```
    x3 = gradeToNumber(grade3)
    x4 = gradeToNumber(grade4)
    xlow = min(x1, x2, x3, x4)
    avg = (x1 + x2 + x3 + x4 - xlow) / 3
    print(numberToGrade(avg))

    return False
```

Step 6 Write the main function.

The main function is now utterly trivial. We keep calling processLine while it returns True.

```
def main() :
   done = False
   while not done :
      done = processLine()
```

We place all functions into a single Python file. See ch05/grades.py in your source code for the complete program.

5.8 Variable Scope

As your programs get larger and contain more variables, you may encounter problems where you cannot access a variable that is defined in a different part of your program, or where two variable definitions conflict with each other. In order to resolve these problems, you need to be familiar with the concept of *variable scope*.

The scope of a variable is the part of the program in which it is visible.

The **scope** of a variable is the part of the program in which you can access it. For example, the scope of a function's parameter variable is the entire function. In the following code segment, the scope of the parameter variable sideLength is the entire cubeVolume function but not the main function.

```
def main() :
   print(cubeVolume(10))

def cubeVolume(sideLength) :
   return sideLength ** 3
```

A variable that is defined within a function is called a **local variable**. When a local variable is defined in a block, it becomes available from that point until the end of the function in which it is defined. For example, in the code segment below, the scope of the square variable is highlighted.

```
def main() :
   sum = 0
   for i in range(11) :
      square = i * i
      sum = sum + square

   print(square, sum)
```

A loop variable in a `for` statement is a local variable. As with any local variable, its scope extends to the end of the function in which it was defined:

```
def main() :
   sum = 0
   for i in range(11) :
      square = i * i
      sum = sum + square

   print(i, sum)
```

Here is an example of a scope problem:

```
def main() :
   sideLength = 10
   result = cubeVolume()
   print(result)

def cubeVolume() :
   return sideLength ** 3   # Error

main()
```

Note the scope of the variable `sideLength`. The `cubeVolume` function attempts to read the variable, but it cannot—the scope of `sideLength` does not extend outside the `main` function. The remedy is to pass it as an argument, as we did in Section 5.2.

It is possible to use the same variable name more than once in a program. Consider the `result` variables in the following example:

```
def main() :
   result = square(3) + square(4)
   print(result)

def square(n) :
   result = n * n
   return result

main()
```

Each `result` variable is defined in a separate function, and their scopes do not overlap.

Python also supports **global variables**: variables that are defined outside functions. A global variable is visible to all functions that are defined after it. However, any function that wishes to update a global variable must include a `global` declaration, like this:

```
balance = 10000   # A global variable

def withdraw(amount) :
   global balance   # This function intends to update the global balance variable
   if balance >= amount :
      balance = balance - amount
```

If you omit the `global` declaration, then the `balance` variable inside the `withdraw` function is considered a local variable.

Generally, global variables are not a good idea. When multiple functions update global variables, the result can be difficult to predict. Particularly in larger programs

In the same way that there can be a street named "Main Street" in different cities, a Python program can have multiple variables with the same name.

that are developed by multiple programmers, it is very important that the effect of each function be clear and easy to understand. You should avoid global variables in your programs.

SELF CHECK

For the Self Check problems that follow, consider this sample program.

```
 1  y = 8
 2
 3  def main() :
 4     x = 4
 5     x = mystery(x + 1)
 6     print(s)
 7
 8  def mystery(x) :
 9     s = 0
10     for i in range(x) :
11        x = i + 1
12        s = s + x
13     return s
```

30. Which lines are in the scope of the variable `i` used in line 10?
31. Which lines are in the scope of the parameter variable `x` defined in line 8?
32. The program defines two local variables with the same name whose scopes don't overlap. What are they?
33. Which line defines a global variable?
34. There is a scope error in the `main` function. What is it, and how do you fix it?

Practice It Now you can try these exercises at the end of the chapter: R5.8, R5.9.

Programming Tip 5.6

Avoid Global Variables

Programs with global variables are difficult to maintain and extend because you can no longer view each function as a "black box" that simply receives arguments and returns a result. When functions modify global variables, it becomes more difficult to understand the effect of function calls. As programs get larger, this difficulty mounts quickly. Instead of using global variables, use function parameter variables and return values to transfer information from one part of a program to another.

Global constants, however, are fine. You can place them at the top of a Python source file and access (but not modify) them in any of the functions in the file. Do not use a `global` declaration to access constants.