Searching for Occurrences of a Value

Each of the count, index, and __contains__ methods proceed through iteration of the sequence from left to right. In fact, Code Fragment 2.14 of Section 2.4.3 demonstrates how those behaviors might be implemented. Notably, the loop for computing the count must proceed through the entire sequence, while the loops for checking containment of an element or determining the index of an element immediately exit once they find the leftmost occurrence of the desired value, if one exists. So while count always examines the n elements of the sequence, index and __contains__ examine n elements in the worst case, but may be faster. Empirical evidence can be found by setting data = list(range(10000000)) and then comparing the relative efficiency of the test, 5 **in** data, relative to the test, 9999995 **in** data, or even the failed test, -5 **in** data.

Lexicographic Comparisons

Comparisons between two sequences are defined lexicographically. In the worst case, evaluating such a condition requires an iteration taking time proportional to the length of the *shorter* of the two sequences (because when one sequence ends, the lexicographic result can be determined). However, in some cases the result of the test can be evaluated more efficiently. For example, if evaluating [7, 3, ...] $<$ [7, 5, ...], it is clear that the result is True without examining the remainders of those lists, because the second element of the left operand is strictly less than the second element of the right operand.

Creating New Instances

The final three behaviors in Table 5.3 are those that construct a new instance based on one or more existing instances. In all cases, the running time depends on the construction and initialization of the new result, and therefore the asymptotic behavior is proportional to the *length* of the result. Therefore, we find that slice data[6000000:6000008] can be constructed almost immediately because it has only eight elements, while slice data[6000000:7000000] has one million elements, and thus is more time-consuming to create.

Mutating Behaviors

The efficiency of the mutating behaviors of the list class are described in Table 5.3. The simplest of those behaviors has syntax data[j] = val, and is supported by the special __setitem__ method. This operation has worst-case $O(1)$ running time because it simply replaces one element of a list with a new value. No other elements are affected and the size of the underlying array does not change. The more interesting behaviors to analyze are those that add or remove elements from the list.