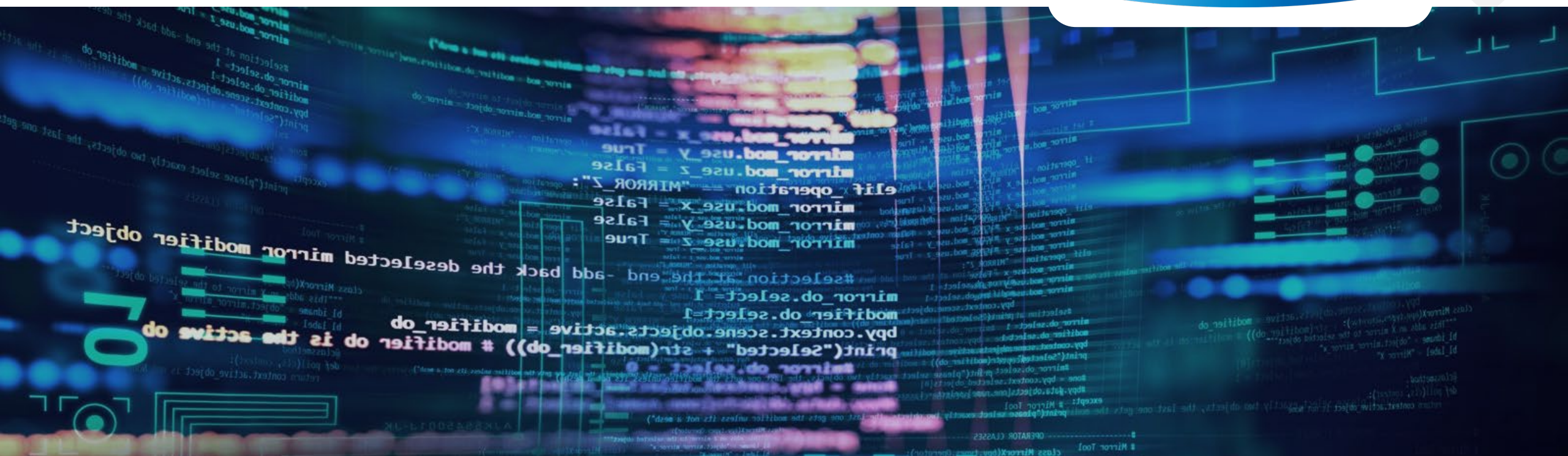


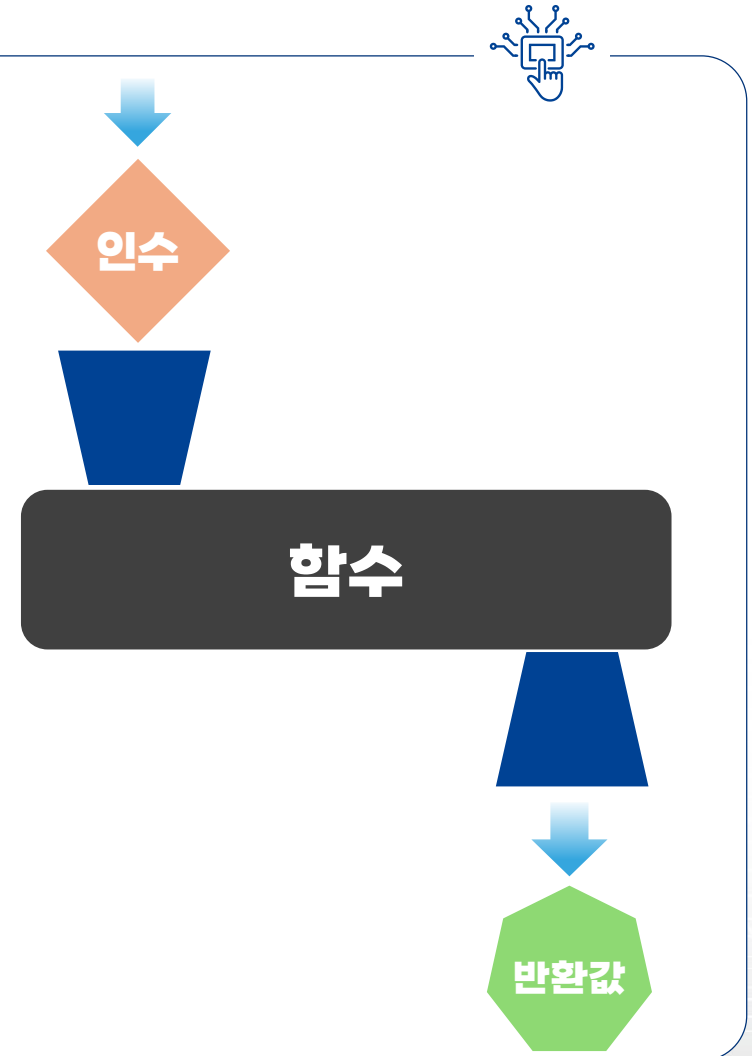
Lecture 6

함수:매개변수와 반환값

KAIST



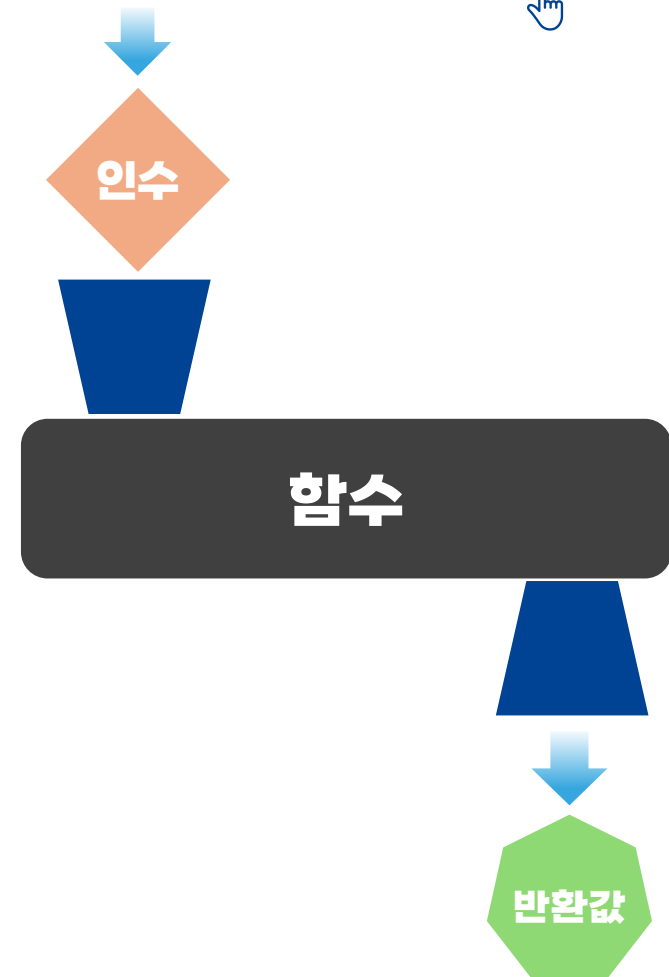
- 함수를 직접 정의를 해봤다
- 하지만 우리가 정의한 함수들은 인수도 받지 않고 반환값도 돌려주지 않았다





- 괄호 안에 인수의 값을 받을 변수 이름을 넣으면 된다
- 인수의 값을 받는 변수를 매개변수(parameter)라고 부른다
- 함수 안에 실행할 코드를 들여쓰기 해서 작성

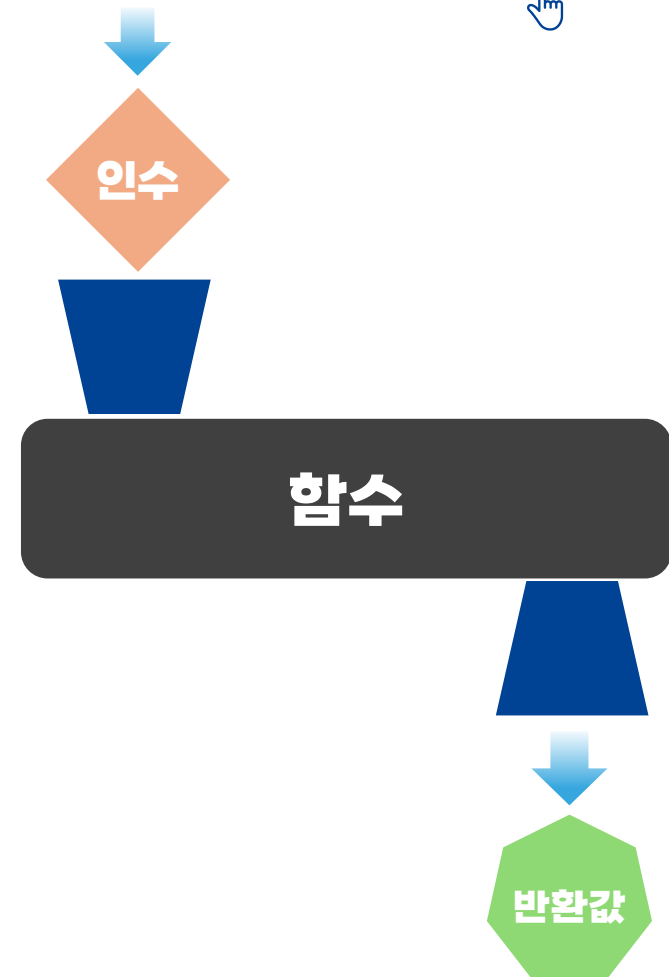
```
1  def function_name(parameter1):  
2      # first line of code  
3      # .  
4      # .  
5      # .
```





- 반환값은 return 키워드로 반환
- return이 실행되면 함수는 반환값을 돌려주고 바로 종료된다

```
1  def function_name(parameter1):  
2      # first line of code  
3      # .  
4      # .  
5      # .  
6      return some_result  
7  
8      #code that will never run
```





- 함수에게 인수를 줄 때 기본적으로 순서대로 **인수와 매개변수를 매칭**시킨다 (positional)
- 인수의 순서를 다르게 주고 싶으면 **매개변수의 이름과 = 기호**를 쓰면 된다 (keyword)

같은 코드



```
1  def function_name(parameter1, parameter2):  
2      # first line of code  
3      # ...  
4      return some_result  
5  
6  
7  
8  a = function_name(1, 2)  
9  a = function_name(parameter2 = 2, parameter1 = 1)
```





➤ 자동으로 매칭시키고 싶은 positional 인수는 무조건 keyword 인수들 전에 나와야 한다

```
1  def function_name(parameter1, parameter2):  
2      # first line of code  
3      # ...  
4      return some_result  
5  
6  
7  a = function_name(parameter2 = 2, 1) ❌  
8  
9  a = function_name(1, parameter2 = 2) ○
```





- 함수의 매개변수의 초기값도 정할 수 있다
- 정의할 때 괄호 안에 대입문처럼 (매개변수이름 = 값)
- 초기값이 있는 매개변수는 인수를 못 받으면 초기값을 대입시킨다
- 초기값이 없는 매개변수는 인수를 꼭 받아야 된다

```
1 def add(parameter1, parameter2 = 1):  
2     return parameter1 + parameter2  
  
>>> add(1, 2)  
3  
  
>>> add(0)  
1  
  
>>> add()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: add() missing 1 required positional  
argument: 'parameter1'
```

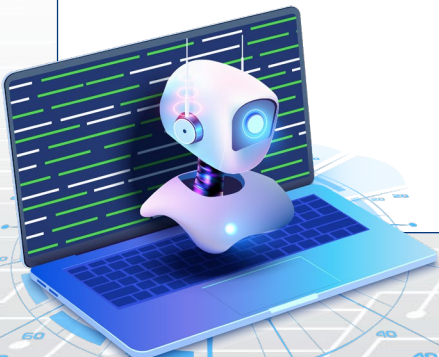


- Positional 및 keyword 인수와 비슷하게 함수 정의에 초기값 있는 매개변수들은 **무조건 초기값 없는 매개변수보다 나중에** 나와야 한다

```
def function_name(parameter1, parameter2 = 1, parameter3):
```



```
def function_name(parameter1, parameter2, parameter3 = 1):
```





➤ 인수를 받지 않는 함수도 구현 가능



```
1 def function1():  
2     # ...  
3     return some result
```

➤ return을 쓰지 않아 반환값을
돌려주지 않는 함수 구현 가능

- 파이썬이 알아서 None을 반환해준다



```
1 def function2(param):  
2     pass
```

➤ 여러 값을 반환하는 함수 구현 가능



```
1 def function3():  
2     return result1, result2
```



- VS Code에서 마우스를 함수 위에 놓으면 함수의 정의를 보여준다
- 매개변수, 매개변수의 타입, 초기값, 반환값의 타입을 보여준다

```
(function) def print(  
    *values: object,  
    sep: str | None = " ",  
    end: str | None = "\n",  
    file: SupportsWrite[str] | None = None,  
    flush: Literal[False] = False  
) -> None
```

Prints the values to a stream, or to sys.stdout by default.

sep

string inserted between values, default a space.

end

string appended after the last value, default a newline.

file

print(1)





- [Python Standard Library](#) – 파이썬이 **미리 구현해서 제공**한 코드 (자료형, 함수 등)
- **math** 라이브러리는 수학 함수, **random** 라이브러리는 의사 난수 생성을 위한 코드, 등등
- 라이브러리의 코드를 사용하고 싶다면 **import 예약어**를 쓰면 된다

```
1  import math
2
3  print(math.pi) # 3.1415
4  print(math.sin(0)) # 0
```





- import만 쓰면 라이브러리에 있는 **모든 코드**를 갖고 온다
- **특정 코드**만 사용하고 싶다면 **from**을 쓰면 된다
- from을 쓰면 빌려온 코드를 사용할 때 라이브러리 이름을 쓰면 안 된다

```
1  from math import pi
2
3  print(pi) # 3.1415
4  print(math.pi) # error
5  print(math.sin(0)) # error
```

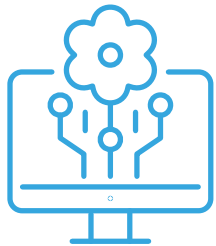




➤ import한 코드의 이름을 바꾸기 위한 예약어다

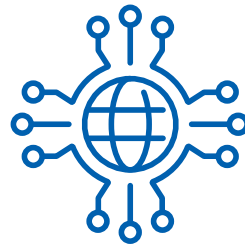
```
1  import math as m
2
3  print(m.pi) # 3.1415
4  print(math.pi) # error
```





math

<https://docs.python.org/ko/3/library/math.html>



random

<https://docs.python.org/ko/3/library/random.html>



time

<https://docs.python.org/ko/3/library/time.html>



- 함수 정의에 자신이 들어간 함수
- 자신을 호출하는 줄이 무조건 실행된다면 호출 스택에 무한적으로 추가된다

```
1  def recursive_function():  
2      if some_condition:  
3          return recursive_function()
```





➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

```
1  def print_nums(n):  
2      if n > 0:  
3          print_nums(n - 1)  
4          print(n - 1)  
5  
→ 6  print_nums(3)
```



print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

→

```
1 def print_nums(n):  
2     if n > 0: ○  
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```



print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

```
1 def print_nums(n):  
2     if n > 0:  
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```



print_nums(2), 3


print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

→

```
1 def print_nums(n):  
2     if n > 0:   
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```



print_nums(2), 3

print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

```
1 def print_nums(n):  
2     if n > 0:  
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```



print_nums(1), 3

print_nums(2), 3


print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

→

```
1 def print_nums(n):  
2     if n > 0:   
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```



print_nums(1), 3

print_nums(2), 3

print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

```
1 def print_nums(n):  
2     if n > 0:  
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```



print_nums(0), 3

print_nums(1), 3

print_nums(2), 3

print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

→

```
1 def print_nums(n):  
2     if n > 0: ✗  
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```



print_nums(0), 3

print_nums(1), 3

print_nums(2), 3

print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

```
1 def print_nums(n):  
2     if n > 0: ✗  
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```



print_nums(0), 3

print_nums(1), 3

print_nums(2), 3

print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

→

```
1 def print_nums(n):  
2     if n > 0: ✗  
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```



print_nums(1), 3

print_nums(2), 3

print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

```
1 def print_nums(n):  
2     if n > 0: ✗  
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```

0

print_nums(1), 3

print_nums(2), 3

print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

```
1 def print_nums(n):  
2     if n > 0: ✗  
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```

0

print_nums(1), 3

print_nums(2), 3

print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

→

```
1 def print_nums(n):  
2     if n > 0: ✗  
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```

0

print_nums(2), 3

print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

```
1 def print_nums(n):  
2     if n > 0: ✗  
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```

```
0  
1
```

print_nums(2), 3

print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

→

```
1 def print_nums(n):  
2     if n > 0: ✗  
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```

```
0  
1
```

print_nums(2), 3

print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

→

```
1 def print_nums(n):  
2     if n > 0: ✗  
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```

```
0  
1
```

print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

```
1 def print_nums(n):  
2     if n > 0: ✗  
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```

```
0  
1  
2
```

print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

```
1 def print_nums(n):  
2     if n > 0: ✗  
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```

```
0  
1  
2
```

print_nums(3), 6

호출 스택



➤ 반복적인 작업을 **반복문 대신 재귀함수**로 구현할 수 있다 (iteration vs. recursion)

```
1 def print_nums(n):  
2     if n > 0: ✗  
3         print_nums(n - 1)  
4         print(n - 1)  
5  
6 print_nums(3)
```

```
0  
1  
2
```

호출 스택



➤ 재귀함수를 구현할 때 함수가 이미 잘 구현이 되었다고 가정하고 정의하면 쉽다

```
1  def print_nums(n):  
2      # ...  
3      # ...  
4      # ...  
5  
6  print_nums(3)
```

**n보다 작은 인수를 받았을 때
print_nums가 진짜로 0부터
인수-1까지의 정수를 잘 출력한다면,
print_nums(n)을
어떻게 구현할 수 있을까?**



➤ 재귀함수를 구현할 때 함수가 이미 잘 구현이 되었다고 가정하고 정의하면 쉽다

```
1  def print_nums(n):  
2      # ...  
3      print_nums(n-1)  
4      print(n-1)  
5  
6  print_nums(3)
```

**print_nums(n-1)로
0부터 n-2를 출력한 후,
n-1을 직접 출력하면 되지 않을까?**



➤ 재귀함수를 구현할 때 함수가 이미 잘 구현이 되었다고 가정하고 정의하면 쉽다

```
1  def print_nums(n):  
2      # ...  
3      print_nums(n-1)  
4      print(n-1)  
5  
6  print_nums(3)
```

이제 무한으로 호출하는 것을
방지하기 위해 **base case**를 넣으면 된다



➤ 재귀함수를 구현할 때 함수가 이미 잘 구현이 되었다고 가정하고 정의하면 쉽다

```
1  def print_nums(n):  
2      if n > 0;  
3          print_nums(n-1)  
4          print(n-1)  
5  
6  print_nums(3)
```

**0부터 출력하니까 n이 1일 때
줄4가 마지막으로 호출되어야 한다**