` 모듈을 살펴 보자. 다음과 같이 불러들여 사용할 수 있다.

```
>>> import datetime          # 날짜와 시간을 다루는 모듈
>>> datetime.datetime.now()
datetime.datetime(2021, 1, 2, 6, 57, 27, 904565)
```

```
>>> today = datetime.date.today()
>>> print(today)
2021-09-14
>>> today
datetime.date(2021, 9, 14)
>>> today.year
2021
>>> today.month
9
>>> today.day
14
```

6.14 나만의 모듈을 만들고 불러서 사용해보자

- 우리가 만든 함수들을 여러 군데에서 다시 사용할 수 있도록 함수를 저장해 놓은 모듈을 만들어 보자. 여러 개의 함수를 담은 모듈과 하나의 함수를 담은 모듈 모두 같은 방식으로 만들고 사용하므로 여기서는 아래와 같이 간단한 함수 하나를 담은 모듈을 만들어 보자.

```
# filename: my_func.py
def mf_print(msg, n = 1) :
    print(msg * n)
```

- 모듈이라는 것은 파이썬 파일이다. 그리고 그 파일의 이름이 바로 모듈의 이름이 된다.
- 모듈이름.함수이름() 형태로 호출하면 된다.

```
# filename: main.py
import my_func

my_func.mf_print("my_func was imported ", 3) # my_func 모듈의 mf_print() 함수 호출
```

```
my_func was imported my_func was imported my_func was imported
```

```
# filename: main.py
import my_func as mf

mf.mf_print('[alias]', 5) # mf라는 별명을 사용해서 메시지를 5회 반복출력
```

```
[alias][alias][alias][alias][alias]
```

```
# filename: main.py
from my_func import mf_print

mf_print('-no module name-', 2) # mf_print()를 호출할 때 my_func.을 사용안해도 됨
```

```
-no module name--no module name-
```

```
# filename: main.py
from my_func import *

mf_print('-no module name-', 2)
```

```
-no module name--no module name-
```



summary

핵심 정리



- 함수가 무엇인지를 학습하였다.
- 인자와 매개변수가 무엇인지를 학습하였다.
- 어떻게 함수로 인자를 전달할 수 있는지를 학습하였다.
- 여러 개의 인자를 함수로 전달하는 방법을 학습하였다.
- 함수가 값을 반환하는 방법을 학습하였다.
- 지역변수와 전역변수의 차이점에 대하여 학습하였다.
- **global** 키워드를 사용하여서 함수 안에서 전역변수를 사용하는 방법을 학습하였다.
- 함수가 자기자신을 호출하여 문제를 해결할 수 있다는 것을 알게 되었다.
- 모듈을 사용하여 여러가지 함수를 구현해 두고 다른 코드에서 불러와서 사용할 수 있다.

따라하며 배우는

파이썬과 데이터 과학



Questions?