



Ordering Information: [Python How to Program](#)

- View the complete [Table of Contents](#)
- Read the [Preface](#)
- Download the [Code Examples](#)

To view all the Deitel products and services available, visit the Deitel Kiosk on InformIT at www.informIT.com/deitel.

To follow the Deitel publishing program, sign-up now for the *DEITEL™ BUZZ ONLINE* e-mail newsletter at www.deitel.com/newsletter/subscribeinformIT.html
To learn more about Deitel instructor-led corporate training delivered at your location, visit www.deitel.com/training or contact Christi Kelsey at (978) 461-5880 or e-mail: christi.kelsey@deitel.net.

Note from the Authors: This article is an excerpt from Appendix O, Section O.2 of *Python How to Program 1/e*. In this article, we motivate the need for iterators and define a class that supports iterators. Iterators are new to the Python language—they were added in version 2.2. Classes that hold sequences of values are prime candidates for iterators. If a class supports iterators, then a client of the class can use a for loop to progress through the items in the class's sequence. Readers should be familiar with OOP (constructors, the object reference argument, private attributes, class customization) and exception handling. The code examples included in this article show readers examples using the Deitel™ signature *LIVE-CODE™ Approach*, which presents all concepts in the context of complete working programs followed by the screen shots of the actual inputs and outputs.

O.2 Iterators

In Chapter 12, Exceptions, we mentioned that Python **for** loops use exceptions to determine when the end of a sequence has been reached. This makes it possible to write classes whose objects may be used with **for** loops. In earlier versions of Python, the class author provided method `__getitem__`, which enabled clients to perform list- or dictionary-like subscript access for objects of the class. As an example, Fig.O.1 defines class **MyRange**, a class that simulates a sequence returned from a call to built-in function **range**. A client creates an object of class **MyRange** by passing to the constructor values for parameters **start**, **stop** and **step**, which the constructor simply passes to function **range** (line 10). The class stores the return value from **range** in attribute `__sequence`.

```

1  # Fig. O.1: NewRange.py
2  # __getitem__ used in a for loop.
3
4  class MyRange:
5      """Simple class to simulate a range of integer values"""
6
7      def __init__( self, start, stop, step ):
8          """Class MyRange constructor; takes start, stop and step"""
9
10         self.__sequence = range( start, stop, step )
11
12         def __getitem__( self, subscript ):
13             """Overridden sequence element access"""
14
15             return self.__sequence[ subscript ]

```

```

Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from NewRange import MyRange
>>> myRange = MyRange( 0, 10, 1 )
>>>
>>> print myRange[ 0 ]
0
>>> print myRange[ 8 ]
8
>>>
>>> for value in myRange:
...     print value,
...
0 1 2 3 4 5 6 7 8 9

```

Fig. O.1 `__getitem__` method to emulate iteration.

Method `__getitem__` provides subscript access (using operator `[]`) for objects of the class. The method takes one argument—**subscript**—and returns the element stored in `__sequence[subscript]`. If the specified subscript lies outside the bounds of the sequence, Python raises an **IndexError** exception.

The interactive session that follows the class definition creates an object of class **MyRange** with the values 0–9 and demonstrates random attribute access (lines 4–10). When the interpreter encounters the statements in lines 12–13 of the interactive session, the interpreter translates the code into successive subscript operations on object **myRange**. The interactive session in Fig.O.2 simulates how Python executes the **for** loop from the previous session. The interpreter starts at index 0 (variable **currentIndex**). The session then enters a **for** loop that attempts to access the value stored in object **myRange** at index **currentIndex**, which implicitly calls the object's `__getitem__` method. If the index lies outside the bounds for that object, method `__getitem__` raises an **IndexError** exception; otherwise, the interpreter assigns the returned value to control variable **value**. If the attribute access in line 11 of the interactive session does not raise an **IndexError** exception, the interpreter prints variable **value** and increments the control variable. Then, program control returns to the top of the **while** loop, where the interpreter attempts to access the second value in sequence **myRange**. The interpreter continues to increment the control variable and access the sequence's corresponding element until method `__getitem__` raises an **IndexError** exception. At this point, the **except** clause in lines 12-13 of the interactive session breaks out of the **while** loop.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from NewRange import MyRange
>>> myRange = MyRange( 0, 10, 1 )
>>>
>>> currentIndex = 0
>>>
>>> while 1:
...     try:
...         value = myRange[ currentIndex ]
...     except IndexError:
...         break
...     else:
...         print value,
...         currentIndex += 1
...
0 1 2 3 4 5 6 7 8 9
```

Fig. O.2 `__getitem__` and **for** loops.

Although method `__getitem__` provides a way for clients to use an object with **for/in** syntax, method `__getitem__` was intended for random attribute access (e.g., index access for lists or key lookup for dictionaries), rather than iteration.¹ The designers of the language felt it would be cleaner to separate the random attribute access operation from the object iteration operation.² To address this issue, Python 2.2 introduces the concept of *iterators*—special objects that define operations for progressing through sequences.

1. A. Kuchling, "What's New in Python 2.2," <www.amk.ca/python/2.2>.
2. G. van Rossum and K. Yee, "Iterators," 30 April 2001 <python.sourceforge.net/peps/pep-0234.html>.

Iterators are useful for many kinds of sequences, including list-like objects (such as built-in types `list` and `tuple`), programmer-defined container objects (e.g., trees) or unbounded sequences—sequences whose length is unknown or cannot be determined in advance (e.g., input and output streams).

Figure O.3 contains a new implementation of class `MyRange`—called `RangeIterator`—that supports iterators. Before we discuss the class definition, consider the main program (lines 29–71) that uses iterators to progress through the sequences defined by objects of class `RangeIterator`. Line 32 creates object `rangel` that contains the values 0–9. Lines 36–37 use the object in a `for` loop, to iterate over the object’s elements. The `for`-loop syntax for an object of class `RangeIterator` is identical to that of an object of the class as defined in Fig. O.1, even though the new class definition does not define method `__getitem__`.

```

1  # Fig. O.3: NewRangeIterator.py
2  # Iterator class that defines a sequence.
3
4  class RangeIterator:
5      """Simple class to simulate a range"""
6
7      def __init__( self, start, stop, step ):
8          """RangeIterator constructor; takes start, stop and step"""
9
10         self.__sequence = range( start, stop, step )
11         self.__nextValue = 0 # subscript of next value to produce
12
13         def __iter__( self ):
14             """Returns iterator for object of class RangeIterator"""
15
16             return self
17
18         def next( self ):
19             """Iterator method to produce next value in sequence"""
20
21             try:
22                 value = self.__sequence[ self.__nextValue ]
23             except IndexError:
24                 raise StopIteration
25             else:
26                 self.__nextValue += 1
27                 return value
28
29     def main():
30
31         # create object of class RangeIterator, use for loop to iterate
32         rangel = RangeIterator( 0, 10, 1 )
33
34         print "Iterate over the values in rangel using a for loop:"
35
36         for value in rangel:
37             print value,
```

Fig. O.3 Iterator class that defines a sequence.

```
38
39     print
40
41     # create object of class RangeIterator, call next to iterate
42     range2 = RangeIterator( 0, 10, 1 )
43     range2Iterator = iter( range2 ) # retrieve iterator for range2
44
45     print "\nCall method next for range2Iterator:"
46
47     while 1:
48
49         try:
50             value = range2Iterator.next()
51         except StopIteration:
52             break
53         else:
54             print value,
55
56     print
57
58     # create one object of class RangeIterator two iterators
59     # for that object
60     range3 = RangeIterator( 0, 10, 1 )
61     range3Iterator1 = iter( range3 )
62     range3Iterator2 = iter( range3 )
63
64     print "\nCall next for two iterators of the same object:"
65
66     for i in range( 10 ):
67         print "Loop iteration %d: range3Iterator1.next() = %d" % \
68             ( i, range3Iterator1.next() )
69         print "Loop iteration %d: range3Iterator2.next() = %d" % \
70             ( i, range3Iterator2.next() )
71         print
72
73 if __name__ == "__main__":
74     main()
```

Fig. O.3 Iterator class that defines a sequence.

```
Iterate over the values in range1 using a for loop:
0 1 2 3 4 5 6 7 8 9

Call method next for range2Iterator:
0 1 2 3 4 5 6 7 8 9

Call next for two iterators of the same object:
Loop iteration 0: range3Iterator1.next() = 0
Loop iteration 0: range3Iterator2.next() = 1

Loop iteration 1: range3Iterator1.next() = 2
Loop iteration 1: range3Iterator2.next() = 3

Loop iteration 2: range3Iterator1.next() = 4
Loop iteration 2: range3Iterator2.next() = 5

Loop iteration 3: range3Iterator1.next() = 6
Loop iteration 3: range3Iterator2.next() = 7

Loop iteration 4: range3Iterator1.next() = 8
Loop iteration 4: range3Iterator2.next() = 9

Traceback (most recent call last):
  File "newrange3.py", line 74, in ?
    main()
  File "newrange3.py", line 67, in main
    print "Loop iteration %d: range3Iterator1.next() = %d" % \
  File "newrange3.py", line 24, in next
    raise StopIteration
StopIteration
```

Fig. O.3 Iterator class that defines a sequence.

Just as earlier versions of Python “rewrite” **for** loops to call an object’s `__getitem__` method, Python 2.2 rewrites **for** loops to progress through an object’s iterator. Lines 42–54 simulate how a Python **for** loop works on an object that supports iterators. Line 42 creates `range2`—an object of class `RangeIterator`. All objects that support (i.e., provide) iterators define two methods—`__iter__` and `next`. A client obtains an iterator for an object by passing the object’s name to built-in function `iter` (line 43). This function implicitly invokes the object’s `__iter__` method, which allows Python to create an iterator for that object. The **while** loop in lines 47–54 corresponds to the **for** loop in lines 36–37. A program progresses through the values in an object’s sequence by calling method `next` for an iterator of that object (line 50). This method either returns the next element in the object’s sequence—in which case line 54 prints the value—or raises the `StopIteration` exception, to indicate that the sequence contains no more values—in which case line 52 breaks out of the **while** loop.



Software Engineering Observation O.1

If a program uses a **for** loop to iterate over an object that does not support iteration through methods `__iter__` and `next`, Python uses the object's `__getitem__` method instead. If that object also does not define a `__getitem__` method, Python raises a **TypeError** exception when the program attempts to iterate over the object.

A client may obtain more than one iterator for a particular sequence (lines 61–62). In this case, each call to the iterator's `next` method retrieves the next value in the sequence. Sometimes, this can lead to subtle logic errors. The **for** suite in lines 67–71 executes 10 times, each time calling `next` on two iterators for the same object. Even though `range3Iterator1` and `range3Iterator2` are two different iterator objects, their `next` methods retrieve values from the same sequence (i.e., `range3`). Therefore, the program exhausts `range3`'s values after only 5 executions of the **for** suite, causing the program to terminate on a **StopIteration** exception. It is possible for a class to provide a unique iterator object for each call to function `iter`. For example, calling `iter` on an object that is a built-in list returns a unique iterator for that list's sequence. For programmer-defined classes, returning a unique iterator for an object sometimes requires complex state information. In these cases, the programmer-defined class may provide a generator (discussed in Section O.3) to produce the values in an object's sequence.



Common Programming Error O.1

Iterating past the end of a sequence raises a **StopIteration** exception. Enclose in a **try** statement any code that calls an iterator's `next` method explicitly and which may therefore iterate past the end of a sequence.

We now discuss the class definition in lines 4–27. The constructor defines attribute `__sequence`, which contains an object's sequence values, and attribute `__nextValue`, a control variable used to mark the next value to produce in this sequence. Any class whose objects support iterators must define method `__iter__`, either explicitly or through inheritance. This method takes only the object reference argument (`self`) and should return an iterator object for the class. An iterator object is an object that defines method `next`, which clients can call to retrieve the next value in the sequence, and which raises exception **StopIteration** when the sequence values have been exhausted. Line 16 simply returns the object reference argument, because class `Range-Iterator` defines an appropriate `next` method.

Method `next` (lines 18–27) retrieves the next value from an object of class `RangeIterator`. The method attempts to retrieve the value that corresponds to the current location in an object's sequence (i.e., the index indicated by attribute `__nextValue`). If the current subscript location is out of bounds for the sequence, Python raises an **IndexError** exception. We catch this exception in lines 23–24 and raise a **StopIteration** exception to indicate that the sequence values have been exhausted; otherwise, lines 26–27 increment the control variable and return the retrieved value.