2.3.2 Operator Overloading and Python's Special Methods

Python's built-in classes provide natural semantics for many operators. For example, the syntax a + b invokes addition for numeric types, yet concatenation for sequence types. When defining a new class, we must consider whether a syntax like a + b should be defined when a or b is an instance of that class.

By default, the + operator is undefined for a new class. However, the author of a class may provide a definition using a technique known as ***operator overloading***. This is done by implementing a specially named method. In particular, the + operator is overloaded by implementing a method named `__add__`, which takes the right-hand operand as a parameter and which returns the result of the expression. That is, the syntax, a + b, is converted to a method call on object a of the form, `a.__add__(b)`. Similar specially named methods exist for other operators. Table 2.1 provides a comprehensive list of such methods.

When a binary operator is applied to two instances of different types, as in `3 * 'love me'`, Python gives deference to the class of the *left* operand. In this example, it would effectively check if the int class provides a sufficient definition for how to multiply an instance by a string, via the `__mul__` method. However, if that class does not implement such a behavior, Python checks the class definition for the right-hand operand, in the form of a special method named `__rmul__` (i.e., "right multiply"). This provides a way for a new user-defined class to support mixed operations that involve an instance of an existing class (given that the existing class would presumably not have defined a behavior involving this new class). The distinction between `__mul__` and `__rmul__` also allows a class to define different semantics in cases, such as matrix multiplication, in which an operation is noncommutative (that is, A * x may differ from x * A).

Non-Operator Overloads

In addition to traditional operator overloading, Python relies on specially named methods to control the behavior of various other functionality, when applied to user-defined classes. For example, the syntax, str(foo), is formally a call to the constructor for the string class. Of course, if the parameter is an instance of a user-defined class, the original authors of the string class could not have known how that instance should be portrayed. So the string constructor calls a specially named method, `foo.__str__()`, that must return an appropriate string representation.

Similar special methods are used to determine how to construct an int, float, or bool based on a parameter from a user-defined class. The conversion to a Boolean value is particularly important, because the syntax, **if** foo:, can be used even when foo is not formally a Boolean value (see Section 1.4.1). For a user-defined class, that condition is evaluated by the special method `foo.__bool__()`.

14.3 Graph Traversals

Greek mythology tells of an elaborate labyrinth that was built to house the monstrous Minotaur, which was part bull and part man. This labyrinth was so complex that neither beast nor human could escape it. No human, that is, until the Greek hero, Theseus, with the help of the king's daughter, Ariadne, decided to implement a ***graph traversal*** algorithm. Theseus fastened a ball of thread to the door of the labyrinth and unwound it as he traversed the twisting passages in search of the monster. Theseus obviously knew about good algorithm design, for, after finding and defeating the beast, Theseus easily followed the string back out of the labyrinth to the loving arms of Ariadne.

Formally, a ***traversal*** is a systematic procedure for exploring a graph by examining all of its vertices and edges. A traversal is efficient if it visits all the vertices and edges in time proportional to their number, that is, in linear time.

Graph traversal algorithms are key to answering many fundamental questions about graphs involving the notion of ***reachability***, that is, in determining how to travel from one vertex to another while following paths of a graph. Interesting problems that deal with reachability in an undirected graph G include the following:

- Computing a path from vertex u to vertex v, or reporting that no such path exists.
- Given a start vertex s of G, computing, for every vertex v of G, a path with the minimum number of edges between s and v, or reporting that no such path exists.
- Testing whether G is connected.
- Computing a spanning tree of G, if G is connected.
- Computing the connected components of G.
- Computing a cycle in G, or reporting that G has no cycles.

Interesting problems that deal with reachability in a directed graph $\vec{G}$ include the following:

- Computing a directed path from vertex u to vertex v, or reporting that no such path exists.
- Finding all the vertices of $\vec{G}$ that are reachable from a given vertex s.
- Determine whether $\vec{G}$ is acyclic.
- Determine whether $\vec{G}$ is strongly connected.

In the remainder of this section, we present two efficient graph traversal algorithms, called ***depth-first search*** and ***breadth-first search***, respectively.

R-14.21 Compute a topological ordering for the directed graph drawn with solid edges in Figure 14.3d.

R-14.22 Bob loves foreign languages and wants to plan his course schedule for the following years. He is interested in the following nine language courses: LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, and LA169. The course prerequisites are:

- LA15: (none)
- LA16: LA15
- LA22: (none)
- LA31: LA15
- LA32: LA16, LA31
- LA126: LA22, LA32
- LA127: LA16
- LA141: LA22, LA16
- LA169: LA32

In what order can Bob take these courses, respecting the prerequisites?

R-14.23 Draw a simple, connected, weighted graph with 8 vertices and 16 edges, each with unique edge weights. Identify one vertex as a "start" vertex and illustrate a running of Dijkstra's algorithm on this graph.

R-14.24 Show how to modify the pseudo-code for Dijkstra's algorithm for the case when the graph is directed and we want to compute shortest directed paths from the source vertex to all the other vertices.

R-14.25 Draw a simple, connected, undirected, weighted graph with 8 vertices and 16 edges, each with unique edge weights. Illustrate the execution of the Prim-Jarník algorithm for computing the minimum spanning tree of this graph.

R-14.26 Repeat the previous problem for Kruskal's algorithm.

R-14.27 There are eight small islands in a lake, and the state wants to build seven bridges to connect them so that each island can be reached from any other one via one or more bridges. The cost of constructing a bridge is proportional to its length. The distances between pairs of islands are given in the following table.

	1	2	3	4	5	6	7	8
1	-	240	210	340	280	200	345	120
2	-	-	265	175	215	180	185	155
3	-	-	-	260	115	350	435	195
4	-	-	-	-	160	330	295	230
5	-	-	-	-	-	360	400	170
6	-	-	-	-	-	-	175	205
7	-	-	-	-	-	-	-	305
8	-	-	-	-	-	-	-	-

Find which bridges to build to minimize the total construction cost.