9.3. `collections` — High-performance container datatypes¶

New in version 2.4.

This module implements high-performance container datatypes. Currently, there are two datatypes, `deque` and `defaultdict`, and one datatype factory function, `namedtuple()`.

Changed in version 2.5: Added `defaultdict`.

Changed in version 2.6: Added `namedtuple()`.

The specialized containers provided in this module provide alternatives to Python's general purpose built-in containers, `dict`, `list`, `set`, and `tuple`.

Besides the containers provided here, the optional `bsddb` module offers the ability to create in-memory or file based ordered dictionaries with string keys using the `bsddb.btopen()` method.

In addition to containers, the collections module provides some ABCs (abstract base classes) that can be used to test whether a class provides a particular interface, for example, is it hashable or a mapping.

Changed in version 2.6: Added abstract base classes.

9.3.1. ABCs - abstract base classes¶

The collections module offers the following ABCs:

ABC	Inherits	Abstract Methods	Mixin Methods
`Container`		`__contains__`	
`Hashable`		`__hash__`	
`Iterable`		`__iter__`	
`Iterator`	`Iterable`	`__next__`	`__iter__`
`Sized`		`__len__`	
`Callable`		`__call__`	
`Sequence`	`Sized`, `Iterable`, `Container`	`__getitem__`	`__contains__`. `__iter__`, `__reversed__`. `index`, and `count`
`MutableSequence`	`Sequence`	`__setitem__` `__delitem__`, and `insert`	Inherited Sequence methods and `append`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`
`Set`	`Sized`, `Iterable`, `Container`		`__le__`, `__lt__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__and__`, `__or__` `__sub__`, `__xor__`, and `isdisjoint`
`MutableSet`	`Set`	`add` and `discard`	Inherited Set methods and `clear`, `pop`, `remove`, `__ior__`, `__iand__`, `__ixor__`, and `__isub__`
`Mapping`	`Sized`, `Iterable`, `Container`	`__getitem__`	`__contains__`, `keys`, `items`, `values`, `get`, `__eq__`, and `__ne__`
`MutableMapping`	`Mapping`	`__setitem__` and `__delitem__`	Inherited Mapping methods and `pop`, `popitem`, `clear`, `update`, and `setdefault`
`MappingView`	`Sized`		`__len__`
`KeysView`	`MappingView`, `Set`		`__contains__`, `__iter__`
`ItemsView`	`MappingView`, `Set`		`__contains__`, `__iter__`
`ValuesView`	`MappingView`		`__contains__`, `__iter__`

These ABCs allow us to ask classes or instances if they provide particular functionality, for example:

```
size = None
if isinstance(myvar, collections.Sized):
   size = len(myvar)
```

Several of the ABCs are also useful as mixins that make it easier to develop classes supporting container APIs. For example, to write a class supporting the full `Set` API, it only necessary to supply the three underlying abstract methods: `__contains__()`, `__iter__()`, and `__len__()`. The ABC supplies the remaining methods such as `__and__()` and `isdisjoint()`

```
class ListBasedSet(collections.Set):
    ''' Alternate set implementation favoring space over speed
        and not requiring the set elements to be hashable. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)
    def __iter__(self):
        return iter(self.elements)
    def __contains__(self, value):
        return value in self.elements
    def __len__(self):
        return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2            # The __and__() method is supported automatically
```

Notes on using `Set` and `MutableSet` as a mixin:

1. Since some set operations create new sets, the default mixin methods need a way to create new instances from an iterable. The class constructor is assumed to have a signature in the form `ClassName(iterable)`. That assumption is factored-out to an internal classmethod called `_from_iterable()` which calls `cls(iterable)` to produce a new set. If the `Set` mixin is being used in a class with a different constructor signature, you will need to override `from_iterable()` with a classmethod that can construct new instances from an iterable argument.
2. To override the comparisons (presumably for speed, as the semantics are fixed), redefine `__le__()` and then the other operations will automatically follow suit.
3. The `Set` mixin provides a `_hash()` method to compute a hash value for the set; however, `__hash__()` is not defined because not all sets are hashable or immutable. To add set hashabilty using mixins, inherit from both `Set()` and `Hashable()`, then define `__hash__ = Set._hash`.

See also

- OrderedSet recipe for an example built on `MutableSet`.
- For more about ABCs, see the `abc` module and **PEP 3119**.

9.3.2. deque objects¶

class `collections.deque`([*iterable*[, *maxlen*]])¶

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.

Deque objects support the following methods:

`append`(*x*)¶
Add *x* to the right side of the deque.

`appendleft`(*x*)¶
Add *x* to the left side of the deque.

`clear`()¶
Remove all elements from the deque leaving it with length 0.

`extend`(*iterable*)¶

Extend the right side of the deque by appending elements from the iterable argument.

`extendleft`(*iterable*)¶

Extend the left side of the deque by appending elements from *iterable*. Note, the series of left appends results in reversing the order of elements in the iterable argument.

`pop`()¶

Remove and return an element from the right side of the deque. If no elements are present, raises an `IndexError`.

`popleft`()¶

Remove and return an element from the left side of the deque. If no elements are present, raises an `IndexError`.

`remove`(*value*)¶

Removed the first occurrence of *value*. If not found, raises a `ValueError`.

New in version 2.5.

`rotate`(*n*)¶

Rotate the deque *n* steps to the right. If *n* is negative, rotate to the left. Rotating one step to the right is equivalent to: `d.appendleft(d.pop())`.

In addition to the above, deques support iteration, pickling, `len(d)`, `reversed(d)`, `copy.copy(d)`, `copy.deepcopy(d)`, membership testing with the `in` operator, and subscript references such as `d[-1]`. Indexed access is O(1) at both ends but slows to O(n) in the middle. For fast random access, use lists instead.

Example:

```
>>> 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'])

>>> 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'])
```

9.3.2.1. deque Recipes¶

This section shows various approaches to working with deques.

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

```
def tail(filename, n=10):
   'Return the last n lines of a file'
   return deque(open(filename), n)
```

Another approach to using deques is to maintain a sequence of recently added elements by appending to the right and popping to the left:

```
def moving_average(iterable, n=3):
   # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
   # http://en.wikipedia.org/wiki/Moving_average
   it = iter(iterable)
   d = deque(itertools.islice(it, n-1))
   d.appendleft(0)
   s = sum(d)
   for elem in it:
       s += elem - d.popleft()
       d.append(elem)
       yield s / float(n)
```

The `rotate()` method provides a way to implement deque slicing and deletion. For example, a pure Python implementation of `del d[n]` relies on the `rotate()` method to position elements to be popped:

```
def delete_nth(d, n):
   d.rotate(-n)
   d.popleft()
   d.rotate(n)
```

To implement deque slicing, use a similar approach applying `rotate()` to bring a target element to the left side of the deque. Remove old entries with `popleft()`, add new entries with `extend()`, and then reverse the rotation. With minor variations on that approach, it is easy to implement Forth style stack manipulations such as `dup`, `drop`, `swap`, `over`, `pick`, `rot`, and `roll`.

9.3.3. `defaultdict` objects¶

class `collections.defaultdict`([*default_factory*[, *...*]])¶

Returns a new dictionary-like object. `defaultdict` is a subclass of the built-in `dict` class. It overrides one method and adds one writable instance variable. The remaining functionality is the same as for the `dict` class and is not documented here.

The first argument provides the initial value for the `default_factory` attribute; it defaults to `None`. All remaining arguments are treated the same as if they were passed to the `dict` constructor, including keyword arguments.

New in version 2.5.

`defaultdict` objects support the following method in addition to the standard `dict` operations:

`__missing__`(*key*)¶

If the `default_factory` attribute is `None`, this raises a `KeyError` exception with the *key* as argument.

If `default_factory` is not `None`, it is called without arguments to provide a default value for the given *key*, this value is inserted in the dictionary for the *key*, and returned.

If calling `default_factory` raises an exception this exception is propagated unchanged.

This method is called by the `__getitem__()` method of the `dict` class when the requested key is not found; whatever it returns or raises is then returned or raised by `__getitem__()`.

`defaultdict` objects support the following instance variable:

`default_factory`¶

This attribute is used by the `__missing__()` method; it is initialized from the first argument to the constructor, if present, or to `None`, if absent.

9.3.3.1. `defaultdict` Examples¶

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

When each key is encountered for the first time, it is not already in the mapping; so an entry is automatically created using the `default_factory` function which returns an empty `list`. The `list.append()` operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that key) and the `list.append()` operation adds another value to the list. This technique is simpler and faster than an equivalent technique using `dict.setdefault()`:

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> d.items()
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Setting the `default_factory` to `int` makes the `defaultdict` useful for counting (like a bag or multiset in other languages):

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> d.items()
[('i', 4), ('p', 2), ('s', 4), ('m', 1)]
```

When a letter is first encountered, it is missing from the mapping, so the `default_factory` function calls `int()` to supply a default count of zero. The increment operation then builds up the count for each letter.

The function `int()` which always returns zero is just a special case of constant functions. A faster and more flexible way to create constant functions is to use `itertools.repeat()` which can supply any constant value (not just zero):

```
>>> def constant_factory(value):
...     return itertools.repeat(value).next
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

Setting the `default_factory` to `set` makes the `defaultdict` useful for building a dictionary of sets:

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> d.items()
[('blue', set([2, 4])), ('red', set([1, 3]))]
```

9.3.4. `namedtuple()` Factory Function for Tuples with Named Fields¶

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.

`collections.namedtuple`(*typename*, *field_names*[, *verbose*])¶

Returns a new tuple subclass named *typename*. The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being indexable and iterable. Instances of the subclass also have a helpful docstring (with typename and field_names) and a helpful `__repr__()` method which lists the tuple contents in a `name=value` format.

The *field_names* are a single string with each fieldname separated by whitespace and/or commas, for example `'x y'` or `'x, y'`. Alternatively, *field_names* can be a sequence of strings such as `['x', 'y']`.

Any valid Python identifier may be used for a fieldname except for names starting with an underscore. Valid identifiers consist of letters, digits, and underscores but do not start with a digit or underscore and cannot be a `keyword` such as *class*, *for*, *return*, *global*, *pass*, *print*, or *raise*.

If *verbose* is true, the class definition is printed just before being built.

Named tuple instances do not have per-instance dictionaries, so they are lightweight and require no more memory than regular tuples.

New in version 2.6.

Example:

```
>>> Point = namedtuple('Point', 'x y', verbose=True)
class Point(tuple):
       'Point(x, y)'

       __slots__ = ()

       _fields = ('x', 'y')

       def __new__(_cls, x, y):
           return _tuple.__new__(_cls, (x, y))

       @classmethod
       def _make(cls, iterable, new=tuple.__new__, len=len):
           'Make a new Point object from a sequence or iterable'
           result = new(cls, iterable)
           if len(result) != 2:
               raise TypeError('Expected 2 arguments, got %d' % len(result))
           return result

       def __repr__(self):
           return 'Point(x=%r, y=%r)' % self

       def _asdict(t):
           'Return a new dict which maps field names to their values'
           return {'x': t[0], 'y': t[1]}

       def _replace(_self, **kwds):
           'Return a new Point object replacing specified fields with new values'
           result = _self._make(map(kwds.pop, ('x', 'y'), _self))
           if kwds:
               raise ValueError('Got unexpected field names: %r' % kwds.keys())
           return result

       def __getnewargs__(self):
           return tuple(self)

       x = _property(_itemgetter(0))
       y = _property(_itemgetter(1))

>>> 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)
```

Named tuples are especially useful for assigning field names to result tuples returned by the `csv` or `sqlite3` modules:

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
   print emp.name, emp.title

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
```

```
for emp in map(EmployeeRecord._make, cursor.fetchall()):
   print emp.name, emp.title
```

In addition to the methods inherited from tuples, named tuples support three additional methods and one attribute. To prevent conflicts with field names, the method and attribute names start with an underscore.

`somenamedtuple._make`(*iterable*)¶

Class method that makes a new instance from an existing sequence or iterable.

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

`somenamedtuple._asdict`()¶

Return a new dict which maps field names to their corresponding values:

```
>>> p._asdict()
{'x': 11, 'y': 22}
```

`somenamedtuple._replace`(*kwargs*)¶

Return a new instance of the named tuple replacing specified fields with new values:

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=newprices[partnum], timestamp=time.now())
```

`somenamedtuple._fields`¶

Tuple of strings listing the field names. Useful for introspection and for creating new named tuple types from existing named tuples.

```
>>> p._fields            # view the field names
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

To retrieve a field whose name is stored in a string, use the `getattr()` function:

```
>>> getattr(p, 'x')
11
```

To convert a dictionary to a named tuple, use the double-star-operator (as described in *Unpacking Argument Lists*):

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

Since a named tuple is a regular Python class, it is easy to add or change functionality with a subclass. Here is how to add a calculated field and a fixed-width print format:

```
>>> class Point(namedtuple('Point', 'x y')):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f' % (self.x, self.y, self.hypot)

>>> for p in Point(3, 4), Point(14, 5/7.):
...     print p
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

The subclass shown above sets `__slots__` to an empty tuple. This keeps keep memory requirements low by preventing the creation of instance dictionaries.

Subclassing is not useful for adding new, stored fields. Instead, simply create a new named tuple type from the `_fields` attribute:

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

Default values can be implemented by using `_replace()` to customize a prototype instance:

```
>>> Account = namedtuple('Account', 'owner balance transaction_count')
>>> default_account = Account('<owner name>', 0.0, 0)
>>> johns_account = default_account._replace(owner='John')
```

Enumerated constants can be implemented with named tuples, but it is simpler and more efficient to use a simple class declaration:

```
>>> Status = namedtuple('Status', 'open pending closed')._make(range(3))
>>> Status.open, Status.pending, Status.closed
(0, 1, 2)
>>> class Status:
...     open, pending, closed = range(3)
```

See also

Named tuple recipe adapted for Python 2.4.

Table Of Contents