# Ch11. 함수

- **함수 사용하기**
  - 함수(function) : 주어진 값에 정해진 처리를 해서 결과를 반환
  - 함수는 처음 한 번만 작성해 놓으면 나중에 필요할 때 계속 불러 쓸 수 있음
  - 지금까지 사용했던 print, input 등도 모두 파이썬에서 미리 만들어 둔 함수임
  - 장점 : 코드의 용도 구분, 코드의 재사용

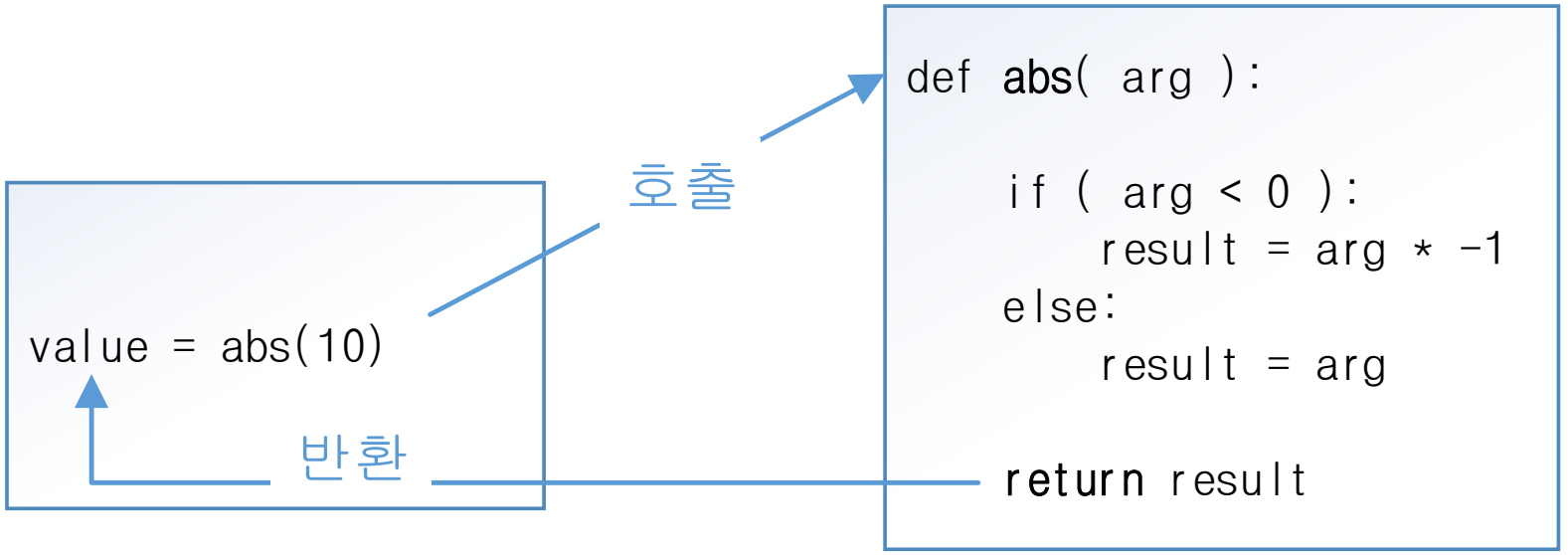
- **함수 용어**
- **정의(Definition)**
  - **어떤 이름을 가진 코드가 구체적으로 어떻게 동작하는지를 "구체적으로 기술"하는 것**
  - 파이썬에서는 함수나 메소드를 정의할 때 **definition(정의)를 줄인 키워드인 def를 사용**

```
>>> def hello():
...     print('Hello, world!')
...
```

• 함수()

```
>>> hello()
Hello, world!
```

- 함수 용어
- 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



- **Hello, world! 출력 함수 만들기**
  - **def에 함수 이름을 지정하고 ( )(괄호)와 :(콜론)을 붙인 뒤**
    **다음 줄에 원하는 코드를 작성함(함수의 이름을 짓는 방법은 변수와 같음)**
  - **코드는 반드시 들여쓰기를 해야 함**(들여쓰기 규칙은 if, for, while과 같음)

```
def 함수이름():
    코드
```

  - def는 정의하다(define)에서 따온 키워드임

```
>>> def hello():
...     print('Hello, world!')
...
```

  - 함수이름 : hello
  - 기능 : print로 ' Hello, world! ' 문자열을 출력

- **함수 호출하기**
  - 함수를 만든 부분 아래에서 hello()와 같이 함수 이름과 ()를 적어주면 함수를 사용할 수 있음

• 함수()

```
>>> hello()
Hello, world!
```

  - 함수를 사용하는 방법 => "함수를 호출(call)한다"

function.py

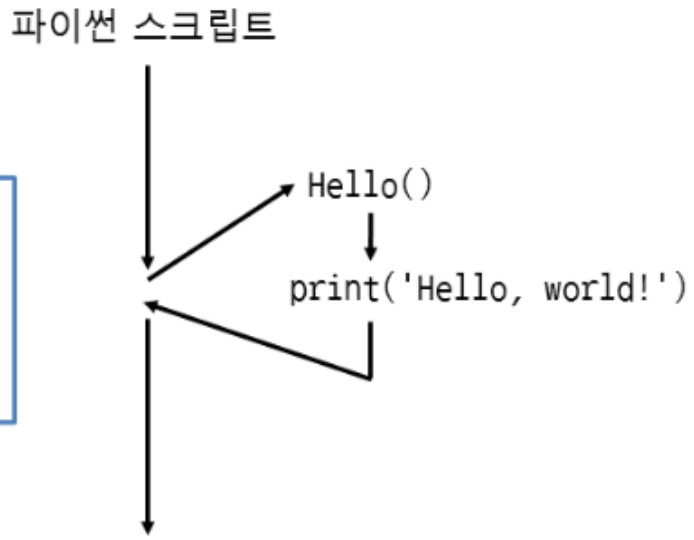
```
def hello():
    print('Hello, world!')

hello()
```

```
Hello, world!
```

- **함수의 실행 순서**
  - hello 함수는 다음과 같은 순서로 실행됨

1. 파이썬 스크립트 최초 실행
2. hello 함수 호출
3. hello 함수 실행
4. print 함수 실행 및 'Hello, world!' 출력
5. hello 함수 종료
6. 파이썬 스크립트 종료

```
①
def hello(): ③
    print('Hello, world!') ④
    ⑤

hello() ②
⑥
```



- **함수 작성과 함수 호출 순서**
  - 함수를 먼저 호출한 뒤 함수를 만들 수는 없음

```
hello()         # hello 함수를 만들기 전에 함수를 먼저 호출

def hello():    # hello 함수를 만듦
    print('Hello, world!')
```

실행 결과

```
Traceback (most recent call last):
  File "C:\project\function.py", line 1, in <module>
    hello()    # hello 함수를 만들기 전에 함수를 먼저 호출
NameError: name 'hello' is not defined
```

  - **함수를 먼저 호출하면 함수가 정의(define)되지 않았다는 에러가 발생**
  - 파이썬 코드는 위에서 아래로 순차적으로 실행되기 때문임
  - 함수를 먼저 만든 뒤에 함수를 호출해야 함

- **Docstring**
  - 함수에 설명문을 추가하는 기능

```
#곱셈을 하는 함수 정의
def mul(a,b):
    '''곱셈을 하는 함수''' #docstring 설정
    return a*b

#정의한 함수 사용
print(mul(2,3))
print(mul(10,3))

help(mul)    #docstring 확인
```

```
6
30
Help on function mul in module __main__:

mul(a, b)
    곱셈을 하는 함수
```

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='₩n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

- **덧셈 함수 만들기**
  - 함수에서 값을 받으려면 ( )(괄호) 안에 변수 이름을 지정해주면 됨
  - 이 변수를 매개변수(parameter)라고 부름
  - **매개(媒介) : 중간에서 둘 사이의 관계를 맺어주는 것을 뜻하는 말**
  - **매개변수 : 호출자와 함수 사이의 관계를 맺어주는 변수를 뜻함**

```
def 함수이름(매개변수1, 매개변수2):
    코드
```

```
>>> def add(a, b):
...     print(a + b)
...
```

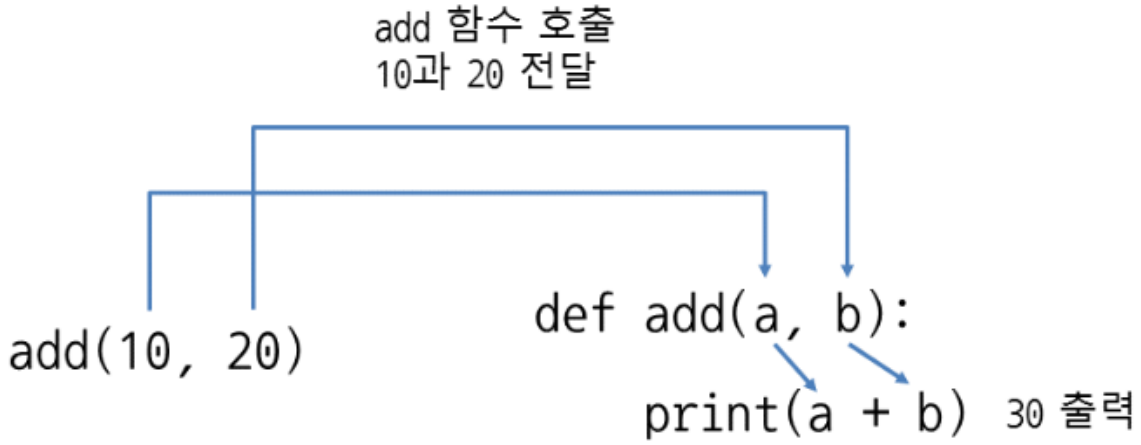
```
>>> add(10, 20)
30
```

  - 함수를 호출할 때 넣는 값을 인수(argument)라고 부름
  - add(10, 20)에서 10과 20이 인수임

- **덧셈 함수 만들기**

```
>>> def add(a, b):
...     print(a + b)
...
```

```
>>> add(10, 20)
30
```



- **함수의 결과를 반환하기**
  - **함수 안에서 return을 사용하면 값을 함수 바깥으로 반환함**
    (return에 값을 지정하지 않으면 None을 반환)

```
def 함수이름(매개변수):
    return 반환값
```

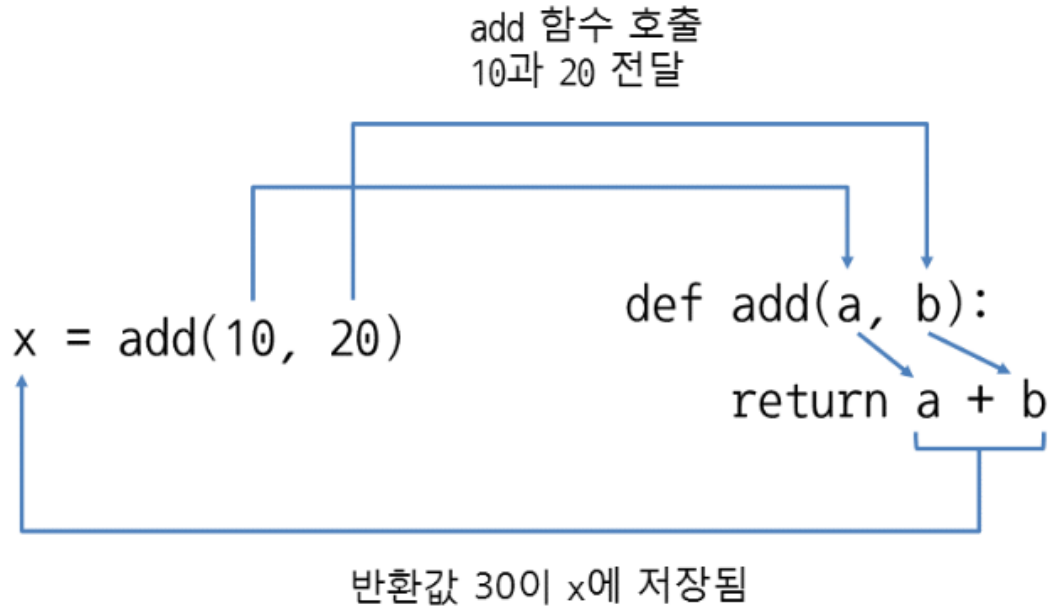
```
>>> def add(a, b):
...     return a + b
...
```

```
>>> x = add(10, 20)
>>> x
30
```

  - **return을 사용하면 값을 함수 바깥으로 반환할 수 있고, 함수에서 나온 값을 변수에 저장할 수 있음**
  - return으로 반환하는 값은 반환값이라고 하며 함수를 호출해준 바깥에 결과를 알려주기 위해 사용함

- **함수의 결과를 반환하기**

```
>>> x = add(10, 20)
>>> x
30
```




– 반환값은 변수에 저장하지 않고 바로 다른 함수에 넣을 수도 있음

```
>>> print(add(10, 20))
30
```

- **함수에서 값을 여러 개 반환하기**
  - **return에 값이나 변수를 ,(콤마)로 구분해서 지정**

```
def 함수이름(매개변수):
    return 반환값1, 반환값2
```

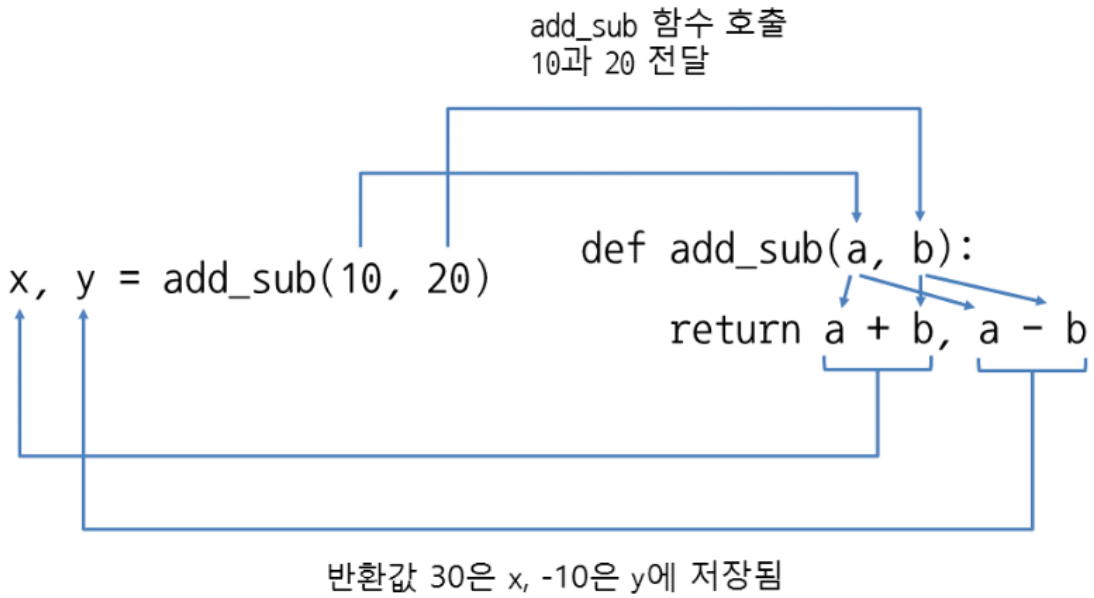
```
>>> def add_sub(a, b):
...     return a + b, a - b
...
```

```
>>> x, y = add_sub(10, 20)
>>> x
30
>>> y
-10
```

- **함수에서 값을 여러 개 반환하기**

```
>>> def add_sub(a, b):
...     return a + b, a - b
...
```

```
>>> x, y = add_sub(10, 20)
>>> x
30
>>> y
-10
```



- **함수에서 값을 여러 개 반환하기**
  - **add_sub의 결과를 변수 한 개에 저장해서 출력해보면 튜플이 반환**

```
>>> x = add_sub(10, 20)
>>> x
(30, -10)
```

  - 튜플이 변수 여러 개에 할당되는 특성을 이용

```
>>> x, y = (30, -10)
>>> x
30
>>> y
-10
```

- **cf) 값 여러 개를 직접 반환**
  - 함수에서 값 여러 개를 직접 반환 하는 경우 : return에 튜플 지정

```
>>> def one_two():
        return (1,2)

>>> one_two()
(1, 2)
```

```
>>> def one_two():
        return 1,2

>>> one_two()
(1, 2)
```

  - 리스트를 직접 반환해도 가능

```
>>> def one_two():
        return [1,2]

>>> x, y = one_two()
>>> print(x,y)
1 2
```

- **함수의 호출 과정 알아보기**
  - 함수 여러 개를 만든 뒤에 각 함수의 호출 과정 : 스택 다이어그램(stack diagram)
  - 스택은 접시 쌓기와 같은데 접시를 차곡차곡 쌓고 꺼낼 때는
    위쪽부터 차례대로 꺼내는 방식임(단, 중간에 있는 접시는 뺄 수 없음)
  - 파이썬에서는 접시 쌓기와 방향이 반대인데, 함수가 아래쪽 방향으로 추가되고
    함수가 끝나면 위쪽 방향으로 사라짐

- **함수의 호출 과정 알아보기**
  - 덧셈 함수 add와 곱셈 함수 mul이 있고, add 함수 안에서 mul 함수를 호출하는 방식으로 만들어져 있음

function_call.py

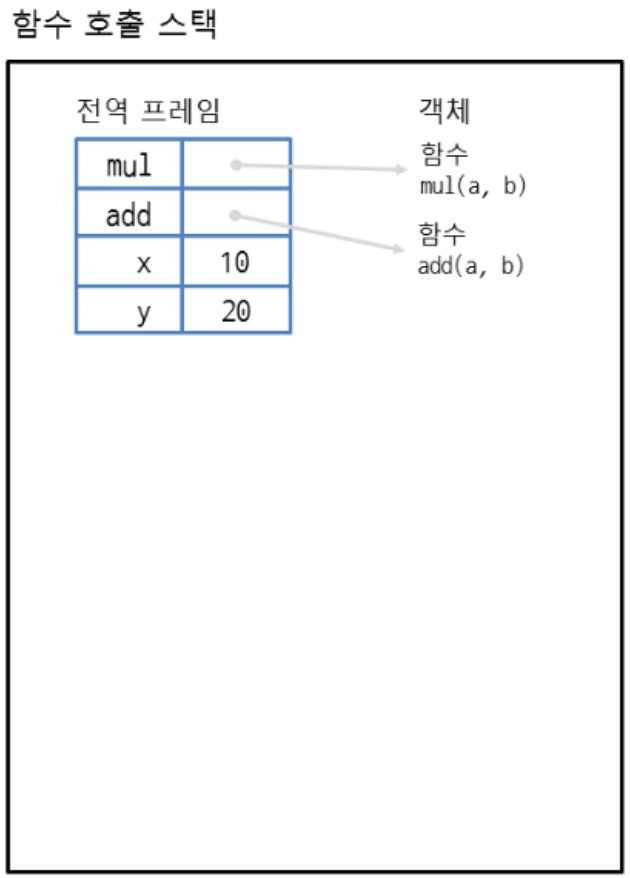
```
def mul(a, b):
    c = a * b
    return c

def add(a, b):
    c = a + b
    print(c)
    d = mul(a, b)
    print(d)

x = 10
y = 20
add(x, y)
```

실행 결과

```
30
200
```

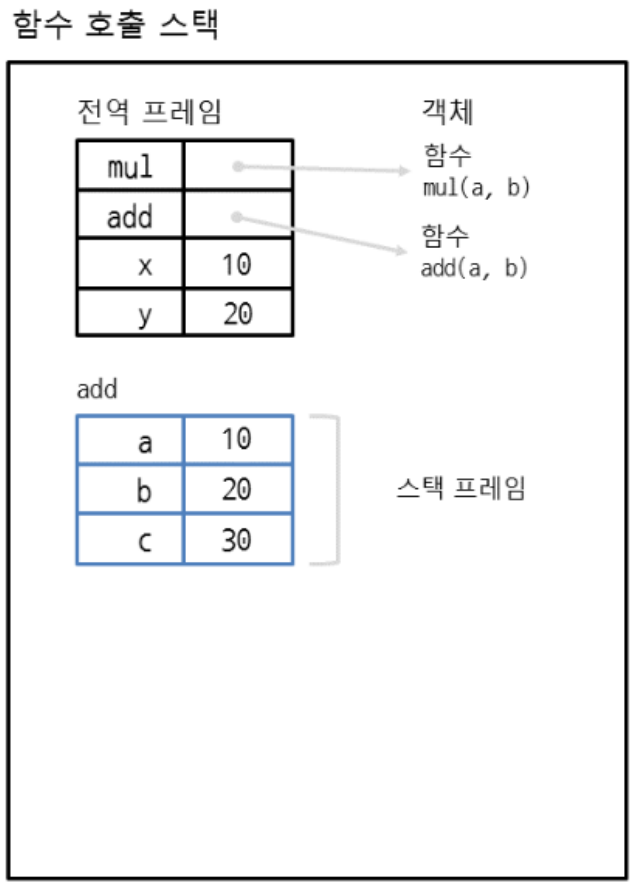
- **함수의 호출 과정 알아보기**

```
1   def mul(a, b):
2       c = a * b
3       return c
4
5   def add(a, b):
6       c = a + b
7       print(c)
8       d = mul(a, b)
9       print(d)
10
11  x = 10
12  y = 20
13  add(x, y)
```



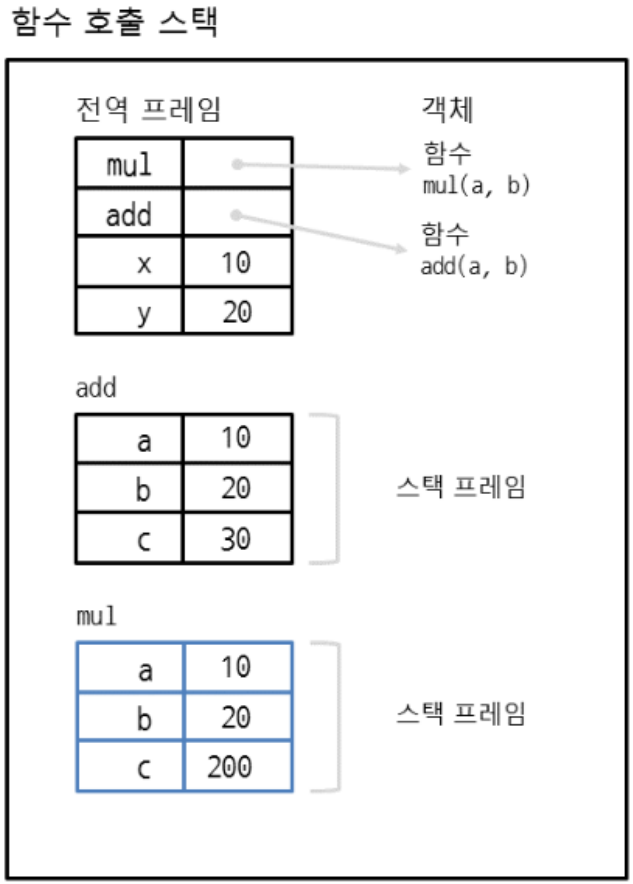
- **함수의 호출 과정 알아보기**

```
1   def mul(a, b):
2       c = a * b
3       return c
4
5   def add(a, b):
6       c = a + b
7       print(c)
8       d = mul(a, b)
9       print(d)
10
11  x = 10
12  y = 20
13  add(x, y)
```



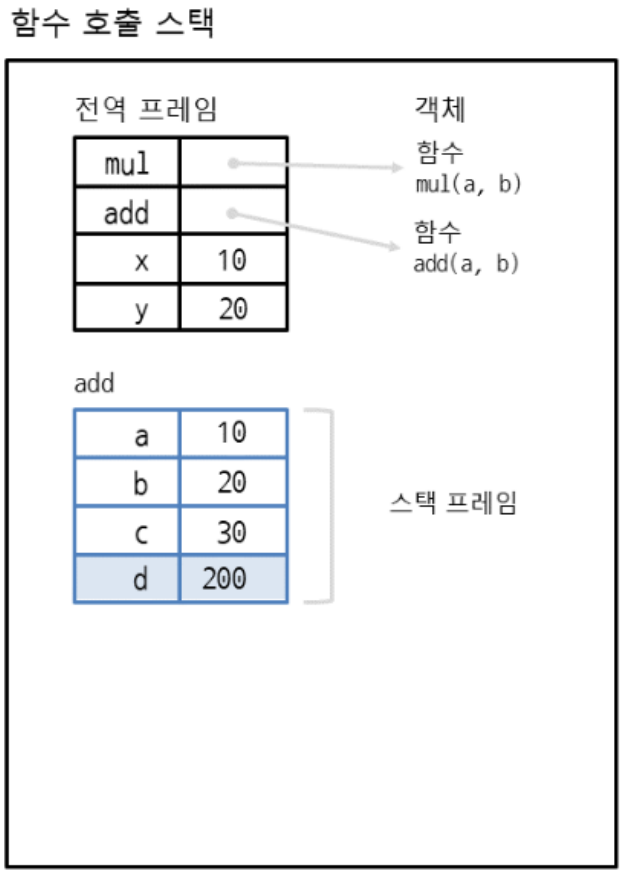
- **함수의 호출 과정 알아보기**

```
1   def mul(a, b):
2       c = a * b
3       return c
4
5   def add(a, b):
6       c = a + b
7       print(c)
8       d = mul(a, b)
9       print(d)
10
11  x = 10
12  y = 20
13  add(x, y)
```



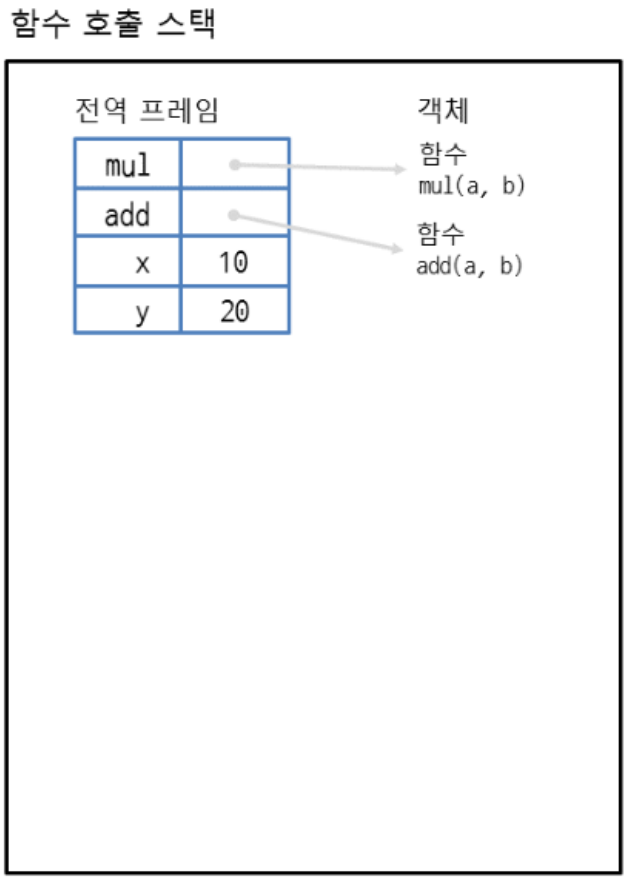
- **함수의 호출 과정 알아보기**

```
1   def mul(a, b):
2       c = a * b
3       return c
4
5   def add(a, b):
6       c = a + b
7       print(c)
8       d = mul(a, b)
9       print(d)
10
11  x = 10
12  y = 20
13  add(x, y)
```



- **함수의 호출 과정 알아보기**

```
1   def mul(a, b):
2       c = a * b
3       return c
4
5   def add(a, b):
6       c = a + b
7       print(c)
8       d = mul(a, b)
9       print(d)
10
11  x = 10
12  y = 20
13  add(x, y)
```



- **함수의 호출 과정 알아보기**
  - 함수는 스택(stack) 방식으로 호출됨
  - 함수를 호출하면 스택의 아래쪽 방향으로 함수가 추가되고
    함수가 끝나면 위쪽 방향으로 사라짐
  - 프레임은 스택 안에 있어서 각 프레임을 스택 프레임이라고 부름
  - 전역 프레임은 스크립트 파일의 실행이 끝나면 사라짐

- **위치 인수와 리스트 언패킹 사용하기**
  - **위치 인수(positional argument) : 함수에 인수를 순서대로 넣는 방식**
  - 인수의 위치가 정해져 있음

```
>>> print(10, 20, 30)
10 20 30
```

- **위치 인수를 사용하는 함수를 만들고 호출하기**
  - 숫자 세 개를 각 줄에 출력하는 함수

```
>>> def print_numbers(a, b, c):
...     print(a)
...     print(b)
...     print(c)
...
```

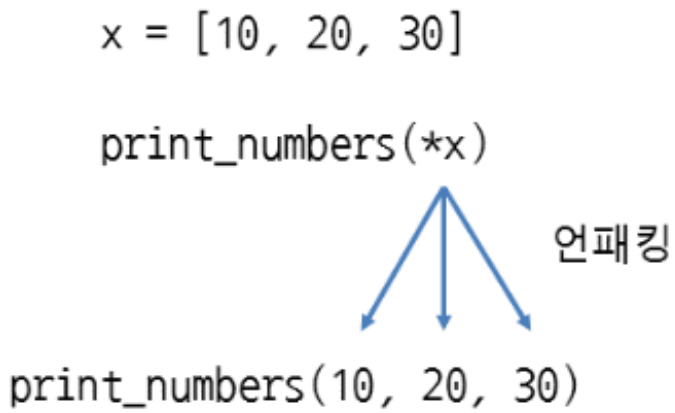
```
>>> print_numbers(10, 20, 30)
10
20
30
```

- **언패킹 사용하기**
  - 인수를 순서대로 넣을 때는 리스트나 튜플을 사용할 수도 있음
  - 리스트 또는 튜플 앞에 *(애스터리스크)를 붙여서 함수에 넣음

- 함수(*리스트)
- 함수(*튜플)

```
>>> x = [10, 20, 30]
>>> print_numbers(*x)
10
20
30
```

  - 리스트(튜플) 앞에 *를 붙이면 언패킹(unpacking)이 되어서 print_numbers(10, 20, 30)과 똑같은 동작이 됨
  - **언패킹 : 리스트의 포장을 푼다**



- **언패킹 사용하기**
  - 리스트 변수 대신 리스트 앞에 바로 *를 붙여도 동작은 같음

```
>>> print_numbers(*[10, 20, 30])
10
20
30
```

  - **단, 이때 함수의 매개변수 개수와 리스트의 요소 개수는 같아야 함**
  - **만약 개수가 다르면 함수를 호출할 수 없음**
  - **def print_numbers(a, b, c):**로 만들었으므로 리스트에는 **요소를 3개 넣어야 함**
  - **요소가 두 개인 리스트를 넣으면 에러가 발생함**

```
>>> print_numbers(*[10, 20])
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    print_numbers(*[10, 20])
TypeError: print_numbers() missing 1 required positional argument: 'c'
```

- **가변 인수 함수 만들기**
  - 위치 인수와 리스트 언패킹은 인수의 개수가 정해지지 않은 가변 인수(variable argument)에 사용함
  - 가변 인수 함수는 매개변수 앞에 *를 붙여서 만듬
  - 상황에 따라 매개변수의 수가 달라짐

```
def 함수이름(*매개변수):
    코드
```

```
>>> def print_numbers(*args):
...     for arg in args:
...         print(arg)
...
```

- **가변 인수 함수 만들기**
  - 넣은 숫자 개수만큼 출력됨

```
>>> print_numbers(10)
10
>>> print_numbers(10, 20, 30, 40)
10
20
30
40
```

  - 함수에 인수 여러 개를 직접 넣어도 되고, 리스트(튜플) 언패킹을 사용해도 됨
  - 숫자가 들어있는 리스트를 만들고 앞에 *를 붙여서 넣음

```
>>> x = [10]
>>> print_numbers(*x)
10
>>> y = [10, 20, 30, 40]
>>> print_numbers(*y)
10
20
30
40
```

- **가변 인수 함수 만들기**
  - **제약**
  - **가변 매개변수 뒤에는 일반 매개변수 올 수 없음**
  - **가변 매개변수는 하나만 사용할 수 있음**

```
def print_n_times(n, *values):
#n번 반복
    for i in range(n):
        #values는 리스트 처럼 활용
        for value in values:
            print(value)

    print()#줄바꿈

print_n_times(3,'안녕하세요','즐거운','파이썬 프로그래밍')
```

```
안녕하세요
즐거운
파이썬 프로그래밍
안녕하세요
즐거운
파이썬 프로그래밍
안녕하세요
즐거운
파이썬 프로그래밍
```

- **가변 인수 함수 만들기**
  - **을 사용하면 딕셔너리로 저장 됩니다

```
def dic_print(**x):
    print(x)

dic_print(one=1,two=2,three=3,four=4)
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

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

```
알리송=GK
메시=FW
손흥민=FW
반다이크=DF
```

- **매개변수에 초기값 지정하기 (기본 매개변수)**

```
def 함수이름(매개변수=값):
    코드
```

  - 매개변수 초기값 : 주 사용하는 값이 있으면서 가끔 다른 값을 사용해야 할 때 활용

```
>>> def personal_info(name, age, address='비공개'):
...     print('이름: ', name)
...     print('나이: ', age)
...     print('주소: ', address)
...
```

```
>>> personal_info('홍길동', 30)
이름:  홍길동
나이:  30
주소:  비공개
```

```
>>> personal_info('홍길동', 30, '서울시 용산구 이촌동')
이름:  홍길동
나이:  30
주소:  서울시 용산구 이촌동
```

- **초기값이 지정된 매개변수의 위치 (기본 매개변수)**
  - 매개변수의 초기값을 지정할 때 **주의할 점은 초기값이 지정된 매개변수 다음에는 초기값이 없는 매개변수가 올 수 없음**
  - address를 두 번째 매개변수로 만들고, 그 다음에 초기값을 지정하지 않은 age가 오도록 생성

```
>>> def personal_info(name, address='비공개', age):
...     print('이름: ', name)
...     print('나이: ', age)
...     print('주소: ', address)
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

  - 함수를 만들어보면 문법 에러가 발생함
  - 다음과 같이 **초기값이 지정된 매개변수는 뒤쪽에 몰아주면 됨**

```
def personal_info(name, age, address='비공개'):
def personal_info(name, age=0, address='비공개'):
def personal_info(name='비공개', age=0, address='비공개'):
```

- **cf) 초기값이 지정된 매개변수 (기본 매개변수)**
  - **기본 매개변수 뒤에 일반 매개변수가 오지 못하게 막은 이유**

```
def print_n_times(value, n=2):
#n번 반복
    for i in range(n):
        print(value)


print_n_times("안녕하세요")
```

```
안녕하세요
안녕하세요
```

  - **만약 여기서 print_n_times(n=2, value)형태로 사용 가능하다면, print_n_times("안녕하세요")라고 입력했을때 "안녕하세요"라는 글자가 첫번째 매개변수에 할당 되어야 하는지 두번째 매개변수에 할당 되어야 하는지 확실하게 알수가 없음**
  - **이에 파이선에서는 내부적으로 기본 매개변수 뒤에 일반 매개변수를 오지 못하게 막음**

- **cf) 기본 매개 변수가 가변 매개변수보다 앞에 올때**

```
def print_n_times(n=2, *values):
#n번 반복
    for i in range(n):
        #values는 리스트 처럼 활용
        for value in values:
            print(value)

    print()#줄바꿈

print_n_times('안녕하세요','즐거운','파이썬 프로그래밍')
```

```
Traceback (most recent call last):
  File "F:/20190828 이현식 님 데이터/DATA/private/python/201
", line 31, in <module>
    print_n_times('안녕하세요','즐거운','파이썬 프로그래밍')
  File "F:/20190828 이현식 님 데이터/DATA/private/python/201
", line 24, in print_n_times
    for i in range(n):
TypeError: 'str' object cannot be interpreted as an integer
```

  - 매개변수가 순서대로 입력 되므로 n에는 '안녕하세요', values에는 ['즐거운', '파이썬 프로그래밍'] 이 들어감
  - 기본 매개변수는 가변 매개 변수 앞에 사용해도 의미 없음

- **cf) 가변 매개변수가 기본 매개변수보다 앞에 올때**

```
def print_n_times(*values, n=2):
#n번 반복
    for i in range(n):
        #values는 리스트 처럼 활용
        for value in values:
            print(value)

    print()#줄바꿈

print_n_times('안녕하세요','즐거운','파이썬 프로그래밍',3)
```

```
안녕하세요
즐거운
파이썬 프로그래밍
3
안녕하세요
즐거운
파이썬 프로그래밍
3
```

  - **가변 매개변수가 우선**

- **키워드 인수 사용하기**
  - ex) 개인 정보 출력 함수 생성

```
>>> def personal_info(name, age, address):
...     print('이름: ', name)
...     print('나이: ', age)
...     print('주소: ', address)
...
```

```
>>> personal_info('홍길동', 30, '서울시 용산구 이촌동')
이름:  홍길동
나이:  30
주소:  서울시 용산구 이촌동
```

- **키워드 인수 사용하기**
  - **키워드 인수(keyword argument) : 인수의 이름을 지정해서 함수를 호출**
    - 함수(키워드=값)
  - personal_info 함수의 키워드 인수 호출

```
>>> personal_info(name='홍길동', age=30, address='서울시 용산구 이촌동')
이름:  홍길동
나이:  30
주소:  서울시 용산구 이촌동
```

  - **키워드 인수를 사용하니 함수를 호출할 때 인수의 용도가 명확하게 보임**
  - **매개변수가 많은 경우 호출자가 매개변수의 이름을 일일이 지정하여 입력**
  - **인수의 순서를 맞추지 않아도 키워드에 해당하는 값이 들어감**

```
>>> personal_info(age=30, address='서울시 용산구 이촌동', name='홍길동')
이름:  홍길동
나이:  30
주소:  서울시 용산구 이촌동
```

- **키워드 인수와 딕셔너리 언패킹 사용하기**
  - **딕셔너리 앞에 **(애스터리스크 두 개)를 붙여서 함수에 입력**

• 함수(**딕셔너리)

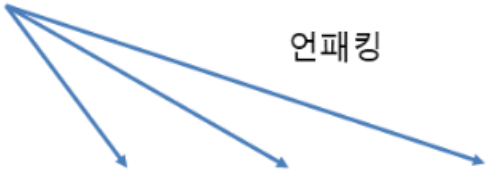
```
>>> def personal_info(name, age, address):
...     print('이름: ', name)
...     print('나이: ', age)
...     print('주소: ', address)
...
```

  - 딕셔너리에 '키워드': 값 형식으로 인수를 저장, 앞에 **를 붙여서 함수에 입력
  - **딕셔너리의 키워드(키)는 문자열 형태**

```
>>> x = {'name': '홍길동', 'age': 30, 'address': '서울시 용산구 이촌동'}
>>> personal_info(**x)
이름:  홍길동
나이:  30
주소:  서울시 용산구 이촌동
```

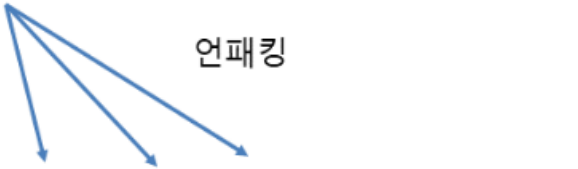
- **키워드 인수와 딕셔너리 언패킹 사용하기**

```
x = {'name': '홍길동', 'age': 30, 'address': '서울시 용산구 이촌동'}

personal_info(**x)
```




```
personal_info(name='홍길동', age=30, address='서울시 용산구 이촌동')
```

```
personal_info(**x)
```




```
personal_info('홍길동', 30, '서울시 용산구 이촌동')
```

- **키워드 인수와 딕셔너리 언패킹 사용하기**
  - 딕셔너리 변수 대신 딕셔너리 앞에 바로 **를 붙여도 동작은 같음

```
>>> personal_info(**{'name': '홍길동', 'age': 30, 'address': '서울시 용산구 이촌동'})
이름:  홍길동
나이:  30
주소:  서울시 용산구 이촌동
```

  - **언패킹을 사용할 때 함수의 매개변수 이름과 딕셔너리의 키 이름이 같아야 함**
  - **매개변수 개수 == 딕셔너리 키의 개수**
  - **이름과 개수가 다르면 함수를 호출할 수 없음**
  - 함수를 def personal_info(name, age, address):로 만들었으므로 딕셔너리도 똑같이 맞춰주어야 함

```
>>> personal_info(**{'name': '홍길동', 'old': 30, 'address':'서울시 용산구 이촌동'})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: personal_info() got an unexpected keyword argument 'old'
>>> personal_info(**{'name': '홍길동', 'age': 30})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: personal_info() missing 1 required positional argument: 'address'
```

- **를 두 번 사용하는 이유
  - 딕셔너리는 **처럼 *를 두 번 사용하는 이유 딕셔너리는 키-값 쌍 형태로 값이 저장
  - *를 한 번만 사용해서 함수를 호출하는 경우

```
>>> x = {'name': '홍길동', 'age': 30, 'address': '서울시 용산구 이촌동'}
>>> personal_info(*x)
이름:  name
나이:  age
주소:  address
```

  - 딕셔너리를 한 번 언패킹하면 키를 사용한다는 뜻
  - **처럼 딕셔너리를 두 번 언패킹하여 값을 사용

```
>>> x = {'name': '홍길동', 'age': 30, 'address': '서울시 용산구 이촌동'}
>>> personal_info(**x)
이름:  홍길동
나이:  30
주소:  서울시 용산구 이촌동
```

- **키워드 인수를 사용하는 가변 인수 함수 만들기**
  - **키워드 인수를 사용하는 가변 인수 함수는 매개변수 앞에 **를 붙여서 생성**

```
def 함수이름(**매개변수):
    코드
```

```
>>> def personal_info(**kwargs):
...     for kw, arg in kwargs.items():
...         print(kw, ': ', arg, sep='')
...
```

  - 매개변수 이름은 원하는 대로 지어도 되지만 관례적으로 keyword arguments를 줄여서 **kwargs로 사용함**
  - **kwargs는 딕셔너리라서 for로 반복할 수 있음**

- **키워드 인수를 사용하는 가변 인수 함수 만들기**
  - personal_info 함수에 키워드와 값을 넣어서 실행해보고 값을 한 개 넣어도 되고, 세 개 넣어도 됨

```
>>> personal_info(name='홍길동')
name: 홍길동
>>> personal_info(name='홍길동', age=30, address='서울시 용산구 이촌동')
name: 홍길동
age: 30
address: 서울시 용산구 이촌동
```

  - 인수를 직접 넣어도 되고, 딕셔너리 언패킹을 사용해도 됨

```
>>> x = {'name': '홍길동'}
>>> personal_info(**x)
name: 홍길동
>>> y = {'name': '홍길동', 'age': 30, 'address': '서울시 용산구 이촌동'}
>>> personal_info(**y)
name: 홍길동
age: 30
address: 서울시 용산구 이촌동
```

- **키워드 인수를 사용하는 가변 인수 함수 만들기**
  - 함수를 만들 때 def personal_info(**kwargs):와 같이 매개변수에 **를 붙여주면 키워드 인수를 사용하는 가변 인수 함수를 만들 수 있음
  - 함수를 호출할 때는 키워드와 인수를 각각 넣거나 딕셔너리 언패킹을 사용하면 됨
  - 보통 **kwargs를 사용한 가변 인수 함수는 다음과 같이 함수 안에서 특정 키가 있는지 확인한 뒤 해당 기능을 만듦

```
def personal_info(**kwargs):
    if 'name' in kwargs:    # in으로 딕셔너리 안에 특정 키가 있는지 확인
        print('이름: ', kwargs['name'])
    if 'age' in kwargs:
        print('나이: ', kwargs['age'])
    if 'address' in kwargs:
        print('주소: ', kwargs['address'])
```

- **cf) 고정 인수와 가변 인수(키워드 인수) 함께 사용하기**
  - **고정 매개변수를 먼저 지정, 이후 매개변수에 **를 붙임**

```
>>> def personal_info(name, **kwargs):
        print(name)
        print(kwargs)

>>> personal_info('이현식')
이현식
{}
>>> personal_info('이현식', age = 30, address='강원도 춘천시 효자동')
이현식
{'age': 30, 'address': '강원도 춘천시 효자동'}
>>> personal_info(**{'name' : '이현식', 'age' : 30, 'addres' : '강원도 춘천시 효자동'})
이현식
{'age': 30, 'addres': '강원도 춘천시 효자동'}
```

  - **주의점 : **kwargs가 고정 매개변수 보다 앞에 오면 안됨, 가장 뒤에 있어야함**

```
>>> def personal_info(**kwargs, name):
        print(kwargs)

SyntaxError: invalid syntax
```

- cf) 위치 인수와 가변 인수(키워드 인수) 함께 사용하기
  - 위치 인수를 받는 *args와 키워드 인수를 받는 **kwargs를 함께 사용 가능
  - Ex) print(위치인수, 키워드 인수)

```
>>> def test_print(*args, **kwargs):
        print(*args, **kwargs)


>>> test_print(1,2,3, sep=' : ', end=' ' )
1 : 2 : 3
```

  - 고정 매개변수, *args, **kwargs를 함께 사용하는 경우 순서는 고정 매개변수, *args, **kwargs순으로 지정

- **함수에서 재귀호출 사용하기**
  - **재귀호출(recursive call) : 함수 안에서 함수 자기자신을 호출하는 방식**
  - 재귀호출은 일반적인 상황에서는 잘 사용하지 않지만 알고리즘을 구현할 때 매우 유용함
  - 보통 알고리즘에 따라서 반복문으로 구현한 코드보다 재귀호출로 구현한 코드가 좀 더 직관적이고 이해하기 쉬운 경우가 많음

- **함수에서 재귀호출 사용하기**

recursive_function_error.py

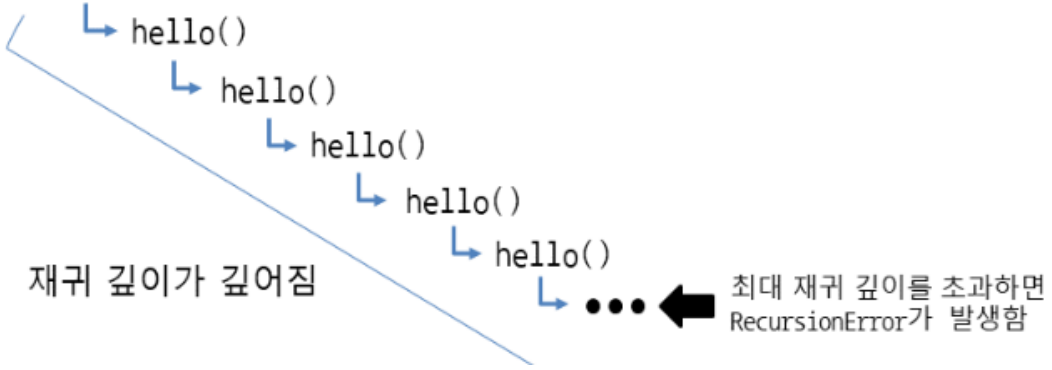
```
def hello():
    print('Hello, world!')
    hello()

hello()
```

실행 결과

```
Hello, world!
Hello, world!
Hello, world!
...(생략)
Traceback (most recent call last):
  File "C:\project\recursive_function_error.py", line 5, in <module>
    hello()
  File "C:\project\recursive_function_error.py", line 3, in hello
    hello()
  File "C:\project\recursive_function_error.py", line 3, in hello
    hello()
  File "C:\project\recursive_function_error.py", line 3, in hello
    hello()
  [Previous line repeated 974 more times]
  File "C:\project\recursive_function_error.py", line 2, in hello
    print('Hello, world!')
RecursionError: maximum recursion depth exceeded while pickling an object
```

- **재귀호출 사용하기**
  - hello 함수가 자기자신을 계속 호출하다가 최대 재귀 깊이를 초과하면 **RecursionError가 발생함**

```
def hello():
    print('Hello, world!')
    hello()
```



- **재귀호출에 종료 조건 만들기**

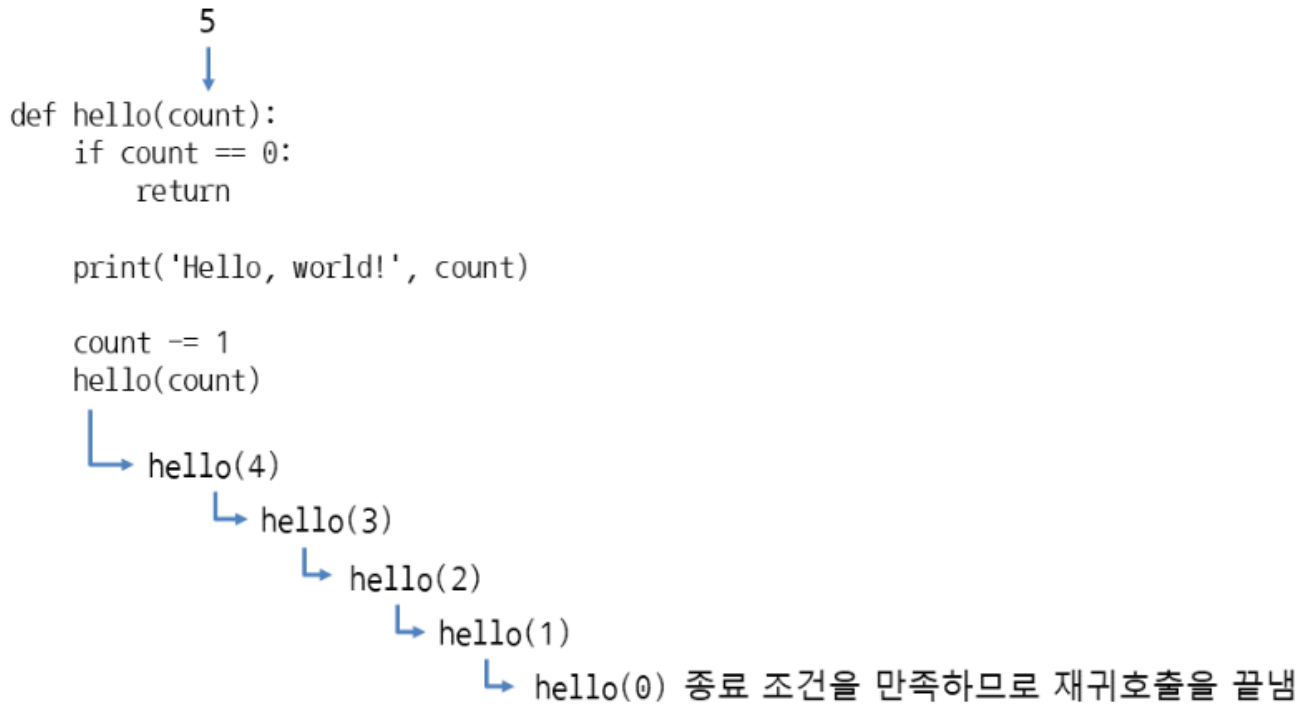
recursive_function_exit_condition.py

```
def hello(count):
    if count == 0:    # 종료 조건을 만듦. count가 0이면 다시 hello 함수를 호출하지 않고 끝냄
        return

    print('Hello, world!', count)

    count -= 1      # count를 1 감소시킨 뒤
    hello(count)    # 다시 hello에 넣음

hello(5)    # hello 함수 호출
```

실행 결과

```
Hello, world! 5
Hello, world! 4
Hello, world! 3
Hello, world! 2
Hello, world! 1
```

- **재귀호출에 종료 조건 만들기**



- **재귀호출로 팩토리얼 구하기**
  - 팩토리얼은 1부터 n까지 양의 정수를 차례대로 곱한 값이며 !(느낌표) 기호로 표기함

factorial.py

```
def factorial(n):
    if n == 1:      # n이 1일 때
        return 1    # 1을 반환하고 재귀호출을 끝냄
    return n * factorial(n - 1)    # n과 factorial 함수에 n - 1을 넣어서 반환된 값을 곱함

print(factorial(5))
```

실행 결과

```
120
```

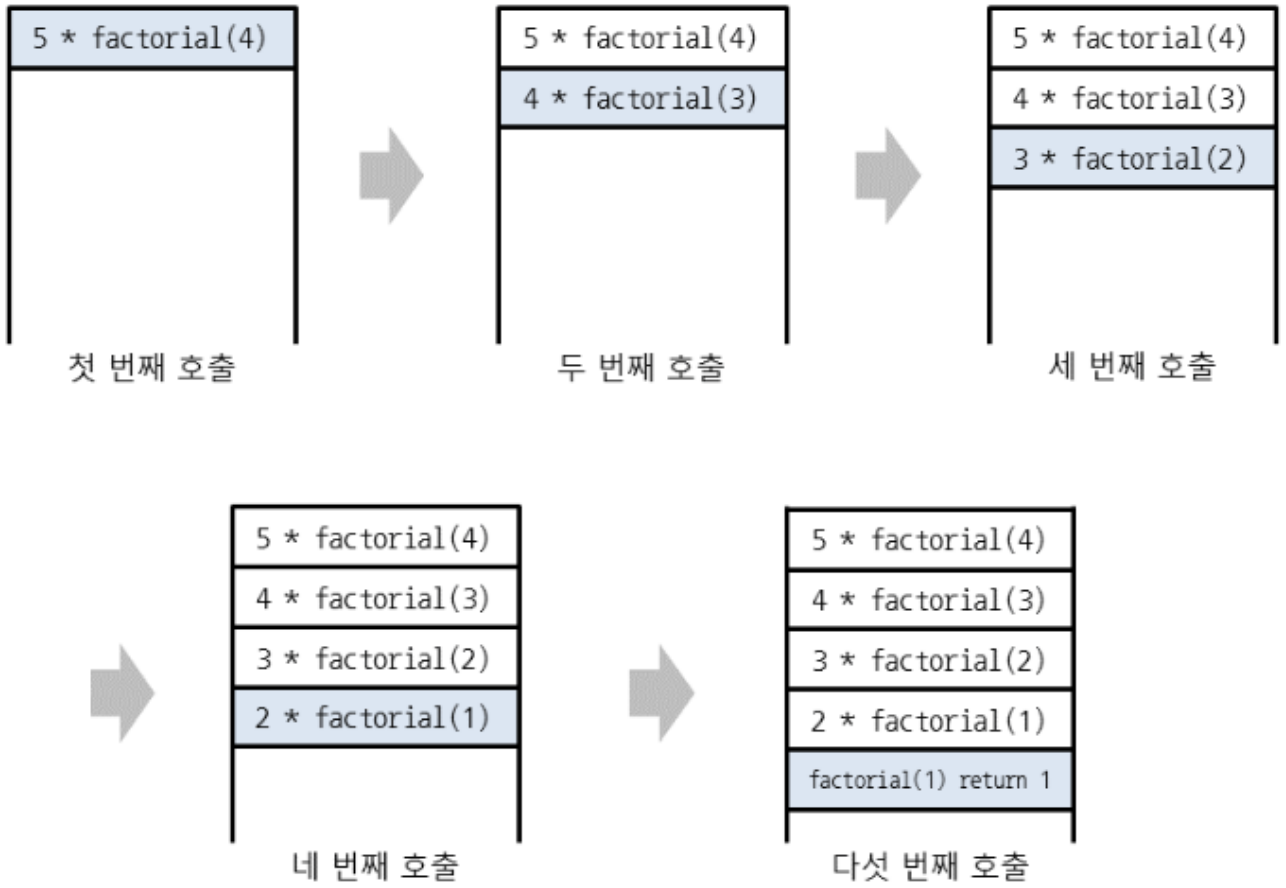
- **재귀호출로 팩토리얼 구하기**

```
def factorial(n):
    if n == 1:      # n이 1일 때
        return 1    # 1을 반환하고 재귀호출을 끝냄
```

  - factorial 함수의 핵심은 반환값 부분임
  - 계산 결과가 즉시 구해지는 것이 아니라 재귀호출로 n - 1을 계속 전달하다가 n이 1일 때 비로소 1을 반환하면서 n과 곱하고 다시 결과값을 반환함
  - 그 뒤 n과 반환된 결과값을 곱하여 다시 반환하는 과정을 반복함

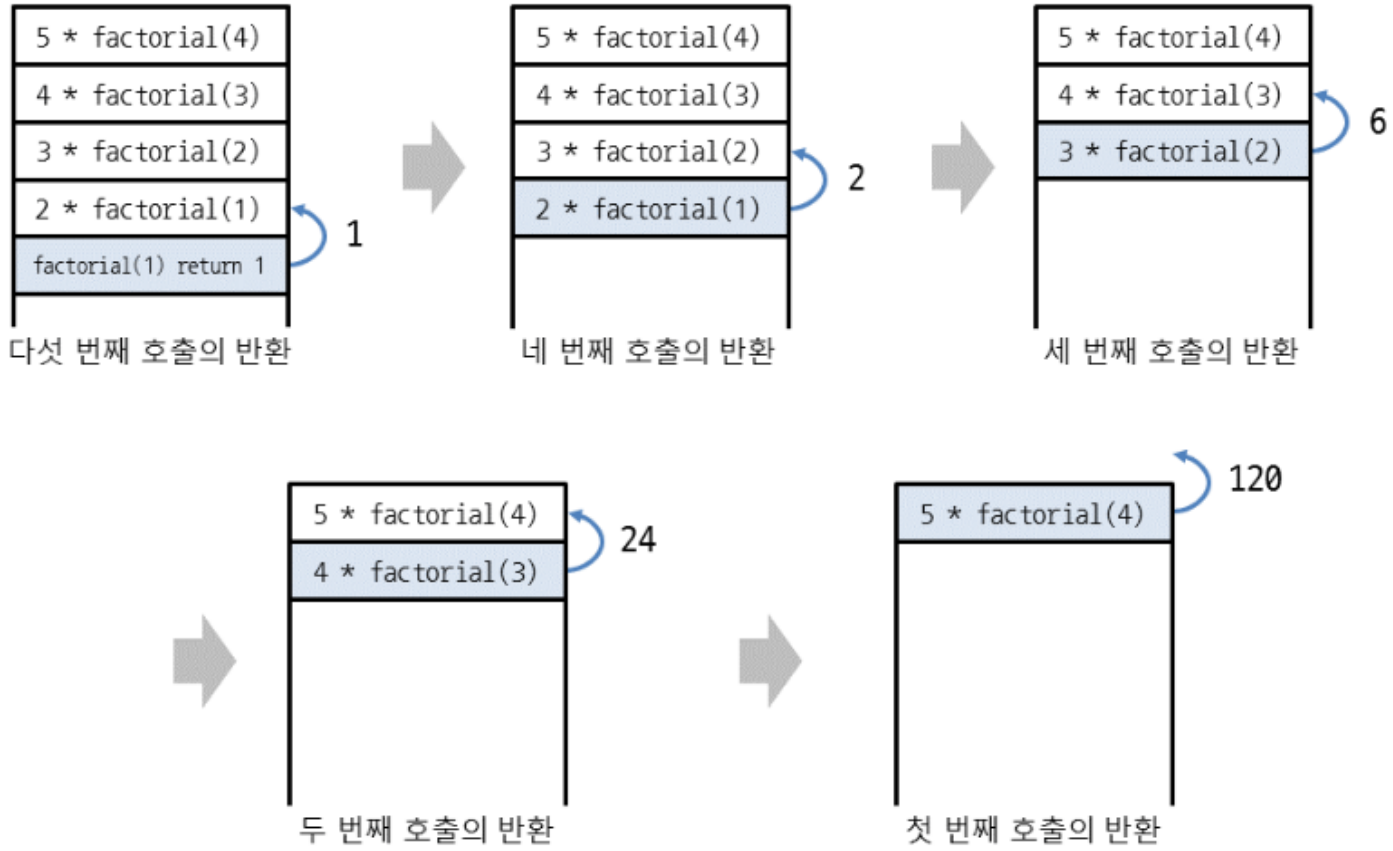
```
    return n * factorial(n - 1)    # n과 factorial 함수에 n - 1을 넣어서 반환된 값을 곱함
```

- **재귀호출로 팩토리얼 구하기**



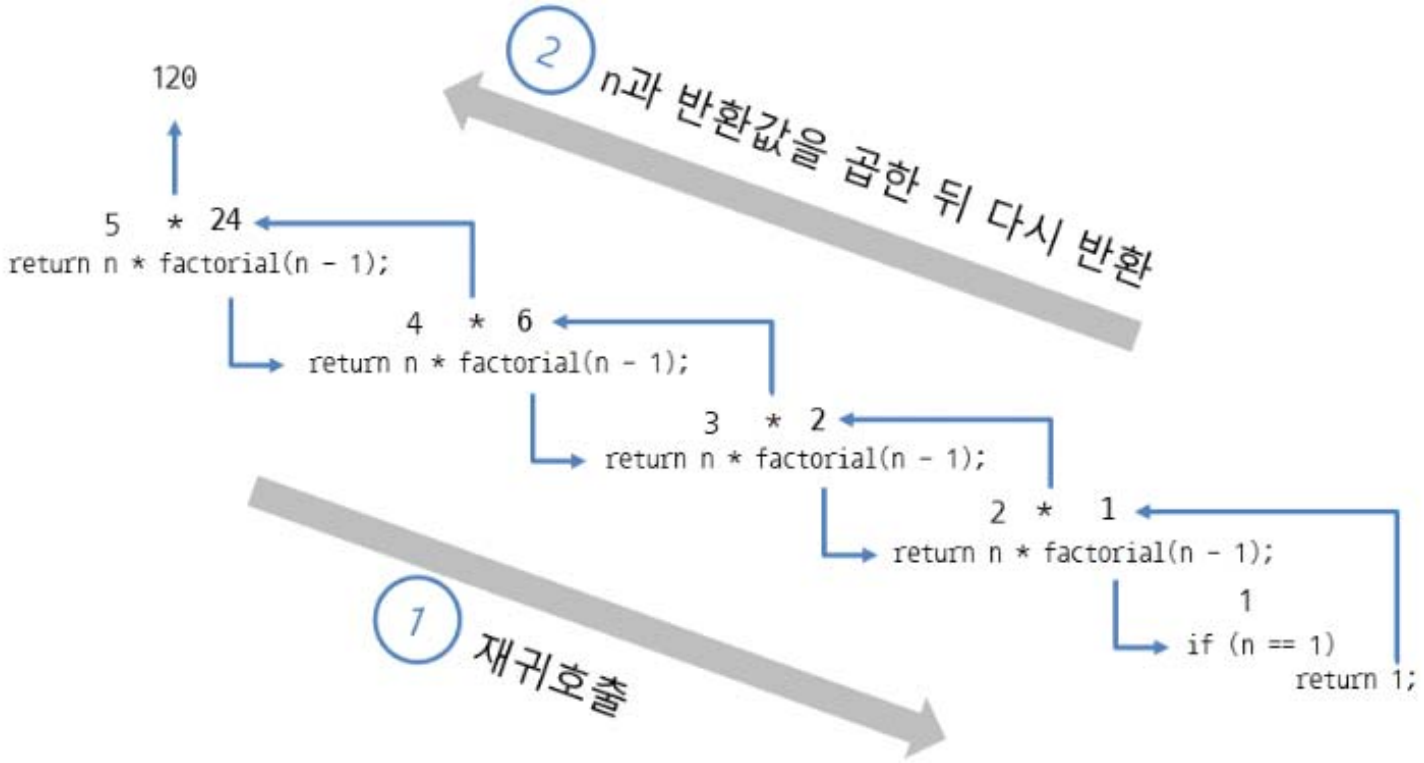

**factorial 함수의 호출**

- **재귀호출로 팩토리얼 구하기**




**factorial 함수의 반환**

- **재귀호출로 팩토리얼 구하기**




**factorial 함수의 호출 순서와 계산 과정**

- **변수의 사용 범위 알아보기**

global_variable.py

```
x = 10          # 전역 변수
def foo():
    print(x)    # 전역 변수 출력

foo()
print(x)        # 전역 변수 출력
```

실행 결과

```
10
10
```

  - **전역 변수(global variable) : 함수 바깥에 존재하며 여러 함수가 공통으로 이용하는 변수**
  - 전역 변수에 접근할 수 있는 범위를 전역 범위(global scope)라고 함

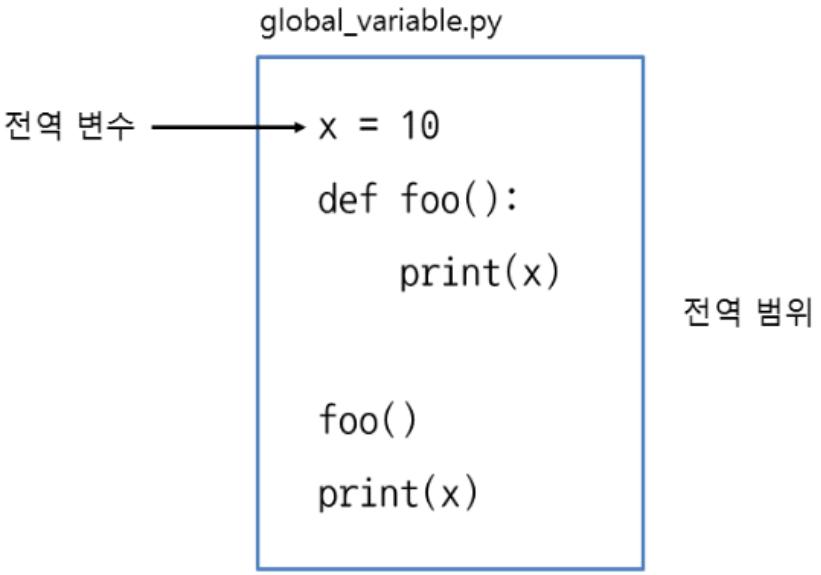
- **변수의 사용 범위 알아보기**

global_variable.py

```
x = 10          # 전역 변수
def foo():
    print(x)    # 전역 변수 출력

foo()
print(x)        # 전역 변수 출력
```

실행 결과

```
10
10
```




- **전역 변수(global variable) : 함수 바깥에 존재하며 여러 함수가 공통으로 이용하는 변수**
- 전역 변수에 접근할 수 있는 범위를 전역 범위(global scope)라고 함

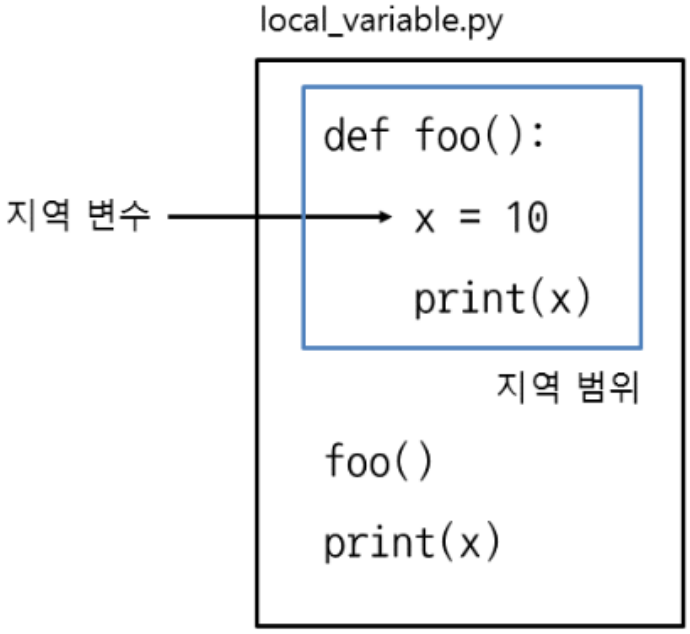
- **변수의 사용 범위 알아보기**

local_variable.py

```
def foo():
    x = 10      # foo의 지역 변수
    print(x)    # foo의 지역 변수 출력

foo()
print(x)        # 에러. foo의 지역 변수는 출력할 수 없음
```

실행 결과

```
10
Traceback (most recent call last):
  File "C:\project\local_variable.py", line 6, in <module>
    print(x)        # 에러. foo의 지역 변수는 출력할 수 없음
NameError: name 'x' is not defined
```




  - **지역 변수(Local value) : 함수 안에서 사용하는 변수, 해당 함수 안에서만 유효**
  - 실행을 해보면 x가 정의되지 않았다는 에러가 발생
    => 변수 x는 함수 foo 안에서 만들었기 때문에 foo의 지역 변수(local variable)임
  - 지역 변수를 접근할 수 있는 범위를 지역 범위(local scope)라고 함

- **함수 안에서 전역 변수 변경하기**

global_local_variable.py

```
x = 10          # 전역 변수
def foo():
    x = 20      # x는 foo의 지역 변수
    print(x)    # foo의 지역 변수 출력

foo()
print(x)        # 전역 변수 출력
```

실행 결과

```
20
10
```

  - 전역 변수 x가 있고, foo에서 지역 변수 x를 새로 만들게 됨
  - 이 둘은 이름만 같을 뿐 서로 다른 변수임

- **함수 안에서 전역 변수 변경하기**
  - 함수 안에서 **전역 변수의 값을 변경하려면 global 키워드를 사용**

• global 전역변수

function_global_keyword.py

```
x = 10          # 전역 변수
def foo():
    global x    # 전역 변수 x를 사용하겠다고 설정
    x = 20      # x는 전역 변수
    print(x)    # 전역 변수 출력

foo()
print(x)        # 전역 변수 출력
```

실행 결과

```
20
20
```

- **함수 안에서 전역 변수 변경하기**
  - 전역 변수가 없을 때 함수 안에서 global을 사용하면 해당 변수는 전역 변수가 됨

```
# 전역 변수 x가 없는 상태
def foo():
    global x    # x를 전역 변수로 만듦
    x = 20      # x는 전역 변수
    print(x)    # 전역 변수 출력

foo()
print(x)        # 전역 변수 출력
```

- **함수 안에서 함수 만들기**
  - **def로 함수를 만들고 그 안에서 다시 def로 함수를 생성**

```
def 함수이름1():
    코드
    def 함수이름2():
        코드
```

  - 간단하게 함수 안에서 문자열을 출력하는 함수를 만들고 호출

function_in_function.py

```
def print_hello():
    hello = 'Hello, world!'
    def print_message():
        print(hello)
    print_message()

print_hello()
```

실행 결과

```
Hello, world!
```

- **함수 안에서 함수 만들기 (중첩 함수)**
  - **중첩 함수는 자신이 소속되어 있는 함수의 매개 변수에 접근 가능**

```
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)

print(stddev(2.3, 1.7, 1.4, 0.7, 1.9))
```

```
0.6
```

- **pass**
  - pass키워드는 함수나 클래스의 구현을 미룰 때 사용

```
def add_fun()
    pass
```

- **지역 변수 변경하기**
  - 안쪽 함수 B에서 바깥쪽 함수 A의 지역 변수 x를 변경해보자

function_local_error.py

```
def A():
    x = 10        # A의 지역 변수 x
    def B():
        x = 20    # x에 20 할당

    B()
    print(x)      # A의 지역 변수 x 출력

A()
```

실행 결과

```
10
```

  - 파이썬에서는 함수에서 변수를 만들면 항상 현재 함수의 지역 변수가 됨

```
def A():
    x = 10        # A의 지역 변수 x
    def B():
        x = 20    # B의 지역 변수 x를 새로 만듦
```