Collections and Frozen Set

Python FrozenSet

- Python frozenset is an unordered collection of distinct hashable objects. Frozenset is an immutable set, so its contents cannot be modified after it’s created.

Set vs Frozenset

- The set type is mutable -- the contents can be changed using methods like add() and remove(). Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set. The frozenset type is immutable and hashable -- its contents cannot be altered after is created; however, it can be used as a dictionary key or as an element of another set.
-

collections

- This module implements specialized container datatypes providing alternatives to Python’s general purpose built-in containers, dict, list, set, and tuple.

Collection List

namedtuple()	factory function for creating tuple subclasses with named fields
deque	list-like container with fast appends and pops on either end
ChainMap	dict-like class for creating a single view of multiple mappings
Counter	dict subclass for counting hashable objects
OrderedDict	dict subclass that remembers the order entries were added
defaultdict	dict subclass that calls a factory function to supply missing values
UserDict	wrapper around dictionary objects for easier dict subclassing
UserList	wrapper around list objects for easier list subclassing
UserString	wrapper around string objects for easier string subclassing

ChainMap

- A ChainMap class is provided for quickly linking a number of mappings so they can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple update() calls.

- >>> baseline = {'music': 'bach', 'art': 'rembrandt'}
- >>> adjustments = {'art': 'van gogh', 'opera': 'carmen'}
- >>> list(ChainMap(adjustments, baseline))
- ['music', 'art', 'opera']

Counter

- A counter tool is provided to support convenient and rapid tallies. For example:A Counter is a dict subclass for counting hashable objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The Counter class is similar to bags or multisets in other languages.

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})
```

Counter

- elements()
- Return an iterator over elements repeating each as many times as its count. Elements are returned in the order first encountered. If an element’s count is less than one, elements() will ignore it.

- c = Counter(a=4, b=2, c=0, d=-2)
- sorted(c.elements())
- ['a', 'a', 'a', 'a', 'b', 'b']

Counter

- subtract([iterable-or-mapping])¶
- Elements are subtracted from an iterable or from another mapping (or counter). Like dict.update() but subtracts counts instead of replacing them. Both inputs and outputs may be zero or negative.

- c = Counter(a=4, b=2, c=0, d=-2)
- d = Counter(a=1, b=2, c=3, d=4)
- c.subtract(d)
- c
- Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})

Counter

- most_common([n])
- Return a list of the n most common elements and their counts from the most common to the least. If n is omitted or None, most_common() returns all elements in the counter. Elements with equal counts are ordered in the order first encountered:

- Counter('abracadabra').most_common(3)
- [('a', 5), ('b', 2), ('r', 2)]

deques

- This section shows various approaches to working with deques.

- Bounded length deques provide functionality similar to the tail filter in Unix:

deque

- >>> d = deque('ghi') # make a new deque with three items
- >>> for elem in d: # iterate over the deque's elements
- ... print(elem.upper())
- G
- H
- I

- >>> d.append('j') # add a new entry to the right side
- >>> d.appendleft('f') # add a new entry to the left side
- >>> d # show the representation of the deque
- deque(['f', 'g', 'h', 'i', 'j'])

deque

- >>> d.pop() # return and remove the rightmost item
- 'j'
- >>> d.popleft() # return and remove the leftmost item
- 'f'
- >>> list(d) # list the contents of the deque
- ['g', 'h', 'i']
- >>> d[0] # peek at leftmost item
- 'g'
- >>> d[-1] # peek at rightmost item
- 'i'

deque

- >>> list(reversed(d)) # list the contents of a deque in reverse
- ['i', 'h', 'g']
- >>> 'h' in d # search the deque
- True
- >>> d.extend('jkl') # add multiple elements at once
- >>> d
- deque(['g', 'h', 'i', 'j', 'k', 'l'])
- >>> d.rotate(1) # right rotation
- >>> d
- deque(['l', 'g', 'h', 'i', 'j', 'k'])
- >>> d.rotate(-1) # left rotation
- >>> d
- deque(['g', 'h', 'i', 'j', 'k', 'l'])

deque

- >>> deque(reversed(d)) # make a new deque in reverse order
- deque(['l', 'k', 'j', 'i', 'h', 'g'])
- >>> d.clear() # empty the deque
- >>> d.pop() # cannot pop from an empty deque
- Traceback (most recent call last):
- File "<pyshell#6>", line 1, in -toplevel-
- d.pop()
- IndexError: pop from an empty deque

- >>> d.extendleft('abc') # extendleft() reverses the input order
- >>> d
- deque(['c', 'b', 'a'])

defaultdict

- Using list as the default_factory, it is easy to group a sequence of key-value pairs into a dictionary of lists:
- s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
- d = defaultdict(list)
- for k, v in s:
- d[k].append(v)

- sorted(d.items())
- [('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]

Named tuples

- Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code. They can be used wherever regular tuples are used, and they add the ability to access fields by name instead of position index.

Named tuples

- name = (“Python”,8, ‘some')
- print(name)

namedtuple

- >>> # Basic example
- >>> Point = namedtuple('Point', ['x', 'y'])
- >>> p = Point(11, y=22) # instantiate with positional or keyword arguments
- >>> p[0] + p[1] # indexable like the plain tuple (11, 22)
- 33
- >>> x, y = p # unpack like a regular tuple
- >>> x, y
- (11, 22)
- >>> p.x + p.y # fields also accessible by name
- 33
- >>> p # readable __repr__ with a name=value style
- Point(x=11, y=22)

namedtuple

- import collections

- User = collections.namedtuple('User', 'name no version')
- Var_name = User(name=‘Python', no=8, version=‘V')
- print(Var_name)

- print('Name of User: {0}'.format(Var_name.name))

Ordered dictionaries

- Ordered dictionaries are just like regular dictionaries but have some extra capabilities relating to ordering operations. They have become less important now that the built-in dict class gained the ability to remember insertion order (this new behavior became guaranteed in Python 3.7).

OrderedDict Features

- The OrderedDict was designed to be good at reordering operations. Space efficiency, iteration speed, and the performance of update operations were secondary.
- Algorithmically, OrderedDict can handle frequent reordering operations better than dict. This makes it suitable for tracking recent accesses (for example in an LRU cache).
- The equality operation for OrderedDict checks for matching order.
- The popitem() method of OrderedDict has a different signature. It accepts an optional argument to specify which item is popped.
- OrderedDict has a move_to_end() method to efficiently reposition an element to an endpoint.

OrderedDict

- >>> d = OrderedDict.fromkeys('abcde')
- >>> d.move_to_end('b')
- >>> ''.join(d.keys())
- 'acdeb'
- >>> d.move_to_end('b', last=False)
- >>> ''.join(d.keys())
- 'bacde'