PYTHON COLLECTIONS 모듈 이해하기

Moon Yong Joon

1. COLLECTIONS

Moon Yong Joon

Collection 구성

collections 제공 요소

tuple, dict 에 대한 확장 데이터 구조를 제공

주요 요소	설명	추가된 버전
namedtuple()	Tuple 타입의 subclass를 만들어 주는 function	*New in version 2.6.*
OrderedDict	dict subclass that remembers the order entries were added	*New in version 2.7.*
Counter	dict subclass for counting hashable objects	*New in version 2.7.*
defaultdict	dict subclass that calls a factory function to supply missing values	*New in version 2.5.*
deque	list-like container with fast appends and pops on either end	*New in version 2.4.*

NAMED TUPLE

Moon Yong Joon

tuple

Tuple 기본 처리

tuple타입에 immutable 타입으로 내부 원소에 대해 갱신이 불가능하여 리스트처리보다 제한적

Slicing은 String 처럼 처리가능

Python Expression	Results	Description
T =(1,)	(1,)	튜플의 원소가 하나인 경우 생성 꼭 한 개일 경우는 뒤에 꼼마(,)를 붙여야 함
T = (1,2,3,4)	(1, 2, 3, 4)	튜플 생성
len((1, 2, 3))	3	Length 함수로 길이 확인
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	튜플을 합치기 Concatenation
('Hi!') * 4	'Hi!Hi!Hi!Hi!'	튜플의 반복을 string으로 표시
3 in (1, 2, 3)	True	튜플 내의 원소들이 Membership
for x in (1, 2, 3): print x,	1 2 3	튜플의 원소들을 반복자 활용 – Iteration

Tuple 메소드

t = 1,2,3,4

Method	example	Description
count(obj)	t.count(3) 1	튜플내의 원소의 갯수
Index(obj)	t.index(2) 1	튜플내의 원소의 위치

namedtuple

tuple vs. namedtuple

Tuple은 index를 기준으로 접근하지만 nametuple은 키를 가지고 접근할 수 있는 것도 추가적으로 지원

내장 타입

tuple

확장 타입

namedtuple

namedtuple 선언

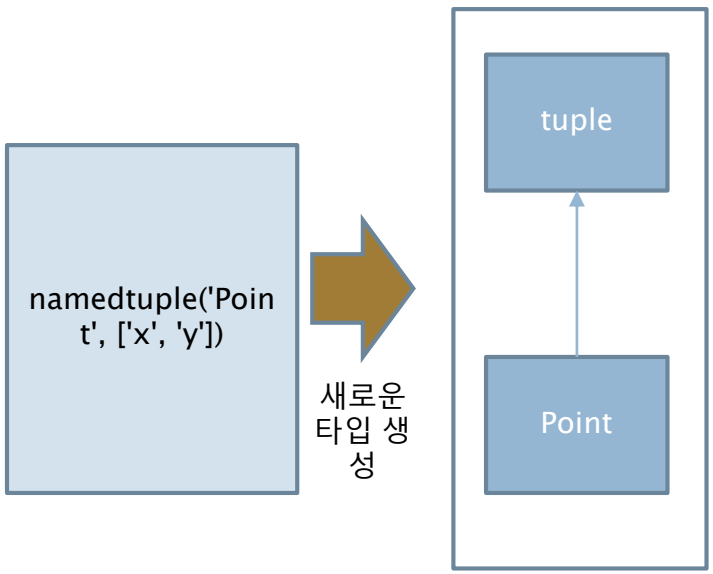
Tuple을 보다 명시적으로 사용하기 위해 index보다 name으로 접근하기 위해 만든 별도의 function class

```
import collections

help(collections.namedtuple)
```

```
Help on function namedtuple in module collections:

namedtuple(typename, field_names, verbose=False, rename=False)
    Returns a new subclass of tuple with named fields.
```

Namedtuple로 타입 만들기

Nametuple은 tuple의 subclass를 만드는 factory기능을 처리




```
import collections

Point = collections.namedtuple('Point', ['x', 'y'])

print("subclass check",issubclass(Point, tuple))
print(Point.__doc__ )

p = Point(11, y=22)

# instantiate with positional args or keywords
print(p[0] + p[1])
print(p.x + p.y)
```

```
subclass check True
Point(x, y)
33
33
```

keyword로 필드명 정의

필드명을 키워드로 정의시 에러가 발생함

```
import collections

with_class = collections.namedtuple('Person', 'name class age gender')
print(with_class._fields)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call l
<ipython-input-43-ae4ff29fdf48> in <module>()
      1 import collections
      2
----> 3 with_class = collections.namedtuple('Person', 'name class age g
      4 print(with_class._fields)

C:\Program Files\Anaconda3\lib\collections\__init__.py in namedtuple(ty
    401             if _iskeyword(name):
    402                 raise ValueError('Type names and field names cannot
--> 403                                  'keyword: %r' % name)
    404         seen = set()
    405         for name in field_names:

ValueError: Type names and field names cannot be a keyword: 'class'
```

필드명만 바꾸기

Nametuple은 class를 정의시 rename=True로 정의하면 필드명이 중복이거나 명명이 불가한 경우 이름을 바꿈

```
import collections

with_class = collections.namedtuple('Person', 'name class age gender', rename=True)
print(with_class._fields)
a = with_class("dahl","a",10,"f")
print(a._1)

two_ages = collections.namedtuple('Person', 'name age gender age', rename=True)
print(two_ages._fields)
```

```
('name', '_1', 'age', 'gender')
a
('name', 'age', 'gender', '_3')
```

Namedtuple 메소드 1

Method	Description
'_asdict',	Namedtuple에서 생성된 타입의 인스턴스를 OrderedDict로 전환
'_fields',	Namedtuple에서 생성된 타입내의 named 변수를 검색
'_make',	Namedtuple에서 생성된 타입을 가지고 새로운 인스턴스를 생성

```
import collections

Person = collections.namedtuple('Person', 'name age gender')
print('Type of Person:', type(Person))

bob = Person(name='Bob', age=30, gender='male')
print('\nRepresentation:', bob)

print(bob._fields)
# ordereddict 타입으로 변환
print(bob._asdict())
#다른 인스턴스를 생성
jane = bob._make(['Jane', 29, 'female'])
print('\nField by name:', jane.name)
```

```
Type of Person: <class 'type'>

Representation: Person(name='Bob', age=30, gender='male')
('name', 'age', 'gender')
OrderedDict([('name', 'Bob'), ('age', 30), ('gender', 'male')])

Field by name: Jane
```

Namedtuple 메소드 2

Method	Description
'_replace',	Namedtuple에서 생성된 타입에 대한 인스턴스 내의 값을 변경
'count',	내부 값에 대한 갯수
'index',	내부 값에 대한 위치

```
import collections

Person = collections.namedtuple('Person', 'name age gender')
print('Type of Person:', type(Person))

bob = Person(name='Bob', age=30, gender='male')
print('\nRepresentation:', bob)

# 변경된 튜플은 새로운 객체로 만듦
bob = bob._replace(age=50)
print(bob)
print(bob.index(50))
print(bob.count(50))
```

```
Type of Person: <class 'type'>

Representation: Person(name='Bob', age=30, gender='male')
Person(name='Bob', age=50, gender='male')
1
1
```

ORDEREDDICT

Moon Yong Joon

OrderedDict 구조

OrderedDict

OrderedDict 은 dict의 subclass로써 새로운 인스턴스를 만드는 클래스

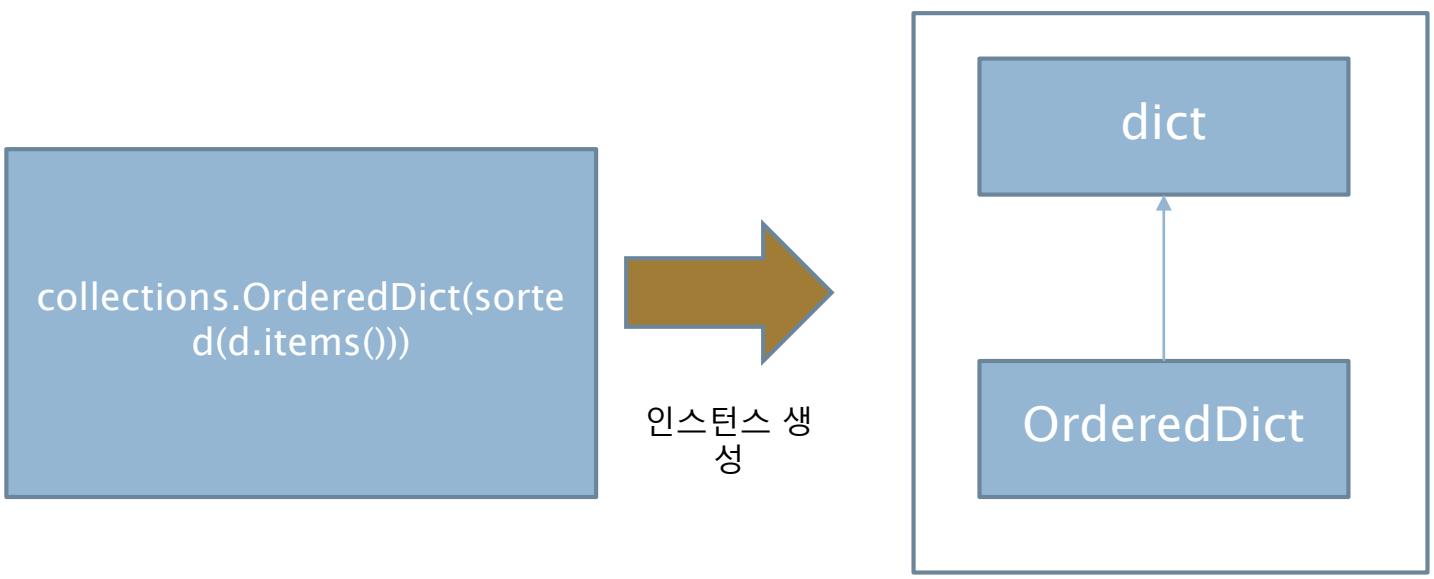
```
import collections

help(collections.OrderedDict)
```

```
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
```

OrderedDict 타입 만들기

OrderedDict 은 dict의 subclass로써 새로운 인스턴스를 만드는 클래스

issubclass(collections.OrderedDict,dict)
➔ True



dict/OrderedDict 차이

dict vs. Ordereddict 차이

collections.OrderedDict은 순서를 유지하기 위해 linked list로 내부에 구성되어 각 순서를 유지함

```
import collections

d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

# dictionary sorted by key
for t in d.items() :
    print(t[0])
print(sorted(d.items(), key=lambda t: t[0]))

do = collections.OrderedDict(sorted(d.items(), key=lambda t: t[0]))
print(do)
```

```
apple
banana
orange
pear
[('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)]
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])
```

메소드

dict 메소드(1)

d= {"k":1,"v":2}

Method	example	Description
dict.clear()	d= {"k":1,"v":2} d.clear() d {}	dict 객체 내의 요소들 클리어
dict.copy()	d1 = d.copy() d1 {'k': 1, 'v': 2}	dict객체를 다른 곳에 deep카피
dict.fromkeys()	d2 =d.fromkeys(d) d2 {'k': None, 'v': None}	dict 객체의 키를 새로운 dict 객체를 생성하는 키로 처리
dict.get(key, default=None)	d.get('k') 1	dict내의 키를 가지고 값을 가져옴
dict.has_key(key)	d.has_key('k') True	dict내의 키 존재 여부
dict.items()	d.items() [('k', 1), ('v', 2)]	dict객체의 키와 값을 순서쌍으로 나타내어 리스트로 전달

dict 메소드(2)

d= {"k":1,"v":2}

Method	example	Description
dict.keys()	:d.keys() ['k', 'v']	dict 내의 키를 리스트로 전달
dict.setdefault(key, default=None)	d.setdefault('s',3) d {'k': 1, 's': 3, 'v': 2}	dict 내의 키와 값을 추가
dict.update(dict2)	d.update({1:1}) Id {1: 1, 'k': 1, 'v': 2}	dict 에 dict을 추가
dict.values()	d.values() [1, 2]	dict 내의 값을 리스틀 전달
dict.pop('key')	d {'k': 1, 's': None, 'v': 2} d.pop('s') d {'k': 1, 'v': 2}	dict 내의 원소를 삭제

pop 메소드

collections.OrderedDict은 순서를 유지하기 위해 linked list로 내부에 구성되어 각 순서를 유지함

```
from collections import *
d = {}
d['a'] = 1
d['b'] = 2
print(d)
od = OrderedDict({'b':2, 'a':1})
print(od)
print(od.pop('b'))
print(od)
print(d.pop('b'))
print(d)
```

```
{'a': 1, 'b': 2}
OrderedDict([('b', 2), ('a', 1)])
2
OrderedDict([('a', 1)])
2
{'a': 1}
```

move_to_end 메소드

collections.OrderedDict은 순서를 유지하고 있어서 dict 타입처럼 처리하기 위해서는 move_to_end 메소드를 이용해서 처리

```
from collections import *
d1 = OrderedDict([('a', '1'), ('b', '2')])
d1.update({'c':'3'})
print(d1)
d1.move_to_end('c', last=False)
print(d1)
```

```
OrderedDict([('a', '1'), ('b', '2'), ('c', '3')])
OrderedDict([('c', '3'), ('a', '1'), ('b', '2')])
```

내부 순서가 바뀔 경우 처리

OrderedDict 클래스에순서가 다르면 동등하지 않은 것으로 인식함

```
import collections

a = collections.OrderedDict(name='name', age=30)
print(a)
print(a["name"])
b = collections.OrderedDict(age=30,name='name')
print(b)
print(a == b)
print(b.move_to_end('age',last=False))
print(b)
print(a == b)
```

```
OrderedDict([('name', 'name'), ('age', 30)])
name
OrderedDict([('name', 'name'), ('age', 30)])
True
None
OrderedDict([('age', 30), ('name', 'name')])
False
```

COUNTER

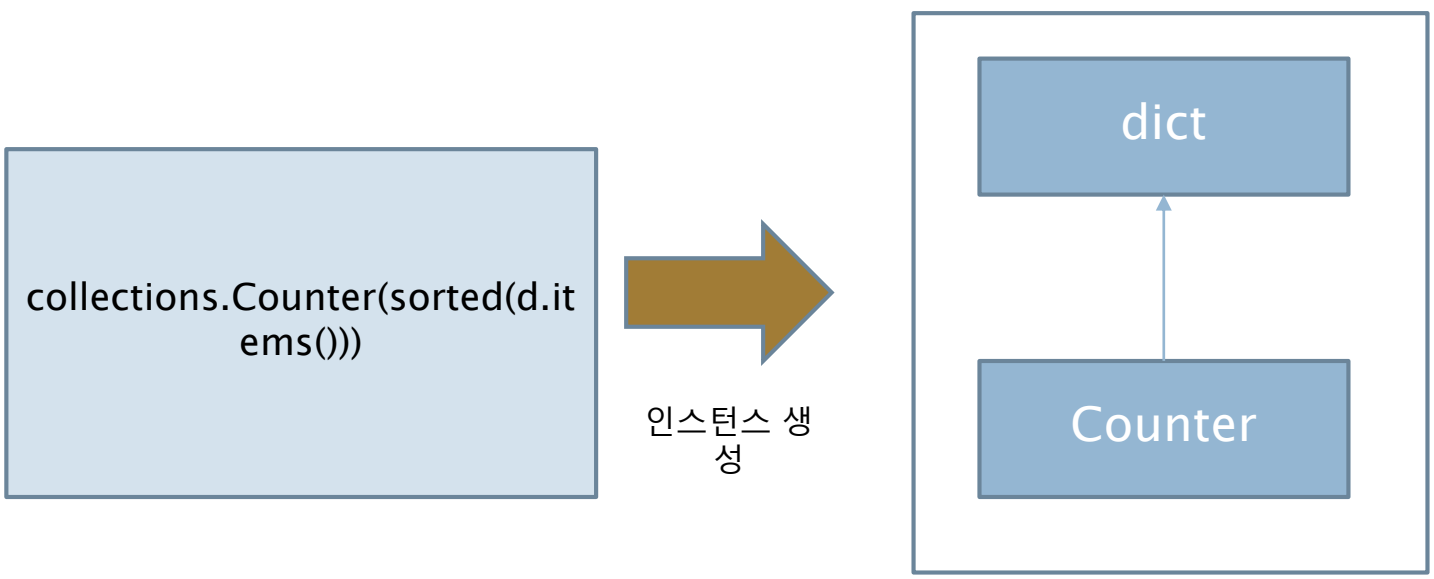
Moon Yong Joon

Counter 구조

Counter 구조

Counter 은 dict의 subclass로써 새로운 인스턴스를 만드는 클래스

issubclass(collections.Counter,dict)
➔ True



Counter 생성 예시

Counter 클래스로 생성하는 이유는 실제 키값들에 연속된 상황이 확인이 필요할 경우 사용

```
>>> import collections
>>> collections.Counter("attacked")
Counter({'a': 2, 't': 2, 'c': 1, 'e': 1, 'd': 1, 'k': 1})
>>>
>>> collections.Counter({1:2,2:2})
Counter({1: 2, 2: 2})
>>> collections.Counter({1:2,2:2}.items())
Counter({(1, 2): 1, (2, 2): 1})
>>> collections.Counter([1,2,3])
Counter({1: 1, 2: 1, 3: 1})
```

Counter 생성 예시

Counter 클래스로 생성하는 이유는 실제 키값들에 연속된 상황이 확인이 필요할 경우 사용

```
import collections

print(collections.Counter("attacked"))

print(collections.Counter({1:2,2:2}))

print(collections.Counter({1:2,2:2}.items()))

print(collections.Counter([1,2,3]))
```

```
Counter({'t': 2, 'a': 2, 'e': 1, 'd': 1, 'k': 1, 'c': 1})
Counter({1: 2, 2: 2})
Counter({(1, 2): 1, (2, 2): 1})
Counter({1: 1, 2: 1, 3: 1})
```

Counter 메소드

Counter 추가 메소드

al = collections.Counter([1,2,3,4])
a2 = collections.Counter({1:2,2:4})

Method	example	Description
elements	al.elements() <itertools.chain at x10542eb0> list(al.elements()) [1, 2, 3, 4]	Counter 인스턴스의 요소를 counter 개수 만큼 보여줌
most_common	list(a2.elements()) [1, 1, 2, 2, 2, 2] a2.most_common() [(2, 4), (1, 2)]	Counter 인스턴스의 값을 튜플로 key/value를 묶어서 리스트로 보여줌
subtract	a2.subtract(al) a2 Counter({2: 3, 1: 1, 3: -1, 4: -1}) a2+al Counter({2: 4, 1: 2})	Counter 인스턴스들간에 값을 빼는 ㄱㅅ

Counter 사칙연산

Counter 인스턴스 내의 키값이 같은 경우에
+/- 연산이 가능하며 zero 값은 표시하지 않음

```
from collections import Counter

s = Counter("abceabde")
s2 = Counter("defabc")
print("s  : ",s)
print("s2 :",s2)

# counter 더하기
sadd = s+s2
print(" s + s2 :",sadd)

#counter  빼기
ssub = s - s2
print(" s - s2 :",ssub)

ssub2 = s2 - s
print(" s2 - s :",ssub2)
```

```
s  :  Counter({'e': 2, 'a': 2, 'b': 2, 'c': 1, 'd': 1})
s2 : Counter({'f': 1, 'b': 1, 'e': 1, 'd': 1, 'a': 1, 'c': 1})
 s + s2 : Counter({'b': 3, 'e': 3, 'a': 3, 'd': 2, 'c': 2, 'f': 1})
 s - s2 : Counter({'a': 1, 'e': 1, 'b': 1})
 s2 - s : Counter({'f': 1})
```

Counter 집합연산

Counter 인스턴스 내의 키값이 같은 경우에 &/| 연산이 가능

```
from collections import Counter

s = Counter("abceabde")
s2 = Counter("defabc")
print( "s  : ",s)
print("s2 :",s2)

# 교집합
sadd = s &s2
print( " s & s2 :",sadd)
# 합집합
ssub = s | s2
print(" s | s2 :",ssub)
```

```
s  :  Counter({'e': 2, 'a': 2, 'b': 2, 'c': 1, 'd': 1})
s2 : Counter({'f': 1, 'b': 1, 'e': 1, 'd': 1, 'a': 1, 'c': 1})
 s & s2 : Counter({'b': 1, 'a': 1, 'e': 1, 'd': 1, 'c': 1})
 s | s2 : Counter({'b': 2, 'e': 2, 'a': 2, 'f': 1, 'd': 1, 'c': 1})
```

Counter 접근

Counter 인스턴스는 dict 타입처럼 키를 통해 접근

```
from collections import Counter

s = Counter("abceabde")

#접근

for i in "abcedf" :
    print (i," : ", s[i])
print(" s element ", [ x for x in s.elements()])
```

```
a  :  2
b  :  2
c  :  1
e  :  2
d  :  1
f  :  0
 s element  ['e', 'e', 'a', 'a', 'c', 'd', 'b', 'b']
```

DEFAULTDICT

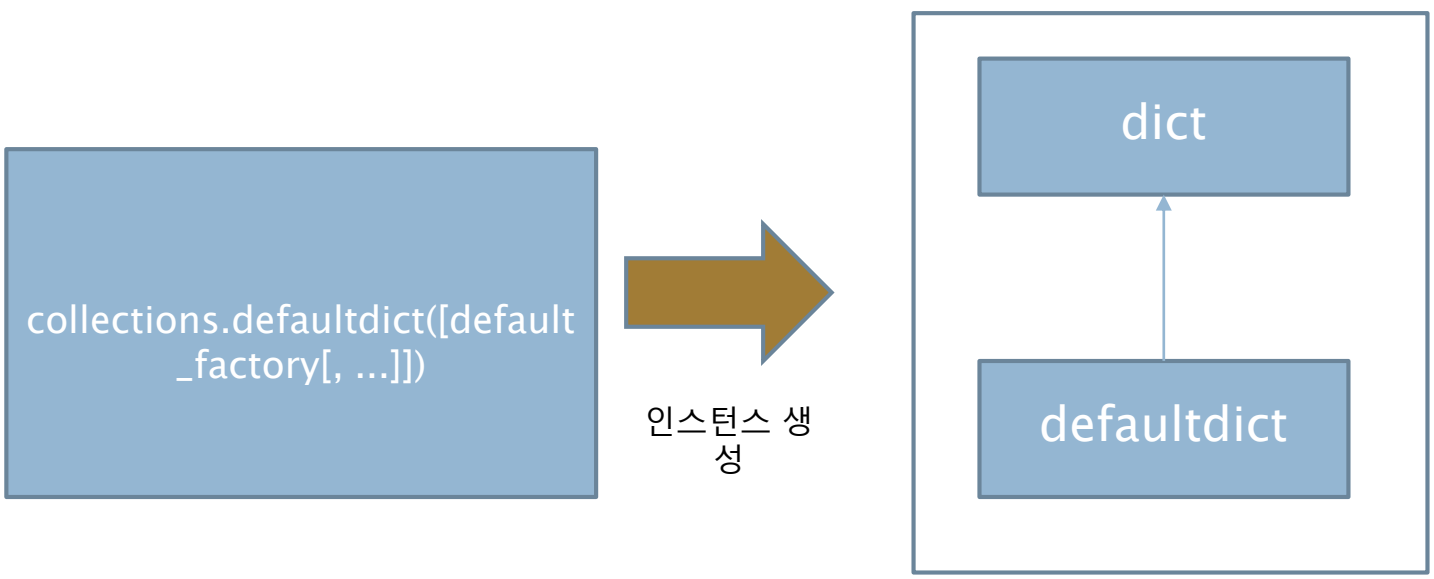
Moon Yong Joon

defaultdict 구조

defaultdict 구조

defaultdict 은 dict의 subclass로써 새로운 인스턴스를 만드는 클래스

issubclass(collections. defaultdict,dict)
➔ True



dict vs, defaultdict

dict 과 defaultdict의 메소드는 거의 유사하지만 차이점은 default dict은 키값이 미존재시 초기값을 자동세팅되어 처리

```
from collections import *
help(defaultdict.default_factory)

a = defaultdict(list)
print(a['key'])

d = dict()
print(d['key'])
```

```
Help on member descriptor collections.defaultdict.default_factory:

default_factory
    Factory for default value called by __missing__().

[]

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-42-33dc99806c53> in <module>()
      6
      7 d = dict()
----> 8 print(d['key'])

KeyError: 'key'
```

defaultdict 생성

defualtdict : list 값 처리

defauldict를 값 객체를 list로 처리하여 순차적인 여러 값(Key: multi- value 구조)을 처리

```
from collections import *

a = defaultdict(list)
print(a)
a['1'].extend([1,2,3])
print(a)
```

```
defaultdict(<class 'list'>, {})
defaultdict(<class 'list'>, {'1': [1, 2, 3]})
```

defualtdict : set 값 처리

defauldict를 값 객체를 list로 처리하여 유일한 원소를 가지는 여러 값(Key: multi- value 구조)을 처리

```
from collections import *

a = defaultdict(set)
print(a)
a['s'].update({1,2,3})
print(a)
```

```
defaultdict(<class 'set'>, {})
defaultdict(<class 'set'>, {'s': {1, 2, 3}})
```

defualtdict : 함수로 값 처리

첫번째 인자에 다양한 데이터 타입이 들어가고 뒤에 인자부터는 dict타입에 맞는 키워드인자로 처리

```
from collections import defaultdict
def default_factory():
    return 'default value'

d = defaultdict(default_factory, foo='bar')
print('d:', d)
print('foo =>', d['foo'])
print('bar =>', d['bar'])

dl = defaultdict(list, foo=[1,2,3])
print ('dl : ',dl)
```

```
d: defaultdict(<function default_factory at 0x0000000004F80F28>, {'foo': 'bar'})
foo => bar
bar => default value
dl :  defaultdict(<class 'list'>, {'foo': [1, 2, 3]})
```

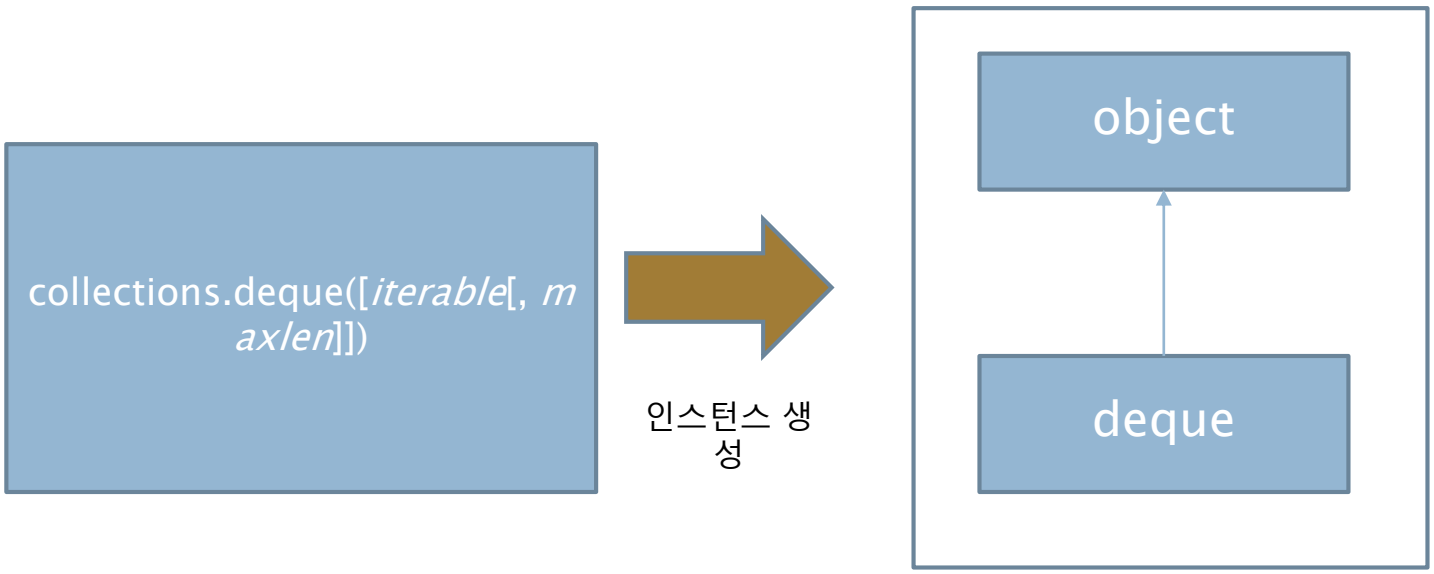
DEQUE

Moon Yong Joon

deque 구조

deque 구조

deque 은 새로운 인스턴스를 만드는 클래스



Deque 메소드

deque 란

양방향에서 처리할 수 있는 queue 자료 구조

deque의 메소드

```
for method in dir(deque) :
    if method.startswith('_') :
        pass
    else :
        print(method)
```

```
append
appendleft
clear
copy
count
extend
extendleft
index
insert
maxlen
pop
popleft
remove
reverse
rotate
```

deque 생성

```
import collections as cols

d = cols.deque([1,2,3], 5)
print(d)
d.extend([4,5])
print(d)

dd = cols.deque([1,2,3])
print(dd)

dd.extend(dd)
print(dd)
```

```
deque([1, 2, 3], maxlen=5)
deque([1, 2, 3, 4, 5], maxlen=5)
deque([1, 2, 3])
deque([1, 2, 3, 1, 2, 3])
```

deque 메소드(1)

deque([])

Method	example	Description
'append',	d.append(1) d deque([1])	우측에 원소 추가
'appendleft',	d.appendleft(3) d deque([3, 1])	좌측에 원소 추가
'clear',	d.clear() d deque([])	요소들을 전부 초기화
'count',	d.count(1) 1	원소의 개수
'extend',	d.extend([2,3,4]) d deque([3, 1, 2, 3, 4])	리스트 등을 기존 인스턴스에 추가
'extendleft',	d.extendleft([10,12]) d deque([12, 10, 3, 1, 2, 3, 4])	리스트 틍을 기존 인스턴스 좌측부터 추가

deque 메소드(2)

d= {"k":1,"v":2}

Method	example	Description
'pop',	deque([3, 10, 12, 4, 3, 2]) d.pop() 2 d deque([3, 10, 12, 4, 3])	우측 끝에 요소를 삭제
'popleft',	deque([3, 10, 12, 4, 3]) d.popleft() 3 d deque([10, 12, 4, 3])	좌측 끝에 요소를 삭제
'remove',	deque([1, 3, 10, 12, 4, 3, 2]) d.remove(1) d deque([3, 10, 12, 4, 3, 2])	값으로 요소를 삭제
'reverse',	deque([2, 3, 4, 12, 10, 3, 1]) d.reverse() d deque([1, 3, 10, 12, 4, 3, 2])	내부 요소들을 역정렬

deque 메소드(3)

d= {"k":1,"v":2}

Method	example	Description
'rotate'	deque([4, 12, 10, 3, 1, 2, 3]) d.rotate(2) d deque([2, 3, 4, 12, 10, 3, 1])	요소들을 n 값만큼 순회

deque 다루기

양방향 queue 다루기

앞과 뒤로 모든 queue 처리가 가능

```
import collections as col
d = col. deque('abcdefg')
print 'Deque:', d
print 'Length:', len(d)
print 'Left end:', d[0]
print 'Right end:', d[-1]

d.remove('c')
print 'remove(c):', d
```

```
Deque: deque(['a', 'b', 'c', 'd', 'e', 'f', 'g'])
Length: 7
Left end: a
Right end: g
remove(c): deque(['a', 'b', 'd', 'e', 'f', 'g'])
```

ITEMGETTER & ATTRGETTER

Moon Yong Joon

collections.itemgetter

Itemgetter : 동일한 키 처리

‘fname’ key를 key값으로 읽는 itg를 생성해서 실제 dict 타입을 파라미터로 주면 값을 결과로 제공

```
rows = [
{'fname': 'Brian', 'lname': 'Jones', 'uid': 1003},
{'fname': 'David', 'lname': 'Beazley', 'uid': 1002},
{'fname': 'John', 'lname': 'Cleese', 'uid': 1001},
{'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
]
from operator import itemgetter
itg = itemgetter('fname')
print(dir(itg))
print(type(itg))
print(itg(rows[0]))
```

```
['__call__', '__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__', '__new__', '__red
uce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
<type 'operator.itemgetter'>
Brian
```

Itemgetter이용해서 sorted

itemgetter이 결과 값을 기준으로 dict 타입 원소를 가지는 list를 sorted 처리하기

```
rows = [
{'fname': 'Brian', 'lname': 'Jones', 'uid': 1003},
{'fname': 'David', 'lname': 'Beazley', 'uid': 1002},
{'fname': 'John', 'lname': 'Cleese', 'uid': 1001},
{'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
]
from operator import itemgetter
itg = itemgetter('fname')
rows_by_fname = sorted(rows, key=itg)

print(rows_by_fname)
```

```
[{'lname': 'Jones', 'uid': 1004, 'fname': 'Big'}, {'lname': 'Jones', 'uid': 1003, 'fname': 'Brian'}, {'lname': 'Beazley', 'ui
d': 1002, 'fname': 'David'}, {'lname': 'Cleese', 'uid': 1001, 'fname': 'John'}]
```

collections.attrgetter

attrgetter 실행

attrgetter에 class의 속성을 부여하고 인스턴스를 파라미터로 받으면 그 결과값인 속성이 값을 가져옴

```
from operator import attrgetter

class User(object):
    def __init__(self, user_id):
        self.user_id = user_id

    def __repr__(self):
        return 'User({})'.format(self.user_id)
x = attrgetter('user_id')
users = [User(1), User(3), User(7)]

print(x(users[0]))
```

```
1
```

attrgetter이용해서 sorted

attrgetter이 결과 값을 기준으로 사용자 클래스의 인스턴스 원소를 가지는 list를 sorted 처리하기

```
class User(object):
    def __init__(self, user_id):
        self.user_id = user_id

    def __repr__(self):
        return 'User({})'.format(self.user_id)

users = [User(1), User(3), User(7)]

from operator import attrgetter
x = sorted(users, key=attrgetter('user_id'))
print(x)
```

```
[User(1), User(3), User(7)]
```

2. COLLECTIONS .ABC

Moon Yong Joon

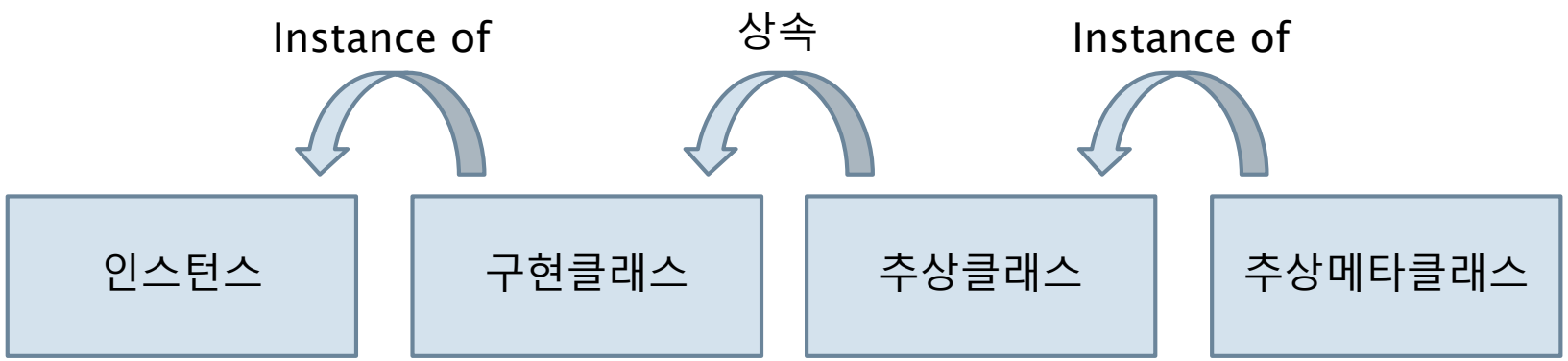
추상클래스 이해

Moon Yong Joon

추상 메타클래스

추상메타클래스란

파이썬에서 추상클래스로 정의시 abc.ABCMeta로 메타클래스를 만들고



추상메타클래스

abc.ABCMeta를 추상 metaclass로써 type을 상속받아 구현된 메타클래스

```
from abc import ABCMeta

print(ABCMeta.__bases__)
print(ABCMeta.mro(ABCMeta))
```

```
(<class 'type'>,)
[<class 'abc.ABCMeta'>, <class 'type'>, <class 'object'>]
```

내장 추상클래스

내장 추상 클래스

abc.ABC를 추상class로 만들어져 있어 이를 상속하면 추상클래스로 처리

```
import abc

print(type(abc.ABC))

class C(ABC) :
    pass

c = C()
print(type(C))
print(C.__bases__)
```

```
<class 'abc.ABCMeta'>
<class 'abc.ABCMeta'>
(<class 'abc.ABC'>,)
```

추상클래스 메소드 처리

abc.ABC를 상속한 추상클래스에 추상화메소드는 반드시 구현클래스에서 정의해서 사용해야 함

```
import abc

print(type(abc.ABC))

class C(ABC) :
    @abc.abstractmethod
    def amethod(self) :
        return
    @abc.abstractmethod
    def bmethod(self) :
        return

class CC(C) :
    def amethod(self) :
        return " concrete CC=> a"
    def bmethod(self) :
        return " concrete CC=> b"
c = CC()
print(type(CC))
print(CC.__bases__)
print(c.amethod())
print(c.bmethod())
```

```
<class 'abc.ABCMeta'>
<class 'abc.ABCMeta'>
(<class '__main__.C'>,)
 concrete CC=> a
 concrete CC=> b
```

사용자 추상클래스

사용자 정의 추상 클래스

abc.ABCMeta를 추상 metaclass에 할당하고 MyABC라는 클래스 정의하고 MyCon에서 MyABC를 상속받음

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass

class MyCon(MyABC) :
    pass
a = MyCon()
print(type(MyABC))
print(type(MyCon))
print(MyCon.__bases__)
print( issubclass(MyCon, MyABC))
print( isinstance(a, MyABC))
```

```
<class 'abc.ABCMeta'>
<class 'abc.ABCMeta'>
(<class '__main__.MyABC'>,)
True
True
```

추상클래스 등록

추상 클래스로 등록

abc.ABCMeta를 추상 metaclass에 할당하고 MyABC라는 클래스정의하고 register 메소드를 통해 class를 등록해서 추상클래스로 사용

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass

MyABC.register(tuple)
print(type(MyABC))
print( issubclass(list, MyABC))
print( issubclass(tuple, MyABC))
```

```
<class 'abc.ABCMeta'>
False
True
```

추상클래스 등록: 데코레이터

추상 클래스와 상속클래스 정의

추상클래스와 상속 클래스 정의

```
import abc


class PB(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def load(self, input):
        """Retrieve data from the input source
        and return an object.
        """

    @abc.abstractmethod
    def save(self, output, data):
        """Save the data object to the output."""

class LocalBaseClass:
    pass
```

추상 클래스와 상속클래스 정의

구현클래스에 추상클래스의 register를 데코레이터로 이용해서 추상클래스로 등록

```
@PB.register
class RegImp(LocalBaseClass):

    def load(self, input):
        return input.read()

    def save(self, output, data):
        return output.write(data)


if __name__ == '__main__':
    print('Subclass:', issubclass(RegImp,PB))
    print('Instance:', isinstance(RegImp(),PB))
```

```
Subclass: True
Instance: True
```

추상메소드 처리

추상메소드

abc 모듈에서는 abstractmethod, abstractclassmethod, abstractstaticmethod, abstractproperty를 지정할 수 있음

```
import abc
dir(abc)
```

```
['ABC',
 'ABCMeta',
 'WeakSet',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'abstractclassmethod',
 'abstractmethod',
 'abstractproperty',
 'abstractstaticmethod',
 'get_cache_token']
```

추상메소드 정의

추상 메소드 정의는 decorator를 이용해서 정의함

```
@abstractmethod
def 메소드명(self, 파라미터) :
        로직
@abstractclassmethod
def 메소드명(cls, 파라미터) :
        로직
 @abstractstaticmethod
 def 메소드명( 파라미터) :
        로직
```

abstractmethod

추상메소드 : 추상클래스 정의

abc 모듈을 이용해서 abstractmethod는 instance method에 대해 지정

```
import abc


class PB(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def load(self, input):
        """Retrieve data from the input source
        and return an object.
        """

    @abc.abstractmethod
    def save(self, output, data):
        """Save the data object to the output."""
```

추상메소드 : 추상클래스 상속

abc 모듈에서는 abstractmethod로 지정한 메소드를 인스턴스메소드로 구현

```
class SubImp(PB):

    def load(self, input):
        return input.read()

    def save(self, output, data):
        return output.write(data)


if __name__ == '__main__':
    print('Subclass:', issubclass(SubImp,PB))
    print('Instance:', isinstance(SubImp(),PB))
```

```
Subclass: True
Instance: True
```

추상 클래스/스태틱 메소드

추상메소드 정의

abc 모듈을 이용해서 abstractclassmethod /abstractstaticmethod 를 정의

```
import abc

print(type(abc.ABC))

class C(ABC) :
    @abc.abstractclassmethod
    def clsmethod(cls) :
        return
    @abc.abstractstaticmethod
    def bmethod() :
        return
```

구현메소드 정의 및 실행

실제 구현 메소드를 정의하고 실행

```
class CC(C) :
    @classmethod
    def clsmethod(cls) :
        return " concrete classmethod CC=> a"
    @staticmethod
    def bmethod() :
        return " concrete staticmethod CC=> b"
c = CC()
print(type(CC))
print(CC.__bases__)
print(c.clsmethod())
print(c.bmethod())
```

```
<class 'abc.ABCMeta'>
<class 'abc.ABCMeta'>
(<class '__main__.C'>,)
 concrete classmethod CC=> a
 concrete staticmethod CC=> b
```

추상프로퍼티 처리

추상property 정의

추상 메소드 정의는 decorator를 이용해서 정의함

```
@abstractproperty
def 메소드명(self, 파라미터) :
        로직
```

abstractproperty

추상클래스에 abstractproperty를 정의하고 구현 메소드에서 property로 로직 구현해야 함

```
from abc import ABCMeta, abstractproperty

class C(metaclass=ABCMeta):
    @abstractproperty
    def x(self):
        pass


class D(C):
    def __init__(self, x) :
        self._x = x
    @property
    def x(self):
        return self._x
    @x.setter
    def x(self, val):
        self._x = val

d = D(10)
print(d)
print(d.x)
d.x = 99
print(d.x)
```

```
<__main__.D object at 0x00000000055D0128>
10
99
```

Abstractmethod로 처리

추상클래스에 abstractmethod를 이용해서 property를 처리시에는 상속된 클래스의 프로퍼티도 동일해야 함

```
from abc import ABCMeta, abstractmethod

class C(metaclass=ABCMeta):
    @property
    def x(self):
        return self._x

    @x.setter
    @abstractmethod
    def x(self, val):
        pass

class D(C):
    def __init__(self, x) :
        self._x = x

    @C.x.setter
    def x(self, val):
        self._x = val

d = D(10)
print(d)
print(d.x)
d.x = 99
print(d.x)
```

```
<__main__.D object at 0x0000000000638A550>
10
99
```

COLLECTIONS.ABC 데이터 구조

Moon Yong Joon

collection 추상 클래스

collections.abc 모듈

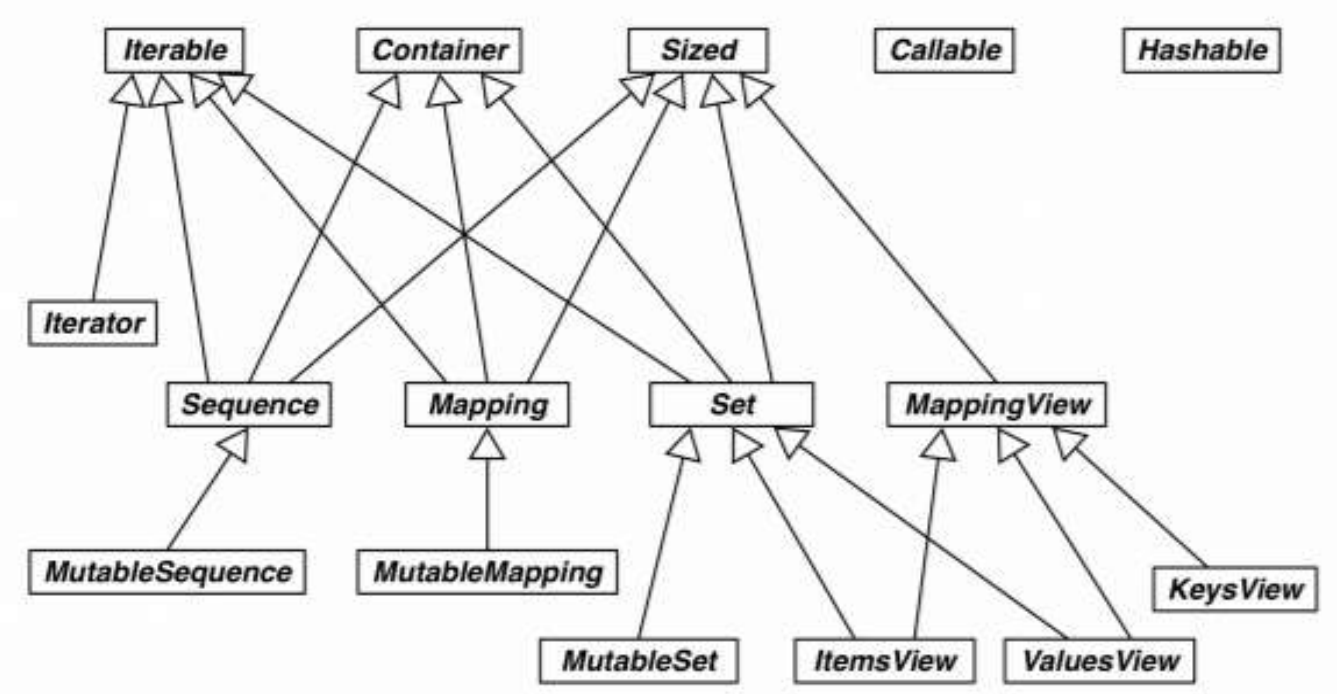
collections.abc모듈은 리스트,문자열, dict, set 타입에 대한 추상클래스를 가진 모듈

```
import collections

dir(collections.abc)
```

['AsyncIterable', 'AsyncIterator', 'Awaitable', 'ByteString', 'Callable', 'Container', 'Coroutine', 'Generator', 'Hashable', 'ItemsView', 'Iterable', 'Iterator', 'KeysView', 'Mapping', 'MappingView', 'MutableMapping', 'MutableSequence', 'MutableSet', 'Sequence', 'Set', 'Sized', 'ValuesView', …]

collections.abc 모듈 diagram

collections.abc 모듈 class diagram



Fluent python 참조

collections.abc 모듈 관계

ABC	Inherits from	Abstract Methods	Mixin Methods
Container		__contains__	
Hashable		__hash__	
Iterable		__iter__	
Iterator	Iterable	__next__	__iter__
Generator	Iterator	send, throw	close, __iter__, __next__
Sized		__len__	
Callable		__call__	
Sequence	Sized, Iterable, Container	__getitem__, __len__	__contains__, __iter__, __reversed__, index, and count
MutableSequence	Sequence	__getitem__, __setitem__, __delitem__, __len__, insert	Inherited Sequence methods and append, reverse, extend, pop,remove, and __iadd__
ByteString	Sequence	__getitem__, __len__	Inherited Sequence methods

collections.abc 모듈 관계

ABC	Inherits from	Abstract Methods	Mixin Methods
Set	Sized, Iterable, Container	__contains__, __iter__, __len__	__le__, __lt__, __eq__, __ne__, __gt__, __ge__, __and__, __or__,__sub__, __xor__, and isdisjoint
MutableSet	Set	__contains__, __iter__, __len__, add, discard	Inherited Set methods and clear, pop, remove, __ior__, __iand__, __ixor__, and __isub__
Mapping	Sized, Iterable, Container	__getitem__, __iter__, __len__	__contains__, keys, items, values, get, __eq__, and __ne__
MutableMapping	Mapping	__getitem__, __setitem__, __delitem__, __iter__, __len__	Inherited Mapping methods and pop, popitem, clear, update, and setdefault
MappingView	Sized		__len__
ItemsView	MappingView, Set		__contains__, __iter__
KeysView	MappingView, Set		__contains__, __iter__
ValuesView	MappingView		__contains__, __iter__

collections.abc 모듈 관계

ABC	Inherits from	Abstract Methods	Mixin Methods
Awaitable		__await__	
Coroutine	Awaitable	send, throw	close
AsyncIterable		__aiter__	
AsyncIterator	AsyncIterable	__anext__	__aiter__

기본 추상 class

기본 추상 class

Container/Hashable/Sized/Iterable/Callable은 기본 추상 클래스

ABC	Abstract Methods
Container	__contains__
Hashable	__hash__
Iterable	__iter__
Sized	__len__
Callable	__call__

기본 추상 class 관계

Container/Hashable/Sized/Iterable/Callable의 상속 및 메타클래스 관계

```
import collections.abc

print(collections.abc.Iterable.__bases__)
print(collections.abc.Callable.__bases__)
print(collections.abc.Hashable.__bases__)
print(collections.abc.Container.__bases__)
print(collections.abc.Sized.__bases__)

print(collections.abc.Iterable.__class__)
print(collections.abc.Callable.__class__)
print(collections.abc.Hashable.__class__)
print(collections.abc.Container.__class__)
print(collections.abc.Sized.__class__)
```

```
(<class 'object'>,)
(<class 'object'>,)
(<class 'object'>,)
(<class 'object'>,)
(<class 'object'>,)
<class 'abc.ABCMeta'>
<class 'abc.ABCMeta'>
<class 'abc.ABCMeta'>
<class 'abc.ABCMeta'>
<class 'abc.ABCMeta'>
```

Iterator/Generator 추상 클래스

Iterator/Generator 타입 상속관계

Iterator는 Iterable을 상속하고 Generator는 Iterator를 상속하는 구조

```
import collections.abc

print(collections.abc.Iterable.__bases__)
print(collections.abc.Iterator.__bases__)
print(collections.abc.Generator.__bases__)
```

```
(<class 'object'>,)
(<class 'collections.abc.Iterable'>,)
(<class 'collections.abc.Iterator'>,)
```

Iterator 타입 처리

문자열 sequence를 iter()함수 처리한 결과는 itoractor 클래스가 생김

```
import collections.abc

s = "strings"
print(isinstance(s, collections.abc.Sequence))
print(isinstance(s, collections.abc.Iterator))

it = iter(s)
print(it)
print(isinstance(it, collections.abc.Sequence))
print(isinstance(it, collections.abc.Iterator))
```

```
True
False
<str_iterator object at 0x0000000005189588>
False
True
```

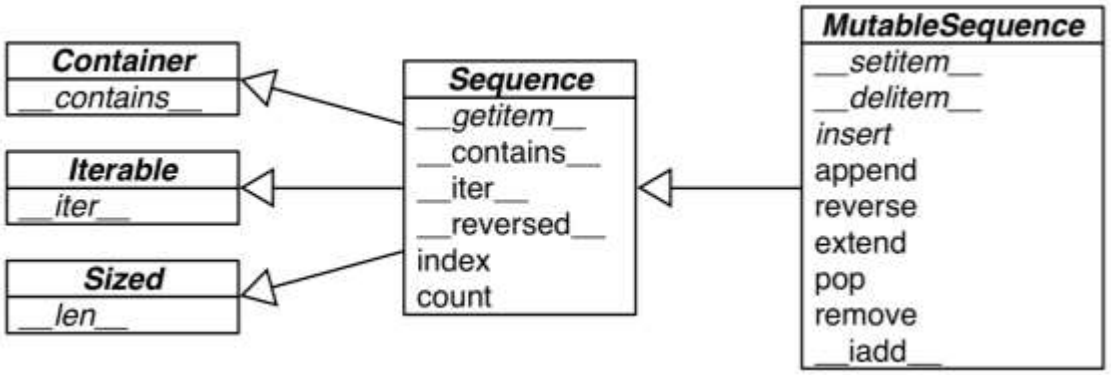
SEQUENCE 추상 클래스

SEQUENCE 상속 class

ABC	Inherits from	Abstract Methods	Mixin Methods
Container		__contains__	
Iterable		__iter__	
Sized		__len__	
Callable		__call__	
Sequence	Sized, Iterable, Container	__getitem__, __len__	__contains__, __iter__, __reversed__, index, and count
MutableSequence	Sequence	__getitem__, __setitem__, __delitem__, __len__, insert	Inherited Sequence methods and append, reverse, extend, pop,remove, and __iadd__
ByteString	Sequence	__getitem__, __len__	Inherited Sequence methods

Sequence 타입 class diagram

Sequence 타입에 대한 class diagram



Fluent python 참조

Sequence 타입 상속관계

Sized, Iterabel, Container를 기본으로 상속해서 {'__iter__', '__len__', '__contains__'} 메소드를 구현

```
import collections.abc as abc
print(abc.Sequence.__bases__)
s = abc.Sized.__dict__.keys()
i = abc.Iterable.__dict__.keys()
c = abc.Container.__dict__.keys()
d1 = s^i
d2 = s^c
print(d1 | d2)
```

(<class 'collections.abc.Sized'>, <class 'collections.abc.Iterable'>, <class 'collections.abc.Container'>)
{'__iter__', '__len__', '__contains__'}

Sequence 타입 내부메소드

Sequence 타입 내부에 스페셜 메소드가 구현

```
import collections.abc as abc
print(abc.Sequence.__bases__)

st = abc.Sequence.__dict__.keys()
s = abc.Sized.__dict__.keys()
i = abc.Iterable.__dict__.keys()
c = abc.Container.__dict__.keys()

s1 = st - s
s2 = st - i
s3 = st - c
s4 = s1 - i
s5 = s1 - c
print(s4 | s5 )
```

(<class 'collections.abc.Sized'>,
<class 'collections.abc.Iterable'>,
<class 'collections.abc.Container'>)
{'count', 'index', '__reversed__',
'__getitem__', '__iter__', '__contains__'}

SET 추상 클래스

Set 타입 상속관계

Sized, Iterabel, Container를 기본으로 상속해서 {'__iter__', '__len__', '__contains__'} 메소드를 구현

```
import collections.abc as abc
print(abc.Sequence.__bases__)
s = abc.Sized.__dict__.keys()
i = abc.Iterable.__dict__.keys()
c = abc.Container.__dict__.keys()
d1 = s^i
d2 = s^c
print(d1 | d2)
```

(<class 'collections.abc.Sized'>, <class 'collections.abc.Iterable'>, <class 'collections.abc.Container'>)
{'__iter__', '__len__', '__contains__'}

Set 타입 내부메소드

Set 타입 내부에 스페셜 메소드가 구현

```
import collections.abc as abc
print(abc.Set.__bases__)

st = abc.Set.__dict__.keys()
s = abc.Sized.__dict__.keys()
i = abc.Iterable.__dict__.keys()
c = abc.Container.__dict__.keys()

s1 = st - s
s2 = st - i
s3 = st - c
s4 = s1 - i
s5 = s1 - c
print(s4 | s5 )
```

(<class 'collections.abc.Sized'>,
<class 'collections.abc.Iterable'>,
<class 'collections.abc.Container'>)
{'__hash__', '__rand__',
'_from_iterable', '__lt__', 'isdisjoint',
'__or__', '__and__', '__ge__', '_hash',
'__rxor__', '__ror__', '__eq__', '__le__',
'__xor__', '__rsub__', '__gt__',
'__sub__'}

MAPPING 추상 클래스

MAPPING 모듈 관계

ABC	Inherits from	Abstract Methods	Mixin Methods
Mapping	Sized, Iterable, Container	__getitem__, __iter__, __len__	__contains__, keys, items, values, get, __eq__, __ne__
MutableMapping	Mapping	__getitem__, __setitem__, __delitem__, __iter__, __len__	Inherited Mapping methods pop, popitem, clear, update, and setdefault

Mapping 타입 상속관계

Sized, Iterabel, Container를 기본으로 상속해서 {'__iter__', '__len__', '__contains__'} 메소드를 구현

```
import collections.abc as abc
print(abc.Mapping.__bases__)
s = abc.Sized.__dict__.keys()
i = abc.Iterable.__dict__.keys()
c = abc.Container.__dict__.keys()
d1 = s^i
d2 = s^c
print(d1 | d2)
```

(<class 'collections.abc.Sized'>, <class 'collections.abc.Iterable'>, <class 'collections.abc.Container'>)
{'__iter__', '__len__', '__contains__'}

Mapping 내부메소드

타입 내부에 스페셜 메소드가 구현

```
import collections.abc as abc
print(abc.Mapping.__bases__)

st = abc.Mapping.__dict__.keys()
s = abc.Sized.__dict__.keys()
i = abc.Iterable.__dict__.keys()
c = abc.Container.__dict__.keys()

s1 = st - s
s2 = st - i
s3 = st - c
s4 = s1 - i
s5 = s1 - c
print(s4 | s5 )
```

(<class 'collections.abc.Sized'>, <class 'collections.abc.Iterable'>, <class 'collections.abc.Container'>)
{'keys', '__hash__', 'items', 'get', '__eq__', '__getitem__', 'values', '__contains__'}

VIEW 추상 클래스

VIEW 모듈 관계

ABC	Inherits from	Mixin Methods
MappingView	Sized	__len__
ItemsView	MappingView, Set	__contains__, __iter__
KeysView	MappingView, Set	__contains__, __iter__
ValuesView	MappingView	__contains__, __iter__

view 타입 상속관계

dict 타입 내부의 keys, values, items에 대한 데이터 타입의 추상클래스

```
#mapping view
print(abc.Set.__bases__)
print(abc.MappingView.__bases__)
print(abc.KeysView.__bases__)
print(abc.ValuesView.__bases__)
print(abc.ItemsView.__bases__)

a = {'a':1, 'b':2}
print(type(a.keys()), isinstance(a.keys(), abc.KeysView))
print(type(a.keys()), issubclass(a.keys().__class__, abc.KeysView))
```

view 타입 상속관계 : 결과

View 타입은 Set또는 MappingView를 상속해서 처리

```
(<class 'collections.abc.Sized'>, <class 'collections.abc.Iterable'>,
<class 'collections.abc.Container'>)
 (<class 'collections.abc.Sized'>,)
 (<class 'collections.abc.MappingView'>, <class 'collections.abc.Set'>)
(<class 'collections.abc.MappingView'>,)
 (<class 'collections.abc.MappingView'>, <class 'collections.abc.Set'>)
<class 'dict_keys'>
True
 <class 'dict_keys'>
True
```