Collections

All Versions

Examples

collections.Counter

Counter is a dict sub class that allows you to easily count objects. It has utility methods for working with the frequencies of the objects that you are counting.

```
import collections
counts = collections.Counter([1,2,3])
```

the above code creates an object, counts, which has the frequencies of all the elements passed to the constructor. This example has the value Counter({1: 1, 2: 1, 3: 1})

Constructor examples

Letter Counter

```
>>> collections.Counter('Happy Birthday')
Counter({'a': 2, 'p': 2, 'y': 2, 'i': 1, 'r': 1, 'B': 1, ' ': 1, 'H': 1, 'd': 1, 'h': 1, 't': 1
```

Word Counter

```
>>> collections.Counter('I am Sam Sam I am That Sam-I-am That Sam-I-am! I do not like that Sam-I
Counter({'I': 3, 'Sam': 2, 'Sam-I-am': 2, 'That': 2, 'am': 2, 'do': 1, 'Sam-I-am!': 1, 'that': 1
```

Recipes

```
>>> c = collections.Counter({'a': 4, 'b': 2, 'c': -2, 'd': 0})
```

Get count of individual element

```
>>> c['a']
4
```

Set count of individual element

```
>>> c['c'] = -3
>>> c
Counter({'a': 4, 'b': 2, 'd': 0, 'c': -3})
```

Get total number of elements in counter (4 + 2 + 0 - 3)

```
>>> sum(c.itervalues())  # negative number are counted!
3
```

Get elements (only those with positive counter are kept)

```
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

Remove keys with 0 or negative value

```
>>> c - collections.Counter()
Counter({'a': 4, 'b': 2})
```

Remove everything

```
>>> c.clear()
>>> c
Counter()
```

Add remove individual elements

```
>>> c.update({'a': 3, 'b':3})
>>> c.update({'a': 2, 'c':2})  # adds to existing, sets if they don't exist
>>> c
Counter({'a': 5, 'b': 3, 'c': 2})
>>> c.subtract({'a': 3, 'b': 3, 'c': 3})  # subtracts (negative values are allowed)
>>> c
Counter({'a': 2, 'b': 0, 'c': -1})
```

collections.defaultdict

collections.defaultdict (default_factory) returns a subclass of dict that has a default value for missing keys. The argument should be a function that returns the default value when called with no arguments. If there is nothing passed, it defaults to None .

```
>>> state_capitals = collections.defaultdict(str)
>>> state_capitals
defaultdict(<class 'str'>, {})
```

returns a reference to a defaultdict that will create a string object with its default_factory method.

A typical usage of defaultdict is to use one of the builtin types such as str , int , list or dict as the default_factory, since these return empty types when called with no arguments:

```
>>> str()
''
>>> int()
0
>>> list
[]
```

Calling the defaultdict with a key that does not exist does not produce an error as it would in a normal dictionary.

```
>>> state_capitals['Alaska']
''
>>> state_capitals
defaultdict(<class 'str'>, {'Alaska': ''})
```

Another example with int :

```
>>> fruit_counts = defaultdict(int)
>>> fruit_counts['apple'] += 2  # No errors should occur
>>> fruit_counts
default_dict(int, {'apple': 2})
>>> fruit_counts['banana']  # No errors should occur
0
>>> fruit_counts  # A new key is created
default_dict(int, {'apple': 2, 'banana': 0})
```

Normal dictionary methods work with the default dictionary

```
>>> state_capitals['Alabama'] = 'Montgomery'
>>> state_capitals
defaultdict(<class 'str'>, {'Alabama': 'Montgomery', 'Alaska': ''})
```

Using list as the default_factory will create a list for each new key.

```
>>> s = [('NC', 'Raleigh'), ('VA', 'Richmond'), ('WA', 'Seattle'), ('NC', 'Asheville')]
>>> dd = collections.defaultdict(list)
>>> for k, v in s:
...     dd[k].append(v)
>>> dd
defaultdict(<class 'list'>,
    {'VA': ['Richmond'],
     'NC': ['Raleigh', 'Asheville'],
     'WA': ['Seattle']})
```

collections.namedtuple

Define a new type Person using namedtuple like this:

```
Person = namedtuple('Person', ['age', 'height', 'name'])
```

The second argument is the list of attributes that the tuple will have. You can list these attributes also as either space or comma separated string:

```
Person = namedtuple('Person', 'age, height, name')
```

or

```
Person = namedtuple('Person', 'age height name')
```

Once defined, a named tuple can be instantiated by calling the object with the necessary parameters, e.g.:

```
dave = Person(30, 178, 'Dave')
```

Named arguments can also be used:

```
jack = Person(age=30, height=178, name='Jack S.')
```

Now you can access the attributes of the namedtuple:

```
print(jack.age)  # 30
print(jack.name)  # 'Jack S.'
```

The first argument to the namedtuple constructor (in our example 'Person') is the typename . It is typical to use the same word for the constructor and the typename, but they can be different:

```
Human = namedtuple('Person',  'age, height, name')
dave = Human(30, 178, 'Dave')
print(dave)  # yields: Person(age=30, height=178, name='Dave')
```

collections.OrderedDict

The order of keys in Python dictionaries is arbitrary: they are not governed by the order in which you add them.

For example:

```
>>> d = {'foo': 5, 'bar': 6}
>>> print(d)
{'foo': 5, 'bar': 6}
>>> d['baz'] = 7
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6}
>>> d['foobar'] = 8
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6, 'foobar': 8}
```

(The arbitrary ordering implied above means that you may get different results with the above code to that shown here.)

The order in which the keys appear is the order which they would be iterated over, e.g. using a for loop.

The collections.OrderedDict class provides dictionary objects that retain the order of keys. OrderedDict s can be created as shown below with a series of ordered items (here, a list of tuple key-value pairs):

```
>>> from collections import OrderedDict
>>> d = OrderedDict([('foo', 5), ('bar', 6)])
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6)])
>>> d['baz'] = 7
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7)])
>>> d['foobar'] = 8
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

Or we can create an empty OrderedDict and then add items:

```
>>> o = OrderedDict()
>>> o['key1'] = "value1"
>>> o['key2'] = "value2"
>>> print(o)
OrderedDict([('key1', 'value1'), ('key2', 'value2')])
```

Iterating through an OrderedDict allows key access in the order they were added.

What happens if we assign a new value to an existing key?

```
>>> d['foo'] = 4
>>> print(d)
OrderedDict([('foo', 4), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

The key retains its original place in the OrderedDict .

collections.deque

Returns a new deque object initialized left-to-right (using append()) with data from iterable. If iterable is not specified, the new deque is empty.

Deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction.

Though list objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movement costs for pop(0) and insert(0, v) operations which change both the size and position of the underlying data representation.

New in version 2.4.

If maxlen is not specified or is None , deques may grow to an arbitrary length. Otherwise, the deque is bounded to the specified maximum length. Once a bounded length deque is full, when new items are added, a corresponding number of items are discarded from the opposite end. Bounded length deques provide functionality similar to the tail filter in Unix. They are also useful for tracking transactions and other pools of data where only the most recent activity is of interest.

Changed in version 2.6: Added maxlen parameter.

```
>>> from collections import 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'])

>>> 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'

>>> 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'])
```

Source: https://docs.python.org/2/library/collections.html

Syntax

Parameters

Remarks