

# Python 4주차

## tuple, set, dictionary

서울대학교 전기공학부  
명예교수 성원용



# TODAY

- have seen variable types: `int`, `float`, `bool`, `string`
- introduce new **compound data types**
  - tuples
  - lists
- idea of aliasing
- idea of mutability
- idea of cloning

# PYTHON COMPOUND DATA TYPES

- Python 용도와 형태별로 다양한 array의 형태가 존재한다.
- 기본적으로 파이썬에 내장된 array는
  - tuple – 한번 만들면 내용을 바꿀 수 없다.
  - list – 내용을 바꿀 수 있다. ~~Linked의 형태라 search에 불리.~~
  - set – 집합. 내용에 중복되는 것이 없다. 합집합, 교집합 등.
  - dictionary – key와 value 쌍으로 되어 있다.
- 내부의 element가 각자 자유로운 data type을 가질 수 있다 (heterogeneous)
  - 쓰기는 편리하지만, 속도 등에서는 불리
- 추후에 numpy array는 따로 다룸.

# RECALL - STRINGS

- **sequence** of case sensitive characters
- can **slice** strings using `[start:stop:step]`
- if give two numbers, `[start:stop]`, `step=1` by default
- you can also omit numbers and leave just colons

```
s = "abcdefgh"
```

```
s[3:6] → evaluates to "def", same as s[3:6:1]
```

```
s[3:6:2] → evaluates to "df"
```

```
s[:] → evaluates to "abcdefgh", same as s[0:len(s):1]
```

```
s[::-1] → evaluates to "hgfedcba", same as s[-1:- (len(s)+1) :-1]
```

```
s[4:1:-2] → evaluates to "ec"
```

*If unsure what some  
command does, try it  
out in your console!*

# TUPLE : ()

- 특징 : 처음 선언한 이후 바꿀 수 없음 (immutable).
- 자료형을 섞거나 튜플 안에 다른 array를 이용할 수 있음
- ex) t = (1, 2, 'a', 'b', (3, 4))

```
In [3]: t=(1,2,'a','b',(3,4))  
        print(t[4])  
        print(t[4][0])  
  
        (3, 4)  
        3
```

- 위 예시처럼 t[위치] 로 불러낸 array를 array이름으로 취급하듯 t[위치][내부 위치] 식으로 계속해서 안쪽의 위치를 색인함

# TUPLE : () 사용하기

- 위치로 색인 : t[찾고자 하는 위치]
- 값으로 색인 : t.index(찾고자 하는 값)
- 값으로 카운트 : t.count(찾고자 하는 값)
- 슬라이싱 : t[시작점 : 끝점(미만)]
- \*빈칸으로 두면 각각 [맨 처음:맨 끝]
- 덧셈 : t1 + t2
- 곱하기 : t \* 횟수
- 길이 : len(t)
- 총계 : sum(t)

```
t1=(1,2,3,1,5)
t2=('a','b','c')
print(t1[2], t2[-1]) #indexing
print(t1[1:4])       #slicing
print(t1+t2)         #concatination
print(t2*3)          #repeat
print(len(t1))       #length
print(t1.count(1))   #count element
print(t2.index('b')) #indexing
print(sum(t1))       #summation
```

```
3 c
(2, 3, 1)
(1, 2, 3, 1, 5, 'a', 'b', 'c')
('a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c')
5
2
1
12
```

# TUPLES

- an ordered sequence of elements, can mix element types
- cannot change element values, **immutable**
- represented with parentheses

`te = ()` *empty tuple*  
`t = (2, "mit", 3)`

`t[0]` → evaluates to 2

`(2, "mit", 3) + (5, 6)` → evaluates to `(2, "mit", 3, 5, 6)`

`t[1:2]` → slice tuple, evaluates to `("mit", )`

`t[1:3]` → slice tuple, evaluates to `("mit", 3)`

`len(t)` → evaluates to 3

`t[1] = 4` → gives error, can't modify object

*remember strings?*

*extra comma means a tuple with one element*

# TUPLES

- conveniently used to **swap** variable values

```
x = y
```

```
y = x
```



```
temp = x
```

```
x = y
```

```
y = temp
```



```
(x, y) = (y, x)
```



- used to **return more than one value** from a function

```
def quotient_and_remainder(x, y):
```

```
    q = x // y
```

```
    r = x % y
```

```
    return (q, r)
```

integer  
division

```
(quot, rem) = quotient_and_remainder(4, 5)
```



- `def quotient_and_remainder(x, y):`
- `q = x // y`
- `r = x % y`
- `z = 1`
- `return q, r`
- `#     return (q,r)`
- `q, r = quotient_and_remainder(7,5)`
- `print(q, r)`
- `print((q,r))`
- `print(type((quotient_and_remainder(4,5))))`

- `a, b = 1, 3`
- `print(type(a))`
- `print(a, b)`
- `print(type((a, b)))`
- `print((a,b))`

```
1 2
(1, 2)
<class 'tuple'>
<class 'int'>
1 3
<class 'tuple'>
(1, 3)
```







```
def test():  
    return 'abc', 100
```

source: `return_multiple_values.py`

In Python, comma-separated values are considered tuples without parentheses, except where required by syntax. For this reason, the function in the above example returns a tuple with each value as an element.

**Note that it is actually the comma which makes a tuple, not the parentheses.** The parentheses are optional, except in the empty tuple case, or when they are needed to avoid syntactic ambiguity.

Built-in Types - Tuples — Python 3.7.4 documentation

aTuple: ( <sup>ints</sup>  <sup>strings</sup>  ) ,   ) ,   ) )

## ■ can **iterate** over tuples




```
aTuples = ((3, 'Sung'), (2, 'Samsung'), (1, 'SNU'), (0, 'Korea'))
```




```
def get_data(aTuples):
    nums = ()
    words = ()
    for t in aTuples:
        nums = nums + (t[0],)
        if t[1] not in words:
            words = words + (t[1],)

    min_n = min(nums)
    max_n = max(nums)
    unique_words = len(words)
    return (min_n, max_n, unique_words)
```

```
print(get_data(aTuples))
```

```
----
(0, 3, 4)
```

nums (    )

words (    )

if not already in words  
i.e. unique strings from aTuple

```
>>> tuple(['hello'])
('hello',)
```

```
>>> empty = ()
>>> singleton = 'hello', # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

```
t1 = (123)
print(t1) # int 타입
```

```
t2 = (123, )
print(t2) # tuple 타입
```

```
# 병합
a = (1, 2)
b = (3, 4, 5)
c = a + b
print(c) # (1, 2, 3, 4, 5)
```

```
# 반복
d = a * 3 # 혹은 "d = 3 * a" 도 동일
print(d) # (1, 2, 1, 2, 1, 2)
```

```
name = ("John", "Kim")
print(name)
# 출력: ('John', 'Kim')
```

```
firstname, lastname = ("John", "Kim")
print(lastname, ",", firstname)
# 출력: Kim, John
```

# TUPLE 은 못바꾸지만 바꾸고자 할 때

```
• x = ("apple", "banana", "cherry")  
  y = list(x)  
  y[1] = "kiwi"  
  x = tuple(y)  
  print(x)
```

```
thistuple = ("apple", "banana", "cherry")  
del thistuple  
print(thistuple) #this will raise an  
error because the tuple no longer exists
```

```
• thistuple = ("apple", "banana", "cherry")  
  y = ("orange",)  
  thistuple += y  
  
print(thistuple)      #('apple', 'banana', 'cherry', 'orange')
```

```
• thistuple = ("apple", "banana", "cherry")  
  y = list(thistuple)  
  y.remove("apple")  
  thistuple = tuple(y)
```

# LISTS

- **ordered sequence** of information, accessible by index
- a list is denoted by **square brackets**, [ ]
- a list contains **elements**
  - usually homogeneous (ie, all integers)
  - can contain mixed types (not common)
- list elements can be changed so a list is **mutable**

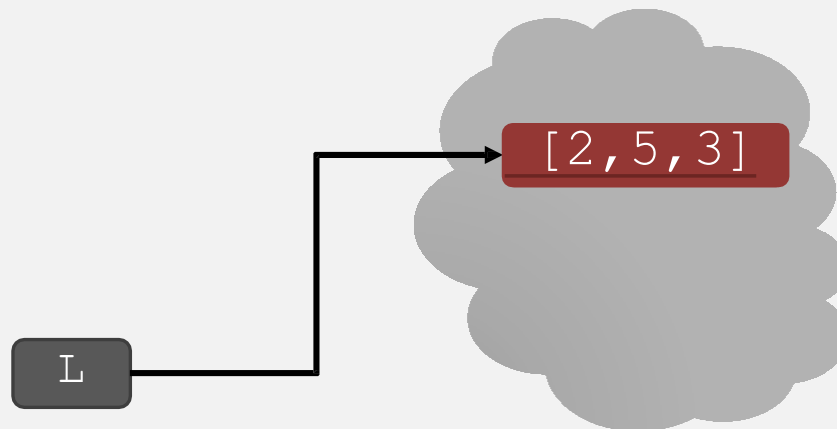
# CHANGING ELEMENTS

- lists are **mutable**!
- assigning to an element at an index changes the value

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

```
L[1] = 5
```

- L is now [2, 5, 3], note this is the **same object** L





```
L = [2, 1, 3]
print(id(L))          2038828464712
L[1] = 5
print(id(L))          2038828464712
L.append(5)
print(id(L))          2038828464712
y = L
print(id(L), id(y))   2038828464712
y.append(10)
print(id(L), id(y))   [2, 5, 3, 5, 10]
print(L)
print(y)
```



# INDICES AND ORDERING

```
a_list = []
```

*empty list*

```
L = [2, 'a', 4, [1, 2]]
```

`len(L)` → evaluates to 4

`L[0]` → evaluates to 2

`L[2]+1` → evaluates to 5

`L[3]` → evaluates to `[1, 2]`, another list!

`L[4]` → gives an error

```
i = 2
```

`L[i-1]` → evaluates to 'a' since `L[1]='a'` above

# LIST : [] 사용하기

- tuple에서 언급했던 내장함수들 모두 list에서 동일하게 사용할 수 있다. list는 수정이 가능하므로 추가적인 기능이 있다.
- 위치 삭제 : `del a[위치]` 혹은 `del a[시작점 : 끝점]`
- 값 제거 : `a.remove(값)`
- 맨 끝에 값 추가 : `a.append(값)`
- 중간에 값 삽입 : `a.insert(위치, 값)`
- pop : `a.pop()` //마지막 값을 반환하면서 list에서 삭제함
- 정렬 : `a.sort()`
- 역전 : `a.reverse()`
- 확장 : `a.extend(b)` //a의 뒤에 list b를 추가함

```
a = [1, 10, 5, 7, 6]
a.reverse()
print(a)  #[6, 7, 5, 10, 1]
```

# HOW IS THE PYTHON LIST IMPLEMENTED

- Pure static array
  - Dynamic array
  - Linked list
- 
- Quora 답
  - Dynamic arrays (미리 넉넉한 크기)
- 
- In Cpython, lists are arrays of pointers. Other implementations of Python may choose to store them in different ways.



# List of pointers

[1, 'text', 2, ...]

0	pointer
1	
2	
3	
4	
5	
6	
7	

pointer

동일하기

1

'text'

3

간혹

attribute는

라도 저장

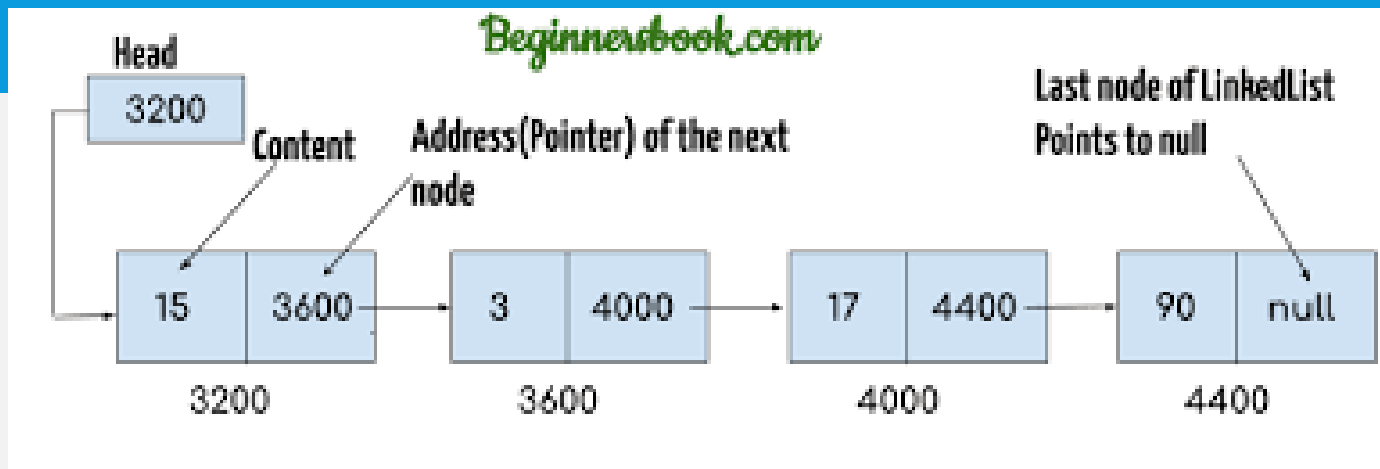
(가변 크기)

✓ 등비  
아까 때  
점

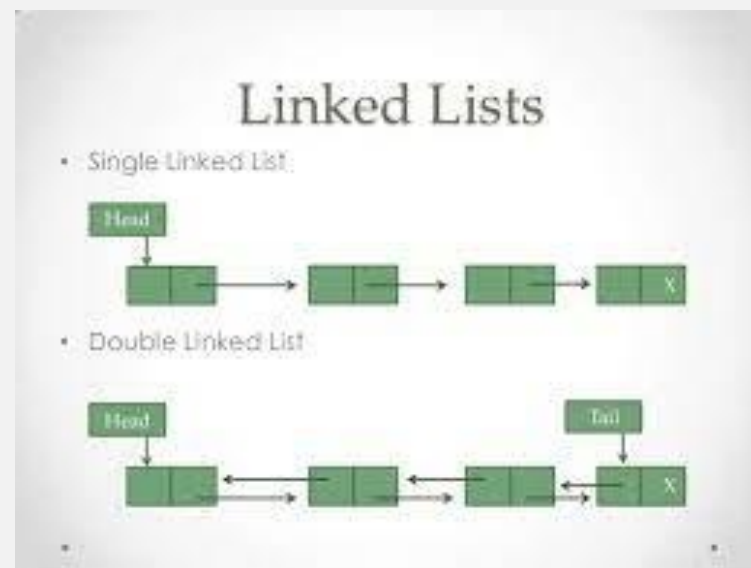
✓ append  
점

✓ Delete / Insert  
분

# LINKED LIST



insertion/deletion에 편리



# ITERATING OVER A LIST

- compute the **sum of elements** of a list
- common pattern, iterate over list elements

```
total = 0
for i in range(len(L)):
    total += L[i]
print total
```

```
total = 0
for i in L:
    total += i
print total
```

like strings,  
can iterate  
over list  
elements  
directly

- notice
  - list elements are indexed 0 to  $\text{len}(L) - 1$
  - `range(n)` goes from 0 to  $n-1$

# OPERATIONS ON LISTS - ADD

- **add** elements to end of list with `L.append(element)`

- **mutates** the list!

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

```
L.append(5) → L is now [2, 1, 3, 5]
```



- what is the dot?
  - lists are Python objects, everything in Python is an object
  - objects have data
  - objects have methods and functions
  - access this information by `object_name.do_something()`
  - will learn more about these later

# OPERATIONS ON LISTS - ADD

- to combine lists together use **concatenation**, + operator, to give you a new list
- **mutate** list with `L.extend(some_list)`

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

```
L2 = [4, 5, 6]
```

```
L3 = L1 + L2
```

→ L3 is [2, 1, 3, 4, 5, 6]  
L1, L2 unchanged

```
L1.extend([0, 6])
```

→ mutated L1 to [2, 1, 3, 0, 6]



# OPERATIONS ON LISTS - REMOVE

- delete element at a **specific index** with `del (L[index])`
- remove element at **end of list** with `L.pop()`, returns the removed element
- remove a **specific element** with `L.remove(element)`
  - looks for the element and removes it
  - if element occurs multiple times, removes first occurrence
  - if element not in list, gives an error

all these operations mutate the list

```
L = [2, 1, 3, 6, 3, 7, 0] # do below in order
L.remove(2) → mutates L = [1, 3, 6, 3, 7, 0]
L.remove(3) → mutates L = [1, 6, 3, 7, 0]
del(L[1]) → mutates L = [1, 3, 7, 0]
L.pop() → returns 0 and mutates L = [1, 3, 7]
```

# CONVERT LISTS TO STRINGS AND BACK

- convert **string to list** with `list(s)`, returns a list with every character from `s` as an element in `L`
- can use `s.split()`, to **split a string on a character** parameter, splits on spaces if called without a parameter
- use `' '.join(L)` to turn a **list of characters into a string**, can give a character in quotes to add char between every element

```
s = "I<3 cs"
```

```
list(s)
```

```
s.split('<')
```

```
L = ['a', 'b', 'c']
```

```
' '.join(L)
```

```
'_'.join(L)
```

→ `s` is a string

→ returns `['I', '<', '3', ' ', 'c', 's']`

→ returns `['I', '3 cs']`

→ `L` is a list

→ returns `"abc"`

→ returns `"a_b_c"`

# OTHER LIST OPERATIONS

- `sort()` and `sorted()`
- `reverse()`

▪ and many more! <https://docs.python.org/3/tutorial/datastructures.html>

`sort()`는 list에만 적용이 되지만, `sorted()`는 다른 구조에도 적용 가능

```
>>> sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'})  
[1, 2, 3, 4, 5]
```

```
L=[9, 6, 0, 3]
```

```
newone = sorted(L) → returns sorted list, does not mutate L
```

```
L.sort() → mutates L=[0, 3, 6, 9]
```

```
L.reverse() → mutates L=[9, 6, 3, 0]
```

# BLANK 2-D ARRAY 만들기

- `m=4; n=2`
- `a = [[0] * m for i in range(n)]`      <- 뒤에 나오는 index 가 크게 변하는 것, `a[n][m]`
- `print(a)`
  
- `b = [[0] * m] * n`
- `print(b)`
  
- `c=[[0 for j in range(m)] for i in range(n)]`
- `print(c)`

```
[[0, 0, 0, 0], [0, 0, 0, 0]]  
[[0, 0, 0, 0], [0, 0, 0, 0]]  
[[0, 0, 0, 0], [0, 0, 0, 0]]
```

- `def test_func(x, l):`

- `x = x+1`

- `l[0] = 10`

- `return None`

- `x = 5`

- `l = [1, 2, 4]`

- `test_func(x,l)`

- `print(x)`

- `print(l)`

# MUTATION, ALIASING, CLONING

---



IMPORTANT  
and  
TRICKY!

*Again, Python Tutor is your best friend  
to help sort this out!*

<http://www.pythontutor.com/>

# LISTS IN MEMORY

- lists are **mutable**
- behave differently than immutable types
- is an object in memory
- variable name points to object
- any variable pointing to that object is affected
- key phrase to keep in mind when working with lists is **side effects**

# AN ANALOGY

- attributes of a person
  - singer, rich
- he is known by many names
- all nicknames point to the **same person**
  - add new attribute to **one nickname** ...

Justin Bieber singer rich troublemaker

- ... **all his nicknames** refer to old attributes AND all new ones

The Bieb singer rich troublemaker JBe

ebs singer rich troublemaker

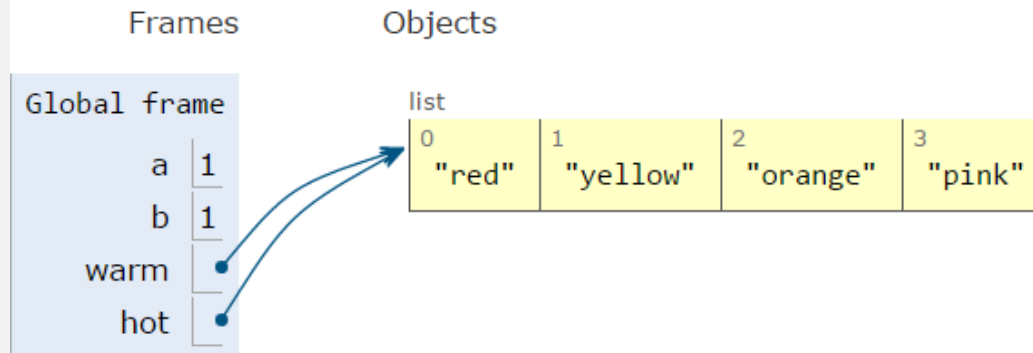


# ALIASES (SHALLOW COPY)

- `hot` is an **alias** for `warm` – changing one changes the other!
- `append()` has a side effect

```
1 a = 1
2 b = a
3 print(a)
4 print(b)
5
6 warm = ['red', 'yellow', 'orange']
7 hot = warm
8 hot.append('pink')
9 print(hot)
10 print(warm)
```

```
1
1
['red', 'yellow', 'orange', 'pink']
['red', 'yellow', 'orange', 'pink']
```



# CLONING A LIST (DEEP COPY)

- create a new list and **copy every element** using  
`chill = cool[:]`

```
1 cool = ['blue', 'green', 'grey']  
2 chill = cool[:]  
3 chill.append('black')  
4 print(chill)  
5 print(cool)
```

```
['blue', 'green', 'grey', 'black']  
['blue', 'green', 'grey']
```

Frames

Objects

Global frame

cool

chill

list

0	1	2
"blue"	"green"	"grey"

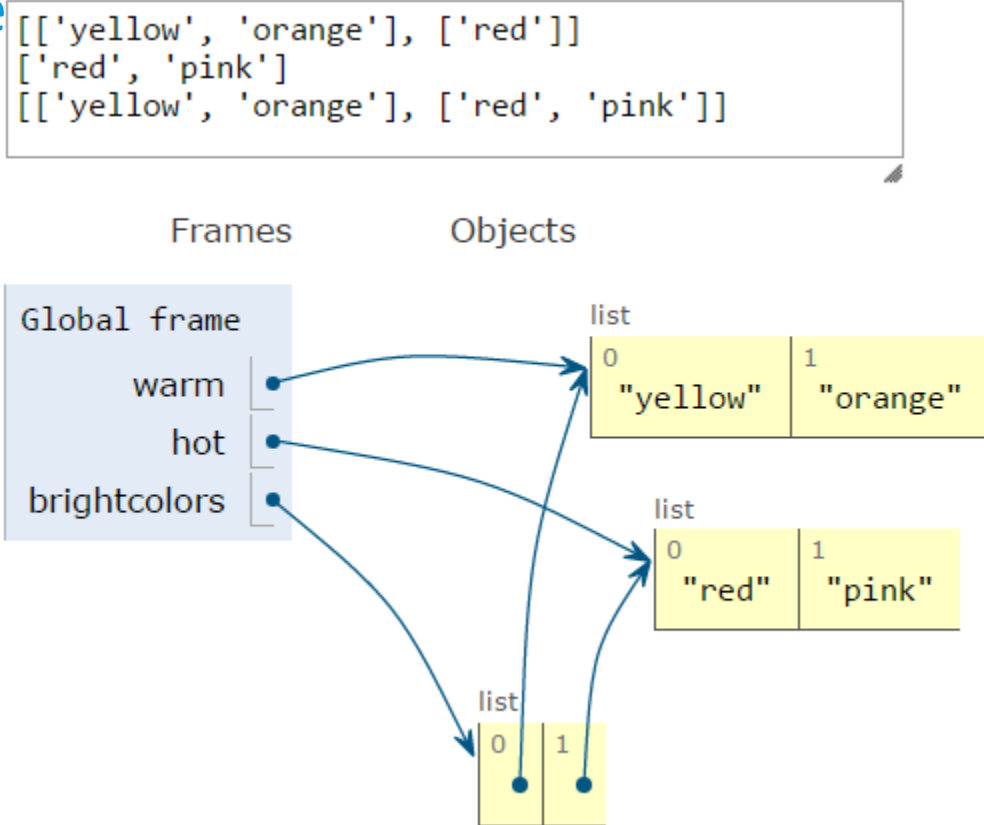
list

0	1	2	3
"blue"	"green"	"grey"	"black"

# LISTS OF LISTS OF LISTS OF....

- can have **nested** lists
- side effects still possible after mutation

```
1 warm = ['yellow', 'orange']
2 hot = ['red']
3 brightcolors = [warm]
4 brightcolors.append(hot)
5 print(brightcolors)
6 hot.append('pink')
7 print(hot)
8 print(brightcolors)
```



- `def test_func(x, l):`

- `x = x+1`

- `l[0] = 10`

- `return None`

- `x = 5`

- `l = [1, 2, 4]`

- `l1 =l[:]`

- `test_func(x,l1)`

- `print(x)`

- `print(l)`



```
import copy
a = [[1, 10, 3], [4, 5, 6]]
b = copy.deepcopy(a)
a[0][1] = 9
b[1][1] = 7
print(a)
print(b)  # b doesn't change ->
Deep Copy
```

```
▪ import copy
▪ def func1(l2):
▪     l2[0][0] = 10
▪     return None
▪ l = [[0, 1, 2], [3, 4, 5]]
▪ l2 = l[:]
▪ # func1(l2)
▪ # print(l)

▪ l3 = copy.deepcopy(l)
▪ func1(l3)
▪ print(l)
```



# LIST COPY

```
import copy
list_1 = [[-1, -2], 2.0, 3.0, 4.0, 'end']
list_2 = list_1
new_list_1 = list_1.copy()
new_list_2 = list_1[:]
new_list_3 = list(list_1)
new_list_4 = [i for i in list_1]
copy_list = copy.copy(list_1)
list_1[0][1] = 10
list_1[3] = 5
print('#1', list_2)    #1 [[-1, 10], 2.0, 3.0, 5, 'end']
print('#2', new_list_1) #2 [[-1, 10], 2.0, 3.0, 4.0, 'end']
print('#3', new_list_2) #3 [[-1, 10], 2.0, 3.0, 4.0, 'end']
print('#4', new_list_3) #4 [[-1, 10], 2.0, 3.0, 4.0, 'end']
print('#5', new_list_4) #5 [[-1, 10], 2.0, 3.0, 4.0, 'end']
```

```
copy_list[0] = 0.0
list_2[0] = '2'
new_list_1[1] = 'a'
new_list_2[2] = 'b'
new_list_3[3] = 'c'
new_list_4[4] = 'd'
print('#6', copy_list) #6 [0.0, 2.0, 3.0, 4.0, 'end']
print('#7', list_1)    #7 ['2', 2.0, 3.0, 5, 'end']
print('#8', new_list_1) #8 [[-1, 10], 'a', 3.0, 4.0, 'end']
print('#9', new_list_2) #9 [[-1, 10], 2.0, 'b', 4.0, 'end']
print('#1', new_list_3) #1 [[-1, 10], 2.0, 3.0, 'c', 'end']
print('#2', new_list_4) #2 [[-1, 10], 2.0, 3.0, 4.0, 'd']
```



# DEEP COPY – FOR NESTED LIST

```
>>> a = [1, 2, 3]
>>> b = [4, a, 5]
>>> b
[4, [1, 2, 3], 5]
>>> c = b[:]
>>> c
[4, [1, 2, 3], 5]
>>> a[0] = 100
>>> c
[4, [100, 2, 3], 5]
>>> b
[4, [100, 2, 3], 5]
```

```
>>> a = [1, 2, 3]
>>> b = [4, a, 5]
>>> c = copy.copy(b)
>>> c
[4, [1, 2, 3], 5]
>>> a[0] = 50
>>> c
[4, [50, 2, 3], 5]
```

```
>>> a = [1, 2, 3]
>>> b = [4, a, 5]
>>> import copy
>>> c = copy.deepcopy(b)
>>> c
[4, [1, 2, 3], 5]
>>> a[0] = -999
>>> c
[4, [1, 2, 3], 5]
>>> b
[4, [-999, 2, 3], 5]
```

개인적으로 고민하지 않고 공유메모리 형태로 사용하는 리스트의 데이터는 무조건 deepcopy로 복사해 버린다. 워낙 관련 문제를 많이 겪었더니...;;;

<http://seorenn.blogspot.com/2011/05/python-copy.html>



# SHALLOW COPY AND DEEP COPY OF LIST

- Shallow copy
- `a = [[1, 2, 3], [4, 5, 6]]`
- `b = (a[0])`
- `a[0][1] = 7`
- `b[2] = 9`
- `print(a[0])`
- `print(b)`
- `print(id(a[0]))`
- `print(id(b))`
- -----
- `[1, 7, 9]`
- `[1, 7, 9]`
- 3796456
- 3796456

Shallow copy: 내용이 아니라 포인터만 카피한다.  
그냥 변수 이름만 공유, 하나가 바뀌면 다른 것도 바뀐다.

Deep copy: 새로운 변수를 만든다. `copy` function 사용.

List가 아닌 그냥 변수는 '=' 는 deep copy이다. 단지 복잡한 구조체에서만 pointer

List 전체를 카피

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

```
b = a[:]
```

```
print(a is b) #False
```

```
b[0]=5
```

```
print(a[0]) #[1, 2, 3]
```

```
print(a[0][1])#2
```

```
print(b) #[5, [4, 5, 6]]
```

```
---
```

```
b = a[0][:]
```

```
b[0]=5
```

```
print(b) #[5, 2, 3]
```

```
----
```

Python 3.6

```
1 a = [[1, 2, 0], [4, 5, 6]]
2 b = a[0]
3 #b=(a[0]) 동일
4 a[0][1] = 7
5 b[2] = 0
6 print(a[0])
7 print(b)
8 print(id(a[0]))
9 print(id(b))
```

[Edit this code](#)

scud  
site



<< First < Prev Next > Last >>

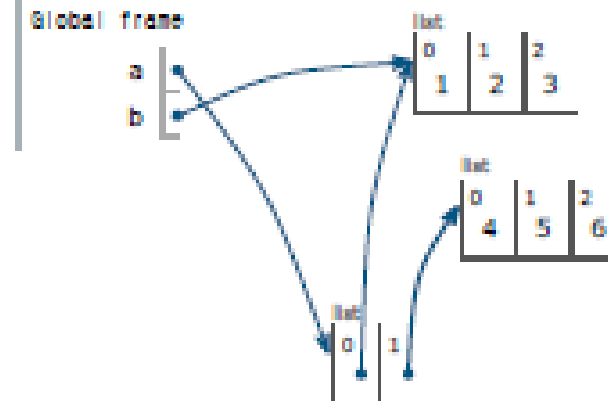
Step 3 of 8

1 (NEW)

Print output (drag lower right corner to resize)

Frames

Objects



```
a = [[1, 2, 3], [4, 5, 6]]
b = a[0]
#b=(a[0]) 동일
a[0][1] = 7
b[2] = 9
print(a[0])
print(b)
print(id(a[0]))
print(id(b))
```

```
[1, 7, 9]
[1, 7, 9]
1866736407368
1866736407368
```

```
a = [[1, 2, 3], [4, 5, 6]]
b = a[:] #b = a.copy() 도 동일한 효과
print(a is b) #False
b[0]=[7, 8, 9]
```

```
print(a)
print(b)

c=a[:]
c[0][0]=10
print(c)
print(a)
```

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

```
a = [[0,1], [2,3]]
```

```
b=a
for i in range(2):
    for j in range(2):
        c[i][j] = a[i][j]
```

```
b[0][0] = 5
c[1][1] = 6
print(a)
print(b)
print(c)
```

```
-----
[[5, 1], [2, 3]]
[[5, 1], [2, 3]]
[[0, 1], [2, 6]]
```

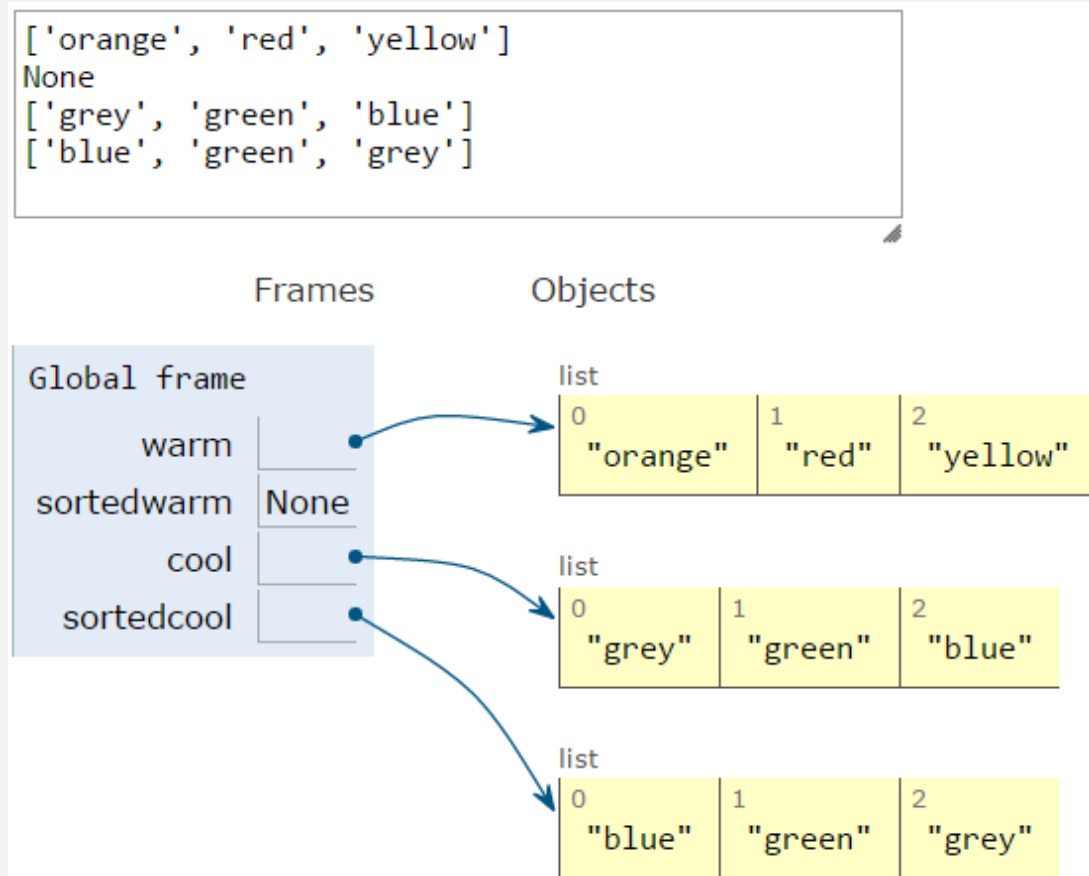
```
oldlist = [[1, 10, 3], [4, 5, 6]]  
newlist = [[i for i in row] for row in oldlist]  
newlist[1][0] = 7  
oldlist[0][0] = 9  
print(oldlist)  
print(newlist)
```

```
-----  
[[9, 10, 3], [4, 5, 6]]  
[[1, 10, 3], [7, 5, 6]]
```

# SORTING LISTS

- calling `sort()` **mutates** the list, returns nothing
- calling `sorted()` **does not mutate** list, must assign result to a variable

```
1 warm = ['red', 'yellow', 'orange']
2 sortedwarm = warm.sort()
3 print(warm)
4 print(sortedwarm)
5
6 cool = ['grey', 'green', 'blue']
7 sortedcool = sorted(cool)
8 print(cool)
9 print(sortedcool)
```



# METHOD와 FUNCTION을 이용한 COPY, SORT

- Method (class와 엮어서 동작하는 함수)를 이용한 것. List나 class 자체의 값이 변한다.
- `a.copy()`
- `a.sort()`
- `a.sort(reverse=True)`
- `print(sorted([4, 2, 3, 5, 1]))`  
#->[1, 2, 3, 4, 5]



# SORT의 KEY옵션

sort의 key 옵션, key 옵션에 지정된 함수의 결과에 따라 정렬, 아래는  
원소의 길이

```
>>> m = '나는 파이썬을 잘하고 싶다'
>>> m = m.split()
>>> m
['나는', '파이썬을', '잘하고', '싶다']
>>> m.sort(key=len)
>>> m
['나는', '싶다', '잘하고', '파이썬을']
```

```
sorted("This is a test string from Andrew".split(), key=str.lower)
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```

```
>>> sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'})
[1, 2, 3, 4, 5]
```

sort(\*, key=None, reverse=False)


This method sorts the list in place, using only < comparisons between items. Exceptions are not suppressed - if any comparison operations fail, the entire sort operation will fail (and the list will likely be left in a partially modified state).

```
>>> a = [1, 10, 5, 7, 6]
>>> a.sort()
>>> a
[1, 5, 6, 7, 10]
>>> a = [1, 10, 5, 7, 6]
>>> a.sort(reverse=True)
>>> a
[10, 7, 6, 5, 1]
```

# MUTATION AND ITERATION


## TRY THIS IN PYTHON TUTORIAL

- **avoid** mutating a list as you are iterating over it



```
def remove_dups(L1, L2):  
    for e in L1:  
        if e in L2:  
            L1.remove(e)
```

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







```
def remove_dups(L1, L2):  
    L1_copy = L1[:]  
    for e in L1_copy:  
        if e in L2:  
            L1.remove(e)
```

clone list first, note  
that `L1_copy = L1`  
does NOT clone

- L1 is [2, 3, 4] not [3, 4] Why?
  - Python uses an internal counter to keep track of index it is in the loop
  - mutating changes the list length but Python doesn't update the counter
  - loop never sees element 2







$i = i + 1$  reassigns  $i$  ,  $i += 1$  increments  $i$  by 1

```
 x = [1, 2]  
another_x = x  
 y = [3]  
  
 x += y  
  
 print(x)  
print(another_x)
```

**Output:**

```
[1, 2, 3]  
[1, 2, 3]
```

```
 x = [1, 2]  
another_x = x  
 y = [3]  
  
 x = x + y  
  
 print(x)  
print(another_x)
```

**Output:**

```
[1, 2, 3]  
[1, 2]
```

# List +[] 와 append 의 차이

```
sample_list = []
n = 10

for i in range(n):

    # i refers to new element
    sample_list = sample_list+[i]
    print(id(sample_list))

print(sample_list)
```

2925898144904  
2925898851464  
2925898959176  
2925898144904  
2925898851464  
2925898959176  
2925898144904  
2925898851464  
2925898959176  
2925898144904  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

## $O(N^2)$ algorithm

```
▶ sample_list = []  
n = 10  
  
for i in range(n):  
    # i refers to new element  
    sample_list.append(i)  
    print(id(sample_list))  
print(sample_list)
```

[illegible]

# 함수관련 MUTABLE, IMMUTABLE

- Mutable – 입력 parameter로 들어간 변수의 원본값을 함수내에서 바꿀 수 있는 것. List, dictionary, numpy (안 바꾸려면 함수내에서 deep copy후 사용)
- Immutable – 원본값이 바뀌지 않는 것. 숫자, 문자, tuple

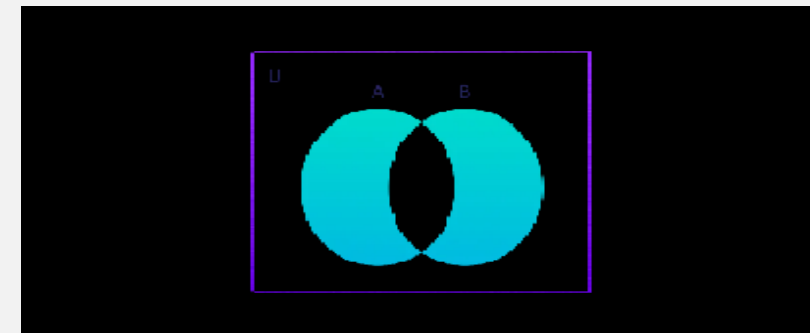
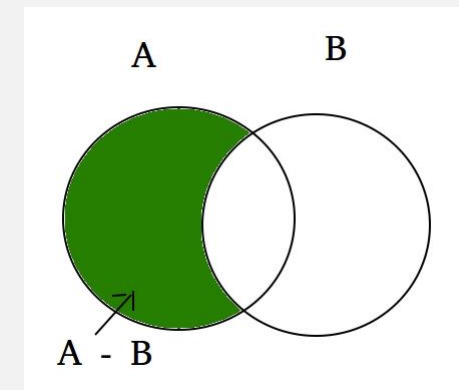
```
def mutable_immutable_check(xin, input_list):  
    xin += 1  
    input_list.append(20)  
  
x=1  
test_list = [1, 2, 3]  
mutable_immutable_check(x, test_list)  
print('x=', x, 'test_list =', test_list)
```

```
x= 1 test_list = [1, 2, 3, 20]
```

- 2~100 사이의 숫자 중에서 소수(prime number)를 찾기. 어떤 수  $n$ 이 소수인지를 알려면  $n/2$  이하의 소수로 나누어지지 않아야 한다. 즉, 13이 소수인지를 알려면, 6이하의 소수 즉, 2, 3, 5로 나누어 지지 않으면 된다. 18의 경우는 9이하의 소수, 2, 3, 5로 나누어 지지 않나를 보면 된다.

# SET : {}

- 순서가 없고 중복을 허용하지 않는 array
- 선언 :  $s = \{\text{원소}\}$  혹은  $s = \text{set}(\text{array자료형})$
- \*string을 넣으면 각 문자가 집합의 원소가 된다.
- 교집합 :  $s1 \ \& \ s2$  혹은  $s1.\text{intersection}(s2)$
- 합집합 :  $s1 \ | \ s2$  혹은  $s1.\text{union}(s2)$
- 차집합 :  $s1 - s2$  혹은  $s1.\text{difference}(s2)$
- symmetric diff :  $s1 \ ^ \ s2$  혹은  $s1.\text{symmetric\_difference}(s2)$
- 값 추가 :  $s1.\text{add}(\text{값})$  또는  $s1.\text{update}(\text{array자료형})$
- 값 제거 :  $s1.\text{remove}(\text{값})$
- 두 집합에 대해 대소 비교를 할 경우 부분집합인지에 대해 판단함



- # set 정의
- myset = { 1, 1, 3, 5, 5 }
- print(myset)
- # 출력: {1, 3, 5}
- 
- # 리스트를 set으로 변환
- mylist = ["A", "A", "B", "B", "B"]
- s = set(mylist)
- print(s)      # 출력: {'A', 'B'}
- -----

- myset = {1, 3, 5}
- 
- # 하나만 추가
- myset.add(7)
- print(myset)
- 
- # 여러 개 추가
- myset.update({4,2,10})
- print(myset)
- 
- # 하나만 삭제
- myset.remove(1)
- print(myset)
- 
- # 모두 삭제
- myset.clear()
- print(myset)

# DICTIONARY : {}

- 각 원소가 key : value의 값으로 대응되어 있는 자료구조 (**hash**)
- `d = {'key' : 'value', 'name' : 'John', 'number' : 3, 'grade' : ['A', 'B-', 'A+']}` 와 같이 선언한다. `d = dict()`로도 가능.
- dictionary의 key는 숫자, string등이 가능하며, value는 거의 모든 자료형이 올 수 있다.
- 각 key는 유일하다(같은 key를 입력하면 나중의 것만 남는듯)

# DICTIONARY : {} 사용하기

- key를 이용해 value반환 : `dict['key']` 혹은 `dict.get('key')`
- \*전자의 경우 없는 key를 쓰면 error, 후자는 None을 반환한다.
- key값들을 반환(list) : `dict.keys()`
- value값들을 반환(list) : `dict.values()`
- (key,value)반환(list) : `dict.items()` -> `[('key','value'),...]`
- 모두 삭제 : `dict.clear()`
- 값으로 key 찾기 : 값 in dict:
- \*for문을 `for key in dict:` 식으로 활용하면 dict의 모든 key에 대한 for문을 만들 수 있다.



#1

```
scores = {"철수": 90, "민수": 85, "영희": 80}
```

```
v = scores["민수"] # 특정 요소 읽기
```

```
scores["민수"] = 88 # 쓰기
```

# 2. Tuple List로부터 dict 생성

```
persons = [('김기수', 30), ('홍대길', 35),  
('강찬수', 25)]
```

```
mydict = dict(persons)
```

```
age = mydict["홍대길"]
```

```
print(age) # 35
```

# 3. Key=Value 파라미터로부터 dict 생성

```
scores = dict(a=80, b=90, c=85)
```

```
print(scores['b']) #90
```

```
scores = {"철수": 90, "민수": 85, "영희": 80}
```

```
scores["민수"] = 88 # 수정
```

```
scores["길동"] = 95 # 추가
```

```
del scores["영희"]
```

```
print(scores)
```

```
# 출력 {'철수': 90, '길동': 95, '민수': 88}
```

#순서는 random 하게 나옴

```
for key in scores:
```

```
    val = scores[key]
```

```
    print("%s : %d" % (key, val))
```

# keys

```
keys = scores.keys()
```

```
for k in keys:
```

```
    print(k)
```

# values

```
values = scores.values()
```

```
for v in values:
```

```
    print(v)
```

```
items = scores.items()
```

```
print(items)
```

```
# 출력: dict_items([('민수', 85), ('영희', 80), ('철수', 90)])
```

```
# dict_items를 리스트로 변환할 때  
itemsList = list(items)
```

```
-----  
scores = {"철수": 90, "민수": 85, "영희": 80}  
v = scores.get("민수") # 85  
v = scores.get("길동") # None  
v = scores["길동"]    # 에러 발생
```

```
# 멤버쉽연산자 in 사용  
if "길동" in scores:  
    print(scores["길동"])
```

```
scores.clear() # 모두 삭제  
print(scores)
```

<http://pythonstudy.xyz/python/article/14-%EC%BB%AC%EB%A0%89%EC%85%98--Dictionary>

- List Comprehension (LC)
- List comprehension은 리스트를 쉽게 생성하기 위한 방법이다. 파이썬에서 보편적으로 사용되는 기능으로 다양한 조건으로 리스트를 생성할 수 있는 강력한 기능중 하나이다.
- # 20까지의 짝수를 출력하기 위해 다음과 같은 LC를 사용할 수 있다
- `evens = [x * 2 for x in range(11)]`
- # `[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]`
- # 리스트의 모든 원소값을 정규화 시킨 후 상수값을 더하는 LC
- `vals = [32, 12, 96, 42, 32, 93, 31, 23, 65, 43, 76]`
- `amount = sum(vals)`
- `norm_and_move = [(x / amount) + 1 for x in vals]`
- # `[1.0587155963302752, 1.0220183486238532, 1.1761467889908257, ....]`

- # 100 이하의 제곱수가 아닌 수를 찾는 LC
- `from math import sqrt`
- `non_squars = [x for x in range(101)  
if sqrt(x)**2 != x]`
- # [2, 3, 5, 6, 7, 8, 10, 12, 13, 15,  
18, 19, 20, 23, 24, 26, 28, 29,

- #  $a^2 + b^2 = c^2$  ( $a < b < c$ )를 만족하는 피타고라스 방정식의 해를 찾는 LC `solutions = [(x, y, z) for x in range(1, 30) for y in range(x, 30) for z in range(y, 30) if x**2 + y**2 == z**2]`  
# [(3, 4, 5), (5, 12, 13), (6, 8, 10), (7, 24, 25), (8, 15, 17), (9, 12, 15), (10, 24, 26), (12, 16, 20), (15, 20, 25), (20, 21, 29)] # 단어에서 모음을 제거하는 LC `word = 'mathematics'`  
`without_vowels = ''.join([c for c in word if c not in ['a', 'e', 'i', 'o', 'u']])` # 'mthmtcs' # 행렬을 일차원화 시키는 LC `matrix = [ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], ]` `flatten = [e for r in matrix for e in r]` # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

- Set comprehension은 LC와 정확히 동일하며 단지 list가 아닌 set을 생성한다는 것만 다르다.

- # 다음의 LC는 중복된 값들을 포함한다

- `no_primes = [j for i in range(2, 9) for j in range(i * 2, 50, i)]`

- # [4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, ..

- # SC를 사용하면 중복값이 없는 집합을 얻을 수 있다

- `no_primes = {j for i in range(2, 9) for j in range(i * 2, 50, i)}`

- # {4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25, 26, 27, 28, ...

- # 두 리스트를 하나의 dict로 합치는 DC. 하나는 key, 또 다른 하나는 value로 사용한다
- subjects = ['math', 'history', 'english', 'computer engineering']
- scores = [90, 80, 95, 100]
- score\_dict = {key: value for key, value in zip(subjects, scores)}
- # {'math': 90, 'history': 80, 'english': 95, 'computer engineering': 100}
- # 튜플 리스트를 dict 형태로 변환하는
- DC score\_tuples = [('math', 90), ('history', 80), ('english', 95), ('computer engineering', 100)] score\_dict = {t[0]: t[1] for t in score\_tuples}
- # {'math': 90, 'history': 80, 'english': 95, 'computer engineering': 100}

- `#zip: merge two data structures into one`
- `a=['jack', 'peter']`
- `b=[10, 40]`
- `d=list(zip(a, b))`
- `tel=dict(zip(a,b))`
- `print(tel)`
- `print(d)`
  - `{'jack': 10, 'peter': 40}`
  - `[('jack', 10), ('peter', 40)]`

- `lst=[1, 5, 3, 2]`

- `lst.sort()`

- `sorted(lst)`

- List 나 Tuple에 순서와 숫자 부여