8. A turtle can leave an imprint of its shape on the screen by use of the _____ method.

9. In order to create a new turtle object, the _____ method is called.

ANSWERS: 1. False, 2. position, heading, pen, 3. absolute/relative 4. False, 5. False, 6. (a), 7. False, 8. stamp, 9. Turtle

COMPUTATIONAL PROBLEM SOLVING 6.3 Horse Race Simulation Program

In this section, we design, implement and test a program that simulates a horse race.

6.3.1 The Problem

The problem is to create a visualization of a horse race in which horses are moved ahead a random distance at fixed intervals until there is a winner, as shown in Figure 6-30.

FIGURE 6-30 Example Horse Race Simulation

6.3.2 Problem Analysis

The program needs a source of random numbers for advancing the horses a random distance in the race. We can use the random number generator of the Python standard library module random that we used in Chapter 3 in the Coin Change Exercise example. The remaining part of the problem is in the creation of appropriate graphics for producing a visualization of a horse race. We shall make use of the turtle graphics module from the Python standard library to do

this.

6.3.3 Program Design

Meeting the Program Requirements

There are no specific requirements for this problem, other than to create an appropriate simulation of a horse race. Therefore, the requirement is essentially the generation of a horse race in which the graphics look sufficiently compelling, and each horse has an equal chance of winning a given race. Since a specific number of horses was not specified, we will design the program for ten horses in each race.

Data Description

The essential information for this program is the current location of each of the ten horses in a given race. Each turtle is an object, whose attributes include its shape and its coordinate position on the turtle screen. Therefore, we will maintain a list of ten turtle objects with the shape attribute of a horse image for this purpose. Thus, suitable horse images must be found or created for this purpose.

Algorithmic Approach

There is no algorithm, per se, needed in this program other than to advance each horse a random distance at fixed time intervals until one of the horses reaches a certain point on the turtle screen (the “finish line”).

Overall Program Steps

The overall steps in this program design are given in Figure 6-31.

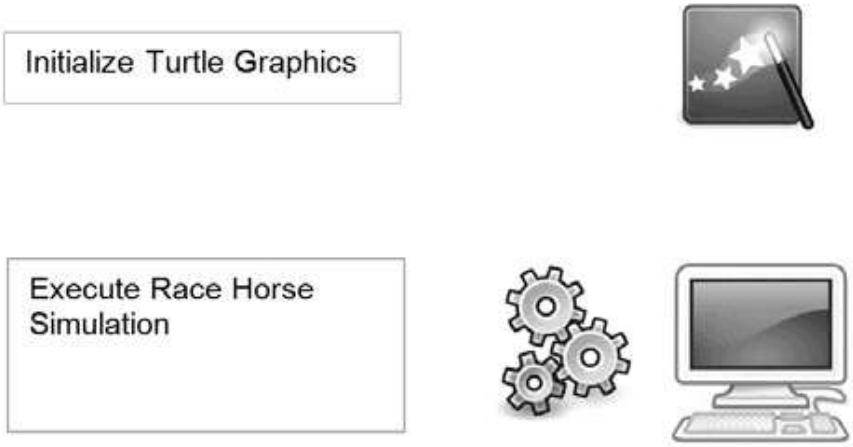


FIGURE 6-31 Overall Steps of the Horse Race Simulation Program

6.3.4 Program Implementation and Testing
Stage 1—Creating an Initial Turtle Screen Layout

We first develop and test an initial program that lays out the positions of the starting horses on the turtle graphics screen, as shown in Figure 6-32. Figure 6-33 provides this first stage of the program.

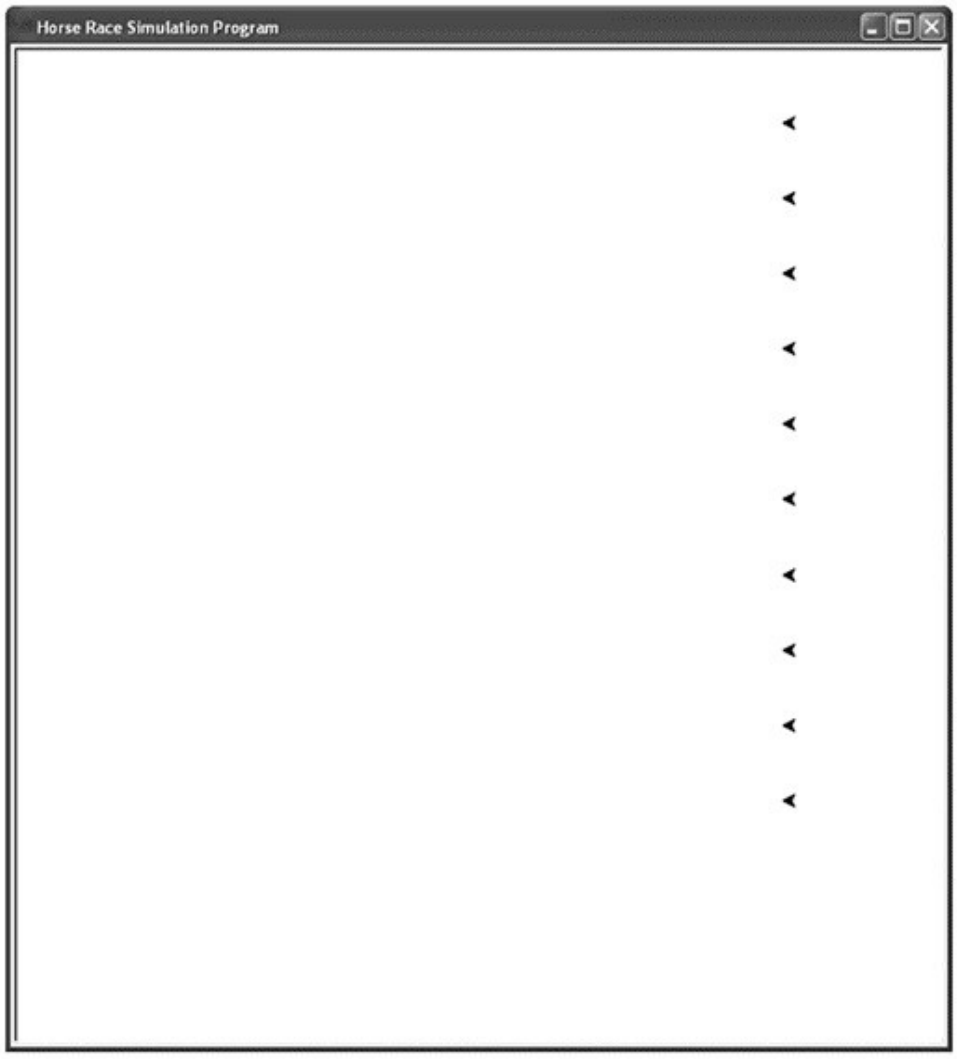


FIGURE 6-32
Output of Stage 1 of the Horse Race Simulation Program

At **line 3** the turtle module is imported. Since the import *module_name* form of import is used, each call to a method of this module must be prefixed with the module name. For example, turtle. setup(750, 800) on **line 31** (which sets the turtle screen size to a width of 750 and a height of 800 pixels).

The intent of this version of the program is to ensure that the turtle screen is appropriately sized and that the initial layout of horse locations is achieved. Therefore, only the default turtle shape is used at this point. In the next version we will focus on generating a set of horse images on the screen. Thus, on **line 34**, the turtle screen object is retrieved (by the call to turtle. Sreen()) and its reference assigned to variable window. The start location of the first (lowest)

horse is set to an x coordinate value of 240, and a y coordinate value of 2200. This puts the turtle screen object at the lower right corner of the screen. The amount of vertical separation between the horses is assigned to variable track_separation.These values were determined from knowledge of the screen coordinates in turtle graphics and a little trial and error.

Next, on **line 44** a call is made to function generateHorses (at **lines 9–15**). This function returns a list of ten new turtle objects, and assigned to variable horses. Function newHorse (**lines 5–7**) is called by function generateHorses to create each new horse turtle object. At this stage, function newHorse simply creates and returns a regular turtle object. In the next stage however, it will be responsible for returning new turtle objects with an appropriate horse shape.

The position for each of these horses is determined by function placeHorses on **lines 17– 23**. It is passed the list of horse turtle objects, the location of the first turtle, and the amount of separation between each (established as 60 pixels on **line 41**). Function placeHorses, therefore,

```
# Horse Racing Program (Stage 1)

import turtle

def newHorse():
    horse = turtle.Turtle()
    return horse

def generateHorses(num_horses):
    horses = []
    for k in range(0, num_horses):
        horse = newHorse()
        horses.append(horse)

    return horses

def placeHorses(horses, loc, separation):
    for k in range(0, len(horses)):
        horses[k].hideturtle()
        horses[k].penup()
        horses[k].setposition(loc[0], loc[1] + k * separation)
        horses[k].setheading(180)
        horses[k].showturtle()

# ---- main

# init number of horses
num_horses = 10

# set window size
turtle.setup(750, 800)

# get turtle window
window = turtle.Screen()

# set window title bar
window.title('Horse Race Simulation Program')

# init screen layout parameters
start_loc = (240, -200)
track_separation = 60

# generate and init horses
horses = generateHorses(num_horses)

# place horses at starting line
placeHorses(horses, start_loc, track_separation)

# terminate program when close window
turtle.exitonclick()
```

FIGURE 6-33 Stage 1 of the Horse Race Simulation Program

contains a for loop that iterates over the list of horse objects and makes them

initially hidden with their pen up (**lines 19–20**), moves each to its starting position (**line 21**), sets the heading of each to 180 degrees to move left (**line 22**), and then makes each visible (**line 23**). Finally, method exitonclick() is called so that the program will terminate when the user clicks on the program window's close box.

In the next stage, we further develop the program to include the specific shapes and images for the simulation.

Stage 2—Adding the Appropriate Shapes and Images

We next develop and test the program with additional code that adds the horse shapes (images) needed. The resulting turtle screen is shown in Figure 6-34. Figure 6-35 shows this second stage of the program.

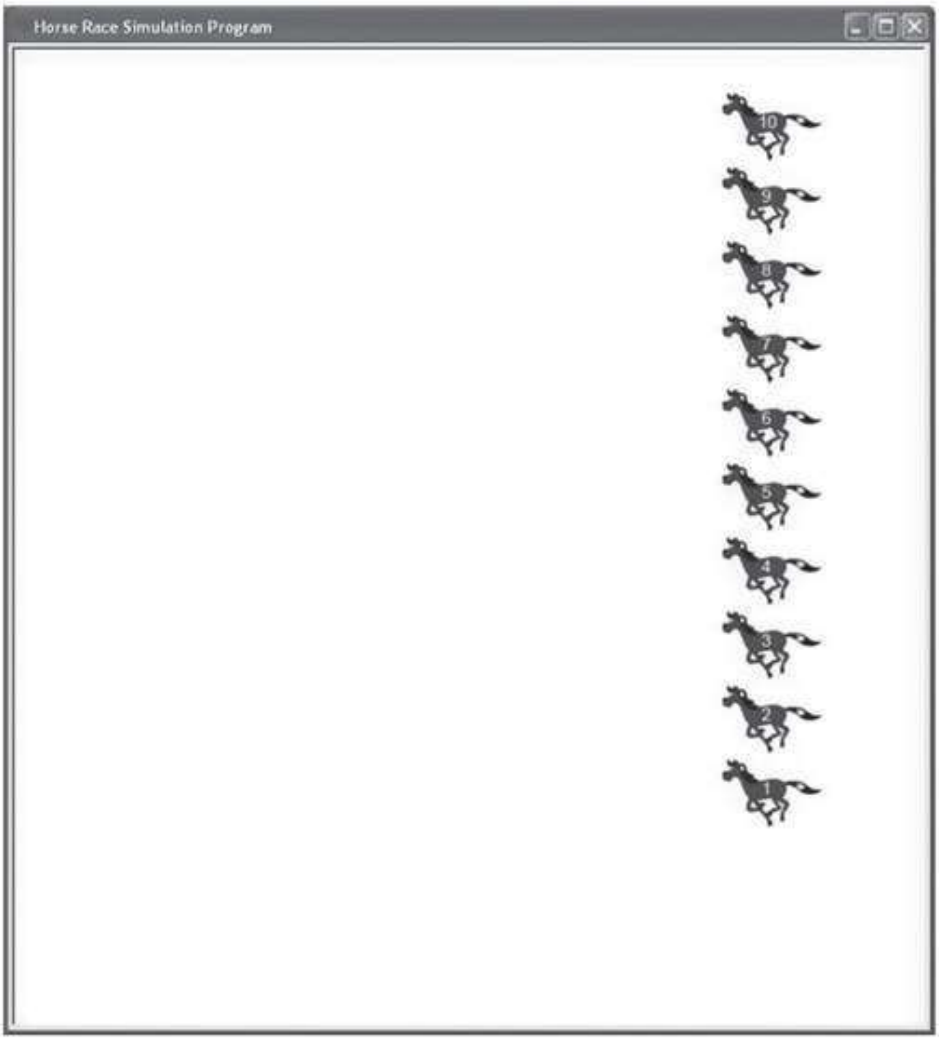


FIGURE 6-34
Output of Stage 2 of the Horse Race Simulation Program

In this stage of the program we add functions getHorseImages and registerHorseImages, called from **lines 61** and **62** of the main program section. Function getHorseImages returns a list of GIF image files. Each image contains the same horse image, each with a unique number 1 to 10. Function registerHorseImages does the required registering of images in turtle graphics by calling method turtle.register_shape on each.

Function generateHorses (**lines 26–32**) is implemented the same way as in stage 1 to return a list of horse turtle objects, except that it is altered to be passed an argument containing a list of horse images. Thus, the call to generateHorses in **line 65** is altered to pass the list of images in variable horse_images. Function newHorse (**lines 19–24**) is altered as well to be passed a particular horse image for the horse that is created, horse.shape(image_file).

Stage 3—Animating the Horses

Next we develop and test the program with additional code that animates the horses so that they are randomly advanced until a horse crosses the finish line. The resulting turtle screen is shown in Figure 6-36. Figure 6-37 provides this third stage of the program.

```
1  # Horse Racing Program (Stage 2)
2
3   import turtle
4
5  def getHorseImages(num_horses):
6      # init empty list
7      images = []
8
9      # get all horse images
10     for k in range(0, num_horses):
11         images = images + ['horse_' + str(k + 1) + '_image.gif']
12
13     return images
14
15 def registerHorseImages(images):
16     for k in range(0, len(images)):
17         turtle.register_shape(images[k])
18
19 def newHorse(image_file):
20     horse = turtle.Turtle()
21     horse.hideturtle()
22     horse.shape(image_file)
23
24     return horse
25
26 def generateHorses(images, num_horses):
27     horses = []
28     for k in range(0, num_horses):
29         horse = newHorse(images[k])
30         horses.append(horse)
31
32     return horses
33
34 def placeHorses(horses, loc, separation):
35     for k in range(0, len(horses)):
36         horses[k].hideturtle()
37         horses[k].penup()
38         horses[k].setposition(loc[0], loc[1] + k * separation)
39         horses[k].setheading(180)
40         horses[k].showturtle()
41
```

FIGURE 6-35 Stage 2 of the Horse Simulation Race Program (*Continued*)

Two new functions are added in this version of the program, startHorses and displayWinner. Function startHorses (**lines 44–58**) is passed the list of horse turtle objects, the location of the finish line (as an x coordinate value on the turtle screen) and the fundamental increment amount—each horse is advanced by one to three times this amount. The while loop for incrementally moving the horses is on **line 49**. The loop iterates until a winner is found, that is, until the

variable have_winner is True. Therefore, have_winner is initialized to False in **line 46**. Variable k, initialized on **line 48**, is used to index into the list of horse turtle objects. Since each horse in turn is advanced some amount during the race, variable k is incremented by one, modulo the number of horses in variable num_horses (10) (**line 57**). When k becomes equal to num_horses 21 (9), it is reset to 0 (for horse 1).

The amount that each horse is advanced is a factor of one to three randomly determined by call to method randint(1,3)of the Python standard library module random in **line 51**. Variable forward_incr is multiplied by this factor to move the horses forward an appropriate amount.

```
42 # ---- main
43
44 # init number of horses
45 num_horses = 10
46
47 # set window size
48 turtle.setup(750, 800)
49
50 # get turtle window
51 window = turtle.Screen()
52
53 # set window title bar
54 window.title('Horse Race Simulation Program')
55
56 # init screen layout parameters
57 start_loc = (240, -200)
58 track_separation = 60
59
60 # register images
61 horse_images = getHorseImages()
62 registerHorseImages(horse_images)
63
64 # generate and init horses
65 horses = generateHorses(horse_images)
66
67 # place horses at starting line
68 placeHorses(horses, start_loc, track_separation)
69
70 # terminate program when close window
71 turtle.exitonclick()
```

FIGURE 6-35 Stage 2 of the Horse Simulation Race Program

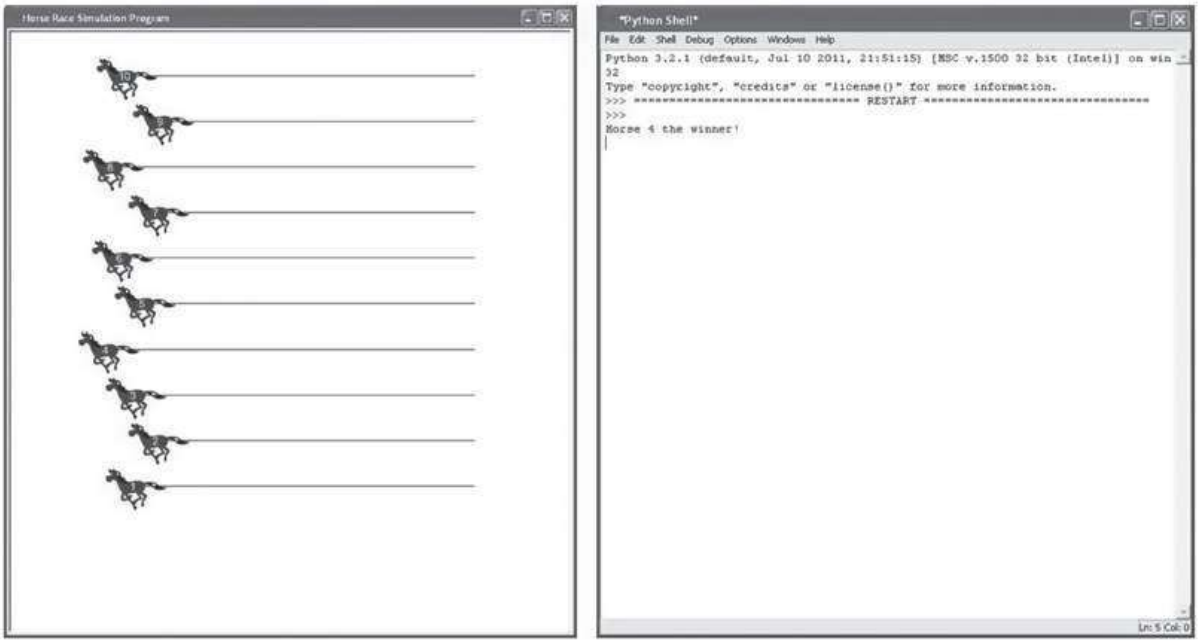


FIGURE 6-36 Output of Stage 3 of the Horse Race Simulation Program

```
# Horse Racing Program (Stage 3)

import turtle
import random

def getHorseImages(num_horses):
    # init empty list
    images = []

    # get all horse images
    for k in range(0, num_horses):
        images = images + ['horse_' + str(k+1) + '_image.gif']

    return images

def registerHorseImages(images):
    for k in range(0, len(images)):
        turtle.register_shape(images[k])

def newHorse(image_file):
    horse = turtle.Turtle()
    horse.hideturtle()
    horse.shape(image_file)

    return horse

def generateHorses(images, num_horses):
    horses = []
    for k in range(0, num_horses):
        horse = newHorse(images[k])
        horses.append(horse)

    return horses

def placeHorses(horses, loc, separation):
    for k in range(0, len(horses)):
        horses[k].hideturtle()
        horses[k].penup()
        horses[k].setposition(loc[0], loc[1] + k * separation)
        horses[k].setheading(180)
        horses[k].showturtle()
        horses[h].pendown()

def startHorses(horses, finish_line, forward_incr):
    # init
    have_winner = False

    k = 0
    while not have_winner:
        horse = horses[k]
        horse.forward(random.randint(1, 3) * forward_incr)

        # check for horse over finish line
        if horse.position()[0] < finish_line:
            have_winner = True
        else:
            k = (k + 1) % len(horses)
    return k

```

FIGURE 6-37 Stage 3 of the Horse Race Simulation Program (*Continued*)

```
60 def displayWinner(winning_horse):
61     print('Horse', winning_horse, 'the winner!')
62
63 # ---- main
64
65 # init number of horses
66 num_horses = 10
67
68 # set window size
69 turtle.setup(750, 800)
70
71 # get turtle window
72 window = turtle.Screen()
73
74 # set window title bar
75 window.title('Horse Race Simulation Program')
76
77 # init screen layout parameters
78 start_loc = (240, -200)
79 finish_line = -240
80 track_separation = 60
81 forward_incr = 6
82
83 # register images
84 horse_images = getHorseImages(num_horses)
85 registerHorseImages(horse_images)
86
87 # generate and init horses
88 horses = generateHorses(horse_images, num_horses)
89
90 # place horses at starting line
91 placeHorses(horses, start_loc, track_separation)
92
93 # start horses
94 winner = startHorses(horses, finish_line, forward_incr)
95
96 # display winning horse
97 displayWinner(winner + 1)
98
99 # terminate program when close window
100 turtle.exitonclick()
```

FIGURE 6-37 Stage 3 of the Horse Race Simulation Program

The value of forward_incr is initialized in the main program section. This value can be adjusted to speed up or slow down the overall speed of the horses.

Function displayWinner displays the winning horse number in the Python shell (**lines 60–61**). This function will be rewritten in the next stage of program

development to display a “winner” banner image in the turtle screen. Thus, this implementation of the function is for testing purposes only.

The main program section (**lines 63–100**) is the same as in the previous stage of program development, except for the inclusion of the calls to functions startHorses and displayWinner in **lines 94** and **97**.

Final Stage—Adding Race Banners

Finally, we add the code for the displaying of banners at various points in the race as shown earlier in Figure 6-30. In Figure 6-38 is the final stage of the program. This final version imports one additional module, Python Standard Library module time (**line 5**), used to control the blink rate of the winning horse.

While the race progresses within the while loop at **line 102**, checks for the location of the lead horse are made in two places—before and after the halfway mark of the race (on **line 108**). If the

```
# Horse Racing Program (Final Stage)

import turtle
import random
import time

def getHorseImages(num_horses):
    # init empty list
    images = []

    # get all horse images
    for k in range(0, num_horses):
        images = images + ['horse_' + str(k + 1) + '_image.gif']

    return images

def getBannerImages(num_horses):
    # init empty list
    all_images = []

    # get "They're Off" banner image
    images = ['theyre_off_banner.gif']
    all_images.append(images)

    # get early lead banner images
    images = []
    for k in range(0, num_horses):
        images = images + ['lead_at_start_' + str(k + 1) + '.gif']
    all_images.append(images)

    # get mid-way lead banner images
    images = []
    for k in range(0, num_horses):
        images = images + ['looking_good_' + str(k + 1) + '.gif']
    all_images.append(images)

    # get "We Have a Winner" banner image
    images = ['winner_banner.gif']
    all_images.append(images)

    return all_images

def registerHorseImages(images):
    for k in range(0, len(images)):
        turtle.register_shape(images[k])

```

FIGURE 6-38 Final Stage of the Horse Race Simulation Program (*Continued*)

```
47  def registerBannerImages(images):
48      for k in range(0, len(images)):
49          for j in range(0, len(images[k])):
50              turtle.register_shape(images[k][j])
51
52  def newHorse(image_file):
53      horse = turtle.Turtle()
54      horse.hideturtle()
55      horse.shape(image_file)
56
57      return horse
58
59  def generateHorses(images, num_horses):
60      horses = []
61      for k in range(0, num_horses):
62          horse = newHorse(images[k])
63          horses.append(horse)
64
65      return horses
66
67  def placeHorses(horses, loc, separation):
68      for k in range(0, len(horses)):
69          horses[k].hideturtle()
70          horses[k].penup()
71          horses[k].setposition(loc[0], loc[1] + k * separation)
72          horses[k].setheading(180)
73          horses[k].showturtle()
74          horses[k].pendown()
75
76  def findLeadHorse(horses):
77      # init
78      lead_horse = 0
79
80      for k in range(1, len(horses)):
81          if horses[k].position()[0] <  \
82             horses[lead_horse].position()[0]:
83              lead_horse = k
84      return lead_horse
85
86  def displayBanner(banner, position):
87      the_turtle = turtle.getturtle()
88      the_turtle.setposition(position[0], position[1])
89      the_turtle.shape(banner)
90      the_turtle.stamp()
91
```

FIGURE 6-38 Final Stage of the Horse Race Simulation Program (*Continued*)

x coordinate location of the lead horse is less then 125, the “early lead banner” is displayed on **line 117** by a call to function displayBanner. Otherwise, if one second has elapsed, then the “midrace lead banner” is displayed on **line 111**.

The sleep method of the time module is used to control the blinking of the winning horse in function displayWinner. A “count-down” variable, blink_counter, is set to 5 on **line 133**. This will cause the winning horse to blink five times. The following while loop decrements

```
92  def startHorses(horses, banners, finish_line, forward_incr):
93      # init
94      have_winner = False
95      early_leading_horse_displayed = False
96      midrace_leading_horse_displayed = False
97
98      # display "They're Off" banner image
99      displayBanner(banner_images[0][0], (70, -300))
100
101     k = 0
102     while not have_winner:
103         horse = horses[k]
104         horse.forward(random.randint(1, 3) * forward_incr)
105
106         # display mid-race lead banner
107         lead_horse = findLeadHorse(horses)
108         if horses[lead_horse].position()[0] < -125 and \
109            not midrace_leading_horse_displayed:
110
111             displayBanner(banners[2][lead_horse], (40, -300))
112             midrace_leading_horse_displayed = True
113
114         # display early lead banner
115         elif horses[lead_horse].position()[0] < 125 and \
116            not early_leading_horse_displayed:
117             displayBanner(banners[1][lead_horse], (10, -300))
118             early_leading_horse_displayed = True
119
120         # check for horse over finish line
121         if horse.position()[0] < finish_line:
122             have_winner = True
123         else:
124             k = (k + 1) % len(horses)
125     return k
126
```

FIGURE 6-38 Final Stage of the Horse Race Simulation Program (*Continued*)

blink_counter and continues to iterate until blink_counter is 0. Variable show, initialized to False on **line 132**, is used to alternately show and hide the turtle based on its current (Boolean) value, which is toggled back and forth between True and False each time through the loop. The sleep method is called on **line 143** to cause the program execution to suspend for four-tenths of a second so that the switch between the visible and invisible horse appears slowly enough to

cause a blinking effect. This version of displayWinner replaces the previous version that simply displayed the winning horse number in the Python shell window.

Added functions getBannerImages (**lines 17–41**), registerBannerImages (**lines 47–50**), and displayBanner (**lines 86–90**) incorporate the banner images into the program the same way that the horse images were incorporated in the previous program version. Function startHorses was modified to take another parameter, banners, containing the list of registered banners displayed during the race, passed to it from the main program section.

```
127 def displayWinner(winning_horse, winner_banner):
128     # display "We Have a Winner" banner
129     displayBanner(winner_banner, (20, -300))
130 
131     # blink winning horse
132     show = False
133     blink_counter = 5
134     while blink_counter != 0:
135         if show:
136             winning_horse.showturtle()
137             show = False
138             blink_counter = blink_counter - 1
139         else:
140             winning_horse.hideturtle()
141             show = True
142 
143         time.sleep(.4)
144 
145 # ---- main
146 
147 # init number of horses
148 num_horses = 10
149 
150 # set window size
151 turtle.setup(750, 800)
152 
153 # get turtle window
154 window = turtle.Screen()
155 
156 # set window title
157 window.title('Horse Race Simulation Program')
158 
159 # hide default turtle and keep from drawing
160 the_turtle.hideturtle()
161 the_turtle.penup()
162 
163 # init screen layout parameters
164 start_loc = (240, -200)
165 finish_line = -240
166 track_separation = 60
167 forward_incr = 6
168 
169 # register images
170 horse_images = getHorseImages()
171 banner_images = getBannerImages()
172 registerHorseImages(horse_images)
173 registerBannerImages(banner_images)
174 
```

FIGURE 6-38 Final Stage of the Horse Race Simulation Program (*Continued*)

Finally, the default turtle (created with the turtle graphics window) is utilized in

function displayBanners and in the main section. It is used to display the various banners at the bottom of the screen. To do this, the turtle's "shape" is changed to the appropriate banner images stored in list banner_images. To prevent the turtle from drawing lines when moving from the initial (0, 0)coordinate location to where banners are displayed, the default turtle is hidden and its pen attribute is set to "up" (**lines 160–161**).

Chapter Exercises 243

```
175 # generate and init horses
176 horses = generateHorses(horse_images)
177
178 # place horses at starting line
179 placeHorses(horses, start_loc, track_separation)
180
181 # start horses
182 winner = startHorses(horses, banner_images, finish_line,
183                      forward_incr)
184
185 # light up for winning horse
186 displayWinner(horses[winner], banner_images[3][0])
187
188 # terminate program when close window
189 turtle.exitonclick()
```

FIGURE 6-38 Final Stage of the Horse Race Simulation Program

CHAPTER SUMMARY General Topics

Software Objects/Methods
References/Reference vs. Dereferenced Values Reference Assignment
Memory Allocation/Deallocation
Garbage Collection
Shallow vs. Deep Copy Operations

Python-Specific Programming Topics Objects and Turtle Graphics in Python
CHAPTER EXERCISES Section 6.1
1. Indicate exactly what the contents of lst1 and lst2 would be after each of the following set of assignments,

(a) lst1 5 [10, 20, 30] **(b)** lst1 5 [10, 20, 30] **(c)** lst1 5 [10, 20, 30] lst2 5 [10, 20, 30] lst2 5 lst1 lst2 5 list(lst1) lst1[2] 5 50 lst1[2] lst1[2]

2. Indicate which of the following set of assignments would result in automatic

garbage collection in Python.

(a) lst1 5 [1, 2, 3] **(b)** str1 5 'Hello World' **(c)** tuple1 5 (1, 2, 3) lst2 5 [5, 6, 7] str2 ' Nice Day ' tuple2 tuple1 lst1 lst2 str3 tuple1

3. For the set of assignments in question 1, indicate how both the id method and is operator can be used to determine if lists lst1 and lst2 are each referencing the same list instance in memory.
Section 6.2
4. Give a set of instructions to create a turtle window of size 400 pixels wide and 600 pixels high, with a title of 'Turtle Graphics Window'.

5. Give a set of instructions that gets the default turtle and sets it to an actual turtle shape. **6.** For each of the following method calls on turtle the_turtle, indicate in what part of the screen the turtle will be placed relative to the center of the screen.
(a) the_turtle.setposition(0, 0)
(b) the_turtle.setposition(2100, 0)
(c) the_turtle.setposition(250, 0)
(d) the_turtle.setposition(0, 250)

7. For the following method calls on turtle the_turtle, describe the shape that will be drawn. the_turtle.penup()
the_turtle.setposition(2100, 0)
the_turtle.pendown()
the_turtle.setposition(100, 0)
the_turtle.setposition(100, 50)
the_turtle.setposition(2100, 50)
the_turtle.setposition(2100, 0)

8. What color line will be drawn in the following?
turtle.colormode(255)
the_turtle.pencolor(128, 0, 0)
the_turtle.pendown()
the_turtle.forward(100)

9. What will be displayed by the following turtle actions?
the_turtle.pendown()
the_turtle.showturtle()

```
the_turtle.forward(25)
the_turtle.penup()
the_turtle.hide_turtle()
the_turtle.forward(25)
the_turtle.pendown()
the_turtle.showturtle()
the_turtle.forward(25)
```

PYTHON PROGRAMMING EXERCISES

P1. Give a set of instructions for controlling the turtle to draw a line from the top-left corner of the screen to the bottom-right corner, and from the top-right corner to the bottom-left corner, thereby making a big X on the screen. There should be no other lines drawn on the screen.

P2. Using relative positioning, give a set of instructions for controlling the turtle to draw an isosceles triangle on the screen (that is, a triangle with two equal-length sides).

P3. Give a set of instructions for controlling the turtle to draw the letter W using relative positioning.

P4. Give a set of instructions for controlling the turtle to create three concentric circles, each of different color and line width.

P5. Give a set of instructions that sets the turtle to an actual turtle shape, and moves it from the bottom of the screen towards the top, getting smaller as it moves along.

P6. Give a set of instructions that moves the turtle with an actual turtle shape from the bottom of the screen toward the top, changing its fill color when it crosses the x axis of the grid coordinates.

P7. Give a set of instructions to create your own polygon shape and create an interesting design with it.

Program Modification Problems 245

P8. Give a set of instructions so that the turtle initially moves slowly around the edge of the screen, then moves faster and faster as it goes around.

P9. Give a set of instructions to create two turtle objects each with circle shape that move to various locations of the turtle screen, each stamping their circle shape of varying sizes and colors.

PROGRAM MODIFICATION PROBLEMS

M1. Bouncing Balls with Color
Modify the bouncing balls simulation program so that exactly three balls are created, each with a different color.

M2. Bouncing Balls with Changing Color
Modify the bouncing balls simulation program so that each time a ball hits an edge of the turtle graphics screen, it changes color.

M3. Bouncing Balls with Trailing Lines
Modify the bouncing balls simulation program so that a trail is left on the screen of each ball's path.

M4. Bouncing Ball Chase
Modify the bouncing balls simulation program so that there are exactly three balls generated, with the first ball started in a random direction (heading), and the other two balls following it closely behind.

M5. Horse Racing Program: Multiple Races and Score Keeping
Modify the Horse Racing program so that the user can continue to play another race without having to rerun the program. Also, the cumulative wins of all the horses should be displayed in the shell window. This allows the user to see if some horses are a more "winning" horse than others.

M6. Horse Racing Program: Handicap Racing
Modify the Horse Racing program so that the user can assign a handicap to one or more horses on a scale of 1 to 5. (A "handicap" in racing is a means of giving advantage to less competitive horses over more competitive ones.) If a horse is assigned a handicap of 1, it should move ahead one-fifth farther than usual. A handicap of 2 would increase its move by two-fifths, and so forth. The list of handicaps should be displayed in the shell window each time before the race begins.

M7. Horse Racing Program: Pari-mutuel Betting
Modify the Horse Racing program to allow individuals to enter their name to "register themselves" to place bets. The program should be modified so that races can be consecutively run without having to restart the program. Before each race, bets can be placed by registered players. Each bet is for which horse will win. The payout will be based on the rules of *pari-mutuel betting* described below. The amount of money gained or lost by each registered player should be constantly displayed in the shell.

Example of pari-mutuel betting
Each horse has a certain amount of money wagered on it (assuming eight horses):
12345678 $30.00 $70.00 $12.00 $55.00 $110.00 $47.00 $150.00 $40.00

Thus, the total *pool* of money on this particular wagering event is $514.00. Following the start of the event, no more wagers are accepted. The event is decided and the winning outcome is determined to be outcome 4 with $55.00 wagered. The *payout* is now calculated. First, the *commission* or *take* for the wagering company is deducted from the pool. For example, with a commission rate of 14.25% the pool is: $514 3 (1 2 0.1425) 5 $440.76. This remaining amount in the pool is now distributed to those who wagered on outcome 4: $440.76 / $55 ≈ $8 per $1 wagered. This payout includes the $1 wagered plus an additional $7 profit. Thus, the odds of outcome 4 are 7-to-1.

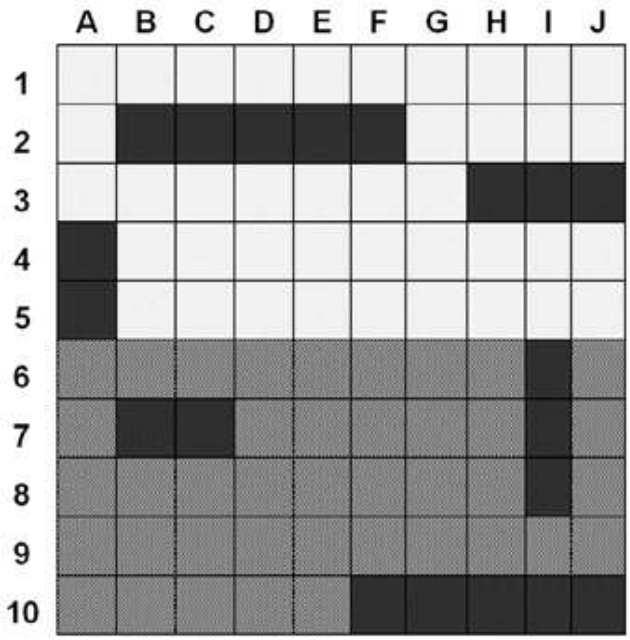
Wikipedia contributors. “Parimutuel betting.” *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, May 7, 2011. Web, May 11, 2011.
PROGRAM DEVELOPMENT PROBLEMS

D1. Drunkard’s Walk
A *random walk* is a trajectory taken by a sequence of random steps. Random walks can be used to model the travel of molecules, the path that animals take when looking for food, and financial fluctuations, for example. A specific form of random walk is called the “Drunkard’s Walk.” A (drunken) man tries to find his way home. He does so by making a random choice at each street intersection of which of the four paths to take: continue in the same direction; go back from the direction he came; turn left; or turn right. Thus, the man is traveling the same distance after each choice of direction (one city block). Implement and test a Python program using turtle graphics to display random walks. Select an appropriate number of pixels as the length of a city block.

D2. Name Reversal
Implement and test a Python program using turtle graphics to allow the user to enter their first name, and have it displayed in the turtle window as a reverse mirror image.

D3. Battleship Game Visualization
Implement and test a Python program using turtle graphics to provide a

visualization for the game of Battleship discussed in Program Development problem D3 in Chapter 4.



Modular Design CHAPTER 7

Until now, we have looked at programs comprised of a set of individual functions. In complex software systems, however, programs are organized at a higher level as a set of modules, each module containing a set of functions (or objects). Modules, like functions, are a fundamental building block in software development.

OBJECTIVES
After reading this chapter and completing the exercises, you will be able to: ♦ Explain the use of modular design in software development ♦ Explain the specification of modules
♦ Explain the process of top-down design
♦ Describe the concept of a stack
♦ Differentiate between unit testing and integration testing ♦ Become familiar with the use of docstrings in Python ♦ Explain the use of modules and namespaces in Python ♦ Describe and use the different forms of module import in Python ♦ Develop well-specified Python programs using top-down design

CHAPTER CONTENTS
Motivation
Fundamental Concepts
7.1 Modules
7.2 Top-Down Design
7.3 Python Modules
Computational Problem Solving
7.4 Calendar Year Program (function version)

MOTIVATION

Software systems are some of the most complex entities ever created. Web browsers and operating systems, for example, contain as many as 50–100 million lines of code. Developing programs of such size and complexity can easily take 10,000 person-years of effort to develop. It is natural, therefore, to find ways to divide the task of software development among various individuals (or groups of individuals).

We can make complex systems more manageable by designing them as a set of subsystems, or modules. For example, NASA's space shuttle (Figure 7-1) is one of the most complex systems ever engineered, containing more than 2.5 million parts. The major components are the orbiter vehicle, a large external liquid-fuel tank, and two solid rocket boosters. The orbiter vehicle itself is composed of several subsystems (e.g., the communications system), which in turn may be composed of sub-subsystems (e.g., the data network system), as shown in Figure 7-2.

FIGURE 7-1 Space Shuttle

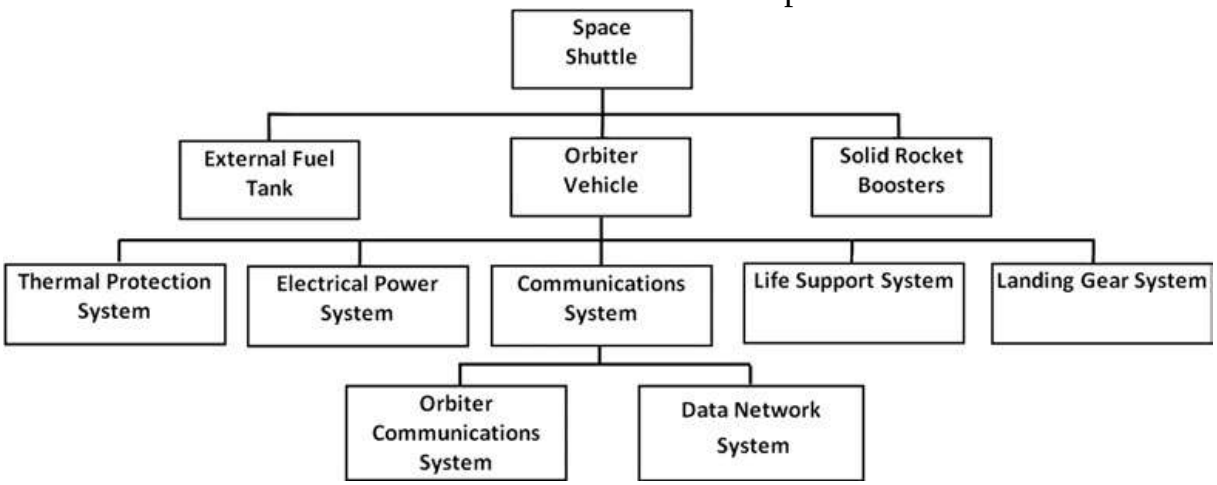


FIGURE 7-2 Modular Design of the NASA Space Shuttle

Although many of the technicians involved with the shuttle may understand its overall design, only a few select individuals need to understand or be involved with the detailed design, implementation, and testing of any specific subsystem. The same is true for the modular design of software. In this chapter, we look at the issue of modular program design.

FUNDAMENTAL CONCEPTS

7.1 Modules

7.1.1 What Is a Module?

An important aspect of well-designed software is that programs are designed as a

collection of modules. The term “module,” broadly speaking, refers to the design and/or implementation of
7.1 Modules 249

specific functionality to be incorporated into a program. While an individual function may be considered a module, modules generally consists of a collection of functions (or other entities). The Python turtle module is an example of a software module. The use of modules has a number of advantages as shown in Figure 7-3.

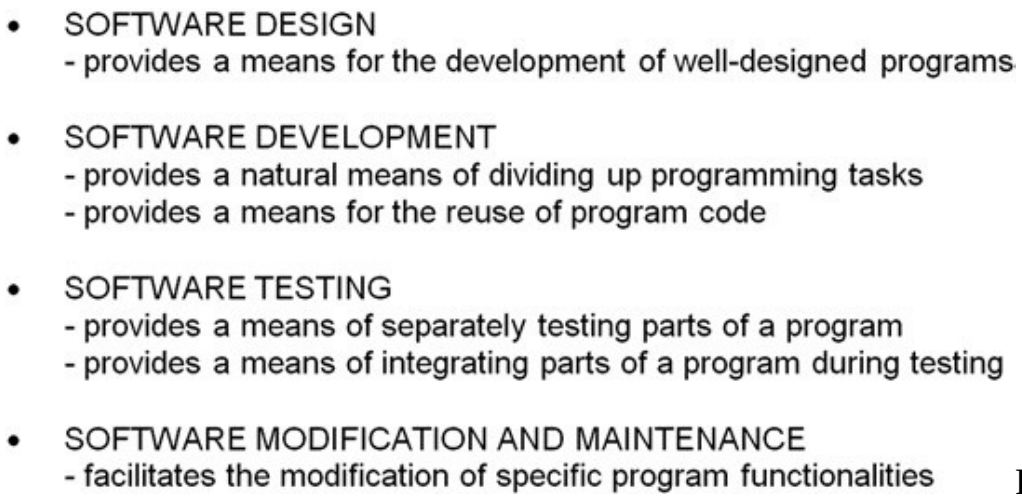


FIGURE 7-3
Advantages of Modular Programming

Modular design allows large programs to be broken down into manageable size parts, in which each part (module) provides a clearly specified capability. It aids the software development process by providing an effective way of separating programming tasks among various individuals or teams. It allows modules to be individually developed and tested, and eventually integrated as a part of a complete system. Finally, modular design facilitates program modification since the code responsible for a given aspect of the software is contained within specific modules, and not distributed throughout the program.

The term “**module**” refers to the design and/or implementation of specific functionality to be incorporated into a program.

7.1.2 Module Specification

Every module needs to provide a *specification* of how it is to be used. This is referred to as the module’s **interface**. Any program code making use of a particular module is referred to as a **client** of the module. A module’s

specification should be sufficiently *clear* and *complete* so that its clients can effectively utilize it. For example, numPrimes is a function that returns the number of primes in a given integer range, as shown in Figure 7-4.

```
def numPrimes(start, end):

    """ Returns the number of primes between start and end. """
```

FIGURE 7-4 docstring Specification

The function's specification is provided by the line immediately following the function header, called a *docstring* in Python. A **docstring** is a string literal denoted by triple quotes given as the first line of certain program elements. The docstring of a particular program element can be displayed by use of the __doc__ extension,

```
... print(numPrimes.__doc__)
Returns the number of primes between start and end.
```

This provides a convenient way for discovering how to use a particular function without having to look at the function definition itself. Some software development tools also make use of docstrings. Let's consider how complete this specification is for this function. At first look it may seem sufficient. However, it does not answer whether the number of primes returned includes the endpoints of the range or not. Also, it is not clear what will happen if the function is called with a first argument (start) greater than the second (end). Thus, a more complete specification is needed. This is given in Figure 7-5.

```
def numprimes(start, end):

    """Returns the number of primes between start and end, inclusive.

       Returns -1 if start is greater than end.
    """
```

FIGURE 7-5 A More Complete docstring Specification

This is now a reasonable specification of the function. This docstring follows the Python convention of putting a blank line after the first line of the docstring, which should be an overall description of what the function does, followed by an arbitrary number of lines providing additional details. These additional lines must be indented at the same level, as shown in the figure. Appropriate use of

this function by a client is given below.

```
.
.
first_num 5 int(input('Enter the start of the range: '))
second_num 5 int(input('Enter the end of the range: '))

result 5 numprimes(first_num, second_num)
if result 55 21:
print('* Invalid range entered *')
else:
print('The number of primes between', first_num, 'and', second_num, 'is', result)
```

In this example, the user inputs a start and end value for the range of integers to check. Since the user may inappropriately enter a start value greater than the value of the end value, a check is made for a returned value of 21 after the call to numprimes. If 21 is found, then an error message is output; otherwise, the result is displayed.

There are potential problems when returning both a computed result and an error result as a function's return value. First, there is no guarantee that the client will perform the necessary check for the special error value. Thus, an incorrect result may be displayed to the user,

The number of primes between 100 and 1 is 21

Second, there may not be a special value that *can* be returned for error reporting. For example, if there is a function meant to return any integer value, including negative numbers, there is no special integer value that can be returned. We will see a better means of error reporting by a function when we discuss exception handling in Chapter 8.

A module's **interface** is a specification of what it provides and how it is to be used. Any program code making use of a given module is called a **client** of the module. A **docstring** is a string literal denoted by triple quotes used in Python for providing the specification of certain program elements.

Self-Test Questions

1. Which of the following is not an advantage in the use of modules in software

development? **(a)** Provides a natural means of dividing up programming tasks.
(b) Provides a means of reducing the size of a program.
(c) Provides a means for the reuse of program code.
(d) Provides a means of separately testing individual parts of a program.
(e) Provides a means of integrating parts of a program during testing.
(f) Facilitates the modification of specific program functionalities.

2. A specification of how a particular module is used is called the module's _____.
3. Program code that makes use of a given module is called a _____ of the module.

4. Indicate which of the following are true. A docstring in Python is
(a) A string literal denoted by triple or double quotes.
(b) A means of providing specification for certain program elements in Python.
(c) A string literal that may span more than one line.

ANSWERS: 1. (b), 2. interface, 3. client, 4. (b), (c)

7.2 Top-Down Design

One method of deriving a modular design is called **top-down design**. In this approach, the overall design of a system is developed first, deferring the specification of more detailed aspects of the design until later steps. We next consider a modular design using a top-down approach for the calendar year program from Chapter 4.

Top-down design is an approach for deriving a modular design in which the overall design of a system is developed first, deferring the specification of more detailed aspects of the design until later steps.

7.2.1 Developing a Modular Design of the Calendar Year Program

We will develop a modular design for the calendar year program from Chapter 4 (implemented there without the use of functions) using a top-down design approach. The three overall steps of the program are getting the requested year from the user, creating the calendar year structure, and displaying the year. This is depicted in Figure 7-6.

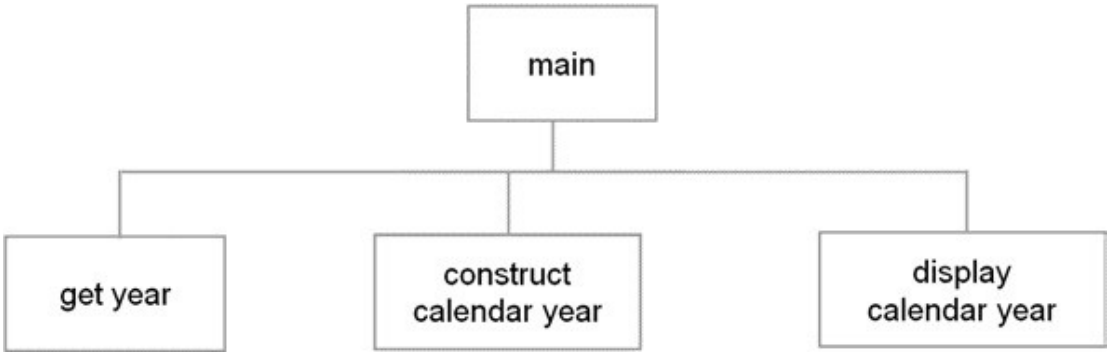


FIGURE 7-6 First Stage of a Modular Design of the Calendar Year Program

We then consider whether any of these modules needs to be further broken down. Making such a decision is more of an art than a science. The goal of modular design is that each module provides clearly defined functionality, which collectively provide all of the required functionality of the program. Modules get year and display calendar year are not complex enough to require further breakdown. Module construct calendar year, on the other hand, is where most of the work is done, and is therefore further broken down. Figure 7-7 contains the modules of this next design step.

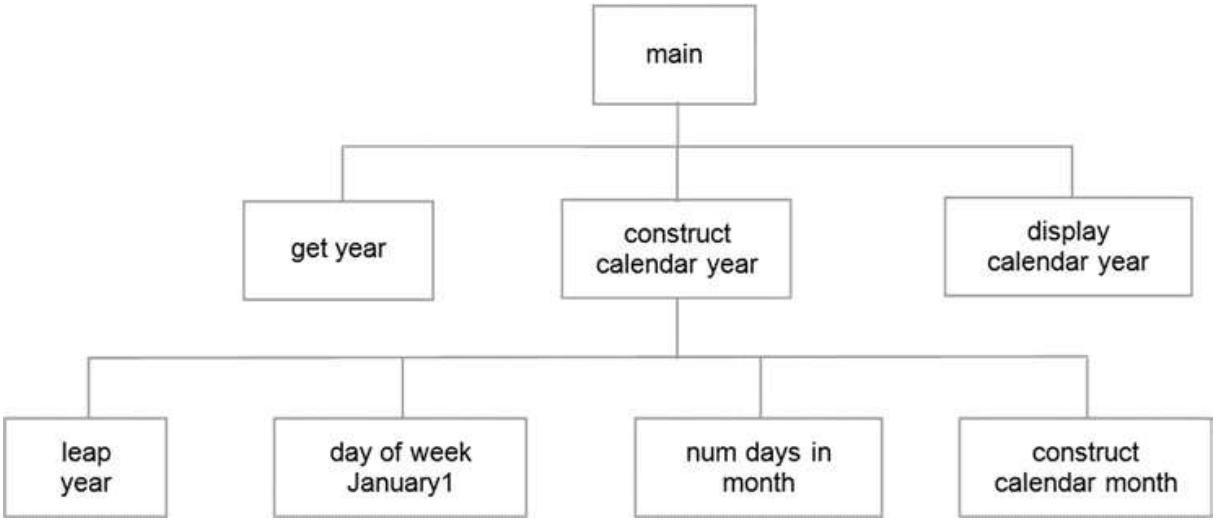


FIGURE 7-7 Second Stage of Modular Design of a Calendar Year Program

In order to construct a calendar year, it must be determined whether the year is a leap year, what day of the week January 1st of that year falls on, and how many days are in each month (accounting for leap years). Thus, modules leap year, day of week January1, and num days in month are added as *submodules* of module construct calendar year. The calendar month for each of the twelve months must then be individually constructed, handled by module construct calendar month.

The goal of top-down design is that each module provides clearly defined functionality, which collectively provide all of the required functionality of the program.

7.2.2 Specification of the Calendar Year Program Modules

The modular design of the calendar year program provides a high-level view of the program. However, there are many issues yet to resolve in the design. Since each module is to be implemented as a function, we need to specify the details of each function. For example, for each function it needs to be decided if it is a value-returning function or a non-value-returning function; what parameters it will take; and what results it will produce. We give such a specification in Figure 7-8 using Python docstrings.

This stage of the design provides sufficient detail from which to implement the program. The main module provides the overall construction of the program. It simply displays the program

```
def getYear():

    """ Returns an integer value between 1800-2099, inclusive, or  -1."""

def leapYear(year):

    """ Returns True if provided year a leap year, otherwise returns False."""

def dayOfWeekJan1(year, leap_year):

    """ Returns the day of the week for January 1 of the provided year.

        year must be between 1800 and 2099. leap_year must be True if
        year a leap year, and False otherwise.
    """

def numDaysInMonth(month_num, leap_year):

    """ Returns the number of days in a given month.

        month_num must be in the range 1-12, inclusive.
        leap_year must be True if month in a leap year, otherwise False.
    """

def constructCalMonth(month_num, first_day_of_month, num_days_in_month):

    """ Returns a formatted calendar month for display on the screen.

        month_num must be in the range 1-12, inclusive.
        first_day_of_month must be in the range 0-6 (1-Sun, 2-Mon, ..., 0-Sat).

        Returns a list of strings of the form,
        [month_name, week1, week2, week3, week4, week5, week6].
     """

def constructCalYear(year):

    """ Returns a formatted calendar year for display on the screen.

        year must be in the range 1800-2099, inclusive.
        Returns a list of list of strings of the form,

        [year, month1, month2, month3, …, month12].
     """

def displayCalendar(calendar_year):

     """ Displays the provided calendar_year on the screen three months across.

         Provided calendar year should be in the form of a list of twelve sublists, in which each
         sublist is of the form [monthname, week1, week2, …, ]
     """
```

FIGURE 7-8 Calendar Year Module Specification (*Continued*)

```
# main
terminate = False

print('This program will display a calendar year for a given year')

while not terminate:
    year = getYear()

    if year ==  -1:
       terminate = True
    else:
        calendar_year = constructCalYear(year)
        displayCalendar(calendar_year)
```

FIGURE 7-8 Calendar Year Module Specification

greeting, calls module getYear to get the year from the user, calls module constructCalYear to construct the year, and finally calls module displayCalendar to display the calendar year. The only detail that the main module is concerned with is allowing the user to keep displaying another calendar year, or enter 21 to terminate the program. This is controlled by Boolean variable terminate.

The first module called, getYear, returns the integer value entered by the user. The module's specification indicates that it returns an integer value between 1800 and 2099, inclusive; or 21 (if the user decides to terminate the program). Therefore, it is the responsibility of the module to ensure that no other value is returned. This relieves the main module of having to check for bad input.

The next module that the main module uses is module constructCalYear. This module returns a list of twelve sublists, one for each month, in which each sublist begins with the name of the month (as a string value), followed by each of the weeks in the month, each week formatted as a single string. The module's specification indicates that it is to be passed a year between 1800 and 2099, inclusive. Therefore, any year given to it that is outside that range violates its condition for use, and therefore the results are not guaranteed.

The last module called from the main module is module displayCalendar. It is given a constructed calendar year, as constructed by module constructCalMonth. Based on the modular design of the calendar year program, constructCalYear is the only module relying on the use of submodules, specifically modules leapYear, dayOfWeekJan1, numDaysInMonth, and constructCal. The leapYear module determines whether a given a year is a leap year or not, returning a

Boolean result. Module dayOfWeekJan1 returns the day of the week for January 1st of the provided year. Boolean value leap_year must also be provided to the module, needed in the day of the week algorithm on which the module is based. Module numDaysInMonth must be passed an integer in the range 1–12, as well as a Boolean value for leap_year. This is so that the module can determine the number of days in the month for the month of February. Finally, constructCalMonth is given a month number, the day of the week of the first day of the month (1-Sun, 2-Mon, . . ., 0-Sat) and the number of days in the month. With this information, module constructCalYear can construct the calendar list and its sublists to be displayed.
Finally, module displayCalendarMonth is given a formatted calendar year. Its job is to display the calendar year three months across.
This more detailed modular design provides the details of how each module is to be incorporated into a complete program. We will discuss the implementation of this design at the end of the chapter.

Self-Test Questions

1. In top-down design (select one),
(a) The details of a program design are addressed before the overall design. **(b)** The overall design of a program is addressed before the details.

2. All modular designs are a result of a top-down design process. (TRUE/FALSE) **3.** In top-down design, every module is broken down into the same number of submodules. (TRUE/FALSE)

4. Which of the following advantages of modular design apply to the design of the calendar program.
(a) Provides a means for the development of well-designed programs.
(b) Provides a natural means of dividing up programming tasks.
(c) Provides a means of separately testing individual parts of a program.

ANSWERS: 1. (b), 2. False, 3. False, 4. (a), (b), (c)

7.3 Python Modules

7.3.1 What Is a Python Module?

A Python module is a file containing Python definitions and statements. When a Python file is directly executed, it is considered the *main module* of a program. Main modules are given the special name __main__. Main modules provide the

basis for a complete Python program. They may *import* (include) any number of other modules (and each of those modules import other modules, etc.). Main modules are not meant to be imported into other modules.

As with the main module, imported modules may contain a set of statements. The statements of imported modules are executed only once, the first time that the module is imported. The purpose of these statements is to perform any initialization needed for the members of the imported module. The Python Standard Library contains a set of predefined *Standard (built-in) modules*. We have in fact seen some of these modules already, such as the math and random Standard Library modules.

Python modules provide all the benefits of modular software design we have discussed. By convention, modules are named using all lower case letters and optional underscore characters. We will look more closely at Python modules in the next section.

LET'S TRY IT Create a Python module by entering the following in a file name simple.py. Then execute the instructions in the Python shell as shown and observe the results. # module simple
print('module simple loaded')

def func1():
print('func1 called') ... import simple ???
... simple.func1() ???

def func2():
print('func2 called') ... simple.func2() ???

A **Python module** is a file containing Python definitions and statements. The Python Standard Library contains a set of predefined **standard (built-in) modules**.
7.3.2 Modules and Namespaces

A **namespace** is a container that provides a named context for a set of identifiers. Namespaces enable programs to avoid potential *name clashes* by associating each identifier with the namespace from which it originates. In software development, a **name clash** is when two otherwise distinct entities with the same name become part of the same scope. Name clashes can occur, for example, if two or more Python modules contain identifiers with the same name

and are imported into the same program, as shown in Figure 7-9.

FIGURE 7-9 Example Name Clash of Imported Functions

In this example, module1 and module2 are imported into the same program. Each module contains an identifier named double, which return very different results. When the function call double(num_list) is executed in main, there is a name clash. Thus, it cannot be determined which of these two functions should be called. Namespaces provide a means for resolving such problems.

In Python, each module has its own namespace. This includes the names of all items in the module, including functions and global variables—variables defined within the module and outside the scope of any of its functions. Thus, two instances of identifier double, each defined in their own module, are distinguished by being *fully qualified* with the name of the module in which each is defined: module1.double and module2.double. Figure 7-10 illustrates the use of fully qualified identifiers for calls to function double.

The use of namespaces to resolve problems associated with duplicate naming is not restricted to computer programming. In fact, it occurs in everyday situations. Imagine, for instance, that you

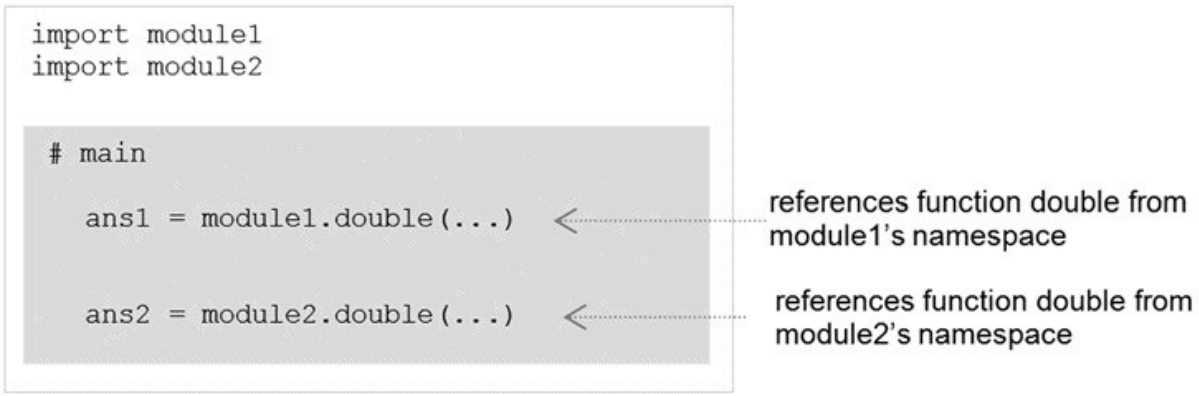


FIGURE 7-10 Example Use of Fully Qualified Function Names

run into a friend who tells you that "Paul is getting married." In fact, you have two friends in common named Paul. Because you are not certain which Paul your friend is referring to, you may respond "Paul from back home, or Paul from the dorm?" In this case, you are asking your friend to respond with a fully qualified name to resolve the ambiguity: "home:Paul" vs. "dorm:Paul." Next, we look at different ways that Python modules can be imported.

LET'S TRY IT
Enter each of the following functions in their own modules named mod1.py and mod2.py. Enter and execute the following and observe the results.

```
# mod1
def average(lst):
print('average of mod1 called')

... import mod1, mod2
... mod1.average([10, 20, 30]) ???
... mod2.average([10, 20, 30])

# mod2
def average(lst):
print('average of mod2 called')

???
... average([10, 20, 30]) ???
```

A **namespace** provides a context for a set of identifiers. Every module in Python has its own namespace. A **name clash** is when two otherwise distinct entities with the same identifier become part of the same scope.

7.3.3 Importing Modules

In Python, the **main module** of any program is the first (“top-level”) module executed. When working interactively in the Python shell, the Python interpreter functions as the main module, containing the *global namespace*. The namespace is reset every time the interpreter is started (or when selecting Shell → Restart Shell). Next we look at various means of importing modules in Python. (We note that module __builtins__ is automatically imported in Python programs, providing all the built-in constants, functions, and classes.)

In Python, the **main module** of any program is identified as the first (“top-level”) module executed.
The “import modulename” Form of Import

When using the import *modulename* form of import, the namespace of the imported module becomes *available to*, but not *part of*, the importing module. Identifiers of the imported module, therefore, must be fully qualified (prefixed with the module’s name) when accessed. Using this form of import prevents any possibility of a name clash. Thus, as we have seen, if two modules, module1 and module2, both have the same identifier, identifier1, then module1.identifier1 denotes the entity of the first module and module2.identifier1 denotes the entity of the second module.

LET’S TRY IT
Enter the following into the Python shell and observe the results.

... factorial(5) ... import math ??? ... factorial(5)

???
... math.factorial(5)
??? ... math.factorial(5)

???
With the import *modulename* form of import in Python, the namespace of the imported module becomes available to, but does not become part of, the namespace of the importing module.
The “from-import” Form of Import
Python also provides an alternate import statement of the form
from *modulename* **import** *something*
where *something* can be a list of identifiers, a single renamed identifier, or an

asterisk, as shown below,

(a) **from** *modulename* **import** func1, func2 (b)**from** *modulename* **import** func1 **as** new_func1 (c)**from** *modulename* **import** *

In example (a), only identifiers func1 and func2 are imported. In example (b), only identifier func1 is imported, renamed as new_func1 in the importing module. Finally, in example (c), *all* of the identifiers are imported, except for those that begin with two underscore characters, which are meant to be private in the module, which will be discussed later.

There is a fundamental difference between the from *modulename* import and import *modulename* forms of import in Python. When using import *modulename*, the namespace of the imported module does not become part of the namespace of the importing module, as mentioned. Therefore, identifiers of the imported module must be fully qualified (e.g., *modulename*. func1) in the importing module. In contrast, when using from-import, the imported module's namespace *becomes part of* the importing module's namespace. Thus, imported identifiers are referenced without being fully qualified (e.g., func1).

The from *modulename* import func1 as new_func1 form of import is used when identifiers in the imported module's namespace are known to be identical to identifiers of the importing module. In such cases, the renamed imported function can be used without needing to be fully qualified. Finally, using the from *modulename* import * form of import in example (c), although convenient, makes name clashes more likely. This is because the names of the imported identifiers are not explicitly listed in the import statement, creating a greater chance that the programmer will unintentionally define an identifier with the same name as in the importing module. And since the from-import form of import allows imported identifiers to be accessed without being fully qualified, it is unclear in the importing module where these identifiers come from.
We provide an example of this in Figure 7-11.

```
# module somemodule

def func1(n):
  return n * 10

def func2(n1, n2):
  return n1 * n2
```

```
# ---- main

from somemodule import *

# NAMESPACES AND FUNCTION USE

def func2(n):
    return n * n            # definition of func2 masks imported func2

print(func1(8))             # outputs 80 (func1 of somemodule called)
print(somemodule.func1(8))  # NameError: name 'somemodule' is not defined

print(func2(5))             # outputs 25 (func2 of MAIN called)
print(func2(3, 8))          # TypeError: func2() takes exactly 1 argument

print(somemodule.func2(3, 8)) # NameError: name 'somemodule' is not
                                defined
```

FIGURE 7-11 Example Use of from-import Form of Import

Module somemodule contains functions func1 and func2. Since somemodule is imported with from somemodule import *, identifiers func1 and func2 become part of the main module's namespace. However, since the module's namespace already contains identifier func2 (denoting the function defined there), access to func2 of somemodule is *masked*, and therefore is inaccessible. Using the fully qualified form somemodule.func2 does not work either, since somemodule is not part of the imported namespace for this form of import.

Finally, it is recommended Python style that standard modules be imported before the programmer-defined ones, with each section of imports separated by a blank line as shown below.
import standardmodule1 # standard modules
import standardmodule2
import somemodule1 # programmer-defined modules import somemodule2
LET'S TRY IT Enter the following into the Python shell and observe the results.

... from math import factorial ... from math import factorial as fact ... factorial(5) ... fact(5)

??? ???

... def factorial(n):
print('my factorial') ... factorial(5)
??? ??? ... def factorial(n):
print('my factorial') ... factorial(5)
... math.factorial(5) ... fact(5) ??? ???

With the from-import form of import, imported identifiers become part of the importing module's namespace. Because of the possibility of name clashes, import *modulename* is the preferred form of import in Python.

Module Private Variables

In Python, all identifiers in a module are "public"—that is, accessible by any other module that imports it. Sometimes, however, entities (variables, functions, etc.) in a module are meant to be "private"—used within the module, but not meant to be accessed from outside it.

Python does not provide any means for preventing access to variables or other entities meant to be private. Instead, there is a convention that names beginning with two underscores (_) are intended to be private. Such entities, therefore, *should not* be accessed. It does not mean that they *cannot* be accessed, however. There is one situation in which access to private variables is restricted. When the from *modulename* import * form of import is used to import *all* the identifiers of a module's namespace, names beginning with double underscores are not imported. Thus, such entities become inaccessible from within the importing module.

In Python, all the variables in a module are "public," with the convention that variables beginning with an two underscores are intended to be private.
7.3.4 Module Loading and Execution

Each imported module of a Python program needs to be located and loaded into memory. Python first searches for modules in the current directory. If the module is not found, it searches the directories specified in the PYTHONPATH environment variable. If the module is still not found (or PYTHONPATH is not defined), a Python installation-specific path is searched (e.g., C:\Python32\Lib). If the program still does not find the module, an error (ImportError exception) is reported. For our purposes, all of the modules of a program will be kept in the

same directory. However, if you wish to develop a module made available to other programs, then the module can be saved in your own Python modules directory specified in the PYTHONPATH, or stored in the particular Python installation Lib directory. When a module is loaded, a compiled version of the module with file extension .pyc is automatically produced. Then, the next time that the module is imported, the compiled .pyc file is loaded, rather than the .py file, to save the time of recompiling. A new compiled version of a module is automatically produced whenever the compiled version is out of date with the source code version of the module when loading, based on the dates that the files were created/modified.

Built-in Function **dir()**

Built-in function dir() is very useful for monitoring the items in the namespace of the main module for programs executing in the Python shell. For example, the following gives the namespace of a newly started shell,

```
... dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

The following shows the namespace after importing and defining variables,

```
... import random
... n 5 10
... dir()
['__builtins__', '__doc__', '__name__', '__package__', 'n', 'random']
```

Selecting Shell→ Restart Shell (Ctrl-F6) in the shell resets the namespace,

```
(after Restart Shell selected)
... dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

LET’S TRY IT Create the following Python module named simplemodule, import it, and call function displayGreeting as shown from the Python shell and observe the results.

```
# simplemodule
def displayGreeting():
print('Hello World!')
... import simplemodule
... simplemodule.displayGreeting()
```

Modify module simplemodule to display 'Hey there world!', import and again execute function displayGreeting as shown. Observe the results.
... import simplemodule
... simplemodule.displayGreeting()
Finally, reload the module as shown and again call function displayGreeting.

... reload(simplemodule)
... simplemodule.displayGreeting() ???

When a module is loaded, a compiled version of the module with file extension .pyc is automatically produced. When using the Python shell, an updated module can be forced to be reloaded and recompiled by use of the reload() function.

7.3.5 Local, Global, and Built-in Namespaces in Python

During a Python program's execution, there are as many as three namespaces that are referenced ("active")—the built-in namespace, the global namespace, and the local namespace. The **built-in namespace** contains the names of all the built-in functions, constants, and so on, in Python. The **global namespace** contains the identifiers of the currently executing module. And the **local namespace** is the namespace of the currently executing function (if any).

When Python looks for an identifier, it first searches the local namespace (if defined), then the global namespace, and finally the built-in namespace. Thus, if the same identifier is defined in more than one of these namespaces, it becomes masked, as depicted in Figure 7-12.

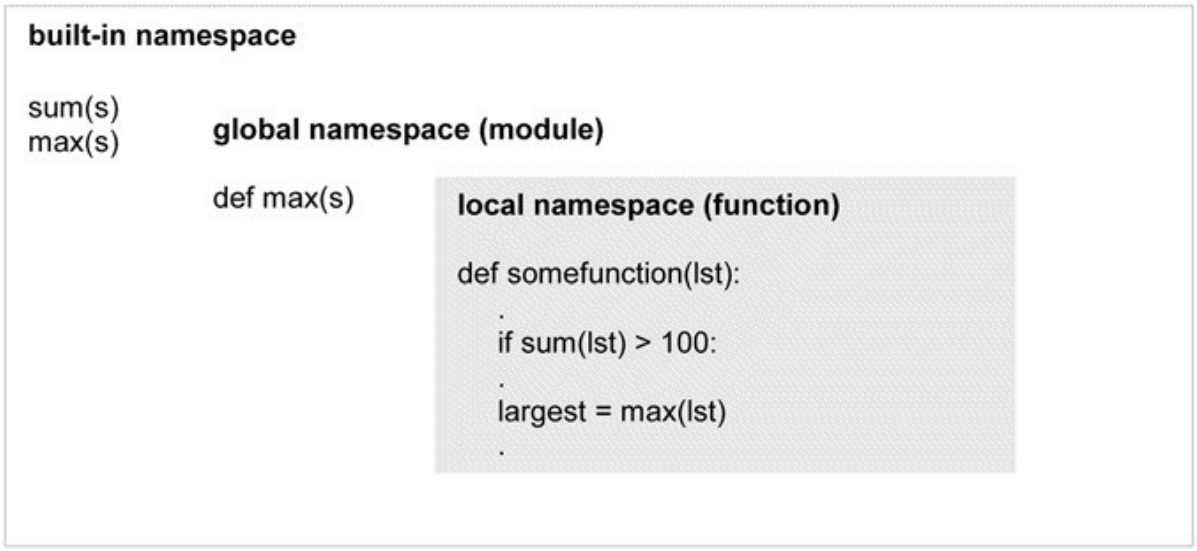


FIGURE 7-12 Local, Global, and Built-in Namespaces of Python

Functions sum and max are built-in functions in Python, and thus in the built-in namespace. Built-in function sum returns the sum of a sequence (list or tuple) or integers. Built-in function max returns the largest value of a string, list, tuple, and other types.

In the module (and thus part of the global namespace) is defined another function named max. This programmer-defined function returns the index of the largest value from an ordered collection of items, not the value itself as built-in function max is designed to do. This is demonstrated below.

max([4, 2, 7, 1, 9, 6]) → 9 (built-in function max)
max([4, 2, 7, 1, 9, 6]) → 4 (programmer-defined function max)

Which specific functions are called from within somefunction depends on where the functions are defined. Function sum, for example, is not defined within the global namespace. Therefore, built-in function sum of the built-in namespace is called. The call to function max, on the other hand, does not access the built-in function max; rather, it calls function max of the more closely defined global namespace. This demonstrates how issues of scope, if not clearly considered, can result in subtle and unexpected program errors. Consider the example program in Figure 7-13.

```
# grade_calc module

def max(grades):
    largest = 0

    for k in grades:
        if k > 100:
            largest = 100
        elif k > largest:
            largest = k

    return largest

def grades_highlow(grades):
    return (min(grades), max(grades))
```

```
# classgrades (main module)

from grade_calc import *

class_grades = [86, 72, 94, 102, 89, 76, 96]

low_grade, high_grade = grades_highlow(class_grades)
print('Highest adjusted grade on the exam was', high_grade)
print('Lowest grade on the exam was', low_grade)

print('Actual highest grade on exam was', max(class_grades))
```

FIGURE 7-13 Inadvertent Masking of Identifier in the Built-in Namespace

This program is meant to read in the exam grades of a class. (The grades are hard-coded here for the sake of an example.) The main module imports module grade_calc that contains function grades_ highlow, which returns as a tuple the highest grade (with extra credit grades over 100 returned as 100) and lowest grade in a list of grades. Upon executing this program, we find the following results,

Highest adjusted grade on the exam was 100
Lowest grade on the exam was 72
The actual highest grade on the exam was 100
...

For the list of grades 86, 72, 94, 102, 89, 76, 96, the high and low grades of 72

and 100 is correct (counting the grade 102 as a grade of 100). Then the program is to display the actual highest grade of 102. However, a grade of 100 is displayed instead.

The problem is that the grade_calc module was imported using from grade_calc import *. Thus, all of the entities of the module were imported, including the defined max function. This function returns a “truncated” maximum grade of 100 from a list of grades, used as a supporting function for function grades_highlow. However, since it is defined in the global (module) namespace of the classgrades program, that definition of function max masks the built-in max function of the built-in namespace. Thus, when called from within the classgrades program, it also produces a truncated highest grade, thus returning the actual highest grade of 100 instead of 102.

This example shows the care that must be taken in the use and naming of global identifiers, especially with the from-import * form of import. Note that if function max were named as a private member of the module, __max, then it would not have been imported into the main module and the actual highest grade displayed would have been correct.

Enter the following in the Python shell: ... sum([1, 2, 3])
???

LET’S TRY IT
Create a file with the following module:

# module max_test_module
def test_max():
print 'max 5', max([1, 2, 3]) ... def sum(n1, n2, n3): total 5 n1 1 n2 1 n3 return total
... sum([1, 2, 3])
???
... sum(1, 2, 3)
???

Create and execute the following program: import max_test_module

def max():
print('max:local namespace called') print(max_test_module.test_max())
At any given point in a Python program’s execution, there are three possible

namespaces referenced (“active”)—the **built-in namespace**, the **global namespace**, and the **local namespace**.

7.3.6 A Programmer-Defined Stack Module

In order to demonstrate the development of a programmer-defined module, we present an example stack module. A **stack** is a very useful mechanism in computer science. Stacks are used to temporarily store and retrieve data. They have the property that the last item placed on the stack is the first to be retrieved. This is referred to as LIFO—“last in, first out.” A stack can be viewed as a list that can be accessed only at one end, as depicted in Figure 7-14.

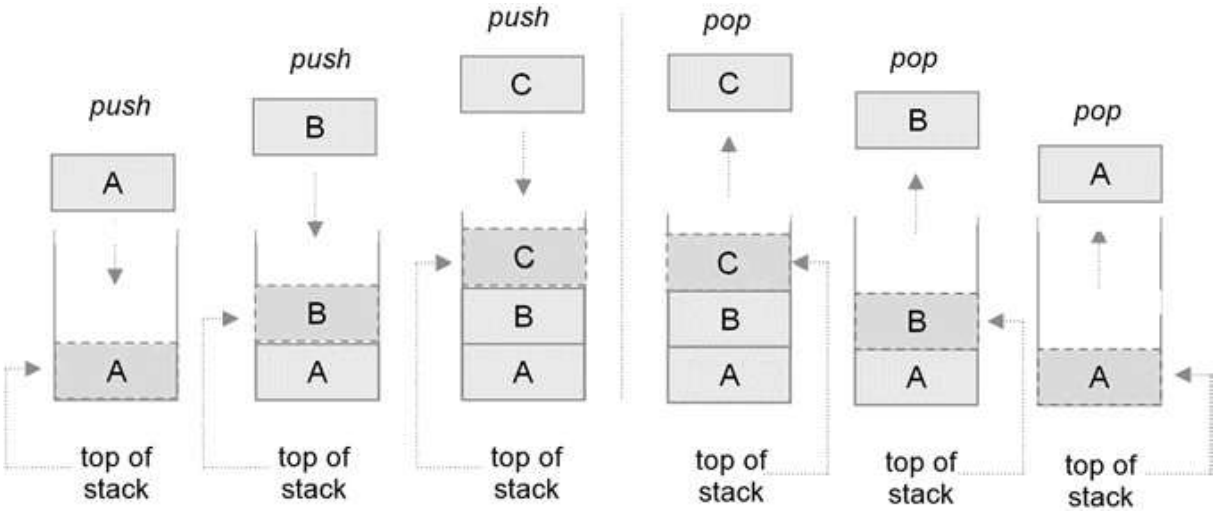


FIGURE 7-14 Stack Mechanism

In this example, three items are *pushed* on the stack, denoted by A, B, and C. First, item A is pushed, followed by item B, and then item C. After the three items have been placed on the stack, the only item that can be accessed or removed is item C, located at the *top of stack*. When C is retrieved, it is said to be *popped* from the stack, leaving item B as the top of stack. Once item B is popped, item A becomes the top of stack. Finally, when item A is popped, the stack becomes empty. It is an error to attempt to pop an empty stack.

In Figure 7-15 is a Python module containing a set of functions that implements this stack behavior. For demonstration purposes, the program displays and pushes the values 1 through 4 on the stack. It then displays the numbers popped off the stack, retrieved in the reverse order that they were pushed.

The stack module consists of five functions—getStack, isEmpty, top, push, and

pop. The stack is implemented as a list. Only the last element in the list is accessed—that is where

```
1  # stack Module
2
3  def getStack():
4
5      """Creates and returns an empty stack."""
6
7      return []
8
9  def isEmpty(s):
10
11     """Returns True if stack empty, otherwise returns False. """
12
13     if s == []:
14         return True
15     else:
16         return False
17
18 def top(s):
19
20     """Returns value of the top item of stack, if stack not empty.
21        Otherwise, returns None.
22     """
23
24     if isEmpty(s):
25         return None
26     else:
27         return s[len(s) - 1]
28
29 def push(s, item):
30
31     """Pushes item on the top of stack. """
32
33     s.append(item)
34
35 def pop(s):
36
37     """Returns top of stack if stack not empty. Otherwise, returns None."""
38
39     if isEmpty(s):
40         return None
41     else:
42         item = s[len(s) - 1]
43         del s[len(s) - 1]
44         return item
```

FIGURE 7-15 Programmer-Defined Stack Module

all items are “pushed” and “popped” from. Thus, the end of the list logically functions as the “top” of stack.

Function getStack**(lines 3–7)** creates and returns a new empty stack as an empty list. Function isEmpty **(lines 9–16)** returns whether a stack is empty or not (by checking if an empty list). Function top (**lines 18–27**) returns the top item of a stack without removing it. Functions push **(lines 29–33)** and pop **(**

lines 35–44) provide the essential stack operations. The push function pushes an item on the stack by appending it to the end of the list. The pop function removes the item from the top of stack by retrieving the last element of the list, and then deleting it. If either pop or top are called on an empty stack, the special value None is returned.

```
   # main
1  import stack
2
3  mystack = stack.getStack()
4
5  for item in range(1, 5):
6      stack.push(mystack, item)
7      print('Pushing', item, 'on stack')
8
9  while not stack.isEmpty(mystack):
10     item = stack.pop(mystack)
11     print('Popping', item,'from stack')
```

FIGURE 7-16 Demonstration of the Programmer-Defined Stack Module

The small program in Figure 7-16 demonstrates the use of the stack module. First, a new stack is created by assigning variable mystack to the result of the call to function getStack **(line 3)**. Even though a list is returned, it is intended to be more specifically a stack type. Therefore, only the stack-related functions should be used with this variable—the list should not be directly accessed, otherwise the stack may become corrupted. Then the values 1 through 4 are pushed on the stack, and popped in the reverse order,

Pushing 1 on stack Pushing 2 on stack Pushing 3 on stack Pushing 4 on stack Popping 4 from stack Popping 3 from stack Popping 2 from stack Popping 1 from stack

Ensuring that only the provided functions can be used on the stack represents a fundamental advantage of objects. Since an object consists of data and methods (routines), only the methods of the object can be used to access and alter the data. Thus, the stack type is best implemented as an object, as all other values in Python are. We will see how do to this after the introduction of object-oriented programming in Chapter 10.

7.3.7 Let’s Apply It—A Palindrome Checker Program

The program in Figure 7-18 determines if a given string is a palindrome. A

palindrome is something that reads the same forwards and backwards. For example, the words “level” and “radar” are palindromes. The program imports the stack module developed in the previous section. This program utilizes the following programming feature:

Programmer-defined module ➤
Example execution of the program is given in Figure 7-17.

```
Program execution ...


This program can determine if a given string is a palindrome

(Enter return to exit)
Enter string to check: look
look is NOT a palindrome

Enter string to check: radar
radar is a palindrome

Enter string to check:
>>>
```

FIGURE 7-17 Execution of the Palindrome Checker Program

The stack module is imported on **line 3** of the program. The “import *modulename*” form of import is used. Therefore, each stack function is referenced by stack.*function_name*. **Lines 6–7** displays a simple program welcome. The following lines perform the required initialization for the program. **Line 10**sets char_stack to a new empty stack by call to getStack(). **Line 11** initializes variable empty_string to the empty string. This is used in the program for determining if the user has finished entering all the words to check.

The string to check is input by the user on **line 14**. If the string is of length one, then by definition the string is a palindrome. This special case is handled in **lines 17–18**. Otherwise, the complete string is checked. First, variable palindrome is initialized to True. On **line 24**, variable compare_ length is set to half the length of the input string, using integer division to truncate the length to an equal number of characters. This represents the number of characters from the front of the string (working forward) that must match the number of characters on the rear of the string (working backwards). If there are an odd number of characters, then the middle character has no other character to match against.

On **lines 27–28** the second half of the string chars are pushed character-by-character onto the stack. Then, on lines **31–37** the characters are popped from the stack one by one, returning in the reverse order that they were pushed. Thus, the first character popped (the *last* character pushed on the stack) is compared to the *fi r s t* character of the complete string. This continues until there are no more characters to be checked. If characters are found that do not match, then is_palindrome is set to False (**lines 34–35**) and the while loop terminates. Otherwise, is_palindrome remains True. **Lines 40–43** output whether the input string is a palindrome or not, based on the final value of is_palindrome. **Lines 45–46** prompt the user for another string to enter, and control returns to the top of the while loop.

```
# Palindrome Checker Program

import stack

# welcome
print('This program can determine if a given string is a palindrome\n')
print('(Enter return to exit)')

# init
char_stack = stack.getStack()
empty_string = ''

# get string from user
chars = input('Enter string to check: ')

while chars != empty_string:
    if len(chars) == 1:
        print('A one letter word is by definition a palindrome\n')
    else:
        # init
        is_palindrome = True

        # to handle strings of odd length
        compare_length = len(chars) // 2

        # push second half of input string on stack
        for k in range(compare_length, len(chars)):
            stack.push(char_stack, chars[k])

        # pop chars and compare to first half of string
        k = 0
        while k < compare_length and is_palindrome:
            ch = stack.pop(char_stack)
            if chars[k].lower() != ch.lower():
                is_palindrome = False

            k = k + 1

        # display results
        if is_palindrome:
            print(chars, 'is a palindrome\n')
        else:
            print(chars, 'is NOT a palindrome\n')

    # get next string from user
    chars = input('Enter string to check: ')
```

FIGURE 7-18 Palindrome Checker Program

It is important to mention that the problem of palindrome checking could be done more efficiently without the use of a stack. A for loop can be used that compares the characters k locations from each end of the given string. Thus, our use of a stack for this problem was for demonstration purposes only. We leave the checking of palindromes by iteration as a chapter exercise.

Self-Test Questions

1. Any initialization code in a Python module is only executed once, the first time that the module is loaded. (TRUE/FALSE)
2. With the "import *moduleName*" form of import, any utilized entities from the imported module must be prefixed with the module name. (TRUE/FALSE)
3. By convention, variables names in a module beginning with two _____ characters are meant to be treated as private variables of the module.
4. When importing modules, all Python Standard Library modules must be imported before any programmer-defined modules, otherwise a runtime error will occur. (TRUE/FALSE)

5. If a particular module is imported more than once in a Python program, the Python interpreter will ensure that the module is only loaded and executed the first time that it is imported. (TRUE/FALSE)

6. The _____ command can be used to force the reloading of a given module, useful for when working interactively in the Python shell.
7. The three active namespaces that may exist during the execution of any given Python program are the _____, _____ and _____ namespaces.

ANSWERS: 1. True, 2. True, 3. Underscore, 4. False, 5. True, 6. reload, 7. built-in, global, local

COMPUTATIONAL PROBLEM SOLVING

7.4 Calendar Year Program (function version)

In this section, we implement and test the modular design for the calendar year program in section 7.2.

7.4.1 The Problem

The problem is the same as that given in Chapter 4—to display a calendar year from 1800 to 2099, displayed as shown in Figure 7-19.

7.4.2 Problem Analysis

Since the same problem is being solved, the algorithms utilized are also the same—an algorithm for computing the day of the week (for the years 1800 to 2099), an algorithm for storing the calendar year, and a means of displaying a calendar year three months across.

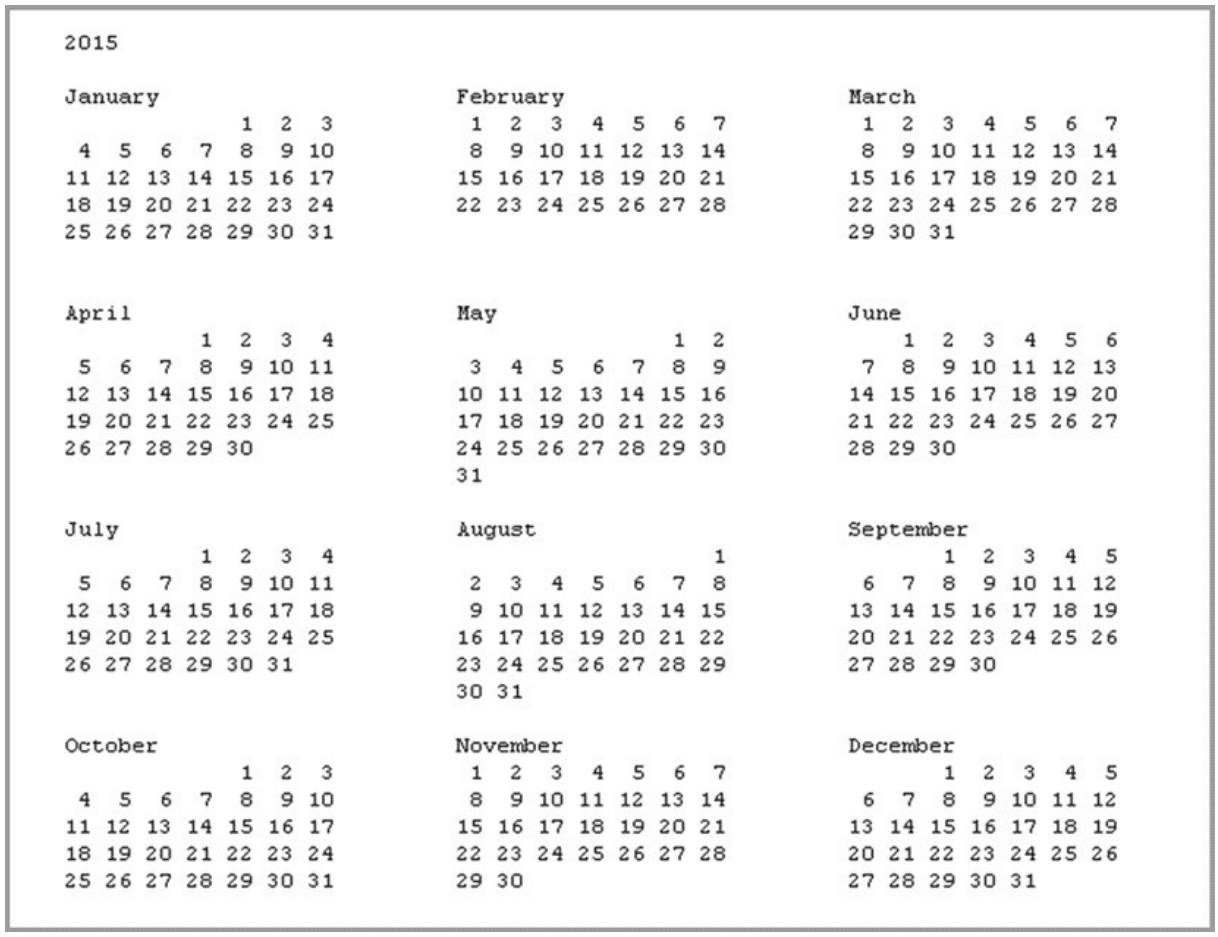
7.4.3 Program Design

The program requirements, data structures, algorithms, and overall program steps are the same as specified in Chapter 4. This program will only differ in that we will utilized a modular design for the program, as introduced in section 7.2.

This will result in a program that is more easily understood, more easily modified, and more amenable to program testing.

7.4.4 Program Implementation and Testing

Now that we are using modular design, we can more easily incrementally test the program. Given the specification of each module (function), we can first test each function separately to see if the implementation correctly meets the specification. The individual testing of modules is called **unit testing**. Once each module is individually tested, they can all be tested together.



```
2015

January                  February                 March
          1  2  3          1  2  3  4  5  6  7     1  2  3  4  5  6  7
 4  5  6  7  8  9 10       8  9 10 11 12 13 14     8  9 10 11 12 13 14
11 12 13 14 15 16 17      15 16 17 18 19 20 21    15 16 17 18 19 20 21
18 19 20 21 22 23 24      22 23 24 25 26 27 28    22 23 24 25 26 27 28
25 26 27 28 29 30 31                              29 30 31

April                    May                      June
       1  2  3  4                   1  2             1  2  3  4  5  6
 5  6  7  8  9 10 11       3  4  5  6  7  8  9     7  8  9 10 11 12 13
12 13 14 15 16 17 18      10 11 12 13 14 15 16    14 15 16 17 18 19 20
19 20 21 22 23 24 25      17 18 19 20 21 22 23    21 22 23 24 25 26 27
26 27 28 29 30            24 25 26 27 28 29 30    28 29 30
                          31

July                     August                   September
       1  2  3  4                         1             1  2  3  4  5
 5  6  7  8  9 10 11       2  3  4  5  6  7  8     6  7  8  9 10 11 12
12 13 14 15 16 17 18       9 10 11 12 13 14 15    13 14 15 16 17 18 19
19 20 21 22 23 24 25      16 17 18 19 20 21 22    20 21 22 23 24 25 26
26 27 28 29 30 31         23 24 25 26 27 28 29    27 28 29 30
                          30 31

October                  November                 December
          1  2  3          1  2  3  4  5  6  7           1  2  3  4  5
 4  5  6  7  8  9 10       8  9 10 11 12 13 14     6  7  8  9 10 11 12
11 12 13 14 15 16 17      15 16 17 18 19 20 21    13 14 15 16 17 18 19
18 19 20 21 22 23 24      22 23 24 25 26 27 28    20 21 22 23 24 25 26
25 26 27 28 29 30 31      29 30                   27 28 29 30 31
```

FIGURE 7-19 Calendar Year Display

This is called **integration testing**. We demonstrate unit testing and integration testing of the calendar program in this section.

Implementation of the Calendar Year Program

Figure 7-20 shows an implementation of the Calendar Year program based on the modular design developed.

The main section of the program, **lines 188–207** , follows directly from our modular design, developed in section 7.2. The first function call is to function getYear (**lines 3–12**). This function simply contains an input statement and while loop to ensure that the year returned is in the range 1800 to 2099, inclusive.

Ne xt, function constructCalYear (**lines 121–147**) is called. As seen from the modular design, it relies on calls to functions leapYear, dayOfWeekJan1, numDaysInMonth, and constructCalMonth. This function essentially consists of a for loop that iterates once for each of the twelve months, calling constructCalMonth to construct each month and append each to the list calendar_year. The constructed year is a list of lists as given below,

[year, [month, week1, week2,...], [month, week1, week2,...],...]

```
# Calendar Year Program (Function Version)

def getYear():

    """Returns a year between 1800-2099, inclusive, or the value -1."""

    year = int(input('Enter year (yyyy) (-1 to quit): '))
    while (year < 1800 or year > 2099) and year != -1:
        print('INVALID INPUT - Year must be between 1800 and 2099')
        year = int(input('Enter year (1-12): '))

    return year

def leapYear(year):

    """Returns True if year a leap year, otherwise returns False."""

    if (year % 4 == 0) and (not (year % 100 == 0) or
        (year % 400 == 0)):
        leap_year = True
    else:
        leap_year = False

    return leap_year

def dayOfWeekJan1(year, leap_year):

    """Returns the day of the week for January 1 of a given year.

        year must be between 1800 and 2099. leap_year must be True if
        year a leap year, and False otherwise.
    """
    century_digits = year // 100
    year_digits = year % 100
    value = year_digits + (year_digits // 4)

    if century_digits == 18:
        value = value + 2
    elif century_digits == 20:
        value = value + 6

    # adjust for leap years
    if not leap_year:
        value = value + 1

    # return first day of month for Jan 1
    return (value + 1) % 7

```

FIGURE 7-20 Implementation of the Calendar Year Program (*Continued*)
Functions dayOfWeekJan1 (**lines 26–47**) and numDaysInMonth (**lines 49–63**) are straightforward, neither relying on a call to any other function.

Function constructCalMonth is the largest of all the functions. It uses the same basic approach as before to properly align the weeks for display. In this function, a list of strings is created to be output to the screen. Finally, function

displayCalendar is given a particular calendar year structure and displays it in four rows of months, with three months across each row.

```
def numDaysInMonth(month_num, leap_year):
#-----------------------------------------------------------------------
    """Returns the number of days in a given month.

       month_num in the range 1-12, inclusive.
       leap_year True if month in a leap year, otherwise False.
    """
    num_days_in_month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

    # special check for February in leap year
    if (month_num == 2) and leap_year:
        num_days = 29
    else:
        num_days = num_days_in_month[month_num - 1]
    return num_days

def constructCalMonth(month_num, first_day, num_days_in_month):
#-----------------------------------------------------------------------
    """Returns a formatted calendar month for display on the screen.

       month_num in the range 1-12, inclusive.
       first_day in the range 0-6 (1-Sun, 2-Mon, ..., 0-Sat)

       Returns a list of strings of the form,
       [month_name, week1, week2, week3, week4, ...]
    """
    # init
    empty_str = ''
    blank_col = format(' ', '3')
    blank_week = format(' ', '21')
    month_names = ('January', 'February', 'March', 'April', 'May',
                   'June', 'July', 'August', 'September', 'October',
                   'November', 'December')

    calendar_month = [' ' + format(month_names[month_num - 1], '<20')]
    current_day = 1
    current_col = 1
    calendar_week = ''

    # init starting column
    if first_day == 0:
        starting_col = 7
    else:
        starting_col = first_day

    # add any needed leading spaces for first week of month
    while current_col < starting_col:
        calendar_week = calendar_week + blank_col
        current_col = current_col + 1

```

```
    # construct month for proper number of days
    while current_day <= num_days_in_month:

        # store day of month in field of length 3
        calendar_week = calendar_week + format(str(current_day), '>3')

        # append new week to month if at end of week
        if current_col == 7:
            calendar_month = calendar_month + [calendar_week]
            calendar_week = empty_str
            current_col = 1
        else:
            current_col = current_col + 1

        current_day = current_day + 1

    # if there is a final week, append to constructed month
    if calendar_week != empty_str:
        calendar_month = calendar_month + [calendar_week]

    return calendar_month

def constructCalYear(year):

    """Returns a formatted calendar year for display on the screen.

       year in the range 1800-2099, inclusive
       Returns a list beginning with the year, followed by
       twelve constructed months

          [year, month1, month2, week3, ..., month12]
    """
    # init
    leap_year = leapYear(year)
    first_day_of_month = dayOfWeekJan1(year, leap_year)
    calendar_year = [year]

    # construct calendar from twelve constructed months
    for month_num in range(1, 13):
        num_days_in_month = numDaysInMonth(month_num, leap_year)

        calendar_year = calendar_year + \
                        constructCalMonth(month_num, first_day_of_month,
                                          num_days_in_month)

        first_day_of_month = (first_day_of_month + \
                              num_days_in_month) % 7

    return calendar_year

```

```
def displayCalendar(calendar_year):

    """Displays a calendar_year on the screen three months across. """

    # init
    month_separator = format(' ', '8')
    blank_week = format(' ', '21')

    # display year
    print('\n', calendar_year[0])

    # display months three across
    for month_index in [1, 4, 7, 10]:

        # init
        week = 1
        lines_to_print = True

        while lines_to_print:

            # init
            lines_to_print = False

            # print weeks of months side-by-side
            for k in range(month_index, month_index + 3):
                if week < len(calendar_year[k]):
                    print(calendar_year[k][week-1], end='')
                    lines_to_print = True
                else:
                    print(blank_week, end='')

                print(month_separator, end='')

            # move to next screen line
            print()

            # increment week
            week = week + 1

# --- main

# initialization
terminate = False

# program greeting
print('This program will display a calendar year for a given year')

```

FIGURE 7-20 Implementation of the Calendar Year Program (*Continued*)

Development and Unit Testing of Individual Modules

To begin, we can create a program that contains each of the function headers and docstring specifications, and implement and test them one by one. This is given in Figure 7-21, with an implementation for function getYear.

Running this version of the program will not produce any output. However, it

will define function getYear. Therefore, this function can be unit tested in the Python shell,
... getYear()
Enter year (yyyy) (21 to quit): 1800
1800

```
196 # continue to display calendar years until -1 entered
197 while not terminate:
198     year = getYear()
199 
200     if year == -1:
201         terminate = True
202     else:
203         # construct calendar
204         calendar_year = constructCalYear(year)
205 
206         # display calendar
207         displayCalendar(calendar_year)
```

FIGURE 7-20 Implementation of the Calendar Year Program

```
# Calendar Year Program
# This program will display any calendar year between 1800-2099.

def getYear():

    """Returns a year between 1800-2099, inclusive, or -1 if end of
       user input
    """

    year = int(input('Enter year (yyyy) (-1 to quit): '))

    while (year < 1800 or year > 2099) and year != -1:
        print('INVALID INPUT - Year must be between 1800 and 2099')
        year = int(input('Enter year: '))

    return year


#def leapYear(year):

#    """Returns True if year a leap year, otherwise returns False. """
.
.
```

FIGURE 7-21 Implementation and Unit Testing of Function getyear

... getYear()
Enter year (yyyy) (21 to quit): 2099
2099
... getYear()

Enter year (yyyy) (21 to quit): 1985
1985
... getYear()
Enter year (yyyy) (21 to quit): 1799
INVALID INPUT – Year must be between 1800 and 2099 Enter year: 2100
INVALID INPUT – Year must be between 1800 and 2099 Enter year: 2050
2050
...

Since the getYear function tested OK, we can next implement and test the leapYear function. This is given in Figure 7-22.

```
# Calendar Year Program
# This program will display any calendar year between 1800-2099.

##def getYear():

##    """Returns a year between 1800-2099, inclusive, or -1 if end of
##       user input.
##    """
##    year = int(input('Enter year (yyyy) (-1 to quit): '))
##
##    while (year < 1800 or year > 2099) and year != -1:
##        print('INVALID INPUT - Year must be between 1800 and 2099')
##        year = int(input('Enter year: '))
##
##    return year

def leapYear(year):

    """Returns True if year a leap year, otherwise returns False."""

    if (year % 4 == 0) and (not (year % 100 == 0) or \
        (year % 400 == 0)):
        leap_year = True
    else:
        leap_year = False

    return leap_year
.
.
```

FIGURE 7-22 Implementation and Unit Testing of Function leapYear
Running this version of the program defines function leapYear, which can then be unit tested.

... leapYear(1803)
False
... leapYear(1800)

```
False
... leapYear(1860)
True
... leapYear(1900)
False
... leapYear(1964)
True
... leapYear(2000)
True
... leapYear(2012)
True
```

Based on the test results, both getYear and leapYear are properly working. Next, we will unit test function dayOfWeekJan1. The implementation of this function is given in Figure 7-23.

```
.
.

##def leapYear(year):

##    """Returns True if year a leap year, otherwise returns False. """
##
##    if (year % 4 == 0) and (not (year % 100 == 0) or
##        (year % 400 == 0)):
##        leap_year = True
##    else:
##        leap_year = False
##
##    return leap_year

def dayOfWeekJan1(year, leap_year):

    """Returns the day of the week for January 1 of a given year.

        year must be between 1800 and 2099. leap_year must be True if
        year a leap year, and False otherwise.
    """
    century_digits = year // 100
    year_digits = year % 100
    value = year_digits + (year_digits // 4)

    if century_digits == 18:
        value = value + 2
    elif century_digits == 20:
        value = value + 6

    # adjust for leap years
    if not leap_year:
        value = value + 1

    # return first day of month for Jan 1
    return (value + 1) % 7

.
.
```

FIGURE 7-23 Implementation and Unit Testing of Function dayOfWeekJan1

As with previous stages of development, running this version of the program defines function leapYear, which we can then unit test.

```
... dayOfWeekJan1(1800, False)
4
... dayOfWeekJan1(1864, True)
6
... dayOfWeekJan1(1900, False)
```

2
... dayOfWeekJan1(2000, True)
0
... dayOfWeekJan1(2012, True)
1
... dayOfWeekJan1(2013, False)
3

Recalling that the values 0–6 represent the days of the week Saturday to Sunday (where Saturday is 0), checking these results with other sources, they are found to be correct. Next, we unit test function numDaysInMonth as shown in Figure 7-24.

```
.
.

##    # adjust for leap years
##    if not leap_year:
##        value = value + 1
##
##    # return first day of month for Jan 1
##    return (value + 1) % 7


def numDaysInMonth(month_num, leap_year):

    """Returns the number of days in a given month.

        month_num in the range 1-12, inclusive.
        leap_year True if month in a leap year, otherwise False.
    """
    num_days_in_month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

    # special check for February in leap year
    if (month_num == 2) and leap_year:
        num_days = 29
    else:
        num_days = num_days_in_month[month_num]

    return num_days
```

FIGURE 7-24 Implementation and Unit Testing of Function numDaysInMonth
By executing this version of the program and unit testing function numDaysInMonth, we get the following results.

... numDaysInMonth(1, False)
28

```
... numDaysInMonth(2, False)
31
... numDaysInMonth(2, True)
29
... numDaysInMonth(3, False)
30
... numDaysInMonth(12,False)
Traceback (most recent call last):

File “ pyshell#5 .”, line 1, in ,module .
numDaysInMonth(12,False)
File “C:\My Python Programs\CalendarYearFunc.py”, line 65, in
numDaysInMonth
num_days num_days_in_month[month_num]
IndexError: tuple index out of range
...
```

Obviously, something is wrong with this function. First, we have to make sure that we called it with proper values. The function specification indicates that it should be given values in the range 1 to 12, inclusive, as the first argument, and a Boolean value (indicating whether the year is a leap year or not) for the second. Therefore, it is being called correctly.

Looking at the results produced for the consecutive months tested, it seems that the results are off by one month. That is, numDaysInMonth(1, False) is giving the result 28, which would be correct for February (of a non-leap year), and numDaysInMonth(2, False) is giving the result 31, which would be the correct results for March (of either a leap year or a non-leap year). So, if the index value used to index list num_days_in_month were one greater than it should be, this would be consistent with the test results. Furthermore, the fact that an index out of range error occurs for a month number of 12 further supports the likely “off by one” error.

However, there is one result that is inconsistent with this hypotheses. For February of a leap year, numDaysInMonth(2, True), we get the correct result of 29. This could be explained if the case of February in a leap year is handled as a special case in the function code. In fact, that is exactly the case by the if-else statement. Therefore, we make the needed correction in the function, given in Figure 7-25.

```
def numDaysInMonth(month_num, leap_year):

    """Returns the number of days in a given month.

        month_num in the range 1-12, inclusive.
        leap_year True if month in a leap year, otherwise False.
    """
    num_days_in_month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

    # special check for February in leap year
    if (month_num == 2) and leap_year:
        num_days = 29
    else:
        num_days = num_days_in_month[month_num - 1]

    return num_days
```

FIGURE 7-25 Corrected numDaysInMonth Function

We again test the function, and this time get the correct result for each test case.

... numDaysInMonth(1, False)
31
... numDaysInMonth(2, False)
28
... numDaysInMonth(2, True)
29
... numDaysInMonth(3, False)
31
... numDaysInMonth(4, False)
30
etc.

Since there are only 13 possible sets of input values to the function (including two sets for February), we test each one and find that the function is working for each of these cases.

Function constructCalMonth is tested next. It is passed a month number (1 to 12), a day of the week value (0 to 6) and the number of days in the month, and returns a constructed calendar month as given below,

... constructCalMonth(1, 0, 31)
[' January ', ' 1', ' 2 3 4 5 6 7 8', ' 9 10 11 12 13 14 15', ' 16 17 18 19 20 21 22', ' 23 24 25 26 27 28 29', ' 30 31']
...

Additional testing of this function indicates that it is working correctly.

Integration Testing of Modules

The remaining functions to test are constructCalYear and displayCalendar. Function constructCalYear relies on the use of functions leapYear, dayOfWeekJan1, numDaysInMonth, and constructCalMonth. Since each of these has been tested and found correct, we can now perform integration testing of all these functions with function constructCalYear. We therefore execute a version of the program in which all the code developed so far, except these functions, are commented out, as depicted in Figure 7-26.

```
##
##    # if there is a final week, append to constructed month
##    if calendar_week != empty_str:
##        calendar_month = calendar_month + [calendar_week]
##
##    return calendar_month

def constructCalYear(year):

    """Returns a formatted calendar year for display on the screen.

        year in the range 1800-2099, inclusive
        Returns a list beginning with the year, followed by
        twelve constructed months

          [year, month1, month2, week3, ..., month12]
    """
    # init
    leap_year = leapYear(year)
    first_day_of_month = dayOfWeekJan1(year, leap_year)
    calendar_year = [year]

    # construct calendar from twelve constructed months
    for month_num in range(1, 13):
        num_days_in_month = numDaysInMonth(month_num, leap_year)

        calendar_year = calendar_year + \
                        constructCalMonth(month_num, first_day_of_month,
                                          num_days_in_month)

        first_day_of_month = (first_day_of_month + num_days_in_month) % 7

    return calendar_year
```

FIGURE 7-26 Integration Testing with Function constructCalYear

Test results from a call to ConstructCalYear are given below.

... constructCalYear(2012)

[2012, [' January ', ' 1 2 3 4 5 6 7', ' 8 9 10 11 12 13 14 ', ' 15 16 17 18 19 20 21', ' 22 23 24 25 26 27 28', ' 29 30 31'], [' February

', ' 1 2 3 4', ' 5 6 7 8 9 10 11', ' 12 13 14 15 16
17 18', ' 19 20 21 22 23 24 25', ' 26 27 28 29'], [' March ', ' 1 2 3', ' 4 5 6 7 8 9
10', ' 11 12 13 14 15 16 17', ' 18 19 2
0 21 22 23 24', ' 25 26 27 28 29 30 31', [' April ', ' 1 2 3 4 5 6 7', ' 8 9 10 11 12 13
14', ' 15 16 17 18 19 20 21', ' 22 23 24 25 26
27 28', ' 29 30'], [' May ', ' 1 2 3 4 5', ' 6 7 8
9 10 11 12', ' 13 14 15 16 17 18 19', ' 20 21 22 23 24 25 26', ' 27 28 29 30 3
1'], [' June ', ' 1 2', ' 3 4 5 6 7 8 9', ' 10 11 12 13 14 15 16', ' 17 18 19 20 21 22 23',
' 24 25 26 27 28 29 30'], [' July ', ' 1 2 3 4 5 6 7', ' 8 9 10 11 12 13 14', ' 15 16
17 18 19 20 21', ' 22 23 24 25 26 27 28', ' 29 30 31'], [' August ', ' 1 2 3 4', ' 5 6 7
8 9 10 11', ' 12 13 14 15 16 17 18', ' 19 20 21 22 23 24 25', ' 26 27 28 29 30 31'],
[' September ', ' 1', ' 2 3 4 5 6 7 8', ' 9 10 11 12 13 14 15', ' 16 17 18 19
20 21 22', ' 23 24 25 26 27 28 29', ' 30'], [' October ', ' 1 2 3 4 5 6', ' 7 8 9 10 11
12 13', ' 14 15 16 17 18 19 20', ' 21 22 23 24
25 26 27', ' 28 29 30 31'], [' November ', ' 1 2 3', ' 4 5 6 7 8 9 10', ' 11 12 13 14 15
16 17', ' 18 19 20 21 22 23 24', ' 25 2
6 27 28 29 30',] [' December ', ' 1', ' 2 3 4 5 6 7 8', ' 9 10 11 12 13 14 15', ' 16 17
18 19 20 21 22', ' 23 24 25 26 27
28 29', ' 30 31']]
...

The constructed year looks correct, and so we include the final function to be tested, displayCalendar, in this integration testing. Test results from a call to displayCalendar are given below.

This program will display a calendar year for a given year Enter year (yyyy) (21 to quit): 2012
2012
J
a 1 8
n
u
a 2 9
r
y 1

3 0
1
4 1
1
5 2
.
.
.

This output does not look even close to what it should be. The output displayed by function displayCalendar depends on providing it with a correctly structured calendar year. So maybe we should think through each step of function displayCalendar with the test output from function constructCalYear to see if we can gain any insight into the problem. The output from function constructCalYear for the year 2012 is partially reproduced below.

... constructCalYear(2012)
[2012, ' January ', ' 1 2 3 4 5 6 7', ' 8 9 10 11 12 13 14 ', ' 15 16 17 18 19 20 21', ' 22 23 24 25 26 27 28', ' 29 30 31'], [' February

', ' 1 2 3 4', ' 5 6 7 8 9 10 11', ' 12 13 14 15 16 17 18', ' 19 20 21 22 23 24 25', ' 26 27 28 29'], [' March ', '
The body of function constructCalYear, for parameter year, is shown below.

```
1 # init
2 leap_year 5 leapYear(year)
3 first_day_of_month 5 dayOfWeekJan1(year, leap_year)
4 calendar_year 5 [year]
5
6 # construct calendar from twelve constructed months
7 for month_num in range(1, 13):
8 num_days_in_month 5 numDaysInMonth(month_num, leap_year)
9
10 calendar_year 5 calendar_year 1 \
11 constructCalMonth(month_num, first_day_of_month,
12 num_days_in_month)
13
14 first_day_of_month 5 (first_day_of_month 1 num_days_in_month) % 7
15
```

16 return calendar_year

Lines 2 and **3** make calls to functions leapYear and dayOfWeekJan1, which have both been tested, so we can assume, for now, that the results of these function calls are correct. On **line 4**, list calendar_year is initialized to a list of one element, the integer representing the year. The for loop at **line 7** is used to construct each of the calendar months one at a time, appending each to list calendar_year.

One thing to double-check is the range of the for loop, since there is always a chance that we have an “off by one” error. In this case, range is called with a first argument of 1, and a second argument of 13. Thus, variable month_num will be assigned to the values 1,2,...,12. The loop will iterate, therefore, 12 times—once for each of the 12 months. This is the correct number of iterations. Now we should check to see if they are the correct range of values.

V ariable month_num is only used in the call to function constructCalMonth on **line 11**. Thus, we check the specification of the function to see if it requests month values in the range 1–12, or if it requests their index values 0–11. Looking back at the specification for this function in Figure 7-8, 1–12 is the correct range of month values to be passed to it.

Next, we consider the concatenation of calendar_year with a newly constructed calendar month each time through the loop. This occurs in **lines 10–12**, given below.
calendar_year 5 calendar_year 1 \
constructCalMonth(month_num,first_day_of_month, num_days_in_month) We know that calendar_year is initialized to the list [2012]. Also, calls to constructCalMonth return a constructed calendar month as a list of the form,
[' January ', ' 1', ' 2 3 4 5 6 7 8', ' 9 10 11 12 13 14 15', ' 16 17 18 19 20 21 22', ' 23 24 25 26 27 28 29', ' 30 31']
Thus, the above instruction would result in the following,

calendar_year 5 [2012] 1 [' January ', ' 1', ' 2 3 4 5 6 7 8', ' 9 10 11 12 13 14 15', ' 16 17 18 19 20 21 22', ' 23 24 25 26 27 28 29', ' 30 31']

Now we realize something. The structure of a calendar year is designed to be,
[year, [month, week1, week2,...], [month, week1, week2,...],....]
So we check the initial result of the structure of calendar_year after doing a

simplified version of the above concatenation in the Python shell,
... [2012] 1 [' January ', ' 1'] [2012, ' January ', ' 1']

Here is a problem! The result of the concatenations should be a list of lists, with each sublist containing a constructed month. The result above is a *single* list of values. That would certainly be a cause for the displayCalYear function not working correctly. The error is that instead of concatenating the two lists, each constructed month should be *appended* to the calendar year list, as given below.

```
calendar_year.append(constructCalMonth(
month_num,
first_day_of_month,
num_days_in_month))
```

Testing again in the Python shell, this gives the following (correct) results.
... [2012] 1 [[' January ', ' 1']] [2012, [' January ', ' 1']]

Therefore, we make the correction and retest function displayCalYear. This time we get a correctly displayed calendar year, as shown in Figure 7-27. (Note that the width of the shell window needs to be adjusted for the weeks of the calendar year to properly align.)

```
This program will display a calendar for a given year
Enter year (yyyy) (-1 to quit): 2012

 2012
 January                    February                   March
  1  2  3  4  5  6  7                1  2  3  4                  1  2  3
  8  9 10 11 12 13 14         5  6  7  8  9 10 11         4  5  6  7  8  9 10
 15 16 17 18 19 20 21        12 13 14 15 16 17 18        11 12 13 14 15 16 17
 22 23 24 25 26 27 28        19 20 21 22 23 24 25        18 19 20 21 22 23 24

 April                      May                        June
  1  2  3  4  5  6  7              1  2  3  4  5                  1  2
  8  9 10 11 12 13 14         6  7  8  9 10 11 12         3  4  5  6  7  8  9
 15 16 17 18 19 20 21        13 14 15 16 17 18 19        10 11 12 13 14 15 16
 22 23 24 25 26 27 28        20 21 22 23 24 25 26        17 18 19 20 21 22 23

 July                       August                     September
  1  2  3  4  5  6  7              1  2  3  4                        1
  8  9 10 11 12 13 14         5  6  7  8  9 10 11         2  3  4  5  6  7  8
 15 16 17 18 19 20 21        12 13 14 15 16 17 18         9 10 11 12 13 14 15
 22 23 24 25 26 27 28        19 20 21 22 23 24 25        16 17 18 19 20 21 22
                                                         23 24 25 26 27 28 29

 October                    November                   December
     1  2  3  4  5  6                1  2  3                        1
  7  8  9 10 11 12 13         4  5  6  7  8  9 10         2  3  4  5  6  7  8
 14 15 16 17 18 19 20        11 12 13 14 15 16 17         9 10 11 12 13 14 15
 21 22 23 24 25 26 27        18 19 20 21 22 23 24        16 17 18 19 20 21 22
                                                         23 24 25 26 27 28 29

Enter year (yyyy) (-1 to quit):
```

FIGURE 7-27 Calendar Year Program Output

CHAPTER SUMMARY General Topics Python-Specific Programming Topics

Modules/Module Interface/Specification Top-Down Design
Stacks
Namespaces and Name Clashes
Global Scope/Global Variables
Python Docstring Specification
Built-in/Global/Local Namespaces in Python Main Modules/Standard Modules in Python Forms of Module Import in Python
Global vs. Private Module Variables in Python Module Loading and Execution in Python

CHAPTER EXERCISES Section 7.1

1. For the following function, def hours_of_daylight(month, year)

(a) Give an appropriate docstring specification where hours_of_daylight returns the total number of hours of daylight for the month and year given (each passed an integer value) designed so that the function does not check for invalid parameter values.

(b) Give a print statement that displays the docstring for this function.
Section 7.2

2. Develop a modular design depicting the components of a typical computer, similar to that for the Space Shuttle in Figure 7-2. Assume that the major components of a computer system consist of a CPU (central processing unit), busses (connections between components), main memory (RAM), secondary memory (hard drive, USB drive, etc.), and input/output devices (mouse, keyboard, etc.). Search online for more specific subsystems and devices to complete your design.

3. For the hours_of_daylight function in exercise 1, give a code segment that prompts the user for a month and year, and appropriately calls function hours_of_daylight according to its docstring specification, displaying the result.

Section 7.3
4. For module1, module2, and the client module shown below, indicate which of the imported identifiers would result in a name clash if the imported identifiers were not fully qualified.

```
# module1

def func_1(n):
    etc.

def func_2(n):
    etc.
```

```
# module2

def func_2(n):
    etc.

def func_3(n):
    etc.
```

```
from module1 import *
from module2 import *

def func_3(n):
    etc.
```

5. Depict what is left on stack s after the following series of push and pop operations (starting with the first column of operations and continuing with the second). Assume that the stack is initially empty.

push(s,10) push(s,50)
push(s,20) push(s,60)
push(s,40) push(s,80)
pop(s) pop(s)
pop(s) pop(s)

6. For the program in Figure 7-9 that imports modules module1 and module1, indicate how many total namespaces exist for this program.
7. For the Palindrome Checker program in section 7.3.7, describe the changes that would be needed in the program if the import statement were changed from import Stack to from Stack import *.
8. For the following program and the imported modules, describe any name clashes that would occur for both program version1 and version 2.

```
# module m1

def total(items):

def convert(items):

def show(items)
```

```
# module m2

def totalSum(items):

def convert(items):

def display(items)
```

```
from m1 import *
from m2 import *

def display()

def calc()

def getItems()

# ---- main

items = getItems()
items = convert(items)
show(items)
```

Version 1

```
import m1
import m2

def display()

def calc()

def getItems()

# ---- main

items = getItems()
items = m2.convert(items)
display(items)
```

Version 2

PYTHON PROGRAMMING EXERCISES

P1. Write a function called convertStatus that is passed status code 'f', 's', 'j', or 'r' and returns the string 'freshman', 'sophomore', 'junior', or 'senior', respectively. Design your function so that if an inappropriate letter is passed, an error value is returned. Make sure to include an appropriate docstring with your function.

P2. Write a function called palindromeChecker using iteration to return True if a provided string is a palindrome, and False otherwise. Make sure to include

docstring specification for the function.
P3. Implement a set of functions called getData, extractValues, and calcRatios. Function getData should prompt the user to enter pairs of integers, two per line, with each pair read as a single string, for example,

Enter integer pair (hit Enter to quit):
134 289 (read as '134 289')
etc.

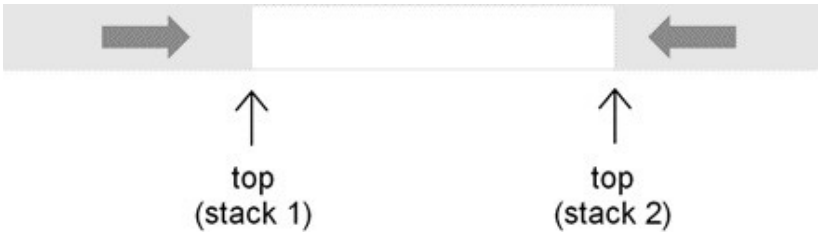
These strings should be passed one at a time as they are being read to function extractValue, which is designed to return the string as a tuple of two integer values,
extractValues('134 289') returns (134, 289) etc.
Finally, each of these tuples is passed to function calcRatios one at a time to calculate and return the ratio of the two values. For example,
calcRatios((134, 289)) returns 0.46366782006920415 etc.
Implement a complete program that displays a list of ratios for an entered series of integer value pairs. Make sure to include docstring specification for each of the functions.
Program Development Problems 287
PROGRAM MODIFICATION PROBLEMS

M1. Stack Module: Limited Stack Size
For the stack module given in Figure 7-15, suppose that the implementation is to limit stacks to no more than 100 items. Redesign and re-implement the relevant parts of the module that need to be changed. Write a small program to demonstrate this new version of the stack module.

M2. Stack Module: Ability to “Peek” in Stack
For the stack module given in Figure 7-15, redesign and re-implement the relevant parts of the module to allow the ability to “peek” into the stack to find out if a given element is on the stack or not. Write a small program to demonstrate this new version of the stack module.

M3. Stack Module: Double-Ended Stacks
A double-ended stack is essentially a pair of stacks that share a fixed amount of storage (memory). The two stacks are designed so that the top of stack of each begins at each end of a list structure. Each top of stack moves towards the other when an element is pushed, as depicted below. This stack implementation has

the advantage of more effectively utilizing a fixed amount of memory than if each stack were allocated its own storage.



Modify and test the stack module to behave as a double-ended stack.
M4. Calendar Year Program: Optional Month/Year Display
Modify the Calendar Year Program so the user can select whether they want to display a complete calendar year, or just a specific calendar month.
M5. Calendar Year Program: Flexible Layout of Months
Modify the Calendar Year Program so the user can select whether they want the calendar displayed “row
oriented” (with January, February, and March in the first row, April, May, and June in the second row,
etc.) or “column oriented” (with January, February, March, and April in the first column, May, June, July,
and August in the second column, etc.).

PROGRAM DEVELOPMENT PROBLEMS

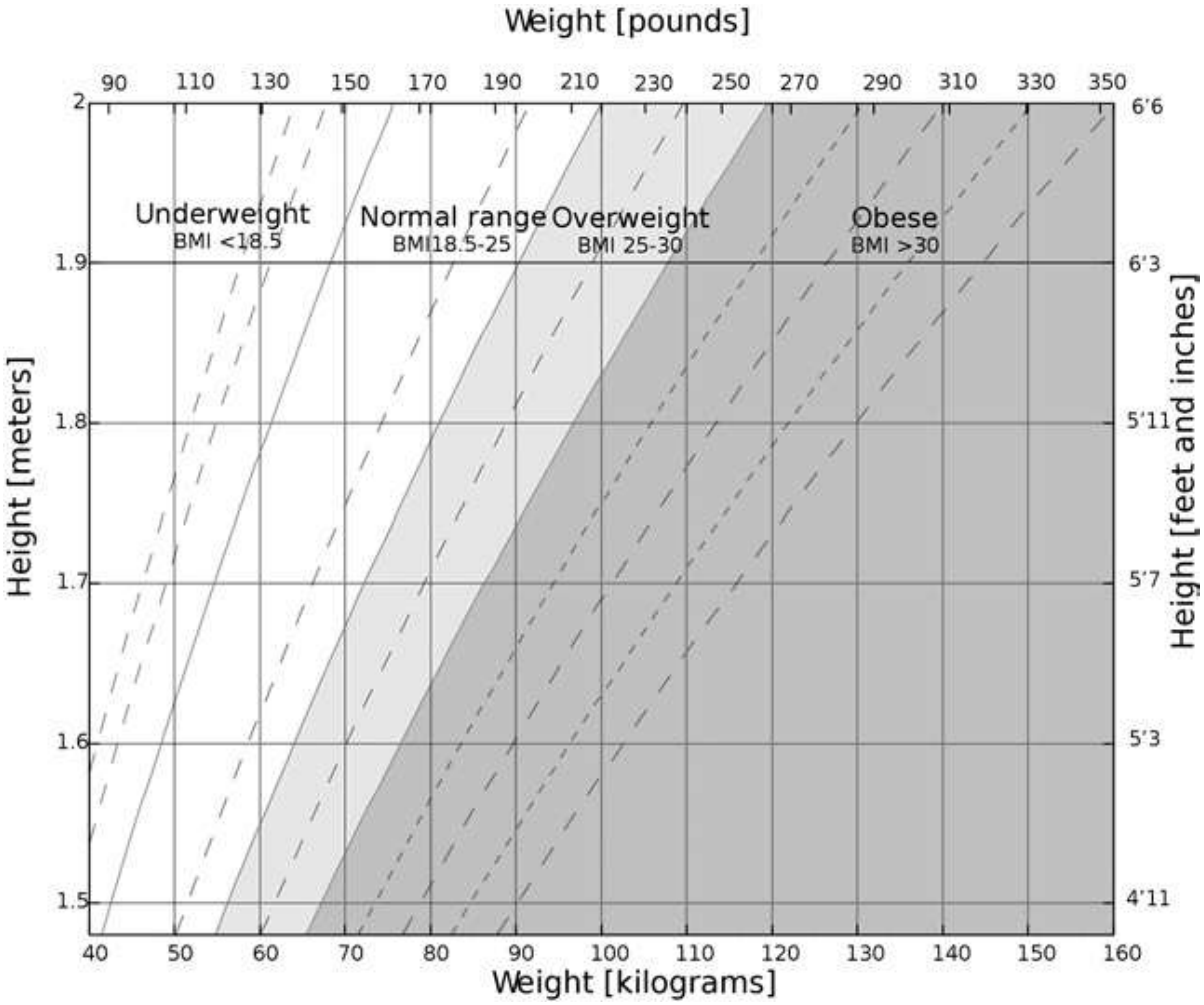
D1. Parentheses Matching Program
Implement and test a Python program that determines if all parentheses in an entered line of code form matching pairs. Note: Pairs of parentheses may be nested.

D2. Determining Body Mass Index (BMI)
Following is a chart for determining one’s body mass index (BMI). BMI is a general indication of the amount of body fat that a person has. The formula for computing BMI is,

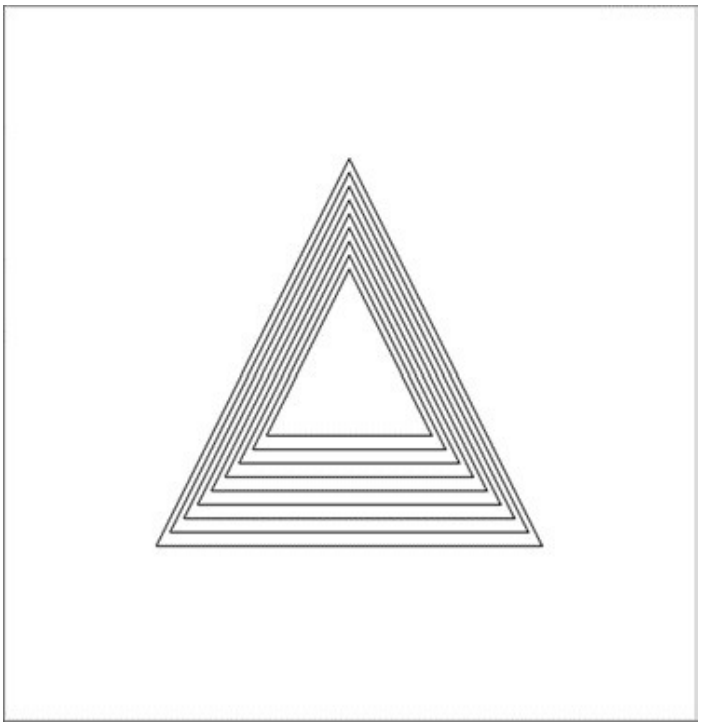
BMI 5 mass / height^2

Implement a Python program that prompts a user for their height and weight. Height should be entered as inches and weight should be entered in pounds. Perform the calculation in units of kilograms and meters as shown in the chart.

Compare the result to the information in the chart. Use functions in your program design.



D3. Create interesting geometric animations by the use of turtle graphics and the stack module provided in the chapter. A snapshot of an example is shown below.



To do this, create a geometric shape in turtle graphics. Then, create multiple invisible turtle objects of this shape, each scaled down to a smaller size than the previous and pushed on a stack. Once the stack has been "loaded" with a sufficient number of such turtle objects, continually pop the stack and make each visible on the screen, pushing the popped turtle object onto a second stack. When the first stack is empty, reverse the process using the second stack (pushing the popped turtle objects from the second stack back onto the first stack), making each visible again. Use the sleep function of the time module of Python to control the speed of the animation.

Text Files CHAPTER 8

We have, up to now, been storing data only in the variables and data structures of programs. However, such data is not available once a program terminates. Therefore, in order for information to persist from one program execution to the next, the data must be stored in a data file. In this chapter we discuss the use of one particular type of data file, text files.

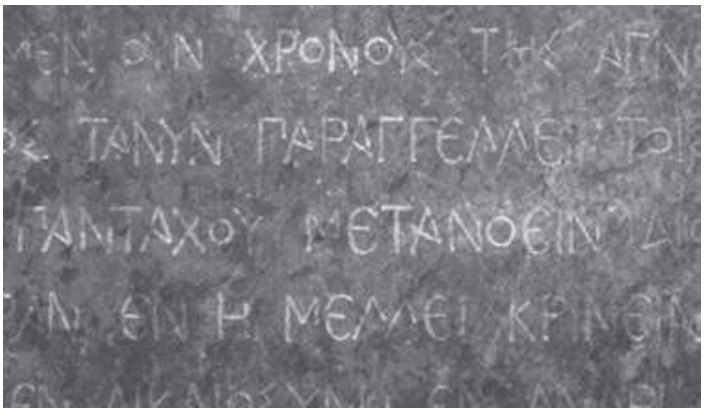
OBJECTIVES
After reading this chapter and completing the exercises, you will be able to: ♦ Explain what a text file is

♦ Differentiate between a text file and a binary file
♦ Explain the process of opening and closing a file
♦ Explain the process of reading and writing files
♦ Explain the process of exception handling
♦ Explain the process of unit testing using test drivers ♦ Effectively utilize text files in Python
♦ Effectively perform string processing in Python
♦ Catch and handle exceptions in Python

CHAPTER CONTENTS

MOTIVATION

The vast amount of information being generated today means that there must also be vast amounts of storage for all this data. Information sharing on the web through Wikipedia, social networks, and other means keeps driving this growth. *It has been estimated that 90% of all the data in the world has been generated in the last two years.*



Throughout the age of computing, data has always been shared in some manner. In the early days data had to be physically carried on storage media such as reels of magnetic tapes. Later, early developments in computer networks allowed the sharing of data between a small number of large and expensive connected computers. With the advent of the Internet, computers are not directly connected, but indirectly through routers that temporarily store and forward data toward its destination. The World Wide Web has made access to information easy and intuitive by the incorporation of hypertext— text that can be clicked on to retrieve more text—effectively making text "three-dimensional."

The evolving technology of "cloud computing" is a further step in the sharing of information. Not only is data easily shared, but also the programs and other services needed to use that data. Figure 8-1 shows the various information storage devices in use today. (Note that a volatile memory device is one that loses its contents when power is lost, whereas a nonvolatile device retains its contents.)

Storage Technology	Example	Characteristics
Magnetic Storage	Hard drive	Nonvolatile
	Magnetic Tape Storage	Nonvolatile
Semiconductor Memory	Main memory	Volatile
	USB (Thumb) Drive	Nonvolatile
Optical Storage	CD, DVD	Nonvolatile

FIGURE 8-1 Types of Storage Technology

8.1 What Is a Text File?

A **text file** is a file containing characters, structured as individual lines of text. In addition to printable characters, text files also contain the *nonprinting* newline character, \n, to denote the end of each text line. As discussed in Chapter 2, the newline character causes the screen cursor to move to the beginning of the next screen line. Thus, text files can be directly viewed and created using a text editor.

In contrast, **binary files** can contain various types of data, such as numerical values, and are therefore not structured as lines of text. Such files can only be read and written via a computer program. Any attempt to directly view a binary file will result in "garbled" characters on the screen. Our purpose is not to cover all of types of files in Python. Rather, we cover enough to be able to perform simple reading and writing of text files.

LET'S TRY IT Let's view both a text file and a binary file using a simple text editor like notepad. First, create a simple file within IDLE named hello.py containing only two lines :

```
print 'Hello'
print 'There'
```

Execute the program. From the shell window that the program displays the results in, enter the following, ... import hello

This will both execute the program and compile it into a binary file named hello.pyc. Open the Python source file using notepad (or other simple text editor). The two print statements of the program should be displayed. Open the Python compiled file of this program using notepad and observe what is displayed.

A **text file** is a file containing characters, structured as lines of text. A **binary file** is a file that is formatted in a way that only a computer program can read.

8.2 Using Text Files

Fundamental operations of all types of files include *opening* a file, *reading* from a file, *writing* to a file, and *closing* a file. Next we discuss each of these operations when using text files in Python.

8.2.1 Opening Text Files

All files must first be opened before they can be read from or written to. In

Python, when a file is (successfully) opened, a file object is created that provides methods for accessing the file. We look how to open files for either reading (from) or writing (to) a file in this section.

All files must first be opened before they can be used. In Python, when a file is opened, a file object is created that provides methods for accessing the file.
Opening for Reading
To open a file for reading, the built-in open function is used as shown,
input_file 5 open('myfile.txt','r')

The first argument is the file name to be opened, 'myfile.txt'. The second argument, 'r', indicates that the file is to be opened for reading. (The second argument is optional when opening a file for reading.) If the file is successfully opened, a file object is created and assigned to the provided identifier, in this case identifier input_file.

When opening a file for reading, there are a few reasons why an *I/O error* may occur. (We look at how to include the ability of a program to catch and handle such errors in the discussion of exception handling in section 8.4.) First, if the file name does not exist, then the program will terminate with a “no such file or directory” error,

... open('testfile.txt','r')
Traceback (most recent call last):

File " pyshell#1 .", line 1, in ,module .
open('testfile.txt','r')
IOError: [Errno 2] No such file or directory: 'testfile.txt'

This error can also occur if the file name is not found in the location looked for (uppercase and lowercase letters are treated the same for file names). When a file is opened, it is first searched for in the same folder/directory that the program resides in. The programs in the text are written this way. However, an alternate location can be specified in the call to open by providing a path to the file,

input_file 5 open('data/myfile.txt','r')

In this case, the file is searched for in a subdirectory called data of the directory in which the program is contained. Thus, its location is relative to the program location. (Although some operating systems use forward slashes, and other

backward slashes in path names, directory paths in Python are always written with forward slashes and are automatically converted to backward slashes when required by the operating system executing on.) Absolute paths can also be provided giving the location of a file anywhere in the file system,

input_file 5 open('C:/mypythonfiles/data/myfile.txt','r') When the program has finished reading the file, it should be closed by calling the close method on the file object,
input_file.close() Once closed, the file may be reopened (with reading starting at the beginning of the file) by the same, or another program. Next, we look at how to open files for writing in Python.
LET'S TRY IT In a new Python file window, enter the following lines,

Line one
Line two
Line three

Save the file under the name data.txt. (Make sure to save it with extension '.txt' and not '.py'.) Then, in the same folder (directory) as the data file, create the following Python program, file_name 5 input('Enter file name: ')
input_file 5 open(file_name, 'r')
Run this program twice. The first time, enter the file name garbage.txt. The second time, enter the correct file name data.txt and observe the results.
To open a file for reading in Python, the built-in function open is called with (optional) argument value 'r'.
Opening for Writing
To open a file for writing, the open function is used as shown below,
output_file 5 open('mynewfile.txt','w')

Note that, in this case, 'w' is used to indicate that the file is to be opened for writing. If the file already exists, it will be overwritten (starting with the first line of the file). When using a second argument of 'a', the output will be appended to an existing file instead.

It is important to close a file that is written to, otherwise the tail end of the file may not be written to the file (discussed below),
output_file.close()

When opening files for writing, there is not much chance of an I/O error

occurring. The provided file name does not need to exist since it is being created (or overwritten). Thus, the only error that may occur is if the file system (such as the hard disk) is full.

LET'S TRY IT In a new Python file window, enter the following lines,

file_name 5 input('Enter file name: ')
file 5 open(file_name, 'w')
file.close()

Save the file under the name createfile.py and run it. When the program requests a file name, give it any file name you wish with the extension '.txt'. Then, look in the folder in which the program resides to see if a new file with the file name that you entered exists. (This file will be empty—we will see how to write to a file next.)

Modify the open instruction above, changing 'w' to 'r' and rerun the program. When it requests a file name, enter the name of the file you just created. Run it a second time; this time give it the wrong file name. Observe the different results in each case.

To open a file for writing in Python, the built-in function open is called with a second argument of 'w'. A second argument of 'a' will open a file for appending to instead.

8.2.2 Reading Text Files

The readline method returns as a string the next line of a text file, including the end-of-line character, \n. When the end-of-file is reached, it returns an empty string as demonstrated in the while loop of Figure 8-2.

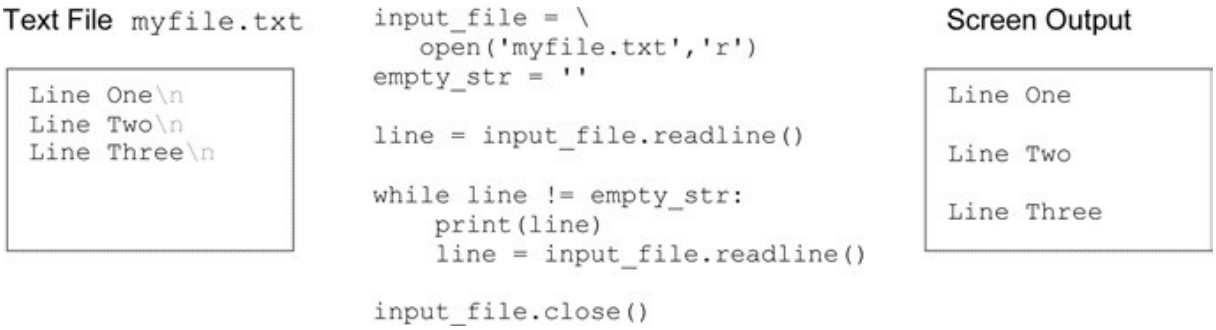


FIGURE 8-2 Reading from a Text File

It is also possible to read the lines of a file by use of the for statement,

input_file 5 \
open('myfile.txt','r')
for line in input_file:

Using a for statement, *all* lines of the file will be read one by one. Using a while loop, however, lines can be read until a given value is found, for example.

Finally, note the blank lines in the screen output. Since read_line returns the newline character, and print adds a newline character, *two* newline characters are output for each line displayed, resulting in a skipped line after each. We will see an easy way to correct this when we discuss string methods.

LET'S TRY IT Create a text file named 'myfile.txt'. Enter and execute the following program and observe the results. Make sure that the program and text file reside in the same folder.
input_file 5 open('myfile.txt','r')
for line in input_file:
print(line)
The readline method returns the next line of a text file, including the end-of-line character. If at the end of the file, an empty string is returned.
8.2.3 Writing Text Files
The write method is used to write strings to a file, as demonstrated in Figure 8-3.

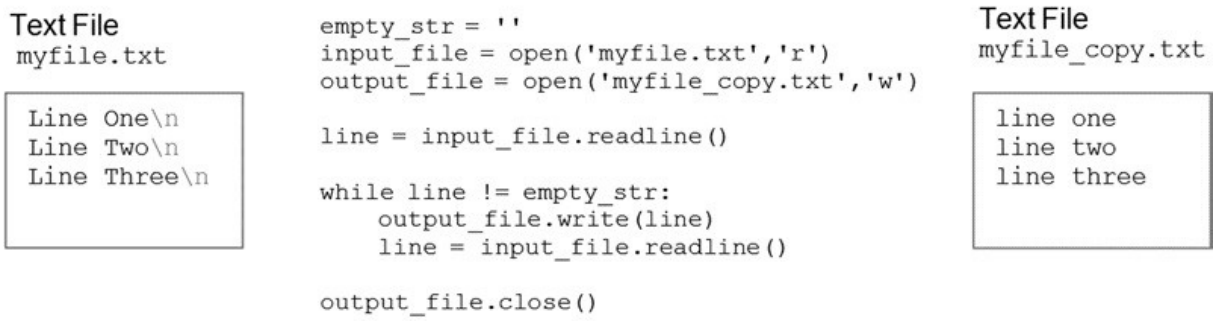


FIGURE 8-3 Writing to a Text File

This code copies the contents of the input file, 'myfile.txt', line by line to the output file, 'myfile_ copy.txt'. In contrast to print when writing to the screen, *the write method does not add a newline character to the output string*. Thus, a newline character will be output only if it is part of the string being written. In this case, each line read contains a newline character.

Finally, when writing to a file, data is first placed in an area of memory called a *buffer*. Only when the buffer becomes full is the data actually written to the file.

(This makes reading and writing files more efficient.) Since the last lines written may not completely fill the buffer, the last buffer's worth of data may not be written. The close() method *flushes* the buffer to force the buffer to be written to the file.

LET'S TRY IT In the Python shell, open an existing file myfile.txt and do the following.

... input_file 5 ('myfile.txt','r')
... output_file 5 ('newfile.txt','w')
... line 5 input_file.readline()
... output_file.write(line)
... output_file.close()

Observe that newfile.txt has been created, and examine its contents.
Use the write()method to output text to a file. To ensure that all data has been written, call the close() method to close the file after all information has been written.

Self-T est Questions
1. Only files that are written to need to be opened first. (TRUE/FALSE)

2. Indicate which of the following reasons an IOError (exception) may occur when opening a file. **(a)** Misspelled file name **(c)** File not found in directory searched **(b)** Unmatched uppercase and lowercase letters

3. Which one of the following is true?

(a) When calling the built-in open function, a second argument of 'r' or 'w' must always be given
(b) When calling the built-in open function, a second argument of 'r' must always be given when opening a file for reading
(c) When calling the built-in open function, a second argument of 'w' must always be given when opening a file for writing

4. Which one of the following is true?
(a) There is more chance of an I/O error when opening a file for reading.
(b) There is more chance of an I/O error when opening a file for writing.

5. The readline method reads every character from a text file up to and including

the next newline character '\n'. (TRUE/FALSE)
6. It is especially important to close a file that is open for writing. (TRUE/FALSE)
ANSWERS: 1. False, 2. (a),(c), 3. (c), 4. (a), 5. True, 6. True

8.3 String Processing

The information in a text file, as with all information, is most likely going to be searched, analyzed, and/or updated. Collectively, the operations performed on strings is called *string processing*. We have already seen some operations on strings—for example, str[k], for accessing individual characters, and len(str)for getting the length of a string. In this section, we revisit sequence operations that apply to strings, and look at additional stringspecific methods.

String processing refers to the operations performed on strings that allow them to be accessed, analyzed, and updated.

8.3.1 String Traversal

We saw in Chapter 4 how any sequence can be traversed, including strings. This is usually done by the use of a for loop. For example, if we want to read a line of a text file and determine the number of blank characters it contains, we could do the following,

space 5 ' '
num_spaces 5 0

line 5 input_file.readline()
for k in range(0,len(line)): if line[k] 55 space:

num_spaces 5 num_spaces 1 1
We also saw that the last lines can be done more simply without the explicit use of an index variable,
for chr in line:
if chr 55 space:
num_spaces 5 num_spaces 1 1 Given the ability to traverse a string, each character can be individually “looked at” for various types of string processing. We look at some string processing operations next.
The characters in a string can be easily traversed, without the use of an explicit index variable, using the for *chr* in *string* form of the for statement.

8.3.2 String-Applicable Sequence Operations

Because strings (unlike lists) are immutable, sequence-modifying operations are not applicable to strings. For example, one cannot add, delete, or replace characters of a string. Therefore, *all string operations that "modify" a string return a new string that is a modified version of the original string*. Sequence operations relevant to string processing are given in Figure 8-4. We look at stringspecific operations in the next section.

Sequences Operations Applicable to Strings			
Length	`len(str)`	Membership	`'h' in s`
Select	`s[index_val]`	Concatenation	`s + w`
Slice	`s[start:end]`	Minimum Value	`min(s)`
Count	`s.count(char)`	Maximum Value	`max(s)`
Index	`s.index(char)`	Comparison	`s == w`

FIGURE 8-4 Sequence Operations on Strings

Recall that as we saw with lists, the slice operator s[start:end] returns the substring starting with index start, *up to but not including* index end. Also, s.index(chr) returns the index of the first occurrence of chr in s. Finally, min and max as applied to strings return the smallest (largest) character based on the underlying Unicode encoding. Thus, for example, all lowercase letters are larger (have a larger Unicode value) than all uppercase letters. We give examples of each of these operations for s 5'Hello Goodbye!'.

... len(s) s.count('o') ... s 1 '!!' 14 3 'Hello Goodbye!!!' ... s[6] ... s.index('b') ... min(s) 'G' 10 ' '
... s[6:10] ... 'a' in s ... max(s) 'Good' False 'y'
We next look at methods in Python that are specific to strings.
Because strings are immutable, sequence modifying operations do not modify the string applied to. Rather, they construct *new* strings that are a modified version of the original.
8.3.3 String Methods
There are a number of methods specific to strings in addition to the general sequence operations. We discuss these methods next.
Checking the Contents of a String

There are times when the individual characters in a string (or substring) needs to be checked. For example, to check whether a character is an appropriate denotation of a musical note, we could do the following,

if char not in ('A', 'B', 'C', 'D', 'E', 'F', 'G'): print('Invalid musical note found')

Checking the Contents of a String			
`str.isalpha()`	Returns True if *str* contains only letters.	`s = 'Hello'`	`s.isalpha()` → `True`
		`s = 'Hello!'`	`s.isalpha()` → `False`
`str.isdigit()`	Returns True if *str* contains only digits.	`s = '124'`	`s.isdigit()` → `True`
		`s = '124A'`	`s.isdigit()` → `False`
`str.islower()` `str.isupper()`	Returns True if *str* contains only lower (upper) case letters.	`s = 'hello'`	`s.islower()` → `True`
		`s = 'Hello'`	`s.isupper()` → `False`
`str.lower()` `str.upper()`	Return lower (upper) case version of *str*.	`s = 'Hello!'`	`s.lower()` → `'hello!'`
		`s = 'hello!'`	`s.upper()` → `'HELLO!'`
Searching the Contents of a String			
`str.find(w)`	Returns the index of the first occurrence of `w` in *str*. Returns -1 if not found.	`s = 'Hello!'`	`s.find('l')` → `2`
		`s = 'Goodbye'`	`s.find('l')` → `-1`
Replacing the Contents of a String			
`str.replace(w,t)`	Returns a copy of *str* with all occurrences of `w` replaced with t.	`s = 'Hello!'`	`s.replace('H', 'J')` → `'Jello'`
		`s = 'Hello'`	`s.replace('ll', 'r')` → `'Hero'`
Removing the Contents of a String			
`str.strip(w)`	Returns a copy of *str* with all leading and trailing characters that appear in w removed.	`s = ' Hello! '` `s = 'Hello\n'`	`s.strip(' !')` → `'Hello'` `s.strip('\n')` → `'Hello'`
Splitting a String			
`str.split(w)`	Returns a list containing all strings in *str* delimited by w.	`s = 'Lu, Chao'`	`s.split(',')` → `['Lu', 'Chao']`

FIGURE 8-5 String Methods in Python

Since the in operator can also be applied to strings, we can also do the following,
if char not in 'ABCDEFG':
print('Invalid musical note character found') We could take a similar approach for determining if a given character is a lowercase or uppercase letter or digit character; for example,

char in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' or \ letter? char in 'abcdefghijklmnopqrstuvwxyz' or \
char in '0123456789'

Since checking for uppercase/lowercase and digit characters is common in programming, Python provides string methods isalpha, isdigit, isupper, and islower (among others). For example, to perform error checking on an entered credit card number could be done as follows,

if not credit_card.isdigit(): print('Invalid card number')

The method isdigit returns True if and only if each character in string credit_card is a digit. If only *part* of a string is to be checked, then a method can be applied to a slice of a string. For example, if the part numbers of a given company all begin with three letters, a check for invalid part numbers could be done as follows,

if not part_num[0:3].isalpha():
print('Invalid part number')

in which isalpha returns True if and only if the first three characters in part_num are letters. Additional string methods are listed in Figure 8-5. We look at the issue of string search and string modification next.

Python provides a number of methods specific to strings, in addition to the general sequence operations.
Searching and Modifying Strings
String processing involves search. For example, to determine the user name and domain parts of an email address, the ampersand character separating the two would be searched for,

... email_addr 5 'jsmith@somecollege.edu'
... amper_index 5 email_addr.find('@')
... username 5 email_addr[0:amper_index]

... domain 5 email_addr[amper_index 1 1:len(email_addr)] ... print('Username:', username, 'Domain:', domain) Username: jsmith Domain: somecollege.edu

The find method returns the index location of the *fi r s t occurrence* of a specified substring. Since in Python strings are immutable, to update the email address, a *new* string would be constructed with the desired replacement as shown,

... email_addr 5 username 1 '@' 1 'newcollege.edu' ... email_addr
jsmith@newcollege.edu

The replace method produces a new string with *every occurrence* of a given substring within the original string replaced with another,

... word 5 'common' ... word 5 'common'
... word.replace('m', 't') ... word 5 word.replace('m', 't') 'cotton' 'cotton'
... word ... word
'common' 'cotton'

Note that for all string modifications, the variable references the same string until it is reassigned. Python also provides a strip method that “strips off” leading and trailing characters from a
string. This is especially useful for stripping off the newline character, \n, from the end of a line in
text processing if needed,

... line 5 'Hello\n' ... line 5 'Hello\n' ... print(line) ... print(line.strip('\n')) 'Hello'
'Hello'

...
...
The find, replace, and strip methods in Python can be used to search and produce modified strings.

8.3.4 Let’s Apply It—Sparse Text Program

Sparse data is data that lacks “density.” A diary containing entries only for holidays and special occasions would contain sparse data. Sparse data can often be compressed. Compressed data should retain all (or most) of the original information. To explore this, the program in Figure 8-7 removes all occurrences of the letter ‘e’ from a provided text file. How much of the compressed text can

be understood indicates how much of the information is retained. This program utilizes the following programming features:

➤ text files ➤ string methods (replace, strip)
Figure 8-6 shows the file produced by the Sparse Text Program for a passage from *Alice's Adventures in Wonderland.*

The main section of the program begins on **line 32**. The program welcome is displayed on **lines 33–34**. In **lines 37–38**, the file name entered by the user is opened for reading and assigned to file object input_file. In **lines 37–40**, a file is opened for writing with the same file name as the input file but with 'e_' added to the beginning and assigned to the file object output_file.

The creation of the modified file is handled by function createModifiedFile (**lines 3–28**) called on **line 44**. The function returns a tuple containing how many occurrences of letter 'e' were removed, and the total character count of the file. The values of the tuple are therefore assigned to variables num_total_char and num_removals. (Note that Python allows such a multiple assignment on lists, tuples, and even on strings.) The input and output files are closed on **lines 47–48**. Finally, both the number of characters removed and the percentage of the file that the removed characters comprised are displayed (**lines 52–54**).

What remains is function createModifiedFile. On **lines 9–11**, variable empty_str is initialized to the empty string, and variables num_total_chars and num_ removals are initialized to 0. On **line 16**, variable orig_line_length is set to the length of the currently read line (in variable line) minus one. This is so that the newline character (\n) at the end of the line is not counted.

A Mad Tea-Party (original version)

There was a table set out under a tree in front of the house, and the March Hare and the Hatter were having tea at it: a Dormouse was sitting between them, fast asleep, and the other two were using it as a cushion, resting their elbows on it, and talking over its head. 'Very uncomfortable for the Dormouse,' thought Alice; 'only, as it's asleep, I suppose it doesn't mind.' The table was a large one, but the three were all crowded together at one corner of it: 'No room! No room!' they cried out when they saw Alice coming. 'There's PLENTY of room!' said Alice indignantly, and she sat down in a large arm-chair at one end of the table.

```
Program Execution ...

This program will display the contents of a provided text file with all
occurrences of the letter 'e' removed.

Enter file name (including file extension): alice_tea_party.txt

Thr was a tabl st out undr a tr in front of th hous, and th
March Har and th Hattr wr having ta at it: a Dormous was sitting
btwn thm, fast aslp, and th othr two wr using it as a cushion,
rsting thir lbows on it, and talking ovr its had. 'Vry uncomfortabl
for th Dormous,' thought Alic; 'only, as it's aslp, I suppos it dosn't
mind.' Th tabl was a larg on, but th thr wr all crowdd togthr
at on cornr of it: 'No room! No room!' thy crid out whn thy saw Alic
coming. 'Thr's PLNTY of room!' said Alic indignantly, and sh sat down in
a larg arm-chair at on nd of th tabl

70 occurrences of the letter 'e' removed
Percentage of data lost: 11 %
Modified text in file e_alice_tea_party.txt
```

FIGURE 8-6 Execution of the Sparse Text Program

On **line 20**, variable modified_line is then set to the new string produced by the replace method applied to the current line. Each occurrence of 'e' and 'E' is replaced with the empty string, thus removing these occurrences from the string. Note the consecutive calls to the replace method,

modified_line 5 line.replace('e',empty_str).replace('E',empty_str)

This is possible to do when the first method call returns a value (object) for which the second method call can be applied. In this case, line.replace('e',empty_str) returns a string (consisting of a copy of the original string with all occurrences of the letter 'e' removed), which is then operated on by the second instance of the method call to remove all instances of 'E'.

On **line 21**, the number of characters removed (num_removals) is updated. The number removed from the current line is determined by taking the difference between the original line length (without the newline character) and the length of

the new modified line minus one, so that the newline character in the modified line is not counted. Finally, the modified line of output to the screen (as well as to the output file) is displayed so the user may observe the results of the file processing as it is occurring. A tuple including the number of total characters in the file and the number of removed characters is returned as the function value.

```
# Sparse Text Program

def createModifiedFile(input_file, outputfile):

    """For text file input_file, creates a new version in file outputfile
        in which all instances of the letter 'e' are removed.
    """

    empty_str = ''
    num_total_chars = 0
    num_removals = 0

    for line in input_file:

        # save original line length
        orig_line_length = len(line) - 1
        num_total_chars = num_total_chars + orig_line_length

        # remove all occurrances of letter 'e'
        modified_line = line.replace('e',empty_str).replace('E',empty_str)
        num_removals = num_removals + \
                        (orig_line_length - (len(modified_line)-1))

        # simulataneouly output line to screen and output file
        print(modified_line.strip('\n'))
        output_file.write(modified_line)

    return (num_total_chars, num_removals)

# --- main

# program welcome
print("This program will display the contents of a provided text file")
print("with all occurrences of the letter 'e' removed.\n”)

# open files for reading and writing
file_name = input('Enter file name (including file extension): ')
input_file = open(file_name,'r')
new_file_name = 'e_' + file_name
output_file = open(new_file_name,'w')

# create file with all letter e removed
print()
num_total_chars, num_removals = createModifiedFile(input_file, output_file)

# close current input and output files
input_file.close()
output_file.close()

# display percentage of characters removed
print()
print(num_removals, "occurrences of the letter 'e' removed")
print('Percentage of data lost:',
      int((num_removals / num_total_chars) * 100), '%')
print('Modified text in file', new_file_name)
```

FIGURE 8-7 Sparse Text Program

Self-Test Questions

1. Some string methods alter the string they are called on, while others return a

new altered version of the string. (TRUE/FALSE)
2. The find method returns the number of occurrences of a character or substring within a given string. (TRUE/FALSE)
3. Which of the results below does s[2:4] return for the string s = 'abcdef'. **(a)** 'cd' **(b)** 'bcd' **(c)** 'bc' **(d)** 'cde'

4. Indicate which of the following is true.
(a) String method isdigit returns true if the string applied to contains any digits.
(b) String method isdigit returns true if the string applied to contains only digits.

5. Indicate which of the following s.replace('c','e')returns for s 5'abcabc'. **(a)** 'abeabc' **(b)** 'abeabe'

6. Which of the results below does s.strip('-')return for the string
s 5'---ERROR---'.
(a) '---ERROR' **(b)** 'ERROR---' **(c)** 'ERROR'

ANSWERS: 1. False, 2. False, 3. (a), 4. (b), 5. (b), 6. (c)

8.4 Exception Handling

Various error messages can occur when executing Python programs. Such errors are called *exceptions*. So far we have let Python handle these errors by reporting them on the screen. Exceptions can be “caught” and “handled” by a program, however, to either correct the error and continue execution, or terminate the program gracefully. We take this opportunity to discuss the fundamentals of exception handling in the use of text files.

8.4.1 What Is an Exception?

An **exception** is a value (object) that is *raised* (“thrown”) signaling that an unexpected, or “exceptional,” situation has occurred. Python contains a predefined set of exceptions referred to as **standard exceptions**. We list some of the standard exceptions in Figure 8-8.

ImportError	raised when an import (or from…import) statement fails
IndexError	raised when a sequence index is out of range
NameError	raised when a local or global name is not found
TypeError	raised when an operation or function is applied to an object of inappropriate type
ValueError	raised when a built-in operation or function is applied to an appropriate type, but of inappropriate value
IOError	raised when an input/output operation fails (e.g., “file not found”)

FIGURE 8-8 Some Standard Exceptions in Python

The standard exceptions are defined within the exceptions module of the Python Standard Library, which is automatically imported into Python programs. We have seen a number of these exceptions before in our programming,

... lst 5 [1, 2, 3]
... lst[3]
Traceback (most recent call last):

File " ,pyshell#4.", line 1, in ,module. lst[3]
IndexError: list index out of range ... 2 1 '3'
Traceback (most recent call last):

File " ,pyshell#7.", line 1, in ,module. 2 1 '3'
TypeError: unsupported operand type(s) for 1: 'int' and 'str' ... lsst[0]
Traceback (most recent call last):
File ",pyshell#5.", line 1, in ,module. lsst[0]
NameError: name 'lsst' is not defined

... int('12.04')

Traceback (most recent call last): File ",pyshell#9.", line 1, in ,module. int('12.04')

ValueError: invalid literal for int() with base 10: '12.04'

Raising an exception is a way for a function to inform its client a problem has occurred that *the function itself cannot handle*. For example, suppose a function called getYN prompts a user to enter 'y' or 'n'. If the user enters something other than these two values, the function can simply prompt the user to re-enter. On the other hand, if a function called isEven is to be passed a numeric value, but is passed a string instead, it cannot correct the problem. It can only notify the client of the problem, leaving it up to the client to determine what to do. We next discuss in detail the raising and catching of exceptions.

An **exception** is a value (object) that is "raised" by a function signaling that an unexpected, or "exceptional," situation has occurred that the function itself cannot handle.

8.4.2 The Propagation of Raised Exceptions

Raised exceptions are not required to be handled in Python. When an exception is raised and not handled by the client code, it is automatically propagated back to the *client's* calling code (and *its* calling code, etc.) until handled. If an exception is thrown all the way back to the top level (main module) and not handled, then the program terminates and displays the details of the exception as depicted in Figure 8-9.

For example, a program might not be able to determine whether a password is valid at the source of the input. Rather, the password must be verified in a password file that is accessed only after a *chain* of function calls have been made. If the password is found invalid, an exception is propagated from the function identifying the error all the way back to the function responsible for user input, which can then prompt the user to re-enter their password.

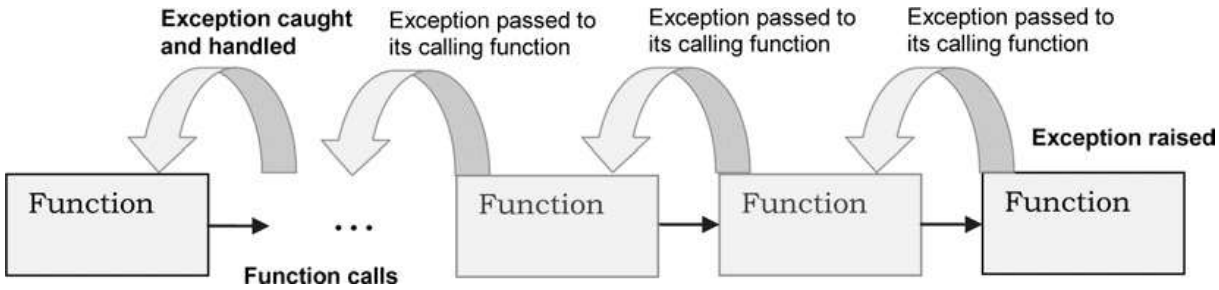


FIGURE 8-9 The Propagation of Exceptions

An exception is either handled by the client code, or automatically propagated back to the client's calling code, and so on, until handled. If an exception is thrown all the way back to the main module (and not handled), the program

terminates displaying the details of the exception.

8.4.3 Catching and Handling Exceptions

Many of the functions in the Python Standard Library raise exceptions. For example, the factorial function of the Python math module raises a ValueError exception when a negative value is passed to it, as shown in Figure 8-10.

```
import math

num = int(input('Enter number to compute factorial of: '))
print('The factorial of', num, 'is', math.factorial(num))
```

```
Enter number to computer factorial of: -5
The factorial of -5 is
Traceback (most recent call last):
  File "C:\My Python Programs\exception_example1.py", line 4, in <module>
    print 'The factorial of', num, 'is', math.factorial(num)
ValueError: factorial() not defined for negative values
```

FIGURE 8-10 Simple Program without Exception Handling

When 25 is passed to the factorial function, an exception is raised, thrown back to the client code to catch and handle. Since the client code (here, the code in the main module) does not attempt to catch the exception, the exception is caught by the Python interpreter, causing the program to terminate and display an error message indicating the exception type. The last line of the message indicates that a ValueError exception occurred within the factorial function. The previous lines indicate where in the client code this function was called (line 4). In Figure 8-11 we give a version of the program that catches the raised exception.

```
import math

num = int(input('Enter number to compute factorial of: '))
try:
    print(math.factorial(num))
except ValueError:
    print('Cannot compute the factorial of negative numbers')
```

```
Enter number to compute factorial of: -5
Cannot compute the factorial of negative numbers
```

FIGURE 8-11 Simple Program with Exception Handling

The call to the factorial function is contained within a **try suite** (**try block**)—the block of code surrounded by try and except headers. The suite following the except header is referred to as an **exception handler**. This exception header "catches" exceptions of type ValueError. The exception handler in this case simply outputs an error message to the user before terminating the program. This program termination is much more user-friendly than the previous version.

Finally we show a version of the program that recovers from the exception. Rather than terminating the program, the user is prompted again for input so that the program can continue executing. This new version is given in Figure 8-12.

```
import math

num = int(input('Enter number to compute factorial of: '))
valid_input = False

while not valid_input:
    try:
        result = math.factorial(num)
        print(result)
        valid_input = True
    except ValueError:
        print('Cannot compute factorial of negative numbers')
        num = int(input('Please re-enter: '))
```

```
Enter number to compute factorial of: -5
Cannot compute the factorial of negative numbers
Please re-enter: 5
120
```

FIGURE 8-12 Program Recovery via Exception Handling

Note that within a try suite in Python, any statement making a call (either directly or indirectly) to a function that raises an exception causes the rest of the statements in the suite to be skipped, as depicted in the figure.
Exceptions are caught and handled in Python by use of a **try block** and **exception handler**.

8.4.4 Exception Handling and User Input
Besides exceptions raised by built-in functions, programmer-defined functions may raise exceptions as well. Suppose we prompted the user to enter the current month as a number,
month 5 input('Enter current month (1–12): ') The input function will return

whatever is entered as a string. We can do integer type conversion on this value to make it an integer type,
month 5 int(input('Enter current month (1–12): '))

If the input string contained non-digit characters (except for 1and 2), the int function would raise a ValueError exception. However, there also needs to be a check for values outside the range 1–12. This can be done as shown in Figure 8-13.

```
valid = False

while not valid:
    try:
        month = int(input('Enter current month (1-12): '))

        while month < 1 or month > 12:
            print('Invalid Input - Must be in the range 1-12'
            month = int(input('Enter current month (1-12): '))
        valid = True
    except ValueError:
        print('Invalid Month Value')
```

FIGURE 8-13 Input Error Checking (Version 1)

Athough this works, a better approach is to design a function called getMonth that raises a ValueError exception for either error condition—if the user enters non-digit characters, or if the user enters a numeric value outside the range 1–12. Such a function is given in Figure 8-14.

```
def getMonth():
    month = int(input('Enter current month (1-12): '))

    if month < 1 or month > 12:
        raise ValueError('Invalid Month Value')

    return month
```

FIGURE 8-14 Programmer-Defined Function with Raised Exception
In this version of getMonth, if non-digit characters are entered, the ValueError exception is automatically raised by the built-in int type conversion function,

... getMonth()
Enter current month (1–12): 1a
Traceback (most recent call last):

File " ,pyshell#18 .", line 1, in ,module .
getMonth()
File " C:\My Python Programs\exception /Raising Excpt Test.py", line 2, in getMonth
month 5 int(input('Enter current month (1–12): ')) ValueError: invalid literal for int() with base 10: '1a'

Since there is no try block around the input assignment statement, the exception is not caught and therefore is thrown back to the Python interpreter. If a valid numeric string is entered, but the value is outside the range 1–12, then a ValueError is raised by function getMonth and is thrown back to the client,

... getMonth()
Enter current month (1–12): 14
Traceback (most recent call last):

File " ,pyshell#23 .", line 1, in ,module .
getMonth()
File " C:\My Python Programs\exception /Raising Excpt Test.py", line 5, in getMonth
raise ValueError('Invalid Month Value')
ValueError: Invalid Month Value

Note that raised exceptions can optionally have a description that further describes the error, as has been done here,
raise ValueError('Invalid Month Value')

This error message is displayed when the ValueError generated is the one thrown by function getMonth. With this function, the less elegant error checking code in Figure 8-13 can now be replaced with that in Figure 8-15.

```
valid = False

while not valid:
    try:
        month = getMonth()
        valid = True
    except ValueError:
        print('Invalid Month Entry\n')
```

FIGURE 8-15 Input Error Checking (Version 2)
In this case, when an invalid input is entered, we get the following,

Enter current month (1–12): 1a Invalid Month Value
Enter current month (1–12): 14 Invalid Month Value

This is a much cleaner error reporting for the user. The error message Invalid Month Value, however, doesn't indicate *why* the input was invalid (that is, either that an invalid character was found, or an integer outside the range 1–12 was entered). We could go one step further and output the error message associated with the specific ValueError exception thrown, as shown in Figure 8-16.

```
valid = False

while not valid:
    try:
        month = getMonth()
        valid = True
    except ValueError as err_mesg:
        print(err_mesg,'\n')
```

FIGURE 8-16 Input Error Checking (Version 3)

The line except ValueError as err_mesg not only catches the ValueError exceptions, but with as *identifier* added (here as err_mesg), the error message of the particular ValueError exception is assigned to the specified identifier name,

Enter current month (1–12): 1a
invalid literal for int() with base 10: '1a' Enter current month (1–12): 14
Invalid Month Value
Enter current month (1–12): 12
...
We next look at exception handling related to file processing.
Programmer-defined functions may raise exceptions in addition to the exceptions raised by the built-in functions of Python.

8.4.5 Exception Handling and File Processing

We saw that when opening a file for reading, an exception is raised if the file cannot be found. In this case, the standard IOError exception is raised and the program terminates with a 'No such file or directory' error message. We can catch this exception and handle the error as shown in Figure 8-17.

```
file_name = input('Enter file name: ')
empty_str = ''

input_file_opened = False

while not input_file_opened:
    try:
        input_file = open(file_name, 'r')
        input_file_opened = True

        line = input_file.readline()

        while line != empty_str:
            print(line.strip('\n'))
            line = input_file.readline()
    except IOError:
        print('File Open Error\n')
        file_name = input('Enter file name: ')
```

FIGURE 8-17 Exception Handling of Open File Error

V ariable file_name stores the file name entered by the user. Variable input_file_opened is initialized to False. The while loop continues to iterate as long as input_file_opened is False. Because of the try block within the loop, every time that open(file_name, 'r') raises an exception, the remaining lines in the try block are skipped, and the exception handler following the except header is executed. Only when the call to open does not throw an exception do all the instructions in the try block get executed, with the program continuing after the while loop. A similar, but much less likely exception can be raised when opening a file for writing and the file system (hard disk, for example) we want to write to is full.

Note that when reading from a text file, the readline method does not raise any exceptions. When the end of file is reached, readline returns an empty string rather than throwing an exception.
IOError exceptions raised as a result of a file open error can be caught and handled.

8.4.6 Let's Apply It—Word Frequency Count Program

The following Python program (Figure 8-19) prompts the user for the name of a text file to open and a word to search for, and displays the number of times that the word occurs within the file. This program utilizes the following Python programming features:

➤ text files/readline()
➤ string methods lower(), index()
Example execution of the program is given in Figure 8-18.

Program execution begins at **line 77**. First, the program welcome is displayed. On **line 81**, function getFile is called, which prompts the user for the file name to open for reading. It returns a tuple containing both the file name and the associated input file object, assigned to variables file_name and input_file, respectively.

Text File: wordtest.txt

```
This is a sample text file.
The word 'the' in this file is to be counted.
Must be careful not to count words like their and there.
Should only count the word "the" when it is a separate word.
This is the last line of the file.
```

Program Execution ...

```
This program will display the number of occurrences of a
specified word within a given text file

Enter input file name: word
Input file not found - please reenter

Enter input file name: wordtest.txt
Enter word to search: apple
No occurrences of word 'apple' found in file wordtest.txt
```

Program Execution ...

```
This program will display the number of occurrences of a
specified word within a given text file

Enter input file name: wrdtest.txt
Input file not found - please reenter

Enter input file name: wordtest.txt
Enter word to search: the
The word 'the' occurs 6 times in file wordtest.txt
```

FIGURE 8-18 Execution of the Word Frequency Count Program

On **line 84** the user is prompted for the word to search for, stored in variable search_word. In the following line, search_word is reassigned to all lowercase characters by use of method lower(). The file lines read in function countWords are also converted to lowercase so that the matching of words does not depend

on whether letters are in uppercase or lowercase (in other words, so that it is not *case sensitive*).

The function that does most of the work, function countWords, is called on **line 88**. It is passed the file object, input_file, as well as the words to search for. The function returns, as an integer, the number of occurrences found. Finally, in **lines 91–98**, the results of the search are displayed.

The getFile function called from the main section of the program prompts the user for a file name to open (**line 12**). The file is opened for reading on **line 13**. If the file is not found, an IOException is raised and caught by the except clause on **line 15**. In this case, an error message is printed (**line 16**) and the while loop at **line 10** iterates again, prompting the user to re-enter. If an exception is not thrown, then **line 14** is executed setting input_file_opened to True, causing the loop to terminate and the function to return both the file name and input file object as a tuple.

Function countWords, the final function of the program, is on **lines 21–72**. It is passed the input_file object and the search word, and returns the number of occurrences of the search word within the file. **Lines 28–31** perform the initialization for the function. Variable space is assigned to a string containing only a single blank character (to aid in the readability of the program). Variable num_occurrences, which keeps count of the number of times the search word appears in the file, is initialized to 0. And word_delimiters is set to a tuple

```
# Word Frequency Count Program

def getFile():

    """Returns the file name and associated file object for reading
       the file as a tuple of the form (file_name, input_file).
    """

    input_file_opened = False
    while not input_file_opened:
        try:
            file_name = input('Enter input file name: ')
            input_file = open(file_name, 'r')
            input_file_opened = True
        except IOError:
            print('Input file not found - please reenter\n')

    return (file_name, input_file)

def countWords(input_file, search_word):

    """Returns the number of occurrences of search_word in the
       provided input_file object.
    """

    # init
    space = ' '
    num_occurrences = 0
    word_delimiters = (space, ',', ';', ':', '.','\n',
                       '"',"'", '(', ')')

    search_word_len = len(search_word)

    for line in input_file:
        end_of_line = False

        # convert line read to all lower case chars
        line = line.lower()

        # scan line until end of line reached
        while not end_of_line:
            try:
                # search for word in current line
                index = line.index(search_word)

                # if word at start of line followed by a delimiter
                if index == 0 and line[search_word_len] in
                   word_delimiters:
                        found_search_word = True
```

FIGURE 8-19 Word Frequency Count Program (*Continued*)

containing all the possible word delimiters—a space character, a comma, a semicolon, a colon, a period, and a newline character.

Because the length of the search word is used multiple times, it is calculated

once and stored in variable search_word_len (**line 33**). The for loop on **line 35** reads each line of the file. Each line is scanned for all occurrences of the search word. Variable end_of_line is initialized to

```
53                  # if search word within line, check chars before/after
54                  elif line[index - 1] in word_delimiters and \
55                     line[index + search_word_len] in word_delimiters:
56                      found_search_word = True
57
58                  # if found within other letters, then not search word
59                  else:
60                      found_search_word = False
61
62                  # if search word found, increment count
63                  if found_search_word:
64                       num_occurrences = num_occurrences + 1
65
66                  # reset line to rest of line following search word
67                  line = line[index + search_word_len: len(line)]
68
69              except ValueError:
70                  end_of_line = True
71
72      return num_occurrences
73
74  # ---- main
75
76  # program welcome
77  print('This program will display the number of occurrences of a')
78  print('specified word within a given text file\n')
79
80  # open file to search
81  file_name, input_file = getFile()
82
83  # get search word
84  search_word = input('Enter word to search: ')
85  search_word = search_word.lower()
86
87  # count all occurrences of search word
88  num_occurrences = countWords(input_file, search_word)
89
90  # display results
91  if num_occurrences == 0:
92      print('No occurrences of word', "'" + search_word + "'",
93            'found in file', file_name)
94  else:
97      print('The word', "'" + search_word + "'", 'occurs',
98            num_occurrences, 'times in file', file_name)
```

FIGURE 8-19 Word Frequency Count Program

False (**line 36**) for the while loop below it. On **line 39**, all characters in the line are converted to lowercase to allow for the matching of words of different case (mentioned above). Then, beginning on **line 42**, a while loop is used to search for all occurrences of the search word in the current line, continuing until the end of line is found.

On **line 45**the index string method is called on the current line to search for the first occurrence of the search word. It therefore performs the same task as the find method. However, whereas find returns a 21 if the string is not found, the index method raises a ValueError exception instead. Since we have now introduced exception handling in Python, we here use the index method.

The if statement on **line 48** checks if the search string matches the beginning characters of the line (followed by a delimiter). This determines if the first word of the current line matches the search string. If so, then found_search_string is set to True (**line 50**). Otherwise, a check is made on **line 54** to see if the matched string in the current line is immediately preceded and followed by a delimiter. If found, then found_search_string is also set to True, otherwise, found_ search_string is set to False. On **lines 63–64**, if found_search_word is True, num_ occurences is incremented by 1.

V ariable line is reassigned to the substring following the last word found (**line 67**). This is needed because the index method only finds the *fi r s t* occurrence of a given substring within a string, and not *all* occurrences. Therefore, the line is continually shortened and scanned until no further instances of the word are found within the line. Once the complete line is scanned, the while loop terminates and control is returned to the top of the for loop (on **line 35**). When there are no more lines of the file to read, execution continues at **line 72** and the value of num_ occurrences is returned.

Self-Test Questions
1. An exception is,
(a) an object **(b)** a standard module **(c)** a special function
2. Which of the following is not a standard exception in Python?
(a) ValueError **(b)** AssignmentError **(c)** NameError **(d)** IOError **3.** The standard exceptions are automatically imported into Python programs. (TRUE/FALSE)
4. All raised standard exceptions must be handled in Python. (TRUE/FALSE) **5.** Which one of the following is true?

(a) When calling the built-in open function for file handling, it *must* be called from within a try block with an appropriate exception handler.
(b) When calling the built-in open function for file handling, it *should* be called from within a try block with an appropriate exception handler.

6. In addition to catching standard exceptions in Python, a program may also

raise standard exceptions. (TRUE/FALSE)

ANSWERS: 1. (a), 2. (b), 3. True, 4. False, 5. (b), 6. True

COMPUTATIONAL PROBLEM SOLVING

8.5 Cigarette Use/Lung Cancer Correlation Program

In this section, we design a program using data from the Centers for Disease Control and Prevention (CDC) for computing the correlation between cigarette use and incidences of lung cancer (Figure 8-20). One data set gives the percentage of the population that smoke cigarettes within the United States by state. The other gives the rate of lung cancer per 100,000 individuals by state. The computed correlation, along with other factors, can be used to determined if there is a causal relationship between the two.

8.5.1 The Problem

The problem is to calculate a correlation value for two sets of data. Correlation is measured on a scale of 21 to 1, with 1 indicating a *perfect positive correlation*, and 21 a *perfect negative correlation*. For example, there is a positive correlation between the amount of ice cream sold and the current temperature—as temperatures rise, so do the sales of ice cream. There is a negative correlation between the number of snow blowers sold and the current temperature—as temperatures rise, the number of snow blowers sold decreases.

In a perfect correlation, knowing one value allows one to determine the *exact* value of the other. In real-world situations, perfect correlations are almost never found. For example, many other factors can affect exactly how much ice cream is sold—there may be a truckers' strike reducing deliveries, there may be a recall of a certain brand of ice cream, and so forth. Therefore, most correlation values fall somewhere between 21 and 1.

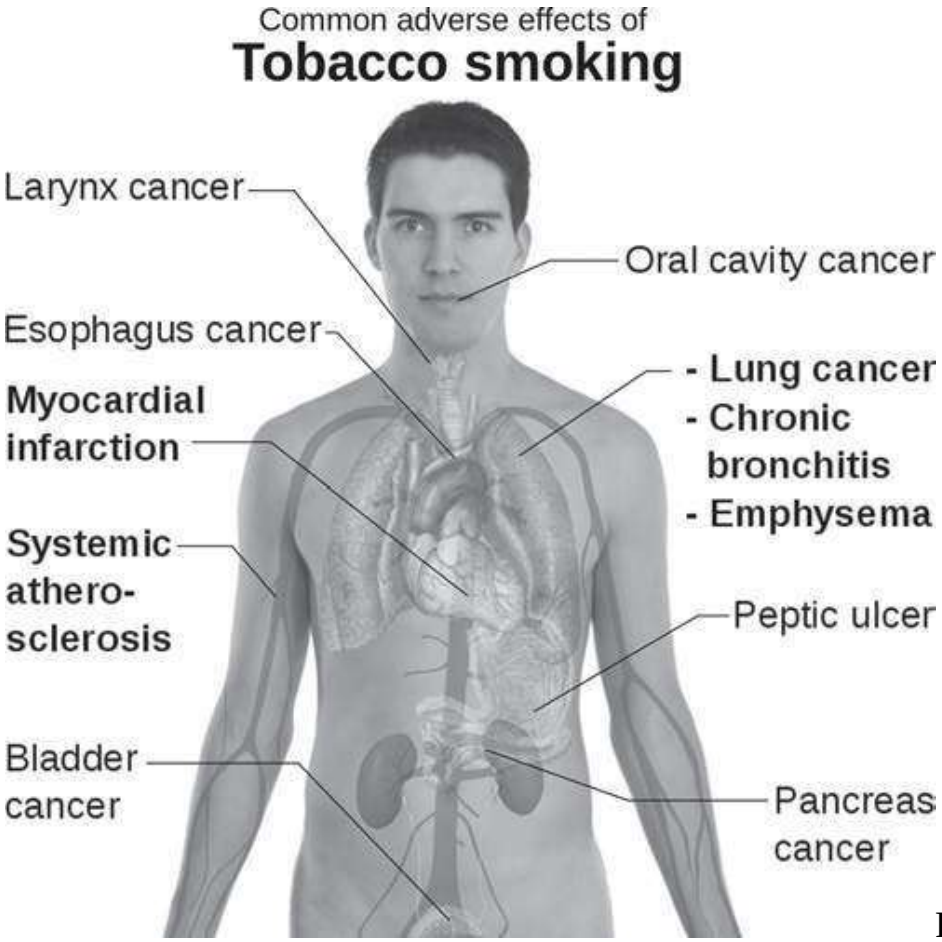


FIGURE 8-20
Adverse Effects of Cigarette Smoking

8.5.2 Problem Analysis

In the two sets of data provided in Figure 8-21, one contains the percentage of the population that smokes state-by-state in the United States, and the other the state-by-state rates of incidence of lung cancer per 100,000 individuals.

The mathematical formula for computing correlation is given below.

$$r = \frac{N\sum xy - \sum x \sum y}{\sqrt{(N\sum x^2 - (\sum x)^2)(N\sum y^2 - (\sum y)^2)}}$$

where,
N is equal to the number of pairs of values in the data
x and y are a given pair of values
$\sum xy$ is the sum of the products of paired scores
$\sum x$ is the sum of the scores of one data set
$\sum y$ is the sum of the scores of the other data set

$\sum x^2$ is the sum of the squares of the scores of one data set
$\sum y^2$ is the sum of the squares of the scores of the other data set

For our data, *N* is equal to 48 (all states except Arizona and Wisconsin, which were not provided in the data sets from the CDC). An example of *x* and *y* values are the two values for Alabama, *x* 5 23.3 (percent of population that smokes) and *y* 5 75.1 (lung cancer cases per 100,000 individuals). Note that to apply this formula, the values of *x* and *y* do not have to be in the same units.

```
State, Percent Cigarette Smokers
Alabama,23.3
Alaska,24.2
Arkansas,23.7
California,14.9
Colorado,17.9
Connecticut,17
Delaware,21.7
Florida,21
Georgia,20
Hawaii,17.5
Idaho,16.8
Illinois,20.5
Indiana,24.1
Iowa,21.5
Kansas,20
Kentucky,28.6
Louisiana,23.4
Maine,20.9
Maryland,17.8
Massachusetts,17.8
Michigan,22.4
Minnesota,18.3
Mississippi,25.1
Missouri,23.3
Montana,19
Nebraska,18.6
Nevada,22.2
New Hampshire,18.7
New Jersey,18.1
New Mexico,20.2
New York,18.3
North Carolina,22.1
North Dakota,19.6
Ohio,22.5
Oklahoma,25.1
Oregon,18.5
Pennsylvania,21.5
Rhode Island,19.3
South Carolina,22.3
South Dakota,20.4
Tennessee,22.6
Texas,18.1
Utah,9.8
Vermont,18
Virginia,19.3
Washington,17.1
West Virginia,25.7
Wyoming,21.6
```

CDC Data on the Percentage of the Population that Smoke Cigarettes within the U.S. by State (2006)

```
Cases Lung Cancer per 100,000
Alabama,75.1
Alaska,69.8
Arkansas,77.7
California,51.6
Colorado,48.4
Connecticut,68.3
Delaware,83.5
Florida,68.7
Georgia,71
Hawaii,51.7
Idaho,54
Illinois,71.4
Indiana,77.2
Iowa,67.1
Kansas,68.4
Kentucky,97.2
Louisiana,77.5
Maine,80.1
Maryland,62.9
Massachusetts,64.5
Michigan,72
Minnesota,55.8
Mississippi,77.1
Missouri,77.9
Montana,60.8
Nebraska,61.3
Nevada,73.3
New Hampshire,66.8
New Jersey,64.5
New Mexico,43.9
New York,63
North Carolina,75.3
North Dakota,53
Ohio,72.8
Oklahoma,80.8
Oregon,64.4
Pennsylvania,70
Rhode Island,69.5
South Carolina,70.7
South Dakota,59.2
Tennessee,82.2
Texas,62
Utah,28
Vermont,79.2
Virginia,65.8
Washington,65.2
West Virginia,91
Wyoming,48.9
```

CDC Data on the Rate of Lung/Bronchus Cancer per 100,000 Individuals within the U.S. by State (2006)

FIGURE 8-21 Cigarette Smoking and Incidence of Lung Cancer Data

8.5.3 Program Design

The data is provided in two comma-separated (CSV) files. Thus, we must first

read the data from these file. Using this data, we must then compute the correlation of the two sets of data on a scale of 21 (a perfect negative correlation) to 1 (a perfect positive correlation).

Meeting the Program Requirements

The program must only display the correlation value between the two sets of data. Since the program requirements do not specify how the result is to be presented (for example, on a scale), the raw numerical value will simply be displayed on the screen.

Data Description

The data from the files will be stored in two lists so that the values can be easily accessed during the calculation of the correlation. Thus, the data is simply organized as two corresponding “parallel” lists of floating-point values.

Algorithmic Approach
The algorithm for this program is the means of calculating the correlation values using the mathematical formula given above.
Overall Program Steps
The overall steps in the program design are given in Figure 8-22.

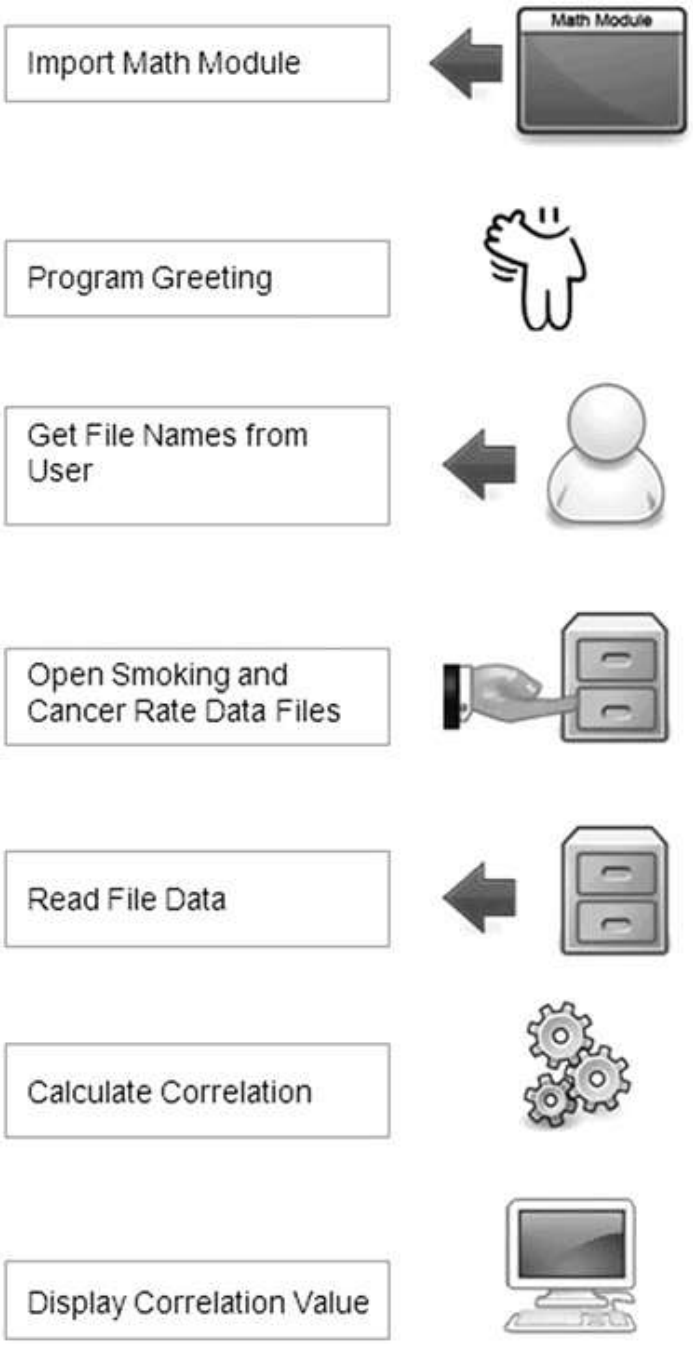


FIGURE 8-22 Overall Steps of the Smoking/
Lung Cancer Correlation Program

Modular Design

The modular design for this program is given in Figure 8-23.

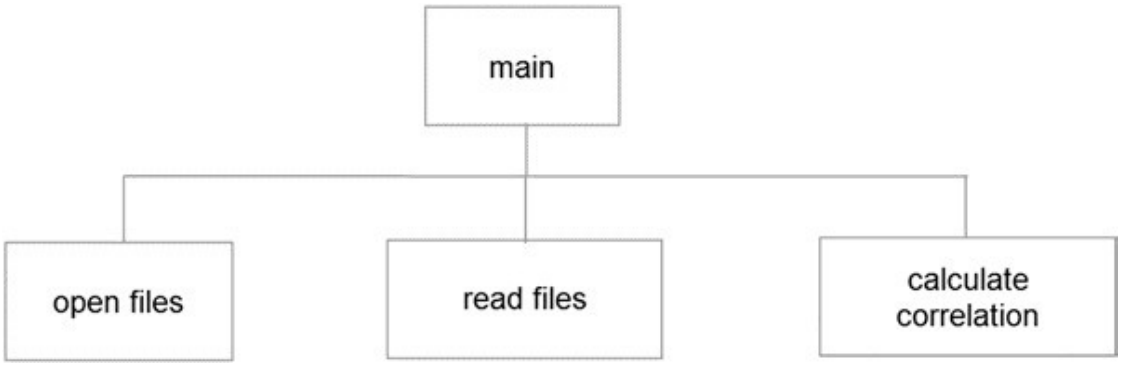


FIGURE 8-23 Modular Design of the Smoking/Lung Cancer Correlation Program

Following this modular design, there are three functions in the program: openFiles, readFiles, and calculateCorrelation. The main module utilizing these functions is given in Figure 8-24.

```
# ---- main

# program greeting
print('This program will determine the correlation between')
print('cigarette smoking and incidences of lung cancer\n')

try:
    # open data files
    smoking_datafile, cancer_datafile = openFiles()

    # read data
    smoking_data, cancer_data = readFiles(smoking_datafile, cancer_datafile)

    # calculate correlation value
    correlation = calculateCorrelation(smoking_data, cancer_data)

    # display correlation value
    print('r_value = ', correlation)
except IOError as e:
    print(str(e))
    print('Program terminated ...')
```

FIGURE 8-24 Main Module of the Smoking/Lung Cancer Correlation Program Design

8.5.4 Program Implementation and Testing

We develop and test the program in stages by developing and unit testing each of the three functions of the modular design. We begin with function openFiles.

Development and Unit Testing of Function **openFiles**

In the main module, we see that openFiles is designed as a value-returning function with no arguments, returning file objects for each of the opened data

files. An implementation of this function is given in Figure 8-25.

```
1  # Module module_openFiles
2
3  def openFiles():
4      """
5          Prompts the user for the file names to open, opens the files, and
6          returns the file objects for each in a tuple of the form
7          (smoking_datafile, cancer_datafile).
8
9          Raises an IOError exception if the files are not successfully
10         opened after four attempts of entering file names.
11     """
12     # init
13     smoking_datafile_opened = False
14     cancer_datafile_opened = False
15     num_attempts = 4
16
17     # prompt for file names and attempt to open files
18     while ((not smoking_datafile_opened) or (not cancer_datafile_opened)) \
19            and (num_attempts > 0):
20         try:
21             if not smoking_datafile_opened:
22                 file_name = input('Enter smoking data file name: ')
23                 smoking_datafile = open(file_name, 'r')
24                 smoking_datafile_opened = True
25
26             if not cancer_datafile_opened:
27                 file_name = input('Enter lung cancer data file name: ')
28                 cancer_datafile = open(file_name, 'r')
29                 cancer_datafile_opened = True
30         except IOError:
31             print('File not found:', file_name + '.', 'Please reenter\n')
32             num_attempts = num_attempts - 1
33
34     # if one of more file not opened, raise IOError exception
35     if not smoking_datafile_opened or not cancer_datafile_opened:
36         raise IOError
37     else:
38         return (smoking_datafile, cancer_datafile)
```

FIGURE 8-25 Implementation of Function openFiles

Function openFiles is initially put in its own Python file so that it can be easily imported for unit testing. The function handles the task of inputting from the user the data file names (for both the smoking-related data file and the lung cancer–related data file), opening the files for reading, and returning the two file objects created as a tuple of the form (smoking_datafile, cancer_ datafile).

In the init section, variables smoking_datafile_opened and cancer_datafile_ opened are initialized to False (**lines 13–14**). They are used in the control of the

following while loop so that the user is continually prompted until correct file names for the data files are entered. On **line 15**, variable num_attempts is initialized to 4. This variable is decremented each time that an invalid file name (for either file) is entered. If the counter reaches 0, the while loop terminates and the final if statement checks if the files have been successfully opened. If not, then an IOError exception is raised, terminating the function.

The while loop at **line 18** continues to iterate as long as either of the two data files has not been successfully opened, and as long as the value of variable num_attempts is greater than zero. The statements in the loop are contained within a try block (**lines 21–29**), used to catch any IOError exceptions raised when attempting to open either of the two data files by call to method open,

smoking_datafile 5 open(file_name, 'r')
cancer_datafile 5 open(file_name, 'r')

Following the calls to method *open* for the smoking data file (**line 23**) and the cancer data file (**line 28**) is an assignment statement that assigns smoking_datafile_opened and cancer_datafile_ opened, respectively, to True. Since these statements would not be reached if the call to the open method raised an exception, these Boolean variables provide a means of determining which, if any, of the two files has been opened successfully. If either call to method open raises an exception, it is caught by the except clause (**line 30**). As a result, on **line 31** the error message 'File not found: *filename*. Please reenter', for the *filename* in variable file_name is displayed. In addition, variable num_attempts is decremented by 1 (**line 32**), and control returns to the top of the while loop. Finally, when the loop terminates with both files successfully opened, a tuple containing each of the file objects is returned (**line 38**).

Function openFiles is not easily tested interactively in the Python shell. Therefore, we develop a *test driver* program that simply calls function openFile (imported from module_ openFiles) to open the data files, and displays the first line of each file on the screen. The test driver program is given in Figure 8-26.

```
# TEST DRIVER for Function openFiles of Smoking/Cancer Correlation Program

from module_openFiles import *

empty_str = ''
print('TESTING FUNCTION openFiles')

try:
    # open data files
    smoking_datafile, cancer_datafile = openFiles()

    # display sucessfully opened
    print('Data files successfully opened')

    # display first line of smoking data file
    print('\nReading first line of smoking data file ...')
    line = smoking_datafile.readline()
    print(line.strip('\n'))

    # display first line of cancer data file
    print('\nReading first line of cancer data file ...')
    line = cancer_datafile.readline()
    print(line.strip('\n'))
except IOError:
    print('Too many attempts of opening input files')
    print('Program terminated ...')
```

FIGURE 8-26 Test Driver for Function openFiles

Since function openFiles is designed to raise an IOError exception if the files are not successfully opened after four attempts, we test this aspect of the program by purposely entering incorrect file names four times. The result of this testing is given in Figure 8-27.

```
TESTING FUNCTION openFiles
Enter smoking data file name: CDC_Cigarette_Smking_Data.csv
File not found: CDC_Cigarette_Smking_Data.csv. Please reenter

Enter smoking data file name: CDC_Cigarette_Smokin_Data.csv
File not found: CDC_Cigarette_Smokin_Data.csv. Please reenter

Enter smoking data file name: CDC_Cigarette_Smoking_Data.csv
Enter file name with lung cancer data: CDC_Lng_Cancer_Data.csv
File not found: CDC_Lng_Cancer_Data.csv. Please reenter

Enter lung cancer data file name: CDC_Lung_Caner_Data.csv
File not found: CDC_Lung_Caner_Data.csv. Please reenter

Too many attempts of reading input file
Program terminated ...
>>>
```

FIGURE 8-27 Invalid File Names Test Case for Function openFiles

Thus, this aspect of the function appears to be working. We next give it correct file names to see if the function successfully opens and correctly displays the first line of each file. The results are shown in Figure 8-28.

```
TESTING FUNCTION openFiles
Enter smoking data file name: CDC_Cigarette_Smoking_Data.csv
Enter lung cancer data file name: CDC_Lung_Cancer_Data.csv
Data files successfully opened

Reading first line of smoking data file ...
State, Percent Cigarette Smokers , CDC 2006 Data (except AZ, WI)

Reading first line of cancer data file ...
Cases Lung Cancer per 100000 CDC Data 2006 (except AZ,WI)
>>>
```

FIGURE 8-28 Valid File Names Test Case for Function openFiles

The output is as expected. We can assume that function openFiles is implemented correctly. We therefore unit test function readFiles next.

Development and Unit Testing of Function **readFiles**

As done for function openFiles, we put function readFiles in its own file to be imported by its test driver, named module_readFiles.py. An implementation of function readFiles, implemented as a separate module, is given in Figure 8-29.

```
1  # Module module_readFiles
2
3  def readFiles(smoking_datafile, cancer_datafile):
4
5       """Reads the data from the provided file objects smoking_datafile
6          and cancer_datafile. Returns a list of the data read from each
7          in a tuple of the form ([smoking data], [cancer data]).
8      """
9
10     # init
11     smoking_data = []
12     cancer_data = []
13     empty_str = ''
14
15     # read past file headers
16     smoking_datafile.readline()
17     cancer_datafile.readline()
18
19     # read data files
20     eof = False
21
22     while not eof:
23
24         # read line of data from each file
25         s_line = smoking_datafile.readline()
26         c_line = cancer_datafile.readline()
27
28         # check if at end-of-file of both files
29         if s_line == empty_str and c_line == empty_str:
30             eof = True
31
32         # check if end of smoking data file only
33         elif s_line == empty_str:
34             raise IOError('Unexpected end-of-file: smoking data file')
35
36         # check if at end of cancer data file only
37         elif c_line == empty_str:
38             raise IOError('Unexpected end-of-file: cancer data file')
39
40         # append line of data to each list
41         else:
42             smoking_data.append(s_line.strip().split(','))
43             cancer_data.append(c_line.strip().split(','))
44
45     # return list of data from each file
46     return (smoking_data, cancer_data)
```

FIGURE 8-29 Implementation of Function readFiles

Function readFiles parameters are passed the file objects of the smoking-related and cancer-related data files (**line 3**). It reads the data from each file and returns it in two lists. On **lines 11–12**, variables smoking_data and cancer_data are

initialized to an empty list, and on **line 13**variable empty_str is assigned to an empty string. The first line is then read from each file (**line 16–17**), which contain file headers (descriptions of the file contents),

State, Percent Cigarette Smokers Cases Lung Cancer per 100,000
(header of smoking-related data file) (header of cancer-related data file)

On **line 20**, variable eof is set to False. The value of this variable indicates whether the end- of-file has been reached for the files. The while loop at **line 22** continues to iterate as long as eof is False. A line from each file is then read (**line 25–26**) and stored in variables s_line (from the smoking-related data file) and c_line (from the cancer-related data file). The following if statement (**line 29**) checks if each of these variables is equal to the empty string. If true, then the end of file of each has been reached, and thus variable eof is assigned to True (**line 30**) If not, then a check is made (on **line 33** and **line 37**) if either s_line or c_line is equal to the empty string. If so, then the end of one file has been reached before the other. In that case, an IOError exception is raised (on **line 34** or **38**) with one of the following error message strings,

'Unexpected end-of-file: smoking data file' or
'Unexpected end-of-file: cancer data file'

If the strings in variables s_line and c_line are each found non-empty, then on **lines 42–43** they are appended to lists smoking_data and cancer_data, respectively. Since each line returned by readLine contains a final newline character, method strip is used to strip off the last character of each line. Also, because each line of the files is of the form

state_name, data_value
the split string method is used to split each string into two values (state_name and data_ value) using the comma as the separation character. This returns the two values in a list of the form [*state_name, data_value*]
Finally, when all the lines of the files have been read, the data in lists smoking_data and cancer_data are returned as a tuple.
As we did for function openFile, we develop a simple test driver program for testing function readFiles. The test driver (in Figure 8-30) first prompts the user for each of the file names

```
# TEST DRIVER for Function readFiles of Smoking/Cancer Correlation Program

from module_readFiles import *

try:
    file_name = input('Enter file name with smoking data: ')
    smoking_datafile = open(file_name, 'r')

    file_name = input('Enter file name with lung cancer data: ')
    cancer_datafile = open(file_name, 'r')

    smoking_data, cancer_data = readFiles(smoking_datafile, cancer_datafile)

    print('\nList of smoking data read')
    print(smoking_data)
    print('\nList of cancer data read')
    print(cancer_data)
except IOError as e:
    print(str(e))
```

FIGURE 8-30 Test Driver for Function readFiles

without doing any I/O error checking. This is because in the complete program version, function openFile will handle this, and therefore we can alleviate the test driver of that. Thus, the files are opened (assuming that correct file name are given) and the contents read and displayed on the screen.

Function readFiles raises an exception if the two data files do not have the same number of lines. Therefore, we test for that condition by purposely entering the names of two data files in which one file is shorter than the other. The results of this test case is given in Figure 8-31.

```
Enter file name with smoking data: CDC_Cigarette_Smoking_Data.csv
Enter file name with lung cancer data: CDC_Lung_Cancer_Data_Short.csv
Unexpected end-of-file: cancer data file
>>>
```

FIGURE 8-31 Unexpected End-of-File Test Case for Function readFiles

We next enter the names of equal length data files to test if the lists returned by readFiles contain the proper data. The results are given in Figure 8-32.

```
Enter file name with smoking data: CDC_Cigarette_Smoking_Data.csv
Enter file name with lung cancer data: CDC_Lung_Cancer_Data.csv

List of smoking data read
[['Alabama', '23.3'], ['Alaska', '24.2'], ['Arkansas', '23.7'],
['California', '14.9'], ['Colorado', '17.9'], ['Connecticut', '17'],
['Delaware', '21.7'], ['Florida', '21'], ['Georgia', '20'], ['Hawaii',
'17.5'], ['Idaho', '16.8'], ['Illinois', '20.5'], ['Indiana', '24.1'],
['Iowa', '21.5'], ['Kansas', '20'], ['Kentucky', '28.6'], ['Louisiana',
'23.4'], ['Maine', '20.9'], ['Maryland', '17.8'], ['Massachusetts', '17.8'],
['Michigan', '22.4'], ['Minnesota', '18.3'], ['Mississippi', '25.1'],
['Missouri', '23.3'], ['Montana', '19'], ['Nebraska', '18.6'], ['Nevada',
'22.2'], ['New Hampshire', '18.7'], ['New Jersey', '18.1'], ['New Mexico',
'20.2'], ['New York', '18.3'], ['North Carolina', '22.1'], ['North Dakota',
'19.6'], ['Ohio', '22.5'], ['Oklahoma', '25.1'], ['Oregon', '18.5'],
['Pennsylvania', '21.5'], ['Rhode Island', '19.3'], ['South Carolina',
'22.3'], ['South Dakota', '20.4'], ['Tennessee', '22.6'], ['Texas', '18.1'],
['Utah', '9.8'], ['Vermont', '18'], ['Virginia', '19.3'], ['Washington',
'17.1'], ['West Virginia', '25.7'], ['Wyoming', '21.6']]

List of cancer data read
[['Alabama', '75.1'], ['Alaska', '69.8'], ['Arkansas', '77.7'],
['California', '51.6'], ['Colorado', '48.4'], ['Connecticut', '68.3'],
['Delaware', '83.5'], ['Florida', '68.7'], ['Georgia', '71'], ['Hawaii',
'51.7'], ['Idaho', '54'], ['Illinois', '71.4'], ['Indiana', '77.2'],
['Iowa', '67.1'], ['Kansas', '68.4'], ['Kentucky', '97.2'], ['Louisiana',
'77.5'], ['Maine', '80.1'], ['Maryland', '62.9'], ['Massachusetts', '64.5'],
['Michigan', '72'], ['Minnesota', '55.8'], ['Mississippi', '77.1'],
['Missouri', '77.9'], ['Montana', '60.8'], ['Nebraska', '61.3'], ['Nevada',
'73.3'], ['New Hampshire', '66.8'], ['New Jersey', '64.5'], ['New Mexico',
'43.9'], ['New York', '63'], ['North Carolina', '75.3'], ['North Dakota',
'53'], ['Ohio', '72.8'], ['Oklahoma', '80.8'], ['Oregon', '64.4'],
['Pennsylvania', '70'], ['Rhode Island', '69.5'], ['South Carolina',
'70.7'], ['South Dakota', '59.2'], ['Tennessee', '82.2'], ['Texas', '62'],
['Utah', '28'], ['Vermont', '79.2'], ['Virginia', '65.8'], ['Washington',
'65.2'], ['West Virginia', '91'], ['Wyoming', '48.9']]
>>>
```

FIGURE 8-32 Proper Data Files Test Case for Function readFiles

Looking at the output, it appears that the data from both files has been properly read and properly constructed in each list.

Development and Unit Testing of Function **calculateCorrelation** The final function of the program to unit test is function calculateCorrelation. An implementation of this function is given in Figure 8-33.

```
import math

def calculateCorrelation(smoking_data, cancer_data):

    """ Calculates and returns the correlation value for the data
        provided in lists smoking_data and cancer_data
    """

    # init
    sum_smoking_vals = sum_cancer_vals = 0
    sum_smoking_sqrd = sum_cancer_sqrd = 0
    sum_products = 0

    # calculate intermediate correlation values
    num_values = len(smoking_data)

    for k in range(0,num_values):
        sum_smoking_vals = sum_smoking_vals + float(smoking_data[k][1])
        sum_cancer_vals = sum_cancer_vals + float(cancer_data[k][1])

        sum_smoking_sqrd = sum_smoking_sqrd +  \
                           float(smoking_data[k][1]) ** 2
        sum_cancer_sqrd = sum_cancer_sqrd +  \
                           float(cancer_data[k][1]) ** 2

        sum_products = sum_products + float(smoking_data[k][1]) *  \
                       float(cancer_data[k][1])

    # calculate and display correlation value
    numer = (num_values * sum_products) - \
            (sum_smoking_vals * sum_cancer_vals)

    denom = math.sqrt(abs( \
        ((num_values * sum_smoking_sqrd) - (sum_smoking_vals ** 2)) * \
        ((num_values * sum_cancer_sqrd) - (sum_cancer_vals ** 2)) \
        ))

    return numer / denom
```

FIGURE 8-33 Implementation of Function calculateCorrelation

The test driver for this function is given in Figure 8-34. For testing the correctness of the calculation performed by function calculateCorrelation, we do not use the data from the data files. Instead, we use data for which we know what the calculated correlation value is. Therefore, three sets of data are hard-coded in the test driver—values in which there is a perfect correlation, another set of values with no correlation, and a final set of values in which there is a perfect negative correlation.

```
1  # TEST DRIVER for Function calculateCorrelation of Smoking/Cancer
2  # Correlation Program
3
4  from module_calculateCorrelation import *
5
6  # calculate perfect positive correlation
7  print('Calculating perfect positive correlation ...')
8  smoking_data = [['A', 10], ['B', 20], ['C', 30], ['D', 40]]
9  cancer_data = [['A', 100], ['B', 200], ['C', 300], ['D', 400]]
10
11 print('Correlation value:', calculateCorrelation(smoking_data, cancer_data))
12
13 # calculate zero correlation
14 print('\nCalculating zero correlation ...')
15 smoking_data = [['A', 10], ['B', 20], ['C', 30], ['D', 40]]
16 cancer_data = [['A', 100], ['B', 0], ['C', 300], ['D', 0]]
17
18 print('Correlation value:', calculateCorrelation(smoking_data,
19 cancer_data))
20
21 # calculate perfect negative correlation
22 print('\nCalculating perfect negative correlation ...')
23 smoking_data = [['A', 10], ['B', 20], ['C', 30], ['D', 40]]
24 cancer_data = [['A', 400], ['B', 300], ['C', 200], ['D', 100]]
25
26 print('Correlation value:', calculateCorrelation(smoking_data, cancer_data))
```

FIGURE 8-34 Test Driver for Function calculateCorrelation

In the first set of lists, the values in list smoking_data (**line 8**) range from 10 to 40 in increments of ten. The values in list cancer_data (**line 9**) range from 100 to 400, in increments of 100. Since the rise in value in each list in proportion to the other, there is a perfect correlation; thus there is, a correlation value of 1. (Note that we simply use the letters ‘A’, ‘B’, etc., for the names of the states since this information is irrelevant for this testing.)

In the second set of lists, half of the values in list cancer_data (**line 15**) rise as the values in list smoking_data rise (**line 16**), and the other half of values decrease as the values in smoking_data rise (each by proportional amounts). Therefore, there is no correlation between these two lists of values; thus, there is a correlation value of 0.

Finally, in the third set of lists, the values in list smoking_data (**line 23**) increase as the values in list cancer_data decrease (**line 24**), each by proportional amounts. Therefore, there is a perfect negative correlation; thus there is a correlation value of 21. The results of the execution of the test driver are given in

Figure 8-35.

At this point, each of the three functions has been successfully unit tested. What follows next is to perform integrating testing by incorporating each of the functions into the main module and testing the program as a whole.

Integration Testing of the Smoking/Cancer Correlation Program

For the purposes of unit testing, we developed each function in its own module. Now that we are ready to perform integration testing, we put all function definitions into one program file with the main module. The complete program is given in Figure 8-36.

```
Calculating perfect positive correlation ...
Correlation value: 1.0

Calculating zero correlation ...
Correlation value: 0.0

Calculating perfect negative correlation ...
Correlation value: -1.0
>>>
```

FIGURE 8-35 Test Results for Function calculateCorrelation

```
Cigarette Use / Lung Cancer Correlation Program
import math

def openFiles():

    """ Prompts the user for the file names to open, opens the files,
        and returns the file objects for each in a tuple of the form
        (smoking_datafile, cancer_datafile).

        Raises an IOError exception if the files are not successfully
        opened after four attempts.
    """

    # init
    smoking_datafile_opened = False
    cancer_datafile_opened = False
    num_attempts = 4

    # prompt for file names and attempt to open files
    while ((not smoking_datafile_opened) or \
           (not cancer_datafile_opened)) \
            and (num_attempts > 0):
        try:
            if not smoking_datafile_opened:
                file_name = input('Enter smoking data file name: ')
                smoking_datafile = open(file_name, 'r')
                smoking_datafile_opened = True

            if not cancer_datafile_opened:
                file_name = input('Enter lung cancer data file name: ')
                cancer_datafile = open(file_name, 'r')
                cancer_datafile_opened = True
        except IOError:
            print('File not found:',file_name + '.','Please reenter\n')
            num_attempts = num_attempts - 1

    # if one of more file not opened, raise IOError exception
    if not smoking_datafile_opened or not cancer_datafile_opened:
        raise IOError('Too many attempts of reading input files')

    # return file objects if successfully opened
    else:
        return (smoking_datafile, cancer_datafile)

```

```
45 def readFiles(smoking_datafile, cancer_datafile):
46
47     """ Reads the data from the provided file objects smoking_datafile
48         and cancer_datafile. Returns a list of the data read from each
49         in a tuple of the form (smoking_datafile, cancer_datafile).
50     """
51
52     # init
53     smoking_data = []
54     cancer_data = []
55     empty_str = ''
56
57     # read past file headers
58     smoking_datafile.readline()
59     cancer_datafile.readline()
60
61     # read data files
62     eof = False
63
64     while not eof:
65
66         # read line of data from each file
67         s_line = smoking_datafile.readline()
68         c_line = cancer_datafile.readline()
69
70         # check if at end-of-file of both files
71         if s_line == empty_str and c_line == empty_str:
72             eof = True
73
74         # check if end of smoking data file only
75         elif s_line == empty_str:
76             raise IOError('Unexpected end-of-file: smoking data file')
77
78         # check if at end of cancer data file only
79         elif c_line == empty_str:
80             raise IOError('Unexpected end-of-file: cancer data file')
81
82         # append line of data to each list
83         else:
84             smoking_data.append(s_line.strip().split(','))
85             cancer_data.append(c_line.strip().split(','))
86
87     # return list of data from each file
88     return (smoking_data, cancer_data)
89
90 def calculateCorrelation(smoking_data, cancer_data):
91
92     """ Calculates and returns the correlation value for the data
93         provided in lists smoking_data and cancer_data
94     """
95
96     # init
97     sum_smoking_vals = sum_cancer_vals = 0
98     sum_smoking_sqrd = sum_cancer_sqrd = 0
99     sum_products = 0
100
```

```
    # calculate intermediate correlation values
    num_values = len(smoking_data)

    for k in range(0,num_values):

        sum_smoking_vals = sum_smoking_vals + float(smoking_data[k][1])
        sum_cancer_vals = sum_cancer_vals + float(cancer_data[k][1])

        sum_smoking_sqrd = sum_smoking_sqrd +  \
                           float(smoking_data[k][1]) ** 2
        sum_cancer_sqrd = sum_cancer_sqrd +  \
                           float(cancer_data[k][1]) ** 2

        sum_products = sum_products + float(smoking_data[k][1]) *  \
                       float(cancer_data[k][1])

    # calculate and display correlation value
    numer = (num_values * sum_products) - \
            (sum_smoking_vals * sum_cancer_vals)

    denom = math.sqrt(abs( \
        ((num_values * sum_smoking_sqrd) - (sum_smoking_vals ** 2)) * \
        ((num_values * sum_cancer_sqrd) - (sum_cancer_vals ** 2)) \
        ))

    return numer / denom

# ---- main

# program greeting
print('This program will determine the correlation (-1 to 1) between')
print('data on cigarette smoking and incidences of lung cancer\n')

try:
    # open data files
    smoking_datafile, cancer_datafile = openFiles()

    # read data
    smoking_data, cancer_data = readFiles(smoking_datafile, cancer_datafile)

    # calculate correlation value
    correlation = calculateCorrelation(smoking_data, cancer_data)

    # display correlation value
    print('r_value = ', correlation)
except IOError as e:
    print(str(e))
    print('Program terminated ...')
```

FIGURE 8-36 Cigarette Use/Lung Cancer Correlation Program

We test the program with the same data used for the unit testing of function calculateCorrelation. In this case, however, the data is contained in files to be opened and read by the program. The files and their contents are given below.

```
Smoking Test Data 1
A, 10
B, 20
C, 30
D, 40
```

```
Cancer Test Data 1
A, 100
B, 200
C, 300
D, 400
```

```
Smoking Test Data 2
A, 10
B, 20
C, 30
D, 40
```

```
Cancer Test Data 2
A, 100
B, 0
C, 300
D, 0
```

```
Smoking Test Data 3
A, 10
B, 20
C, 30
D, 40
```

```
Cancer Test Data 3
A, 400
B, 300
C, 200
D, 100
```

The three test results of the program for the data sets above is given below in Figure 8-37. The results are as expected. We get the same results as in the unit testing of
calculateCorrelation. Therefore, all three functions are properly integrated into the program. We next run the program on the actual data files CDC_Cigaratte_Smoking_Data and
CDC_Lung_Cancer_Data.

```
This program will determine the correlation (-1 to 1) between
data on cigarette smoking and incidences of lung cancer

Enter smoking data file name: Smoking_Test_Data_1.csv
Enter lung cancer data file name: Cancer_Test_Data_1.csv
r_value =  1.0
>>> ================================ RESTART ================================
>>>
This program will determine the correlation (-1 to 1) between
data on cigarette smoking and incidences of lung cancer

Enter smoking data file name: Smoking_Test_Data_2.csv
Enter lung cancer data file name: Cancer_Test_Data_2.csv
r_value =  0.0
>>> ================================ RESTART ================================
>>>
This program will determine the correlation (-1 to 1) between
data on cigarette smoking and incidences of lung cancer

Enter smoking data file name: Smoking_Test_Data_3.csv
Enter lung cancer data file name: Cancer_Test_Data_3.csv
r_value =  -1.0
>>>
```

FIGURE 8-37 Integration Testing Results for Cigarette Use/Lung Cancer Correlation Program

Chapter Summary 331

8.5.5 Determining the Correlation Between Smoking and Lung Cancer The output from the execution of this program on the actual data is given in Figure 8-38.

```
This program will calculate the correlation value (-1 to 1) between
provided data on cigarette smoking and incidences of lung cancer

Enter the names of the two data files, including file extension
(files must contain same number of lines of data)

Enter name of cigarette smoking file: CDC Cigarette Smoking Data.csv
Enter name lung cancer file: CDC Lung Cancer Data.csv
r_value =  0.786159246886
```

FIGURE 8-38 Output of the Cigarette Use/Lung Cancer Correlation Program

In the output, we see a correlation value of approximately 0.79. That indicates a very strong correlation, depicted in Figure 8-39.

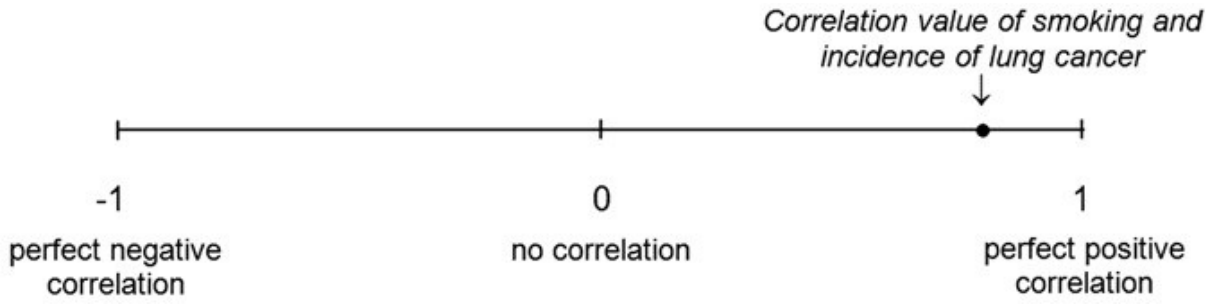


FIGURE 8-39 Depiction of Results on a Correlation Scale

So, can we conclude from this result alone that smoking *causes* lung cancer? No. Causation cannot be easily determined even with a perfect correlation. For example, there is a strong correlation between the amount of ice cream sold and the number of drownings that occur in the United States. Does that mean that eating ice cream raises the likelihood of drowning? Common sense tells us no, that the correlation results from them both being summertime activities.

To interpret our results, we have to decide that either (a) smoking causes lung cancer; (b) lung cancer causes the urge to smoke; (c) there exists a third factor simultaneously causing the desire for smoking and lung cancer; or (d) there is no causal relationship. Since a strong correlation is not enough to argue that causation exists, other evidence needs to be considered, such as the lung tissue of those who have died from lung cancer. This, in fact, is what medical research has investigated. Based on the findings, along with the demonstrated correlation, tells us that yes, smoking does cause lung cancer.

CHAPTER SUMMARY General Topics

Text Files vs. Binary Files Opening and Closing Files Reading and Writing Text Files String Processing
Exception Handling
Comma Separated Data Files Python-Specific Programming Topics

Built-in Functions open and close in Python File Object Methods readline and write in

Python
String Methods in Python
Catching and Handling Exceptions in Python Standard Exceptions in Python

CHAPTER EXERCISES

Section 8.1

1. Explain the difference in how numbers are represented in a text file versus how they are represented in a binary file containing numerical values.
2. Explain why binary files do not contain newline characters.

Section 8.2

3. Give an instruction in Python that opens a file named 'datafile.txt' for reading and assigns identifier input_file to the file object created.
4. Give an instruction in Python that opens a file named 'datafile2.txt' for writing and assigns identifier output_file to the file object created.
5. Assume that input_file is a file object for a text file open for reading, and output_file is a file object for a text file open for writing. Explain the contents of the output file after the following code is executed,

empty_str 5 "
line 5 input_file.readline()

while line ! 5 empty_str:
output_file.write(line 1 '\n')
line 5 input_file.readline()

Section 8.3
6. Give a for loop that counts all the letter characters in string line.
7. For variable month which contains the full name of any given month, give an instruction to display just the first three letters of the month.
8. Give an instruction that displays True if the letter ‘r’ appears in a variable named month, otherwise displays False.
9. Give an instruction for determining how many times the letter ‘r’ appears in a variable named month. **10.** For variables first_name and last_name, give an instruction that displays the person’s name in the form *last name, first name*.
11. Give an instruction that determines if a variable named ss_num contains any non-digit characters, other than a dash.
12. Give an instruction that determines the index of the ‘@’ character in an email address in variable email_ addr.
13. For variable date containing a date of the form 12/14/2012, write a function that produces the same date, but with all slashes characters replaced with dashes.
14. For a variable named err_mesg that contains error messages in the form **

error message **, give an instruction that produces a string containing the error message without the leading and trailing asterisks and blank characters.

Section 8.4
15. Identify the error in the following code,
input_file_opened 5 False while not input_file_opened:

Program Modification Problems 333

try:
file_name 5 input('Enter file name: ') input_file 5 open(file_name, 'r')

except:
print('Input file not found – please reenter')
PYTHON PROGRAMMING EXERCISES
P1. Write a Python function called reduceWhitespace that is given a line read from a text file and returns the line with all extra whitespace characters between words removed,
'This line has extra space characters' → 'This line has extra space characters' **P2.** Write a Python function named extractTemp that is given a line read from a text file and displays the one number (integer) found in the string,
'The high today will be 75 degrees' → 75
P3. Write a Python function named checkQuotes that is given a line read from a text file and returns True if each quote characters in the line has a matching quote (of the same type), otherwise returns False. 'Today's high temperature will be 75 degrees' →False

P4. Write a Python function named countAllLetters that is given a line read from a text file and returns a list containing every letter in the line and the number of times that each letter appears (with upper/lower case letters counted together),

'This is a short line' →[('t', 2), ('h', 2), ('i', 3), ('s', 3), ('a', 1), ('o', 1), ('r', 1), ('l', 1), ('n', 1), ('e', 1)] **P5.** Write a Python function named interleaveChars that is given *two* lines read from a text file, and returns a single string containing the characters of each string interleaved,

'Hello', 'Goodbye' →'HGeololdobye'
P6. Write a program segment that opens and reads a text file and displays how many lines of text are in the file.

P7. Write a program segment that reads a text file named original_text, and writes every other line, starting with the first line, to a new file named half_text.
P8. Write a program segment that reads a text file named original_text, and displays how many times the letter ‘e’ occurs.

PROGRAM MODIFICATION PROBLEMS

M1. Sparse Text Program: User-Selected Letter Removed
Modify the Sparse Text program in section 8.3.4 so that instead of the letter ‘e’ being removed, the user is prompted for the letter to remove.

M2. Sparse Text Program: Random Removal of Letters
Modify the Sparse Text program in section 8.3.4 so that instead of a particular letter removed, a percentage of the letters are randomly removed based on a percentage entered by the user.

M3. Word Frequency Count Program: Display of Scanned Lines
Modify the Word Frequency Count program in section 8.4.6 so that the text lines being scanned are at the same time displayed on the screen.

M4. Word Frequency Count Program: Counting of a Set of Words
Modify the Word Frequency Count Program so that the user can enter any number of words to be counted within a given text file.

M5. Word Frequency Count Program: Counting of All Words
Modify the Word Frequency Count program so that all the words in a given text file are counted.
M6. Word Frequency Count Program: Outputting Results to a File
Modify the Word Frequency Count program so that the counts of all words in a given text file are output to a file with the same name as the file read, but with the file extension '.wc' (for ‘word count’).
M7. Lung Cancer Correlation Program: Air Pollution and Lung Cancer
Modify the cigarettes and lung cancer correlation program in the Computational Problem Solving section of the chapter to correlate lung cancer with air pollution instead. Use the data from the following ranking of states from highest to lowest amounts of air pollution given below.

Rank	State	Added cancer risk (per 1,000,000)
1.	NEW YORK	1900
2.	NEW JERSEY	1400
3.	DISTRICT OF COLUMBIA	1100
4.	CALIFORNIA	890
5.	MASSACHUSETTS	890
6.	MARYLAND	870
7.	DELAWARE	860
8.	PENNSYLVANIA	860
9.	CONNECTICUT	850
10.	ILLINOIS	800
11.	OHIO	730
12.	INDIANA	720
13.	RHODE ISLAND	670
14.	MINNESOTA	660
15.	GEORGIA	650
16.	MICHIGAN	640
17.	VIRGINIA	620
18.	LOUISIANA	590
19.	WEST VIRGINIA	560
20.	TEXAS	550
21.	WISCONSIN	540
22.	KENTUCKY	540
23.	TENNESSEE	520
24.	OREGON	520
25.	FLORIDA	510
26.	UTAH	500
27.	WASHINGTON	490
28.	NORTH CAROLINA	480
29.	NEW HAMPSHIRE	470
30.	MISSOURI	460
31.	ARIZONA	440
32.	COLORADO	440
33.	ALABAMA	400
34.	SOUTH CAROLINA	390
35.	NEBRASKA	390
36.	KANSAS	360
37.	IOWA	340
38.	NEVADA	340
39.	ARKANSAS	330
40.	OKLAHOMA	330
41.	MISSISSIPPI	320
42.	VERMONT	270
43.	IDAHO	260
44.	MAINE	240
45.	NEW MEXICO	230
46.	NORTH DAKOTA	200
47.	SOUTH DAKOTA	190
48.	MONTANA	180
49.	WYOMING	140

PROGRAM DEVELOPMENT PROBLEMS

D1. Sentence, Word, and Character Count Program
Develop and test a Python program that reads in any given text file and displays the number of lines, words, and total number of characters there are in the file, including spaces and special characters, but not the newline character, '\n'.

D2. Variation on a Sparsity Program
Develop and test a program that reads the text in a given file, and produces a new file in which the *first occurrence only* of the vowel in each word is removed, unless the removal would leave an empty word (for example, for the word “I”). Consider how readable the results are for various sample text.

D3. Message Encryption/Decryption Program
Develop and test a Python program that reads messages contained in a text file, and encodes the messages saved in a new file. For encoding messages, a simple substitution key should be used as shown below,

A	F
B	G
C	L
D	R
E	P
. .	. .

Each letter in the left column is substituted with the corresponding letter in the right column when encoding. Thus, to decode, the letters are substituted the opposite way. Unencrypted message files will be simple text files with file extension .txt. Encrypted message files will have the same file name, but with file extension .enc. For each message encoded, a new substitution key should be randomly generated and saved in a file with the extension '.key'. Your program should also be able to decrypt messages given a specific encoded message and the corresponding key.

D4. Morse Code Encryption/Decryption Program
Develop and test a Python program that allows a user to open a text file containing a simple message using only the (uppercase) letters A . . . Z, and saves a Morse code version of the message, that is, containing only the characters dash (“-”), dot (“.”). In the encoded version, put the encoding of each character on its own line in the text file. Use a blank line to indicate the end of a word, and two blank lines to indicate the end of a sentence. Your program should be able to both convert an English message file into Morse code, and a Morse code file into English. The Morse code for each letter is given below.

A	•–	N	–•
B	–•••	O	–––
C	–•–•	P	•––•
D	–••	Q	––•–
E	•	R	•–•
F	••–•	S	•••
G	––•	T	–
H	••••	U	••–
I	••	V	•••–
J	•–––	W	•––
K	–•–	X	–••–
L	•–••	Y	–•––
M	––	Z	––••

D5. Universal Product Code Check Digit Verification Program
A *check digit* is a digit added to a string of digits that is derived from other digits in the string. Check digits provide a form of redundancy of information, used for determining if any of the digits in the string are incorrect or misread.

The Universal Product Code on almost all purchase items utilizes a bar code to allow for the scanning of items. Below the bar code is the sequence of digits that the bar code encodes, as illustrated below.
03 6 0 0 0 29 1 4 5 2
The last digit of the product code (2) is a check digit computed as follows,
1. Add up all digits in the odd numbered positions (first, third, fifth, etc., starting with the leftmost digit) excluding the last check digit, and multiply the result by 3,

0 1 6 1 0 1 2 1 1 1 5 5 14, 14 * 3 5 42
2. Add up all digits in the even numbered positions (second, fourth, etc.) excluding the last check digit, 3 1 0 1 0 1 9 1 4 5 16
3. Take the sum of the two previous results mod 10,
(42 1 16) mod 10 5 58 mod 10 5 8
4. Subtract the result from 10 to get the checksum digit.
10] 8 5 **2**
Develop and test a Python program that verifies the check digit of Universal Product Codes.

Dictionaries and Sets CHAPTER 9

Since Chapter 4, we have been using linear (sequential) data structures—lists, tuples, and strings— in which elements are accessed by their index value (that is, by location). There also exist data structures in which elements are not accessed by location, but rather by an associated key value, called dictionaries in Python. We look at both dictionaries and sets in Python in this chapter.

OBJECTIVES
After reading this chapter and completing the exercises, you will be able to: ♦ Explain the concept of an associative data structure ♦ Define and use dictionaries in Python
♦ Define and use sets in Python
♦ Write Python programs using dictionaries and sets
♦ Perform unit testing using test stubs (and test drivers)

CHAPTER CONTENTS
Motivation
Fundamental Concepts
9.1 Dictionary Type in Python
9.2 Set Data Type
Computational Problem Solving
9.3 A Food Co-op’s Worker Scheduling Simulation

MOTIVATION

The vast amounts of data being stored today could not be effectively utilized without being organized in some way. Databases do not simply store large

amounts of data. Rather, they provide a complete database management system (DBMS) used in conjunction with stored data to allow the data to be associated and accessed in various ways.

Database management systems provide access to data without needing to know how the data is physically structured, only how it is *logically* organized. Access is provided by a *query language* that allows arbitrarily complex queries to be made of the data. For example, one can construct a query that returns the set of students who have a perfect 4.0 GPA and are graduating at the end of the current semester. Or, one could construct a more complex query that returns the set of students that have a GPA of 3.6 or above, with senior status, female, a computer science major, and has taken at least twelve credits of biology or eight credits of biology and four credits of chemistry.

Some of the concerns of database management systems include: *integrity of data* —that is, data that is accessible and properly organized; *security* in the control of who has what level of access; and management of *concurrency* issues when multiple users are accessing the same data. Data mining (Figure 9-1) is a relatively new field in which database management systems are used in conjunction with methods from statistics and artificial intelligence to extract patterns from very large amounts of data.

Areas	Data Mining Application
Business	Customer Relationship Management (e.g., to find, attract, retain customers)
Genetics	DNA Analysis (e.g., to understand individual variations in human DNA in relation to disease susceptibility)
National Security	Behavioral Patterns (e.g., to identify possible terrorist activities)
Information Retrieval	Music Information Retrieval (to discover musical patterns used in music retrieval)

FIGURE 9-1 Data Mining Applications

FUNDAMENTAL CONCEPTS

9.1 Dictionary Type in Python

In this section we introduce the notion of an *associative data structure*. Elements of indexed linear data structures, such as lists, are ordered—the first element (at index 0), second element (at index 1), and so forth. In contrast, the elements of an associative data structure are unordered, instead accessed by an associated key value. In Python, an associative data structure is provided by the *dictionary type*. We look at the use of dictionaries in Python next.

9.1.1 What Is a Dictionary?

A **dictionary** is a mutable, associative data structure of variable length. The syntax for declaring dictionaries in Python is given below.

daily_temps 5 {'sun': 68.8, 'mon': 70.2, 'tue': 67.2, 'wed': 71.8, 'thur': 73.2, 'fri': 75.6, 'sat': 74.0}

Dictionary daily_temps stores the average temperature for each day of the week, as we did earlier in Chapter 4 using a list. However, in this case, each temperature has associated with it a unique key value ('sun', 'mon', etc.). Strings are often used as key values. The syntax for accessing an element of a dictionary is the same as for accessing elements of sequence types, except that a key value is used within the square brackets instead of an index value: daily_temps['sun']. A comparison of accessing indexed data structures vs. associative data structures is given in Figure 9-2.

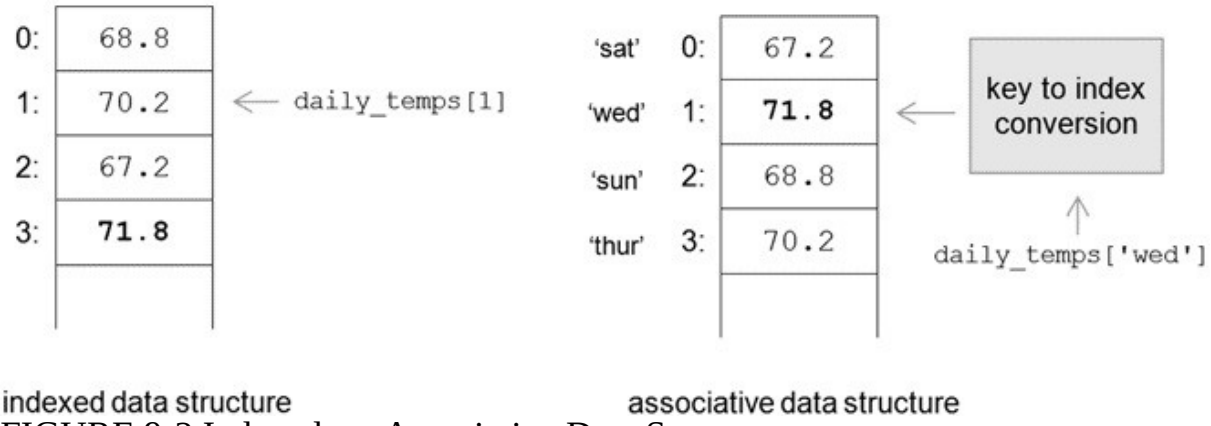


FIGURE 9-2 Indexed vs. Associative Data Structure

On the left is an indexed data structure, and on the right an associative data structure. Although the elements of the associative data structure are physically ordered, the ordering is irrelevant to the way that the structure is utilized. *The location that an element is stored in and retrieved from within an associative data structure depends only on its key value*, thus there is no logical first element, second element, and so forth. The specific location that a value is stored is determined by a particular method of converting key values into index values called *hashing*. Example use of dictionary daily_temps is given below.

if daily_temps['sun'] . daily_temps['sat']:
print 'Sunday was the warmer weekend day'
else
if daily_temps['sun'] , daily_temps['sat']: print 'Saturday was the warmer weekend day' else:
print 'Saturday and Sunday were equally warm'

Although strings are often used as key values, any immutable type may be used as well, such as a tuple (shown in Figure 9-3). In this case, the temperature for a specific date is retrieved by,
temps[('Apr', 14, 2001)] → 74.6

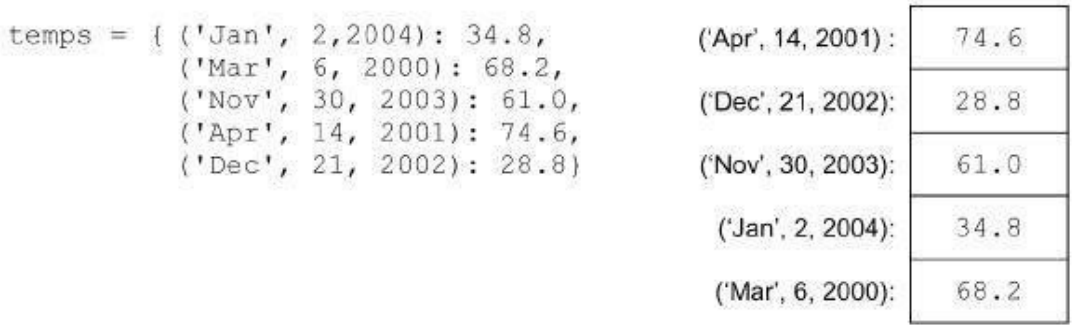


FIGURE 9-3 Associative Data Structure Using Tuple Key Values

Note that this key contains both string and integer values. To give an example of when an associative array may be of benefit over an indexed type, we give two versions of a program that displays the recorded average temperature for any day of the week—one using a list, and the other using a dictionary. We first give the list version of the program in Figure 9-4, which allows a user to enter a day of the week and have the average temperature for that day displayed.

```
# Temperature Display Program (List Version)

daily_temps = [68.8, 70.2, 67.2, 71.8, 73.2, 75.6, 74.0]

print('This program will display the average temperature for a given day')
terminate = False

while not terminate:

    day = input("Enter 'sun', 'mon', 'tue', 'wed', 'thur', 'fri', or 'sat': ")

    if day == 'sun':
        dayname = 'Sunday'
        temp = daily_temps[0]
    elif day == 'mon':
        dayname = 'Monday'
        temp = daily_temps[1]
    elif day == 'tue':
        dayname = 'Tuesday'
        temp = daily_temps[2]
    elif day == 'wed':
        dayname = 'Wednesday'
        temp = daily_temps[3]
    elif day == 'thur':
        dayname = 'Thursday'
        temp = daily_temps[4]
    elif day == 'fri':
        dayname = 'Friday'
        temp = daily_temps[5]
    elif day == 'sat':
        dayname = 'Saturday'
        temp = daily_temps[6]

    print('The average temperature for', dayname, 'was', temp, 'degrees\n')

    response = input('Continue with another day? (y/n): ')
    if response == 'n':
        terminate = True
```

FIGURE 9-4 Temperature Display Program (Indexed Array Version)

The program prompts the user for the day of the week (as 'sun', 'mon', 'tue', etc.) to display the average temperature for. The average temperatures are stored in

list daily_ temps. Once read, an if statement (with elif headers) is used to set variable dayname to the full name of the day entered, as well as retrieve the corresponding temperature from list daily_temps. The result is then displayed to the user. An example execution of this program is given below.

This program will display the average temperature for a given day Enter 'sun', 'mon', 'tue', 'wed', 'thur', 'fri', or 'sat': wed The average temperature for Wednesday was 71.8 degrees ...

The program in Figure 9-5 provides the identical functionality, but instead using an associative array instead of a list for storing the average daily temperatures.

```
# Temperature Display Program (Dictionary Version)

daily_temps = {'sun': 68.8, 'mon': 70.2, 'tue': 67.2, 'wed': 71.8,
               'thur': 73.2, 'fri': 75.6, 'sat': 74.0}

daynames = {'sun': 'Sunday', 'mon': 'Monday', 'tue': 'Tuesday',
            'wed': 'Wednesday', 'thur': 'Thursday', 'fri': 'Friday',
            'sat': 'Saturday'}

print('This program will display the average temperature for a given day')
day = input("Enter 'sun', 'mon', 'tue', 'wed', 'thur', 'fri', or 'sat': ")
print('The average temperature for', daynames[day], 'was',
       daily_temps[day], 'degrees')
```

FIGURE 9-5 Temperature Display Program (Dictionary Version)

This version of the program is much more concise and elegant than the previous one. Rather than using a series of conditions in an if-elif statement for checking the day of the week entered by the user, the entered day is directly used for retrieving the corresponding average temperature from dictionary daily_temps. A second associative array is also used in the program for storing the corresponding full day names, accessed by the same set of key values.

In the previous example, the values of the dictionaries are “hard-coded” into the program. Dictionaries, however, may also be created and modified dynamically (that is, during program execution). There are cases where this capability is needed. For example, if a given dictionary contains thousands of elements, too many to hard-code into a program, then the key/value pairs can be read from a file and the dictionary “built” at runtime. Or, it may be that some of the values to be stored are not yet known, and therefore, the dictionary needs to be expanded and updated at execution time (for example, when storing user-selected passwords). The operations related to the dynamic creation and updating of

dictionaries are shown in Figure 9-6.

Operation	Results
`dict()`	Creates a new, empty dictionary

`dict(s)`	Creates a new dictionary with key values and their associated values from sequence `s`, for example,
	`fruit_prices = dict(fruit_data)`
	where `fruit_data` is (possibly read from a file):
	`[['apples', .66],…,['bananas', .49]]`

`len(d)`	Length (num of key/value pairs) of dictionary `d`.
`d[key] = value`	Sets the associated value for `key` to `value`, used to either add a new key/value pair, or replace the value of an existing key/value pair.
`del d[key]`	Remove key and associated value from dictionary `d`.
`key in d`	`True` if key value `key` exists in dictionary `d`, otherwise returns `False`.

FIGURE 9-6 Some Operations for Dynamically Manipulating Dictionaries
LET'S TRY IT
From the Python Shell, enter the following and observe the results.

... fruit_prices 5 {'apples': .66, 'pears': .25,
'peaches': .74, 'bananas': .49}
... fruit_prices['apples']
???
... fruit_prices[0]
???
... veg_data 5 [['corn', .25], ['tomatoes', .49], ['peas', .39]]
... veg_prices 5 dict(veg_data)
... veg_prices
???
... veg_prices['peas']
???

A **dictionary** in Python is a mutable, associative data structure of variable length denoted by the use of curly braces.
9.1.2 Let's Apply It—Phone Number Spelling Program
The following Python program (Figure 9-8) generates all the possible spellings of the last four digits of any given phone number. The program utilizes the following programming features: ➤ dictionaries
Example execution of the program is given in Figure 9-7.

The program begins at **line 55**. First, a program welcome is displayed on **lines 55–56**. On **line 59**, variable terminate is assigned False. This variable controls the while loop at **line 61**. It is set to True, which terminates the loop, when the user indicates that they do not wish to enter any more phone numbers (**line 69**).

Within the loop, functions getPhoneNum and displayAllSpellings are called. Function getPhoneNum (**lines 3–32**) reads a phone number from the user of the form 123-456-7890.

Program execution ...

```
This program will generate all possible spellings of the
last four digits of any phone number

Enter phone number (xxx-xxx-xxxx): 410-555-7324
410-555-pdag
410-555-pdah
410-555-pdai
410-555-pdbg
410-555-pdbh
410-555-pdbi
410-555-pdcg
410-555-pdch
410-555-pdci
410-555-peag
410-555-peah
410-555-peai
410-555-pebg
410-555-pebh
410-555-pebi
.
.
Enter another phone number? (y/n): y
Enter phone number (xxx-xxx-xxxx): 410-555-4267
410-555-gamp
410-555-gamq
410-555-gamr
410-555-gams
410-555-ganp
410-555-ganq
410-555-ganr
410-555-gans
410-555-gaop
410-555-gaoq
410-555-gaor
410-555-gaos
.
.
```

FIGURE 9-7 Execution of Phone Number Spelling Program

On **line 8**, variable valid_ph_num is initialized to False, used to control the while loop on **line 12**. Only when a valid phone number has been entered is this variable set to True, which terminates the loop and returns the (valid) entered phone number.

Within the while loop, the entered phone number is input as a string (**line 13**). Two checks are made for the validity of the entered string. First, a check (**line 16**) is made as to whether the string is the wrong length (should be twelve characters long) or if the fourth (index 3) or eighth (index 7) characters are not a dash. If any of these errors are found, an error message is displayed (**line 18**) and another iteration of the while loop is executed. If, however, no error is found at this point, a second check is made that the remaining characters in the string (other than the two dashes) are digit characters. In this case, the string is assumed to contain proper digit characters (that is, valid_ph_ num is set to True on **line 22**), unless found otherwise (by use of string method isdigit on **line 26**). Thus, if a non-digit character is found, valid_ph_num is set to False and the inner while loop terminates, continuing with another execution of the outer while loop (at **line 12**). If, however, no non-digits are found, valid_ph_num remains True, thus terminating the outer while loop and returning the entered phone number.

```
1  # Phone Number Spelling Program
2
3  def getPhoneNum():
4
5      """Returns entered phone number in the form 123-456-7890."""
6
7      # init
8      valid_ph_num = False
9      empty_str = ''
10
11     # prompt for phone number
12     while not valid_ph_num:
13         phone_num = input('Enter phone number (xxx-xxx-xxxx): ')
14
15         # check if valid form
16         if len(phone_num) != 12 or phone_num[3] != '-' or \
17            phone_num[7] != '-':
18             print('INVALID ENTRY - Must be of the form xxx-xxx-xxxx\n')
19         else:
20         # check for non-digits
21             k = 0
22             valid_ph_num = True
23             phone_num_digits = phone_num.replace('-', empty_str)
24
25             while valid_ph_num and k < len(phone_num_digits):
26                 if not phone_num_digits[k].isdigit():
27                    print('* Non-digit:', phone_num_digits[k],'*\n')
28                     valid_ph_num = False
29                 else:
30                     k = k + 1
31
32     return phone_num
33
34 def displayAllSpellings(phone_num):
35
36     """Displays all possible phone numbers with the last four digits
37        replaced with a corresponding letter from the phone keys
38     """
39     translate =   {'0': ('0'), '1':('1'), '2': ('a','b','c'),
40                    '3': ('d','e','f'),'4': ('g','h','i'),
41                    '5': ('j','k','l'), '6': ('m','n','o'),
42                    '7': ('p','q','r','s'),'8': ('t','u','v'),
43                    '9': ('w','x','y','z')}
44
45     # display spellings
46     for let1 in translate[phone_num[8]]:
47         for let2 in translate[phone_num[9]]:
48             for let3 in translate[phone_num[10]]:
49                 for let4 in translate[phone_num[11]]:
50                     print(phone_num[0:8] + let1 + let2 + let3 + let4)
51
```

FIGURE 9-8 Phone Number Spelling Program (*Continued)*

In function displayAllSpellings (**lines 34–50**), we see our first example of the use of deeply nested (for) loops. Because we want to generate all combinations

of each of the four possible letters of the last four digits of the entered phone number, we utilize for loops nested four deep (**line 46**). Each digit ranges over its particular set of letters that appear on phone keys. Dictionary translate is defined for this purpose. In the dictionary, each digit 0–9 is a key value. The value associated with each key is a tuple containing the associated keypad letters for that

```
52 #---- main
53
54 # program welcome
55 print('This program will generate all possible spellings of the')
56 print('last four digits of any phone number\n')
57
58 # get phone number and display spellings
59 terminate = False
60
61 while not terminate:
62
63     phone_num = getPhoneNum()
64     displayAllSpellings(phone_num)
65
66     # continue?
67     response = input('Enter another phone number? (y/n): ')
68     if response == 'n':
69         terminate = True
```

FIGURE 9-8 Phone Number Spelling Program

digit. (Digits '0' and '1' return a string equal to themselves, since these phone digits do not have any letters associated with them.) For example, within the first for statement, dictionary translate contains the list of letters associated with the digit currently in the eighth position (the digit right after the second dash) of the entered phone number. Let’s say that digit is 7. Then the retrieved tuple would be ('p','q','r','s'). Therefore, all combinations of strings starting with these letters will be generated, as shown in Figure 9-9.

```
let1 = 'p','q','r','s'
     let2 = 'd','e','f'
          let3 = 'a','b','c'
               let4 = 'g','h','i'
                    410-555-pdag
                    410-555-pdah
                    410-555-pdai

                    410-555-pdbg
                    410-555-pdbh
                    410-555-pdbi

                    410-555-pdcg
                    410-555-pdch
                    410-555-pdci

                    410-555-peag
                    410-555-peah
                    410-555-peai

                    etc.
```

FIGURE 9-9 Nested for Loops in the Generation of Phone Number Spellings

On **line 67**, the user is prompted if they want to continue with another phone number. If they respond no ('n'), variable terminate is set to True, which terminates the outer while loop, and thus ends the program; otherwise, the program prompts for another number.

Self-Test Questions

1. A dictionary type in Python is an associative data structure that is accessed by a _____ rather than an index value.

2. Associative data structures such as the dictionary type in Python are useful for, **(a)** accessing elements more intuitively than by use of an indexed data structure **(b)** maintaining elements in a particular order

3. Which of the following types can be used as a key in Python dictionaries? **(a)** strings
(b) lists
(c) tuples
(d) numerical values

4. Which of the following is a syntactically correct sequence, s, for dynamically creating a dictionary using dict(s).
(a)s 5[[1: 'one'], [2: 'two'], [3: 'three']]
(b)s 5[[1, 'one'], [2, 'two'], [3, 'three']]
(c)s 5{1:'one', 2:'two', 3:'three'}

5. For dictionary d 5{'apples':0.66,'pears':1.25,'bananas':0.49}, which of the following correctly updates the price of bananas.
(a)d[2] 5 0.52
(b)d[0.49] 5 0.52
(c)d['bananas'] 5 0.52

ANSWERS: 1. key value, 2. (a), 3. (a), (c), (d), 4. (b), 5. (c)

9.2 Set Data Type
9.2.1 The Set Data Type in Python

A **set** is a mutable data type with nonduplicate, unordered values, providing the usual mathematical set operations as shown in Figure 9-10.

Set operator	Set A = {1,2,3} Set B = {3,4,5,6}		
membership	`1 in A`	True	*True if 1 is a member of set*
add	`A.add(4)`	{1,2,3,4}	*Adds new member to set*
remove	`A.remove(2)`	{1,3}	*Removes member from set*
union	`A | B`	{1,2,3,4,5,6}	*Set of elements in either set A or set B*
intersection	`A & B`	{3}	*Set of elements in both set A and set B*
difference	`A - B`	{1,2}	*Set of elements in set A, but not set B*
symmetric difference	`A ^ B`	{1,2,4,5,6}	*Set of elements in set A or set B, but not both*
size	`len(A)`	3	*Number of elements in set (general sequence operation)*

FIGURE 9-10 Set Operators
One of the most commonly used set operators is the in operator (which we have been already using with sequences) for determining membership,

... fruit 5 {'apple', 'banana', 'pear', 'peach'} ... fruit
{'pear', 'banana', 'peach', 'apple'}

... 'apple' in fruit True

Note that the items in the set are not displayed in the order that they were defined. Sets, like dictionaries, do not maintain a logical ordering. The order that items are stored is determined by Python, and not by the order in which they were provided. Therefore, it is invalid and makes no sense to access an element of a set by index value.

The add and remove methods allow sets to be dynamically altered during program execution, as shown below,

... fruit.add('pineapple')
... fruit
{'pineapple', 'pear', 'banana', 'peach', 'apple'}

To define an initially empty set, or to initialize a set to the values of a particular sequence, the set constructor is used,

... set1 5 set() ... vegs 5 ['peas', 'corn'] ... vowels 5 'aeiou' ... len(set1) ... set(vegs) ... set(vowels) 0 {'corn', 'peas'} {'a', 'i', 'e', 'u', 'o'}

Note that set(), and not empty braces are not used to create an empty set, since that notation is used to create an empty dictionary. Because sets do not have duplicate elements, adding an already existing item to a set results in no change to the set.

Finally, there are two set types in Python—the mutable set type, and the immutable frozenset type. Methods add and remove are not allowed on sets of frozenset type. Thus, all its members are declared when it is defined,

... apple_colors 5 frozenset(['red', 'yellow', 'green'])
As shown, the values of a set of type frozenset must be provided in a single list when defined. (A frozenset type is needed when a set is used as a key value in a given dictionary.) LET’S TRY IT From the Python shell, enter the following and observe the results.

... s 5 {1,2,3} ... s 5 set(['apple', 'banana', 'pear']) ... 1 in s ... s
??? ???

... s.add(4) ... s.add('pineapple')

... s ???

??? ... s 5 frozenset(['apple', 'banana', 'pear']) ... s 5 set('abcde') ...
s.add('pineapple')
... s ... ???
???

A **set** is a mutable data structure with nonduplicate, unordered values, providing the usual set operations. A **frozenset** is an immutable set type.

9.2.2 Let’s Apply It—Kitchen Tile Visualization Program

The following Python program (Figure 9-12) allows the user to select a particular kitchen tile size, a primary and secondary tile color, the frequency in which the secondary color is to be placed, and a grout color. It then displays the resulting tile pattern. This program utilizes the following programming features:

➤ sets

Example execution of the program is given in Figure 9-11 (shown as grey tones in this image). Program execution begins on **line 154**. Variable tile_area is assigned to a dictionary
that holds the width and height of the turtle window to be created. Variable grout_color_
selection is assigned to a set containing string values 'white', 'gray', 'brown', and 'black'. These strings are recognized in turtle graphics as color values. The user is prompted to
enter one of these values as the grout color, which is set as the background color of the turtle screen
within function layoutTile. Variable scaling is used to appropriately adjust the size of the
displayed tile. Changing this value will make tiles appear larger or smaller on the screen. Function greeting is called on **line 159** to display an explanation of the program to the
user. On **line 162**, the getTileSelections function is called. This function prompts the user
for the tile width and length, the primary and secondary colors of the tiles, the number of primary
colors that should be displayed for each secondary tile color, and the grout color. These values are

returned in a dictionary of the form,

{'tile_size':{'length':tile_length, 'width':tile_width}, 'primary_color':color1, 'secondary_color':color2, 'tile_skip':skip, 'grout_color':grout}

```
Program Execution ...

This program will display the tile pattern for a selected
pair of tile colors, tile size, and grout color.

The repeat frequency for the secondary tile color can be selected:
if skip = 1, secondary tile color placed every other tile,
if skip = 2, secondary tile color placed every third tile, etc.

Enter tile length (1-4 inches): 4
Enter tile width (1-4 inches): 4
Enter primary tile color (6-digit RGB): 006400
Enter secondary tile color (6-digit RGB): D2B4AC
Enter frequency of secondary tile color (1-10): 2
Enter grout color (white, gray, brown or black): white
```

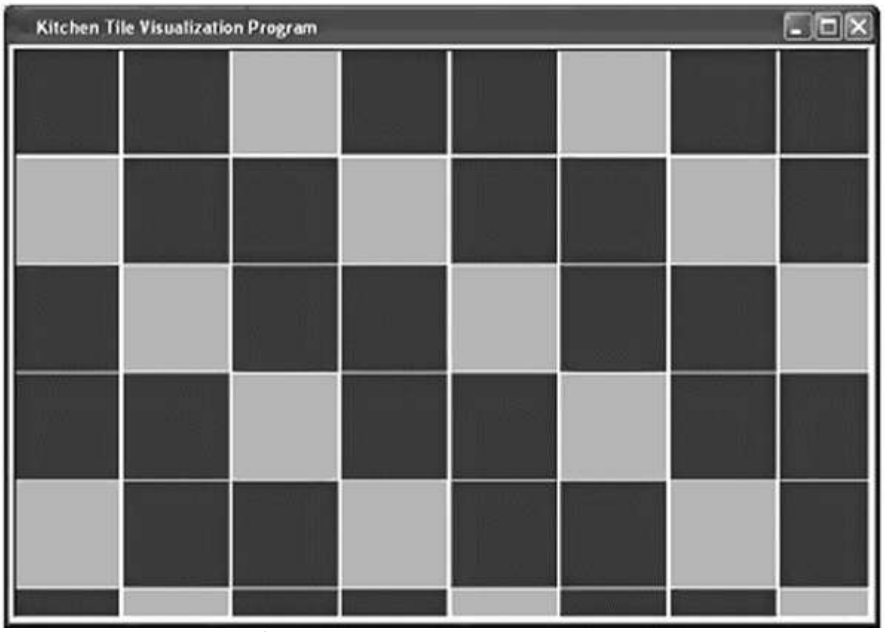


FIGURE 9-11 Execution of the Kitchen Tile Visualization Program

Thus, for example, for variable selections assigned to this dictionary, selections['tile_ size'] gets the selected tile length and width, and selections['primary_color'] gets the primary tile color. Note that the value associated with key 'tile_size' is itself a dictionary of the form,

{'length':tile_length, 'width':tile_width}

Thus, for variable tile_size assigned to this dictionary, tile_size['length'] returns the user-selected tile length, and tile_size['height'] returns the height.

On **line 165**, the setup method of turtle graphics is called to establish the turtle screen. On **line 168**, the reference to the screen created is assigned to variable window. For this particular program, the normal coordinate system in turtle graphics with coordinate (0,0) at the center of the screen is not the most "natural" coordinate system. Turtle graphics allows the coordinate system to be redefined by specifying the coordinates of the bottom-left corner and the top-right corner of the

```
# Kitchen Tile Visualization Program

import turtle

def greeting():

    """Displays the program greeting on the screen. """

    print('This program will display the tile pattern for a selected')
    print('pair of tile colors, tile size, and grout color.\n')
    print('The repeat frequency for the secondary tile color can be selected:')
    print('if skip = 1, secondary tile color placed every other tile,')
    print('if skip = 2, secondary tile color placed every third tile, etc.\n')

def getTileSelections(grout_color_options):

    """Returns dictionary of the form,

        {'tile_size':dict, 'color1':str, 'color2':str, 'skip':int, 'grout':str}

        where the value of tile_size is a dictionary of the form
        {'tile_length': value, 'tile_width': value},

        color1 and color2 are the selected primary and secondary color code
        strings in hexadecimal format, skip is an integer indicating the
        number of primary colors that occur for every secondary color, and
        grout contains the selected grout color in hexadecimal format.
    """

    # init
    empty_set = set()
    hex_digits = {'0','1','2','3','4','5','6','7','8','9',
                  'A','B','C','D','E','F','a','b','c','d','e','f'}

    # prompt user for tile size (length and width)
    tile_length = int(input('Enter tile length (1-4 inches): '))
    while tile_length <1 or tile_length > 4:
        tile_length = int(input('INVALID SIZE SELECTED - please re-enter: \n'))

    tile_width = int(input('Enter tile width (1-4 inches): '))
    while tile_width < 1 or tile_width > 4:
        tile_width = int(input('INVALID SIZE SELECTED - please re-enter: \n'))

    # prompt user for primary tile color
    color1 = input('Enter primary tile color (6-digit RGB): ')
    while (len(color1) != 6) or \
          (set(color1) - hex_digits != empty_set):
        color1 = input('INVALID RGB color, please re-enter: ')

    # prompt user for secondary tile color
    color2 = input('Enter secondary tile color (6-digit RGB): ')
    while (len(color2) != 6) or \
          (set(color2) - hex_digits != empty_set):
        color2 = input('INVALID RGB color, please re-enter: ')

    # prompt user for skip pattern for colors
    skip = int(input('Enter frequency of secondary tile color (1-10): '))
    while (skip < 1 or skip > 10):
        skip = int(input('INVALID - Please enter 1,2,3,etc. '))

```

```
    # prompt user for grout color
    grout = input('Enter grout color (white, gray, brown or black): ')
    while (grout not in grout_color_options):
        grout = input('INVALID grout color selection, please re-enter: ')

    # prepend hash sign to indicate RGB string
    color1 = '#' + color1
    color2 = '#' + color2

    # create dictionary of selections
    selections = {'tile_size':{'length':tile_length, 'width':tile_width},
                  'primary_color':color1, 'secondary_color':color2,
                  'tile_skip':skip, 'grout_color':grout}

    return selections

def layoutTiles(window, selections, tile_area, scaling):

    """Displays the tiles in the window starting with the top and working
        towards the bottom of the window. This function requires that the
        coordinate (0,0) be set as the top corner of the window.
    """

    # set background color (as grout color)
    window.bgcolor(selections['grout_color'])

    # get selected tile size
    tile_size = selections['tile_size']

    # get turtle
    the_turtle = turtle.getturtle()

    # scale size of tiles for display
    scaled_length = scaling * tile_size['length']
    scaled_width = scaling * tile_size['width']

    # scale grout spacing
    tile_spacing =  6

    # create tile shape
    turtle.register_shape('tileshape',
                          ((0,0), (0, scaled_length),
                           (scaled_width, scaled_length), (scaled_width, 0)))

    # set turtle attributes
    the_turtle.setheading(0)
    the_turtle.shape('tileshape')
    the_turtle.hideturtle()
    the_turtle.penup()

    # place first tile at upper left corner
    loc_first_tile = (-10, tile_area['height'] + 10)
    the_turtle.setposition(loc_first_tile)

    # init first tile color and counters
    first_tile_color = 'primary_color'
    skip_counter = selections['tile_skip']
    row_counter = 1

```

```
    terminate_layout = False
    while not terminate_layout:

        # check if current row of tiles is before right edge of window
        if the_turtle.xcor()< tile_area['width']:

            # check if need to switch to secondary tile color
            if skip_counter == 0:
                the_turtle.color(selections['secondary_color'])
                skip_counter = selections['tile_skip']
            else:
                the_turtle.color(selections['primary_color'])
                skip_counter = skip_counter - 1

            # place current tile color at current turtle location
            the_turtle.stamp()

            # move turtle to next tile location of current row
            the_turtle.forward(scaled_length + tile_spacing)

        # check if current row of tiles at bottom edge of window
        elif turtle.ycor() + scaled_width > 0:

            the_turtle.setposition(loc_first_tile[0],
                                   loc_first_tile[1]- row_counter *
                                   scaled_width - row_counter * tile_spacing)

            row_counter = row_counter + 1
        else:
            terminate_layout = True

# ---- main

# init
tile_area = {'width': 660, 'height': 440}
grout_color_selection = {'white', 'gray', 'brown', 'black'}
scaling = 20

# program greeting
greeting()

# get tile selection and layout details
selections = getTileSelections(grout_color_selection)

# set window size
turtle.setup(tile_area['width'], tile_area['height'])

# get reference to turtle window
window = turtle.Screen()

# set window title
window.title('Kitchen Tile Visualization Program')

# set coordinate system
window.setworldcoordinates(0, 0, tile_area['width'], tile_area['height'])
window.mode('world')

# layout tiles in window
layoutTiles(window, selections, tile_area, scaling)

# terminate program when window closed
turtle.exitonclick()
```

FIGURE 9-12 Kitchen Tile Visualization Program

screen. This is done by call to method setworldcoordinates on **line 174**. We therefore set the bottom-left corner to be coordinate (0,0), and the top-right corner to be coordinate (tile_ area['width'], tile_area['height']). Thus, for the current tile width of 660 and tile height of 400, the top-right coordinate would be (660,400). This is depicted in Figure 9-13.

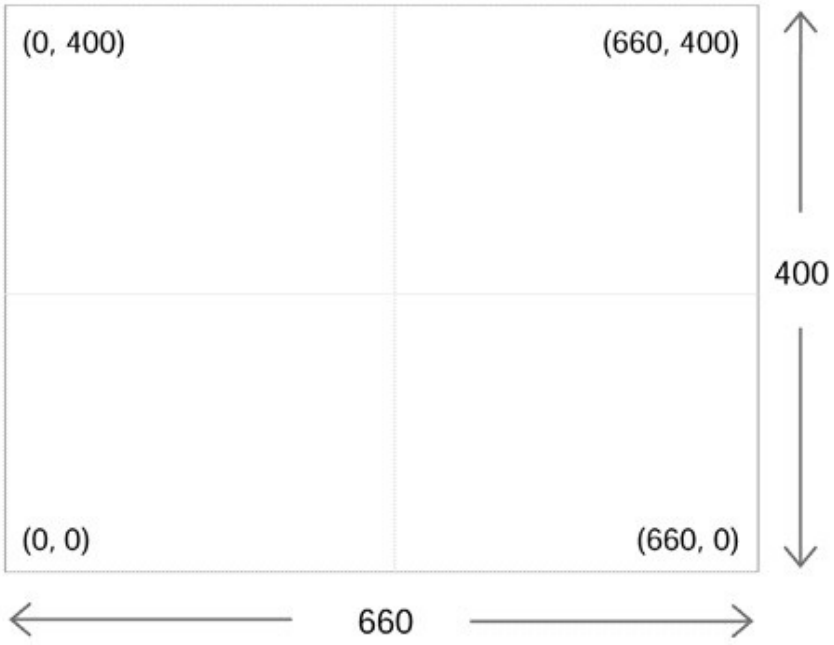


FIGURE 9-13 Screen Coordinates for the Kitchen Tile Visualization Program

In order to utilize the new coordinate system, the turtle screen mode must be set to 'world' (on line **175**).

The final steps of the program includes a call (**line 178**) to function layoutTiles. It is passed variable window, holding a reference to the turtle screen created, variable selections, which holds all the user-selected options, variable tile_area, which holds the dimensions of the turtle screen, and variable scaling, which indicates how the tile dimensions in dictionary selections are to be scaled. In the last line of the program, **line 181**, turtle method exitonclick() is called so that the program terminates when the turtle window is closed.

We now look at program functions getTileSelections and layoutTile. Function getTileSelections (**lines 15–75**) begins by setting variable empty_set to an empty set, empty_set 5set(). Recall that empty braces are used to create an empty dictionary, and therefore set() is used to created an empty set. Variable empty_set is used for input error checking as we will see. Variable hex_digits is

assigned to a set containing each of the digits 0–9, as well as the uppercase and lowercase letters A–F. Collectively, these are the symbols used in denoting hexadecimal numbers. This set is also used for input error checking. In **lines 35–64**, the user is prompted to enter all the tile selection options. At **line 36**, the user is prompted for the tile length, limited to a size of four inches. Following, at **line 40**, the user is likewise prompted for the tile width, also limited to four inches. At **line 45**, the user is prompted for the primary tile color. This is the color that will appear most often (or just as often, if the selected pattern is one primary color followed by one secondary color, etc.). Finally, at **line 51**, the user is prompted for the secondary color.

Color values are entered as six hexadecimal digits in the range 000000 (black) to FFFFFF (white). Since color values may contain the letters A–F (a–f), as well as the digits 0–9, any entered color value containing characters other than that is invalid. Example color values are given in Figure 9-14.

white	FFFFFF		DarkSalmon	E9967A		Amethyst	9966CC
ivory	FFFFF0		LightSalmon	FFA07A		DarkViolet	9400D3
Linen	FAF0E6		FireBrick	B22222		DarkOrchid	9932CC
Beige	F5F5DC		DarkRed	8B0000		DarkMagenta	8B008B
LightGray	D3D3D3		Tomato	FF6347		Indigo	4B0082
DarkGray	A9A9A9		OrangeRed	FF4500		CornSilk	FFF8DC
SlateGray	708090		DarkOrange	FF8C00		BlanchedAlmond	FFEBCD
PaleGreen	98FB98		Orange	FFA500		Bisque	FFE4C4
SeaGreen	2E8B57		Gold	FFD700		Wheat	F5DEB3
DarkSlateGray	2F4F4F		Yellow	FFFF00		BurlyWood	DEB887
DarkGreen	006400		LightYellow	FFFFE0		Tan	D2B48C
Olive	808000		LemonChiffon	FFFACD		SandyBrown	F4A460
Teal	008080		Moccasin	FFE4B5		GoldenRod	DAA520
Aquamarine	7FFFD4		PaleGoldenrod	EEE8AA		DarkGoldenrod	B8860B
CadetBlue	5F9EA0		Thistle	D8BFD8		Peru	CD853F
SkyBlue	87CEEB		Plum	DDA0DD		Chocolate	D2691E
CornFlowerBlue	6495ED		MediumPurple	9370DB		SaddleBrown	8B4513
SteelBlue	4682B4		Orchid	DA70D6		Sienna	A0522D
Navy	000080		MediumOrchid	BA55D3		Brown	A52A2A

FIGURE 9-14 Selected Hexadecimal Color Codes

The check for invalid color values occurs on **lines 46** and **52**. First, the color values must be six characters long. Then, the color value strings are converted into a set using set(color1) and set(color2). Thus, for example, for color1 equal to FF3243, set(color1) would produce the following set,

{'3', '2', '4', 'F'}
Then, the set difference operator, 2, is applied to this set, and the set hexdigits defined at the start of the function,
set(color1) 2 hexdigits

Recall that the set difference operator, A – B, returns any items that are in set A,

but not in set B. Since set A is the entered color string and set hexdigits contains all the valid characters of a hexadecimal number, if this difference is not empty (the empty set), then color1 (or color 2) must contain invalid characters.

The final items for which the user is prompted is the skip pattern of the tiles and the grout color. The value of skip indicates how many primary color tiles are placed before a secondary color tile is placed. Thus, for a skip value of 1, the primary and secondary tiles will alternate every other tile. For a skip value of 2, two primary color tiles will be placed for every secondary color tile, and so forth. The grout color entered by the user must be one of the string values in set grout_color_options. Thus, if the entered string is not in this set, the user is prompted to re-enter.

Finally, on lines **67** and **68**, the '#' symbol is prepended to each of the color strings, color1 and color2. This indicates in Python that the string contains a hexadecimal value. At **line 71**, a dictionary is built containing all of the selected values to be returned as the function value (**line 75**).

Function layoutTile, **lines 77–153**, is passed all the information needed to lay out the tiles according to the turtle screen size, the user's selections, and the scaling factor to be used. On **line 85**, the background color of the turtle screen is set to the user-selected grout color. On **line 88**, variable tile_size is set to the value for key 'tile_size' in the selections dictionary. Recall that this value is itself a dictionary. Thus, variable tile_size is holding a dictionary with keys 'length' and 'width'.

On **line 91**, the reference to the turtle is set to variable the_turtle. On **lines 94–95** the dimensions of each tile are scaled by the scaling factor in variable scaling, and assigned to variables scaled_length and scaled_width. Since the user is entering tile size in inches, a fourinch by four-inch tile would be displayed as four pixels by four pixels, which is too small for the purpose of displaying tile patterns. Thus, with a scaling factor of 20, for example, a four-by-four inch tile would be displayed as 80 pixels by 80 pixels. The spacing between tiles (in pixels) is set to 6, assigned to variable tile_spacing on **line 98**.

In order to create the appropriate tile shape as specified by the user, the register_shape method in turtle graphics is used to both describe the shape and provide a shape name. Thus, on **line 101**, register_shape is called with the name 'tileshape' as the first parameter value, followed by a list of coordinate values

defining the desired shape. The coordinate values are specified in terms of the values scaled_length and scaled_width to define the specific size of the tile shape to match the user's specified size.

On **lines 106–109**, the turtle attributes are set for laying out tiles. Since the tiles will be laid out from the top of the screen going left to right, the turtle heading is set to 0 (facing right) and the shape is set to 'tileshape' (on **lines 106–107)**. Since the turtle is used to stamp its image where each tile is to be placed, and not be visible or draw lines on the screen, the turtle is hidden (**line 108**) with its pen set to up (**line 109**).

In order for the tile to appear in the correct location, a ten-pixel adjustment is made. Thus, on **line 112**, variable loc_first_tile is set to the coordinate (210, tile_area['height'] 1 10). The turtle is then positioned at this location (**line 113**) to begin the placement of tiles. Since the primary color tile is always the first tile placed, variable first-tile-color is set to the key value 'primary_color' (of dictionary selections). Variable skip_counter is also set to the skip value stored in the selections dictionary with the key value 'tile_skip'. In order to determine where to begin the placement of tiles for each new row, variable row_counter is initialized to 1 and incremented for each next row of tiles. Thus, knowing the number of each new row allows for the coordinate value of each new row to be calculated (since each row of tiles is the same height).

Lines 120–149 perform the placement of tiles. Variable terminate_layout is initialized to False on **line 120**, which controls the while loop on **line 121**. It is not set to True until the last row of tiles has been completed. The if statement at **line 124** checks whether the x-coordinate value of the turtle’s current location is past the right edge of the screen. If not, then the color of the next tile to be placed is determined based on the current value of variable skip_ counter. This variable is used as a “count-down” counter. That is, it is decremented by one each time another tile has been placed. When skip_counter reaches 0, the next tile is set to the secondary color, and the skip_counter is reset to the value in variable tile_skip to again count down to zero (**lines 127–132**). Once the color of the tile has been determined, the stamp method is called on the turtle to “imprint” its image on the screen (**line 135**). The turtle is then moved forward (to the right) by an amount equal to the tile length plus the amount of tile spacing (**line 138**). Thus, the turtle is positioned for the placement of the next tile, and execution returns to the top of the while loop.

If the condition of the if statement on **line 124** is false, that is, the current tile is past the right edge of the screen, then a check is made on **line 141** if the current tile position is past the bottom of the screen (that is, the_turtle.ycor() is less than zero). If not, then the turtle is repositioned to the beginning of the next row of tiles (**lines 143–145**) making use of variable row_counter, as mentioned. Because a new row of tiles is started, row_counter is incremented by one. Finally, if the conditions at both **line 124** and **line 141** are found false (that is, the current turtle position is at the end of the last row of tiles), then terminate_layout is set to True and thus the while loop terminates, also terminating the function.

Self-Test Questions

1. Indicate all of the following that are syntactically correct for creating a set. **(a)** set([1, 2, 3])
(b)set((1, 2, 3))
(c) {1, 2, 3}

2. For set s containing values 1, 2, 3, and set t containing 3, 4, 5, which of the following are the correct results for each given set operation?
(a) s | t → {3}
(b)s & t → {1, 2, 3, 4, 5}
(c) s 2 t → {1, 2}
(d)s ^ t → {1, 2, 4, 5}

3. For set s containing values 1, 2, 3 and set w of type frozenset containing values 'a','b','c', which of the following are valid set operations?
(a) 'a' in s
(b)'a' in w
(c) len(s) 1 len(w)
(d)s.add(4)
(e) w.add('d')
(f) s | w
(g) s & w
(h)s 2 w

ANSWERS: 1. (a), (b), (c), 2. (c), (d), 3. (a), (b), (c), (d), (f), (g)

COMPUTATIONAL PROBLEM SOLVING

9.3 A Food Co-op’s Worker Scheduling Simulation

In this section, we develop a program for simulating the number of people that

show up for work at a food co-op using two different worker scheduling methods —one in which workers sign up for certain time slots, and the other in which workers show up whenever they want.

9.3.1 The Problem

A food co-op in a university town found an interesting solution to a scheduling problem. The co-op offered two prices for everything— one price for members, and a higher price for nonmembers. To qualify for the lower prices, each member had to volunteer to work at the co-op for a couple hours each week.

The problem was that the co-op needed two people to be working at all times. Members would be asked to sign up for time slots, limited to two workers for each slot. Too often, however, members who had signed up for a given time did not show up, leaving the co-op either completely or partially uncovered.

The co-op eventually devised an effective, albeit daring solution to this problem. They decided to just let members come in to work whenever they wanted to, with no planned scheduling! This unscheduled approach ran the risk of there being times in which the co-op was left without any workers. What was interesting was that they found this approach to be a better solution than the scheduled approach. (We will look at whether this is also better for the individual members or not.)

In this section, we develop a program capable of performing a simulation of both approaches based on assumed probabilities of typical human behavior. We then compare the effectiveness of the scheduled vs. the unscheduled approach both from the co-op’s and the members’ point of view.

9.3.2 Problem Analysis

The computational issue for this problem is to model and simulate the behavior of individuals for assumed probabilities of certain actions. The behaviors in this case are related to fulfilling a commitment to work a given number of hours each week. In one scenario, workers sign up in advance to work certain time slots; in the other scenario, workers show up to work whenever they feel like it.

Besides assumed probabilities of workers showing up for work, there are also assumed probabilities of workers showing up late or leaving early. Since each of these actions is probabilistic, there needs to be a computational means of determining when such actions take place. We use a random number generator for this. For example, if there is an assumed 10% chance that any given person may show up late, a random number between 1 and 10 is generated. If the generated value is 1, the action is assumed to occur; otherwise, the action is assumed not to occur.

We will assume a certain schedule of hours that the co-op is open, given below.

Sunday 12:00 pm–6:00 pm Monday–Thursday 8:00 am–6:00 pm Friday, Saturday 8:00 am–8:00 pm

We also assume that every time slot is two hours long. Thus, for example, there would be three time slots on Sundays, 12:00–2:00 pm, 2:00–4:00 pm, and 4:00–6:00 pm. Based on this, there are 35 time slots in a week. We also assume that the co-op has 75 members.

Because of the different natures of the two scheduling approaches, we assume different probabilities for the behaviors of members. For the scheduled approach,

we assume a probability of 15% that a given member *will not* show up for their time slot. For the unscheduled approach, we assume a probability that any given worker *will* decide to show up for a given time slot to be 5%. Also, we assume that the chance that a scheduled worker will show up late for the start of the time slot is greater than in the unscheduled approach, since in the unscheduled approach workers show up for the time slot that is convenient for them. On the other hand, we assume a greater chance that unscheduled workers will leave fifteen minutes earlier than scheduled workers, since they may feel less committed to working the complete time slot. We assume the probabilities for these behaviors as given in Figure 9-15.

Scheduled Approach Probabilities	Unscheduled Approach Probabilities
Chance of arriving 15 minutes late is 15%.	Chance of arriving 15 minutes late is 5%.
Chance of arriving 30 minutes late is 5%.	Chance of arriving 30 minutes late is 2%.
Chance of arriving 45 minutes late is 2%.	Chance of arriving 45 minutes late is 1%.
Chance of leaving 15 minutes early is 5%	Chance of leaving 15 minutes early is 10%
Chance of leaving 30 minutes early is 3%	Chance of leaving 30 minutes early is 3%
Chance of not showing up at all is 15%	Change of deciding to show up is 5%

FIGURE 9-15 Assumed Probabilities for the Food Co-op Simulation Program

9.3.3 Program Design

Meeting the Program Requirements

The program must simulate the number of workers that show up for each of the time slots that the co-op is open by utilizing assumed probabilities. The co-op requires two workers in the store at all times that it is open. The program must also utilize the probabilities that workers will show up 15, 30, or 45 minutes late for work, or leave 15 or 30 minutes early.

Data Description

The data that needs to be represented in this program includes the number of co-op members (stored as an integer), the two-hour time slots that a worker may work (stored as a tuple), the probabilities of each of the actions that may occur (stored as a dictionary), and the names of the days of the week (stored as a tuple) as given below:

num_members 5 75
time_slots 5 ('8:00am','10:00am','12:00pm','2:00pm','4:00pm','6:00pm') days 5 ('Sunday','Monday','Tuesday','Wednesday','Thursday','Friday',

'Saturday')
sched_probabilities 5 { 'CLate_15':15, 'CLate_30':5, 'CLate_45':2, 'CLEarly_15':5, 'CLEarly_30':3,'CNoshow':15} unsched_probabilities 5 { 'CLate_15':5, 'CLate_30':2, 'CLate_45':1, 'CLEarly_15':10, 'CLEarly_30':3,'CShowup':5}

In dictionary probabilities, CLate_15, CLate_30, and CLate_45 contain the chances that a worker will arrive 15, 30, or 45 minutes late, respectively; CLEarly_15 and CLEarly_30 contain the chances that a worker will leave 15 or 30 minutes early, respectively; CNoshow contains the chance that a scheduled worker will not show up for their time slot; and CShowup is the chance that an unscheduled worker will decide to show up to work.

Algorithmic Approach

The algorithmic approach for this problem relies on the use of randomization to run the simulation. One complete week of co-op staffing is simulated, including the number of workers showing up for each of the 35 time slots in a week, how many show up a given number of minutes late, and how many leave a given number of minutes early. For the assumed probability of each of these actions, a random number will be generated for simulating whether each event has occurred or not.

Overall Program Steps
The overall steps of the program are given in Figure 9-16.

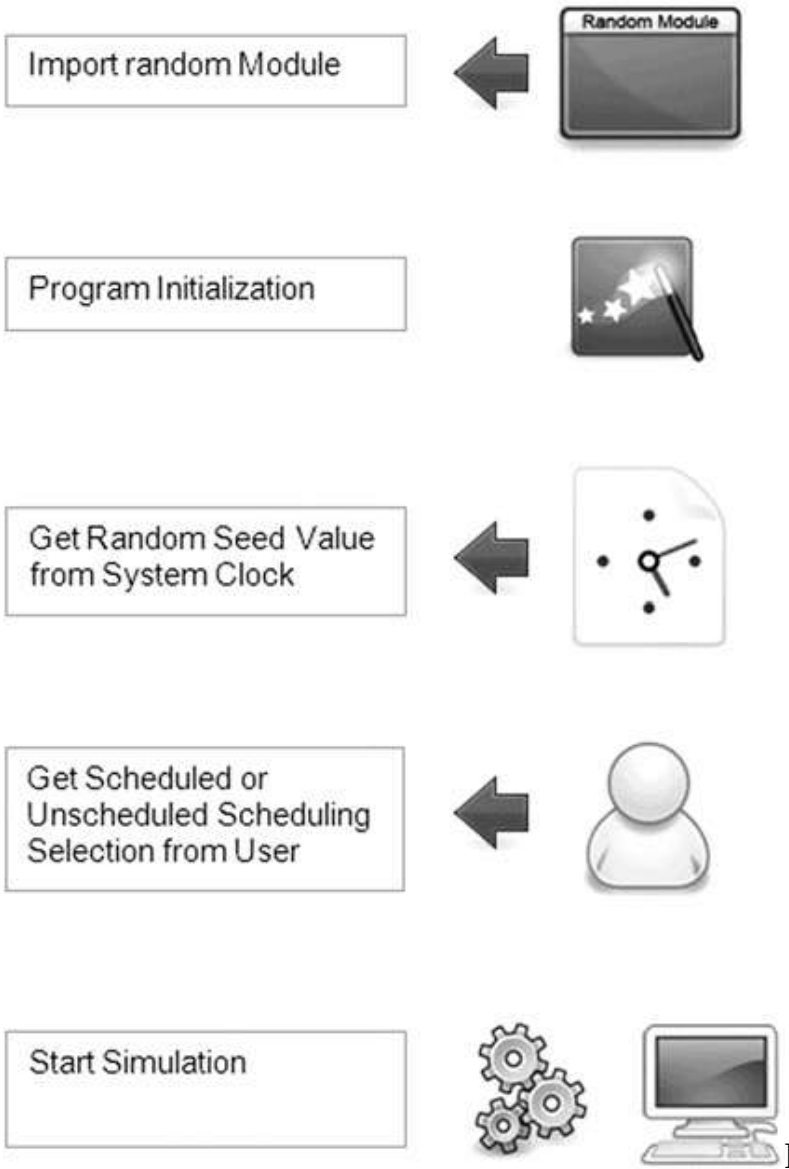


FIGURE 9-16 Overall Steps of Food Co-op Program

Modular Design

The modular design for this program is given in Figure 9-17. Following this design, there are six functions in the program. Function execute ScheduledSimulation determines the time slots for a given day of the week by calling function displayScheduledWorkerHours to probabilistically determine whether a scheduled worker shows up, and if so, if they show up a certain number of minutes late and/or leave a certain number of minutes early. The function, in turn, relies on the use of function eventOccurred, which is provided a probability value, and returns True or False to

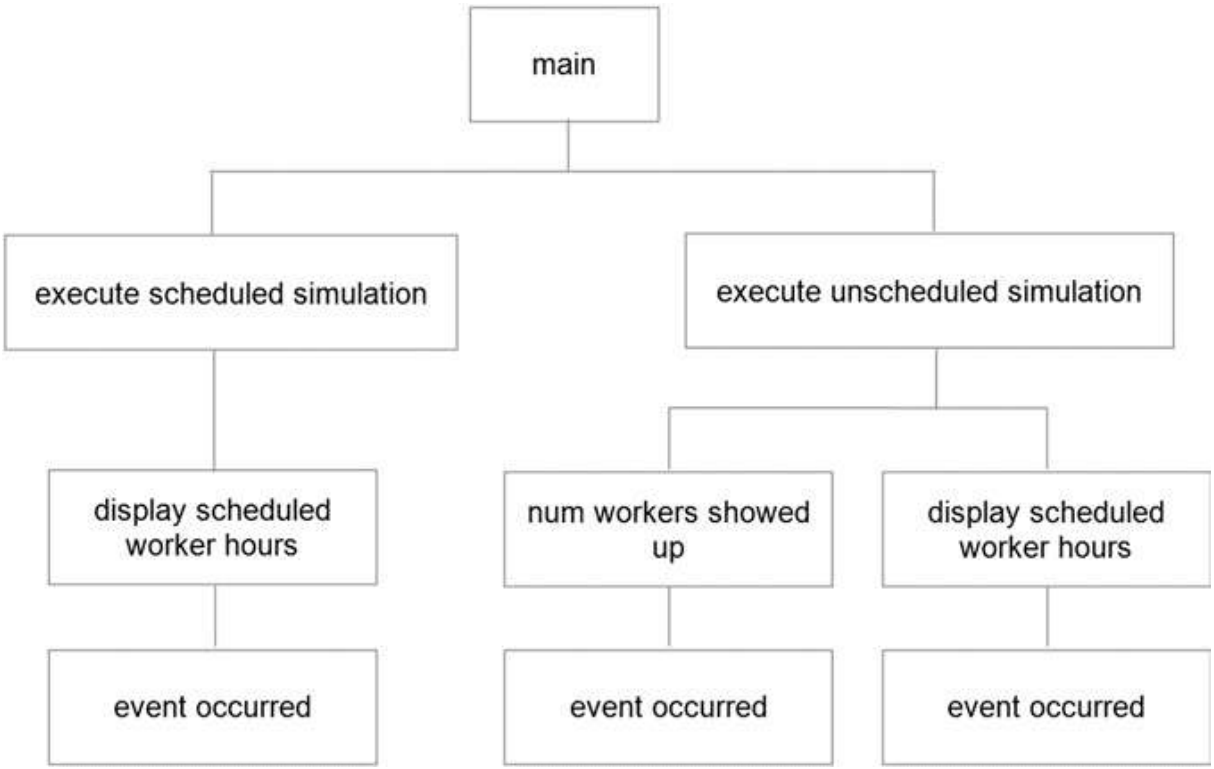


FIGURE 9-17 Modular Design of the Food Co-op Simulation Program

simulate whether the event has occurred or not. Function executeUnscheduledSimulation is designed similarly, except that it also needs to determine how many (unscheduled) workers decide to show up at their convenience. Thus, the function relies on function numWorkersShowedUp (which in turn relies on function eventOccurred). We give the specification for each of these functions in Figure 9-18. From this we can begin to implement and test the program.

9.3.4 Program Implementation and Testing

The implementation of the main module of this program is given in Figure 9-19. The main module gives the overall steps of the program. First, all values that determine the model's behaviors are initialized. Variable num_members is assigned the assumed number of co-op members for the simulation, 75 (**line 4**). Tuple time_slots is assigned the start time of all the two-hour time slots that workers may work (**line 5**), and tuple days is assigned the days of the week that the co-op is open (every day of the week) on **line 8**. The dictionary sched_probabilities contains the assumed probabilities for when scheduled workers show up late, leave early, or don't show up at all (**lines 11–12**). The

dictionary unsched_probabilities contains the assumed probabilities for when unscheduled workers show up late, leave early, and decide to show up to work for a given time slot (**lines 14–15**).

On **line 18**, the seed function for the random number generator is called (the required import random statement is provided at the top of the program). When random.seed() is called without an argument, the seed value is taken from the system clock (usually from the lower order digits of the time in the milliseconds range). This ensures that each time the program is run, a different sequence of random numbers most likely will be generated.

The remainder of the program handles the task of prompting the user for the type of simulation to execute (“scheduled” or “unscheduled”). Based on the user’s response, either function executeScheduledSimulation (at **line 36**) or function executeUnscheduledSimulation (at **line 40**) is called. Variable valid_input is initialized to False (at **line 23**), set to True only when a valid response of 1 or 2 is entered for the desired simulation (at **line 43**). Thus,

```
def eventOccurred(chances):
    """For given integer value chances as a percentage value (1-100), returns
       True if randomly generated number in range (1,100) is less than or equal
       to chances, otherwise returns False.
    """
```

```
def numWorkersShowedUp(indiv_chance, num_individuals):
    """For given integer values indiv_chance and num_individuals, returns how
       many times that num_individual calls to event_occurred with argument
       indiv_chance returns True.
    """
```

```
def displayScheduledWorkerHours(workernum, probabilities):
    """For workernum equal to 1 or 2, and the proability values given in dictionary
       probabilities of the form,

         {'CLate_15':<num>, 'CLate_30':<num>, 'CLate_45':<num>,
          'CLEarly_15':<num>, 'CLEarly_30':<num>,'CNoshow':<20>,
          'CShowup':<num>}

       where each <num> is a value between 1-100, displays one of

         worker <worker_num> -- no show --   or
         worker <worker_num> <mins_late> mins late  and/or
         worker <worker_num> left <mins_early> mins early

       where <mins_late> is 15, 30, or 45,  <mins_early> is 15 or 30.
    """
```

```
def displayUnscheduledWorkerHours(workernum, probabilities):
    """For workernum equal to 1 or 2, and the proabilities given in dictionary
       probabilities of the form,

         {'CLate_15':<num>, 'CLate_30':<num>, 'CLate_45':<num>,
          'CLEarly_15':<num>, 'CLEarly_30':<num>,'CNoshow':<20>,
          'CShowup':<num>}

       where each <num> is a value between 1-100, displays one of

         worker <worker_num> -- no show --   or
         worker <worker_num> <mins_late> mins late  and/or
         worker <worker_num> left <mins_early> mins early

       where <mins_late> is 15 or 30,
             <mins_early> is 15 or 30.
    """
```

```
def executeScheduledSimulation(probabilities, days, time_slots):
    """Displays a simulated week of workers in attendance for all time slots,
       based on a scheduled worker schedule.

    """
```

```
def executeUnscheduledSimulation(probabilities, days, time_slots):
    """Displays a simulated week of workers in attendance for all time slots,
        based on an unscheduled worker schedule.
    """
```

FIGURE 9-18 Function Specifications of the Food Co-op Simulation Program

```
# ---- main

# init
num_members = 75
time_slots = ('8:00am', '10:00am', '12:00pm', '2:00pm', '4:00pm',
              '6:00pm')

days = ('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',
        'Friday', 'Saturday')

sched_probabilities = {'CLate_15':15, 'CLate_30':5, 'CLate_45':2,
                       'CLEarly_15':5, 'CLEarly_30':3,'CNoshow':15}

unsched_probabilities = {'CLate_15':5, 'CLate_30':2, 'CLate_45':1,
                         'CLEarly_15':10, 'CLEarly_30':3,'CShowup':5}

# seed random number generator with system clock
random.seed()

# get type of simulation
print('Welcome to the Food Co-op Schedule Simulation Program')

valid_input = False
while not valid_input:
    try:
        response = \
            int(input('(1)scheduled, (2)unscheduled simulation? '))

        while (response != 1) and (response != 2):
            print('Invalid Selection\n')
            response = \
                int(input('(1)scheduled, (2)unscheduled simulation? '))

        if response == 1:
            print('<< SCHEDULED WORKER SIMULATION >>\n')
            executeScheduledSimulation(sched_probabilities, days,
                                       time_slots)
        else:
            print('<< UNSCHEDULED WORKER SIMULATION >>\n')
            executeUnscheduledSimulation(unsched_probabilities,
                                         days, time_slots)

        valid_input = True

    except ValueError:
        print('Please enter numerical value 1 or 2\n')
```

FIGURE 9-19 Main Module of the Food Co-op Simulation Program

the while loop continues to execute as long as an invalid response is entered. Because the input value is converted to an integer type,

response 5 int(input('(1) scheduled, (2) unscheduled simulation? '))

a ValueError will be raised (by function input) if the user enters a non-digit

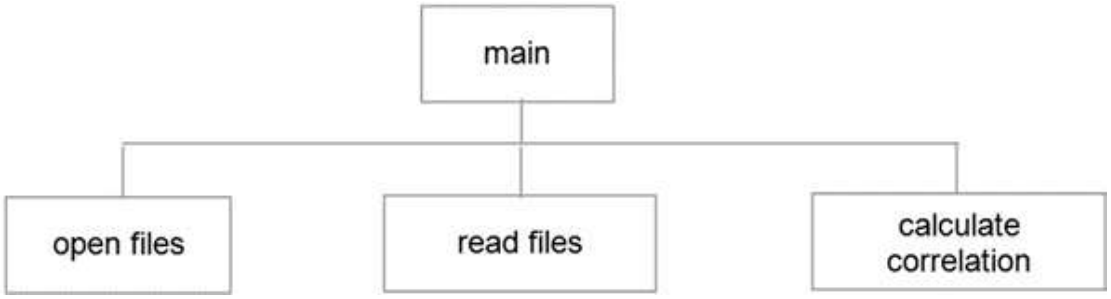
character. Therefore, this line (and the rest of the code) is placed within a try-except block (within **lines 25–45**). Thus, if a non-digit is entered, the raised exception is caught by the except ValueError clause, and the message 'Please enter numerical value 1 or 2' is displayed (at **line 46**). Since valid_input is still False in this case, the while loop performs another iteration, again prompting the user for their selection.

Because a user may enter a numerical value other than 1 or 2, the while loop at **line 29**catches such errors and immediately re-prompts the user. Once the loop terminates, the input is known to be valid. Therefore, based on the selection, either function executeScheduledSimulation or function executeUnscheduleSimulation is called, and valid_input is set to True causing the outer while loop at **line 24** to terminate.

Test Drivers and Test Stubs

In the development of the smoking/lung cancer correlation program of Chapter 8, we utilized unit testing, testing each function separately by developing an appropriate test driver for each. This was followed by integration testing, which tested the complete program with all functions working together.

We use unit testing and integration testing for this program as well. However, in the smoking/ lung cancer program, there were only three functions, each directly called by the main module as given below.



Since these functions did not call any other function of the program, they were simply individually implemented and tested. However, based on the design of the current program, there are functions that call other functions. Thus, these functions cannot be tested without having some version of the other functions to call. This is true for function executeScheduledSimulation for example, shown in Figure 9-20.

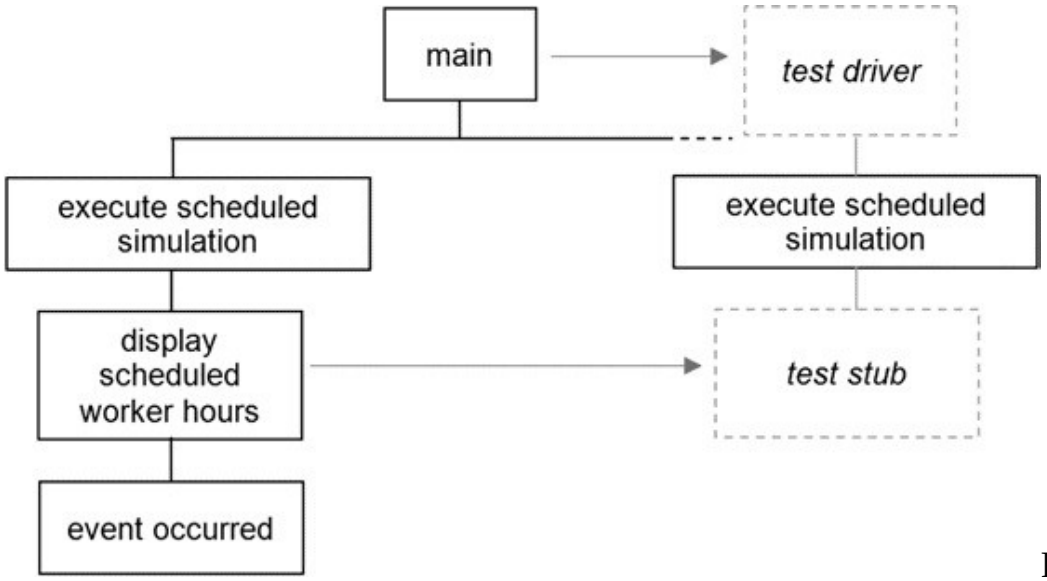


FIGURE 9-20 Unit Testing of Function executeScheduledSimulation

As shown in the figure, both a test driver and a test stub are needed for testing the function. A *test driver*, as we saw in Chapter 8, is code developed simply for calling and testing a given function. A *test stub*, on the other hand, is a simple, incomplete implementation of a function for testing functions that call it. For example, a test stub for a function designed to calculate the GPA of a given student might be implemented to simply return an arbitrary floating-point value, *as if* it were a calculated result. Thus, test drivers and test stubs are developed for testing purposes only, and do not become part of the ultimate implementation.

Unit Testing Function **executeScheduledSimulation**
We begin with the unit testing of function executeScheduledSimulation, given in Figure 9-21.

```
def executeScheduledSimulation(probabilities, days, time_slots):

    """Displays a simulated week of workers in attendance for all
       time slots, based on a scheduled worker schedule.
    """

    # for each day of the week
    for day in days:
        print(day)
        print('-------')

        if (day == 'Sunday'):  # sunday?
            current_timeslot = '12:00pm'
            num_timeslots = 3
        elif day in days[1:5]: # mon-thurs?
            current_timeslot = '8:00am'
            num_timeslots = 5
        else:  # friday, saturday
            current_timeslot = '8:00am'
            num_timeslots = 6

        # find loc of current_timeslot in tuple time_slots
        index_val = time_slots.index(current_timeslot)

        # iterate through num_timeslots starting with current time slot
        for time_slot in time_slots[index_val:num_timeslots]:
            print('{0:>7}'.format(time_slot), ' ', end='')

            for k in range(0,2):
                displayScheduledWorkerHours(k+1, probabilities)
                print()
            print()
```

FIGURE 9-21 Implementation of Function executeScheduledSimulation

We develop a test driver that calls function executeScheduledSimulation with values for parameters probabilities, days, and time_slots defined in the main module, given in Figure 9-22.

In addition, since the function depends on the use of function displayScheduledWorkerHours, a test stub is developed for this function for executeScheduled Simulation to call, given in Figure 9-23. All this test stub does is display the value of workernum. The time slots and day of the week are displayed by function execute ScheduledSimulation. Finally, we put function

```
# TEST DRIVER for Function executeScheduledSimulation for Food Co-op Program

from executescheduledsimulation import *

# init required data
sched_probabilities = {'CLate_15':15, 'CLate_30':5, 'CLate_45':2,
                       'CLEarly_15':5, 'CLEarly_30':3,'CNoshow':15}

days = ('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',
        'Friday', 'Saturday')

time_slots = ('8:00am', '10:00am', '12:00pm', '2:00pm', '4:00pm',
              '6:00pm', '8:00pm')

# call function
executeScheduledSimulation(sched_probabilities, days, time_slots)
```

FIGURE 9-22 Test Driver for Function executeScheduledSimulation

```
# TEST STUB for Function displayScheduledWorkerHours for Food Co-op Program

def displayScheduledWorkerHours(workernum, probabilities):

    if workernum == 1:
        print('workernum = ', workernum, end=' ')
    else:
        print('workernum = ', workernum)
```

FIGURE 9-23 Test Stub for Function displayScheduledWorkerHours

execute ScheduledSimulation in its own file named executescheduledsimulation. py, as well as the test driver and test stub to be used as modules. Since module executescheduledsimulation needs to call function stub displayScheduledWorkerHours, it contains an import statement of the form from displayscheduledworkerhours import *. Also, since the driver makes a call to function executeScheduledSimulation, it contains an import statement of the form from executescheduledsimulation import *. The result of the testing is given in Figure 9-24.

The output shows the scheduled time slots for each day of the week. However, we see an error for the time slots for Sundays. Only one time slot is displayed (for 8:00 am) but there should be three (8:00 am, 10:00 am, and 12:00 pm). To track down this error, we do some interactive testing in the Python shell. First, we display the value of variable days to make sure it contains each weekday,

... days
('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', ' Saturday')

This is correct. We also check the value of variable time_slots,
... time_slots
('8:00am', '10:00am', '12:00pm', '2:00pm', '4:00pm', '6:00pm')

```
Sunday
-------
12:00pm  workernum =  1 workernum =  2

Monday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2

Tuesday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2

Wednesday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2

Thursday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2

Friday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2
 6:00pm  workernum =  1 workernum =  2

Saturday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2
 6:00pm  workernum =  1 workernum =  2
>>>
```

FIGURE 9-24 Unit Testing Results for Function displayScheduledWorkerHours

This is also correct. We then consider the part of function executeScheduledSimulation that determines the time slots for each day,

```
for day in days:
print(day)
print('-------')

if (day 55 'Sunday'): # sunday?
current_timeslot 5 '12:00pm'
num_timeslots 5 3

elif day in days[1:5]: # mon–thurs?
current_timeslot 5 '8:00am'
num_timeslots 5 5

else: # friday, saturday
current_timeslot 5 '8:00am'
num_timeslots 5 6

# find loc of current_timeslot in tuple time_slots
index_val 5 time_slots.index(current_timeslot)
# iterate through num_timeslots starting with current time slot for time_slot in
time_slots[index_val:num_timeslots]: print('{0: .7}'.format(time_slot), ' ', end 5
")
```

We see that variable num_timeslots is set to 3 for Sundays. Since that seems correct, we see what happens in the following code. Variable index is set to the index value of the first occurrence of the first time slot for the day, held by variable current_timeslot. Since the value of current_timeslot should be '12:00pm', we call method index to be sure of the index value returned,

```
... index_val 5 time_slots.index('12:00pm') ... index_val
2
```

This is correct. We therefore focus on the for loop,

```
for time_slot in time_slots[index_val: num_timeslots]: print('{0:
.7}'.format(time_slot), ' ', end 5 ")
```

For all other days, this loop iterates the correct number of times, once for each time slot of each day. We check to see the slice of time_slots that is returned specifically for index_value equal to 2 and num_timeslots equal to 3.

... time_slots[2:3] ('12:00pm',)

Ah, there is a problem! The result should have been ('12:00pm', '2:00pm', '4:00pm'). But why does it work correctly for all the other days of the week? We see that Sunday is the only day that the first time slot does not begin at 8:00 am. Thus, all other slices of list time_slots are of the form,

time_slots[0:k]

whereas for Sunday, it is of the form, time_slots[3:k]

It must come down to how the index values are used in the slice notation. We checked on this and found that we had incorrectly used the slice operation. The first index value is the starting location of the slice, and the second index value is one greater than the last index of the substring. We had written the code as if the second index indicated *how many* elements to include. So the actual code should be,

for time_slot in time_slots[index_val: **index_val** 1 **num_timeslots**]: print('{0: .7}'.format(time_slot), ' ', end 5")

It worked for all the other days of the week because in those cases, index_val was 0, and therefore index_val 1num_timeslots was equal to num_timeslots. So that explains everything! We find a similar error in function executeUnscheduledSimulation, so we make the correction there as well.

We should feel good about ourselves. Not only did we find the error, but we are able to fully explain why this error created the incorrect output. As we have said, making a change that fixes a problem without knowing why the change fixes it is disconcerting and risky. We therefore make this correction to executeScheduledSimulation and again perform unit testing, this time getting the expected output for Sunday (as well as the rest of the days).

```
Sunday
-------
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2

Monday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2

Tuesday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2

Wednesday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2

Thursday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2
```

```
Friday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2
 6:00pm  workernum =  1 workernum =  2

Saturday
-------
 8:00am  workernum =  1 workernum =  2
10:00am  workernum =  1 workernum =  2
12:00pm  workernum =  1 workernum =  2
 2:00pm  workernum =  1 workernum =  2
 4:00pm  workernum =  1 workernum =  2
 6:00pm  workernum =  1 workernum =  2
>>>
```

We leave as an exercise the unit testing of the remaining program functions.

Integration Testing of the Food Co-op Program
Assuming each function of the program has been unit tested, we next perform integration testing. The complete program is given in Figure 9-25.

The main section of the program is in **lines 213–267**. This part of the program has already been discussed. The displayScheduledWorkerHours, displayUnscheduledWorkerHours, executeScheduledSimulation, and executeUnscheduledSimulation functions contain the bulk of the implementation.

Function executeScheduledSimulation (**lines 135–165**) consists of three nested for loops. The outer loop (**line 142**) iterates through each of the days in the schedule, in this simulation, all seven days. The next inner for loop (**line 160**) iterates through each of the time slots for the day. **Lines 146–154** initialize the first time slot (current_timeslot) as well as the number of time slots (num_timeslots) based on the current value of day. Variable time_slots passed to the function contains all the possible time slots for all the days that the co-op is open. In order to find that first time slot for the current day within this list time_slots, the index method is used (**line 157**), as discussed in the unit testing of this function.

Finally, for each time slot, we iterate over the number of workers for the slot in

the innermost for loop in **lines 162–164**. In this case, the for loop iterates exactly two times since, in the scheduled worker approach, that is the number of workers assigned to each time slot. It is within this innermost loop that function displayScheduledWorkerHours is called (**line 163**). The argument in the function call is adjusted by 1 (k +), because function displayScheduledWorkerHours expects to be passed worker numbers starting at 1.

Function displayScheduledWorkerHours (**lines 38–90**) generates the events that may occur for a given scheduled worker; specifically, whether the worker arrives late and/or leaves early or does not show up at all. Thus, at the start of the function (in **lines 57–58**), variables mins_late and mins_left_early are initialized to zero. Following that is an if statement (**line 60**) determining if the (scheduled) worker does not show up for their assigned time slot. The probability of this occurring is contained in the dictionary probabilities. If the event is found to be true, then mins_late is set to the special value 21, to be used later on **line 80**.

```
# Food Co-op Simulation Program

import random

# Probabilties for Simulation (for values between 1-100)
#
# CLate_xx = chance of being late for work,
# CLEarly_xx = chance of leaving early from work,
# CNoshow = chance that a scheduled worker WILL NOT show up for work
# CShowup = chance that an unscheduled worker WILL show up to work


def eventOccurred(chances):

    """For given integer value chances as a percentage value (1-100),
       returns True if randomly generated number in range (1,100) is
       less than or equal to chances, otherwise returns False.
    """
    if random.randint(1,100) <= chances:
        return True
    else:
        return False

def numWorkersShowedUp(indiv_chance, num_individuals):

    """For given integer values indiv_chance and num_individuals,
       returns how many times that num_individual calls to
       event_occurred with argument indiv_chance returns True.
    """
    numworkers_arriving = 0

    for i in range(num_individuals):
        if eventOccurred(indiv_chance):
            numworkers_arriving = numworkers_arriving + 1

    return numworkers arriving

def displayScheduledWorkerHours(workernum, probabilities):

    """For workernum equal to 1 or 2, and the proability values
       given in dictionary probabilities of the form,

          {'CLate_15':<num>, 'CLate_30':<num>, 'CLate_45':<num>,
           'CLEarly_15':<num>, 'CLEarly_30':<num>,'CNoshow':<num>}


       where each <num> is a value between 1-100, displays one of

         worker <worker_num> -- no show --   or
         worker <worker_num> <mins_late> mins late  and/or
         worker <worker_num> left <mins_early> mins early

       where <mins_late> is 15, 30, or 45,  <mins_early> is 15 or 30.
    """

```

```
    # init
    mins_late = 0
    mins_left_early = 0

    if eventOccurred(probabilities['CNoshow']):
        mins_late = -1
    else:
        if eventOccurred(probabilities['CLate_15']):
            mins_late = 15
        elif eventOccurred(probabilities['CLate_30']):
            mins_late = 30
        elif eventOccurred(probabilities['CLate_45']):
            mins_late = 45

        if eventOccurred(probabilities['CLEarly_15']):
            mins_left_early = 15
        elif eventOccurred(probabilities['CLEarly_30']):
            mins_left_early = 30

    if workernum == 1:
        print('{0:>3}'.format('worker'), str(workernum) + ': ', end='')
    else:
        print('{0:>15}'.format('worker'), str(workernum) + ': ', end='')

    if mins_late == -1:
        print('-- no show --', end='')
    else:
        if mins_late != 0:
            print(mins_late, 'mins late  ', end='')

        if mins_left_early != 0:
            print('left', mins_left_early, 'mins early', end='')

        if (mins_late == 0) and (mins_left_early == 0):
            print('whole time', end='')

def displayUnscheduledWorkerHours(workernum, probabilities):

    """For workernum equal to 1 or 2, and the proability values
       given in dictionary probabilities of the form,

         {'CLate_15':<num>, 'CLate_30':<num>, 'CLate_45':<num>,
          'CLEarly_15':<num>, 'CLEarly_30':<num>,'CShowup':<num>}


       where each <num> is a value between 1-100, displays one of

         worker <worker_num> -- no show --   or
         worker <worker_num> <mins_late> mins late  and/or
         worker <worker_num> left <mins_early> mins early

       where <mins_late> is 15 or 30,
             <mins_early> is 15 or 30.
    """
```

```
    mins_late = 0   # init
    mins_left_early = 0

    if eventOccurred(probabilities['CLate_15']):
        mins_late = 15
    elif eventOccurred(probabilities['CLate_30']):
        mins_late = 30
        elif eventOccurred(probabilities['CLate_45']):
            mins_late = 45

    if eventOccurred(probabilities['CLEarly_15']):
        mins_left_early = 15
    elif eventOccurred(probabilities['CLEarly_30']):
        mins_left_early = 30

    print('{0:>15}'.format('worker'), str(workernum) + ': ', end='')
    if mins_late != 0:
        print(mins_late, 'mins late  ', end='')

    if mins_left_early != 0:
        print('left', mins_left_early, 'mins early', end='')

    if (mins_late == 0) and (mins_left_early == 0):
        print('whole time', end='')

def executeScheduledSimulation(probabilities, days, time_slots):

    """Displays a simulated week of workers in attendance for all
       time slots, based on a scheduled worker schedule.
    """

    # for each day of the week
    for day in days:
        print(day)
        print('-------')

        if (day == 'Sunday'):  # sunday?
            current_timeslot = '12:00pm'
            num_timeslots = 3
        elif day in days[1:5]: # mon-thurs?
            current_timeslot = '8:00am'
            num_timeslots = 5
        else:  # friday, saturday
            current_timeslot = '8:00am'
            num_timeslots = 6

        # find loc of current_timeslot in tuple time_slots
        index_val = time_slots.index(current_timeslot)

        # iterate through num_timeslots starting with current time slot
        for time_slot in time_slots[index_val:index_val + num_timeslots]:
            print('{0:>7}'.format(time_slot), ' ', end='')
            for k in range(0,2):
                displayScheduledWorkerHours(k+1, probabilities)
                print()
            print()
```

FIGURE 9-25 Food Co-op Simulation Program (*Continued*)

If the “no show” event does not occur, the simulation checks events for when the volunteers are 15, 30, or 45 minutes late (**lines 63–68**). Because showing up late and leaving early are independent events, the program also checks for the particular worker leaving 15 or 30 minutes early (**lines 70–73**). The remaining lines of function display_scheduledworkerhours

```
167 def executeUnscheduledSimulation(probabilities, days, time_slots):
168
169     """Displays a simulated week of workers in attendance for all
170        time slots, based on an unscheduled worker schedule.
171     """
172
173     # for each day in the week
174     for day in days:
175         print(day)
176         print('-------')
177
178         if (day == 'Sunday'):   # sunday
179             current_timeslot = '12:00pm'
180             num_timeslots = 3
181         elif day in days[1:5]: # mon-thurs?
182             current_timeslot = '8:00am'
183             num_timeslots = 5
184         else:  # fri, sat
185             current_timeslot = '8:00am'
186             num_timeslots = 6
187
188         # find loc of current_timeslot in tuple time_slots
189         index_val = time_slots.index(current_timeslot)
190
191         # iterate through num_timeslots starting with current time slot
192         for time_slot in time_slots[index_val:index_val + num_timeslots]:
193             numworkers = \
194              numWorkersShowedUp(probabilities['CShowup'], num_members)
195             print('{0:>7}'.format(time_slot), ' ', end='')
196
197             if numworkers == 1:
198                 print(numworkers, 'worker came')
199                 num_stayed = 1
200             elif numworkers == 0 or numworkers == 2:
201                 print(numworkers, 'workers came')
202                 num_stayed = numworkers
203             else:
204                 print(numworkers, 'workers came (' + \
205                 str(numworkers - 2), 'went home)' )
206                 num_stayed = 2
207
208             for k in range(num_stayed): # at most 2 workers stay to work
209                 displayUnscheduledWorkerHours(k+1, probabilities)
210
211             print()
212
```

FIGURE 9-25 Food Co-op Simulation Program (*Continued*)

(**lines 75–90**) involve checking which events occurred and displaying (and properly spacing) the appropriate output.

Functions executeUnscheduledSimulation (**lines 167–211**) and its supporting function displayUnscheduledWorkerHours (**lines 92–133**) are very similar to the corresponding functions executeScheduledSimulation and displayScheduled WorkerHours. One difference is the way that the simulation handles the number of possible workers for a given time slot in function executeUnscheduledSimulation. Because in the unscheduled approach any number of workers can show up, this section indicates how many workers actually stay to work that time

```
213 # ---- main
214
215 # init
216 num_members = 75
217 time_slots = ('8:00am', '10:00am', '12:00pm', '2:00pm', '4:00pm',
218               '6:00pm')
219
220 days = ('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',
221         'Friday', 'Saturday')
222
223 sched_probabilities = {'CLate_15':15, 'CLate_30':5, 'CLate_45':2,
224                        'CLEarly_15':5, 'CLEarly_30':3,'CNoshow':15}
225
226 unsched_probabilities = {'CLate_15':5, 'CLate_30':2, 'CLate_45':1,
227                          'CLEarly_15':10, 'CLEarly_30':3,'CShowup':5}
228
229 # seed random number generator with system clock
230 random.seed()
231
232 # get type of simulation
233 print('Welcome to the Food Co-op Schedule Simulation Program')
234
235 valid_input = False
236 while not valid_input:
237     try:
238         response = int(input('(1)scheduled, (2)unscheduled simulation? '))
239
240         while (response != 1) and (response != 2):
241             print('Invalid Selection\n')
242             response = \
243                 int(input('(1)scheduled, (2)unscheduled simulation? '))
244
245         if response == 1:
246             print('<< SCHEDULED WORKER SIMULATION >>\n')
247             executeScheduledSimulation(sched_probabilities,
248                                        days, time_slots)
249         else:
250             print('<< UNSCHEDULED WORKER SIMULATION >>\n')
251             executeUnscheduledSimulation(unsched_probabilities,
252                                          days, time_slots)
253
254         valid_input = True
255
256     except ValueError:
257         print('Please enter numerical value 1 or 2\n')
```

FIGURE 9-25 Food Co-op Simulation Program

slot—no more than two—and how many go home. Another minor difference is the way that displayUnscheduledWorkerHours is designed. This is due to the fact that a “no show” only applies to scheduled workers, and not to unscheduled workers since unscheduled workers never sign up for a time slot to work.

Function event_occurred (**lines 13–22**) is a simple function that returns a Boolean (True/False) value used to simulate whether a given event has occurred or not. The function is passed an integer value between 1 and 100 representing the probability of the event (where 100 represents certainty). It then makes a call to the random number generator to randomly generate an integer in the same range of 1 to 100 (in **line 19**). If the number is less than or equal to the value passed in parameter chance, then the event is assumed to have occurred, and the value True is returned. Otherwise, the event is assumed *not* to have occurred, and thus returns False. This function is what drives the simulation.

Finally , function numworkersShowedUp (**lines 24–36**) is a supporting function, called from **line 194** of function executeUnscheduledSimulation. In the simulation of the unscheduled approach, any number of members may show up for work for any given time slot. To simulate this, function numworkershowedup is designed to be given the chances that an individual member may show up (in parameter indiv_chance), as well as the total number of members in the co-op (in parameter num_individuals).

9.3.5 Analyzing a Scheduled vs. Unscheduled Co-op Worker Approach We now look at the simulation results for worker coverage at the food co-op, one using a scheduled approach (Figure 9-26), and the other using an unscheduled approach (Figure 9-27).

We can compare these two simulation runs. Since each run is based on assumed probabilities of events, the results are only as accurate as the probability estimates. Also, multiple simulation runs would provide a more accurate picture of likely events. Comparative results are given in Figure 9-28.

So with these caveats in mind , what can we conclude from this simulation? We first look at the reason that the food co-op chose to try an unscheduled approach —that too often, time slots were left partially or completely uncovered. From our simulation of a scheduled approach, we do see three times in which time slots

are completely uncovered, and fourteen times in which there is only partial coverage (that is, only one person showing up). In the simulation of the unscheduled approach, there were no time slots left uncovered, and only three time slots in which there was partial coverage. This, therefore, coincides with the co-op's experience that there is better coverage in an unscheduled approach.

We next focus on the number of times a worker was 15 or 30 minutes late. In the scheduled approach, members were 15 minutes late six times, 30 minutes late two times, and 45 minutes late one time. For the unscheduled approach, members were 15 minutes late six times, 30 minutes late three times, and never 45 minutes late. Thus, there is no appreciable difference in the amount of time workers arrive late based on the simulation. When we look at the number of times workers left early, we see that in the scheduled approach, workers left 15 minutes early three times and 30 minutes early four times. In the unscheduled approach, workers left 15 minutes early four times, and 30 minutes early only once. Thus, there is also no appreciable difference in the amount of times workers leave early, except that scheduled workers were more likely to leave 30 minutes early than unscheduled workers (based on our assumed probabilities of human behavior).

Overall, the results indicate that an unscheduled approach does improve the problem of uncovered time slots. This, therefore, improves the functioning of the co-op. However, is there a hidden cost to this? A downside of the unscheduled approach is that there are times when more than the

```
Welcome to the Food Co-op Schedule Simulation
Program
(1)scheduled, (2)unscheduled simulation? 1
<< SCHEDULED WORKER SIMULATION >>

Sunday
-------
12:00pm  worker 1: whole time
         worker 2: whole time

 2:00pm  worker 1: whole time
         worker 2: -- no show --

 4:00pm  worker 1: whole time
         worker 2: -- no show --

Monday
-------
 8:00am  worker 1: -- no show --
         worker 2: whole time

10:00am  worker 1: whole time
         worker 2: whole time

12:00pm  worker 1: -- no show --
         worker 2: -- no show --

 2:00pm  worker 1: left 30 mins early
         worker 2: whole time

 4:00pm  worker 1: whole time
         worker 2: whole time

Tuesday
-------
 8:00am  worker 1: -- no show --
         worker 2: -- no show --

10:00am  worker 1: whole time
         worker 2: whole time

12:00pm  worker 1: whole time
         worker 2: 15 mins late

 2:00pm  worker 1: whole time
         worker 2: -- no show --

 4:00pm  worker 1: whole time
         worker 2: -- no show --

Wednesday
-------
 8:00am  worker 1: whole time
         worker 2: whole time

10:00am  worker 1: 30 mins late
         worker 2: 30 mins late  left 15 mins early

12:00pm  worker 1: 45 mins late  left 30 mins early
         worker 2: whole time

 2:00pm  worker 1: left 15 mins early
         worker 2: 30 mins late

 4:00pm  worker 1: -- no show --
         worker 2: whole time

Thursday
-------
 8:00am  worker 1: whole time
         worker 2: -- no show --

10:00am  worker 1: -- no show --
         worker 2: whole time

12:00pm  worker 1: whole time
         worker 2: 15 mins late

 2:00pm  worker 1: whole time
         worker 2: whole time

 4:00pm  worker 1: whole time
         worker 2: 15 mins late

Friday
-------
 8:00am  worker 1: -- no show --
         worker 2: whole time

10:00am  worker 1: 15 mins late
         worker 2: -- no show --

12:00pm  worker 1: whole time
         worker 2: -- no show --

 2:00pm  worker 1: whole time
         worker 2: left 30 mins early

 4:00pm  worker 1: 15 mins late
         worker 2: whole time

 6:00pm  worker 1: whole time
         worker 2: whole time

Saturday
-------
 8:00am  worker 1: whole time
         worker 2: -- no show --

10:00am  worker 1: whole time
         worker 2: left 30 mins early

12:00pm  worker 1: 15 mins late
         worker 2: whole time

 2:00pm  worker 1: left 15 mins early
         worker 2: -- no show --

 4:00pm  worker 1: -- no show --
         worker 2: -- no show --

 6:00pm  worker 1: whole time
         worker 2: -- no show --
```

FIGURE 9-26 Scheduled Worker Simulation

```
Welcome to the Food Co-op Schedule Simulation
Program
(1)scheduled, (2)unscheduled simulation? 2
<< UNSCHEDULED WORKER SIMULATION >>

Sunday
-------
12:00pm  7 workers came (5 went home)
         worker 1: 15 mins late
         worker 2: whole time

 2:00pm  1 worker came
         worker 1: whole time

 4:00pm  5 workers came (3 went home)
         worker 1: whole time
         worker 2: whole time

Monday
-------
 8:00am  7 workers came (5 went home)
         worker 1: whole time
         worker 2: whole time

10:00am  4 workers came (2 went home)
         worker 1: whole time
         worker 2: whole time

12:00pm  4 workers came (2 went home)
         worker 1: whole time
         worker 2: whole time

 2:00pm  4 workers came (2 went home)
         worker 1: left 15 mins early
         worker 2: left 30 mins early

 4:00pm  4 workers came (2 went home)
         worker 1: 15 mins late
         worker 2: whole time

Tuesday
-------
 8:00am  4 workers came (2 went home)
         worker 1: whole time
         worker 2: 15 mins late

10:00am  5 workers came (3 went home)
         worker 1: 15 mins late
         worker 2: whole time

12:00pm  3 workers came (1 went home)
         worker 1: whole time
         worker 2: whole time

 2:00pm  4 workers came (2 went home)
         worker 1: 15 mins late
         worker 2: whole time

 4:00pm  1 worker came
         worker 1: whole time

Wednesday
-------
 8:00am  4 workers came (2 went home)
         worker 1: left 15 mins early
         worker 2: whole time

10:00am  3 workers came (1 went home)
         worker 1: whole time
         worker 2: left 15 mins early

12:00pm  2 workers came
         worker 1: whole time
         worker 2: left 15 mins early

 2:00pm  6 workers came (4 went home)
         worker 1: whole time
         worker 2: whole time

 4:00pm  6 workers came (4 went home)
         worker 1: whole time
         worker 2: whole time

Thursday
-------
 8:00am  6 workers came (4 went home)
         worker 1: whole time
         worker 2: whole time

10:00am  3 workers came (1 went home)
         worker 1: whole time
         worker 2: whole time

12:00pm  6 workers came (4 went home)
         worker 1: whole time
         worker 2: 30 mins late

 2:00pm  6 workers came (4 went home)
         worker 1: whole time
         worker 2: whole time

 4:00pm  5 workers came (3 went home)
         worker 1: whole time
         worker 2: whole time

Friday
-------
 8:00am  4 workers came (2 went home)
         worker 1: whole time
         worker 2: whole time

10:00am  3 workers came (1 went home)
         worker 1: whole time
         worker 2: whole time

12:00pm  3 workers came (1 went home)
         worker 1: whole time
         worker 2: whole time

 2:00pm  4 workers came (2 went home)
         worker 1: whole time
         worker 2: 30 mins late
```

FIGURE 9-27 Unscheduled Worker Simulation (*Continued*)