3.3.5. Emulating callable objects

object.\_\_call\_\_(self[, args...])

Called when the instance is “called” as a function; if this method is defined, x(arg1, arg2, ...) is a shorthand for x.\_\_call\_\_(arg1, arg2, ...).

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.\_\_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 keys().

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.

3.3.7. Emulating numeric types

The following methods can be defined to emulate numeric objects. Methods corresponding to operations that are not supported by the particular kind of number implemented (e.g., bitwise operations for non-integral numbers) should be left undefined.

object.\_\_add\_\_(self, other)object.\_\_sub\_\_(self, other)object.\_\_mul\_\_(self, other)object.\_\_truediv\_\_(self, other)object.\_\_floordiv\_\_(self, other)object.\_\_mod\_\_(self, other)object.\_\_divmod\_\_(self, other)object.\_\_pow\_\_(self, other[, modulo])object.\_\_lshift\_\_(self, other)object.\_\_rshift\_\_(self, other)object.\_\_and\_\_(self, other)object.\_\_xor\_\_(self, other)object.\_\_or\_\_(self, other)

These methods are called to implement the binary arithmetic operations (+, -, \*, /, //, %, divmod(), pow(), \*\*, <<, >>, &, ^, |). For instance, to evaluate the expression x + y, where x is an instance of a class that has an \_\_add\_\_() method, x.\_\_add\_\_(y) is called. The \_\_divmod\_\_() method should be the equivalent to using \_\_floordiv\_\_() and \_\_mod\_\_(); it should not be related to \_\_truediv\_\_(). Note that \_\_pow\_\_() should be defined to accept an optional third argument if the ternary version of the built-in pow() function is to be supported.

If one of those methods does not support the operation with the supplied arguments, it should return NotImplemented.

object.\_\_radd\_\_(self, other)object.\_\_rsub\_\_(self, other)object.\_\_rmul\_\_(self, other)object.\_\_rtruediv\_\_(self, other)object.\_\_rfloordiv\_\_(self, other)object.\_\_rmod\_\_(self, other)object.\_\_rdivmod\_\_(self, other)object.\_\_rpow\_\_(self, other)object.\_\_rlshift\_\_(self, other)object.\_\_rrshift\_\_(self, other)object.\_\_rand\_\_(self, other)object.\_\_rxor\_\_(self, other)object.\_\_ror\_\_(self, other)

These methods are called to implement the binary arithmetic operations (+, -, \*, /, //, %, divmod(), pow(), \*\*, <<, >>, &, ^, |) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation and the operands are of different types. [2] For instance, to evaluate the expression x - y, where y is an instance of a class that has an \_\_rsub\_\_() method, y.\_\_rsub\_\_(x) is called if x.\_\_sub\_\_(y) returns NotImplemented.

Note that ternary pow() will not try calling \_\_rpow\_\_() (the coercion rules would become too complicated).

Note:

If the right operand’s type is a subclass of the left operand’s type and that subclass provides the reflected method for the operation, this method will be called before the left operand’s non-reflected method. This behavior allows subclasses to override their ancestors’ operations.

object.\_\_iadd\_\_(self, other)object.\_\_isub\_\_(self, other)object.\_\_imul\_\_(self, other)object.\_\_itruediv\_\_(self, other)object.\_\_ifloordiv\_\_(self, other)object.\_\_imod\_\_(self, other)object.\_\_ipow\_\_(self, other[, modulo])object.\_\_ilshift\_\_(self, other)object.\_\_irshift\_\_(self, other)object.\_\_iand\_\_(self, other)object.\_\_ixor\_\_(self, other)object.\_\_ior\_\_(self, other)

These methods are called to implement the augmented arithmetic assignments (+=, -=, \*=, /=, //=, %=, \*\*=, <<=, >>=, &=, ^=, |=). These methods should attempt to do the operation in-place (modifying self) and return the result (which could be, but does not have to be, self). If a specific method is not defined, the augmented assignment falls back to the normal methods. For instance, if x is an instance of a class with an \_\_iadd\_\_() method, x += y is equivalent to x = x.\_\_iadd\_\_(y) . Otherwise, x.\_\_add\_\_(y) and y.\_\_radd\_\_(x) are considered, as with the evaluation of x + y. In certain situations, augmented assignment can result in unexpected errors (see Why does a\_tuple[i] += [‘item’] raise an exception when the addition works?), but this behavior is in fact part of the data model.

object.\_\_neg\_\_(self)object.\_\_pos\_\_(self)object.\_\_abs\_\_(self)object.\_\_invert\_\_(self)

Called to implement the unary arithmetic operations (-, +, abs() and ~).

object.\_\_complex\_\_(self)object.\_\_int\_\_(self)object.\_\_float\_\_(self)object.\_\_round\_\_(self[, n])

Called to implement the built-in functions complex(), int(), float() and round(). Should return a value of the appropriate type.

object.\_\_index\_\_(self)

Called to implement operator.index(), and whenever Python needs to losslessly convert the numeric object to an integer object (such as in slicing, or in the built-in bin(), hex() and oct() functions). Presence of this method indicates that the numeric object is an integer type. Must return an integer.

Note:

In order to have a coherent integer type class, when \_\_index\_\_() is defined \_\_int\_\_() should also be defined, and both should return the same value.