The FIFO strategy is quite simple to implement, as it only requires a queue Q to store references to the pages in the cache. Pages are enqueued in Q when they are referenced by a browser, and then are brought into the cache. When a page needs to be evicted, the computer simply performs a dequeue operation on Q to determine which page to evict. Thus, this policy also requires $O(1)$ additional work per page replacement. Also, the FIFO policy incurs no additional overhead for page requests. Moreover, it tries to take some advantage of temporal locality.

The LRU strategy goes a step further than the FIFO strategy, for the LRU strategy explicitly takes advantage of temporal locality as much as possible, by always evicting the page that was least-recently used. From a policy point of view, this is an excellent approach, but it is costly from an implementation point of view. That is, its way of optimizing temporal and spatial locality is fairly costly. Implementing the LRU strategy requires the use of an adaptable priority queue Q that supports updating the priority of existing pages. If Q is implemented with a sorted sequence based on a linked list, then the overhead for each page request and page replacement is $O(1)$. When we insert a page in Q or update its key, the page is assigned the highest key in Q and is placed at the end of the list, which can also be done in $O(1)$ time. Even though the LRU strategy has constant-time overhead, using the implementation above, the constant factors involved, in terms of the additional time overhead and the extra space for the priority queue Q, make this policy less attractive from a practical point of view.

Since these different page replacement policies have different trade-offs between implementation difficulty and the degree to which they seem to take advantage of localities, it is natural for us to ask for some kind of comparative analysis of these methods to see which one, if any, is the best.

From a worst-case point of view, the FIFO and LRU strategies have fairly unattractive competitive behavior. For example, suppose we have a cache containing m pages, and consider the FIFO and LRU methods for performing page replacement for a program that has a loop that repeatedly requests $m+1$ pages in a cyclic order. Both the FIFO and LRU policies perform badly on such a sequence of page requests, because they perform a page replacement on every page request. Thus, from a worst-case point of view, these policies are almost the worst we can imagine—they require a page replacement on every page request.

This worst-case analysis is a little too pessimistic, however, for it focuses on each protocol's behavior for one bad sequence of page requests. An ideal analysis would be to compare these methods over all possible page-request sequences. Of course, this is impossible to do exhaustively, but there have been a great number of experimental simulations done on page-request sequences derived from real programs. Based on these experimental comparisons, the LRU strategy has been shown to be usually superior to the FIFO strategy, which is usually better than the random strategy.

Page Replacement Algorithms

Some of the better-known page replacement policies include the following (see Figure 15.3):

- **First-in, first-out (FIFO)**: Evict the page that has been in the cache the longest, that is, the page that was transferred to the cache furthest in the past.
- **Least recently used (LRU)**: Evict the page whose last request occurred furthest in the past.

In addition, we can consider a simple and purely random strategy:

- **Random**: Choose a page at random to evict from the cache.

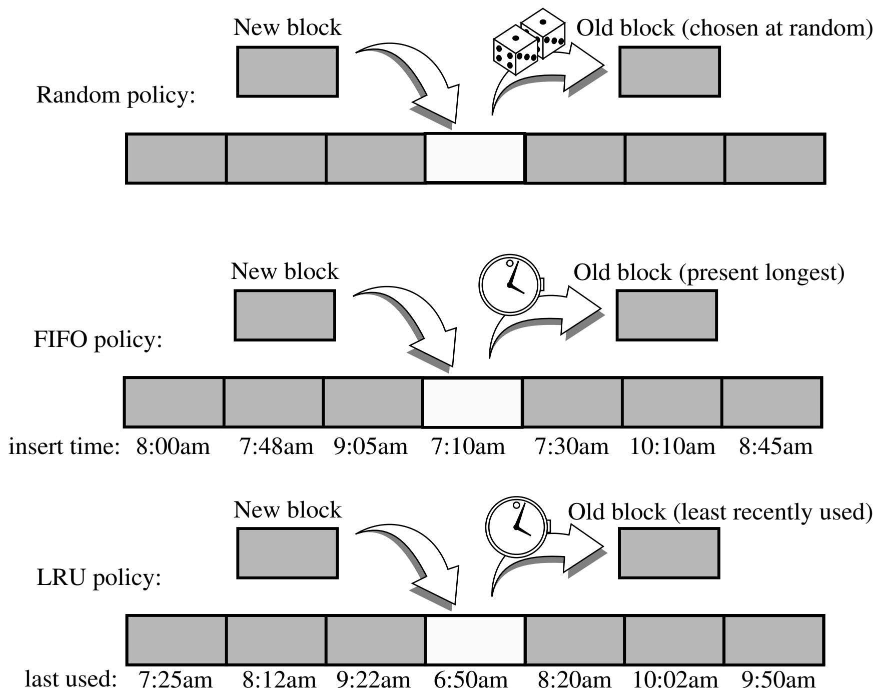
New block
Old block (chosen at random)
Random policy:
New block
Old block (present longest)
FIFO policy:
insert time: 8:00am 7:48am 9:05am 7:10am 7:30am 10:10am 8:45am
New block
Old block (least recently used)
LRU policy:
last used: 7:25am 8:12am 9:22am 6:50am 8:20am 10:02am 9:50am

Figure 15.3: The random, FIFO, and LRU page replacement policies.

The random strategy is one of the easiest policies to implement, for it only requires a random or pseudo-random number generator. The overhead involved in implementing this policy is an $O(1)$ additional amount of work per page replacement. Moreover, there is no additional overhead for each page request, other than to determine whether a page request is in the cache or not. Still, this policy makes no attempt to take advantage of any temporal locality exhibited by a user's browsing.

15.5 Exercises

For help with exercises, please visit the site, www.wiley.com/college/goodrich.

Reinforcement

R-15.1 Julia just bought a new computer that uses 64-bit integers to address memory cells. Argue why Julia will never in her life be able to upgrade the main memory of her computer so that it is the maximum-size possible, assuming that you have to have distinct atoms to represent different bits.

R-15.2 Describe, in detail, algorithms for adding an item to, or deleting an item from, an (a,b) tree.

R-15.3 Suppose T is a multiway tree in which each internal node has at least five and at most eight children. For what values of a and b is T a valid (a,b) tree?

R-15.4 For what values of d is the tree T of the previous exercise an order-d B-tree?

R-15.5 Consider an initially empty memory cache consisting of four pages. How many page misses does the LRU algorithm incur on the following page request sequence: $(2,3,4,1,2,5,1,3,5,4,1,2,3)$?

R-15.6 Consider an initially empty memory cache consisting of four pages. How many page misses does the FIFO algorithm incur on the following page request sequence: $(2,3,4,1,2,5,1,3,5,4,1,2,3)$?

R-15.7 Consider an initially empty memory cache consisting of four pages. What is the maximum number of page misses that the random algorithm incurs on the following page request sequence: $(2,3,4,1,2,5,1,3,5,4,1,2,3)$? Show all of the random choices the algorithm made in this case.

R-15.8 Draw the result of inserting, into an initially empty order-7 B-tree, entries with keys $(4,40,23,50,11,34,62,78,66,22,90,59,25,72,64,77,39,12)$, in this order.

Creativity

C-15.9 Describe an efficient external-memory algorithm for removing all the duplicate entries in an array list of size n.

C-15.10 Describe an external-memory data structure to implement the stack ADT so that the total number of disk transfers needed to process a sequence of k push and pop operations is $O(k/B)$.

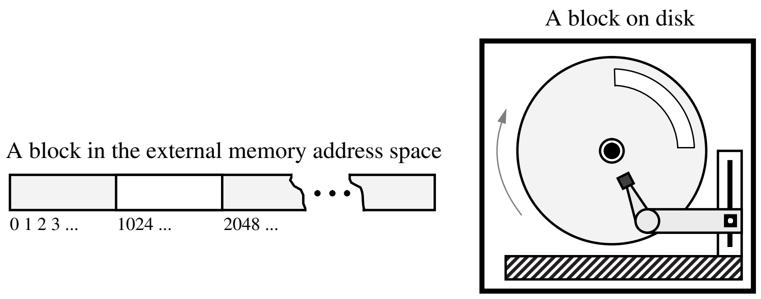

Figure 15.2: Blocks in external memory.

When implemented with caching and blocking, virtual memory often allows us to perceive secondary-level memory as being faster than it really is. There is still a problem, however. Primary-level memory is much smaller than secondary-level memory. Moreover, because memory systems use blocking, any program of substance will likely reach a point where it requests data from secondary-level memory, but the primary memory is already full of blocks. In order to fulfill the request and maintain our use of caching and blocking, we must remove some block from primary memory to make room for a new block from secondary memory in this case. Deciding which block to evict brings up a number of interesting data structure and algorithm design issues.

Caching in Web Browsers

For motivation, we consider a related problem that arises when revisiting information presented in Web pages. To exploit temporal locality of reference, it is often advantageous to store copies of Web pages in a ***cache*** memory, so these pages can be quickly retrieved when requested again. This effectively creates a two-level memory hierarchy, with the cache serving as the smaller, quicker internal memory, and the network being the external memory. In particular, suppose we have a cache memory that has m "slots" that can contain Web pages. We assume that a Web page can be placed in any slot of the cache. This is known as a ***fully associative*** cache.

As a browser executes, it requests different Web pages. Each time the browser requests such a Web page p, the browser determines (using a quick test) if p is unchanged and currently contained in the cache. If p is contained in the cache, then the browser satisfies the request using the cached copy. If p is not in the cache, however, the page for p is requested over the Internet and transferred into the cache. If one of the m slots in the cache is available, then the browser assigns p to one of the empty slots. But if all the m cells of the cache are occupied, then the computer must determine which previously viewed Web page to evict before bringing in p to take its place. There are, of course, many different policies that can be used to determine the page to evict.

C-15.20 Consider the page caching strategy based on the ***least frequently used*** (LFU) rule, where the page in the cache that has been accessed the least often is the one that is evicted when a new page is requested. If there are ties, LFU evicts the least frequently used page that has been in the cache the longest. Show that there is a sequence P of n requests that causes LFU to miss $\Omega(n)$ times for a cache of m pages, whereas the optimal algorithm will miss only $O(m)$ times.

C-15.21 Suppose that instead of having the node-search function $f(d) = 1$ in an order-d B-tree T, we have $f(d) = \log d$. What does the asymptotic running time of performing a search in T now become?

Projects

P-15.22 Write a Python class that simulates the best-fit, worst-fit, first-fit, and next-fit algorithms for memory management. Determine experimentally which method is the best under various sequences of memory requests.

P-15.23 Write a Python class that implements all the methods of the ordered map ADT by means of an (a,b) tree, where a and b are integer constants passed as parameters to a constructor.

P-15.24 Implement the B-tree data structure, assuming a block size of 1024 and integer keys. Test the number of "disk transfers" needed to process a sequence of map operations.

Chapter Notes

The reader interested in the study of the architecture of hierarchical memory systems is referred to the book chapter by Burger *et al.* [21] or the book by Hennessy and Patterson [50]. The mark-sweep garbage collection method we describe is one of many different algorithms for performing garbage collection. We encourage the reader interested in further study of garbage collection to examine the book by Jones and Lins [56]. Knuth [62] has very nice discussions about external-memory sorting and searching, and Ullman [97] discusses external memory structures for database systems. The handbook by Gonnet and Baeza-Yates [44] compares the performance of a number of different sorting algorithms, many of which are external-memory algorithms. B-trees were invented by Bayer and McCreight [11] and Comer [28] provides a very nice overview of this data structure. The books by Mehlhorn [76] and Samet [87] also have nice discussions about B-trees and their variants. Aggarwal and Vitter [3] study the I/O complexity of sorting and related problems, establishing upper and lower bounds. Goodrich *et al.* [46] study the I/O complexity of several computational geometry problems. The reader interested in further study of I/O-efficient algorithms is encouraged to examine the survey paper of Vitter [99].

Creativity

C-1.13 Write a pseudo-code description of a function that reverses a list of n integers, so that the numbers are listed in the opposite order than they were before, and compare this method to an equivalent Python function for doing the same thing.

C-1.14 Write a short Python function that takes a sequence of integer values and determines if there is a distinct pair of numbers in the sequence whose product is odd.

C-1.15 Write a Python function that takes a sequence of numbers and determines if all the numbers are different from each other (that is, they are distinct).

C-1.16 In our implementation of the scale function (page 25), the body of the loop executes the command data[j] *= factor. We have discussed that numeric types are immutable, and that use of the *= operator in this context causes the creation of a new instance (not the mutation of an existing instance). How is it still possible, then, that our implementation of scale changes the actual parameter sent by the caller?

C-1.17 Had we implemented the scale function (page 25) as follows, does it work properly?

```
def scale(data, factor):
  for val in data:
    val *= factor
```

Explain why or why not.

C-1.18 Demonstrate how to use Python's list comprehension syntax to produce the list [0, 2, 6, 12, 20, 30, 42, 56, 72, 90].

C-1.19 Demonstrate how to use Python's list comprehension syntax to produce the list ['a', 'b', 'c', ..., 'z'], but without having to type all 26 such characters literally.

C-1.20 Python's random module includes a function shuffle(data) that accepts a list of elements and randomly reorders the elements so that each possible order occurs with equal probability. The random module includes a more basic function randint(a, b) that returns a uniformly random integer from a to b (including both endpoints). Using only the randint function, implement your own version of the shuffle function.

C-1.21 Write a Python program that repeatedly reads lines from standard input until an EOFError is raised, and then outputs those lines in reverse order (a user can indicate end of input by typing ctrl-D).

Documentation

Python provides integrated support for embedding formal documentation directly in source code using a mechanism known as a ***docstring***. Formally, any string literal that appears as the *first* statement within the body of a module, class, or function (including a member function of a class) will be considered to be a docstring. By convention, those string literals should be delimited within triple quotes ("""). As an example, our version of the scale function from page 25 could be documented as follows:

```
def scale(data, factor):
  """Multiply all entries of numeric data list by the given factor."""
  for j in range(len(data)):
    data[j] *= factor
```

It is common to use the triple-quoted string delimiter for a docstring, even when the string fits on a single line, as in the above example. More detailed docstrings should begin with a single line that summarizes the purpose, followed by a blank line, and then further details. For example, we might more clearly document the scale function as follows:

```
def scale(data, factor):
  """Multiply all entries of numeric data list by the given factor.

  data      an instance of any mutable sequence type (such as a list)
            containing numeric elements

  factor    a number that serves as the multiplicative factor for scaling
  """
  for j in range(len(data)):
    data[j] *= factor
```

A docstring is stored as a field of the module, function, or class in which it is declared. It serves as documentation and can be retrieved in a variety of ways. For example, the command help(x), within the Python interpreter, produces the documentation associated with the identified object x. An external tool named `pydoc` is distributed with Python and can be used to generate formal documentation as text or as a Web page. Guidelines for *authoring* useful docstrings are available at:

`http://www.python.org/dev/peps/pep-0257/`

In this book, we will try to present docstrings when space allows. Omitted docstrings can be found in the online version of our source code.

Revisiting the Problem of Finding the Maximum of a Sequence

For our next example, we revisit the find_max algorithm, given in Code Fragment 3.1 on page 123, for finding the largest value in a sequence. Proposition 3.7 on page 125 claimed an $O(n)$ run-time for the find_max algorithm. Consistent with our earlier analysis of syntax data[0], the initialization uses $O(1)$ time. The loop executes n times, and within each iteration, it performs one comparison and possibly one assignment statement (as well as maintenance of the loop variable). Finally, we note that the mechanism for enacting a **return** statement in Python uses $O(1)$ time. Combining these steps, we have that the find_max function runs in $O(n)$ time.

Further Analysis of the Maximum-Finding Algorithm

A more interesting question about find_max is how many times we might update the current "biggest" value. In the worst case, if the data is given to us in increasing order, the biggest value is reassigned $n-1$ times. But what if the input is given to us in random order, with all orders equally likely; what would be the expected number of times we update the biggest value in this case? To answer this question, note that we update the current biggest in an iteration of the loop only if the current element is bigger than all the elements that precede it. If the sequence is given to us in random order, the probability that the j^{th} element is the largest of the first j elements is $1/j$ (assuming uniqueness). Hence, the expected number of times we update the biggest (including initialization) is $H_n = \sum_{j=1}^{n} 1/j$, which is known as the n^{th} ***Harmonic number***. It turns out (see Proposition B.16) that H_n is $O(\log n)$. Therefore, the expected number of times the biggest value is updated by find_max on a randomly ordered sequence is $O(\log n)$.

Prefix Averages

The next problem we consider is computing what are known as ***prefix averages*** of a sequence of numbers. Namely, given a sequence S consisting of n numbers, we want to compute a sequence A such that $A[j]$ is the average of elements $S[0],\ldots,S[j]$, for $j=0,\ldots,n-1$, that is,

$$A[j] = \frac{\sum_{i=0}^{j} S[i]}{j+1}.$$

Computing prefix averages has many applications in economics and statistics. For example, given the year-by-year returns of a mutual fund, ordered from recent to past, an investor will typically want to see the fund's average annual returns for the most recent year, the most recent three years, the most recent five years, and so on. Likewise, given a stream of daily Web usage logs, a Web site manager may wish to track average usage trends over various time periods. We analyze three different implementations that solve this problem but with rather different running times.

C-5.18 Give a formal proof that any sequence of n append or pop operations on an initially empty dynamic array takes $O(n)$ time, if using the strategy described in Exercise C-5.16.

C-5.19 Consider a variant of Exercise C-5.16, in which an array of capacity N is resized to capacity precisely that of the number of elements, any time the number of elements in the array goes strictly below $N/4$. Give a formal proof that any sequence of n append or pop operations on an initially empty dynamic array takes $O(n)$ time.

C-5.20 Consider a variant of Exercise C-5.16, in which an array of capacity N, is resized to capacity precisely that of the number of elements, any time the number of elements in the array goes strictly below $N/2$. Show that there exists a sequence of n operations that requires $\Omega(n^2)$ time to execute.

C-5.21 In Section 5.4.2, we described four different ways to compose a long string: (1) repeated concatenation, (2) appending to a temporary list and then joining, (3) using list comprehension with join, and (4) using generator comprehension with join. Develop an experiment to test the efficiency of all four of these approaches and report your findings.

C-5.22 Develop an experiment to compare the relative efficiency of the extend method of Python's list class versus using repeated calls to append to accomplish the equivalent task.

C-5.23 Based on the discussion of page 207, develop an experiment to compare the efficiency of Python's list comprehension syntax versus the construction of a list by means of repeated calls to append.

C-5.24 Perform experiments to evaluate the efficiency of the remove method of Python's list class, as we did for insert on page 205. Use known values so that all removals occur either at the beginning, middle, or end of the list. Report your results akin to Table 5.5.

C-5.25 The syntax data.remove(value) for Python list data removes only the first occurrence of element value from the list. Give an implementation of a function, with signature remove_all(data, value), that removes *all* occurrences of value from the given list, such that the worst-case running time of the function is $O(n)$ on a list with n elements. Not that it is not efficient enough in general to rely on repeated calls to remove.

C-5.26 Let B be an array of size $n \geq 6$ containing integers from 1 to $n-5$, inclusive, with exactly five repeated. Describe a good algorithm for finding the five integers in B that are repeated.

C-5.27 Given a Python list L of n positive integers, each represented with $k = \lceil \log n \rceil + 1$ bits, describe an $O(n)$-time method for finding a k-bit integer not in L.

C-5.28 Argue why any solution to the previous problem must run in $\Omega(n)$ time.

10.3.1 Sorted Search Tables

Several data structures can efficiently support the sorted map ADT, and we will examine some advanced techniques in Section 10.4 and Chapter 11. In this section, we begin by exploring a simple implementation of a sorted map. We store the map's items in an array-based sequence A so that they are in increasing order of their keys, assuming the keys have a naturally defined order. (See Figure 10.8.) We refer to this implementation of a map as a ***sorted search table***.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Figure 10.8: Realization of a map by means of a sorted search table. We show only the keys for this map, so as to highlight their ordering.

As was the case with the unsorted table map of Section 10.1.5, the sorted search table has a space requirement that is $O(n)$, assuming we grow and shrink the array to keep its size proportional to the number of items in the map. The primary advantage of this representation, and our reason for insisting that A be array-based, is that it allows us to use the ***binary search*** algorithm for a variety of efficient operations.

Binary Search and Inexact Searches

We originally presented the binary search algorithm in Section 4.1.3, as a means for detecting whether a given target is stored within a sorted sequence. In our original presentation (Code Fragment 4.3 on page 156), a binary_search function returned True of False to designate whether the desired target was found. While such an approach could be used to implement the __contains__ method of the map ADT, we can adapt the binary search algorithm to provide far more useful information when performing forms of inexact search in support of the sorted map ADT.

The important realization is that while performing a binary search, we can determine the index at or near where a target might be found. During a successful search, the standard implementation determines the precise index at which the target is found. During an unsuccessful search, although the target is not found, the algorithm will effectively determine a pair of indices designating elements of the collection that are just less than or just greater than the missing target.

As a motivating example, our original simulation from Figure 4.5 on page 156 shows a successful binary search for a target of 22, using the same data we portray in Figure 10.8. Had we instead been searching for 21, the first four steps of the algorithm would be the same. The subsequent difference is that we would make an additional call with inverted parameters high=9 and low=10, effectively concluding that the missing target lies in the gap between values 19 and 22 in that example.

C-15.11 Describe an external-memory data structure to implement the queue ADT so that the total number of disk transfers needed to process a sequence of k enqueue and dequeue operations is $O(k/B)$.

C-15.12 Describe an external-memory version of the PositionalList ADT (Section 7.4), with block size B, such that an iteration of a list of length n is completed using $O(n/B)$ transfers in the worst case, and all other methods of the ADT require only $O(1)$ transfers.

C-15.13 Change the rules that define red-black trees so that each red-black tree T has a corresponding $(4,8)$ tree, and vice versa.

C-15.14 Describe a modified version of the B-tree insertion algorithm so that each time we create an overflow because of a split of a node w, we redistribute keys among all of w's siblings, so that each sibling holds roughly the same number of keys (possibly cascading the split up to the parent of w). What is the minimum fraction of each block that will always be filled using this scheme?

C-15.15 Another possible external-memory map implementation is to use a skip list, but to collect consecutive groups of $O(B)$ nodes, in individual blocks, on any level in the skip list. In particular, we define an ***order-d B-skip list*** to be such a representation of a skip list structure, where each block contains at least $\lceil d/2 \rceil$ list nodes and at most d list nodes. Let us also choose d in this case to be the maximum number of list nodes from a level of a skip list that can fit into one block. Describe how we should modify the skip-list insertion and removal algorithms for a B-skip list so that the expected height of the structure is $O(\log n / \log B)$.

C-15.16 Describe how to use a B-tree to implement the partition (union-find) ADT (from Section 14.7.3) so that the union and find operations each use at most $O(\log n / \log B)$ disk transfers.

C-15.17 Suppose we are given a sequence S of n elements with integer keys such that some elements in S are colored "blue" and some elements in S are colored "red." In addition, say that a red element e ***pairs*** with a blue element f if they have the same key value. Describe an efficient external-memory algorithm for finding all the red-blue pairs in S. How many disk transfers does your algorithm perform?

C-15.18 Consider the page caching problem where the memory cache can hold m pages, and we are given a sequence P of n requests taken from a pool of $m+1$ possible pages. Describe the optimal strategy for the offline algorithm and show that it causes at most $m + n/m$ page misses in total, starting from an empty cache.

C-15.19 Describe an efficient external-memory algorithm that determines whether an array of n integers contains a value occurring more than $n/2$ times.

1.4 Control Flow

In this section, we review Python's most fundamental control structures: conditional statements and loops. Common to all control structures is the syntax used in Python for defining blocks of code. The colon character is used to delimit the beginning of a block of code that acts as a body for a control structure. If the body can be stated as a single executable statement, it can technically placed on the same line, to the right of the colon. However, a body is more typically typeset as an ***indented block*** starting on the line following the colon. Python relies on the indentation level to designate the extent of that block of code, or any nested blocks of code within. The same principles will be applied when designating the body of a function (see Section 1.5), and the body of a class (see Section 2.3).

1.4.1 Conditionals

Conditional constructs (also known as **if** statements) provide a way to execute a chosen block of code based on the run-time evaluation of one or more Boolean expressions. In Python, the most general form of a conditional is written as follows:

```
if first_condition:
  first_body
elif second_condition:
  second_body
elif third_condition:
  third_body
else:
  fourth_body
```

Each condition is a Boolean expression, and each body contains one or more commands that are to be executed conditionally. If the first condition succeeds, the first body will be executed; no other conditions or bodies are evaluated in that case. If the first condition fails, then the process continues in similar manner with the evaluation of the second condition. The execution of this overall construct will cause precisely one of the bodies to be executed. There may be any number of **elif** clauses (including zero), and the final **else** clause is optional. As described on page 7, nonboolean types may be evaluated as Booleans with intuitive meanings. For example, if response is a string that was entered by a user, and we want to condition a behavior on this being a nonempty string, we may write

```
if response:
```

as a shorthand for the equivalent,

```
if response != '':
```

1.4.2 Loops

Python offers two distinct looping constructs. A **while** loop allows general repetition based upon the repeated testing of a Boolean condition. A **for** loop provides convenient iteration of values from a defined series (such as characters of a string, elements of a list, or numbers within a given range). We discuss both forms in this section.

While Loops

The syntax for a **while** loop in Python is as follows:

```
while condition:
  body
```

As with an **if** statement, *condition* can be an arbitrary Boolean expression, and *body* can be an arbitrary block of code (including nested control structures). The execution of a while loop begins with a test of the Boolean condition. If that condition evaluates to True, the body of the loop is performed. After each execution of the body, the loop condition is retested, and if it evaluates to True, another iteration of the body is performed. When the conditional test evaluates to False (assuming it ever does), the loop is exited and the flow of control continues just beyond the body of the loop.

As an example, here is a loop that advances an index through a sequence of characters until finding an entry with value 'X' or reaching the end of the sequence.

```
j = 0
while j < len(data) and data[j] != 'X':
  j += 1
```

The len function, which we will introduce in Section 1.5.2, returns the length of a sequence such as a list or string. The correctness of this loop relies on the short-circuiting behavior of the **and** operator, as described on page 12. We intentionally test j < len(data) to ensure that j is a valid index, prior to accessing element data[j]. Had we written that compound condition with the opposite order, the evaluation of data[j] would eventually raise an IndexError when 'X' is not found. (See Section 1.7 for discussion of exceptions.)

As written, when this loop terminates, variable j's value will be the index of the leftmost occurrence of 'X', if found, or otherwise the length of the sequence (which is recognizable as an invalid index to indicate failure of the search). It is worth noting that this code behaves correctly, even in the special case when the list is empty, as the condition j < len(data) will initially fail and the body of the loop will never be executed.

Return Statement

A **return** statement is used within the body of a function to indicate that the function should immediately cease execution, and that an expressed value should be returned to the caller. If a return statement is executed without an explicit argument, the None value is automatically returned. Likewise, None will be returned if the flow of control ever reaches the end of a function body without having executed a return statement. Often, a return statement will be the final command within the body of the function, as was the case in our earlier example of a count function. However, there can be multiple return statements in the same function, with conditional logic controlling which such command is executed, if any. As a further example, consider the following function that tests if a value exists in a sequence.

```
def contains(data, target):
  for item in target:
    if item == target:                    # found a match
      return True
  return False
```

If the conditional within the loop body is ever satisfied, the return True statement is executed and the function immediately ends, with True designating that the target value was found. Conversely, if the for loop reaches its conclusion without ever finding the match, the final return False statement will be executed.

1.5.1 Information Passing

To be a successful programmer, one must have clear understanding of the mechanism in which a programming language passes information to and from a function. In the context of a function signature, the identifiers used to describe the expected parameters are known as ***formal parameters***, and the objects sent by the caller when invoking the function are the ***actual parameters***. Parameter passing in Python follows the semantics of the standard ***assignment statement***. When a function is invoked, each identifier that serves as a formal parameter is assigned, in the function's local scope, to the respective actual parameter that is provided by the caller of the function.

For example, consider the following call to our count function from page 23:

```
prizes = count(grades, 'A')
```

Just before the function body is executed, the actual parameters, grades and 'A', are implicitly assigned to the formal parameters, data and target, as follows:

```
data = grades
target = 'A'
```

Simultaneous Assignments

The combination of automatic packing and unpacking forms a technique known as ***simultaneous assignment***, whereby we explicitly assign a series of values to a series of identifiers, using a syntax:

```
x, y, z = 6, 2, 5
```

In effect, the right-hand side of this assignment is automatically packed into a tuple, and then automatically unpacked with its elements assigned to the three identifiers on the left-hand side.

When using a simultaneous assignment, all of the expressions are evaluated on the right-hand side before any of the assignments are made to the left-hand variables. This is significant, as it provides a convenient means for swapping the values associated with two variables:

```
j, k = k, j
```

With this command, j will be assigned to the *old* value of k, and k will be assigned to the *old* value of j. Without simultaneous assignment, a swap typically requires more delicate use of a temporary variable, such as

```
temp = j
j = k
k = temp
```

With the simultaneous assignment, the unnamed tuple representing the packed values on the right-hand side implicitly serves as the temporary variable when performing such a swap.

The use of simultaneous assignments can greatly simplify the presentation of code. As an example, we reconsider the generator on page 41 that produces the Fibonacci series. The original code requires separate initialization of variables a and b to begin the series. Within each pass of the loop, the goal was to reassign a and b, respectively, to the values of b and a+b. At the time, we accomplished this with brief use of a third variable. With simultaneous assignments, that generator can be implemented more directly as follows:

```
def fibonacci():
  a, b = 0, 1
  while True:
    yield a
    a, b = b, a+b
```

When an identifier is indicated in a command, Python searches a series of namespaces in the process of name resolution. First, the most locally enclosing scope is searched for a given name. If not found there, the next outer scope is searched, and so on. We will continue our examination of namespaces, in Section 2.5, when discussing Python's treatment of object-orientation. We will see that each object has its own namespace to store its attributes, and that classes each have a namespace as well.

First-Class Objects

In the terminology of programming languages, ***first-class objects*** are instances of a type that can be assigned to an identifier, passed as a parameter, or returned by a function. All of the data types we introduced in Section 1.2.3, such as int and list, are clearly first-class types in Python. In Python, functions and classes are also treated as first-class objects. For example, we could write the following:

```
scream = print          # assign name 'scream' to the function denoted as 'print'
scream('Hello')         # call that function
```

In this case, we have not created a new function, we have simply defined scream as an alias for the existing print function. While there is little motivation for precisely this example, it demonstrates the mechanism that is used by Python to allow one function to be passed as a parameter to another. On page 28, we noted that the built-in function, max, accepts an optional keyword parameter to specify a non-default order when computing a maximum. For example, a caller can use the syntax, max(a, b, key=abs), to determine which value has the larger absolute value. Within the body of that function, the formal parameter, key, is an identifier that will be assigned to the actual parameter, abs.

In terms of namespaces, an assignment such as scream = print, introduces the identifier, scream, into the current namespace, with its value being the object that represents the built-in function, print. The same mechanism is applied when a user-defined function is declared. For example, our count function from Section 1.5 beings with the following syntax:

```
def count(data, target):
  ...
```

Such a declaration introduces the identifier, count, into the current namespace, with the value being a function instance representing its implementation. In similar fashion, the name of a newly defined class is associated with a representation of that class as its value. (Class definitions will be introduced in the next chapter.)

C-1.22 Write a short Python program that takes two arrays a and b of length n storing **int** values, and returns the dot product of a and b. That is, it returns an array c of length n such that $c[i] = a[i] \cdot b[i]$, for $i = 0, \ldots, n-1$.

C-1.23 Give an example of a Python code fragment that attempts to write an element to a list based on an index that may be out of bounds. If that index is out of bounds, the program should catch the exception that results, and print the following error message:
"`Don't try buffer overflow attacks in Python!`"

C-1.24 Write a short Python function that counts the number of vowels in a given character string.

C-1.25 Write a short Python function that takes a string s, representing a sentence, and returns a copy of the string with all punctuation removed. For example, if given the string `"Let's try, Mike."`, this function would return `"Lets try Mike"`.

C-1.26 Write a short program that takes as input three integers, a, b, and c, from the console and determines if they can be used in a correct arithmetic formula (in the given order), like "$a+b=c$," "$a=b-c$," or "$a*b=c$."

C-1.27 In Section 1.8, we provided three different implementations of a generator that computes factors of a given integer. The third of those implementations, from page 41, was the most efficient, but we noted that it did not yield the factors in increasing order. Modify the generator so that it reports factors in increasing order, while maintaining its general performance advantages.

C-1.28 The ***p-norm*** of a vector $v = (v_1, v_2, \ldots, v_n)$ in n-dimensional space is defined as

$$\|v\| = \sqrt[p]{v_1^p + v_2^p + \cdots + v_n^p}.$$

For the special case of $p = 2$, this results in the traditional ***Euclidean norm***, which represents the length of the vector. For example, the Euclidean norm of a two-dimensional vector with coordinates $(4,3)$ has a Euclidean norm of $\sqrt{4^2+3^2} = \sqrt{16+9} = \sqrt{25} = 5$. Give an implementation of a function named `norm` such that `norm(v, p)` returns the p-norm value of v and `norm(v)` returns the Euclidean norm of v. You may assume that v is a list of numbers.

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.

A Geometric Progression Class

Our second example of a specialized progression is a geometric progression, in which each value is produced by multiplying the preceding value by a fixed constant, known as the ***base*** of the geometric progression. The starting point of a geometric progression is traditionally 1, rather than 0, because multiplying 0 by any factor results in 0. As an example, a geometric progression with base 2 proceeds as $1, 2, 4, 8, 16, \ldots$.

Code Fragment 2.10 presents our implementation of a GeometricProgression class. The constructor uses 2 as a default base and 1 as a default starting value, but either of those can be varied using optional parameters.

```
class GeometricProgression(Progression):            # inherit from Progression
  """Iterator producing a geometric progression."""

  def __init__(self, base=2, start=1):
    """Create a new geometric progression.

    base          the fixed constant multiplied to each term (default 2)
    start         the first term of the progression (default 1)
    """
    super().__init__(start)
    self._base = base

  def _advance(self):                                # override inherited version
    """Update current value by multiplying it by the base value."""
    self._current *= self._base
```

Code Fragment 2.10: A class that produces a geometric progression.

A Fibonacci Progression Class

As our final example, we demonstrate how to use our progression framework to produce a ***Fibonacci progression***. We originally discussed the Fibonacci series on page 41 in the context of generators. Each value of a Fibonacci series is the sum of the two most recent values. To begin the series, the first two values are conventionally 0 and 1, leading to the Fibonacci series $0, 1, 1, 2, 3, 5, 8, \ldots$. More generally, such a series can be generated from any two starting values. For example, if we start with values 4 and 6, the series proceeds as $4, 6, 10, 16, 26, 42, \ldots$.

```
class FibonacciProgression(Progression):
  """Iterator producing a generalized Fibonacci progression."""

  def __init__(self, first=0, second=1):
    """Create a new fibonacci progression.

    first         the first term of the progression (default 0)
    second        the second term of the progression (default 1)
    """
    super().__init__(first)                 # start progression at first
    self._prev = second − first             # fictitious value preceding the first

  def _advance(self):
    """Update current value by taking sum of previous two."""
    self._prev, self._current = self._current, self._prev + self._current
```

Code Fragment 2.11: A class that produces a Fibonacci progression.

We use our progression framework to define a new FibonacciProgression class, as shown in Code Fragment 2.11. This class is markedly different from those for the arithmetic and geometric progressions because we cannot determine the next value of a Fibonacci series solely from the current one. We must maintain knowledge of the two most recent values. The base Progression class already provides storage of the most recent value as the _current data member. Our FibonacciProgression class introduces a new member, named _prev, to store the value that proceeded the current one.

With both previous values stored, the implementation of _advance is relatively straightforward. (We use a simultaneous assignment similar to that on page 45.) However, the question arises as to how to initialize the previous value in the constructor. The desired first and second values are provided as parameters to the constructor. The first should be stored as _current so that it becomes the first one that is reported. Looking ahead, once the first value is reported, we will do an assignment to set the new current value (which will be the second value reported), equal to the first value plus the "previous." By initializing the previous value to (second − first), the initial advancement will set the new current value to first + (second − first) = second, as desired.

Testing Our Progressions

To complete our presentation, Code Fragment 2.12 provides a unit test for all of our progression classes, and Code Fragment 2.13 shows the output of that test.

2.6 Shallow and Deep Copying

In Chapter 1, we emphasized that an assignment statement foo = bar makes the name foo an ***alias*** for the object identified as bar. In this section, we consider the task of making a ***copy*** of an object, rather than an alias. This is necessary in applications when we want to subsequently modify either the original or the copy in an independent manner.

Consider a scenario in which we manage various lists of colors, with each color represented by an instance of a presumed color class. We let identifier warmtones denote an existing list of such colors (e.g., oranges, browns). In this application, we wish to create a new list named palette, which is a copy of the warmtones list. However, we want to subsequently be able to add additional colors to palette, or to modify or remove some of the existing colors, without affecting the contents of warmtones. If we were to execute the command

```
palette = warmtones
```

this creates an alias, as shown in Figure 2.9. No new list is created; instead, the new identifier palette references the original list.

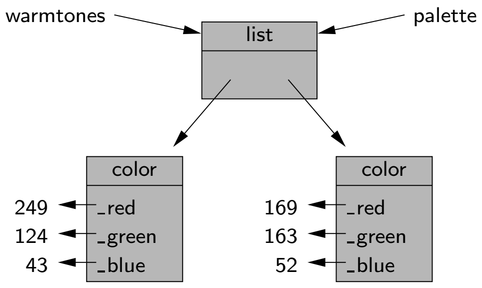

Figure 2.9: Two aliases for the same list of colors.

Unfortunately, this does not meet our desired criteria, because if we subsequently add or remove colors from "palette," we modify the list identified as warmtones.

We can instead create a new instance of the list class by using the syntax:

```
palette = list(warmtones)
```

In this case, we explicitly call the list constructor, sending the first list as a parameter. This causes a new list to be created, as shown in Figure 2.10; however, it is what is known as a ***shallow copy***. The new list is initialized so that its contents are precisely the same as the original sequence. However, Python's lists are ***referential*** (see page 9 of Section 1.2.3), and so the new list represents a sequence of references to the same elements as in the first.

We note that most handheld calculators have a button marked LOG, but this is typically for calculating the logarithm base-10, not base-two.

Computing the logarithm function exactly for any integer n involves the use of calculus, but we can use an approximation that is good enough for our purposes without calculus. In particular, we can easily compute the smallest integer greater than or equal to $\log_b n$ (its so-called ***ceiling***, $\lceil \log_b n \rceil$). For positive integer, n, this value is equal to the number of times we can divide n by b before we get a number less than or equal to 1. For example, the evaluation of $\lceil \log_3 27 \rceil$ is 3, because $((27/3)/3)/3 = 1$. Likewise, $\lceil \log_4 64 \rceil$ is 3, because $((64/4)/4)/4 = 1$, and $\lceil \log_2 12 \rceil$ is 4, because $(((12/2)/2)/2)/2 = 0.75 \leq 1$.

The following proposition describes several important identities that involve logarithms for any base greater than 1.

Proposition 3.1 (Logarithm Rules): *Given real numbers $a > 0$, $b > 1$, $c > 0$ and $d > 1$, we have:*

1. $\log_b(ac) = \log_b a + \log_b c$
2. $\log_b(a/c) = \log_b a - \log_b c$
3. $\log_b(a^c) = c \log_b a$
4. $\log_b a = \log_d a / \log_d b$
5. $b^{\log_d a} = a^{\log_d b}$

By convention, the unparenthesized notation $\log n^c$ denotes the value $\log(n^c)$. We use a notational shorthand, $\log^c n$, to denote the quantity, $(\log n)^c$, in which the result of the logarithm is raised to a power.

The above identities can be derived from converse rules for exponentiation that we will present on page 121. We illustrate these identities with a few examples.

Example 3.2: *We demonstrate below some interesting applications of the logarithm rules from Proposition 3.1 (using the usual convention that the base of a logarithm is 2 if it is omitted).*

- $\log(2n) = \log 2 + \log n = 1 + \log n$, *by rule 1*
- $\log(n/2) = \log n - \log 2 = \log n - 1$, *by rule 2*
- $\log n^3 = 3 \log n$, *by rule 3*
- $\log 2^n = n \log 2 = n \cdot 1 = n$, *by rule 3*
- $\log_4 n = (\log n)/\log 4 = (\log n)/2$, *by rule 4*
- $2^{\log n} = n^{\log 2} = n^1 = n$, *by rule 5.*

As a practical matter, we note that rule 4 gives us a way to compute the base-two logarithm on a calculator that has a base-10 logarithm button, LOG, *for*

$$\log_2 n = \mathsf{LOG}\, n / \mathsf{LOG}\, 2.$$

3.3 Asymptotic Analysis

In algorithm analysis, we focus on the growth rate of the running time as a function of the input size n, taking a "big-picture" approach. For example, it is often enough just to know that the running time of an algorithm ***grows proportionally to*** n.

We analyze algorithms using a mathematical notation for functions that disregards constant factors. Namely, we characterize the running times of algorithms by using functions that map the size of the input, n, to values that correspond to the main factor that determines the growth rate in terms of n. This approach reflects that each basic step in a pseudo-code description or a high-level language implementation may correspond to a small number of primitive operations. Thus, we can perform an analysis of an algorithm by estimating the number of primitive operations executed up to a constant factor, rather than getting bogged down in language-specific or hardware-specific analysis of the exact number of operations that execute on the computer.

As a tangible example, we revisit the goal of finding the largest element of a Python list; we first used this example when introducing for loops on page 21 of Section 1.4.2. Code Fragment 3.1 presents a function named find_max for this task.

```
def find_max(data):
  """Return the maximum element from a nonempty Python list."""
  biggest = data[0]                        # The initial value to beat
  for val in data:                         # For each value:
    if val > biggest                       # if it is greater than the best so far,
      biggest = val                        # we have found a new best (so far)
  return biggest                           # When loop ends, biggest is the max
```

Code Fragment 3.1: A function that returns the maximum value of a Python list.

This is a classic example of an algorithm with a running time that grows proportional to n, as the loop executes once for each data element, with some fixed number of primitive operations executing for each pass. In the remainder of this section, we provide a framework to formalize this claim.

3.3.1 The "Big-Oh" Notation

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq cg(n), \quad \text{for} \quad n \geq n_0.$$

This definition is often referred to as the "big-Oh" notation, for it is sometimes pronounced as "$f(n)$ is ***big-Oh*** of $g(n)$." Figure 3.5 illustrates the general definition.

In general, an interval with a central tick length $L \geq 1$ is composed of:

- An interval with a central tick length $L-1$
- A single tick of length L
- An interval with a central tick length $L-1$

Although it is possible to draw such a ruler using an iterative process (see Exercise P-4.25), the task is considerably easier to accomplish with recursion. Our implementation consists of three functions, as shown in Code Fragment 4.2. The main function, draw_ruler, manages the construction of the entire ruler. Its arguments specify the total number of inches in the ruler and the major tick length. The utility function, draw_line, draws a single tick with a specified number of dashes (and an optional string label, that is printed after the tick).

The interesting work is done by the recursive draw_interval function. This function draws the sequence of minor ticks within some interval, based upon the length of the interval's central tick. We rely on the intuition shown at the top of this page, and with a base case when $L = 0$ that draws nothing. For $L \geq 1$, the first and last steps are performed by recursively calling draw_interval$(L-1)$. The middle step is performed by calling the function draw_line(L).

```
def draw_line(tick_length, tick_label=''):
  """Draw one line with given tick length (followed by optional label)."""
  line = '-' * tick_length
  if tick_label:
    line += ' ' + tick_label
  print(line)

def draw_interval(center_length):
  """Draw tick interval based upon a central tick length."""
  if center_length > 0:                     # stop when length drops to 0
    draw_interval(center_length - 1)        # recursively draw top ticks
    draw_line(center_length)                # draw center tick
    draw_interval(center_length - 1)        # recursively draw bottom ticks

def draw_ruler(num_inches, major_length):
  """Draw English ruler with given number of inches, major tick length."""
  draw_line(major_length, '0')              # draw inch 0 line
  for j in range(1, 1 + num_inches):
    draw_interval(major_length - 1)         # draw interior ticks for inch
    draw_line(major_length, str(j))         # draw inch j line and label
```

Code Fragment 4.2: A recursive implementation of a function that draws a ruler.

4.3 Recursion Run Amok

Although recursion is a very powerful tool, it can easily be misused in various ways. In this section, we examine several problems in which a poorly implemented recursion causes drastic inefficiency, and we discuss some strategies for recognizing and avoid such pitfalls.

We begin by revisiting the ***element uniqueness problem***, defined on page 135 of Section 3.3.3. We can use the following recursive formulation to determine if all n elements of a sequence are unique. As a base case, when $n = 1$, the elements are trivially unique. For $n \geq 2$, the elements are unique if and only if the first $n-1$ elements are unique, the last $n-1$ items are unique, and the first and last elements are different (as that is the only pair that was not already checked as a subcase). A recursive implementation based on this idea is given in Code Fragment 4.6, named unique3 (to differentiate it from unique1 and unique2 from Chapter 3).

```
def unique3(S, start, stop):
  """Return True if there are no duplicate elements in slice S[start:stop]."""
  if stop − start <= 1: return True                  # at most one item
  elif not unique(S, start, stop−1): return False    # first part has duplicate
  elif not unique(S, start+1, stop): return False    # second part has duplicate
  else: return S[start] != S[stop−1]                 # do first and last differ?
```

Code Fragment 4.6: Recursive unique3 for testing element uniqueness.

Unfortunately, this is a terribly inefficient use of recursion. The nonrecursive part of each call uses $O(1)$ time, so the overall running time will be proportional to the total number of recursive invocations. To analyze the problem, we let n denote the number of entries under consideration, that is, let n= stop − start.

If $n = 1$, then the running time of unique3 is $O(1)$, since there are no recursive calls for this case. In the general case, the important observation is that a single call to unique3 for a problem of size n may result in two recursive calls on problems of size $n-1$. Those two calls with size $n-1$ could in turn result in four calls (two each) with a range of size $n-2$, and thus eight calls with size $n-3$ and so on. Thus, in the worst case, the total number of function calls is given by the geometric summation

$$1+2+4+\cdots+2^{n-1},$$

which is equal to $2^n - 1$ by Proposition 3.5. Thus, the running time of function unique3 is $O(2^n)$. This is an incredibly inefficient function for solving the element uniqueness problem. Its inefficiency comes not from the fact that it uses recursion—it comes from the fact that it uses recursion poorly, which is something we address in Exercise C-4.11.

Constructing a Multidimensional List

To quickly initialize a one-dimensional list, we generally rely on a syntax such as data = [0] ∗ n to create a list of n zeros. On page 189, we emphasized that from a technical perspective, this creates a list of length n with all entries referencing the same integer instance, but that there was no meaningful consequence of such aliasing because of the immutability of the int class in Python.

We have to be considerably more careful when creating a list of lists. If our goal were to create the equivalent of a two-dimensional list of integers, with r rows and c columns, and to initialize all values to zero, a flawed approach might be to try the command

```
data = ([0] * c) * r                # Warning: this is a mistake
```

While([0] ∗ c) is indeed a list of c zeros, multiplying that list by r unfortunately creates a single list with length $r \cdot c$, just as [2,4,6] ∗ 2 results in list [2, 4, 6, 2, 4, 6].

A better, yet still flawed attempt is to make a list that contains the list of c zeros as its only element, and then to multiply that list by r. That is, we could try the command

```
data = [ [0] * c ] * r              # Warning: still a mistake
```

This is much closer, as we actually do have a structure that is formally a list of lists. The problem is that all r entries of the list known as data are references to the same instance of a list of c zeros. Figure 5.23 provides a portrayal of such aliasing.

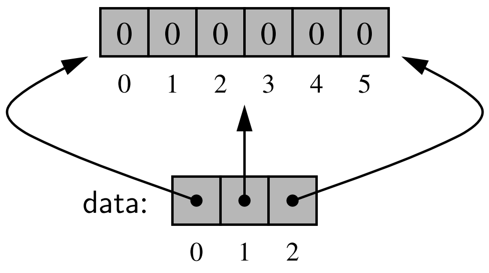

Figure 5.23: A flawed representation of a 3×6 data set as a list of lists, created with the command data = [[0] ∗ 6] ∗ 3. (For simplicity, we overlook the fact that the values in the secondary list are referential.)

This is truly a problem. Setting an entry such as data[2][0] = 100 would change the first entry of the secondary list to reference a new value, 100. Yet that cell of the secondary list also represents the value data[0][0], because “row” data[0] and “row” data[2] refer to the same secondary list.

5.7 Exercises

For help with exercises, please visit the site, www.wiley.com/college/goodrich.

Reinforcement

R-5.1 Execute the experiment from Code Fragment 5.1 and compare the results on your system to those we report in Code Fragment 5.2.

R-5.2 In Code Fragment 5.1, we perform an experiment to compare the length of a Python list to its underlying memory usage. Determining the sequence of array sizes requires a manual inspection of the output of that program. Redesign the experiment so that the program outputs only those values of k at which the existing capacity is exhausted. For example, on a system consistent with the results of Code Fragment 5.2, your program should output that the sequence of array capacities are 0, 4, 8, 16, 25, ...

R-5.3 Modify the experiment from Code Fragment 5.1 in order to demonstrate that Python's list class occasionally shrinks the size of its underlying array when elements are popped from a list.

R-5.4 Our DynamicArray class, as given in Code Fragment 5.3, does not support use of negative indices with __getitem__. Update that method to better match the semantics of a Python list.

R-5.5 Redo the justification of Proposition 5.1 assuming that the the cost of growing the array from size k to size $2k$ is $3k$ cyber-dollars. How much should each append operation be charged to make the amortization work?

R-5.6 Our implementation of insert for the DynamicArray class, as given in Code Fragment 5.5, has the following inefficiency. In the case when a resize occurs, the resize operation takes time to copy all the elements from an old array to a new array, and then the subsequent loop in the body of insert shifts many of those elements. Give an improved implementation of the insert method, so that, in the case of a resize, the elements are shifted into their final position during that operation, thereby avoiding the subsequent shifting.

R-5.7 Let A be an array of size $n \geq 2$ containing integers from 1 to $n-1$, inclusive, with exactly one repeated. Describe a fast algorithm for finding the integer in A that is repeated.

R-5.8 Experimentally evaluate the efficiency of the pop method of Python's list class when using varying indices as a parameter, as we did for insert on page 205. Report your results akin to Table 5.5.

Implementing a Stack Using a Python List

We use the adapter design pattern to define an ArrayStack class that uses an underlying Python list for storage. (We choose the name ArrayStack to emphasize that the underlying storage is inherently array based.) One question that remains is what our code should do if a user calls pop or top when the stack is empty. Our ADT suggests that an error occurs, but we must decide what type of error. When pop is called on an empty Python list, it formally raises an IndexError, as lists are index-based sequences. That choice does not seem appropriate for a stack, since there is no assumption of indices. Instead, we can define a new exception class that is more appropriate. Code Fragment 6.1 defines such an Empty class as a trivial subclass of the Python Exception class.

```
class Empty(Exception):
  """Error attempting to access an element from an empty container."""
  pass
```

Code Fragment 6.1: Definition for an Empty exception class.

The formal definition for our ArrayStack class is given in Code Fragment 6.2. The constructor establishes the member self._data as an initially empty Python list, for internal storage. The rest of the public stack behaviors are implemented, using the corresponding adaptation that was outlined in Table 6.1.

Example Usage

Below, we present an example of the use of our ArrayStack class, mirroring the operations at the beginning of Example 6.3 on page 230.

```
S = ArrayStack( )             # contents: [ ]
S.push(5)                     # contents: [5]
S.push(3)                     # contents: [5, 3]
print(len(S))                 # contents: [5, 3];     outputs 2
print(S.pop( ))               # contents: [5];        outputs 3
print(S.is_empty( ))          # contents: [5];        outputs False
print(S.pop( ))               # contents: [ ];        outputs 5
print(S.is_empty( ))          # contents: [ ];        outputs True
S.push(7)                     # contents: [7]
S.push(9)                     # contents: [7, 9]
print(S.top( ))               # contents: [7, 9];     outputs 9
S.push(4)                     # contents: [7, 9, 4]
print(len(S))                 # contents: [7, 9, 4];  outputs 3
print(S.pop( ))               # contents: [7, 9];     outputs 4
S.push(6)                     # contents: [7, 9, 6]
```

Shrinking the Underlying Array

A desirable property of a queue implementation is to have its space usage be $\Theta(n)$ where n is the current number of elements in the queue. Our ArrayQueue implementation, as given in Code Fragments 6.6 and 6.7, does not have this property. It expands the underlying array when enqueue is called with the queue at full capacity, but the dequeue implementation never shrinks the underlying array. As a consequence, the capacity of the underlying array is proportional to the maximum number of elements that have ever been stored in the queue, not the current number of elements.

We discussed this very issue on page 200, in the context of dynamic arrays, and in subsequent Exercises C-5.16 through C-5.20 of that chapter. A robust approach is to reduce the array to half of its current size, whenever the number of elements stored in it falls below *one fourth* of its capacity. We can implement this strategy by adding the following two lines of code in our dequeue method, just after reducing self._size at line 38 of Code Fragment 6.6, to reflect the loss of an element.

```
if 0 < self._size < len(self._data) // 4:
  self._resize(len(self._data) // 2)
```

Analyzing the Array-Based Queue Implementation

Table 6.3 describes the performance of our array-based implementation of the queue ADT, assuming the improvement described above for occasionally shrinking the size of the array. With the exception of the _resize utility, all of the methods rely on a constant number of statements involving arithmetic operations, comparisons, and assignments. Therefore, each method runs in worst-case $O(1)$ time, except for enqueue and dequeue, which have ***amortized*** bounds of $O(1)$ time, for reasons similar to those given in Section 5.3.

Operation	Running Time
Q.enqueue(e)	$O(1)^*$
Q.dequeue( )	$O(1)^*$
Q.first( )	$O(1)$
Q.is_empty( )	$O(1)$
len(Q)	$O(1)$

*amortized

Table 6.3: Performance of an array-based implementation of a queue. The bounds for enqueue and dequeue are amortized due to the resizing of the array. The space usage is $O(n)$, where n is the current number of elements in the queue.

7.1.1 Implementing a Stack with a Singly Linked List

In this section, we demonstrate use of a singly linked list by providing a complete Python implementation of the stack ADT (see Section 6.1). In designing such an implementation, we need to decide whether to model the top of the stack at the head or at the tail of the list. There is clearly a best choice here; we can efficiently insert and delete elements in constant time only at the head. Since all stack operations affect the top, we orient the top of the stack at the head of our list.

To represent individual nodes of the list, we develop a lightweight _Node class. This class will never be directly exposed to the user of our stack class, so we will formally define it as a nonpublic, nested class of our eventual LinkedStack class (see Section 2.5.1 for discussion of nested classes). The definition of the _Node class is shown in Code Fragment 7.4.

```
class _Node:
  """Lightweight, nonpublic class for storing a singly linked node."""
  __slots__ = '_element', '_next'          # streamline memory usage

  def __init__(self, element, next):       # initialize node's fields
    self._element = element                # reference to user's element
    self._next = next                      # reference to next node
```

Code Fragment 7.4: A lightweight _Node class for a singly linked list.

A node has only two instance variables: _element and _next. We intentionally define __slots__ to streamline the memory usage (see page 99 of Section 2.5.1 for discussion), because there may potentially be many node instances in a single list. The constructor of the _Node class is designed for our convenience, allowing us to specify initial values for both fields of a newly created node.

A complete implementation of our LinkedStack class is given in Code Fragments 7.5 and 7.6. Each stack instance maintains two variables. The _head member is a reference to the node at the head of the list (or None, if the stack is empty). We keep track of the current number of elements with the _size instance variable, for otherwise we would be forced to traverse the entire list to count the number of elements when reporting the size of the stack.

The implementation of push essentially mirrors the pseudo-code for insertion at the head of a singly linked list as outlined in Code Fragment 7.1. When we push a new element e onto the stack, we accomplish the necessary changes to the linked structure by invoking the constructor of the _Node class as follows:

```
self._head = self._Node(e, self._head)    # create and link a new node
```

Note that the _next field of the new node is set to the ***existing*** top node, and then self._head is reassigned to the new node.

Python Implementation of a Linked Binary Tree Structure

In this section, we define a concrete LinkedBinaryTree class that implements the binary tree ADT by subclassing the BinaryTree class. Our general approach is very similar to what we used when developing the PositionalList in Section 7.4: We define a simple, nonpublic _Node class to represent a node, and a public Position class that wraps a node. We provide a _validate utility for robustly checking the validity of a given position instance when unwrapping it, and a _make_position utility for wrapping a node as a position to return to a caller.

Those definitions are provided in Code Fragment 8.8. As a formality, the new Position class is declared to inherit immediately from BinaryTree.Position. Technically, the BinaryTree class definition (see Code Fragment 8.7) does not formally declare such a nested class; it trivially inherits it from Tree.Position. A minor benefit from this design is that our position class inherits the __ne__ special method so that syntax p != q is derived appropriately relative to __eq__.

Our class definition continues, in Code Fragment 8.9, with a constructor and with concrete implementations for the methods that remain abstract in the Tree and BinaryTree classes. The constructor creates an empty tree by initializing _root to None and _size to zero. These accessor methods are implemented with careful use of the _validate and _make_position utilities to safeguard against boundary cases.

Operations for Updating a Linked Binary Tree

Thus far, we have provided functionality for examining an existing binary tree. However, the constructor for our LinkedBinaryTree class results in an empty tree and we have not provided any means for changing the structure or content of a tree.

We chose not to declare update methods as part of the Tree or BinaryTree abstract base classes for several reasons. First, although the principle of encapsulation suggests that the outward behaviors of a class need not depend on the internal representation, the *efficiency* of the operations depends greatly upon the representation. We prefer to have each concrete implementation of a tree class offer the most suitable options for updating a tree.

The second reason we do not provide update methods in the base class is that we may not want such update methods to be part of a public interface. There are many applications of trees, and some forms of update operations that are suitable for one application may be unacceptable in another. However, if we place an update method in a base class, any class that inherits from that base will inherit the update method. Consider, for example, the possibility of a method T.replace(p, e) that replaces the element stored at position *p* with another element *e*. Such a general method may be unacceptable in the context of an ***arithmetic expression tree*** (see Example 8.7 on page 312, and a later case study in Section 8.5), because we may want to enforce that internal nodes store only operators as elements.

R-8.11 Find the value of the arithmetic expression associated with each subtree of the binary tree of Figure 8.8.

R-8.12 Draw an arithmetic expression tree that has four external nodes, storing the numbers 1, 5, 6, and 7 (with each number stored in a distinct external node, but not necessarily in this order), and has three internal nodes, each storing an operator from the set $\{+,-,\times,/\}$, so that the value of the root is 21. The operators may return and act on fractions, and an operator may be used more than once.

R-8.13 Draw the binary tree representation of the following arithmetic expression: "$(((5+2)*(2-1))/((2+9)+((7-2)-1))*8)$".

R-8.14 Justify Table 8.2, summarizing the running time of the methods of a tree represented with a linked structure, by providing, for each method, a description of its implementation, and an analysis of its running time.

R-8.15 The LinkedBinaryTree class provides only nonpublic versions of the update methods discussed on page 319. Implement a simple subclass named MutableLinkedBinaryTree that provides public wrapper functions for each of the inherited nonpublic update methods.

R-8.16 Let T be a binary tree with n nodes, and let $f()$ be the level numbering function of the positions of T, as given in Section 8.3.2.

a. Show that, for every position p of T, $f(p) \leq 2^n - 2$.
b. Show an example of a binary tree with seven nodes that attains the above upper bound on $f(p)$ for some position p.

R-8.17 Show how to use the Euler tour traversal to compute the level number $f(p)$, as defined in Section 8.3.2, of each position in a binary tree T.

R-8.18 Let T be a binary tree with n positions that is realized with an array representation A, and let $f()$ be the level numbering function of the positions of T, as given in Section 8.3.2. Give pseudo-code descriptions of each of the methods root, parent, left, right, is_leaf, and is_root.

R-8.19 Our definition of the level numbering function $f(p)$, as given in Section 8.3.2, began with the root having number 0. Some authors prefer to use a level numbering $g(p)$ in which the root is assigned number 1, because it simplifies the arithmetic for finding neighboring positions. Redo Exercise R-8.18, but assuming that we use a level numbering $g(p)$ in which the root is assigned number 1.

R-8.20 Draw a binary tree T that simultaneously satisfies the following:

- Each internal node of T stores a single character.
- A *preorder* traversal of T yields EXAMFUN.
- An *inorder* traversal of T yields MAFXUEN.

R-8.21 In what order are positions visited during a preorder traversal of the tree of Figure 8.8?

R-10.24 Give a pseudo-code description of the __delitem__ map operation when using a skip list.

R-10.25 Give a concrete implementation of the pop method, in the context of a MutableSet abstract base class, that relies only on the five core set behaviors described in Section 10.5.2.

R-10.26 Give a concrete implementation of the isdisjoint method in the context of the MutableSet abstract base class, relying only on the five primary abstract methods of that class. Your algorithm should run in $O(\min(n,m))$ where n and m denote the respective cardinalities of the two sets.

R-10.27 What abstraction would you use to manage a database of friends' birthdays in order to support efficient queries such as "find all friends whose birthday is today" and "find the friend who will be the next to celebrate a birthday"?

Creativity

C-10.28 On page 406 of Section 10.1.3, we give an implementation of the method setdefault as it might appear in the MutableMapping abstract base class. While that method accomplishes the goal in a general fashion, its efficiency is less than ideal. In particular, when the key is new, there will be a failed search due to the initial use of __getitem__, and then a subsequent insertion via __setitem__. For a concrete implementation, such as the UnsortedTableMap, this is twice the work because a complete scan of the table will take place during the failed __getitem__, and then another complete scan of the table takes place due to the implementation of __setitem__. A better solution is for the UnsortedTableMap class to override setdefault to provide a direct solution that performs a single search. Give such an implementation of UnsortedTableMap.setdefault.

C-10.29 Repeat Exercise C-10.28 for the ProbeHashMap class.

C-10.30 Repeat Exercise C-10.28 for the ChainHashMap class.

C-10.31 For an ideal compression function, the capacity of the bucket array for a hash table should be a prime number. Therefore, we consider the problem of locating a prime number in a range $[M,2M]$. Implement a method for finding such a prime by using the ***sieve algorithm***. In this algorithm, we allocate a $2M$ cell Boolean array A, such that cell i is associated with the integer i. We then initialize the array cells to all be "true" and we "mark off" all the cells that are multiples of 2, 3, 5, 7, and so on. This process can stop after it reaches a number larger than $\sqrt{2M}$. (Hint: Consider a bootstrapping method for finding the primes up to $\sqrt{2M}$.)

11.1 Binary Search Trees

In Chapter 8 we introduced the tree data structure and demonstrated a variety of applications. One important use is as a ***search tree*** (as described on page 332). In this chapter, we use a search tree structure to efficiently implement a ***sorted map***. The three most fundamental methods of a map M (see Section 10.1.1) are:

M[k]: Return the value v associated with key k in map M, if one exists; otherwise raise a KeyError; implemented with __getitem__ method.

M[k] = v: Associate value v with key k in map M, replacing the existing value if the map already contains an item with key equal to k; implemented with __setitem__ method.

del M[k]: Remove from map M the item with key equal to k; if M has no such item, then raise a KeyError; implemented with __delitem__ method.

The sorted map ADT includes additional functionality (see Section 10.3), guaranteeing that an iteration reports keys in sorted order, and supporting additional searches such as find_gt(k) and find_range(start, stop).

Binary trees are an excellent data structure for storing items of a map, assuming we have an order relation defined on the keys. In this context, a ***binary search tree*** is a binary tree T with each position p storing a key-value pair (k, v) such that:

- Keys stored in the left subtree of p are less than k.
- Keys stored in the right subtree of p are greater than k.

An example of such a binary search tree is given in Figure 11.1. As a matter of convenience, we will not diagram the values associated with keys in this chapter, since those values do not affect the placement of items within a search tree.

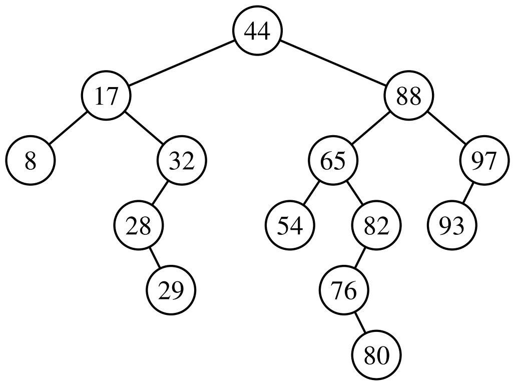

Figure 11.1: A binary search tree with integer keys. We omit the display of associated values in this chapter, since they are not relevant to the order of items within a search tree.

It is easy to see that bucket-sort runs in $O(n+N)$ time and uses $O(n+N)$ space. Hence, bucket-sort is efficient when the range N of values for the keys is small compared to the sequence size n, say $N = O(n)$ or $N = O(n\log n)$. Still, its performance deteriorates as N grows compared to n.

An important property of the bucket-sort algorithm is that it works correctly even if there are many different elements with the same key. Indeed, we described it in a way that anticipates such occurrences.

Stable Sorting

When sorting key-value pairs, an important issue is how equal keys are handled. Let $S = ((k_0, v_0), \ldots, (k_{n-1}, v_{n-1}))$ be a sequence of such entries. We say that a sorting algorithm is ***stable*** if, for any two entries (k_i, v_i) and (k_j, v_j) of S such that $k_i = k_j$ and (k_i, v_i) precedes (k_j, v_j) in S before sorting (that is, $i < j$), entry (k_i, v_i) also precedes entry (k_j, v_j) after sorting. Stability is important for a sorting algorithm because applications may want to preserve the initial order of elements with the same key.

Our informal description of bucket-sort in Code Fragment 12.7 guarantees stability as long as we ensure that all sequences act as queues, with elements processed and removed from the front of a sequence and inserted at the back. That is, when initially placing elements of S into buckets, we should process S from front to back, and add each element to the end of its bucket. Subsequently, when transferring elements from the buckets back to S, we should process each $B[i]$ from front to back, with those elements added to the end of S.

Radix-Sort

One of the reasons that stable sorting is so important is that it allows the bucket-sort approach to be applied to more general contexts than to sort integers. Suppose, for example, that we want to sort entries with keys that are pairs (k, l), where k and l are integers in the range $[0, N-1]$, for some integer $N \geq 2$. In a context such as this, it is common to define an order on these keys using the ***lexicographic*** (dictionary) convention, where $(k_1, l_1) < (k_2, l_2)$ if $k_1 < k_2$ or if $k_1 = k_2$ and $l_1 < l_2$ (see page 15). This is a pairwise version of the lexicographic comparison function, which can be applied to equal-length character strings, or to tuples of length d.

The ***radix-sort*** algorithm sorts a sequence S of entries with keys that are pairs, by applying a stable bucket-sort on the sequence twice; first using one component of the pair as the key when ordering and then using the second component. But which order is correct? Should we first sort on the k's (the first component) and then on the l's (the second component), or should it be the other way around?

Example 13.2: *Consider the pattern P =* "amalgamation" *from Figure 13.5. The Knuth-Morris-Pratt (KMP) failure function, $f(k)$, for the string P is as shown in the following table:*

k	0	1	2	3	4	5	6	7	8	9	10	11
$P[k]$	a	m	a	l	g	a	m	a	t	i	o	n
$f(k)$	0	0	1	0	0	1	2	3	0	0	0	0

Implementation

Our implementation of the KMP pattern-matching algorithm is shown in Code Fragment 13.3. It relies on a utility function, compute_kmp_fail, discussed on the next page, to compute the failure function efficiently.

The main part of the KMP algorithm is its **while** loop, each iteration of which performs a comparison between the character at index j in T and the character at index k in P. If the outcome of this comparison is a match, the algorithm moves on to the next characters in both T and P (or reports a match if reaching the end of the pattern). If the comparison failed, the algorithm consults the failure function for a new candidate character in P, or starts over with the next index in T if failing on the first character of the pattern (since nothing can be reused).

```
def find_kmp(T, P):
  """Return the lowest index of T at which substring P begins (or else -1)."""
  n, m = len(T), len(P)                      # introduce convenient notations
  if m == 0: return 0                        # trivial search for empty string
  fail = compute_kmp_fail(P)                 # rely on utility to precompute
  j = 0                                      # index into text
  k = 0                                      # index into pattern
  while j < n:
    if T[j] == P[k]:                         # P[0:1+k] matched thus far
      if k == m − 1:                         # match is complete
        return j − m + 1
      j += 1                                 # try to extend match
      k += 1
    elif k > 0:
      k = fail[k−1]                          # reuse suffix of P[0:k]
    else:
      j += 1
  return −1                                  # reached end without match
```

Code Fragment 13.3: An implementation of the KMP pattern-matching algorithm. The compute_kmp_fail utility function is given in Code Fragment 13.4.

14.3.2 DFS Implementation and Extensions

We begin by providing a Python implementation of the basic depth-first search algorithm, originally described with pseudo-code in Code Fragment 14.4. Our DFS function is presented in Code Fragment 14.5.

```
def DFS(g, u, discovered):
  """Perform DFS of the undiscovered portion of Graph g starting at Vertex u.

  discovered is a dictionary mapping each vertex to the edge that was used to
  discover it during the DFS. (u should be "discovered" prior to the call.)
  Newly discovered vertices will be added to the dictionary as a result.
  """
  for e in g.incident_edges(u):          # for every outgoing edge from u
    v = e.opposite(u)
    if v not in discovered:              # v is an unvisited vertex
      discovered[v] = e                  # e is the tree edge that discovered v
      DFS(g, v, discovered)              # recursively explore from v
```

Code Fragment 14.5: Recursive implementation of depth-first search on a graph, starting at a designated vertex *u*.

In order to track which vertices have been visited, and to build a representation of the resulting DFS tree, our implementation introduces a third parameter, named discovered. This parameter should be a Python dictionary that maps a vertex of the graph to the tree edge that was used to discover that vertex. As a technicality, we assume that the source vertex *u* occurs as a key of the dictionary, with None as its value. Thus, a caller might start the traversal as follows:

```
result = {u : None}          # a new dictionary, with u trivially discovered
DFS(g, u, result)
```

The dictionary serves two purposes. Internally, the dictionary provides a mechanism for recognizing visited vertices, as they will appear as keys in the dictionary. Externally, the DFS function augments this dictionary as it proceeds, and thus the values within the dictionary are the DFS tree edges at the conclusion of the process.

Because the dictionary is hash-based, the test, "**if** v **not in** discovered," and the record-keeping step, "discovered[v] = e," run in $O(1)$ ***expected*** time, rather than worst-case time. In practice, this is a compromise we are willing to accept, but it does violate the formal analysis of the algorithm, as given on page 643. If we could assume that vertices could be numbered from 0 to $n-1$, then those numbers could be used as indices into an array-based lookup table rather than a hash-based map. Alternatively, we could store each vertex's discovery status and associated tree edge directly as part of the vertex instance.

Although the DFS_complete function makes multiple calls to the original DFS function, the total time spent by a call to DFS_complete is $O(n+m)$. For an undirected graph, recall from our original analysis on page 643 that a single call to DFS starting at vertex s runs in time $O(n_s+m_s)$ where n_s is the number of vertices reachable from s, and m_s is the number of incident edges to those vertices. Because each call to DFS explores a different component, the sum of n_s+m_s terms is $n+m$. The $O(n+m)$ total bound applies to the directed case as well, even though the sets of reachable vertices are not necessarily disjoint. However, because the same discovery dictionary is passed as a parameter to all DFS calls, we know that the DFS subroutine is called once on each vertex, and then each outgoing edge is explored only once during the process.

The DFS_complete function can be used to analyze the connected components of an undirected graph. The discovery dictionary it returns represents a ***DFS forest*** for the entire graph. We say this is a forest rather than a tree, because the graph may not be connected. The number of connected components can be determined by the number of vertices in the discovery dictionary that have None as their discovery edge (those are roots of DFS trees). A minor modification to the core DFS method could be used to tag each vertex with a component number when it is discovered. (See Exercise C-14.44.)

The situation is more complex for finding strongly connected components of a directed graph. There exists an approach for computing those components in $O(n+m)$ time, making use of two separate depth-first search traversals, but the details are beyond the scope of this book.

Detecting Cycles with DFS

For both undirected and directed graphs, a cycle exists if and only if a ***back edge*** exists relative to the DFS traversal of that graph. It is easy to see that if a back edge exists, a cycle exists by taking the back edge from the descendant to its ancestor and then following the tree edges back to the descendant. Conversely, if a cycle exists in the graph, there must be a back edge relative to a DFS (although we do not prove this fact here).

Algorithmically, detecting a back edge in the undirected case is easy, because all edges are either tree edges or back edges. In the case of a directed graph, additional modifications to the core DFS implementation are needed to properly categorize a nontree edge as a back edge. When a directed edge is explored leading to a previously visited vertex, we must recognize whether that vertex is an ancestor of the current vertex. This requires some additional bookkeeping, for example, by tagging vertices upon which a recursive call to DFS is still active. We leave details as an exercise (C-14.43).

```
  def get_edge(self, u, v):
    """Return the edge from u to v, or None if not adjacent."""
    return self._outgoing[u].get(v)            # returns None if v not adjacent

  def degree(self, v, outgoing=True):
    """Return number of (outgoing) edges incident to vertex v in the graph.

    If graph is directed, optional parameter used to count incoming edges.
    """
    adj = self._outgoing if outgoing else self._incoming
    return len(adj[v])

  def incident_edges(self, v, outgoing=True):
    """Return all (outgoing) edges incident to vertex v in the graph.

    If graph is directed, optional parameter used to request incoming edges.
    """
    adj = self._outgoing if outgoing else self._incoming
    for edge in adj[v].values( ):
      yield edge

  def insert_vertex(self, x=None):
    """Insert and return a new Vertex with element x."""
    v = self.Vertex(x)
    self._outgoing[v] = { }
    if self.is_directed( ):
      self._incoming[v] = { }          # need distinct map for incoming edges
    return v

  def insert_edge(self, u, v, x=None):
    """Insert and return a new Edge from u to v with auxiliary element x."""
    e = self.Edge(u, v, x)
    self._outgoing[u][v] = e
    self._incoming[v][u] = e
```

Code Fragment 14.3: Graph class definition (continued from Code Fragment 14.2). We omit error-checking of parameters for brevity.

15.1.1 Memory Allocation

With Python, all objects are stored in a pool of memory, known as the ***memory heap*** or ***Python heap*** (which should not be confused with the "heap" data structure presented in Chapter 9). When a command such as

```
w = Widget()
```

is executed, assuming Widget is the name of a class, a new instance of the class is created and stored somewhere within the memory heap. The Python interpreter is responsible for negotiating the use of space with the operating system and for managing the use of the memory heap when executing a Python program.

The storage available in the memory heap is divided into ***blocks***, which are contiguous array-like "chunks" of memory that may be of variable or fixed sizes. The system must be implemented so that it can quickly allocate memory for new objects. One popular method is to keep contiguous "holes" of available free memory in a linked list, called the ***free list***. The links joining these holes are stored inside the holes themselves, since their memory is not being used. As memory is allocated and deallocated, the collection of holes in the free lists changes, with the unused memory being separated into disjoint holes divided by blocks of used memory. This separation of unused memory into separate holes is known as ***fragmentation***. The problem is that it becomes more difficult to find large continuous chunks of memory, when needed, even though an equivalent amount of memory may be unused (yet fragmented). Therefore, we would like to minimize fragmentation as much as possible.

There are two kinds of fragmentation that can occur. ***Internal fragmentation*** occurs when a portion of an allocated memory block is unused. For example, a program may request an array of size 1000, but only use the first 100 cells of this array. There is not much that a run-time environment can do to reduce internal fragmentation. ***External fragmentation***, on the other hand, occurs when there is a significant amount of unused memory between several contiguous blocks of allocated memory. Since the run-time environment has control over where to allocate memory when it is requested, the run-time environment should allocate memory in a way to try to reduce external fragmentation as much as reasonably possible.

Several heuristics have been suggested for allocating memory from the heap so as to minimize external fragmentation. The ***best-fit algorithm*** searches the entire free list to find the hole whose size is closest to the amount of memory being requested. The ***first-fit algorithm*** searches from the beginning of the free list for the first hole that is large enough. The ***next-fit algorithm*** is similar, in that it also searches the free list for the first hole that is large enough, but it begins its search from where it left off previously, viewing the free list as a circularly linked list (Section 7.2). The ***worst-fit algorithm*** searches the free list to find the largest hole of available memory, which might be done faster than a search of the entire free list