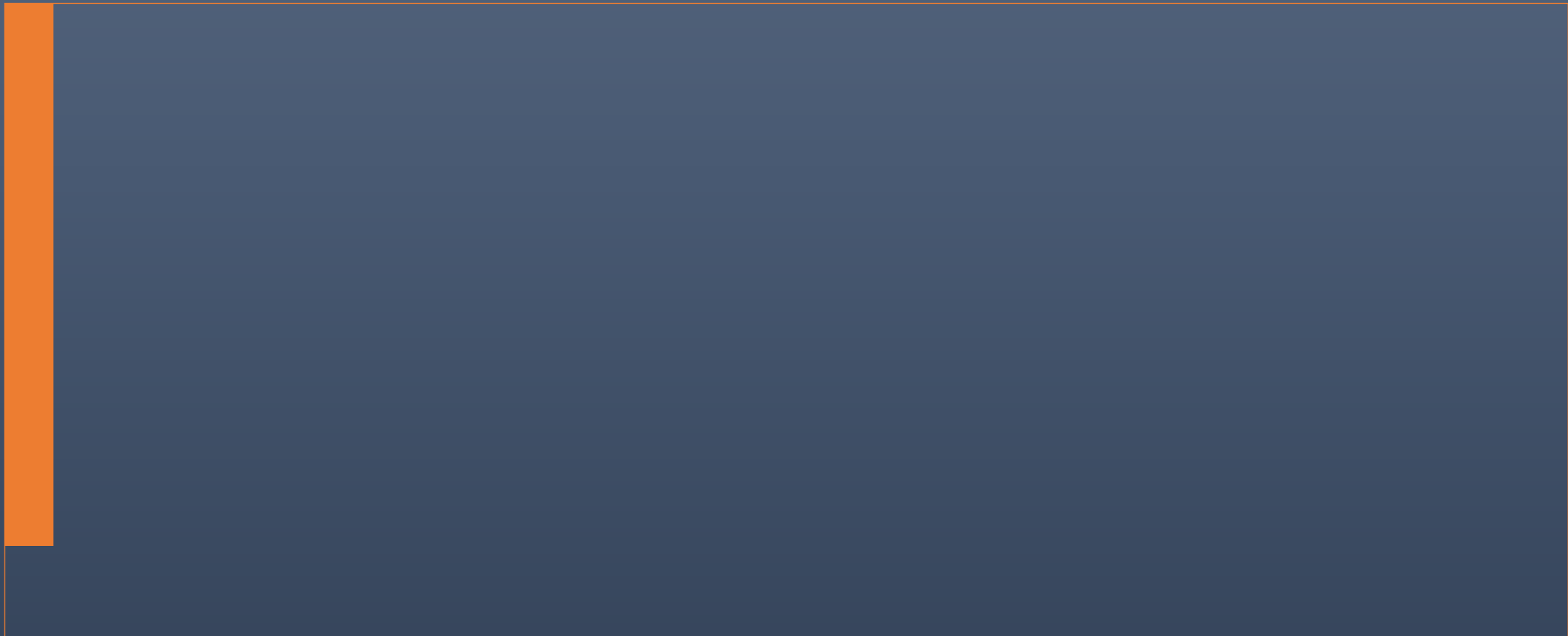




복습 자료



출력 함수 print()

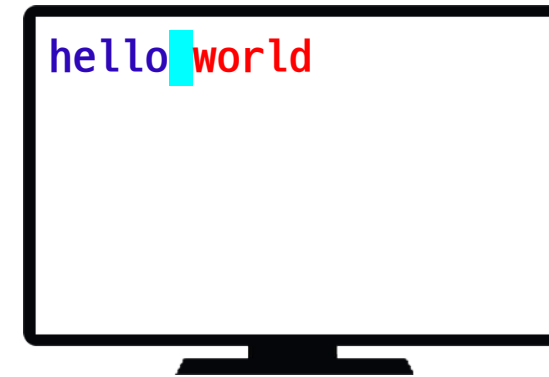
```
print("hello")  
print("world")
```

기본값은 줄을 바꿔서('\n') 출력



```
hello  
world
```

```
print("hello", end=" ")  
print("world")
```



```
hello world
```

```
print("사과", "배", "귤")  
print("사과", "배", "귤", sep="랑 ")  
print("사과", "배", "귤", sep="")
```



```
사과 배 귤  
사과랑 배랑 귤  
사과배귤
```

출력 서식 지정 : f-string

```
pi = 3.141592
```

```
print("파이는 pi입니다.")
```

따옴표("")안에 있으므로,
pi는 문자데이터임.

```
print(f"파이는 pi입니다.")
```

따옴표("")안에 있으므로,
pi는 문자데이터임.

```
print(f"파이는 {pi}입니다.")
```

따옴표("") 앞에 f를 쓰고,
중괄호({})를 해주면,
변수나 수식으로 처리됨.

```
print(f"파이는 {pi:.2f}입니다.")
```

파이는 pi입니다.

파이는 pi입니다.

파이는 3.141592입니다.

파이는 3.14입니다.

컨테이너(container) 자료형 객체

- 컨테이너(군집 자료형) 자료형
 - 여러 값을 하나의 객체로 묶어 관리하는 자료형들을 의미
 - 각 컨테이너는 특정한 목적과 특징을 가지고 있음



변경가능 : mutable
변경불가 : immutable)

시퀀스(Sequence) 자료형 객체

순서를 보장하고, 인덱스 접근이 가능한 객체

컨테이너	원소 접근	순서 보장	중복 허용	변경가능?
리스트(list) L = [<u>0</u> , 1]	인덱스 접근 L[0]	보장	가능	가능
튜플(tuple) T = (<u>0</u> , 1)	인덱스 접근 T[0]	보장	가능	불가능
문자열(str) SS = " <u>0</u> 1"	인덱스 접근 SS[0]	보장	가능	불가능
딕셔너리(dict) D = {0:" <u>zero</u> ", 1:"one"}	키 접근 D[0]	보장	키는 중복 불가	가능
세트(set) S = { <u>0</u> , 1}	직접 접근 불가 X	보장 안됨	불가	가능

컨테이너 중 시퀀스 자료형의 반복문

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

```
for i in range(len(L)):
    print(L[i], end=' ')
```

```
print()
```

```
for i in L:
    print(i, end=' ')
```

```
2 3 5 4 1
2 3 5 4 1
```

```
T = (2, 3, 5, 4, 1)
```

```
for i in range(len(T)):
    print(T[i], end=' ')
```

```
print()
```

```
for i in T:
    print(i, end=' ')
```

```
2 3 5 4 1
2 3 5 4 1
```

```
S = "Hello"
```

```
for i in range(len(S)):
    print(S[i], end=' ')
```

```
print()
```

```
for i in S:
    print(i, end=' ')
```

```
H e l l o
H e l l o
```

컨테이너 중 비시퀀스 자료형의 반복문

```
S = {2, 3, 5, 4, 1}
```

```
for i in range(len(S)):  
    print(S[i], end=' ')
```

불가

```
D = {1:'H', 2:'He', 3:'Li'}
```

```
for i in range(len(D)):  
    print(D[i], end=' ')
```

불가

```
S = {2, 3, 5, 4, 1}
```

```
for i in S:  
    print(i, end=' ')
```

가능

```
D = {1:'H', 2:'He', 3:'Li'}
```

```
for i in D:  
    print(i, end=' ')
```

가능

리스트의 차원 #1

`i = 5` # 0차원

`L1 = [1, 2, 3]` # 1차원

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

`print(L1)` # 1차원
`print(L1[0])` # 0차원

`print(L2)` # 2차원
`print(L2[0])` # 1차원
`print(L2[1][2])` # 0차원

i

5

L1

1	2	3
---	---	---

 [0] [1] [2]

L2 [0]

1	2	3
---	---	---

 [1]

4	5	6
---	---	---

 [0] [1] [2]

`[1, 2, 3]`

`1`

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

`[1, 2, 3]`

`6`

리스트의 차원 #2

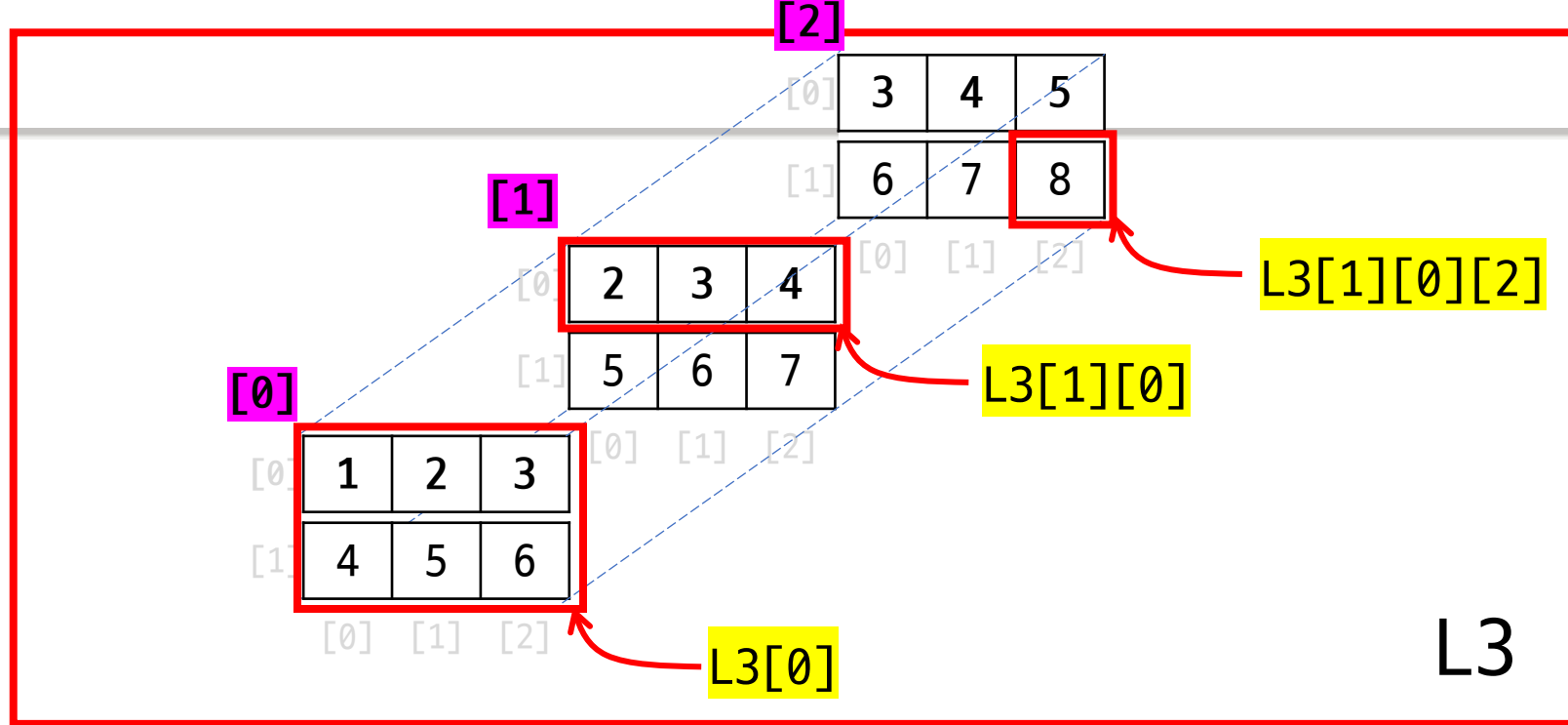
```
L3 = [[[1, 2, 3],  
        [4, 5, 6]],  
       [[2, 3, 4],  
        [5, 6, 7]],  
       [[2, 3, 4],  
        [5, 6, 7]]]
```

```
print(L3)           # 3차원
```

```
print(L3[0])        # 2차원
```

```
print(L3[1][0])     # 1차원
```

```
print(L3[1][0][2])  # 0차원
```



```
[[[1, 2, 3], [4, 5, 6]], [[2, 3, 4], [5, 6, 7]],  
 [[2, 3, 4], [5, 6, 7]]]
```

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

```
[2, 3, 4]
```

```
4
```


리스트의 연산

```
S1 = 'Hello'  
S2 = 'World'
```

```
S3 = S1 + S2  
S4 = S1 * 2
```

```
print(S3)  
print(S4)
```

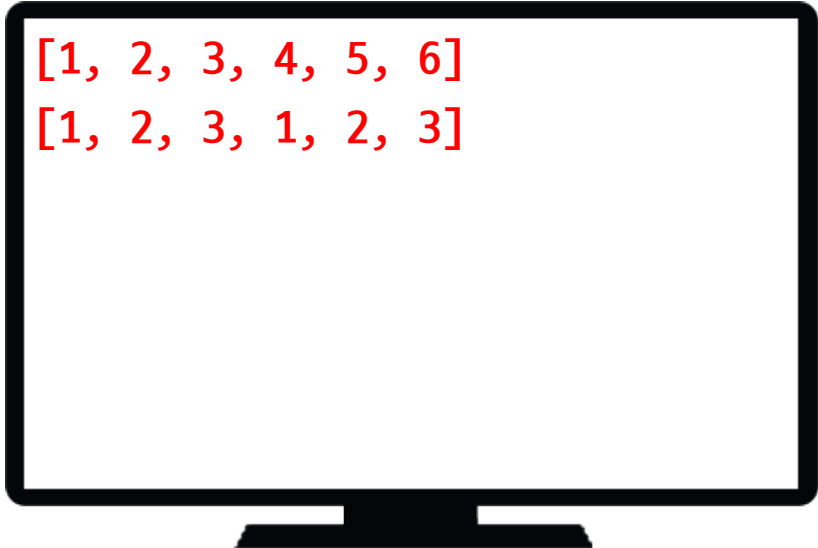


```
HelloWorld  
HelloHello
```

```
L1 = [1, 2, 3]  
L2 = [4, 5, 6]
```

```
L3 = L1 + L2  
L4 = L1 * 2
```

```
print(L3)  
print(L4)
```



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

리스트의 append() 메서드 #1

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

```
L = [1]               # [1]  
  
L.append(2)           # [1, 2]  
  
L.append([3, 4])      # [1, 2, [3, 4]]  
  
L.append([5, 6])      # [1, 2, [3, 4], [5, 6]]
```

```
L=[]                  # []  
  
A = [1, 2, 3]  
B = [4, 5, 6]  
  
L.append(A)           # [[1, 2, 3]]  
  
L.append(B)           # [[1, 2, 3], [4, 5, 6]]
```

```
L=[]  
  
A = [1, 2, 3]  
B = [4, 5, 6]  
  
L.append(A)           # [[1, 2, 3]]  
  
L = L + B             # [[1, 2, 3], 4, 5, 6]
```

리스트의 append() 메서드 #2

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

```
L.append(10)
```

결과 : *[[1, 2, 3], [4, 5, 6], 10]*

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

```
L.append([10, 20, 30])
```

결과 : *[[1, 2, 3], [4, 5, 6], [10, 20, 30]]*

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

```
L[0].append(10)
```

결과 : *[[1, 2, 3, 10], [4, 5, 6]]*

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

```
L[1].append([10, 20, 30])
```

결과 : *[[1, 2, 3], [4, 5, 6, [10, 20, 30]]]*

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

```
L[1].append(10)
```

결과 : *[[1, 2, 3], [4, 5, 6, 10]]*

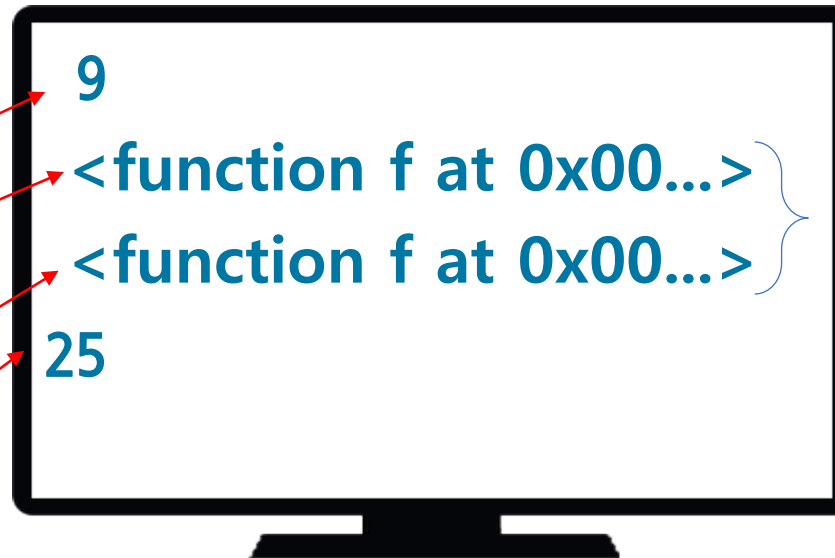
f()와 f의 차이점

- **f()** 는 함수 f를 **실행(호출)하라**는 의미이다.
- **f** 는 함수 f 자체이다. 즉, **함수 객체 자체**의 의미이다.

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

```
print(f(3))  
print(f)
```

```
a = f  
print(a)  
print(a(5))
```



a와 **f**는 같은
객체를 가리키므로
같은 값(ID)이 출력

f □ □ □ □ □ a □ □ □ (□ □ □ , a □ f □ □ □ □ □ .)

함수도 객체다.

```
L = [2, 7, 3, 5, 8]
```

```
F = [sum, len, max, min]
```

함수 객체 리스트



```
[sum, len, max, min]
```

```
for f in F:
```

```
    print( f(L) )
```

```
25 # sum(L)
```

```
5  # len(L)
```

```
8  # max(L)
```

```
2  # min(L)
```

콜백 함수와 고차 함수

- 콜백 함수(CallBack Function)란?
 - 다른 함수에 인자로 전달되어 특정 작업이 완료된 후 호출되는 함수(아래에서 `on_greet_complete`)
- 고차 함수(Higher-Order Function)란?
 - 다른 함수를 인자로 받거나, 또는 반환값으로 함수를 반환하는 함수(아래에서 `greet`가 고차 함수)

고차 함수(다른 함수를 인자로 받는 함수)

```
def greet(name, callback):  
    print(f"안녕, {name}!")  
    callback() # 작업이 끝난 후 콜백 함수 호출
```

콜백 함수 정의

```
def on_greet_complete():  
    print("인사 끝...")
```

고차 함수 호출(실행)에, 콜백 함수를 인자로 사용

```
greet("민찬", on_greet_complete)
```

`on_greet_complete` : 함수 객체를 전달인자로 사용한다는 의미
(`on_greet_complete()`로 쓰지 않았음.)

안녕, 민찬!
인사 끝...

콜백함수와 고차함수 사용 예 #1

```
L = ["1", "12", "12345"]
```

```
R1 = list(map(int, L))    # L의 모든 원소를 정수로
```

```
R2 = list(map(len, L))   # L의 모든 원소의 길이
```

```
print(R1)    # [1, 12, 12345]
```

```
print(R2)    # [1, 2, 5]
```



```
[1, 12, 12345]
```

```
[1, 2, 5]
```

map()은 고차함수 : **int()**, **len()**을 전달인자로 받아 사용했으므로

int(), **len()**은 콜백함수 : **map()**의 전달인자로 사용되었으므로

콜백함수와 고차함수 사용 예 #2

```
def odd(x) :  
    if x % 2 == 1 : return True  
    else          : return False
```

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

```
L_odd = list(filter(odd, L)) # 홀수만 걸러냄
```

```
print(L)  
print(L_odd)
```

map(), filter()은 고차함수
int(), odd()은 콜백함수

2 3 8 4 5
[2, 3, 8, 4, 5]
[3, 5]

filter()함수?

- 파이썬의 내장함수
- **특정 조건을 만족하는 요소들만 걸러내는 함수**

변수의 scope : 전역변수, 지역변수

```
def func(a) :
```

```
    global b    # b는 전역변수
```

```
    b = 20
```

```
    c = 30
```

```
    print(a, b, c)
```

```
a, b, c = 1, 2, 3
```

```
func(a)
```

```
print(a, b, c)
```

<func frame> 지역
(local)

a 1

b (전역변수 b)

c 30

<Global frame> 전역
(global)

a 1

b 2 → 20

c 3

1 20 30
1 20 3



람다함수(lambda)



이름이 없이 간단하게 사용하는 함수

람다(Lambda) 함수

- 람다함수?
 - 이름 없이 사용할 수 있으며, 간단하게 정의할 수 있는 함수에 활용
 - 복잡한 함수는 기존의 def 키워드로 함수 정의

- 정의형식

lambda 매개변수(들) : 표현식(연산식)

- 예시

일반함수

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

```
result = odd(3)
```

```
print(result)
```

5

람다함수

```
odd = lambda x : (2 * x - 1)  
# 람다함수 객체를 odd 변수에 할당
```

```
print(odd(3))
```

람다 함수를 자주 사용하는 사례

사례1) `map()` 함수와 같이 사용해 각 원소의 제곱

```
L = [1, 2, 3, 4, 5]
squared = list(map(lambda x : (x ** 2), L))
print(squared)    # [1, 4, 9, 16, 25]
```

[1, 4, 9, 16, 25]

사례2) `filter()` 함수와 같이 사용해 짝수만 추출

```
L = [1, 2, 3, 4, 5]
even_data = list(filter(lambda x : (x % 2 == 0), L))
print(even_data)    # [2, 4]
```

[2, 4]

사례3) `sort()` 함수의 `key` 콜백함수로 활용

```
L = [[1, 5], [2, 3], [1, 3]]
L.sort(key=lambda x : (x[0], -x[1]))
print(L)    # [[1, 5], [1, 3], [2, 3]]
```

0번열을 기준으로 오름차순,
1번열을 기준으로 내림차순

[[1, 5], [1, 3], [2, 3]]

람다 함수로 L.sort()의 콜백함수 역할 수행

이름 [0]	점수 [1]	나이 [2]
민지	85	19
혜인	85	15
다니엘	85	18
하니	87	19
헤린	84	17

순위의 기준

1순위 : **점수가 높은 사람(점수[1]의 내림차순)**

→ 동점이면,

2순위 : **나이가 적은 사람(나이[2]의 오름차순)**

```
L = ["민지", 85, 19],  
     ["혜인", 85, 15],  
     ["다니엘", 85, 18],  
     ["하니", 92, 19],  
     ["헤린", 92, 17]]
```

```
L.sort(key=lambda x : (-x[1], x[2]))
```

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

```
['헤린', 92, 17]  
['하니', 92, 19]  
['혜인', 85, 15]  
['다니엘', 85, 18]  
['민지', 85, 19]
```



재귀함수



비선형 공간 탐색에 재귀함수가 유용함.

올바른 재귀함수 설계

재귀호출 무한번 수행

오류 발생

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

f(1)

오류가 발생하지
않도록
base case 추가
(어떤 경우가 되면,
재귀호출을 멈춰라.)

```
def f(n) :  
    if n > 3 :  
        return  
    else :  
        print(n)  
        f(n+1)  
        return
```

f(1)

→ base case : $n > 3$

base case? 재귀 호출을 끝낼 조건을 뜻함.

→ general case : $n \leq 3$

general case? 재귀 호출을 진행할 조건



else 문을 생략해서 코드를 간결하게...

```
def f(n) :  
    if n > 3 :  
        return  
  
    print(n)  
    f(n+1)  
    return
```

f(1)

→ base case : $n > 3$

base case? 재귀 호출을 끝낼 조건을 뜻함.

→ general case : $n \leq 3$

general case? 재귀 호출을 진행할 조건

단일 재귀함수의 흐름

