Solution If you want to implement a new kind of iteration pattern, define it using a generator function. Here’s a generator that produces a range of floating-point numbers: def frange(start, stop, increment): x = start while x < stop: yield x x += increment To use such a function, you iterate over it using a for loop or use it with some other function that consumes an iterable (e.g., sum(), list(), etc.). For example: >>> for n in frange(0, 4, 0.5): ... print(n) ... 0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 >>> list(frange(0, 1, 0.125)) [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875] >>> Discussion The mere presence of the yield statement in a function turns it into a generator. Unlike a normal function, a generator only runs in response to iteration. Here’s an experiment you can try to see the underlying mechanics of how such a function works: >>> def countdown(n): ... print('Starting to count from', n) ... ... ... ... ... while n > 0: yield n n -= 1 print('Done!') >>> # Create the generator, notice no output appears >>> c = countdown(3) >>> c >>> # Run to first yield and emit a value >>> next(c) Starting to count from 3 3 116 | Chapter 4: Iterators and Generators >>> # Run to the next yield >>> next(c) 2 >>> # Run to next yield >>> next(c) 1 >>> # Run to next yield (iteration stops) >>> next(c) Done! Traceback (most recent call last): File "", line 1, in StopIteration >>> The key feature is that a generator function only runs in response to “next” operations carried out in iteration. Once a generator function returns, iteration stops. However, the for statement that’s usually used to iterate takes care of these details, so you don’t normally need to worry about them. 4.4. Implementing the Iterator Protocol Problem You are building custom objects on which you would like to support iteration, but would like an easy way to implement the iterator protocol. Solution By far, the easiest way to implement iteration on an object is to use a generator function. In Recipe 4.2, a Node class was presented for representing tree structures. Perhaps you want to implement an iterator that traverses nodes in a depth-first pattern. Here is how you could do it: class Node: def \_\_init\_\_(self, value): self.\_value = value self.\_children = [] def \_\_repr\_\_(self): return 'Node({!r})'.format(self.\_value) def add\_child(self, node): self.\_children.append(node) def \_\_iter\_\_(self): return iter(self.\_children) 4.4. Implementing the Iterator Protocol | 117