

`calendar.timegm(tuple)`

An unrelated but handy function that takes a time tuple such as returned by the `gmtime()` function in the `time` module, and returns the corresponding Unix timestamp value, assuming an epoch of 1970, and the POSIX encoding. In fact, `time.gmtime()` and `timegm()` are each others' inverse.

The `calendar` module exports the following data attributes:

`calendar.day_name`

An array that represents the days of the week in the current locale.

`calendar.day_abbr`

An array that represents the abbreviated days of the week in the current locale.

`calendar.month_name`

An array that represents the months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and `month_name[0]` is the empty string.

`calendar.month_abbr`

An array that represents the abbreviated months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and `month_abbr[0]` is the empty string.

`calendar.MONDAY`

`calendar.TUESDAY`

`calendar.WEDNESDAY`

`calendar.THURSDAY`

`calendar.FRIDAY`

`calendar.SATURDAY`

`calendar.SUNDAY`

Aliases for day numbers, where MONDAY is 0 and SUNDAY is 6.

See also:

**Module `datetime`** Object-oriented interface to dates and times with similar functionality to the `time` module.

**Module `time`** Low-level time related functions.

## 8.4 collections — Container datatypes

**Source code:** `Lib/collections/__init__.py`

---

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

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

### 8.4.1 ChainMap objects

New in version 3.3.

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.

The class can be used to simulate nested scopes and is useful in templating.

**class** `collections.ChainMap(*maps)`

A *ChainMap* groups multiple dicts or other mappings together to create a single, updateable view. If no *maps* are specified, a single empty dictionary is provided so that a new chain always has at least one mapping.

The underlying mappings are stored in a list. That list is public and can be accessed or updated using the *maps* attribute. There is no other state.

Lookups search the underlying mappings successively until a key is found. In contrast, writes, updates, and deletions only operate on the first mapping.

A *ChainMap* incorporates the underlying mappings by reference. So, if one of the underlying mappings gets updated, those changes will be reflected in *ChainMap*.

All of the usual dictionary methods are supported. In addition, there is a *maps* attribute, a method for creating new subcontexts, and a property for accessing all but the first mapping:

#### **maps**

A user updateable list of mappings. The list is ordered from first-searched to last-searched. It is the only stored state and can be modified to change which mappings are searched. The list should always contain at least one mapping.

**new\_child** (*m=None*, *\*\*kwargs*)

Returns a new *ChainMap* containing a new map followed by all of the maps in the current instance. If *m* is specified, it becomes the new map at the front of the list of mappings; if not specified, an empty dict is used, so that a call to *d.new\_child()* is equivalent to: *ChainMap({}, \*d.maps)*. If any keyword arguments are specified, they update passed map or new empty dict. This method is used for creating subcontexts that can be updated without altering values in any of the parent mappings.

Changed in version 3.4: The optional *m* parameter was added.

Changed in version 3.10: Keyword arguments support was added.

#### **parents**

Property returning a new *ChainMap* containing all of the maps in the current instance except the first one. This is useful for skipping the first map in the search. Use cases are similar to those for the *nonlocal* keyword used in *nested scopes*. The use cases also parallel those for the built-in *super()* function. A reference to *d.parents* is equivalent to: *ChainMap(\*d.maps[1:])*.

Note, the iteration order of a *ChainMap()* is determined by scanning the mappings last to first:

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

This gives the same ordering as a series of *dict.update()* calls starting with the last mapping:

```
>>> combined = baseline.copy()
>>> combined.update(adjustments)
>>> list(combined)
['music', 'art', 'opera']
```

Changed in version 3.9: Added support for *|* and *|=* operators, specified in [PEP 584](#).

See also:

- The `MultiContext` class in the Enthought `CodeTools` package has options to support writing to any mapping in the chain.
- Django's `Context` class for templating is a read-only chain of mappings. It also features pushing and popping of contexts similar to the `new_child()` method and the `parents` property.
- The `Nested Contexts` recipe has options to control whether writes and other mutations apply only to the first mapping or to any mapping in the chain.
- A greatly simplified read-only version of `Chainmap`.

### ChainMap Examples and Recipes

This section shows various approaches to working with chained maps.

Example of simulating Python's internal lookup chain:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

Example of letting user specified command-line arguments take precedence over environment variables which in turn take precedence over default values:

```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k: v for k, v in vars(namespace).items() if v is not None}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

Example patterns for using the `ChainMap` class to simulate nested contexts:

```
c = ChainMap()           # Create root context
d = c.new_child()        # Create nested child context
e = c.new_child()        # Child of c, independent from d
e.maps[0]                # Current context dictionary -- like Python's locals()
e.maps[-1]               # Root context -- like Python's globals()
e.parents                # Enclosing context chain -- like Python's nonlocals

d['x'] = 1                # Set value in current context
d['x']                   # Get first key in the chain of contexts
del d['x']                # Delete from current context
list(d)                  # All nested values
k in d                   # Check all nested values
len(d)                   # Number of nested values
d.items()                # All nested items
dict(d)                  # Flatten into a regular dictionary
```

The `ChainMap` class only makes updates (writes and deletions) to the first mapping in the chain while lookups will search the full chain. However, if deep writes and deletions are desired, it is easy to make a subclass that updates keys found deeper in the chain:

```
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'
```

(continues on next page)

(continued from previous page)

```

def __setitem__(self, key, value):
    for mapping in self.maps:
        if key in mapping:
            mapping[key] = value
            return
    self.maps[0][key] = value

def __delitem__(self, key):
    for mapping in self.maps:
        if key in mapping:
            del mapping[key]
            return
    raise KeyError(key)

>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'}, {'lion': 'yellow'})
>>> d['lion'] = 'orange'           # update an existing key two levels down
>>> d['snake'] = 'red'             # new keys get added to the topmost dict
>>> del d['elephant']              # remove an existing key one level down
>>> d                             # display result
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange'})

```

## 8.4.2 Counter objects

A counter tool is provided to support convenient and rapid tallies. For example:

```

>>> # 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})

>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554), ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]

```

**class** collections.Counter ([*iterable-or-mapping*])

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.

Elements are counted from an *iterable* or initialized from another *mapping* (or counter):

```

>>> c = Counter()                # a new, empty counter
>>> c = Counter('gallahad')      # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2}) # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)   # a new counter from keyword args

```

Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a *KeyError*:

```

>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                    # count of a missing element is...
↪ zero
0

```

Setting a count to zero does not remove an element from a counter. Use `del` to remove it entirely:

```
>>> c['sausage'] = 0                                # counter entry with a zero count
>>> del c['sausage']                                # del actually removes the entry
```

New in version 3.1.

Changed in version 3.7: As a *dict* subclass, *Counter* inherited the capability to remember insertion order. Math operations on *Counter* objects also preserve order. Results are ordered according to when an element is first encountered in the left operand and then by the order encountered in the right operand.

Counter objects support additional methods beyond those available for all dictionaries:

#### **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']
```

#### **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)]
```

#### **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})
```

New in version 3.2.

#### **total()**

Compute the sum of the counts.

```
>>> c = Counter(a=10, b=5, c=0)
>>> c.total()
15
```

New in version 3.10.

The usual dictionary methods are available for *Counter* objects except for two which work differently for counters.

#### **fromkeys(iterable)**

This class method is not implemented for *Counter* objects.

#### **update([iterable-or-mapping])**

Elements are counted from an *iterable* or added-in from another *mapping* (or counter). Like *dict.update()* but adds counts instead of replacing them. Also, the *iterable* is expected to be a sequence of elements, not a sequence of (key, value) pairs.

Counters support rich comparison operators for equality, subset, and superset relationships: `==`, `!=`, `<`, `<=`, `>`, `>=`. All of those tests treat missing elements as having zero counts so that `Counter(a=1) == Counter(a=1, b=0)` returns true.

New in version 3.10: Rich comparison operations were added.

Changed in version 3.10: In equality tests, missing elements are treated as having zero counts. Formerly, `Counter(a=3)` and `Counter(a=3, b=0)` were considered distinct.

Common patterns for working with `Counter` objects:

```
c.total()           # total of all counts
c.clear()           # reset all counts
list(c)             # list unique elements
set(c)              # convert to a set
dict(c)             # convert to a regular dictionary
c.items()           # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs)) # convert from a list of (elem, cnt) pairs
c.most_common()[:n-1:-1] # n least common elements
+c                 # remove zero and negative counts
```

Several mathematical operations are provided for combining `Counter` objects to produce multisets (counters that have counts greater than zero). Addition and subtraction combine counters by adding or subtracting the counts of corresponding elements. Intersection and union return the minimum and maximum of corresponding counts. Equality and inclusion compare corresponding counts. Each operation can accept inputs with signed counts, but the output will exclude results with counts of zero or less.

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d           # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d           # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d           # intersection:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d           # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
>>> c == d         # equality:  c[x] == d[x]
False
>>> c <= d         # inclusion:  c[x] <= d[x]
False
```

Unary addition and subtraction are shortcuts for adding an empty counter or subtracting from an empty counter.

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

New in version 3.3: Added support for unary plus, unary minus, and in-place multiset operations.

**Note:** Counters were primarily designed to work with positive integers to represent running counts; however, care was taken to not unnecessarily preclude use cases needing other types or negative values. To help with those use cases, this section documents the minimum range and type restrictions.

- The `Counter` class itself is a dictionary subclass with no restrictions on its keys and values. The values are intended to be numbers representing counts, but you *could* store anything in the value field.
- The `most_common()` method requires only that the values be orderable.
- For in-place operations such as `c[key] += 1`, the value type need only support addition and subtraction. So fractions, floats, and decimals would work and negative values are supported. The same is also true for `update()` and `subtract()` which allow negative and zero values for both inputs and outputs.
- The multiset methods are designed only for use cases with positive values. The inputs may be negative or zero, but only outputs with positive values are created. There are no type restrictions, but the value type needs to support addition, subtraction, and comparison.

- The `elements()` method requires integer counts. It ignores zero and negative counts.
- 

**See also:**

- [Bag class](#) in Smalltalk.
- Wikipedia entry for [Multisets](#).
- [C++ multisets](#) tutorial with examples.
- For mathematical operations on multisets and their use cases, see *Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19*.
- To enumerate all distinct multisets of a given size over a given set of elements, see `itertools.combinations_with_replacement()`:

```
map(Counter, combinations_with_replacement('ABC', 2)) # --> AA AB AC BB BC CC
```

### 8.4.3 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.

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.

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.

**copy**()

Create a shallow copy of the deque.

New in version 3.5.

**count**(*x*)

Count the number of deque elements equal to *x*.

New in version 3.2.

**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.

**index** (*x* [, *start* [, *stop* ] ])

Return the position of *x* in the deque (at or after index *start* and before index *stop*). Returns the first match or raises *ValueError* if not found.

New in version 3.5.

**insert** (*i*, *x*)

Insert *x* into the deque at position *i*.

If the insertion would cause a bounded deque to grow beyond *maxlen*, an *IndexError* is raised.

New in version 3.5.

**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*)

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

**reverse** ()

Reverse the elements of the deque in-place and then return *None*.

New in version 3.2.

**rotate** (*n=1*)

Rotate the deque *n* steps to the right. If *n* is negative, rotate to the left.

When the deque is not empty, rotating one step to the right is equivalent to `d.appendleft(d.pop())`, and rotating one step to the left is equivalent to `d.append(d.popleft())`.

Deque objects also provide one read-only attribute:

**maxlen**

Maximum size of a deque or *None* if unbounded.

New in version 3.1.

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[0]` to access the first element. Indexed access is *O*(1) at both ends but slows to *O*(*n*) in the middle. For fast random access, use lists instead.

Starting in version 3.5, deques support `__add__()`, `__mul__()`, and `__imul__()`.

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
```

(continues on next page)



(continued from previous page)

```

>>> 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'])

```

## 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'
    with open(filename) as f:
        return deque(f, 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
    # https://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))

```

(continues on next page)

(continued from previous page)

```
d.appendleft(0)
s = sum(d)
for elem in it:
    s += elem - d.popleft()
    d.append(elem)
    yield s / n
```

A [round-robin scheduler](#) can be implemented with input iterators stored in a [deque](#). Values are yielded from the active iterator in position zero. If that iterator is exhausted, it can be removed with [popleft\(\)](#); otherwise, it can be cycled back to the end with the [rotate\(\)](#) method:

```
def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    iterators = deque(map(iter, iterables))
    while iterators:
        try:
            while True:
                yield next(iterators[0])
                iterators.rotate(-1)
        except StopIteration:
            # Remove an exhausted iterator.
            iterators.popleft()
```

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`.

### 8.4.4 defaultdict objects

**class** `collections.defaultdict` (*default\_factory=None*, /[, ...])

Return 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.

[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\\_\\_\(\)](#).

Note that `__missing__()` is *not* called for any operations besides `__getitem__()`. This means that `get()` will, like normal dictionaries, return `None` as a default rather than using `default_factory`.

`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.

Changed in version 3.9: Added merge (`|`) and update (`|=`) operators, specified in [PEP 584](#).

### **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)
...
>>> sorted(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)
...
>>> sorted(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
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

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 a lambda function which can supply any constant value (not just zero):

```
>>> def constant_factory(value):
...     return lambda: value
>>> 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)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```

## 8.4.5 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, *, rename=False, defaults=None, module=None)`

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

If *rename* is true, invalid fieldnames are automatically replaced with positional names. For example, `['abc', 'def', 'ghi', 'abc']` is converted to `['abc', '_1', 'ghi', '_3']`, eliminating the keyword *def* and the duplicate fieldname *abc*.

*defaults* can be `None` or an *iterable* of default values. Since fields with a default value must come after any fields without a default, the *defaults* are applied to the rightmost parameters. For example, if the fieldnames are `['x', 'y', 'z']` and the defaults are `(1, 2)`, then *x* will be a required argument, *y* will default to 1, and *z* will default to 2.

If *module* is defined, the `__module__` attribute of the named tuple is set to that value.

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

To support pickling, the named tuple class should be assigned to a variable that matches *typename*.

Changed in version 3.1: Added support for *rename*.

Changed in version 3.6: The *verbose* and *rename* parameters became *keyword-only arguments*.

Changed in version 3.6: Added the *module* parameter.

Changed in version 3.7: Removed the *verbose* parameter and the `__source__` attribute.

Changed in version 3.7: Added the *defaults* parameter and the `__field_defaults` attribute.

```
>>> # 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
```

(continues on next page)

(continued from previous page)

```
>>> 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 two attributes. To prevent conflicts with field names, the method and attribute names start with an underscore.

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

Changed in version 3.1: Returns an *OrderedDict* instead of a regular *dict*.

Changed in version 3.8: Returns a regular *dict* instead of an *OrderedDict*. As of Python 3.7, regular dicts are guaranteed to be ordered. If the extra features of *OrderedDict* are required, the suggested remediation is to cast the result to the desired type: `OrderedDict(nt._asdict())`.

`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')
```

(continues on next page)

(continued from previous page)

```
>>> 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)
```

`somenamedtuple._field_defaults`

Dictionary mapping field names to default values.

```
>>> Account = namedtuple('Account', ['type', 'balance'], defaults=[0])
>>> Account._field_defaults
{'balance': 0}
>>> Account('premium')
Account(type='premium', balance=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 [tut-unpacking-arguments](#)):

```
>>> 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 helps 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',))
```

Docstrings can be customized by making direct assignments to the `__doc__` fields:

```
>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
>>> Book.id.__doc__ = '13-digit ISBN'
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last name'
```

Changed in version 3.5: Property docstrings became writeable.

See also:

- See `typing.NamedTuple` for a way to add type hints for named tuples. It also provides an elegant notation using the `class` keyword:

```
class Component(NamedTuple):
    part_number: int
    weight: float
    description: Optional[str] = None
```

- See `types.SimpleNamespace()` for a mutable namespace based on an underlying dictionary instead of a tuple.
- The `dataclasses` module provides a decorator and functions for automatically adding generated special methods to user-defined classes.

### 8.4.6 OrderedDict objects

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).

Some differences from `dict` still remain:

- The regular `dict` was designed to be very good at mapping operations. Tracking insertion order was secondary.
- The `OrderedDict` was designed to be good at reordering operations. Space efficiency, iteration speed, and the performance of update operations were secondary.
- The `OrderedDict` algorithm can handle frequent reordering operations better than `dict`. As shown in the recipes below, this makes it suitable for implementing various kinds of LRU caches.
- The equality operation for `OrderedDict` checks for matching order.

A regular `dict` can emulate the order sensitive equality test with `p == q` and `all(k1 == k2 for k1, k2 in zip(p, q))`.

- The `popitem()` method of `OrderedDict` has a different signature. It accepts an optional argument to specify which item is popped.

A regular `dict` can emulate `OrderedDict`'s `od.popitem(last=True)` with `d.popitem()` which is guaranteed to pop the rightmost (last) item.

A regular `dict` can emulate `OrderedDict`'s `od.popitem(last=False)` with `(k := next(iter(d)), d.pop(k))` which will return and remove the leftmost (first) item if it exists.

- `OrderedDict` has a `move_to_end()` method to efficiently reposition an element to an endpoint.

A regular `dict` can emulate `OrderedDict`'s `od.move_to_end(k, last=True)` with `d[k] = d.pop(k)` which will move the key and its associated value to the rightmost (last) position.

A regular `dict` does not have an efficient equivalent for `OrderedDict`'s `od.move_to_end(k, last=False)` which moves the key and its associated value to the leftmost (first) position.

- Until Python 3.8, `dict` lacked a `__reversed__()` method.

**class** `collections.OrderedDict` (`[items]`)

Return an instance of a `dict` subclass that has methods specialized for rearranging dictionary order.

New in version 3.1.

**popitem** (`last=True`)

The `popitem()` method for ordered dictionaries returns and removes a (key, value) pair. The pairs are returned in LIFO order if `last` is true or FIFO (first-in, first-out) order if false.

**move\_to\_end** (*key*, *last=True*)

Move an existing *key* to either end of an ordered dictionary. The item is moved to the right end if *last* is true (the default) or to the beginning if *last* is false. Raises *KeyError* if the *key* does not exist:

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

New in version 3.2.

In addition to the usual mapping methods, ordered dictionaries also support reverse iteration using *reversed()*.

Equality tests between *OrderedDict* objects are order-sensitive and are implemented as `list(od1.items()) == list(od2.items())`. Equality tests between *OrderedDict* objects and other *Mapping* objects are order-insensitive like regular dictionaries. This allows *OrderedDict* objects to be substituted anywhere a regular dictionary is used.

Changed in version 3.5: The items, keys, and values *views* of *OrderedDict* now support reverse iteration using *reversed()*.

Changed in version 3.6: With the acceptance of **PEP 468**, order is retained for keyword arguments passed to the *OrderedDict* constructor and its *update()* method.

Changed in version 3.9: Added *merge()* and *update()* operators, specified in **PEP 584**.

## OrderedDict Examples and Recipes

It is straightforward to create an ordered dictionary variant that remembers the order the keys were *last* inserted. If a new entry overwrites an existing entry, the original insertion position is changed and moved to the end:

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        self.move_to_end(key)
```

An *OrderedDict* would also be useful for implementing variants of *functools.lru\_cache()*:

```
from time import time

class TimeBoundedLRU:
    "LRU Cache that invalidates and refreshes old entries."

    def __init__(self, func, maxsize=128, maxage=30):
        self.cache = OrderedDict()  # { args : (timestamp, result) }
        self.func = func
        self.maxsize = maxsize
        self.maxage = maxage

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            timestamp, result = self.cache[args]
            if time() - timestamp <= self.maxage:
                return result
        result = self.func(*args)
        self.cache[args] = time(), result
```

(continues on next page)



(continued from previous page)

```
if len(self.cache) > self.maxsize:
    self.cache.popitem(0)
return result
```

```
class MultiHitLRUCache:
    """ LRU cache that defers caching a result until
        it has been requested multiple times.

        To avoid flushing the LRU cache with one-time requests,
        we don't cache until a request has been made more than once.

    """

    def __init__(self, func, maxsize=128, maxrequests=4096, cache_after=1):
        self.requests = OrderedDict() # { uncached_key : request_count }
        self.cache = OrderedDict()    # { cached_key : function_result }
        self.func = func
        self.maxrequests = maxrequests # max number of uncached requests
        self.maxsize = maxsize         # max number of stored return values
        self.cache_after = cache_after

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            return self.cache[args]
        result = self.func(*args)
        self.requests[args] = self.requests.get(args, 0) + 1
        if self.requests[args] <= self.cache_after:
            self.requests.move_to_end(args)
            if len(self.requests) > self.maxrequests:
                self.requests.popitem(0)
        else:
            self.requests.pop(args, None)
            self.cache[args] = result
            if len(self.cache) > self.maxsize:
                self.cache.popitem(0)
        return result
```

## 8.4.7 UserDict objects

The class, *UserDict* acts as a wrapper around dictionary objects. The need for this class has been partially supplanted by the ability to subclass directly from *dict*; however, this class can be easier to work with because the underlying dictionary is accessible as an attribute.

**class** `collections.UserDict` (`[initialdata]`)

Class that simulates a dictionary. The instance's contents are kept in a regular dictionary, which is accessible via the *data* attribute of *UserDict* instances. If *initialdata* is provided, *data* is initialized with its contents; note that a reference to *initialdata* will not be kept, allowing it to be used for other purposes.

In addition to supporting the methods and operations of mappings, *UserDict* instances provide the following attribute:

**data**

A real dictionary used to store the contents of the *UserDict* class.

### 8.4.8 `UserList` objects

This class acts as a wrapper around list objects. It is a useful base class for your own list-like classes which can inherit from them and override existing methods or add new ones. In this way, one can add new behaviors to lists.

The need for this class has been partially supplanted by the ability to subclass directly from `list`; however, this class can be easier to work with because the underlying list is accessible as an attribute.

**class** `collections.UserList` (`[list]`)

Class that simulates a list. The instance's contents are kept in a regular list, which is accessible via the `data` attribute of `UserList` instances. The instance's contents are initially set to a copy of `list`, defaulting to the empty list `[]`. `list` can be any iterable, for example a real Python list or a `UserList` object.

In addition to supporting the methods and operations of mutable sequences, `UserList` instances provide the following attribute:

**data**

A real `list` object used to store the contents of the `UserList` class.

**Subclassing requirements:** Subclasses of `UserList` are expected to offer a constructor which can be called with either no arguments or one argument. List operations which return a new sequence attempt to create an instance of the actual implementation class. To do so, it assumes that the constructor can be called with a single parameter, which is a sequence object used as a data source.

If a derived class does not wish to comply with this requirement, all of the special methods supported by this class will need to be overridden; please consult the sources for information about the methods which need to be provided in that case.

### 8.4.9 `UserString` objects

The class, `UserString` acts as a wrapper around string objects. The need for this class has been partially supplanted by the ability to subclass directly from `str`; however, this class can be easier to work with because the underlying string is accessible as an attribute.

**class** `collections.UserString` (`seq`)

Class that simulates a string object. The instance's content is kept in a regular string object, which is accessible via the `data` attribute of `UserString` instances. The instance's contents are initially set to a copy of `seq`. The `seq` argument can be any object which can be converted into a string using the built-in `str()` function.

In addition to supporting the methods and operations of strings, `UserString` instances provide the following attribute:

**data**

A real `str` object used to store the contents of the `UserString` class.

Changed in version 3.5: New methods `__getnewargs__`, `__rmod__`, `casefold`, `format_map`, `isprintable`, and `maketrans`.

## 8.5 `collections.abc` — Abstract Base Classes for Containers

New in version 3.3: Formerly, this module was part of the `collections` module.

**Source code:** `Lib/_collections_abc.py`

This module provides *abstract base classes* that can be used to test whether a class provides a particular interface; for example, whether it is *hashable* or whether it is a mapping.

An `issubclass()` or `isinstance()` test for an interface works in one of three ways.

1) A newly written class can inherit directly from one of the abstract base classes. The class must supply the required abstract methods. The remaining mixin methods come from inheritance and can be overridden if desired. Other methods may be added as needed:

```
class C(Sequence):
    def __init__(self): ...
    def __getitem__(self, index): ...
    def __len__(self): ...
    def count(self, value): ...
```

*# Direct inheritance*  
*# Extra method not required by the ABC*  
*# Required abstract method*  
*# Required abstract method*  
*# Optionally override a mixin method*

```
>>> issubclass(C, Sequence)
True
>>> isinstance(C(), Sequence)
True
```

2) Existing classes and built-in classes can be registered as “virtual subclasses” of the ABCs. Those classes should define the full API including all of the abstract methods and all of the mixin methods. This lets users rely on `issubclass()` or `isinstance()` tests to determine whether the full interface is supported. The exception to this rule is for methods that are automatically inferred from the rest of the API:

```
class D:
    def __init__(self): ...
    def __getitem__(self, index): ...
    def __len__(self): ...
    def count(self, value): ...
    def index(self, value): ...
```

*# No inheritance*  
*# Extra method not required by the ABC*  
*# Abstract method*  
*# Abstract method*  
*# Mixin method*  
*# Mixin method*

```
Sequence.register(D)
```

*# Register instead of inherit*

```
>>> issubclass(D, Sequence)
True
>>> isinstance(D(), Sequence)
True
```

In this example, class `D` does not need to define `__contains__`, `__iter__`, and `__reversed__` because the in-operator, the `iteration` logic, and the `reversed()` function automatically fall back to using `__getitem__` and `__len__`.

3) Some simple interfaces are directly recognizable by the presence of the required methods (unless those methods have been set to `None`):

```
class E:
    def __iter__(self): ...
    def __next__(next): ...
```

```
>>> issubclass(E, Iterable)
True
>>> isinstance(E(), Iterable)
True
```

Complex interfaces do not support this last technique because an interface is more than just the presence of method names. Interfaces specify semantics and relationships between methods that cannot be inferred solely from the presence of specific method names. For example, knowing that a class supplies `__getitem__`, `__len__`, and `__iter__` is insufficient for distinguishing a *Sequence* from a *Mapping*.

New in version 3.9: These abstract classes now support `[]`. See *Generic Alias Type* and [PEP 585](#).

## 8.5.1 Collections Abstract Base Classes

The collections module offers the following *ABCs*:

ABC	Inherits from	Abstract Methods	Mixin Methods
<i>Container</i> <sup>1</sup>		<code>__contains__</code>	
<i>Hashable</i> <sup>1</sup>		<code>__hash__</code>	
<i>Iterable</i> <sup>1,2</sup>		<code>__iter__</code>	
<i>Iterator</i> <sup>1</sup>	<i>Iterable</i>	<code>__next__</code>	<code>__iter__</code>
<i>Reversible</i> <sup>1</sup>	<i>Iterable</i>	<code>__reversed__</code>	
<i>Generator</i> <sup>1</sup>	<i>Iterator</i>	<code>send, throw</code>	<code>close, __iter__, __next__</code>
<i>Sized</i> <sup>1</sup>		<code>__len__</code>	
<i>Callable</i> <sup>1</sup>		<code>__call__</code>	
<i>Collection</i> <sup>1</sup>	<i>Sized</i> , <i>Iterable</i> , <i>Container</i>	<code>__contains__</code> , <code>__iter__</code> , <code>__len__</code>	
<i>Sequence</i>	<i>Reversible</i> , <i>Collection</i>	<code>__getitem__</code> , <code>__len__</code>	<code>__contains__</code> , <code>__iter__</code> , <code>__reversed__</code> , <code>index</code> , and <code>count</code>
<i>MutableSequence</i>	<i>Sequence</i>	<code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__len__</code> , <code>insert</code>	Inherited <i>Sequence</i> methods and <code>append</code> , <code>reverse</code> , <code>extend</code> , <code>pop</code> , <code>remove</code> , and <code>__iadd__</code>
<i>ByteString</i>	<i>Sequence</i>	<code>__getitem__</code> , <code>__len__</code>	Inherited <i>Sequence</i> methods
<i>Set</i>	<i>Collection</i>	<code>__contains__</code> , <code>__iter__</code> , <code>__len__</code>	<code>__le__</code> , <code>__lt__</code> , <code>__eq__</code> , <code>__ne__</code> , <code>__gt__</code> , <code>__ge__</code> , <code>__and__</code> , <code>__or__</code> , <code>__sub__</code> , <code>__xor__</code> , and <code>isdisjoint</code>
<i>MutableSet</i>	<i>Set</i>	<code>__contains__</code> , <code>__iter__</code> , <code>__len__</code> , <code>add</code> , <code>discard</code>	Inherited <i>Set</i> methods and <code>clear</code> , <code>pop</code> , <code>remove</code> , <code>__ior__</code> , <code>__iand__</code> , <code>__ixor__</code> , and <code>__isub__</code>
<i>Mapping</i>	<i>Collection</i>	<code>__getitem__</code> , <code>__iter__</code> , <code>__len__</code>	<code>__contains__</code> , <code>keys</code> , <code>items</code> , <code>values</code> , <code>get</code> , <code>__eq__</code> , and <code>__ne__</code>
<i>MutableMapping</i>	<i>Mapping</i>	<code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__iter__</code> , <code>__len__</code>	Inherited <i>Mapping</i> methods and <code>pop</code> , <code>popitem</code> , <code>clear</code> , <code>update</code> , and <code>setdefault</code>
<i>MappingView</i>	<i>Sized</i>		<code>__len__</code>
<i>ItemsView</i>	<i>MappingView</i> , <i>Set</i>		<code>__contains__</code> , <code>__iter__</code>
<i>KeysView</i>	<i>MappingView</i> , <i>Set</i>		<code>__contains__</code> , <code>__iter__</code>
<i>ValuesView</i>	<i>MappingView</i> , <i>Collection</i>		<code>__contains__</code> , <code>__iter__</code>
<i>Awaitable</i> <sup>1</sup>		<code>__await__</code>	
<i>Coroutine</i> <sup>1</sup>	<i>Awaitable</i>	<code>send, throw</code>	<code>close</code>
<i>AsyncIterable</i> <sup>1</sup>		<code>__aiter__</code>	
<i>AsyncIterator</i> <sup>1</sup>	<i>AsyncIterable</i>	<code>__anext__</code>	<code>__aiter__</code>
<i>AsyncGenerator</i> <sup>1</sup>	<i>AsyncIterator</i>	<code>send, athrow</code>	<code>aclose</code> , <code>__aiter__</code> , <code>__anext__</code>

<sup>1</sup> These ABCs override `object.__subclasshook__()` to support testing an interface by verifying the required methods are present and have not been set to `None`. This only works for simple interfaces. More complex interfaces require registration or direct subclassing.

<sup>2</sup> Checking `isinstance(obj, Iterable)` detects classes that are registered as *Iterable* or that have an `__iter__()` method, but it does not detect classes that iterate with the `__getitem__()` method. The only reliable way to determine whether an object is *iterable* is to call `iter(obj)`.

## 8.5.2 Collections Abstract Base Classes – Detailed Descriptions

**class** `collections.abc.Container`

ABC for classes that provide the `__contains__()` method.

**class** `collections.abc.Hashable`

ABC for classes that provide the `__hash__()` method.

**class** `collections.abc.Sized`

ABC for classes that provide the `__len__()` method.

**class** `collections.abc.Callable`

ABC for classes that provide the `__call__()` method.

**class** `collections.abc.Iterable`

ABC for classes that provide the `__iter__()` method.

Checking `isinstance(obj, Iterable)` detects classes that are registered as *Iterable* or that have an `__iter__()` method, but it does not detect classes that iterate with the `__getitem__()` method. The only reliable way to determine whether an object is *iterable* is to call `iter(obj)`.

**class** `collections.abc.Collection`

ABC for sized iterable container classes.

New in version 3.6.

**class** `collections.abc.Iterator`

ABC for classes that provide the `__iter__()` and `__next__()` methods. See also the definition of *iterator*.

**class** `collections.abc.Reversible`

ABC for iterable classes that also provide the `__reversed__()` method.

New in version 3.6.

**class** `collections.abc.Generator`

ABC for generator classes that implement the protocol defined in **PEP 342** that extends iterators with the `send()`, `throw()` and `close()` methods. See also the definition of *generator*.

New in version 3.5.

**class** `collections.abc.Sequence`

**class** `collections.abc.MutableSequence`

**class** `collections.abc.ByteString`

ABCs for read-only and mutable *sequences*.

Implementation note: Some of the mixin methods, such as `__iter__()`, `__reversed__()` and `index()`, make repeated calls to the underlying `__getitem__()` method. Consequently, if `__getitem__()` is implemented with constant access speed, the mixin methods will have linear performance; however, if the underlying method is linear (as it would be with a linked list), the mixins will have quadratic performance and will likely need to be overridden.

Changed in version 3.5: The `index()` method added support for *stop* and *start* arguments.

**class** `collections.abc.Set`

**class** `collections.abc.MutableSet`

ABCs for read-only and mutable sets.

**class** `collections.abc.Mapping`

**class** `collections.abc.MutableMapping`

ABCs for read-only and mutable *mappings*.

**class** `collections.abc.MappingView`

---

**class** `collections.abc.ItemsView`

**class** `collections.abc.KeysView`

**class** `collections.abc.ValuesView`

ABCs for mapping, items, keys, and values *views*.

**class** `collections.abc.Awaitable`

ABC for *awaitable* objects, which can be used in `await` expressions. Custom implementations must provide the `__await__()` method.

*Coroutine* objects and instances of the *Coroutine* ABC are all instances of this ABC.

---

**Note:** In CPython, generator-based coroutines (generators decorated with `types.coroutine()`) are *awaitables*, even though they do not have an `__await__()` method. Using `isinstance(gencoro, Awaitable)` for them will return `False`. Use `inspect.isawaitable()` to detect them.

---

New in version 3.5.

**class** `collections.abc.Coroutine`

ABC for coroutine compatible classes. These implement the following methods, defined in coroutine-objects: `send()`, `throw()`, and `close()`. Custom implementations must also implement `__await__()`. All *Coroutine* instances are also instances of *Awaitable*. See also the definition of *coroutine*.

---

**Note:** In CPython, generator-based coroutines (generators decorated with `types.coroutine()`) are *awaitables*, even though they do not have an `__await__()` method. Using `isinstance(gencoro, Coroutine)` for them will return `False`. Use `inspect.isawaitable()` to detect them.

---

New in version 3.5.

**class** `collections.abc.AsyncIterable`

ABC for classes that provide `__aiter__` method. See also the definition of *asynchronous iterable*.

New in version 3.5.

**class** `collections.abc.AsyncIterator`

ABC for classes that provide `__aiter__` and `__anext__` methods. See also the definition of *asynchronous iterator*.

New in version 3.5.

**class** `collections.abc.AsyncGenerator`

ABC for asynchronous generator classes that implement the protocol defined in [PEP 525](#) and [PEP 492](#).

New in version 3.6.

### 8.5.3 Examples and Recipes

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

```
size = None
if isinstance(myvar, collections.abc.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 is 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.abc.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 or regular method 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 `__ge__()`, 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 hashability 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](#).

## 8.6 heapq — Heap queue algorithm

Source code: [Lib/heapq.py](#)

---

This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

Heaps are binary trees for which every parent node has a value less than or equal to any of its children. This implementation uses arrays for which `heap[k] <= heap[2*k+1]` and `heap[k] <= heap[2*k+2]` for all *k*, counting elements from zero. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that its smallest element is always the root, `heap[0]`.

The API below differs from textbook heap algorithms in two aspects: (a) We use zero-based indexing. This makes the relationship between the index for a node and the indexes for its children slightly less obvious, but is more suitable since Python uses zero-based indexing. (b) Our `pop` method returns the smallest item, not the largest (called a “min heap” in textbooks; a “max heap” is more common in texts because of its suitability for in-place sorting).