3.4.6. 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. (For backwards compatibility, the method`__getslice__()` (see below) can also be defined to handle simple, but not extended slices.) It is also recommended that mappings provide the methods **`keys()`**, **`values()`**, **`items()`**,**`has_key()`**, **`get()`**, **`clear()`**, **`setdefault()`**,**`iterkeys()`**, **`itervalues()`**, **`iteritems()`**,**`pop()`**, **`popitem()`**, **`copy()`**, and **`update()`**behaving similar to those for Python's standard dictionary objects. The`UserDict` module provides a **`DictMixin`**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`__coerce__()` or 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 be equivalent of **`has_key()`**; 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 **`iterkeys()`**; 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 `__nonzero__()` method and whose `__len__()` method returns zero is considered to be false in a Boolean context.

`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, and should also be made available as the method**`iterkeys()`**.

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()`.

New in version 2.6.

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.

3.4.7. Additional methods for emulation of sequence types

The following optional methods can be defined to further emulate sequence objects. Immutable sequences methods should at most only define `__getslice__()`; mutable sequences might define all three methods.

object.**`__getslice__`**(*self*, *i*, *j*)

> *Deprecated since version 2.0:*Support slice objects as parameters to the `__getitem__()`method. (However, built-in types in CPython currently still implement `__getslice__()`. Therefore, you have to override it in derived classes when implementing slicing.)

Called to implement evaluation of `self[i:j]`. The returned object should be of the same type as *self*. Note that missing *i* or *j* in the slice expression are replaced by zero or `sys.maxsize`, respectively. If negative indexes are used in the slice, the length of the sequence is added to that index. If the instance does not implement the `__len__()` method, an `AttributeError` is raised. No guarantee is made that indexes adjusted this way are not still negative. Indexes which are greater than the length of the sequence are not modified. If no `__getslice__()` is found, a slice object is created instead, and passed to `__getitem__()` instead.

object.**`__setslice__`**(*self*, *i*, *j*,*sequence*)

Called to implement assignment to `self[i:j]`. Same notes for *i* and *j* as for `__getslice__()`.

This method is deprecated. If no `__setslice__()` is found, or for extended slicing of the form `self[i:j:k]`, a slice object is created, and passed to `__setitem__()`, instead of `__setslice__()` being called.

object.**`__delslice__`**(*self*, *i*, *j*)

Called to implement deletion of `self[i:j]`. Same notes for *i* and *j*as for `__getslice__()`. This method is deprecated. If no `__delslice__()` is found, or for extended slicing of the form `self[i:j:k]`, a slice object is created, and passed to `__delitem__()`, instead of `__delslice__()` being called.

Notice that these methods are only invoked when a single slice with a single colon is used, and the slice method is available. For slice operations involving extended slice notation, or in absence of the slice methods, `__getitem__()`, `__setitem__()`or `__delitem__()` is called with a slice object as argument.

The following example demonstrate how to make your program or module compatible with earlier versions of Python (assuming that methods `__getitem__()`, `__setitem__()` and `__delitem__()` support slice objects as arguments):

```
class MyClass:
    ...
    def __getitem__(self, index):
        ...
    def __setitem__(self, index, value):
        ...
    def __delitem__(self, index):
        ...

    if sys.version_info < (2, 0):
        # They won't be defined if version is at least 2.0 final

        def __getslice__(self, i, j):
            return self[max(0, i):max(0, j):]
        def __setslice__(self, i, j, seq):
            self[max(0, i):max(0, j):] = seq
        def __delslice__(self, i, j):
            del self[max(0, i):max(0, j):]
    ...
```

Note the calls to `max()`; these are necessary because of the handling of negative indices before the **`__*slice__()`** methods are called. When negative indexes are used, the **`__*item__()`** methods receive them as provided, but the **`__*slice__()`**methods get a “cooked” form of the index values. For each negative index value, the length of the sequence is added to the index before calling the method (which may still result in a negative index); this is the customary handling of negative indexes by the built-in sequence types, and the **`__*item__()`** methods are expected to do this as well. However, since they should already be doing that, negative indexes cannot be passed in; they must be constrained to the bounds of the sequence before being passed to the **`__*item__()`** methods. Calling `max(0, i)`conveniently returns the proper value.