Collection Data Type

1. Sequence

mutable

- list
- (deque)

immutable

- tuple
- str
- range

2. Hash

mutable

- dict
- defaultdict
- set

immutable

- frozenset

✓ mutable : 수정 가능
✓ immutable : 수정 불가능, 필요시 전체 재할당

Comparisons

Sequence

list,tuple,str,deque: ==, <, >, !=, <=, >= range: ==, !=

1. 첫번째 원소부터 순서대로 같은 위치끼리 비교한다.
2. 값이 다른 위치가 나온다면 해당 값의 비교 결과가 객체의 비교결과이다.
3. 값이 전부 같고 길이가 다르다면 짧은 쪽이 작은값(<)이다.
4. 값이 전부 같고 길이도 같다면 두 객체는 같은 값(==)이다.

Hash

set: ==, <, >, !=, <=, >= dict : ==, !=

- 순서 상관 없이 모든 요소가 같으면 같은값(==)
  dict은 (key, value) pair로 판별
- <, >연산은 요소의 포함 여부에 따라 결정

list

```
[1,2,3] == [1,2,3]
[1,2,4] != [1,2,3]
[1,2,3] != [3,2,1]
[1,2,3] < [1,2,4]
[1,2] < [1,2,3]
```

str

```
'abc' == 'abc'
'abc' != 'cba'
'abc' < 'abd'
'ab' < 'abc'
```

set

```
{1,2,3} == {1,2,3}
{1,2,3} == {3,2,1}
{1,2,3} != {1,2,4}
{1,2} < {3,2,1}
```

dict

```
{'a':0, 'b':1} == {'b':1, 'a':0}
{'a':0, 'b':1} != {'b':0, 'a':0}
```

Looping techniques

기본 iterable 탐색

```
for i in range(3)
for s in 'abc'
for s in input().split()
for x in map(int,input().split())
```

두개 이상 iterable 병렬적으로 묶어서 탐색

```
for t in zip(range(3), 'abc')          t = (0,'a') ..
for a,b in zip(range(3), 'abc')        a = 0 ..   b = 'a' ..
```

iterable에 번호를 같이 묶어서 탐색

```
for t in enumerate('abc')              t  = (0,'a') ..
for i, x in enumerate('abc', 1)        i = 1 ..  b = 'a' ..
```

sequence 거꾸로 탐색

```
for x in reversed([1,2,3])             1, 2, 3
```

특정 조건 값 삭제하면서 탐색

* 홀수 삭제하기 예시

1. 임시객체

```
tmp = []
for x in li:
    if x % 2: tmp.append(x)
li = tmp
```

2. while

```
i = 0
while i < len(li):
    if li[i] % 2 == 0: li.pop(i)
    else: i += 1
print(li)
```

3. filter, list comprehension

```
li = list(filter(lambda x: x%2, li))
li = [x for x in li if x%2]
```

Iterable unpacking

우변에 iterable, 좌변에 그 개수에 맞춰 변수를 나열하면 unpacking을 할 수 있다.
좌변에 asterisk(*)을 사용하면 남은 원소를 list로 다시 packing 할 수 있다.
중첩 unpakcing도 가능하다.

list

```
li = [1,2,3]
a,b,c = li          # 1, 2, 3
a,_,_ = li          # 1
a, *b = li          # 1, [2, 3]
*a, b = li          # [1, 2], 3
a, b, *c = li       # 1, 2, [3]
a, b, c, *d = li    # 1, 2, 3, []
```

dict

```
d = {'A': 1, 'B': 2, 'C': 3}
a,b,c = d               # A, B, C
a,b,c = d.keys()    # A, B, C
a,b,c = d.items()   # (A,1), (B,2), (C,3)
a,b,c = d.values()  # 1, 2, 3
```

??

```
l1 = [1,2,3]
l2 = ['str']
l3 = *l1, *l2       # (1, 2, 3, 'str')
```

위 표현은 사실상 다음과 같다.
l3 = tuple(*l1, *l2)

=> 함수의 인자로서의 unpacking은 *를 붙인다.

tuple

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

str

```
s = 'ABC’
a,b,c = s           # A, B, C
```

nested

```
li = [(1,2),'ab',3]
(a,b),c,d = li          # 1, 2, 'ab', 3
a,(b,c),*d = li         # (1,2), a, b, [3]
```

Iterable unpacking

practice1

```
l1 = [1,2,3]
l2 = ['str']
l3 = l1 + [*l2[0]]
a, *b, (c, *d) = l3
```

l3, a, b, c, d = ?

practice2

```
l1 = [1,2,3]
l2 = ['str']
l3 = [*l1, *l2]
a, *b, (c, *d) = l3
```

l3, a, b, c, d = ?

Iterable unpacking

practice1

```
l1 = [1,2,3]
l2 = ['str']
l3 = l1 + [*l2[0]]
a, *b, (c, *d) = l3
```

```
l3 = [1, 2, 3, 's', 't', 'r']
a = 1
b = [2, 3, 's', 't']
c = 'r'
d = []
```

practice2

```
l1 = [1,2,3]
l2 = ['str']
l3 = [*l1, *l2]
a, *b, (c, *d) = l3
```

```
l3 = [1, 2, 3, 'str']
a = 1
b = [2, 3]
c = 's'
d = ['t', 'r']
```

Function argument

argument 함수 호출시에 전달되는 값

- positional 값만 보내진다.
  - value
  - *iterable (unpacking)
- keyword paremeter의 이름이 명시되어 보내진다.
  - name=value
  - **dict (unpacking)

parameter 전달된 인자를 받아들이는 변수

특 징
- 항상 positional 뒤에 keyword가 온다.
- keyward끼리의 순서는 중요하지 않다.

parameter

```
def func(a, b, c)
```

argument

```
func(3, 4, 5)
```

positional, keyword

```
def func(a, b, c):
    print(a,b,c)

func(3, 4, 5)
func(3, b=4, c=5)
func(3, c=5, b=4)
func(*[3,4,5])
func(**dict(c=5,b=4,a=3))
func(*[3], **dict(c=5,b=4))
```

a=3, b=4, c=5

Function argument

default argument values

- 인자의 기본값을 설정하여 호출시 생략 할 수 있다.
- important warning
  - default value는 최초 1회 생성되어 항상 해당 객체를 참조한다.
  - mutable의 경우 의도와 다른 값이 설정될 수 있다.

arbitrary argument lists 가변인자

- 가변인자는 다음과 같이 받아온다. 변수명은 달라도 된다.
  - positional: *args (type=tuple)
  - keyword: **kwargs (type=dict)

- 순서 positional => *args => keyword => **kwargs

default

```
def func(a, b=0, c=1):
    print(a,b,c)

func(3)          # a=3, b=0, c=1
func(3, 4)       # a=3, b=4, c=1
func(3, c=5)     # a=3, b=0, c=5
```

```
def func(a=[]):
    a.append([0])
    print(a)

func()           # a=[0]
func()           # a=[0,0]
func()           # a=[0,0,0]
```

arbitrary

```
def func(a, *args, b=3, **kwargs):
    print(a, args, b, kwargs)

func(0,1,2,d=3,c=4,b=5)
a=0, args=(1, 2), b=5, kwargs={d:3, c:4}
func(0)
a=0, args=(), b=3, kwargs={}
```

Variable scope

LEGB Rule 변수가 값을 찾는 순서 규칙

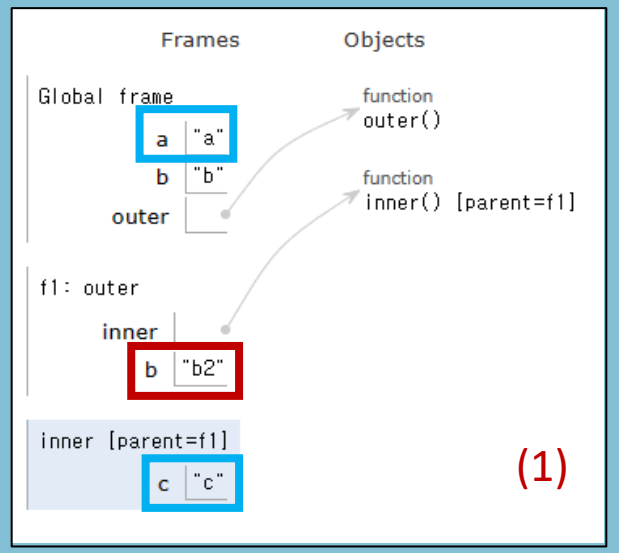
Local -> Enclosed -> Global -> Built-in

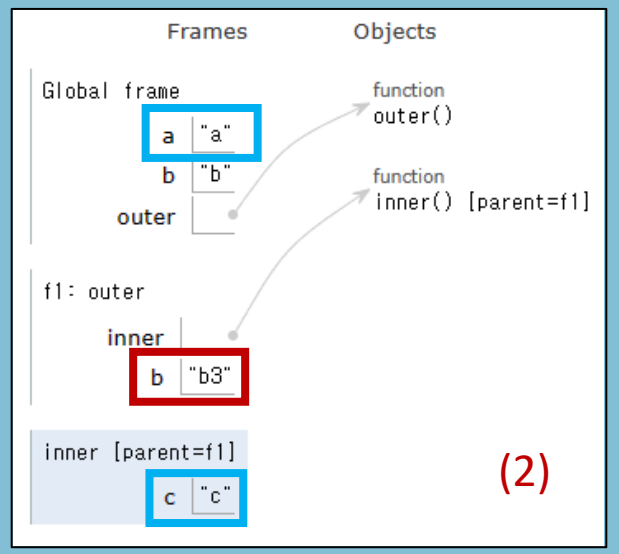
- local: 가장 가까운 함수
- enclosed: 가장 가까운 함수 이후로 가까운 함수 순
- global: 전역 변수
- built-in: 내장 함수, 속성

Function variable 특징

- 값을 할당하면 무조건 로컬 변수가 생성된다.
  - a += 3 : a=a+3
  - a = 3
- 변수가 반복, 조건문 안에서 생성되더라도 동일 함수 내에서는 전부 같은 scope로 취급된다.

```
a, b = 'a', 'b'       # global
def outer():
    b = 'b2'          # outer() local, inner() enclosed
    def inner():
        c = 'c'       # inner() local
        print(a,b,c)
    inner()           # (1)
    b = 'b3'          # outer() local, inner() enclosed
    inner()           # (2)
outer()
```





Variable scope

global

- 함수 안에서 변수 앞에 global 키워드를 붙여주면 전역 변수를 가리킨다.
- 만약, 전역 변수로 선언되지 않았으면 사용시 전역변수가 생성된다.

```
a = 3
def func():
    a = 5
func()
print(a)  # 3
```

지역변수

```
a = 3
def func():
    global a
    a = 5
func()
print(a)   # 5
```

전역변수

```
def func():
    global a
    a = 5
func()
print(a)  # 5
```

전역변수 생성

nonlocal

- 중첩 함수 안에서 변수 앞에 nonlocal 키워드를 붙여주면 enclosed 영역의 변수를 가리키게 된다.
- 만약, 해당 영역에 변수로 선언되지 않았으면 에러가 발생된다.

```
def outer():
    a = 1
    def inner():
        a = 2
        print(a)   # 2
    inner()
    print(a)       # 1
outer()
```

iner()의 로컬변수 a 생성

```
def outer():
    a = 1
    def inner():
        nonlocal a
        a = 2
        print(a)   # 2
    inner()
    print(a)       # 2
outer()
```

outer()의 로컬변수 a 변경

Variable scope

practice

```
a, b, c = 'a', 'b', 'c'
def outer():
    a, b, c = 0, 1, 2
    def inner():
        global a
        nonlocal b
        a,b,c = 'A','B','C'
        print(a,b,c)
    inner()
    print(a,b,c)
outer()
print(a,b,c)
```

outer(), inner()의 a,b,c scope은?
각 print의 결과는?

Variable scope

practice

```
a, b, c = 'a', 'b', 'c'
def outer():
    a, b, c = 0, 1, 2
    def inner():
        global a
        nonlocal b
        a,b,c = 'A','B','C'
        print(a,b,c)
    inner()
    print(a,b,c)
outer()
print(a,b,c)
```

```
global

outer() local, inner() enclosed

global
enclosed
global, enclosed, local
A B C

0 B 2

A b c
```