- 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
```

8.3.3. 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[-1]`. 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 item
>>> for elem in d:                   # iterate over the deque's element
...     print(elem.upper())
G
H
I

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

>>> d.pop()                          # return and remove the rightmost
'j'
>>> d.popleft()                      # return and remove the leftmost i
'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
['i', 'h', 'g']
>>> 'h' in d                         # search the deque
```