```
        return self.value

class B (A):
   ...
class C (A):
   ...
class D(B, C):
   ...
```

Notice that class A is a "grandparent" to class D in two ways, both through class B and class C. A curious question is, then, how many copies of the data fields defined in class A does an instance of class D possess? Write a small simple program to answer this question—that is, a program that will produce one answer if an instance of class D has only one data field and a different answer if an instance of class D has two data fields.

CHAPTER 8

Functional Programming

Programming a computer is a complicated task, and like most complicated tasks, there are many different ways to think about the process. The term *language paradigm* is sometimes used to describe the mental model that the programmer envisions as he or she is creating a program. The model we have used to this point is termed the *imperative paradigm*. This model views the computer as a combination of processor and memory. Instructions (such as assignment statements) have the effect of making changes to memory. The task of programming consists of placing statements in their proper sequence, so that by a large number of small transformations to memory, the desired result is eventually produced.

You might be surprised to learn that this is not the only possible way to think about the process of computation. In this chapter and the following, two alternative models are described. Each differs from the imperative paradigm not in the way the computer operates, but in the way that the programmer *thinks* about the tasks of programming.

THE FUNCTIONAL PROGRAMMING PARADIGM

The term *functional programming* does not imply simply programming with functions; it is used to describe an alternative to the imperative programming paradigm. As the *functional programming* name suggests, the creation of functions is an important part of the paradigm. But simply defining a few functions does not mean that you are programming in a functional style. For example, many functions were defined in earlier chapters, yet we did not call those functional programs.

The key characteristic of a program developed in the functional programming style is that it creates new values by a process of *transformation*. Generally, values are represented as lists, or dictionaries. This simple description requires further explanation. The traditional imperative style of programming produces complex values by modification—by making a large number of small changes to an existing data structure (for example, creating a dictionary of values, and then systematically setting each value independently of the others). Since small changes can often be accompanied by small errors, and small errors may produce only a minimal effect, debugging imperative programs can be frustratingly difficult.

By emphasizing *transformation*, rather than *modification*, functional programs work on a larger scale. Transformations are often more uniform and much simpler to