multiset Documentation

Release 2.0.1.dev0+ngbf561d6.d20170208

Manuel Krebber

Feb 08, 2017

Contents

CHAPTER 1

API Documentation

This page is automatically generated from the docstrings. An implementation of a multiset.

class `multiset.`**`BaseMultiset`**(*self*, *iterable=None*)

A multiset implementation.

A multiset is similar to the builtin `set`, but elements can occur multiple times in the multiset. It is also similar to a `list` without ordering of the values and hence no index-based operations.

The multiset internally uses a `dict` for storage where the key is the element and the value its multiplicity. It supports all operations that the `set` supports.

In contrast to the builtin `collections.Counter`, no negative counts are allowed, elements with zero counts are removed from the `dict`, and set operations are supported.

The multiset comes in two variants, *`Multiset`* and *`FrozenMultiset`* which correspond to the `set` and `frozenset` classes, respectively.

> **Warning:** You cannot instantiate this class directly. Use one of its variants instead.

See https://en.wikipedia.org/wiki/Multiset

`__init__`(*self*, *iterable=None*)

Create a new, empty Multiset object.

And if given, initialize with elements from input iterable. Or, initialize from a mapping of elements to their multiplicity.

Example:

```
>>> ms = Multiset()                  # a new, empty multiset
>>> ms = Multiset('abc')             # a new multiset from an iterable
>>> ms = Multiset({'a': 4, 'b': 2}) # a new multiset from a mapping
```

Parameters **iterable** – An optional iterable of elements or mapping of elements to multiplicity to initialize the multiset from.

combine(*self*, **others*)

Return a new multiset with all elements from the multiset and the others with their multiplicities summed up.

```
>>> ms = Multiset('aab')
>>> sorted(ms.combine('bc'))
['a', 'a', 'b', 'b', 'c']
```

You can also use the + operator for the same effect. However, the operator version will only accept a set as other operator, not any iterable, to avoid errors.

```
>>> ms = Multiset('aab')
>>> sorted(ms + Multiset('a'))
['a', 'a', 'a', 'b']
```

For a variant of the operation which modifies the multiset in place see `update()`.

Parameters **others** – The other sets to add to the multiset. Can also be any `Iterable`[~T] or `Mapping`[~T, `int`] which are then converted to *`Multiset`*[~T].

Returns The multiset resulting from the addition of the sets.

copy(*self*)

Return a shallow copy of the multiset.

difference(*self*, **others*)

Return a new multiset with all elements from the others removed.

```
>>> ms = Multiset('aab')
>>> sorted(ms.difference('bc'))
['a', 'a']
```

You can also use the − operator for the same effect. However, the operator version will only accept a set as other operator, not any iterable, to avoid errors.

```
>>> ms = Multiset('aabbbc')
>>> sorted(ms - Multiset('abd'))
['a', 'b', 'b', 'c']
```

For a variant of the operation which modifies the multiset in place see `difference_update()`.

Parameters **others** – The other sets to remove from the multiset. Can also be any `Iterable`[~T] or `Mapping`[~T, `int`] which are then converted to *`Multiset`*[~T].

Returns The resulting difference multiset.

distinct_elements(*self*)

classmethod **from_elements**(*elements*, *multiplicity*)

Create a new multiset with the given *elements* and each multiplicity set to *multiplicity*.

Uses `dict.fromkeys()` internally.

Parameters

- **elements** – The element for the new multiset.
- **multiplicity** – The multiplicity for all elements.

Returns The new multiset.

get (*self*, *element*, *default*)
Return the multiplicity for *element* if it is in the multiset, else *default*.

Makes the *default* argument of the original `dict.get()` non-optional.

Parameters

- **element** – The element of which to get the multiplicity.
- **default** – The default value to return if the element if not in the multiset.

Returns The multiplicity for *element* if it is in the multiset, else *default*.

intersection (*self*, **others*)
Return a new multiset with elements common to the multiset and all others.

```
>>> ms = Multiset('aab')
>>> sorted(ms.intersection('abc'))
['a', 'b']
```

You can also use the `&` operator for the same effect. However, the operator version will only accept a set as other operator, not any iterable, to avoid errors.

```
>>> ms = Multiset('aab')
>>> sorted(ms & Multiset('aaac'))
['a', 'a']
```

For a variant of the operation which modifies the multiset in place see `intersection_update()`.

Parameters **others** – The other sets intersect with the multiset. Can also be any `Iterable`[~T] or `Mapping`[~T, `int`] which are then converted to *`Multiset`*[~T].

Returns The multiset resulting from the intersection of the sets.

isdisjoint (*self*, *other*)
Return True if the set has no elements in common with other.

Sets are disjoint iff their intersection is the empty set.

```
>>> ms = Multiset('aab')
>>> ms.isdisjoint('bc')
False
>>> ms.isdisjoint(Multiset('ccd'))
True
```

Parameters **other** – The other set to check disjointedness. Can also be an `Iterable`[~T] or `Mapping`[~T, `int`] which are then converted to *`Multiset`*[~T].

issubset (*self*, *other*)
Return True iff this set is a subset of the other.

```
>>> Multiset('ab').issubset('aabc')
True
>>> Multiset('aabb').issubset(Multiset('aabc'))
False
```

You can also use the <= operator for this comparison:

```
>>> Multiset('ab') <= Multiset('ab')
True
```

When using the < operator for comparison, the sets are checked to be unequal in addition:

```
>>> Multiset('ab') < Multiset('ab')
False
```

Parameters **other** – The potential superset of the multiset to be checked.

Returns True iff this set is a subset of the other.

issuperset(*self*, *other*)

Return True iff this multiset is a superset of the other.

```
>>> Multiset('aabc').issuperset('ab')
True
>>> Multiset('aabc').issuperset(Multiset('abcc'))
False
```

You can also use the >= operator for this comparison:

```
>>> Multiset('ab') >= Multiset('ab')
True
```

When using the > operator for comparison, the sets are checked to be unequal in addition:

```
>>> Multiset('ab') > Multiset('ab')
False
```

Parameters **other** – The potential subset of the multiset to be checked.

Returns True iff this set is a subset of the other.

items(*self*)

multiplicities(*self*)

symmetric_difference(*self*, *other*)

Return a new set with elements in either the set or other but not both.

```
>>> ms = Multiset('aab')
>>> sorted(ms.symmetric_difference('abc'))
['a', 'c']
```

You can also use the ^ operator for the same effect. However, the operator version will only accept a set as other operator, not any iterable, to avoid errors.

```
>>> ms = Multiset('aab')
>>> sorted(ms ^ Multiset('aaac'))
['a', 'b', 'c']
```

For a variant of the operation which modifies the multiset in place see `symmetric_difference_update()`.

Parameters **other** – The other set to take the symmetric difference with. Can also be any `Iterable`[~T] or `Mapping`[~T, `int`] which are then converted to *`Multiset`*[~T].

Returns The resulting symmetric difference multiset.

times(*self*, *factor*)

Return a new set with each element's multiplicity multiplied with the given scalar factor.

```
>>> ms = Multiset('aab')
>>> sorted(ms.times(2))
['a', 'a', 'a', 'a', 'b', 'b']
```

You can also use the `*` operator for the same effect:

```
>>> sorted(ms * 3)
['a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b']
```

For a variant of the operation which modifies the multiset in place see `times_update()`.

Parameters **factor** – The factor to multiply each multiplicity with.

union(*self*, **others*)

Return a new multiset with all elements from the multiset and the others with maximal multiplicities.

```
>>> ms = Multiset('aab')
>>> sorted(ms.union('bc'))
['a', 'a', 'b', 'c']
```

You can also use the `|` operator for the same effect. However, the operator version will only accept a set as other operator, not any iterable, to avoid errors.

```
>>> ms = Multiset('aab')
>>> sorted(ms | Multiset('aaa'))
['a', 'a', 'a', 'b']
```

For a variant of the operation which modifies the multiset in place see *union()*.

Parameters ***others** – The other sets to union the multiset with. Can also be any `Iterable`[~T] or `Mapping`[~T, `int`] which are then converted to *Multiset*[~T].

Returns The multiset resulting from the union.

class multiset.**FrozenMultiset**(*self*, *iterable=None*)

Bases: *BaseMultiset*

The frozen multiset variant that is immutable and hashable.

class multiset.**Multiset**(*self*, *iterable=None*)

Bases: *BaseMultiset*

The mutable multiset variant.

add(*self*, *element*, *multiplicity=1*)

Adds an element to the multiset.

```
>>> ms = Multiset()
>>> ms.add('a')
>>> sorted(ms)
['a']
```

An optional multiplicity can be specified to define how many of the element are added:

```
>>> ms.add('b', 2)
>>> sorted(ms)
['a', 'b', 'b']
```

This extends the `MutableSet.add()` signature to allow specifying the multiplicity.

Parameters

- **element** – The element to add to the multiset.
- **multiplicity** – The multiplicity i.e. count of elements to add.

clear(*self*)
Empty the multiset.

difference_update(*self*, **others*)
Remove all elements contained the others from this multiset.

```
>>> ms = Multiset('aab')
>>> ms.difference_update('abc')
>>> sorted(ms)
['a']
```

You can also use the -= operator for the same effect. However, the operator version will only accept a set as other operator, not any iterable, to avoid errors.

```
>>> ms = Multiset('aabbbc')
>>> ms -= Multiset('abd')
>>> sorted(ms)
['a', 'b', 'b', 'c']
```

For a variant of the operation which does not modify the multiset, but returns a new multiset instead see *difference()*.

Parameters **others** – The other sets to remove from this multiset. Can also be any `Iterable`[~T] or `Mapping`[~T, `int`] which are then converted to *Multiset*[~T].

discard(*self*, *element*, *multiplicity=None*)
Removes the `element` from the multiset.

If multiplicity is `None`, all occurrences of the element are removed:

```
>>> ms = Multiset('aab')
>>> ms.discard('a')
2
>>> sorted(ms)
['b']
```

Otherwise, the multiplicity is subtracted from the one in the multiset and the old multiplicity is removed:

```
>>> ms = Multiset('aab')
>>> ms.discard('a', 1)
2
>>> sorted(ms)
['a', 'b']
```

In contrast to *remove()*, this does not raise an error if the element is not in the multiset:

```
>>> ms = Multiset('a')
>>> ms.discard('b')
0
>>> sorted(ms)
['a']
```

It is also not an error to remove more elements than are in the set:

```
>>> ms.remove('a', 2)
1
>>> sorted(ms)
[]
```

Parameters

- **element** – The element to remove from the multiset.
- **multiplicity** – An optional multiplicity i.e. count of elements to remove.

Returns The multiplicity of the element in the multiset before the removal.

intersection_update(*self*, **others*)

Update the multiset, keeping only elements found in it and all others.

```
>>> ms = Multiset('aab')
>>> ms.intersection_update('bc')
>>> sorted(ms)
['b']
```

You can also use the `&=` operator for the same effect. However, the operator version will only accept a set as other operator, not any iterable, to avoid errors.

```
>>> ms = Multiset('aabc')
>>> ms &= Multiset('abbd')
>>> sorted(ms)
['a', 'b']
```

For a variant of the operation which does not modify the multiset, but returns a new multiset instead see *intersection()*.

Parameters **others** – The other sets to intersect this multiset with. Can also be any `Iterable`[~T] or `Mapping`[~T, `int`] which are then converted to *Multiset*[~T].

pop(*self*, *element*, *default*)

If *element* is in the multiset, remove it and return its multiplicity, else return *default*.

Makes the *default* argument of the original `dict.pop()` non-optional.

Parameters

- **element** – The element which is removed.
- **default** – The default value to return if the element if not in the multiset.

Returns The multiplicity for *element* if it is in the multiset, else *default*.

remove(*self*, *element*, *multiplicity=None*)

Removes an element from the multiset.

If no multiplicity is specified, the element is completely removed from the multiset:

```
>>> ms = Multiset('aabbbc')
>>> ms.remove('a')
2
>>> sorted(ms)
['b', 'b', 'b', 'c']
```

If the multiplicity is given, it is subtracted from the element's multiplicity in the multiset:

```
>>> ms.remove('b', 2)
3
>>> sorted(ms)
['b', 'c']
```

It is not an error to remove more elements than are in the set:

```
>>> ms.remove('b', 2)
1
>>> sorted(ms)
['c']
```

This extends the `MutableSet.remove()` signature to allow specifying the multiplicity.

Parameters

- **element** – The element to remove from the multiset.
- **multiplicity** – An optional multiplicity i.e. count of elements to remove.

Returns The multiplicity of the element in the multiset before the removal.

Raises `KeyError` – if the element is not contained in the set. Use *`discard()`* if you do not want an exception to be raised.

setdefault(*self*, *element*, *default*)

If *element* is in the multiset, return its multiplicity. Else add it with a multiplicity of *default* and return *default*.

Makes the *default* argument of the original `dict.setdefault()` non-optional.

Parameters

- **element** – The element which is added if not already present.
- **default** – The default multiplicity to add the element with if not in the multiset.

Returns The multiplicity for *element* if it is in the multiset, else *default*.

symmetric_difference_update(*self*, *other*)

Update the multiset to contain only elements in either this multiset or the other but not both.

```
>>> ms = Multiset('aab')
>>> ms.symmetric_difference_update('abc')
>>> sorted(ms)
['a', 'c']
```

You can also use the ^= operator for the same effect. However, the operator version will only accept a set as other operator, not any iterable, to avoid errors.

```
>>> ms = Multiset('aabbbc')
>>> ms ^= Multiset('abd')
>>> sorted(ms)
['a', 'b', 'b', 'c', 'd']
```

For a variant of the operation which does not modify the multiset, but returns a new multiset instead see *`symmetric_difference()`*.

Parameters **other** – The other set to take the symmetric difference with. Can also be any `Iterable`[~T] or `Mapping`[~T, `int`] which are then converted to *`Multiset`*[~T].

times_update(*self*, *factor*)

Update each this multiset by multiplying each element's multiplicity with the given scalar factor.

```
>>> ms = Multiset('aab')
>>> ms.times_update(2)
>>> sorted(ms)
['a', 'a', 'a', 'a', 'b', 'b']
```

You can also use the *= operator for the same effect:

```
>>> ms = Multiset('ac')
>>> ms *= 3
>>> sorted(ms)
['a', 'a', 'a', 'c', 'c', 'c']
```

For a variant of the operation which does not modify the multiset, but returns a new multiset instead see *times()*.

Parameters **factor** – The factor to multiply each multiplicity with.

union_update(*self*, **others*)

Update the multiset, adding elements from all others using the maximum multiplicity.

```
>>> ms = Multiset('aab')
>>> ms.union_update('bc')
>>> sorted(ms)
['a', 'a', 'b', 'c']
```

You can also use the |= operator for the same effect. However, the operator version will only accept a set as other operator, not any iterable, to avoid errors.

```
>>> ms = Multiset('aab')
>>> ms |= Multiset('bccd')
>>> sorted(ms)
['a', 'a', 'b', 'c', 'c', 'd']
```

For a variant of the operation which does not modify the multiset, but returns a new multiset instead see *union()*.

Parameters **others** – The other sets to union this multiset with. Can also be any Iterable[~T] or Mapping[~T, int] which are then converted to *Multiset*[~T].

update(*self*, **others*)

Like dict.update() but add multiplicities instead of replacing them.

```
>>> ms = Multiset('aab')
>>> ms.update('abc')
>>> sorted(ms)
['a', 'a', 'a', 'b', 'b', 'c']
```

Note that the operator += is equivalent to *update()*, except that the operator will only accept sets to avoid accidental errors.

```
>>> ms += Multiset('bc')
>>> sorted(ms)
['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c']
```

For a variant of the operation which does not modify the multiset, but returns a new multiset instead see *combine()*.

Parameters **others** – The other sets to add to this multiset. Can also be any Iterable[~T] or Mapping[~T, int] which are then converted to *Multiset*[~T].

This package provides a multiset implementation for Python.

A multiset is similar to the builtin set, but it allows an element to occur multiple times. It is an unordered collection of element which have to be hashable just like in a set. It supports the same *methods and operations* as set does, e.g. membership test, union, intersection, and (symmetric) difference. Multisets can be used in combination with regular sets for those operations.

The implementation is based on a dict that maps the elements to their multiplicity in the multiset. Hence, some *dictionary operations* are supported.

In contrast to the collections.Counter from the standard library, it has proper support for set operations and only allows positive counts. Also, elements with a zero multiplicity are automatically removed from the multiset.

There is an immutable version of the multiset called *FrozenMultiset*.

The package also uses the typing module for type hints (see **PEP 484**) so you can specify the type of a multiset like Multiset[ElementType].

CHAPTER 2

API Overview

The following is an overview over the methods of the *Multiset* class. For more details on each method and some examples see the autogenerated *API Documentation*.

class Multiset([*mapping or iterable*])

Return a new multiset object whose elements are taken from the given optional iterable or mapping. If a mapping is given, it must have positive `int` values representing each element's multiplicity. If no iterable or mapping is specified, a new empty multiset is returned.

The elements of a set must be hashable. In contrast to regular sets, duplicate elements will not be removed from the multiset. The *Multiset* class provides the following operations which the builtin `set` also supports:

len(s)

Return the total number of elements in multiset *s* (cardinality of *s*).

Note that this is the sum of the multiplicities and not not the number of distinct elements. You can use *distinct_elements()* to get the number of distinct elements:

```
>>> len(Multiset('aab').distinct_elements())
2
```

x in s

Test *x* for membership in *s*.

x not in s

Test *x* for non-membership in *s*.

isdisjoint(*other*)

Return `True` if the multiset has no elements in common with *other*.

issubset(*other*)

multiset <= other

Test whether every element in the multiset is in *other* and each element's multiplicity in the multiset is less than or equal to its multiplicity in *other*.

multiset < other

Test whether the multiset is a proper subset of *other*, that is, `multiset <= other and multiset != other`.

issuperset(*other*)
multiset >= other
Test whether every element in *other* is in the multiset and each element's multiplicity in *other* is less than or equal to its multiplicity in the multiset.

multiset > other
Test whether the multiset is a proper superset of *other*, that is, `multiset >= other and multiset != other`.

union(**others*)
multiset | other | ...
Return a new multiset with elements from the multiset and all others. The maximal multiplicity over all sets is used for each element.

combine(**others*)
multiset + other + ...
Return a new multiset with elements from the multiset and all others. Each element's multiplicities are summed up for the new set.

intersection(**others*)
multiset & other & ...
Return a new multiset with elements common to the multiset and all others. The minimal multiplicity over all sets is used for each element.

difference(**others*)
multiset - other - ...
Return a new multiset with elements in the multiset that are not in the others. This will subtract all the *others'* multiplicities and remove the element from the multiset if its multiplicity reaches zero.

symmetric_difference(*other*)
multiset ^ other
Return a new multiset with elements from either the multiset or *other* where their multiplicity is the absolute difference of the two multiplicities.

copy()
Multiset(*multiset*)
Return a new shallow copy of the multiset.

The following methods are not supported by *FrozenMultiset*, as it is equivalent to the builtin `frozenset` class and is immutable:

union_update(**others*)
multiset |= other | ...
Update the multiset, adding elements from all others. The maximal multiplicity over all sets is used for each element. For a version of this method that works more closely to `dict.update()`, see *update()*.

intersection_update(**others*)
multiset &= other & ...
Update the multiset, keeping only elements found in it and all others. The minimal multiplicity over all sets is used for each element.

difference_update(**others*)
multiset -= other | ...
Update the multiset, removing elements found in others. This will subtract all the *others'* multiplicities and remove the element from the multiset if its multiplicity reaches zero.

symmetric_difference_update(*other*)
multiset ^= other
Update the multiset, keeping only elements found in either the multiset or *other* but not both. The new multiplicity is the absolute difference of the two original multiplicities.

add(*element*, *multiplicity=1*)
s[element] += multiplicity
s[element] = multiplicity
Add element *element* to the multiset *s*. If the optional multiplicity is specified, more than one element can be added at the same time by adding to its.

Note that adding the same element multiple times will increase its multiplicity and thus change the multiset, whereas for a regular `set` this would have no effect.

You can also set the element's multiplicity directly via key assignment.

remove(*element*, *multiplicity=None*)
del s[element]
Remove all elements *element* from the multiset. Raises `KeyError` if *element* is not contained in the set.

If the optional multiplicity is specified, only the given number is subtracted from the element's multiplicity. This might still completely remove the element from the multiset depending on its original multiplicity.

This method returns the original multiplicity of the element before it was removed.

You can also delete the element directly via key access.

discard(*element*, *multiplicity=None*)
s[element] -= multiplicity
s[element] = 0
Remove element *element* from the set if it is present. If the optional multiplicity is specified, only the given number is subtracted from the element's multiplicity. This might still completely remove the element from the multiset depending on its original multiplicity.

This method returns the original multiplicity of the element before it was removed.

You can also set the element's multiplicity directly via key assignment.

clear()
Remove all elements from the multiset.

Note, the non-operator versions of *union()*, *intersection()*, *difference()*, and *symmetric_difference()*, *issubset()*, and *issuperset()*, *update()*, *intersection_update()*, *difference_update()*, and *symmetric_difference_update()* methods will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes error-prone constructions like `Multiset('abc') & 'cbs'` in favor of the more readable `Multiset('abc').intersection('cbs')`.

The *Multiset* supports set to set comparisons. Two multisets are equal if and only if every element of each multiset is contained in the other and each element's multiplicity is the same in both multisets (each is a subset of the other). A multiset is less than another set if and only if it is a proper subset of the second set (is a subset, but is not equal). A multiset is greater than another set if and only if it is a proper superset of the second set (is a superset, but is not equal). These comparisons work with both sets and multisets:

```
>>> Multiset('ab') == {'a', 'b'}
True
```

Multiset elements, like set elements and dictionary keys, must be hashable.

Binary operations that mix `set` or `frozenset` instances with *Multiset* instances will always return a *Multiset*. Since the *Multiset* internally uses a `dict`, it exposes some of its methods and allows key-based access for elements:

s[element]
Return the multiplicity of *element* in *s*. Returns `0` for elements that are not in the multiset.

s[element] = value
Set the multiplicity of *element* to *value*. Setting the multiplicity to `0` removes the element. This is not supported by *FrozenMultiset*.

del s[element]
See *remove()*. This is not supported by *FrozenMultiset*.

iter(s)
Return an iterator over the elements in the multiset.

In contrast to both the `dict` and `set` implementations, this will repeat elements whose multiplicity is greater than `1`:

```
>>> sorted(Multiset('aab'))
['a', 'a', 'b']
```

To only get distinct elements, use the *distinct_elements()* method.

classmethod from_elements(*elements*, *multiplicity*)
Create a new multiset with elements from *elements* and all multiplicities set to *multiplicity*.

get(*element*, *default*)
Return the multiplicity for *element* if it is in the multiset, else *default*.

items()
Return a new view of the multiset's items (`(element, multiplicity)` pairs). See the documentation of dict view objects.

Note that this view is unordered.

distinct_elements()
keys()
Return a new view of the multiset's distinct elements. See the documentation of dict view objects.

Note that this view is unordered.

update(**others*)
multiset += other + ...
Update the multiset, adding elements from all others. Each element's multiplicities is summed up for the new multiset. This is not supported by *FrozenMultiset*.

For a version of this method that works more closely to the original `set.update()`, see *union_update()*.

pop(*element*, *default*)
If *element* is in the multiset, remove it and return its multiplicity, else return *default*. This is not supported by *FrozenMultiset*.

popitem()
Remove and return an arbitrary `(element, multiplicity)` pair from the multiset. This is not supported by *FrozenMultiset*.

popitem() is useful to destructively iterate over a multiset. If the multiset is empty, calling *popitem()* raises a `KeyError`.

setdefault(*element*, *default*)
If *element* is in the multiset, return its multiplicity. If not, insert *element* with a multiplicity of *default* and return *default*. This is not supported by *FrozenMultiset*.

multiplicities()
values()
Return a new view of the multiset's multiplicities. See the documentation of dict view objects.

Note that this view is unordered.

The multiset also adds some new methods for multiplying a multiset with an `int` factor:

times (*factor*)
multiset * factor
Return a copy of the multiset where each multiplicity is multiplied by *factor*.

times_update (*factor*)
multiset *= factor
Update the multiset, multiplying each multiplicity with *factor*. This is not supported by *FrozenMultiset*.

class FrozenMultiset ([*mapping or iterable*])
This is an immutable version of the *Multiset* and supports all non-mutating methods of it.

Because it is immutable, it is also hashable and can be used e.g. as a dictionary key or in a set.

class BaseMultiset
This is the base class of both *Multiset* and *FrozenMultiset*. While it cannot instantiated directly, it can be used for isinstance checks:

```
>>> isinstance(Multiset(), BaseMultiset)
True
>>> isinstance(FrozenMultiset(), BaseMultiset)
True
```

Python Module Index

m

Index

Symbols

A

B

C

D

F

G

I

K

M

P

R

S

T

U

V