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.

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.

```
1  def DFS(g, u, discovered):
2    """Perform DFS of the undiscovered portion of Graph g starting at Vertex u.
3
4    discovered is a dictionary mapping each vertex to the edge that was used to
5    discover it during the DFS. (u should be "discovered" prior to the call.)
6    Newly discovered vertices will be added to the dictionary as a result.
7    """
8    for e in g.incident_edges(u):          # for every outgoing edge from u
9      v = e.opposite(u)
10     if v not in discovered:              # v is an unvisited vertex
11       discovered[v] = e                  # e is the tree edge that discovered v
12       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.

5.3 Dynamic Arrays and Amortization

When creating a low-level array in a computer system, the precise size of that array must be explicitly declared in order for the system to properly allocate a consecutive piece of memory for its storage. For example, Figure 5.11 displays an array of 12 bytes that might be stored in memory locations 2146 through 2157.

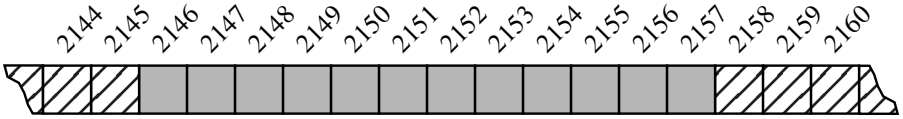

Figure 5.11: An array of 12 bytes allocated in memory locations 2146 through 2157.

Because the system might dedicate neighboring memory locations to store other data, the capacity of an array cannot trivially be increased by expanding into subsequent cells. In the context of representing a Python `tuple` or `str` instance, this constraint is no problem. Instances of those classes are immutable, so the correct size for an underlying array can be fixed when the object is instantiated.

Python's `list` class presents a more interesting abstraction. Although a list has a particular length when constructed, the class allows us to add elements to the list, with no apparent limit on the overall capacity of the list. To provide this abstraction, Python relies on an algorithmic sleight of hand known as a ***dynamic array***.

The first key to providing the semantics of a dynamic array is that a list instance maintains an underlying array that often has greater capacity than the current length of the list. For example, while a user may have created a list with five elements, the system may have reserved an underlying array capable of storing eight object references (rather than only five). This extra capacity makes it easy to append a new element to the list by using the next available cell of the array.

If a user continues to append elements to a list, any reserved capacity will eventually be exhausted. In that case, the class requests a new, larger array from the system, and initializes the new array so that its prefix matches that of the existing smaller array. At that point in time, the old array is no longer needed, so it is reclaimed by the system. Intuitively, this strategy is much like that of the hermit crab, which moves into a larger shell when it outgrows its previous one.

We give empirical evidence that Python's `list` class is based upon such a strategy. The source code for our experiment is displayed in Code Fragment 5.1, and a sample output of that program is given in Code Fragment 5.2. We rely on a function named `getsizeof` that is available from the `sys` module. This function reports the number of bytes that are being used to store an object in Python. For a list, it reports the number of bytes devoted to the array and other instance variables of the list, but *not* any space devoted to elements referenced by the list.

next number in a sequence based upon one or more past numbers that it has generated. Indeed, a simple yet popular pseudo-random number generator chooses its next number based solely on the most recently chosen number and some additional parameters using the following formula.

```
next = (a*current + b) % n;
```

where a, b, and n are appropriately chosen integers. Python uses a more advanced technique known as a ***Mersenne twister***. It turns out that the sequences generated by these techniques can be proven to be statistically uniform, which is usually good enough for most applications requiring random numbers, such as games. For applications, such as computer security settings, where one needs unpredictable random sequences, this kind of formula should not be used. Instead, one should ideally sample from a source that is actually random, such as radio static coming from outer space.

Since the next number in a pseudo-random generator is determined by the previous number(s), such a generator always needs a place to start, which is called its ***seed***. The sequence of numbers generated for a given seed will always be the same. One common trick to get a different sequence each time a program is run is to use a seed that will be different for each run. For example, we could use some timed input from a user or the current system time in milliseconds.

Python's random module provides support for pseudo-random number generation by defining a Random class; instances of that class serve as generators with independent state. This allows different aspects of a program to rely on their own pseudo-random number generator, so that calls to one generator do not affect the sequence of numbers produced by another. For convenience, all of the methods supported by the Random class are also supported as stand-alone functions of the random module (essentially using a single generator instance for all top-level calls).

Syntax	Description
seed(hashable)	Initializes the pseudo-random number generator based upon the hash value of the parameter
random( )	Returns a pseudo-random floating-point value in the interval $[0.0, 1.0)$.
randint(a,b)	Returns a pseudo-random integer in the closed interval $[a, b]$.
randrange(start, stop, step)	Returns a pseudo-random integer in the standard Python range indicated by the parameters.
choice(seq)	Returns an element of the given sequence chosen pseudo-randomly.
shuffle(seq)	Reorders the elements of the given sequence pseudo-randomly.

Table 1.8: Methods supported by instances of the Random class, and as top-level functions of the random module.

Our representation of a 3×3 board will be a list of lists of characters, with `'X'` or `'O'` designating a player's move, or `' '` designating an empty space. For example, the board configuration

O	X	O
	X	
	O	X

will be stored internally as

```
[ ['O', 'X', 'O'], [' ', 'X', ' '], [' ', 'O', 'X'] ]
```

We develop a complete Python class for maintaining a Tic-Tac-Toe board for two players. That class will keep track of the moves and report a winner, but it does not perform any strategy or allow someone to play Tic-Tac-Toe against the computer. The details of such a program are beyond the scope of this chapter, but it might nonetheless make a good course project (see Exercise P-8.68).

Before presenting the implementation of the class, we demonstrate its public interface with a simple test in Code Fragment 5.12.

```
game = TicTacToe( )
# X moves:              # O moves:
game.mark(1, 1);        game.mark(0, 2)
game.mark(2, 2);        game.mark(0, 0)
game.mark(0, 1);        game.mark(2, 1)
game.mark(1, 2);        game.mark(1, 0)
game.mark(2, 0)

print(game)
winner = game.winner( )
if winner is None:
  print('Tie')
else:
  print(winner, 'wins')
```

Code Fragment 5.12: A simple test for our Tic-Tac-Toe class.

The basic operations are that a new game instance represents an empty board, that the mark(i,j) method adds a mark at the given position for the current player (with the software managing the alternating of turns), and that the game board can be printed and the winner determined. The complete source code for the TicTacToe class is given in Code Fragment 5.13. Our mark method performs error checking to make sure that valid indices are sent, that the position is not already occupied, and that no further moves are made after someone wins the game.

if this list were maintained as a priority queue (Chapter 9). In each algorithm, the requested amount of memory is subtracted from the chosen memory hole and the leftover part of that hole is returned to the free list.

Although it might sound good at first, the best-fit algorithm tends to produce the worst external fragmentation, since the leftover parts of the chosen holes tend to be small. The first-fit algorithm is fast, but it tends to produce a lot of external fragmentation at the front of the free list, which slows down future searches. The next-fit algorithm spreads fragmentation more evenly throughout the memory heap, thus keeping search times low. This spreading also makes it more difficult to allocate large blocks, however. The worst-fit algorithm attempts to avoid this problem by keeping contiguous sections of free memory as large as possible.

15.1.2 Garbage Collection

In some languages, like C and C++, the memory space for objects must be explicitly deallocated by the programmer, which is a duty often overlooked by beginning programmers and is the source of frustrating programming errors even for experienced programmers. The designers of Python instead placed the burden of memory management entirely on the interpreter. The process of detecting "stale" objects, deallocating the space devoted to those objects, and returning the reclaimed space to the free list is known as ***garbage collection***.

To perform automated garbage collection, there must first be a way to detect those objects that are no longer necessary. Since the interpreter cannot feasibly analyze the semantics of an arbitrary Python program, it relies on the following conservative rule for reclaiming objects. In order for a program to access an object, it must have a direct or indirect reference to that object. We will define such objects to be ***live objects***. In defining a live object, a ***direct reference*** to an object is in the form of an identifier in an active namespace (i.e., the global namespace, or the local namespace for any active function). For example, immediately after the command `w = Widget()` is executed, identifier `w` will be defined in the current namespace as a reference to the new widget object. We refer to all such objects with direct references as ***root objects***. An ***indirect reference*** to a live object is a reference that occurs within the state of some other live object. For example, if the widget instance in our earlier example maintains a list as an attribute, that list is also a live object (as it can be reached indirectly through use of identifier `w`). The set of live objects are defined recursively; thus, any objects that are referenced within the list that is referenced by the widget are also classified as live objects.

The Python interpreter assumes that live objects are the active objects currently being used by the running program; these objects should ***not*** be deallocated. Other objects can be garbage collected. Python relies on the following two strategies for determining which objects are live.