8.4. `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.

8.4.1. Collections Abstract Base Classes

The collections module offers the following ABCs:

ABC	Inherits from	Abstract Methods	Mixin Methods
`Container`		`__contains__`	
`Hashable`		`__hash__`	
`Iterable`		`__iter__`	
`Iterator`	`Iterable`	`__next__`	`__iter__`
`Reversible`	`Iterable`	`__reversed__`	
`Generator`	`Iterator`	`send`, `throw`	`close`, `__iter__`, `__next__`
`Sized`		`__len__`	
`Callable`		`__call__`	
`Collection`	`Sized`, `Iterable`, `Container`	`__contains__`, `__iter__`, `__len__`	
`Sequence`	`Reversible`, `Collection`	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index`, and `count`
`MutableSequence`	`Sequence`	`__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `insert`	Inherited `Sequence` methods and `append`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`
`ByteString`	`Sequence`		

ABC	Inherits from	Abstract Methods	Mixin Methods
		`__getitem__`, `__len__`	Inherited `Sequence` methods
`Set`	`Collection`	`__contains__`, `__iter__`, `__len__`	`__le__`, `__lt__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__and__`, `__or__`, `__sub__`, `__xor__`, and `isdisjoint`
`MutableSet`	`Set`	`__contains__`, `__iter__`, `__len__`, `add`, `discard`	Inherited `Set` methods and `clear`, `pop`, `remove`, `__ior__`, `__iand__`, `__ixor__`, and `__isub__`
`Mapping`	`Collection`	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `keys`, `items`, `values`, `get`, `__eq__`, and `__ne__`
`MutableMapping`	`Mapping`	`__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__`	Inherited `Mapping` methods and `pop`, `popitem`, `clear`, `update`, and `setdefault`
`MappingView`	`Sized`		`__len__`
`ItemsView`	`MappingView`, `Set`		`__contains__`, `__iter__`
`KeysView`	`MappingView`, `Set`		`__contains__`, `__iter__`
`ValuesView`	`MappingView`		`__contains__`, `__iter__`
`Awaitable`		`__await__`	
`Coroutine`	`Awaitable`	`send`, `throw`	`close`
`AsyncIterable`		`__aiter__`	
`AsyncIterator`	`AsyncIterable`	`__anext__`	`__aiter__`
`AsyncGenerator`	`AsyncIterator`	`asend`, `athrow`	`aclose`, `__aiter__`, `__anext__`

class `collections.abc.`**`Container`**
class `collections.abc.`**`Hashable`**
class `collections.abc.`**`Sized`**
class `collections.abc.`**`Callable`**

ABCs for classes that provide respectively the methods `__contains__()`, `__hash__()`, `__len__()`, and `__call__()`.

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()` or `asyncio.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()` or `asyncio.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.

These 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 auto
```

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

1. Since some set operations create new sets, the default mixin methods need a way to create new instances from an iterable. The class constructor is assumed to have a signature in the form `ClassName(iterable)`. That assumption is factored-out to an internal classmethod called `_from_iterable()` which calls `cls(iterable)` to produce a new set. If the `Set` mixin is being used in a class with a different constructor signature, you will need to override `_from_iterable()` with a classmethod that can construct new instances from an iterable argument.
2. To override the comparisons (presumably for speed, as the semantics are fixed), redefine `__le__()` and `__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**.