3.3.6. Emulating container types

The following methods can be defined to implement container objects. Containers usually are sequences (such as lists or tuples) or mappings (like dictionaries), but can represent other containers as well. The first set of methods is used either to emulate a sequence or to emulate a mapping; the difference is that for a sequence, the allowable keys should be the integers *k* for which `0 <= k < N`where *N* is the length of the sequence, or slice objects, which define a range of items. It is also recommended that mappings provide the methods `keys()`,`values()`, `items()`, `get()`,`clear()`, `setdefault()`, `pop()`,`popitem()`, `copy()`, and `update()`behaving similar to those for Python's standard dictionary objects. The `collections` module provides a `MutableMapping`abstract base class to help create those methods from a base set of`__getitem__()`, `__setitem__()`,`__delitem__()`, and `keys()`. Mutable sequences should provide methods `append()`, `count()`,`index()`, `extend()`, `insert()`,`pop()`, `remove()`, `reverse()` and`sort()`, like Python standard list objects. Finally, sequence types should implement addition (meaning concatenation) and multiplication (meaning repetition) by defining the methods`__add__()`, `__radd__()`,`__iadd__()`, `__mul__()`,`__rmul__()` and `__imul__()`described below; they should not define other numerical operators. It is recommended that both mappings and sequences implement the `__contains__()`method to allow efficient use of the`in` operator; for mappings, `in`should search the mapping's keys; for sequences, it should search through the values. It is further recommended that both mappings and sequences implement the`__iter__()` method to allow efficient iteration through the container; for mappings,`__iter__()` should be the same as`keys()`; for sequences, it should iterate through the values.

`object.`**`__len__`**(*self*)

Called to implement the built-in function `len()`. Should return the length of the object, an integer `>=` 0. Also, an object that doesn't define a`__bool__()` method and whose`__len__()` method returns zero is considered to be false in a Boolean context.

`object.`**`__length_hint__`**(*self*)

Called to implement`operator.length_hint()`. Should return an estimated length for the object (which may be greater or less than the actual length). The length must be an integer `>=` 0. This method is purely an optimization and is never required for correctness.

New in version 3.4.

Note Slicing is done exclusively with the following three methods. A call like

```
a[1:2] = b
```

is translated to

```
a[slice(1, 2, None)] = b
```

and so forth. Missing slice items are always filled in with `None`.

object.**`__getitem__`**(*self*,*key*)

Called to implement evaluation of `self[key]`. For sequence types, the accepted keys should be integers and slice objects. Note that the special interpretation of negative indexes (if the class wishes to emulate a sequence type) is up to the `__getitem__()` method. If *key* is of an inappropriate type, `TypeError` may be raised; if of a value outside the set of indexes for the sequence (after any special interpretation of negative values),`IndexError` should be raised. For mapping types, if *key* is missing (not in the container),`KeyError` should be raised.

Note `for` loops expect that an `IndexError` will be raised for illegal indexes to allow proper detection of the end of the sequence.

object.**`__missing__`**(*self*,*key*)

Called by `dict.__getitem__()`to implement `self[key]` for dict subclasses when key is not in the dictionary.

object.**`__setitem__`**(*self*, *key*,*value*)

Called to implement assignment to `self[key]`. Same note as for`__getitem__()`. This should only be implemented for mappings if the objects support changes to the values for keys, or if new keys can be added, or for sequences if elements can be replaced. The same exceptions should be raised for improper *key* values as for the`__getitem__()` method.

object.**`__delitem__`**(*self*,*key*)

Called to implement deletion of`self[key]`. Same note as for`__getitem__()`. This should only be implemented for mappings if the objects support removal of keys, or for sequences if elements can be removed from the sequence. The same exceptions should be raised for improper *key* values as for the `__getitem__()`method.

object.**`__iter__`**(*self*)

This method is called when an iterator is required for a container. This method should return a new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container.

Iterator objects also need to implement this method; they are required to return themselves. For more information on iterator objects, see Iterator Types.

object.**`__reversed__`**(*self*)

Called (if present) by the`reversed()` built-in to implement reverse iteration. It should return a new iterator object that iterates over all the objects in the container in reverse order.

If the `__reversed__()` method is not provided, the `reversed()`built-in will fall back to using the sequence protocol (`__len__()`and `__getitem__()`). Objects that support the sequence protocol should only provide`__reversed__()` if they can provide an implementation that is more efficient than the one provided by `reversed()`.

The membership test operators (`in`and `not in`) are normally implemented as an iteration through a sequence. However, container objects can supply the following special method with a more efficient implementation, which also does not require the object be a sequence.

object.**__contains__**(*self*,*item*)

Called to implement membership test operators. Should return true if *item* is in*self*, false otherwise. For mapping objects, this should consider the keys of the mapping rather than the values or the key-item pairs.

For objects that don't define`__contains__()`, the membership test first tries iteration via `__iter__()`, then the old sequence iteration protocol via `__getitem__()`, see this section in the language reference.