5.5. Iterator Types

New in version 2.2.

Python supports a concept of iteration over containers. This is implemented using two distinct methods; these are used to allow user-defined classes to support iteration. Sequences, described below in more detail, always support the iteration methods.

One method needs to be defined for container objects to provide iteration support:

container.**__iter__**()

> Return an iterator object. The object is required to support the iterator protocol described below. If a container supports different types of iteration, additional methods can be provided to specifically request iterators for those iteration types. (An example of an object supporting multiple forms of iteration would be a tree structure which supports both breadth-first and depth-first traversal.) This method corresponds to the `tp_iter` slot of the type structure for Python objects in the Python/C API.

The iterator objects themselves are required to support the following two methods, which together form the *iterator protocol*:

iterator.**__iter__**()

> Return the iterator object itself. This is required to allow both containers and iterators to be used with the `for` and `in`statements. This method corresponds to the `tp_iter` slot of the type structure for Python objects in the Python/C API.

iterator.**next**()

> Return the next item from the container. If there are no further items, raise the `StopIteration`exception. This method corresponds to the `tp_iternext`slot of the type structure for Python objects in the Python/C API.

Python defines several iterator objects to support iteration over general and specific sequence types, dictionaries, and other more specialized forms. The specific types are not important beyond their implementation of the iterator protocol.

The intention of the protocol is that once an iterator's `next()` method raises `StopIteration`, it will continue to do so on subsequent calls. Implementations that do not obey this property are deemed broken. (This constraint was added in Python 2.3; in Python 2.2, various iterators are broken according to this rule.)

5.5.1. Generator Types

Python's generators provide a convenient way to implement the iterator protocol. If a container object's __iter__() method is implemented as a generator, it will automatically return an iterator object (technically, a generator object) supplying the __iter__() and next()methods. More information about generators can be found in the documentation for the yield expression.