2.3.5 Example: Range Class

As the final example for this section, we develop our own implementation of a class that mimics Python's built-in range class. Before introducing our class, we discuss the history of the built-in version. Prior to Python 3 being released, range was implemented as a function, and it returned a list instance with elements in the specified range. For example, range(2, 10, 2) returned the list [2, 4, 6, 8]. However, a typical use of the function was to support a for-loop syntax, such as **for** k **in** range(10000000). Unfortunately, this caused the instantiation and initialization of a list with the range of numbers. That was an unnecessarily expensive step, in terms of both time and memory usage.

The mechanism used to support ranges in Python 3 is entirely different (to be fair, the "new" behavior existed in Python 2 under the name xrange). It uses a strategy known as ***lazy evaluation***. Rather than creating a new list instance, range is a class that can effectively represent the desired range of elements without ever storing them explicitly in memory. To better explore the built-in range class, we recommend that you create an instance as r = range(8, 140, 5). The result is a relatively lightweight object, an instance of the range class, that has only a few behaviors. The syntax len(r) will report the number of elements that are in the given range (27, in our example). A range also supports the __getitem__ method, so that syntax r[15] reports the sixteenth element in the range (as r[0] is the first element). Because the class supports both __len__ and __getitem__, it inherits automatic support for iteration (see Section 2.3.4), which is why it is possible to execute a for loop over a range.

At this point, we are ready to demonstrate our own version of such a class. Code Fragment 2.6 provides a class we name Range (so as to clearly differentiate it from built-in range). The biggest challenge in the implementation is properly computing the number of elements that belong in the range, given the parameters sent by the caller when constructing a range. By computing that value in the constructor, and storing it as self._length, it becomes trivial to return it from the __len__ method. To properly implement a call to __getitem__(k), we simply take the starting value of the range plus k times the step size (i.e., for k=0, we return the start value). There are a few subtleties worth examining in the code:

- To properly support optional parameters, we rely on the technique described on page 27, when discussing a functional version of range.
- We compute the number of elements in the range as
  max(0, (stop − start + step − 1) // step)
  It is worth testing this formula for both positive and negative step sizes.
- The __getitem__ method properly supports negative indices by converting an index −k to len(self)−k before computing the result.

Nested Loops and the Quadratic Function

The quadratic function can also arise in the context of nested loops where the first iteration of a loop uses one operation, the second uses two operations, the third uses three operations, and so on. That is, the number of operations is

$$1+2+3+\cdots+(n-2)+(n-1)+n.$$

In other words, this is the total number of operations that will be performed by the nested loop if the number of operations performed inside the loop increases by one with each iteration of the outer loop. This quantity also has an interesting history.

In 1787, a German schoolteacher decided to keep his 9- and 10-year-old pupils occupied by adding up the integers from 1 to 100. But almost immediately one of the children claimed to have the answer! The teacher was suspicious, for the student had only the answer on his slate. But the answer, 5050, was correct and the student, Carl Gauss, grew up to be one of the greatest mathematicians of his time. We presume that young Gauss used the following identity.

Proposition 3.3: *For any integer $n \geq 1$, we have:*

$$1+2+3+\cdots+(n-2)+(n-1)+n = \frac{n(n+1)}{2}.$$

We give two "visual" justifications of Proposition 3.3 in Figure 3.3.

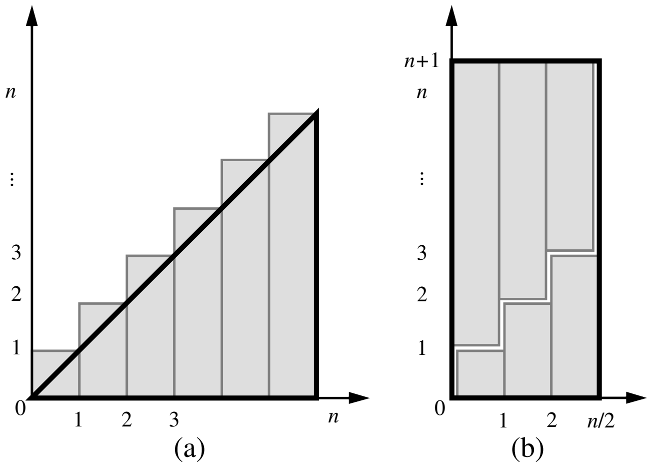

Figure 3.3: Visual justifications of Proposition 3.3. Both illustrations visualize the identity in terms of the total area covered by n unit-width rectangles with heights $1, 2, \ldots, n$. In (a), the rectangles are shown to cover a big triangle of area $n^2/2$ (base n and height n) plus n small triangles of area $1/2$ each (base 1 and height 1). In (b), which applies only when n is even, the rectangles are shown to cover a big rectangle of base $n/2$ and height $n+1$.

C-6.30 Alice has two queues, Q and R, which can store integers. Bob gives Alice 50 odd integers and 50 even integers and insists that she store all 100 integers in Q and R. They then play a game where Bob picks Q or R at random and then applies the round-robin scheduler, described in the chapter, to the chosen queue a random number of times. If the last number to be processed at the end of this game was odd, Bob wins. Otherwise, Alice wins. How can Alice allocate integers to queues to optimize her chances of winning? What is her chance of winning?

C-6.31 Suppose Bob has four cows that he wants to take across a bridge, but only one yoke, which can hold up to two cows, side by side, tied to the yoke. The yoke is too heavy for him to carry across the bridge, but he can tie (and untie) cows to it in no time at all. Of his four cows, Mazie can cross the bridge in 2 minutes, Daisy can cross it in 4 minutes, Crazy can cross it in 10 minutes, and Lazy can cross it in 20 minutes. Of course, when two cows are tied to the yoke, they must go at the speed of the slower cow. Describe how Bob can get all his cows across the bridge in 34 minutes.

Projects

P-6.32 Give a complete ArrayDeque implementation of the double-ended queue ADT as sketched in Section 6.3.2.

P-6.33 Give an array-based implementation of a double-ended queue supporting all of the public behaviors shown in Table 6.4 for the collections.deque class, including use of the maxlen optional parameter. When a length-limited deque is full, provide semantics similar to the collections.deque class, whereby a call to insert an element on one end of a deque causes an element to be lost from the opposite side.

P-6.34 Implement a program that can input an expression in postfix notation (see Exercise C-6.22) and output its value.

P-6.35 The introduction of Section 6.1 notes that stacks are often used to provide "undo" support in applications like a Web browser or text editor. While support for undo can be implemented with an unbounded stack, many applications provide only ***limited*** support for such an undo history, with a fixed-capacity stack. When push is invoked with the stack at full capacity, rather than throwing a Full exception (as described in Exercise C-6.16), a more typical semantic is to accept the pushed element at the top while "leaking" the oldest element from the bottom of the stack to make room. Give an implementation of such a LeakyStack abstraction, using a circular array with appropriate storage capacity. This class should have a public interface similar to the bounded-capacity stack in Exercise C-6.16, but with the desired leaky semantics when full.

P-9.55 Write a program that can process a sequence of stock buy and sell orders as described in Exercise C-9.50.

P-9.56 Let S be a set of n points in the plane with distinct integer x- and y-coordinates. Let T be a complete binary tree storing the points from S at its external nodes, such that the points are ordered left to right by increasing x-coordinates. For each node v in T, let $S(v)$ denote the subset of S consisting of points stored in the subtree rooted at v. For the root r of T, define $top(r)$ to be the point in $S = S(r)$ with maximum y-coordinate. For every other node v, define $top(r)$ to be the point in S with highest y-coordinate in $S(v)$ that is not also the highest y-coordinate in $S(u)$, where u is the parent of v in T (if such a point exists). Such labeling turns T into a ***priority search tree***. Describe a linear-time algorithm for turning T into a priority search tree. Implement this approach.

P-9.57 One of the main applications of priority queues is in operating systems—for ***scheduling jobs*** on a CPU. In this project you are to build a program that schedules simulated CPU jobs. Your program should run in a loop, each iteration of which corresponds to a ***time slice*** for the CPU. Each job is assigned a priority, which is an integer between -20 (highest priority) and 19 (lowest priority), inclusive. From among all jobs waiting to be processed in a time slice, the CPU must work on a job with highest priority. In this simulation, each job will also come with a ***length*** value, which is an integer between 1 and 100, inclusive, indicating the number of time slices that are needed to process this job. For simplicity, you may assume jobs cannot be interrupted—once it is scheduled on the CPU, a job runs for a number of time slices equal to its length. Your simulator must output the name of the job running on the CPU in each time slice and must process a sequence of commands, one per time slice, each of which is of the form "add job *name* with length n and priority p" or "no new job this slice".

P-9.58 Develop a Python implementation of an adaptable priority queue that is based on an unsorted list and supports location-aware entries.

Chapter Notes

Knuth's book on sorting and searching [65] describes the motivation and history for the selection-sort, insertion-sort, and heap-sort algorithms. The heap-sort algorithm is due to Williams [103], and the linear-time heap construction algorithm is due to Floyd [39]. Additional algorithms and analyses for heaps and heap-sort variations can be found in papers by Bentley [15], Carlsson [24], Gonnet and Munro [45], McDiarmid and Reed [74], and Schaffer and Sedgewick [88].