

- 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
```

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
>>> 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
    # 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 / 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`.

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: