

리스트

여러 개(가지)의 자료를 저장할 수 있는 자료구조

- 변수 묶음
- 칸 하나에 값 한개를 쓸 수 있는 목록(표)를 연상

L = [10, 20, 30, 40, 50]

L

10	20	30	40	50
[0]	[1]	[2]	[3]	[4]
[-5]	[-4]	[-3]	[-2]	[-1]

리스트(list)

- ▶ 여러 개(가지)의 자료를 저장할 수 있는 자료구조
- ▶ 자료들을 모아서 사용할 수 있게 해 줌
- ▶ 방법 : 대괄호([]) 내부에 자료들 넣어 선언

▶ `a = 5` # *a*는 변수이며 *1개*의 값만 저장

a

5

▶ `L = [10, 20, 30, 40, 50]` # *L*은 리스트이며 *현재 5개의 값(10개의 값 저장 가능)*을 저장

L

10	20	30	40	50
----	----	----	----	----

 [0] [1] [2] [3] [4]

 [-5] [-4] [-3] [-2] [-1]



리스트(list)의 값 조작하기

리스트 생성

```
L = [10, 20, 30, 40, 50]
```

L	10	20	30	40	50
	[0]	[1]	[2]	[3]	[4]
	[-5]	[-4]	[-3]	[-2]	[-1]

리스트 요소별 읽기 접근

```
print(L[0], L[4], L[-1])
```

```
print(L[1:4])
```

리스트 쓰기 접근

```
L[0] = 100
```

```
L[4] = L[2] + L[3]
```

```
L = [10, 20, 30, 40, 50]
```

```
print(L[0], L[4], L[-1])  
print(L[1:4])
```

```
L[0] = 100
```

```
L[4] = L[2] + L[3]
```

```
print(L)
```

100

70

```
10 50 50  
[20, 30, 40]  
[100, 20, 30, 40, 70]
```

리스트의 덧셈, 곱셈

▶ 문자의 덧셈과 곱셈

▶ $a = \text{"Hi "} + \text{"Jane"}$

$\# a \leftarrow \text{"Hi Jane"} \text{ (문자 + 문자 : 연결)}$

▶ $b = \text{"Hi "} * 3$

$\# b \leftarrow \text{"Hi Hi Hi"} \text{ (문자 * 정수 : 정수번 연결)}$

▶ 리스트의 덧셈과 곱셈

▶ $L1 = [10, 20, 30] + [40, 50]$

$\# L1 \leftarrow [10, 20, 30, 40, 50] \text{ (리스트 + 리스트 : 병합)}$

▶ $L2 = [10, 20, 30] * 2$

$\# L2 \leftarrow [10, 20, 30, 10, 20, 30] \text{ (리스트 * 정수 : 정수번 병합)}$

리스트(list) 생성방법 #1

빈 리스트 - 값으로 리스트 생성

L1 = [10, 20, 30, 40, 50]

빈 리스트

L2 = []

L3 = list()

0인 요소를 10개 갖는 리스트

L4 = [0] * 10 *# [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]*

range()함수를 활용해 a ~ b-1 범위의 정수 리스트

L5 = list(range(5, 11)) *# [5, 6, 7, 8, 9, 10]*

리스트(list) 생성방법 #2

입력값으로 리스트 만들기 - 문자형

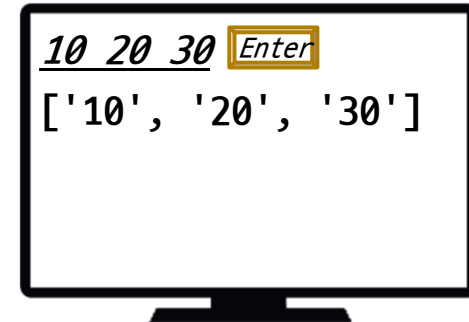
```
L6 = list( input().split() )
```

"10 20 30"

"10", "20", "30"으로 분리

["10", "20", "30"] 리스트로 변환

```
print(L6)
```



입력값으로 리스트 만들기 - 정수형

```
L7 = list( map(int, input().split()) )
```

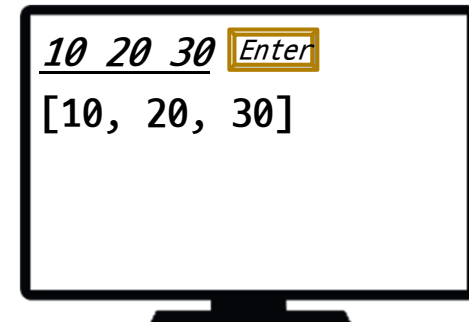
"10 20 30"

"10", "20", "30"으로 분리

10, 20, 30으로 형변환

[10, 20, 30] 리스트로 변환

```
print(L7)
```



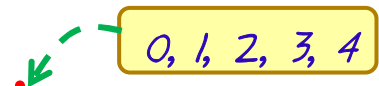
반복문과 리스트

```
L = [2, 8, 7, 6, 5]
```

```
print(L[0])  
print(L[1])  
print(L[2])  
print(L[3])  
print(L[4])
```

```
L = [2, 8, 7, 6, 5]
```

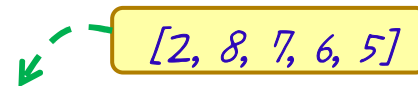
```
for i in range(5) :  
    print(L[i])
```



```
2  
8  
7  
6  
5
```

```
L = [2, 8, 7, 6, 5]
```

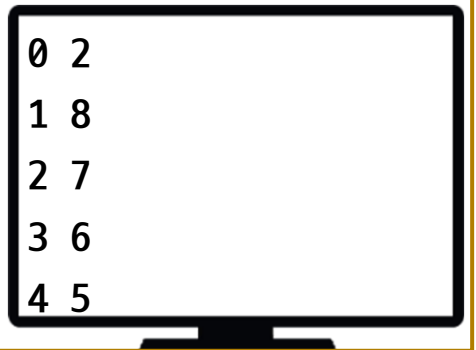
```
for a in L :  
    print(a)
```



참고

```
L = [2, 8, 7, 6, 5]
```

```
for a, b in enumerate(L) :  
    print(a, b)
```



```
0 2  
1 8  
2 7  
3 6  
4 5
```

리스트의 메소드 함수

형식 : 리스트객체.메소드함수()

리스트에 추가

L.append() 리스트에 끝에 값 추가

L.insert() index위치에 값 추가

리스트에서 삭제

L.remove() 리스트에서 원하는 값 삭제

L.pop() 리스트에서 마지막 값 (리턴) 후 삭제

L.clear() 리스트 비우기

탐색

L.index() 리스트에서 찾는 값의 인덱스 위치 리턴

L.count() 찾는 값이 몇 개인지 리턴

정렬

L.sort() 리스트를 정렬

L.reverse() 리스트의 원소 순서 뒤집기

리스트에 값 추가 및 삭제

L = [10, 20, 30]

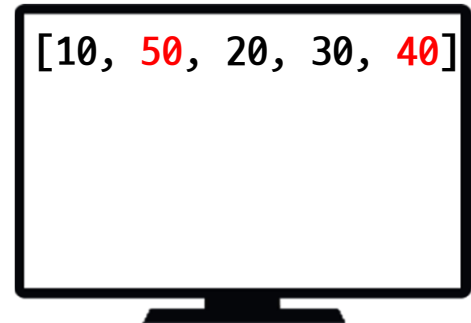
L.append(40)

L.insert(1, 50)

print(L)

10	20	30	40
[0]	[1]	[2]	[3]

10	50	20	30	40
[0]	[1]	[2]	[3]	[4]



[10, 50, 20, 30, 40]

L = [10, 20, 30]

L.remove(20)

L.pop()

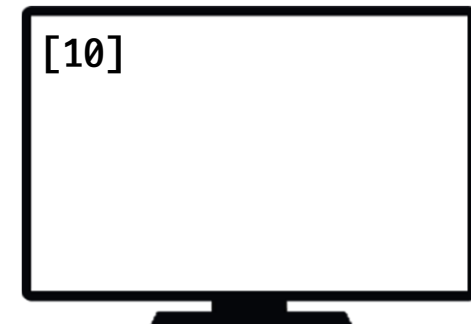
print(L)

10	30
[0]	[1]

20

10
[0]

30



[10]

리스트에서 값 탐색 및 정렬

```
L = [10, 30, 20, 30, 30]
```

```
print(L.index(20))
```

10	30	20	30	30
[0]	[1]	[2]	[3]	[4]

```
print(L.count(30))
```



```
L = [10, 30, 20, 30, 30]
```

```
L.sort()
```

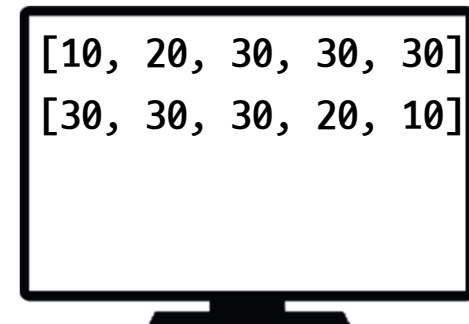
```
print(L)
```

10	20	30	30	30
[0]	[1]	[2]	[3]	[4]

```
L.reverse()
```

```
print(L)
```

30	30	30	20	10
[0]	[1]	[2]	[3]	[4]



리스트에서 값 탐색 및 정렬

```
L = [10, 30, 20, 30, 30]
```

```
print(L.index(20))
```

10	30	20	30	30
[0]	[1]	[2]	[3]	[4]

```
print(L.count(30))
```



```
L = [10, 30, 20, 30, 30]
```

```
L.sort()
```

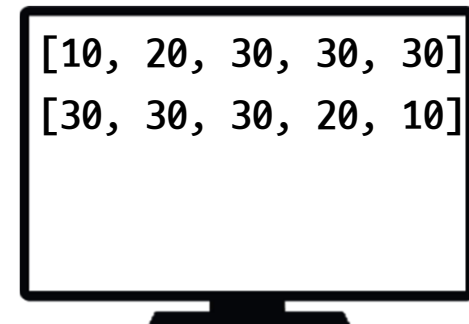
```
print(L)
```

10	20	30	30	30
[0]	[1]	[2]	[3]	[4]

```
L.reverse()
```

```
print(L)
```

30	30	30	20	10
[0]	[1]	[2]	[3]	[4]



리스트에서의 **in** 연산 – True or False

L = [1, 2, 3]

print(3 **in** [1, 2, 3] L) # True : L에 3이 존재함

print(3 **not in** [1, 2, 3] L) # False : L에 3이 존재함

print(30 **in** [1, 2, 3] L) # False : L에 30이 존재하지 않음

print(30 **not in** [1, 2, 3] L) # True : L에 30이 존재하지 않음

“이상한 출석부르기 0” - in, not in 연산자 활용

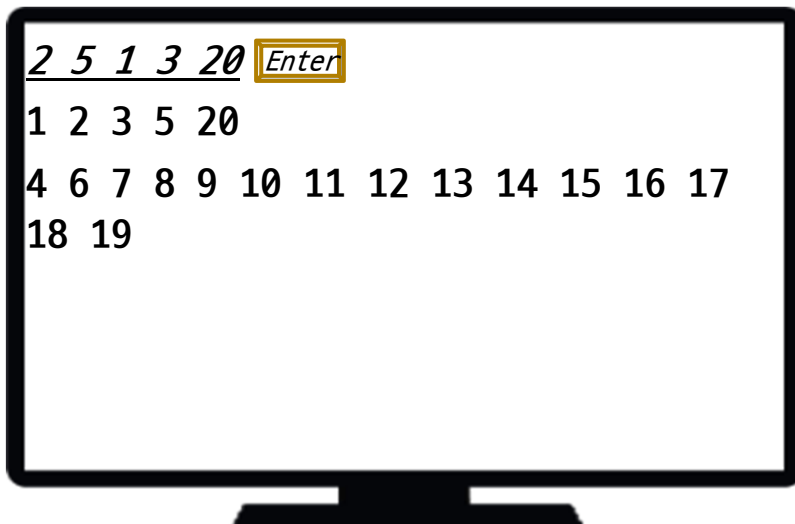
▶ 문제 요지

- ▶ 출석을 부른 학생의 번호가 입력
- ▶ 출석을 부른 학생과 안부른 학생을 분류하여 출력

[입력예시]

2 5 1 3 20 (출석부른 번호)

2	5	1	3	20
[0]	[1]	[2]	[3]	[4]



```
L = list(map(int, input().split()))
```

```
    1 2 3 4 5 ... 20
for i in range(1, 21) :
    2, 5, 1, 3, 20
    if i in L :
        print(i, end=' ')

print()

for i in range(1, 21) :
    if i not in L :
        print(i, end=' ')
```

[합계, 최대값, 최소값 구하기 문제]



```
5
11
2
8
30
9
60 30 2
```

```
n = int(input())

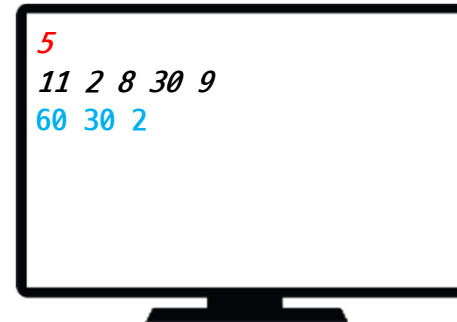
sum_v = 0
max_v = -1000
min_v = 1000
for i in range(n) :
    a = int(input())

    sum_v = sum_v + a

    if max_v < a :
        max_v = a

    if min_v > a :
        min_v = a

print(sum_v, max_v, min_v)
```



```
5
11 2 8 30 9
60 30 2
```

```
n = int(input())

L = list(map(int, input().split()))



|    |   |   |    |   |
|----|---|---|----|---|
| 11 | 2 | 8 | 30 | 9 |
|----|---|---|----|---|



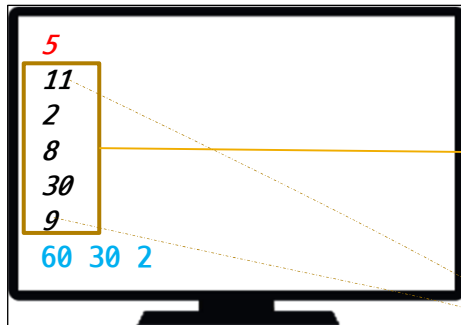
[0][1][2][3][4]



print(sum(L), max(L), min(L))
```

**값들이
리스트에 존재할 때,
편리한 점이 많음.**

[합계, 최대값, 최소값 구하기 문제]



왼쪽처럼 한줄에 한개씩 데이터가 입력되었을 때,
어떻게 리스트로 만들 수 있을까???

11	2	8	30	9
[0]	[1]	[2]	[3]	[4]

```
n = int(input())
```

```
L = [0] * n
```

0	0	0	0	0
[0]	[1]	[2]	[3]	[4]

0, 1, 2, 3, 4

```
for i in range(n):
```

```
    L[i] = int(input())
```

11	2	8	30	9
[0]	[1]	[2]	[3]	[4]

```
print(sum(L), max(L), min(L))
```

List의 append() 메소드 이용

```
n = int(input())
```

```
L = [] # 빈 리스트
```

```
for i in range(n):
```

```
    a = int(input())  
    L.append(a)
```

L.append(int(input()))

```
print(sum(L), max(L), min(L))
```

파이썬의 리스트는 여러 자료형을 담을 수 있다.

```
L = [1, 3.14, [2, 3], "Hello"]  
      [0]   [1]   [2]   [3]
```

```
print( L, type(L) )
```

```
print( L[0], type(L[0]) )
```

```
print( L[1], type(L[1]) )
```

```
print( L[2], type(L[2]) )
```

```
print( L[2][0], type(L[2][0]) )
```

```
print( L[3], type(L[3]) )
```

```
print( L[3][0], type(L[3][0]) )
```

1	3.14	2	3	'H'	'e'	'l'	'l'	'o'
[0]	[1]	[0]	[1]	[0]	[1]	[2]	[3]	[4]
[0]	[1]	[2]	[3]					

```
[1, 3.14, [2, 3], 'Hello'] <class 'list'>
```

```
1 <class 'int'>
```

```
3.14 <class 'float'>
```

```
[2, 3] <class 'list'>
```

```
2 <class 'int'>
```

```
Hello <class 'str'>
```

```
H <class 'str'>
```


2차원 리스트

1차원 리스트

```
L1 = [1, 2, 3]
```

1	2	3
[0]	[1]	[2]

2차원 리스트 – 리스트의 리스트

```
L2 = [[1, 2, 3], [4, 5, 6]]
```

	0열	1열	2열
0행	1	2	3
1행	4	5	6

1차원 리스트 vs 2차원 리스트

▶ 1차원 리스트

- ▶ L1 = [1, 2, 3, 4, 5, 6]
- ▶ print(L1[3]) # 4 출력
- ▶ L1[?] = 20

1	2	3	4	5	6
[0]	[1]	[2]	[3]	[4]	[5]

20

▶ 2차원 리스트 - 리스트의 리스트

- ▶ L2 = [[1, 2, 3], [4, 5, 6]]
- ▶ print(L2[1][2]) # 6 출력
- ▶ print(L2[1]) # [4, 5, 6] 출력
- ▶ print(L2[1]) # [1, 2, 3, 4, 5, 6] 출력
- ▶ L2[?][?] = 50

	0열	1열	2열
0행	1	2	3
1행	4	5	6

50

행 열

1차원 리스트 vs 2차원 리스트

L1 = [1, 2, 3, 4, 5]

L2 = [[1], [2, 3], [4, 5, 6]]

print(len(L1)) # 5

print(len(L2)) # 3

print(len(L2[0])) # 1

print(len(L2[1])) # 2

print(len(L2[2])) # 3

1	2	3	4	5
[0]	[1]	[2]	[3]	[4]

	0열	1열	2열
0행	1		
1행	2	3	
2행	4	5	6

```
L = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

[0]

[1]

[2]

	[0]	[1]	[2]	[3]	[4]	
[0]	1	2	3	4	5	L[0]
[1]	6	7	8	9	10	
[2]	11	12	13	14	15	L[2]

```
L = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

```
print(L)
```

```
print(L[0])
```

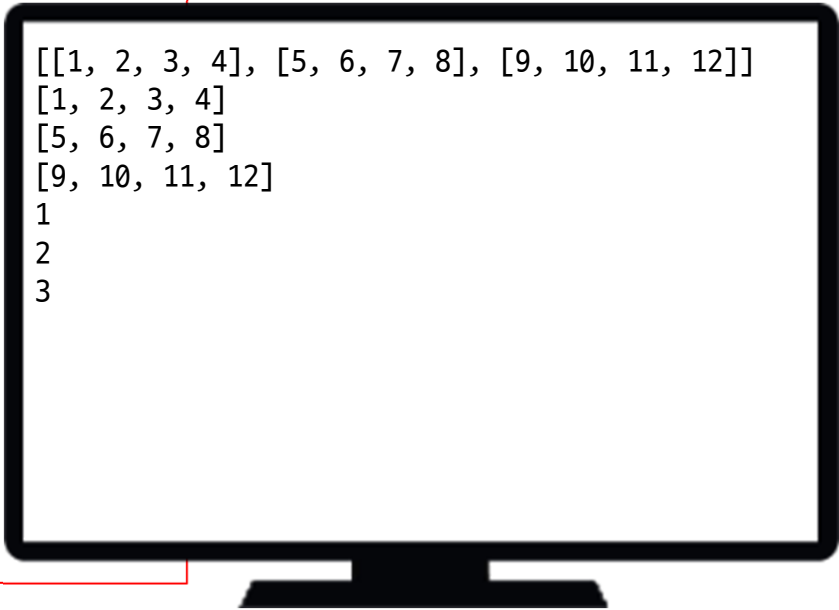
```
print(L[1])
```

```
print(L[2])
```

```
print(L[0][0])
```

```
print(L[0][1])
```

```
print(L[0][2])
```



```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]  
[1, 2, 3, 4]  
[5, 6, 7, 8]  
[9, 10, 11, 12]  
1  
2  
3
```

`L = [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15], [16, 17, 18, 19, 20]]`

← 열 →

	[0]	[1]	[2]	[3]	[4]
↑ 행	[0] 1	2	3	4	5
[1] 6	7	8	9	10	
[2] 11	12	13	14	15	
[3] 16	17	19	19	20	

L[2][4]
리스트 L의
2행 4열

4행 5열 리스트

```
L = [[1, 2, 3, 4, 5],  
      [6, 7, 8, 9, 10],  
      [11, 12, 13, 14, 15],  
      [16, 17, 18, 19, 20]]
```

```
print(L)
```

```
print(L[0])
```

```
print(L[1])
```

```
print(L[2])
```

```
print(L[3])
```

```
print(L[2][4])
```

1차원 리스트 순차적 접근

```
L = [1, 2, 3, 4, 5, 6]
```

0, 1, 2, 3, 4, 5

```
for i in range(6) :  
    print(L[i])
```

1	2	3	4	5	6
[0]	[1]	[2]	[3]	[4]	[5]

```
L = [1, 2, 3, 4, 5, 6]
```

0, 1, 2, 3, 4, 5

```
for i in range(len(L)) :  
    print(L[i])
```



```
L = [1, 2, 3, 4, 5, 6]
```

[1, 2, 3, 4, 5, 6]

```
for i in L:  
    print(i)
```

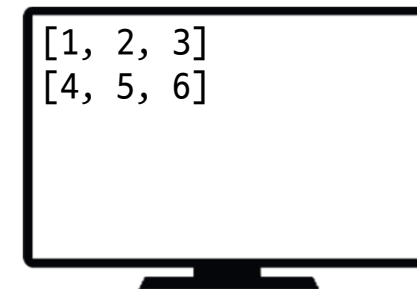
2차원 리스트 순차적 접근

```
L = [[1, 2, 3], [4, 5, 6]]  
for i in range(2):  
    print(L[i])
```

	0열	1열	2열
0행	1	2	3
1행	4	5	6

```
L = [[1, 2, 3], [4, 5, 6]]  
for i in range(len(L)):  
    print(L[i])
```

```
L = [[1, 2, 3], [4, 5, 6]]  
for i in L:  
    print(i)
```

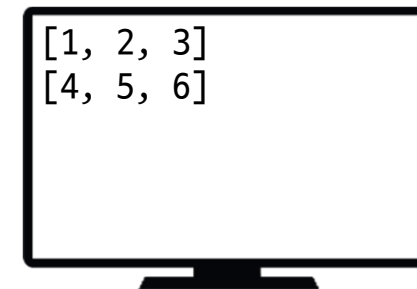


2차원 리스트 순차적 접근

```
L2 = [[1, 2, 3], [4, 5, 6]]  
for i in range(2) :  
    for j in range(3) :  
        print(L2[i][j], end=' ')  
  
    print()
```

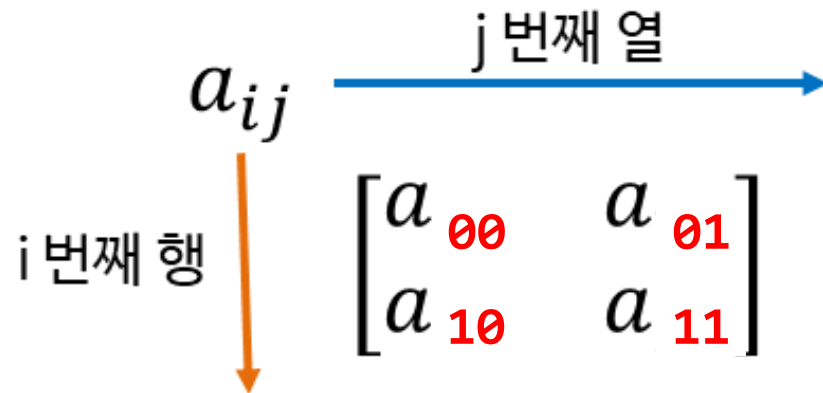
	0열	1열	2열
0행	1	2	3
1행	4	5	6

```
L = [[1, 2, 3], [4, 5, 6]]  
for i in range(len(L)) :  
    for j in range(len(L[i])) :  
        print(L[i][j], end=' ')  
  
    print()
```

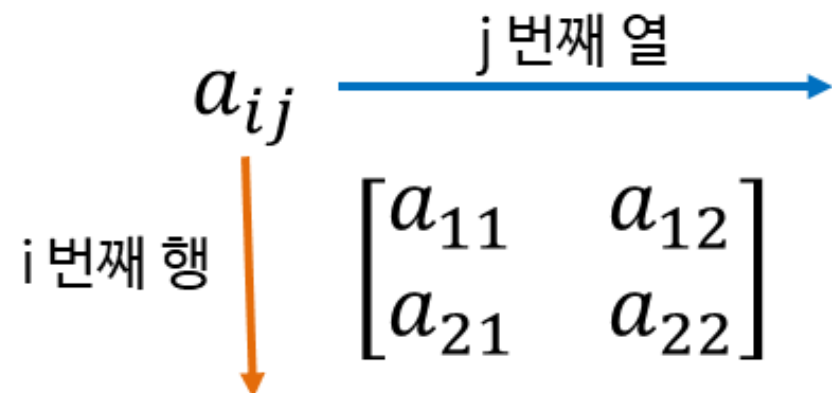


리스트와 수학의 행렬

파이썬 리스트의 행과 열번호



수학 행렬에서 행과 열번호



리스트 컴프리헨션으로 1차원 리스트 만들기

```
L = []
```

```
for i in range(5):  
    L.append(i)
```

```
print(L)
```

0	1	2	3	4
[0]	[1]	[2]	[3]	[4]

```
L = [ i for i in range(5) ]
```

```
print(L)
```

```
L = list(range(5))
```

```
print(L)
```

```
L = []
```

```
for i in range(5):  
    L.append(0)
```

```
print(L)
```

0	0	0	0	0
[0]	[1]	[2]	[3]	[4]

```
L = [ 0 for i in range(5) ]
```

```
print(L)
```

```
L = [0] * 5
```

```
print(L)
```

리스트 컴프리헨션 → 2차원 리스트 만들기

```
L = []
```

```
for i in range(2) :  
    L.append([])  
    for j in range(5) :
```

```
        L[i].append(j)
```

```
print(L)
```

[0]

[1]

0	1	2	3	4
0	1	2	3	4

[0]

[1]

[2]

[3]

[4]

```
L = [ [ j for j in range(5) ] for i in range(2) ]
```

```
print(L)
```

```
L = []
```

```
for i in range(2) :  
    L.append([])  
    for j in range(5) :
```

```
        L[i].append(0)
```

```
print(L)
```

[0]

[1]

0	0	0	0	0
0	0	0	0	0

[0]

[1]

[2]

[3]

[4]

```
L = [ [ 0 for j in range(5) ] for i in range(2) ]
```

```
print(L)
```

컴프리헨션 (comprehension)으로 2차원 리스트 생성

	[0]	[1]	[2]	[3]
[0]	0	0	0	0
[1]	0	0	0	0
[2]	0	0	0	0

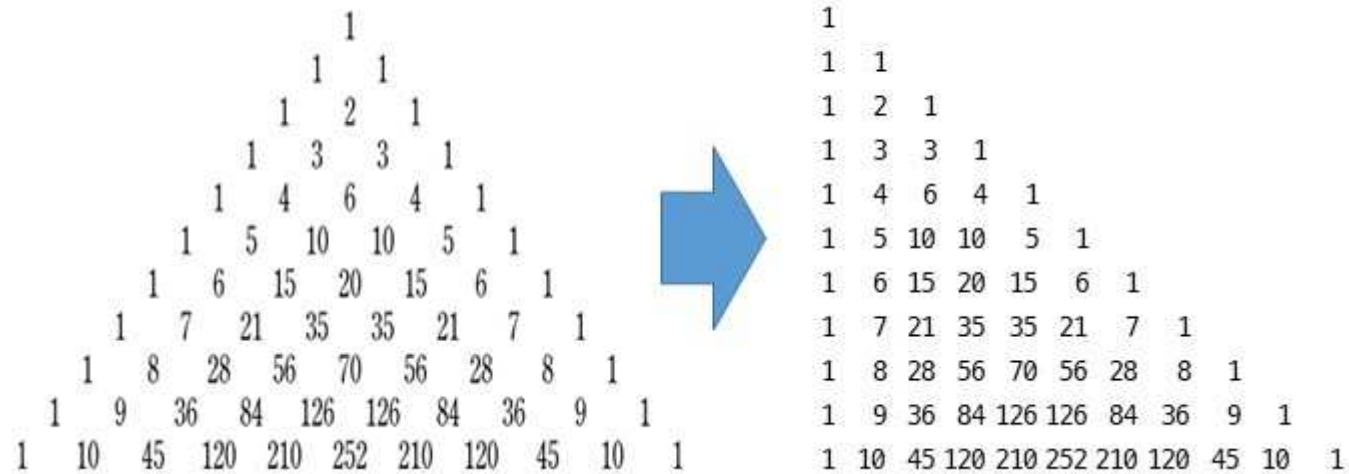
```
L = [ [ 0 for j in range(4) ] for i in range(3) ]
```

```
L = []  
  
for i in range(3) :  
    L.append([])  
    for j in range(4) :  
        L[i].append(0)
```

	[0]	[1]	[2]	[3]	...	[17]	[18]	[19]
[0]	0	0	0	0	0	0	0	0
[1]	0	0	0	0	0	0	0	0
[2]	0	0	0	0	0	0	0	0
[3]	0	0	0	0	0	0	0	0
...	0	0	0	0	0	0	0	0
[17]	0	0	0	0	0	0	0	0
[18]	0	0	0	0	0	0	0	0
[19]	0	0	0	0	0	0	0	0

```
L = [ [ 0 for j in range(19) ] for i in range(19) ]
```

```
L = []  
  
for i in range(19) :  
    L.append([])  
    for j in range(19) :  
        L[i].append(0)
```



만약 n 이 5라면...

	[0]	[1]	[2]	[3]	[4]
[0]	1	0	0	0	0
[1]	1	1	0	0	0
[2]	1	2	1	0	0
[3]	1	3	3	1	0
[4]	1	4	6	4	1

```

if j==0 :
    L[i][j] = 1
else :
    L[i][j] = L[i-1][j-1] + L[i-1][j]

```

변수와 값(객체)의 메모리 할당

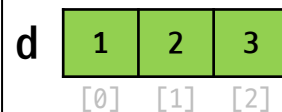
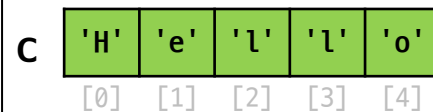
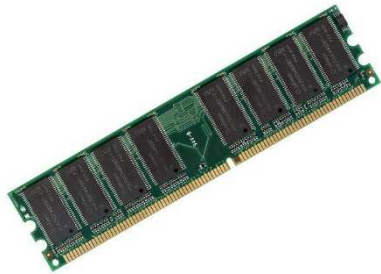
선생님의 거짓말!!!

a = 1

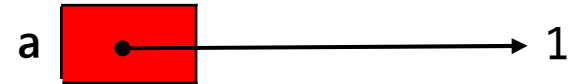
b = 3.14

c = "Hello"

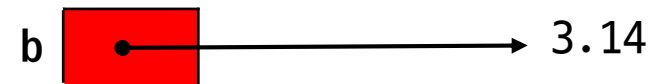
d = [10, 20, 30]



1의 id를 저장



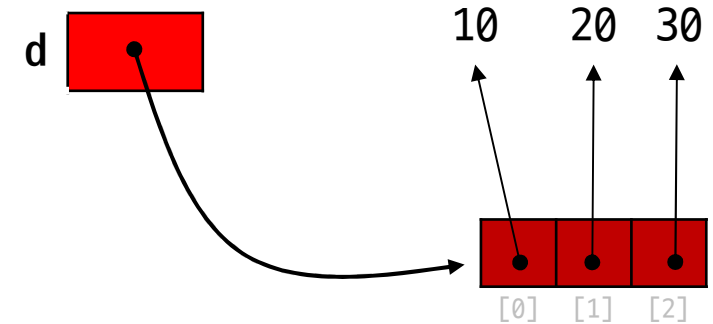
3.14의 id를 저장



"Hello"의 id를 저장



[10, 20, 30]의 id를 저장



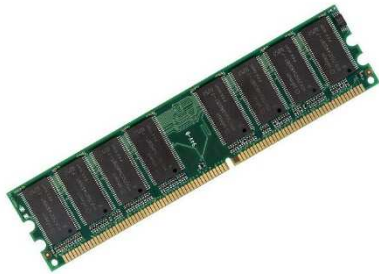
변수는 stack에 실제 객체는 heap에 저장

a = 1

b = 3.14

c = "Hello"

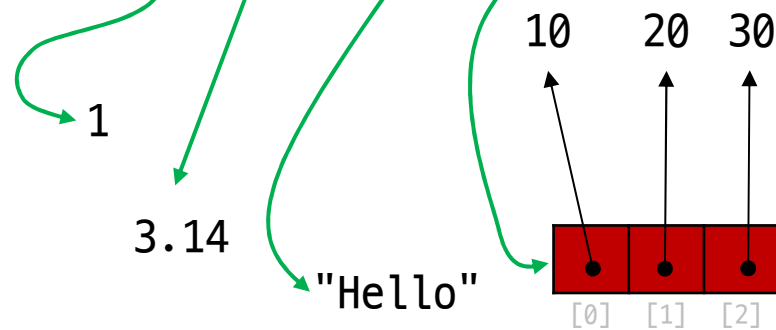
d = [10, 20, 30]



stack



heap



data

code(or text)

Type	Immutable?
int	Yes
float	Yes
bool	Yes
complex	Yes
tuple	Yes
frozenset	Yes
str	Yes
list	No
set	No
dict	No

immutable한 int, float 등

```
1 a = 10  
2 b = 10  
3 b = 20
```

1번 라인 코드에 의해

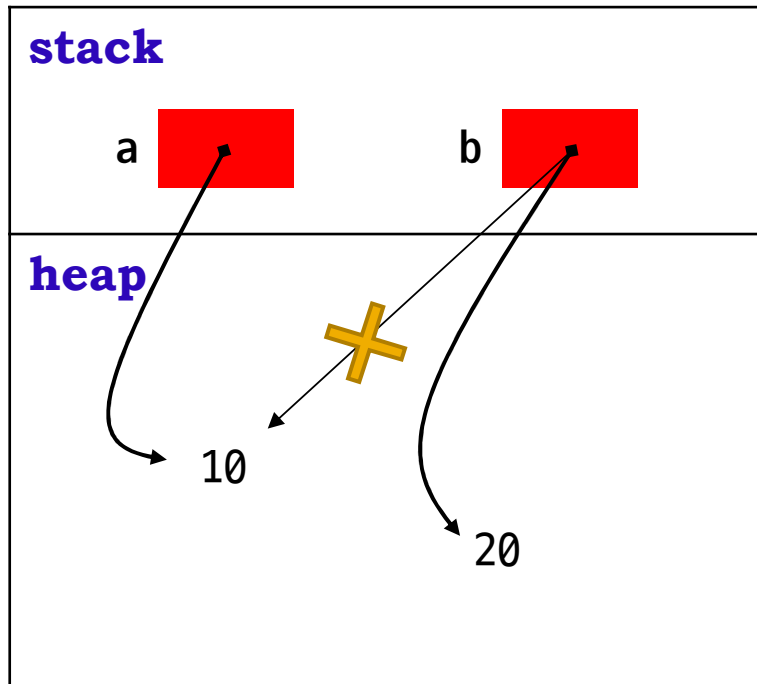
변수 a는 heap의 10 객체를 가리킴

2번 라인 코드에 의해

변수 b도 heap의 10 객체를 가리킴

3번 라인 코드에 의해

변수 b가 heap의 10 객체를 가리키지 않고,
이제 heap의 20 객체를 가리킴

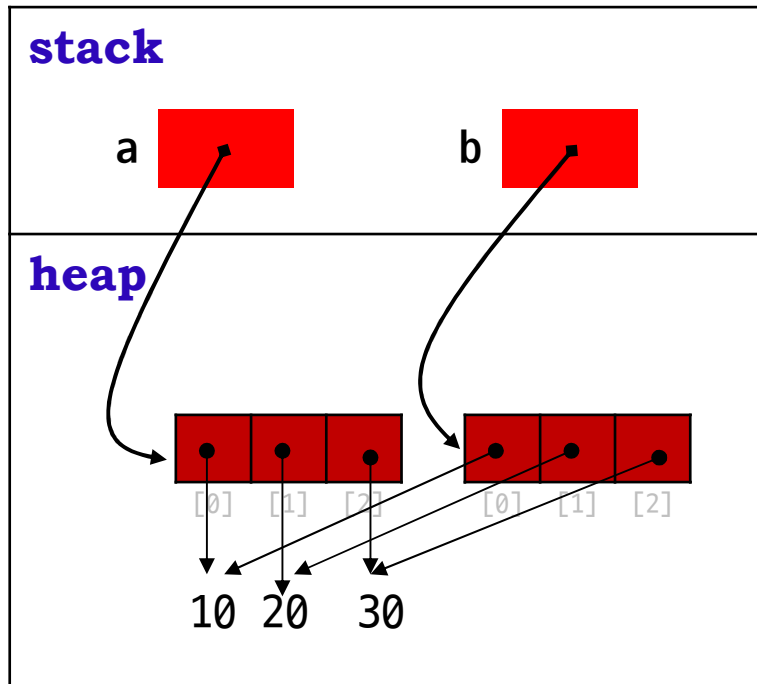


이때, int형은 immutable하기 때문에,
10을 20으로 바꿀 수 없다.

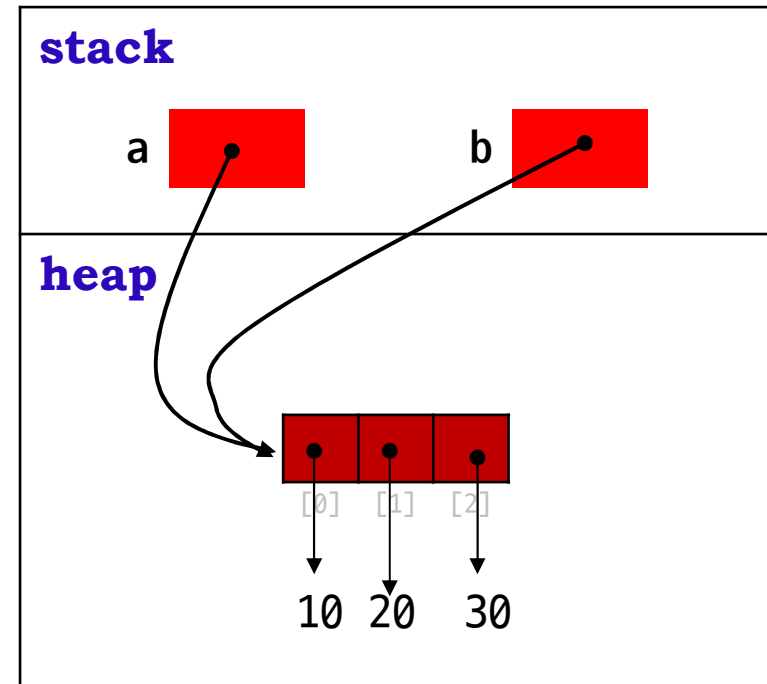
따라서 새로운 int 객체 20을 생성하여
변수 b가 그것을 가리키게 한다.

mutable한 리스트

```
a = [10, 20, 30]  
b = [10, 20, 30]
```

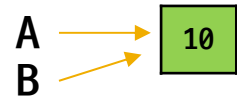


```
a = [10, 20, 30]  
b = a
```

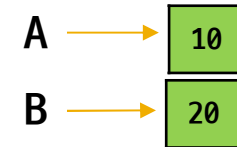


Immutable 객체 정리

A = 10
B = 10

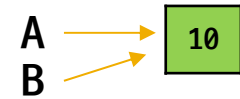


B = 20

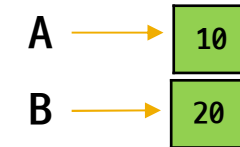


```
print(A)    # 10  
print(B)    # 20
```

A = 10
B = A



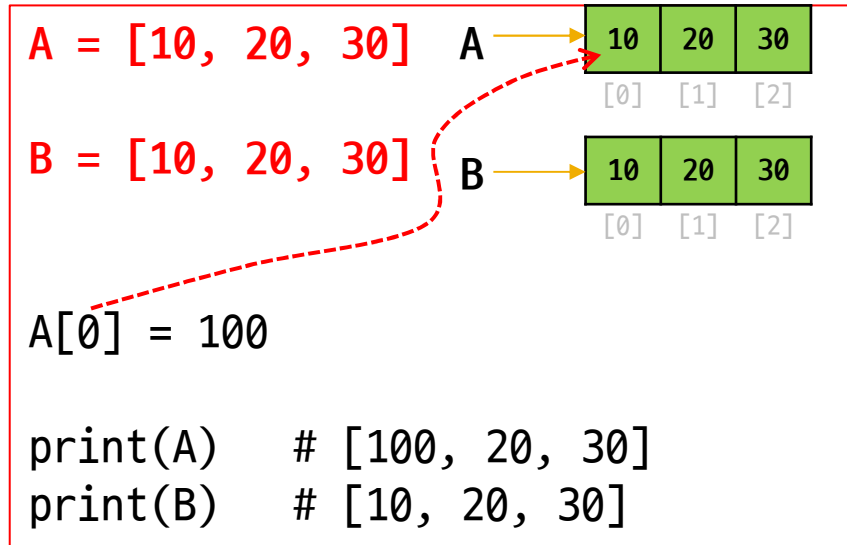
B = 20



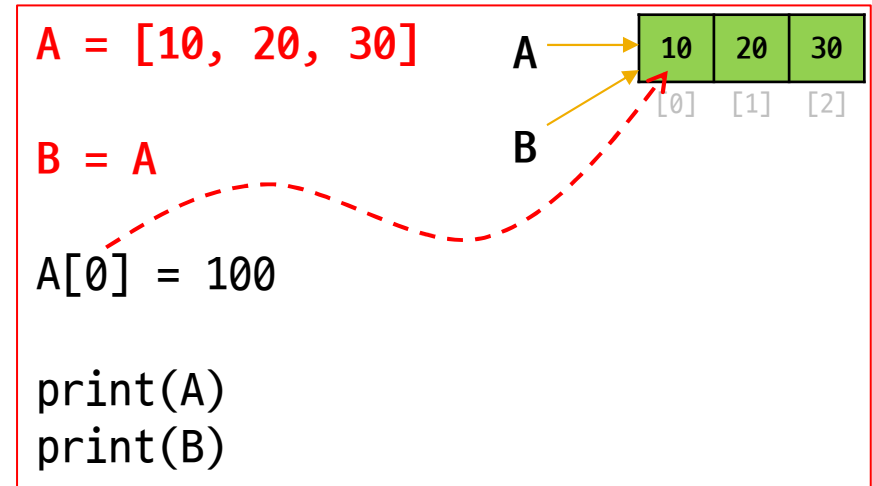
```
print(A)    # 10  
print(B)    # 20
```



Mutable 객체 정리



```
[100, 20, 30]  
[10, 20, 30]
```



```
[100, 20, 30]  
[100, 20, 30]
```

군집자료형

- 수치 자료형
 - int : 1, 0, -10
 - float : 3.14
 - complex : 2+3j

- bool 자료형
 - True False

- 군집 자료형
 - str : "Hello"
 - list : [1, 2, 3]
 - tuple : (1, 2, 3)
 - dictionary : {'H':1.01, 'He' : 4.00 }
 - set : {1, 2, 3}

- * immutable(변경불가)
int, float, complex, bool, str, tuple
- * mutable(변경가능)
list, dictionary, set

군집자료형 - str

- ▶ str 객체는 immutable(수정불가)한 객체
- ▶ str 객체는 큰따옴표(" ")나 작은따옴표(' ')
- ▶ 순서(인덱스 접근)가 있고, 중복이 가능

```
s = "Hello World"
```

'H'	'e'	'l'	'l'	'o'	' '	'W'	'o'	'r'	'l'	'd'
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

```
print(len(s)) # 11(s의 길이)
```

```
print(s[2]) # l(s의 2번 인덱스 값)
```

```
print(s[2:5]) # llo(s의 2이상 5미만 인덱스 값)
```

```
s[2] = 'a' # 에러 발생!, 값 수정 불가
```

```
print("CN" + "SH") # CNSH
```

```
print("CNSH" * 3) # CNSHCNSHCNSH
```

```
# str의 메소드 함수
```

```
print(s.upper())
```

```
# 문자열 객체 s의 모든 문자를 대문자로
```

```
print(s.count('l'))
```

```
# 문자열 객체 s에서 l의 개수를 리턴
```

```
print(s.index('r'))
```

```
# 문자열 객체 s에서 r의 index값을 리턴
```

```
print(s.replace('l', 'L'))
```

```
# 문자열 객체 s에서 모든 l을 L로 치환
```

```
a, b = s.split()
```

```
# 문자열 객체 s를 공백을 기준으로 분리
```


군집자료형 – list

- ▶ list 객체는 mutable(수정가능)한 객체
- ▶ list 객체는 []로 표현
- ▶ 순서(인덱스 접근)가 있고, 중복이 가능
- ▶ 여러 자료형을 한 리스트에 포함 가능(L=[1, 3.5, "sun"])

```
L = [1, 2, 3]
```

1	2	3
[0]	[1]	[2]

```
print(len(L)) # 3
```

```
print(L[1]) # 2
```

```
print(L[1:3]) # [2, 3]
```

```
L[1] = 20 # L은 [1, 20, 3]이 됨
```

리스트에 적용가능한 유용한 내장함수
sum(L), min(L), max(L)

```
L2 = [1, 2] + [3, 4] # [1, 2, 3, 4]
```

```
L3 = [1, 2] * 2 # [1, 2, 1, 2]
```

```
L.append(10) # 맨 뒤에 10을 추가 – 리턴값 없음  
L.insert(1, 10) # 인덱스 1위치에 10 추가 – 리턴값 없음  
L.remove(10) # 10을 제거 – 리턴값 없음  
L.pop() # 맨 뒤에 값을 제거하고 그 값을 리턴  
L.count(10) # 10의 개수를 리턴  
L.index(10) # 10이 있는 index 위치를 리턴  
L.sort() # 정렬(기본값 : 오름차순) – 리턴 값 없음  
L.reverse() # 순서 뒤집기 – 리턴값 없음
```

메소드의 리턴(반환값)의 유무에 따라

```
L = [1, 2, 3]  
print(L.append(3))  
print(L.count(3))
```

None
2

군집자료형 – tuple 수정 불가능 리스트임

- ▶ tuple 객체는 immutable(수정불가)한 객체
- ▶ tuple 객체는 ()로 표현
- ▶ 순서(인덱스 접근)가 있고, 중복이 가능
- ▶ 여러 자료형을 한 리스트에 포함 가능(T=(1, 3.5, "sun"))

```
T = (1, 2, 3, 3)
```

1	2	3	3
[0]	[1]	[2]	[3]

```
print(T) # (1, 2, 3, 3)
```

```
print(len(T)) # 4
```

```
print(T[1]) # 2
```

```
print(T[1:3]) # (2, 3)
```

```
T[1] = 20 # 불가
```

```
print(sum(T), max(T), min(T)) # 9, 3, 1
```

```
print(T.count(3)) # 2
```

```
print(T.index(2)) # 1
```

```
T = (1, 2) + (3, 4) # (1, 2, 3, 4)
```

```
T = (1, 2) * 2 # (1, 2, 1, 2)
```

- 튜플은 거의 리스트와 유사 (단, 값 변경이 불가)
- 프로그래밍시 값이 변경되면 안되는 값들은 튜플형으로...
- 접근속도면에서 리스트보다 빠름
- 튜플을 수정/변경하는 `append()`, `remove()`, `sort()` 등의 메소드가 없다.

군집자료형 – set 수학의 집합과 유사

- ▶ set 객체는 mutable(수정가능)한 객체
- ▶ set 객체는 {값, 값, ...}로 표현
- ▶ 순서가 없고(인덱스 접근 불가), 중복 불가

```
S1 = {1, 2, 3, 4, 5, 5, 4} # 중복된 값이 있으면 자동 제거  
S2 = {4, 5, 6, 7, 8}
```

```
print(S1)      # {1, 2, 3, 4, 5} 중복된 값은 제거됨.
```

```
print(len(S1)) # 5
```

```
print(S1[1]) # 순서가 없으므로 index로 접근 불가
```

```
print(S1 & S2)  # 교집합 {4, 5}                                s1.intersection(s2)
```

```
print(S1 | S2)  # 합집합 {1, 2, 3, 4, 5, 6, 7, 8}             s1.union(s2)
```

```
print(S1 - S2)  # 차집합 {1, 2, 3}                             s1.difference(s2)
```

```
S1.add(100)     # 값 1개 추가 {1, 2, 3, 4, 5, 100}
```

```
S1.update({200, 300}) # 값 2개이상 추가 {1, 2, 3, 4, 5, 100, 200, 300}
```

```
S1.remove(1)    # 원소 중 1 제거 {2, 3, 4, 5, 100, 200, 300}
```

```
print(S1)       # {2, 3, 4, 5, 100, 200, 300}
```

군집자료형 – dict key:value 쌍들의 묶음

- ▶ dict 객체는 mutable(수정가능)한 객체
- ▶ dict 객체는 {키:값, 키:값, ...}로 표현 - 키와 값이 대응관계
- ▶ 순서가 없고(인덱스 접근 불가), 중복 불가

```
D = {"H":1.01, "He":4.00, "Li":6.94}
```

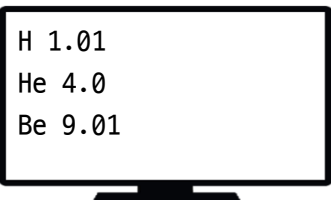
```
print(D) # {'H': 1.01, 'He': 4.0, 'Li': 6.94}
print(len(D)) # 3
```

```
print(D[1]) # 순서가 없으므로 index로 접근 불가
print(D["H"]) # 1.01 혹은 get()메소드를 사용해 print(D.get("H"))
```

```
D["Be"] = 9.01 # "Be":9.01 쌍을 추가
del D["Li"] # "Li":6.94 쌍을 삭제
```

```
print(D.items()) # dict_items([('H', 1.01), ('He', 4.0), ('Be', 9.01)])
print(D.keys()) # dict_keys(['H', 'He', 'Be'])
print(D.values()) # dict_values([1.01, 4.0, 9.01])
```

```
for k, v in D.items():
    print(k, v)
```



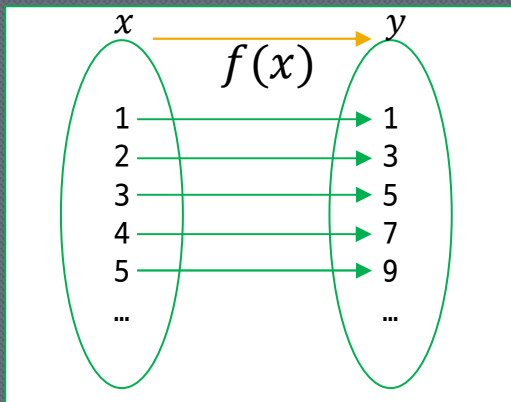
```
H 1.01
He 4.0
Be 9.01
```

사용자 정의 함수 설계

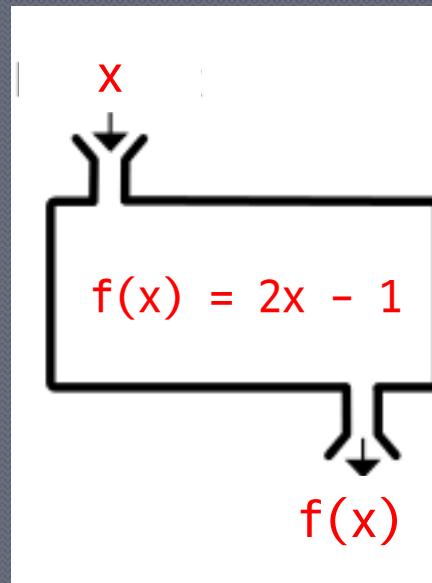
함수의 정의 : 특정 기능을 수행하도록 명령어를 묶음

함수의 호출 : 정의된 함수가 실행되도록 함

x 는 자연수



$$f(x) = 2x - 1$$



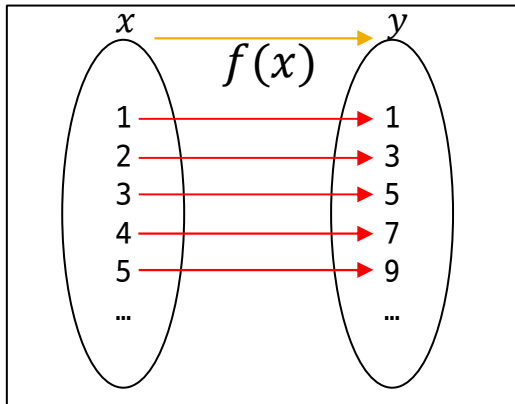
```
def f(x) :  
    y = 2 * x - 1  
    return y
```

x 번째 홀수를 구하는 함수

수학에서
함수의 정의

$$f(x) = 2x - 1$$

x 는 자연수



파이썬에서
함수의 정의

```
def f(x) :  
    y = 2 * x - 1  
    return y
```

```
print( f(1) )  
print( f(2) )  
print( f(3) )  
print( f(4) )  
print( f(5) )  
  
a = f(99)  
print( a )
```

```
main.py  
1 def f(x) :  
2     y = 2 * x - 1  
3     return y  
4  
5 print( f(1) )  
6 print( f(2) )  
7 print( f(3) )  
8 print( f(4) )  
9 print( f(5) )  
10  
11 a = f(99)  
12 print( a )  
13
```

input

1
3
5
7
9
197



x번째 홀수를 구하는 함수의 이름은???

무의미한 이름은
피할 것

```
def f(x) :  
    y = 2 * x - 1  
    return y
```

```
print( f(1) )  
print( f(2) )  
print( f(3) )  
print( f(4) )  
print( f(5) )
```

```
a = f(99)  
print( a )
```

함수 이름은 *f*보다
함수가 가진 기능을
떠올릴 수 있는 단어가
좋겠군...



기능과 관련된
이름으로

```
def odd(x) :  
    y = 2 * x - 1  
    return y
```

```
print( odd(1) )  
print( odd(2) )  
print( odd(3) )  
print( odd(4) )  
print( odd(5) )
```

```
a = odd(99)  
print( a )
```

식별자(identifier)의 이름규칙

* 식별자 : 변수명, 함수명 등

- 알파벳, 숫자, _의 조합
- 숫자로 시작할 수 없음
- 예약어 사용 금지

함수 이름을
holsu
라고 해도
될까???



파이썬 함수의 구조

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

result = add(3, 5) # 전달인자(3, 5)
```

함수정의 키워드 함수명 매개변수(들)

a=3, b=5

1. 함수명 : 변수의 명명규칙과 같음

2. 매개변수 개수 = 전달인자 개수
가. 디폴트 파라미터 혹은
나. 가변인자 사용 함수 등은 예외



3. return의 의미

- 가. **return res**
- 함수 종료
 - 호출한 곳에 **res값을 가지고 복귀**
- 나. **return**
- 함수 종료
 - 호출한 곳에 **복귀**

```
def func(a, b) : # a, b를 parameter(매개변수) - 매개변수는 변수(variable)의 의미
    print(a, b)
```

```
func(1, 2) # 1, 2를 argument((전달)인자 or 전달인수) - 전달인수는 값(value)의 의미
```

```
func(1) # 전달인자와 매개변수 개수가 일치하지 않아 오류
```

```
func(1, 2, 3) # 전달인자와 매개변수 개수가 일치하지 않아 오류
```

return : 함수의 강제 종료

return 값이 **있는** 경우

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

```
res = add(3, 5)  
print( res )
```



return 값이 **없는** 경우

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

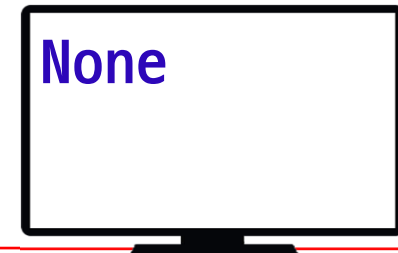
```
res = add(3, 5)  
print( res )
```



return 뒤에 있는 코드는
불필요한 코드

```
def add(a, b) :  
    r = a + b  
    return  
    print("ππ")
```

```
res = add(3, 5)  
print( res )
```



return 값의 개수

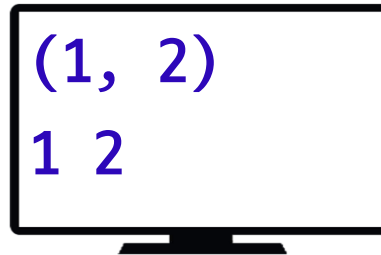
return값이 여러개 일 경우 기본은 **튜플로 리턴**

```
def func1() :  
    return 1  
  
res = func1()  
print(res)
```



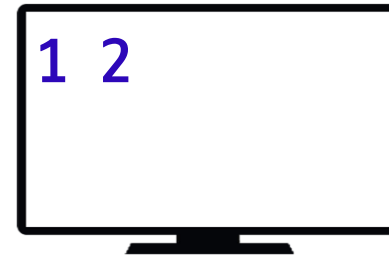
1

```
def func2() :  
    return 1, 2  
  
res = func2()  
print(res)  
print(res[0], res[1])
```



(1, 2)
1 2

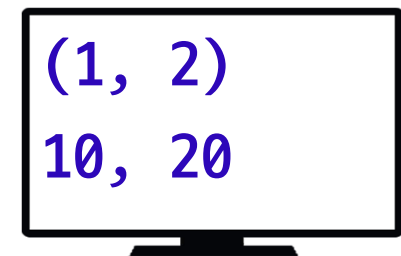
```
def func3() :  
    return 1, 2  
  
res1, res2 = func3()  
print(r1, r2)
```



1 2

배경지식

```
a = 1, 2  
x, y = 10, 20  
print(a)  
print(x, y)
```



(1, 2)
10, 20

매개변수와 전달인자

매개변수 개수 = 전달인자 개수

```
def func1(name, age) :  
    print(a, type(a))  
    print(b, type(b))
```

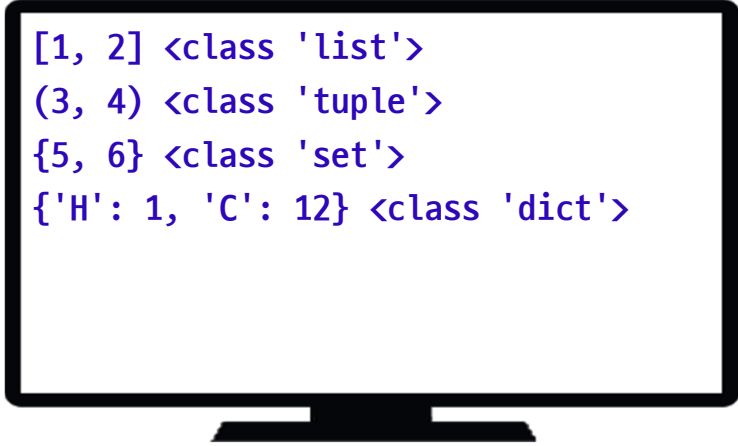
```
func1("홍길동", 17)
```



8

```
def func2(L, T, S, D) :  
    print(L, type(L)) # 리스트  
    print(T, type(T)) # 튜플  
    print(S, type(S)) # 셋(집합)  
    print(D, type(D)) # 딕셔너리
```

```
func2([1, 2], (3, 4), {5, 6}, {"H":1, "C":12})
```



```
[1, 2] <class 'list'>  
(3, 4) <class 'tuple'>  
{5, 6} <class 'set'>  
{'H': 1, 'C': 12} <class 'dict'>
```

디폴트 매개변수 - 매개변수의 기본값 지정

일반적인 매개변수

```
def func1(name, age, school) :  
    print(name, age, school)
```

```
func1("홍길동", 17, "충남과학고")
```

```
func1("홍길동", 17) # 오류
```

디폴트 매개변수 사용

```
def func2(name, age, school="충남과학고") :  
    print(name, age, school)
```

```
func2("홍길동", 17, "충남과학고")
```

```
func2("홍길동", 17) # 오류 아님
```

```
def func(param1, param2=value2, param3=value3) : ( O ) - 디폴트 파라미터는 뒤에서부터 쓸 것...
```

```
def func(param1=value1, param2) : ( X ) - 앞에서 쓰면 에러...
```

내장함수 print()는 아래와 같지 않을까???

```
def print(*values, sep=' ', end='\n')  
    .....
```

```
print("apple", "grape")
```

```
print("apple", "grape", sep=', ')
```

```
print("Hello")
```

```
print("World")
```

```
print("Hello", end=' ')
```

```
print("World")
```



```
apple grape  
apple,grape  
Hello  
World  
Hello World
```

디폴트 매개변수(default parameter)

```
def introduce(name, age, school="충남과학고") :  
    print("이름 :", name, "나이 :", age, "학교 :", school)
```

```
introduce("홍길동", 18, "울도고") # 이름 : 홍길동 나이 : 18 학교 : 울도고
```

```
introduce("김도균", 17) # 이름 : 김도균 나이 : 17 학교 : 충남과학고
```

```
introduce("김도균") # 오류(age 값 없음)
```

`TypeError: introduce() missing 1 required positional argument: 'age'`

```
introduce(17, "김도균") # 이름 : 17 나이 : 김도균 학교 : 충남과학고
```

```
introduce(age=17, name="김도균") # 이름 : 김도균 나이 : 17 학교 : 충남과학고
```



```
이름 : 홍길동 나이 : 18 학교 : 울도고  
이름 : 김도균 나이 : 17 학교 : 충남과학고  
이름 : 17 나이 : 김도균 학교 : 충남과학고  
이름 : 김도균 나이 : 17 학교 : 충남과학고
```

가변인자를 받는 매개변수

```
def func1(a, b) :  
    print(a, b)
```

```
func1(1, 2)
```

1, 2

```
def func2(a, b) :  
    print(a, b)
```

```
func2(1, 2, 3)
```

TypeError: func2() takes 2 positional arguments but 3 were given

```
def func3(a, *b) :  
    print(a, b)
```

```
func3(1, 2, 3)
```

1, (2, 3)

```
def func4(*a) :  
    print(a)
```

```
func4(1, 2, 3)
```

(1, 2, 3)

`def func(param1, *param2) :` (O) - 가변인자를 받는 변수는 뒤에...

`def func(*param1, param2) :` (X) - 앞에서 쓰면 에러...

함수의 호출(call) 복귀(return) 과정

함수의 호출(call)과 복귀(return)

```
def a( ) :  
    ...  
    res = b( c )  
    ...  
    return
```

```
def b(x) :  
    ...  
    res = c( )  
    return r
```

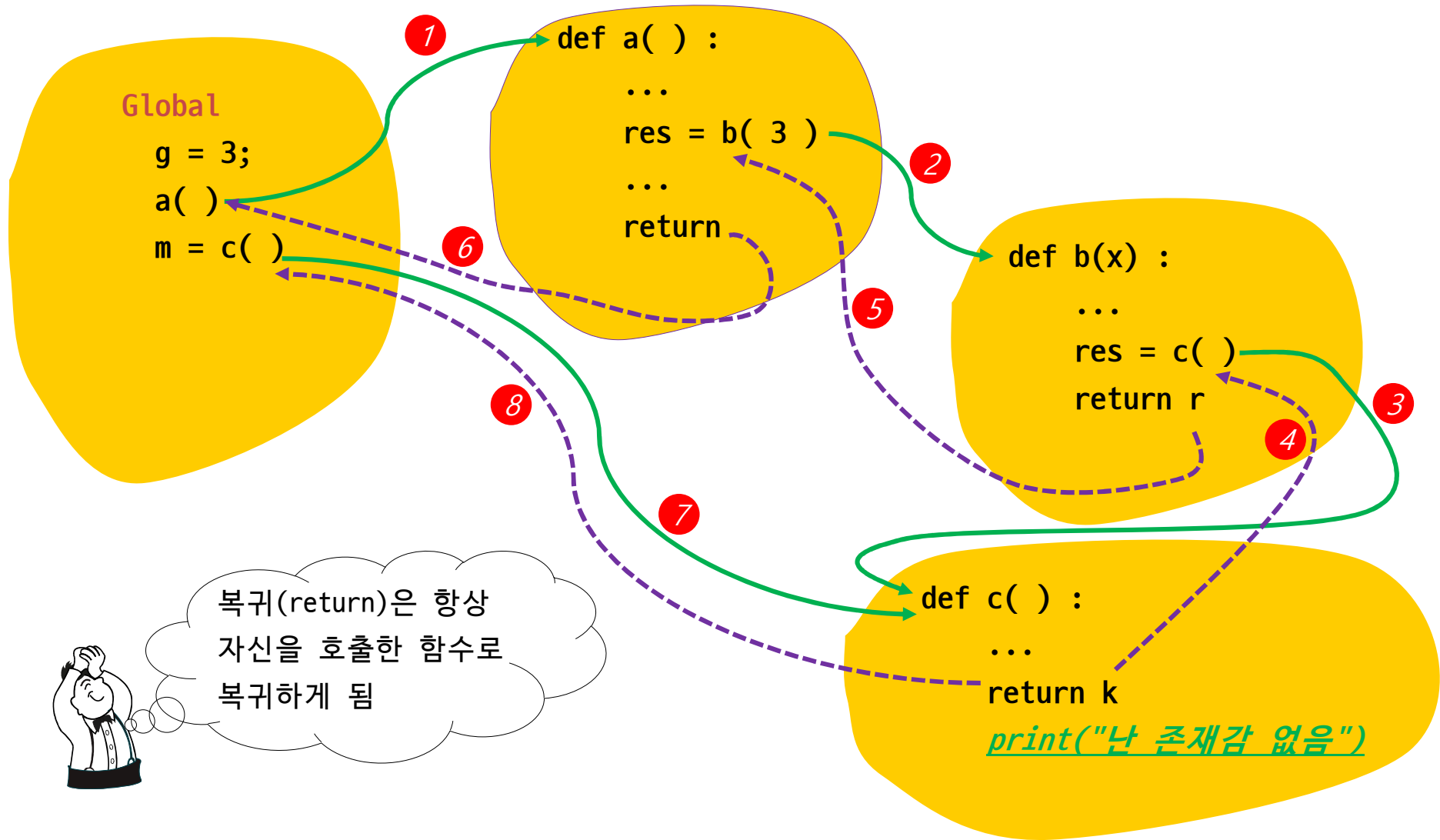
```
def c(n) :  
    ...  
    return k  
    print("난 존재감 없음")
```

```
g = 3;  
a( );  
m = c( )
```

함수가 아무리 많아도
global 영역에서 시작



함수의 호출(call)과 복귀(return)



생략 가능한 **return** 문은???

```
def func1(a, b) :  
    if a > b :  
        return ①  
    print(n)  
    return ②
```

```
def func3(a, b) :  
    print(a)  
    print(b)  
    print(a+b)  
    return ③
```

```
def func2(a, b) :  
    print(a)  
    print(b)  
    return a+b ④
```

함수내의 코드가 실행 완료 후

return 문이 없어도

함수는 종료되고 호출한 실행의
흐름이 되돌아 간다.



지역변수와 전역변수

Namespace(이름공간) – 지역변수 vs 전역변수

네임스페이스 ?

변수 이름과 함수 이름을 정하는 것이 중요한데 이들 모두를 겹치지 않게 정하는 것은 사실상 불가능

특정한 하나의 이름이 통용될 수 있는 범위를 제한. 즉, 소속된 네임스페이스가 다르다면 같은 이름이 다른 개체를 가리키도록 하는 것이 가능

Local Namespace :

함수 내의 지역 변수들의 이름들이 소속

Global Namespace :

모듈 전체에서 통용될 수 있는 이름들이 소속

f local
Name
space

```
def f(x) : # x, y는 f의 지역변수  
    y = 20  
    print(x, y) # 10, 20
```

g local
Name
space

```
def g() : # x, y는 g의 지역변수  
    x = 1  
    y = 2  
    print(x, y) # 1, 2
```

Global
Namespace

```
x = 100 # x, y는 전역변수  
y = 200  
f(10)  
g()  
print(x, y) # 100, 200
```

전역변수 x, y와 지역변수 a, b

	지역변수	전역변수
생명주기	함수 실행 ~ 함수 종료	프로그램 실행 ~ 프로그램 종료
참조범위	같은 지역내 (같은 함수내)	어디서나 참조 변경은 불가 (global 을 통해 변경도 가능)

```
def func() :  
    a = 10  
    b = 20  
  
    print(a, b) # 10 20 (지역변수 a, b 참조)  
    print(x, y) # 1 2 (전역변수 a, b 참조)
```

```
x = 1  
y = 2  
func()  
print(a, b) # 오류(참조불가):지역변수는 해당 지역에서만...  
print(x, y) # 1, 2
```

<func frame>

a 10

b 20

지역
(local)

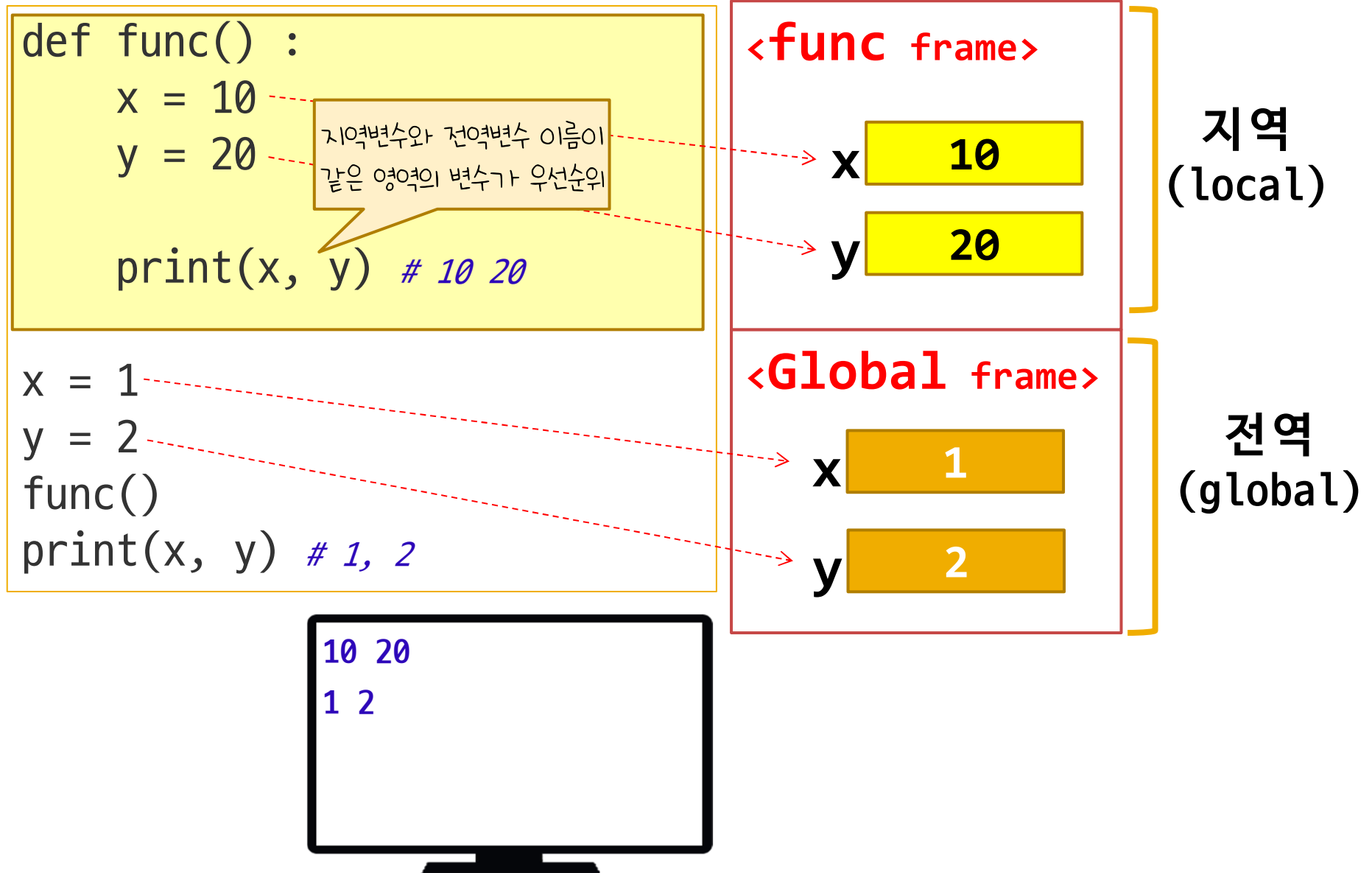
<Global frame>

x 1

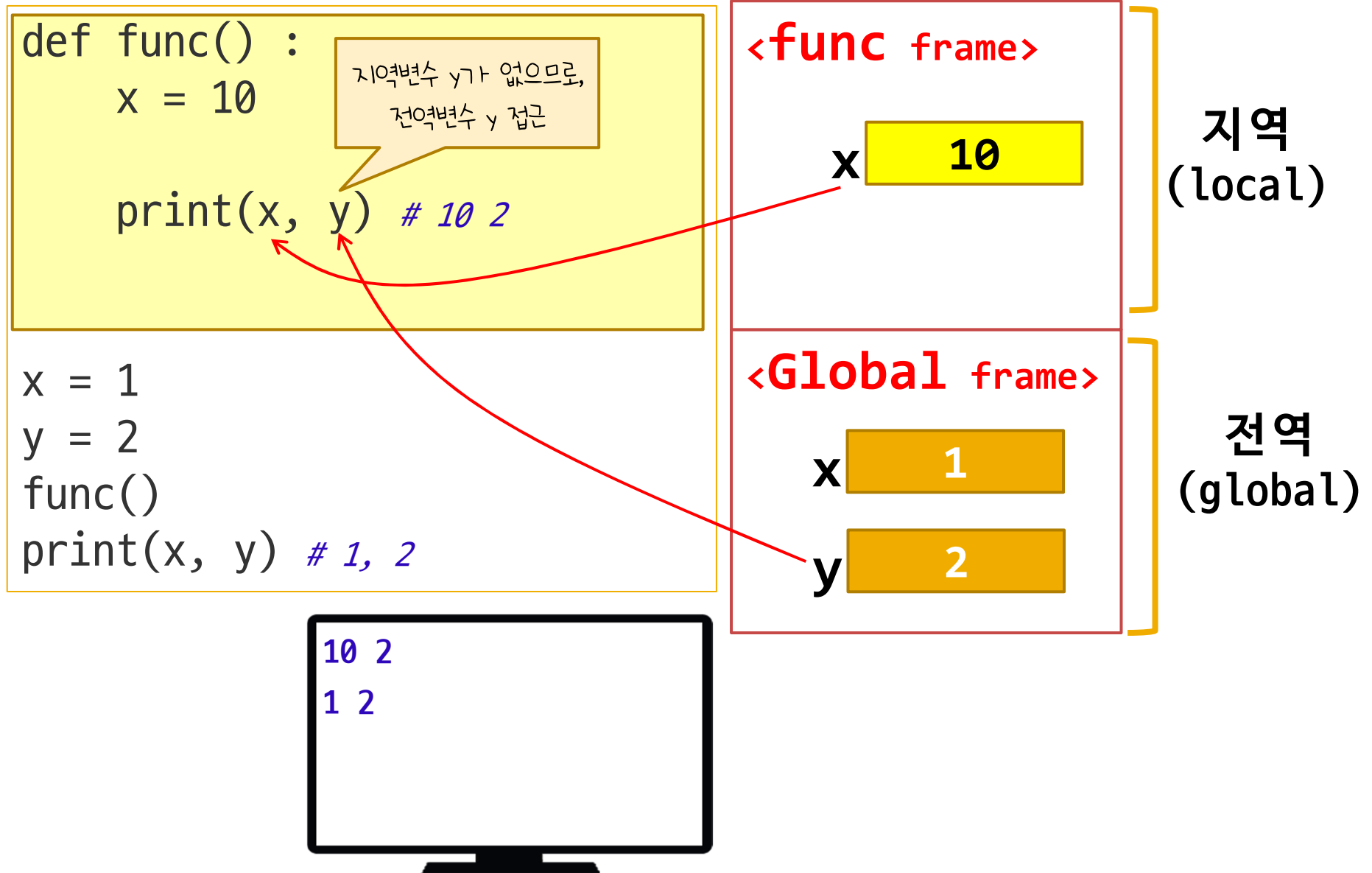
y 2

전역
(global)

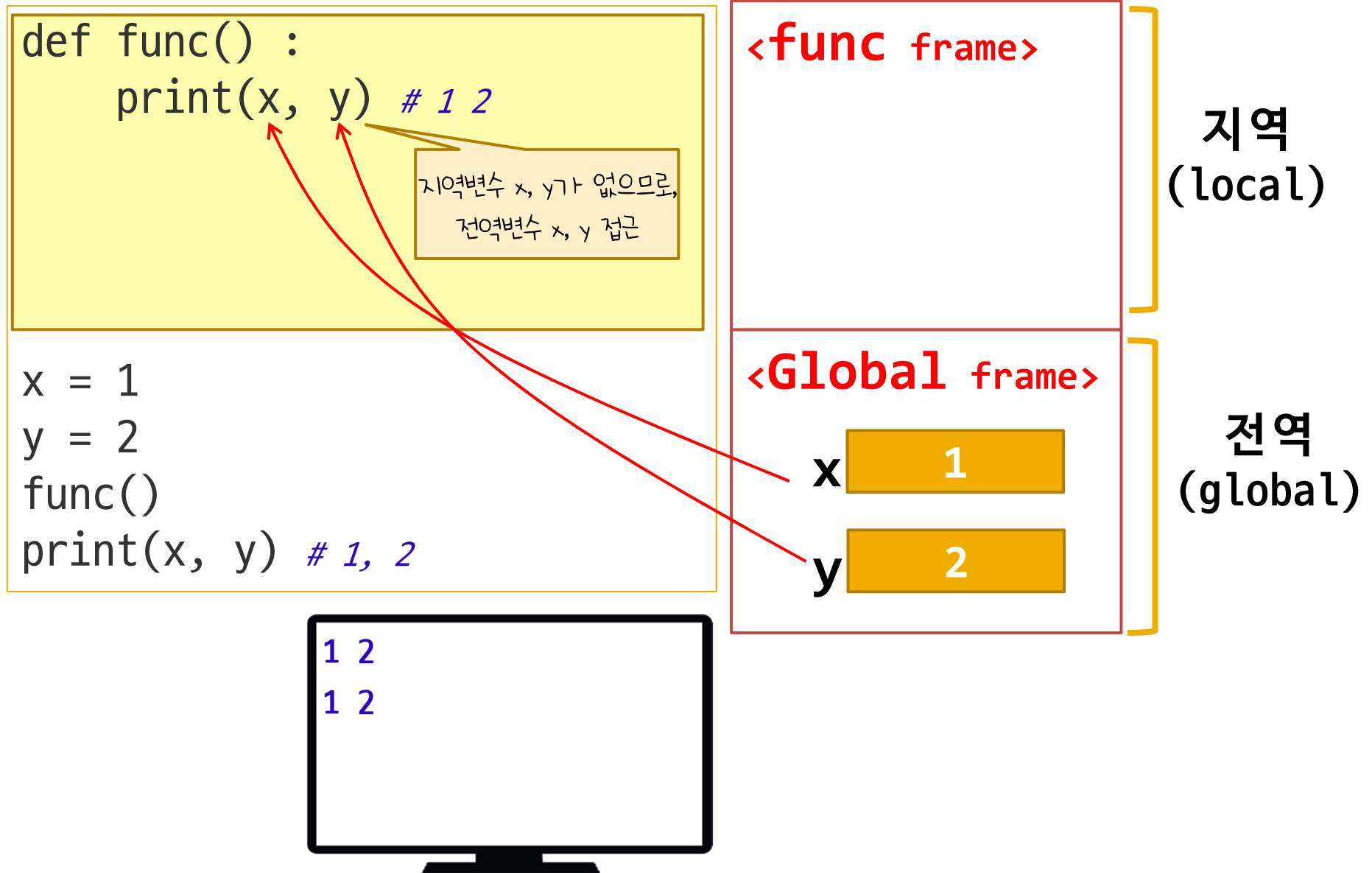
전역변수 x, y와 지역변수 x, y



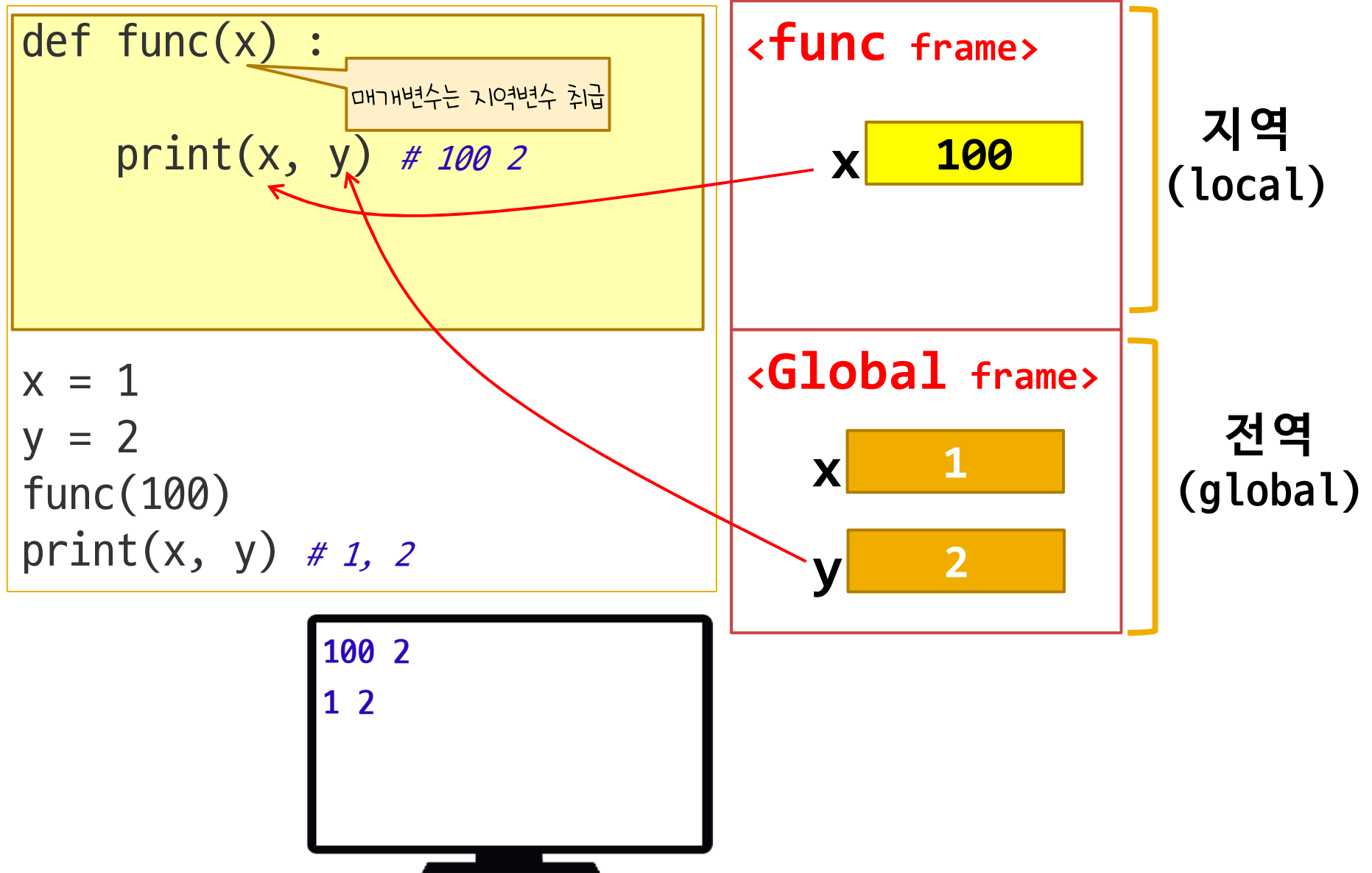
전역변수와 지역변수 - 전역변수 값 참조



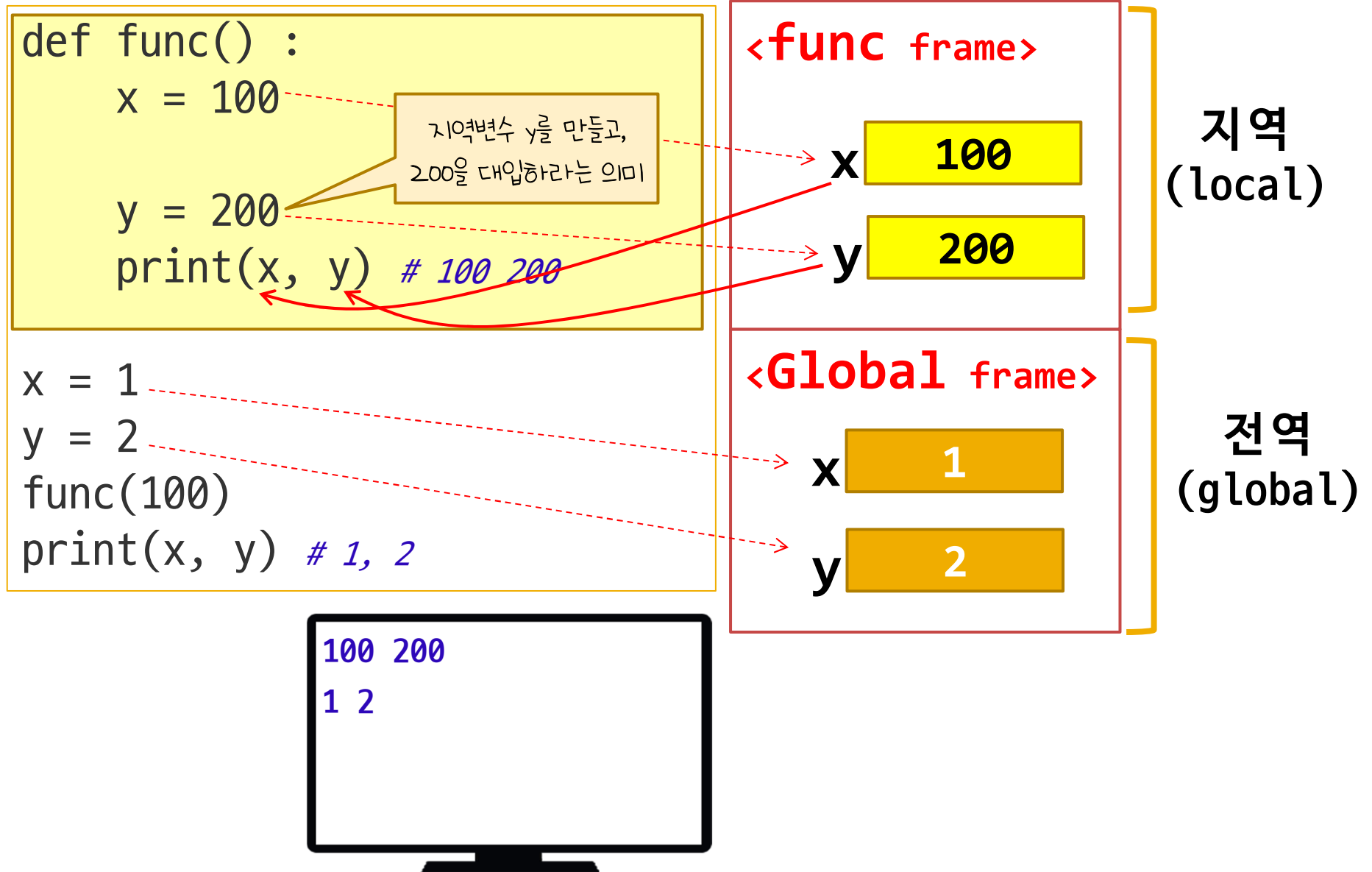
전역변수와 지역변수 - 전역변수 값 참조



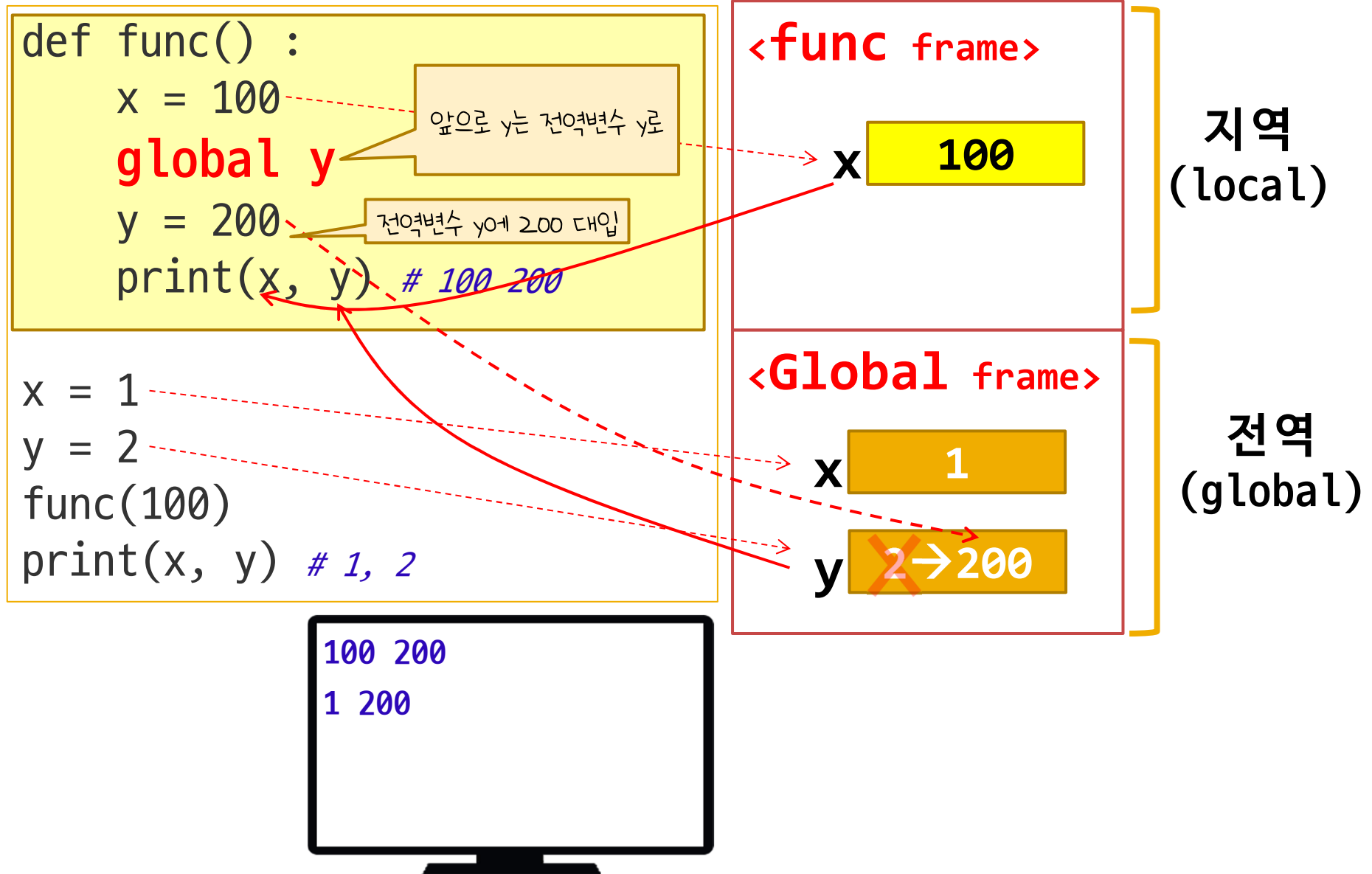
전역변수와 지역변수 - 매개변수는 지역변수



전역변수와 지역변수 - 지역에서 전역변수 값 변경



전역변수와 지역변수 - 지역에서 전역변수 값 변경



Recursion - 재귀호출

함수 $f(x)$ 의 정의 내에, 자기 자신 $f(x)$ 를 호출하는 코드가 있어, 계속 순환반복(recursion)됨.



Elena Filippova. Recursion. 2003
Acrylic on Canvas, 160 cm x 160 cm (63" x 63")

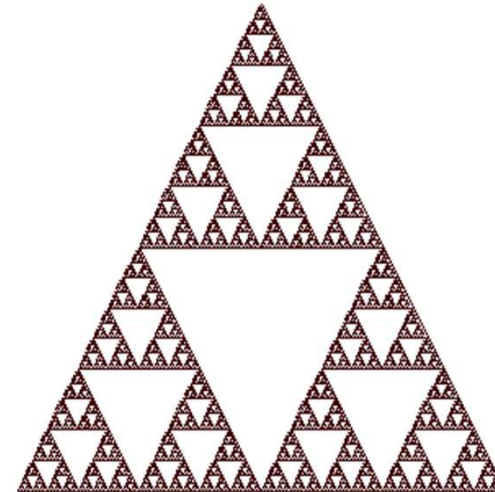
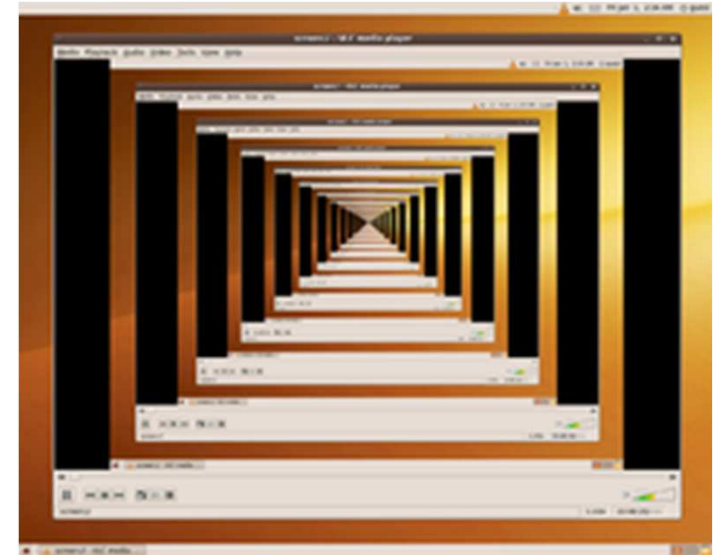
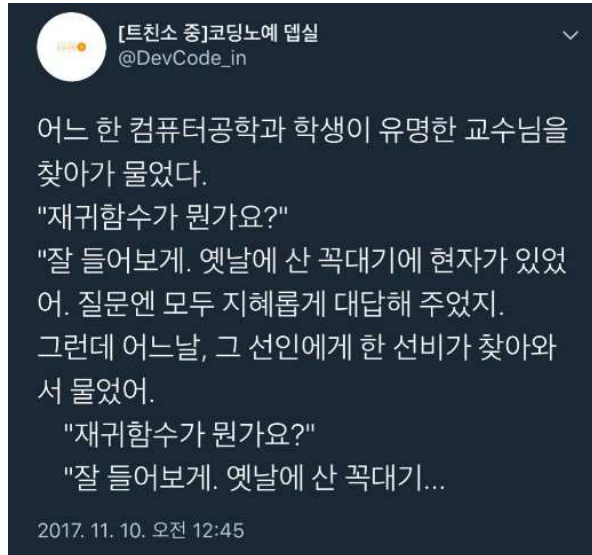
```
def f(n) :  
    print("hello")  
    f(n+1)  
    return
```

f(1)



iteration : for, while 등을 통해 이루어지는 반복
recursion : 재귀호출을 통해 이루어지는 순환

재귀(recursion)



재귀함수로 나무그리기

```
import turtle as t

def tree(depth, length) :
    t.write(str(depth))
    if depth > 6 : return

    t.forward(length) # 앞으로 length값만큼 선긋기

    t.left(40)        # 좌측방향으로 40도
    tree(depth+1, length*0.6) # 재귀호출

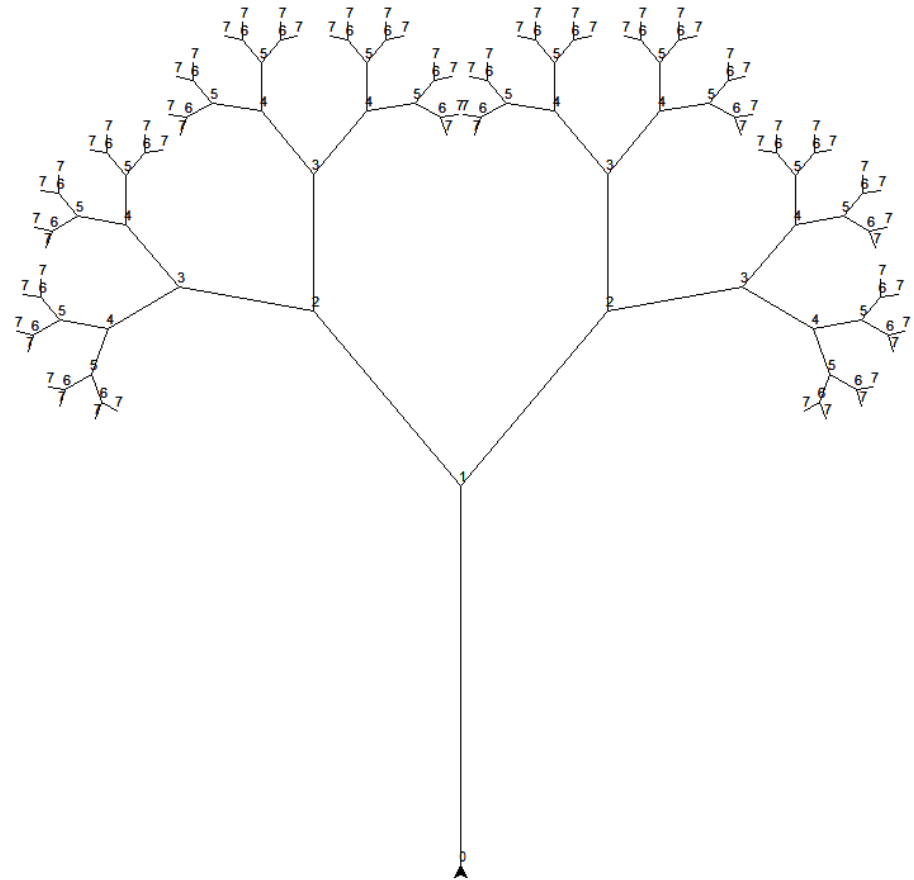
    t.right(80)       # 우측방향으로 80도
    tree(depth+1, length*0.6) # 재귀호출

    t.left(40)        # 좌측방향으로 40도
    t.backward(length) # 뒤로 length값만큼 선긋기

    t.left(90) #
    t.up()
    t.backward(300)
    t.down()

    tree(0, 300)

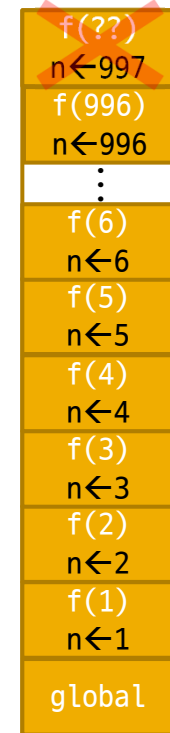
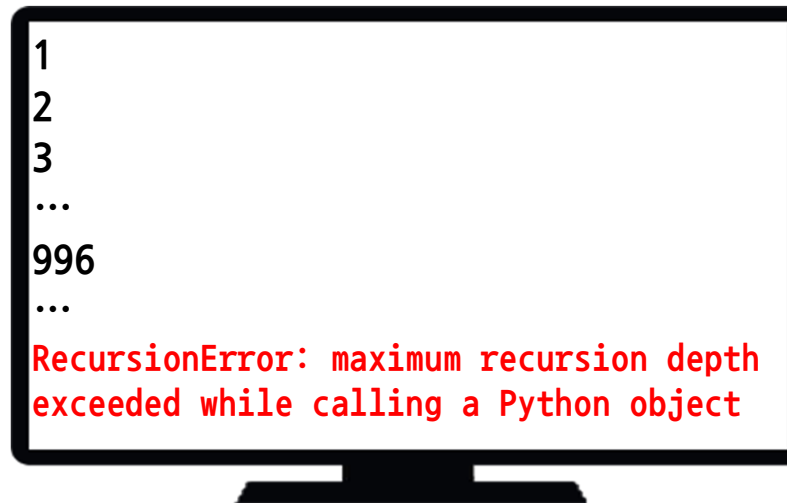
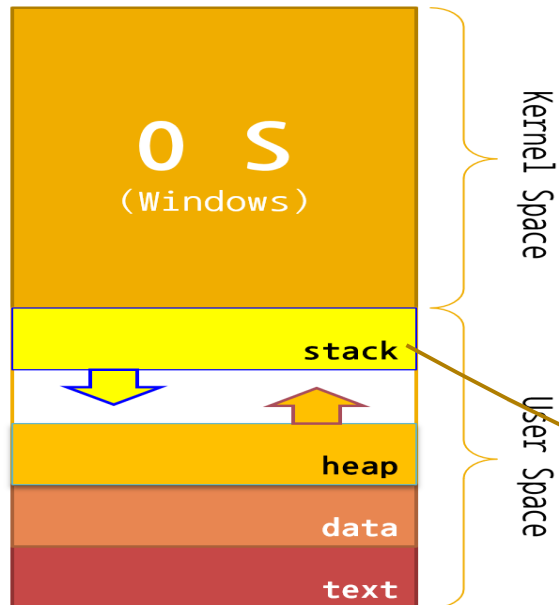
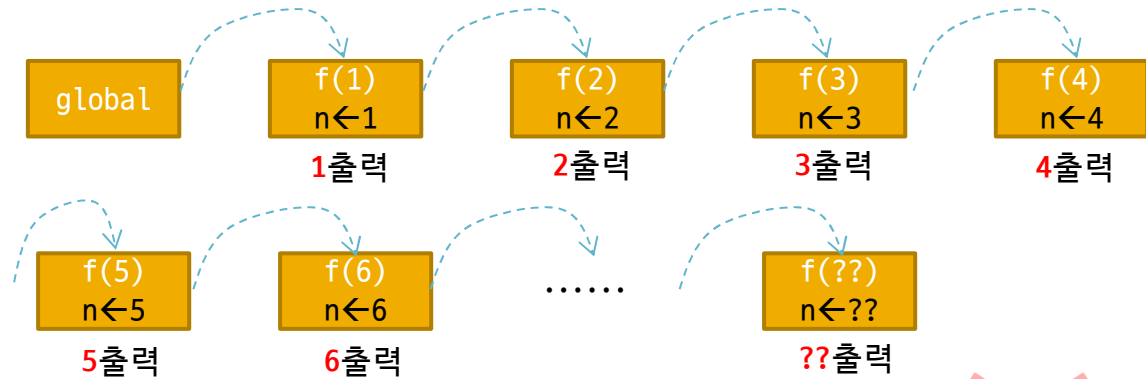
t.done()
```



무한 반복되는 경우

```
def f(n) :  
    print(n)  
    f(n+1)  
    return
```

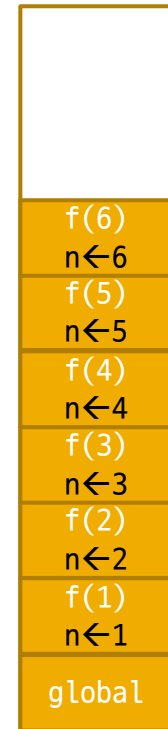
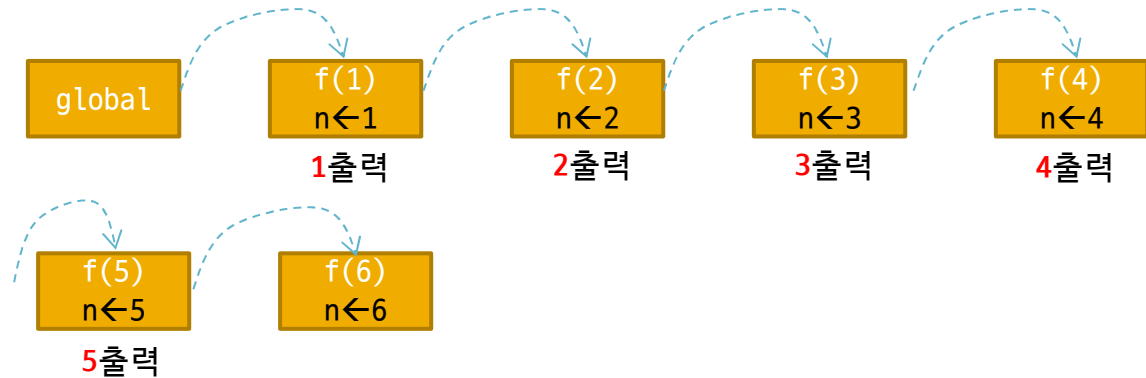
f(1)

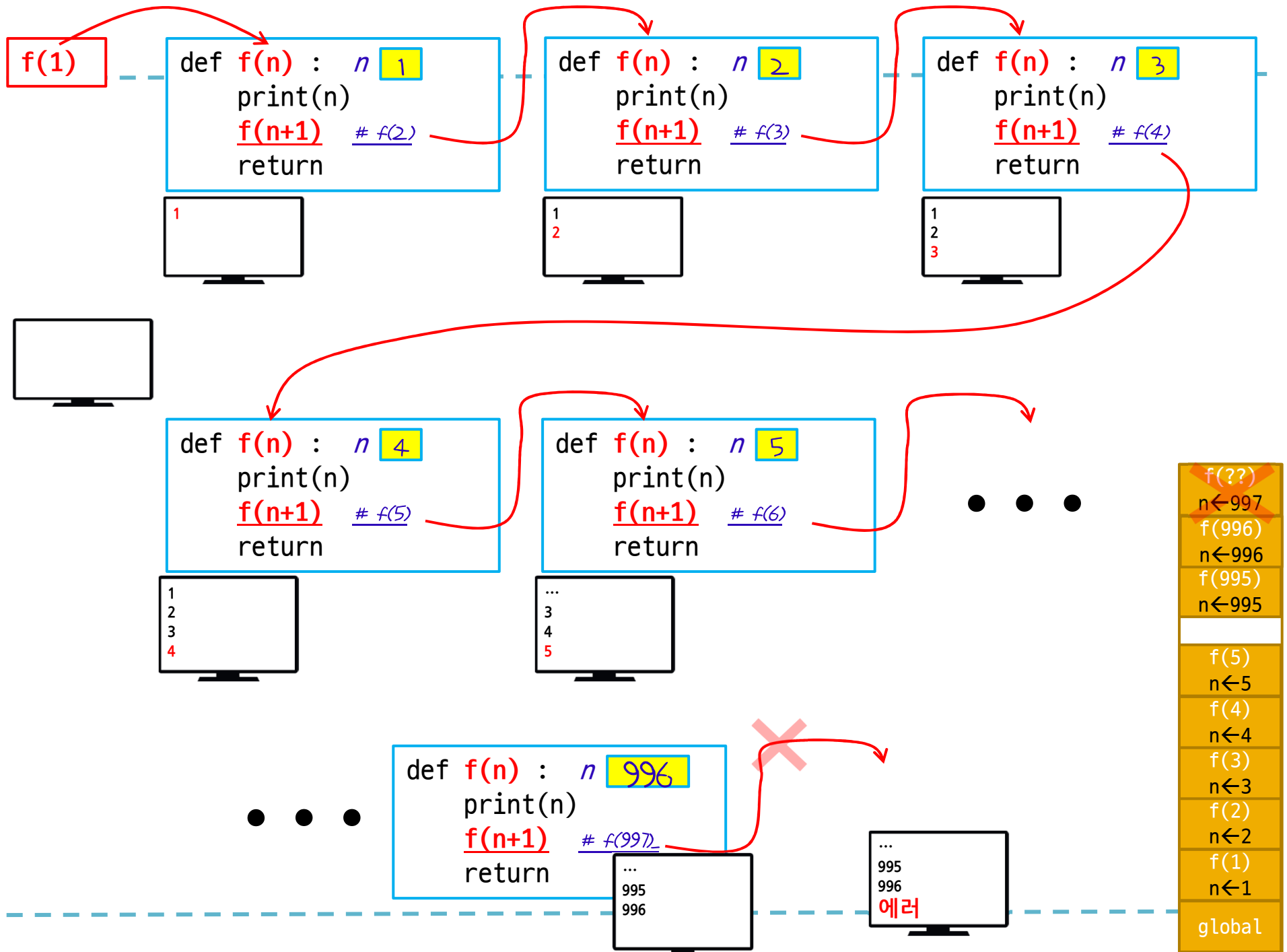


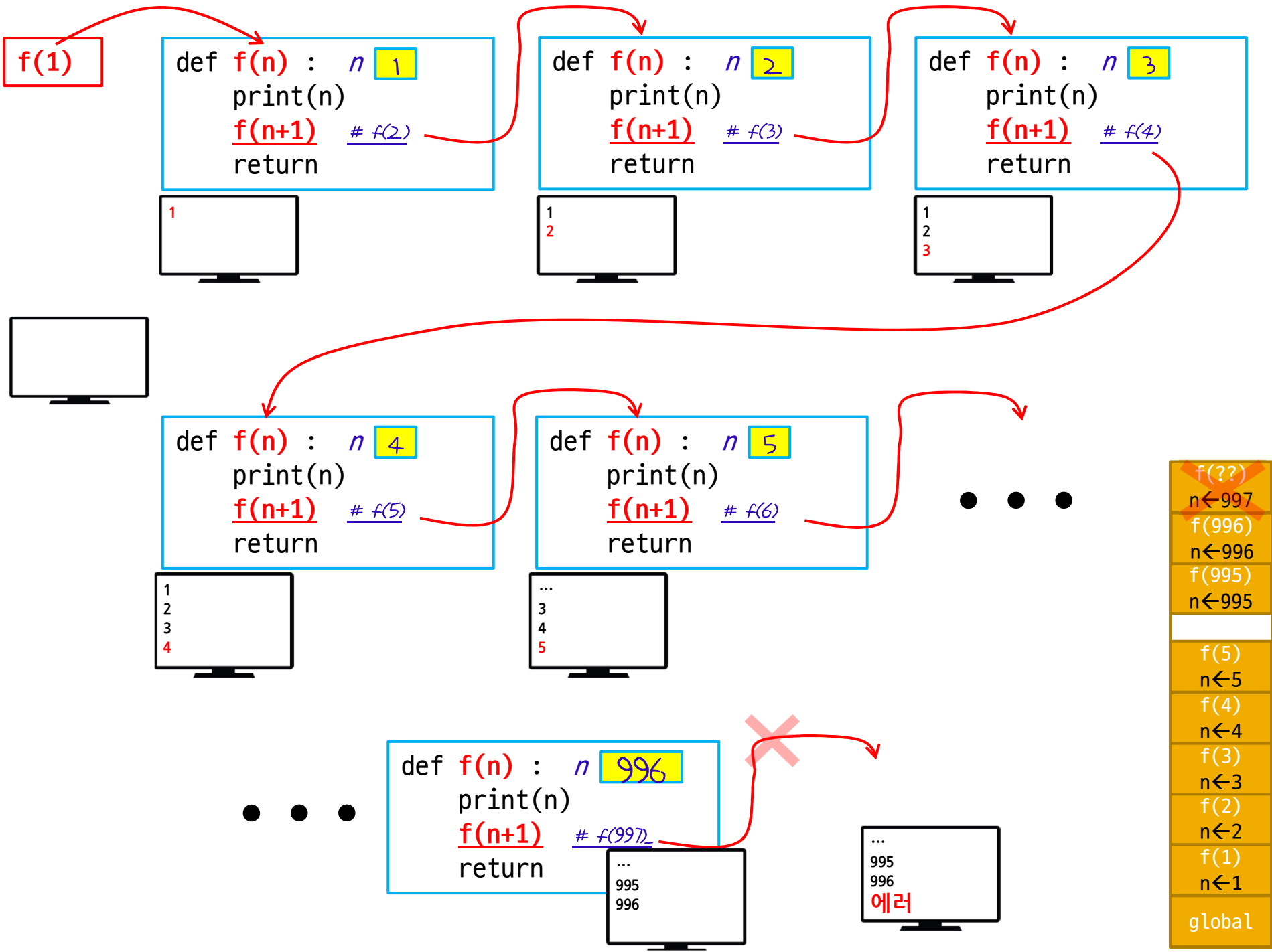
무한 반복을 피하기 위한 방법 #1

```
def f(n) :  
    if(n>5) : return  
    print(n)  
    f(n+1)  
    return
```

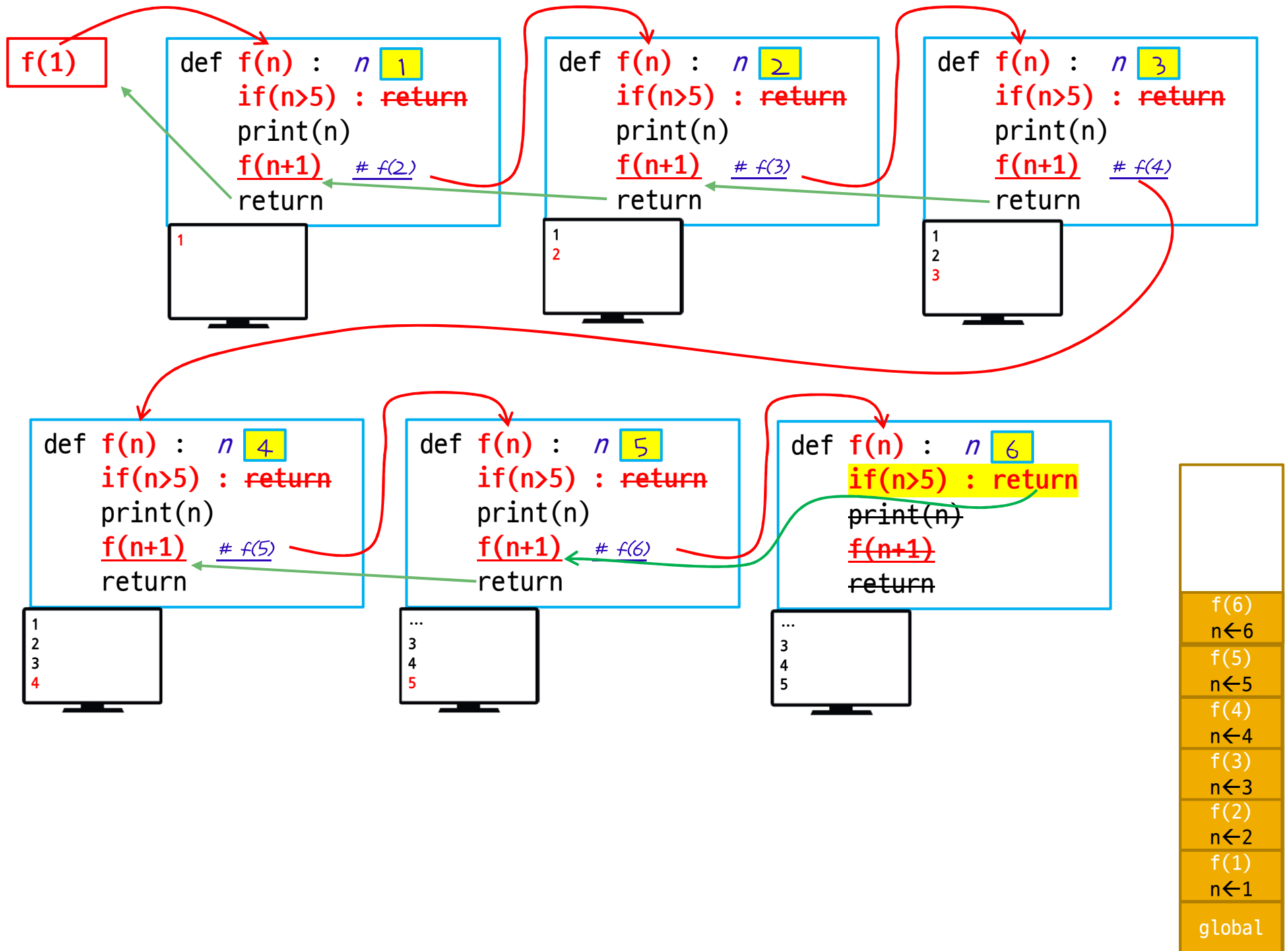
f(1)

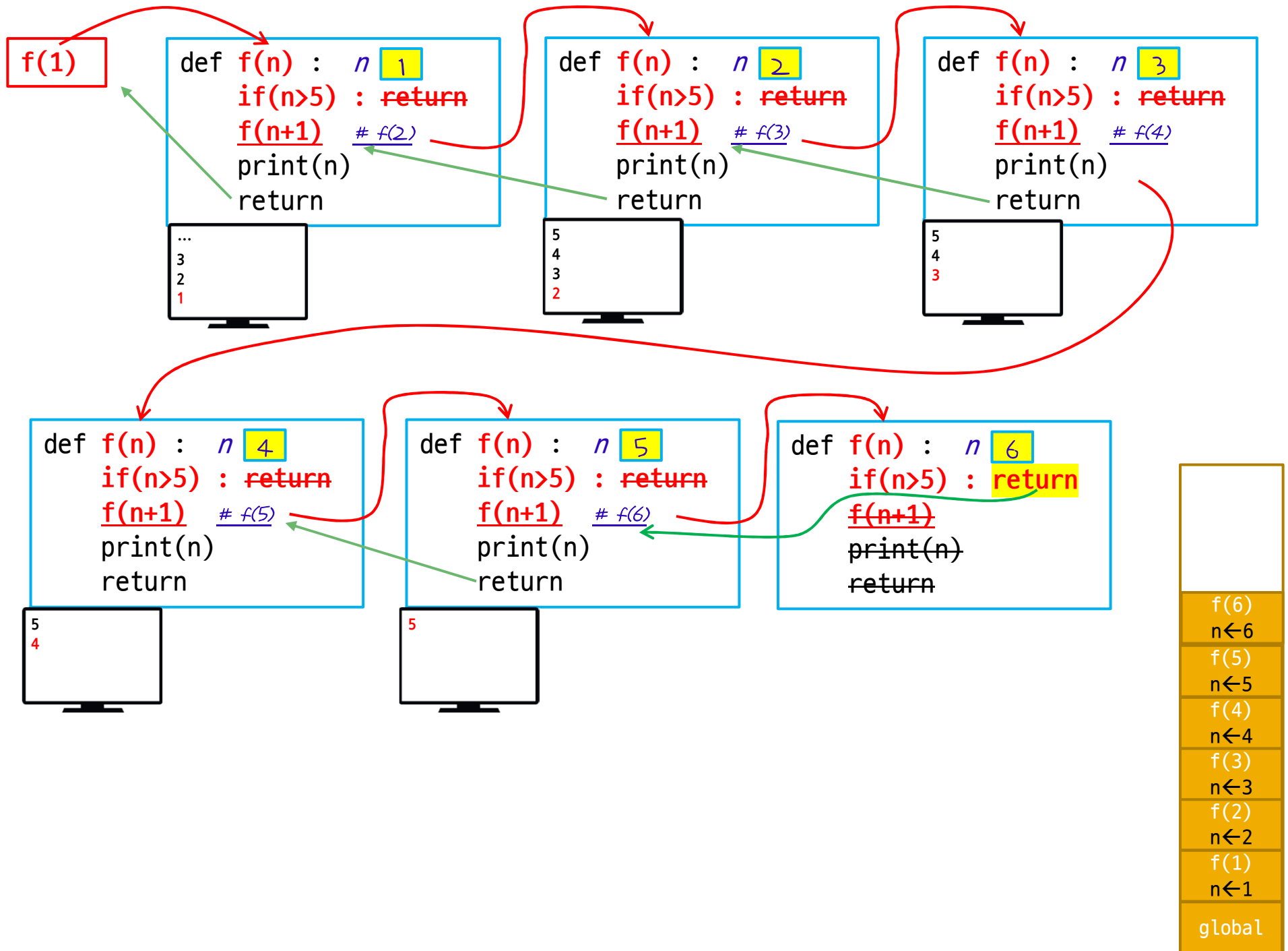


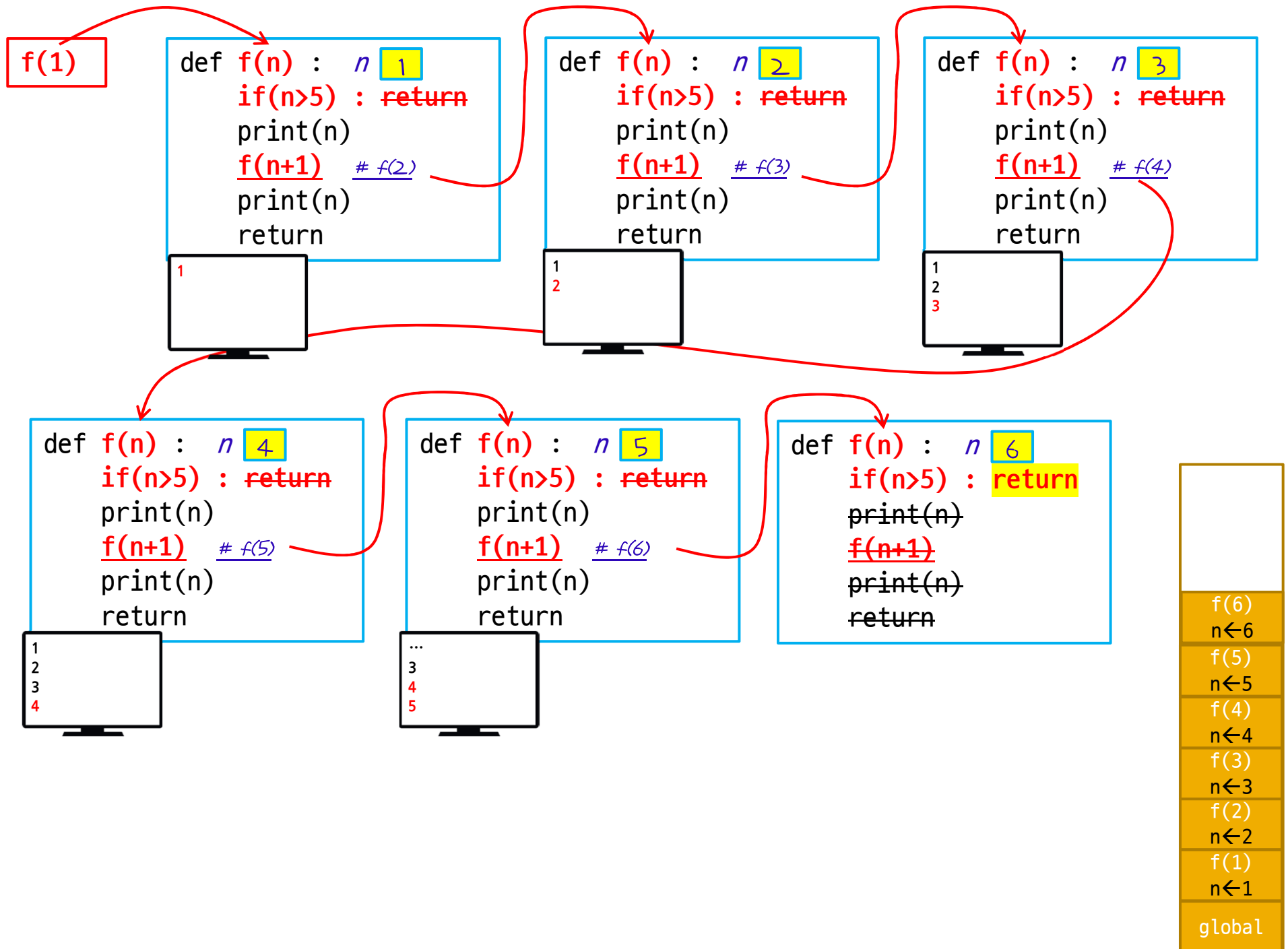


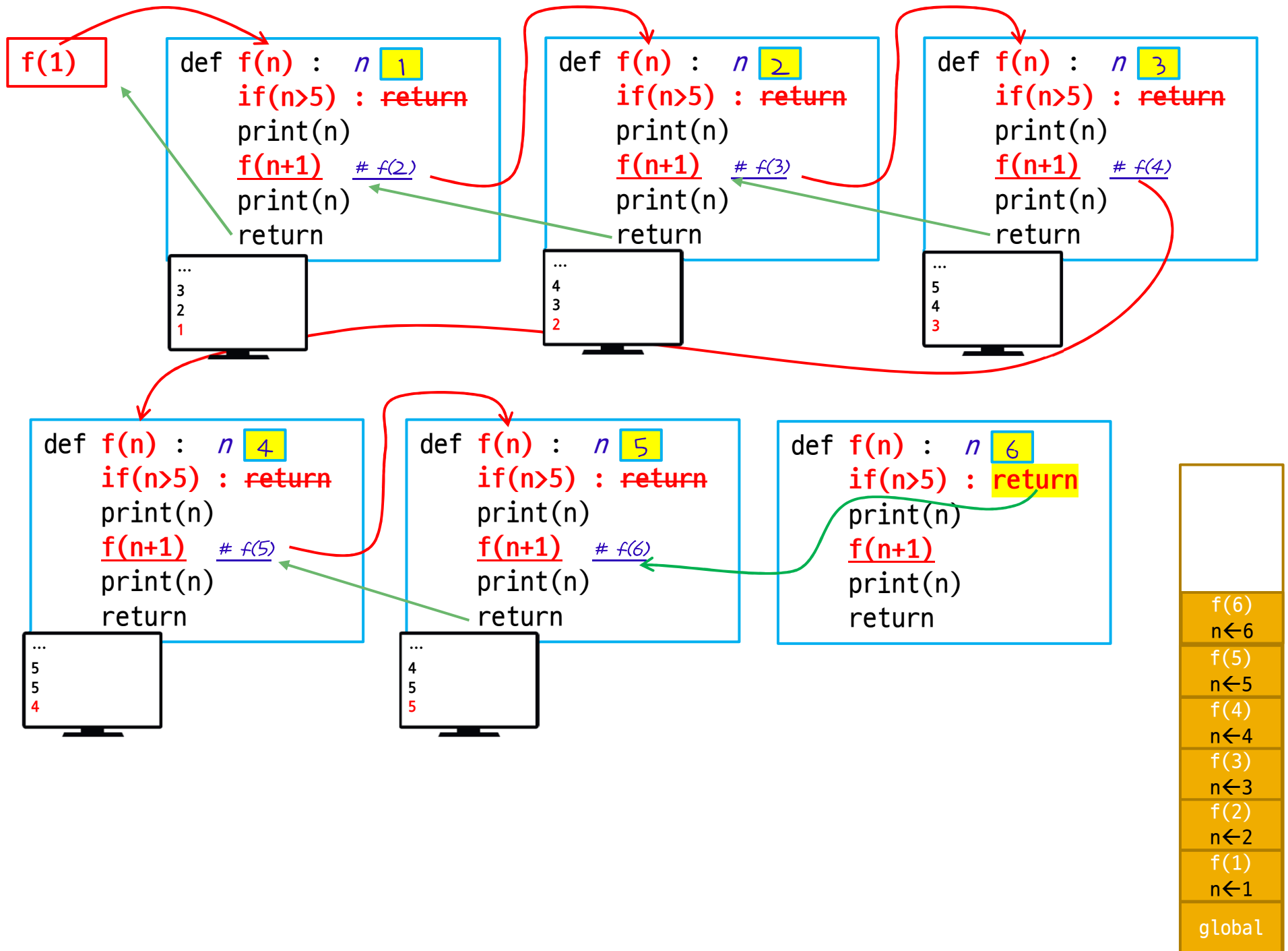


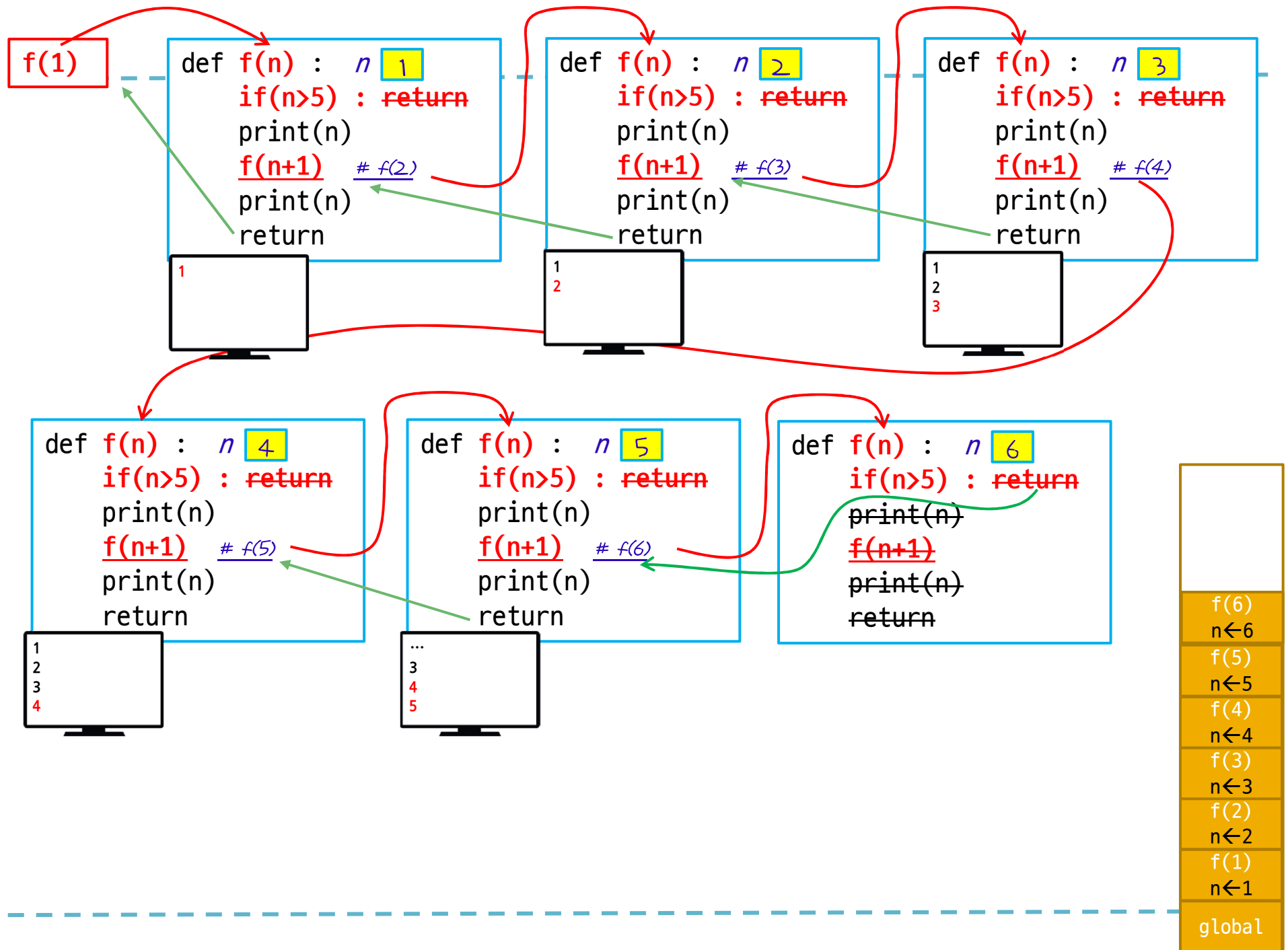
f(??)
n←997
f(996)
n←996
f(995)
n←995
f(5)
n←5
f(4)
n←4
f(3)
n←3
f(2)
n←2
f(1)
n←1
global



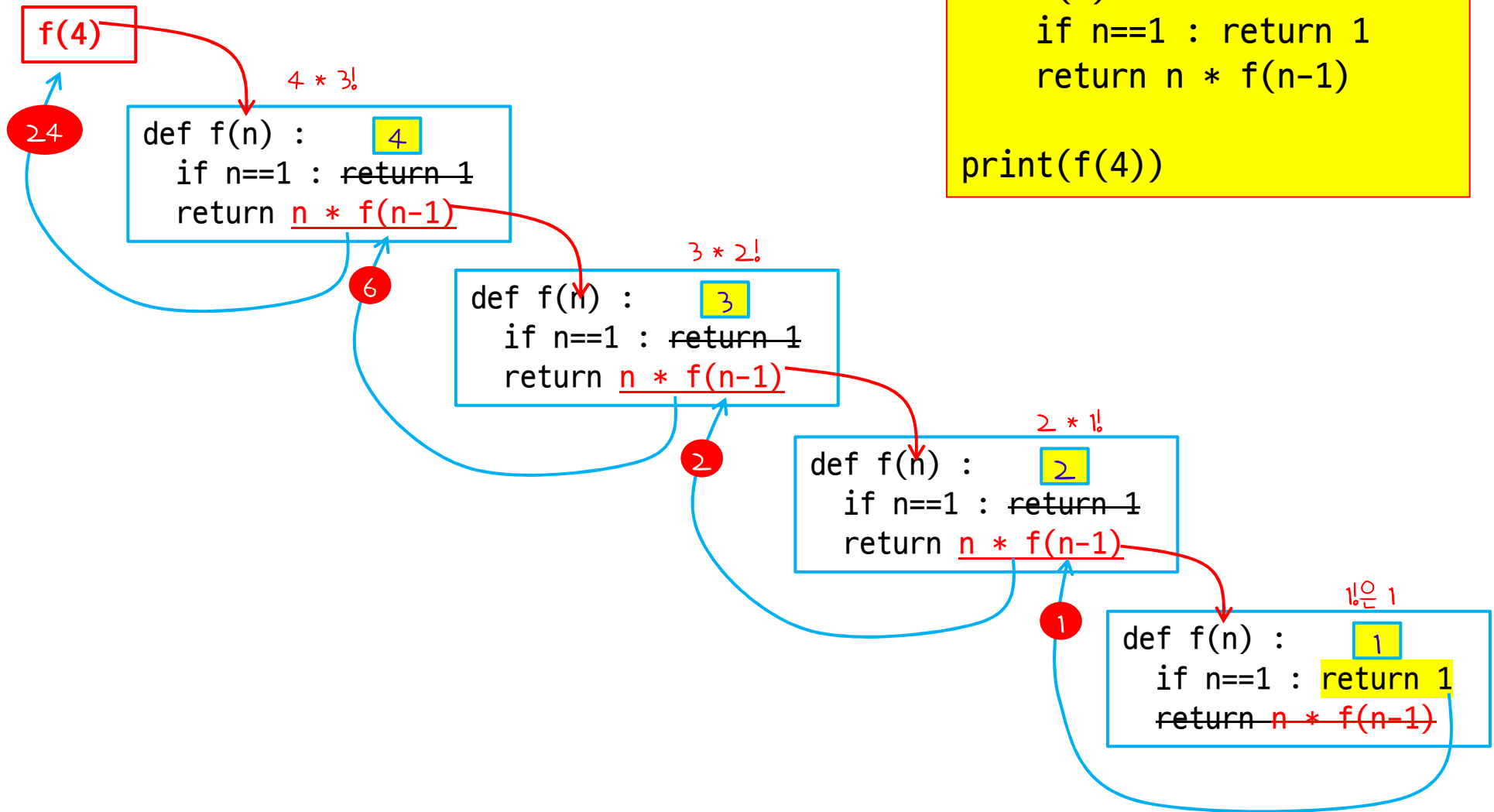








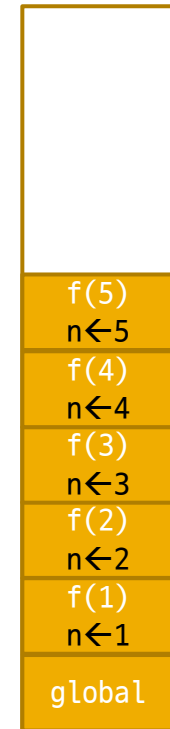
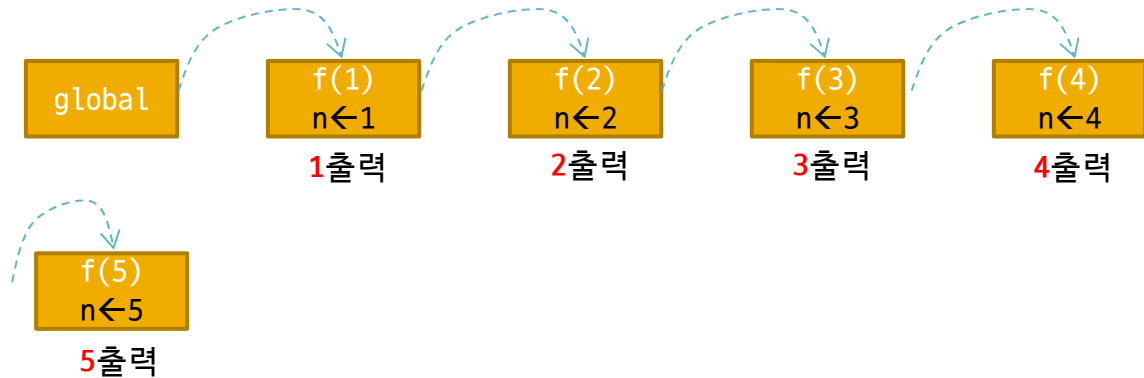

```
def f(n) :  
    if n==1 : return 1  
    return n * f(n-1)  
  
print(f(4))
```



무한 반복을 피하기 위한 방법 #2

```
def f(n) :  
    print(n)  
    if(조건) : f(n+1)  
    return
```

f(1)



무한 반복을 피하기 위한 방법 #2

