((abs(expected_results – actual_results) – 86,400) / expected_results) * 100 5 ((917,110,352 – 518,433) 2 86400) / 917,110,352) * 100 5 99.93 %

Either our algorithmic approach is flawed, or it is not correctly implemented. Since we didn't find any errors in the development of the first and second stages of the program, the problem must be in the calculation of the approximate age in **lines 29–37**. These lines define three variables: numsecs_1900_dob, numsecs_1900_today, and age_in_secs. We can inspect the values of these variables after execution of the program to see if anything irregular pops out at us.

This program computes the approximate age in seconds of an individual based on a provided date of birth. Only ages for dates of birth from 1900 and after can be computed
Enter month born (1-12): 4
Enter day born (1-31): 12
Enter year born: (4-digit)1981
You are approximately 604833 seconds old
...
**... numsecs_1900_dob**
2**59961031015**
**... numsecs_1900_today**
2**59960426182**
**...**

Clearly, this is where the problem is, since we are getting negative values for the times between 1900 and date of birth, and from 1900 to today. We "work backwards" and consider how the expressions could give negative results. This would be explained if, for some reason, the second operand of the subtraction were greater than the first. That would happen if the expression were evaluated, for example, as

numsecs_1900_dob 5 (year_birth 2 **(1900 * avg_numsecs_year)**) 1 \ (month_birth 2 **(1 * avg_numsecs_month)**) 1 \ (day_birth * numsecs_day)

rather than the following intended means of evaluation,

numsecs_1900_dob 5 ( **(year_birth 2 1900)** * avg_numsecs_year) 1 \ ( **(month_birth 2 1)** * avg_numsecs_month) 1 \ (day_birth * numsecs_day)

Now we realize! Because we did not use parentheses to explicitly indicate the proper order of operators, by the rules of operator precedence Python evaluated the expression as the first way above, not the second as it should be. This would also explain why the program gave the correct result for a date of birth one day before the current date. Once we make the corrections and re-run the test plan, we get the following results shown in Figure 2-29.

| Date of Birth | Expected Results | Actual Results | Inaccuracy |
|---|---|---|---|
| January 1, 1900 | 3,520,023,010 ± 86,400 | 3,520,227,600 | < .004 % |
| April 12, 1981 | 955,156,351 ± 86,400 | 955,222,200 | 0 % |
| January 4, 2000 | 364,090,570 ± 86,400 | 364,208,400 | < .009 % |
| December 31, 2009 | 48,821,332 ± 86,400 | 48,929,400 | < .05 % |
| (day before current date) | 86,400 ± 86,400 | 86,400 | 0 % |

FIGURE 2-29 Results of Second Execution of Test Plan

These results demonstrate that our approximation of the number of seconds in a year was sufficient to get very good results, well within the 99% degree of accuracy required for this program. We would expect more recent dates of birth to give less accurate results given that there is less time that is approximated. Still, for test case December 31, 2009 the inaccuracy is less than .05 percent. Therefore, we were able to develop a program that gave very accurate results without involving all the program logic that would be needed to consider all the details required to give an exact result.

CHAPTER SUMMARY General Topics

Numeric and String Literals
Limitations of Floating-Point Representation Arithmetic Overflow and Underflow
Character Representation Schemes

(Unicode/ASCII)
Control Characters

String Formatting Implicit and Explicit Line

Joining/Variables and Variable Use/
Keyboard Input/Identifier Naming/ Keywords Arithmetic
Operators/Expressions/Infix Notation Operator Precedence and Associativity
Data Types/Static vs. Dynamic Typing
Mixed-Type Expressions/Coercion and
Type Conversion

Python-Specific Programming Topics

Numeric Literal and String Literal Values in Python Built-in format Function in Python
Variable Assignment and Storage in Python Immutable Values in Python
Identifier Naming and Keywords in Python Arithmetic Operators in Python
Operator Precedence and Associativity in Python Built-in int() and float() Type Conversion

Functions in Python

CHAPTER EXERCISES Section 2.1
**1.** Based on the information in Figure 2-1, how many novels can be stored in one terabyte of storage?

**2.** Give the following values in the exponential notation of Python, such that there is only one significant digit to the left of the decimal point.
**(a)**4580.5034 **(b)**0.00000046004 **(c)** 5000402.000000000006

**3.** Which of the floating-point values in question 2 would exceed the representation of the precision of floating points typically supported in Python, as mentioned in the chapter?
**4.** Regarding the built-in format function in Python,
**(a)** Use the format function to display the floating-point value in a variable named result with three decimal digits of precision.
**(b)** Give a modified version of the format function in (a) so that commas are included in the displayed results.
**5.** Give the string of binary digits that represents, in ASCII code,
**(a)** The string 'Hi!'
**(b)** The literal string 'I am 24'
**6.** Give a call to print that is provided one string that displays the following

address on three separate lines. John Doe
123 Main Street
Anytown, Maryland 21009

**7.** Use the print function in Python to output It's raining today.
Chapter Exercises 75
Section 2.2

**8.** Regarding variable assignment,
**(a)** What is the value of variables num1 and num2 after the following instructions are executed? num 5 0
k 5 5
num1 5 num 1 k * 2
num2 5 num 1 k * 2

**(b)** Are the values id(num1) and id(num2) equal after the last statement is executed? **9.** Regarding the input function in Python,

**(a)** Give an instruction that prompts the user for their last name and stores it in a variable named last_ name.
**(b)** Give an instruction that prompts the user for their age and stores it as an integer value named age.
**(c)** Give an instruction that prompts the user for their temperature and stores it as a float named current_temperature.

**10.** Regarding keywords and other predefined identifiers in Python, give the result for each of the following, **(a)** 'int' in dir(__builtins__)
**(b)**'import' in dir(__builtins__)

Section 2.3
**11.** Which of the following operator symbols can be used as both a unary operator and a binary operator?

1 , 2, *, /
**12.** What is the exact result of each of the following when evaluated?
**(a)** 12 / 6.0
**(b)**21 // 10
**(c)** 25 // 10.0
**13.** If variable n contains an initial value of 1, what is the largest value that will be assigned to n after the

following assignment statement is executed an arbitrary number of times?
n 5 (n 1 1) % 100
**14.** Which of the following arithmetic expressions could potentially result in arithmetic overflow, where n
and k are each assigned integer values?
**(a)** n * k **(b)** n ** k **(c)** n / k **(d)** n 1 k

Section 2.4

**15.** Evaluate the following expressions in Python.
**(a)** 10 2 (5 * 4)
**(b)**40 % 6
**(c)** 2 (10 / 3) 1 2

**16.** Give all the possible evaluated results for the following arithmetic expression (assuming no rules of operator precedence).

2 * 4 1 25 – 5
**17.** Parenthesize all of the subexpressions in the following expressions following operator precedence in Python.
**(a)** var1 * 8 2 var2 1 32 / var3
**(b)**var1 2 6 ** 4 * var2 ** 3

**18.** Evaluate each of the expressions in question 17 above for var1 5 10, var2 5 30, and var3 5 2. **19.** For each of the following expressions, indicate where operator associativity of Python is used to resolve ambiguity in the evaluation of each expression.
**(a)** var1 * var2 * var3 2 var4
**(b)**var1 * var2 / var3
**(c)** var1 ** var2 ** var3

**20.** Using the built-in type conversion function float(), alter the following arithmetic expressions so that each is evaluated using floating-point accuracy. Assume that var1, var2, and var3 are assigned integer values. Use the minimum number of calls to function float() needed to produce the results. **(a)** var1 1 var2 * var3
**(b)**var1 // var2 1 var3
**(c)** var1 // var2 / var3

PYTHON PROGRAMMING EXERCISES

**P1.** Write a Python program that prompts the user for two integer values and displays the result of the first number divided by the second, with exactly two decimal places displayed.
**P2.** Write a Python program that prompts the user for two floating-point values and displays the result of the first number divided by the second, with exactly six decimal places displayed.
**P3.** Write a Python program that prompts the user for two floating-point values and displays the result of the first number divided by the second, with exactly six decimal places displayed in scientific notation.
**P4.** Write a Python program that prompts the user to enter an upper or lower case letter and displays the corresponding Unicode encoding.
**P5.** Write a Python program that allows the user to enter two integer values, and displays the results when each of the following arithmetic operators are applied. For example, if the user enters the values 7 and 5, the output would be,
7 1 5 5 12
7 2 5 5 2
7 * 5 5 35
7 / 5 5 1.40
7 // 5 5 1
7 % 5 5 2
7 ** 5 5 16,807

All floating-point results should be displayed with two decimal places of accuracy. In addition, all values should be displayed with commas where appropriate.
PROGRAM MODIFICATION PROBLEMS

**M1.** Modify the Restaurant Tab Calculation program of section 2.2.5 so that, instead of the restaurant tax being hard coded in the program, the tax rate is entered by the user.
**M2.** Modify the Restaurant Tab Calculation program of section 2.2.5 so that, in addition to displaying the total of the items ordered, it also displays the total amount spent on drinks and dessert, as well as the percentage of the total cost of the meal (before tax) that these items comprise. Display the monetary amount rounded to two decimal places.
**M3.** Modify the Your Place in the Universe program in section 2.3.3 for international users, so that the user enters their weight in kilograms, and not in pounds.
**M4.** Modify the Temperature Conversion program in section 2.4.6 to convert

from Celsius to Fahrenheit instead. The formula for the conversion is f 5 (c * 9/5) 1 32.

Program Development Problems 77

**M5.** Modify the Age in Seconds program so that it displays the estimated age in number of days, hours, and minutes.
**M6.** Modify the Age in Seconds program so that it determines the difference in age in seconds of two friends.

PROGRAM DEVELOPMENT PROBLEMS

**D1.** Losing Your Head over Chess
The game of chess is generally believed to have been invented in India in the sixth century for a ruling king by one of his subjects. The king was supposedly very delighted with the game and asked the subject what he wanted in return. The subject, being clever, asked for one grain of wheat on the first square, two grains of wheat on the second square, four grains of wheat on the third square, and so forth, doubling the amount on each next square. The king thought that this was a modest reward for such an invention. However, the total amount of wheat would have been more than 1,000 times the current world production.

Develop and test a Python program that calculates how much wheat this would be in pounds, using the fact that a grain of wheat weighs approximately 1/7,000 of a pound.

**D2.** All That Talking
Develop and test a Python program that determines how much time it would take to download all instances of every word ever spoken. Assume the size of this information as given in Figure 2-1. The download speed is to be entered by the user in million of bits per second (mbps). To find your actual connection speed, go to the following website (from Intel Corporation) or similar site,

www.intel.com/content/www/us/en/gamers/broadband-speed-test.html

Because connection speeds can vary, run this connection speed test three times. Take the average of three results, and use that as the connection speed to enter into your program. Finally, determine what is an appropriate unit of time to express your program results in: minutes? hours? days? other?

**D3.** Pictures on the Go
Develop and test a Python program that determines how many images can be stored on a given size USB (flash) drive. The size of the USB drive is to be entered by the user in gigabytes (GB). The number of images that can be stored must be calculated for GIF, JPEG, PNG, and TIFF image file formats. The program output should be formatted as given below.

Enter USB size (GB): 4

xxxxx images in GIF format can be stored xxxxx images in JPEG format can be stored xxxxx images in PNG format can be stored xxxxx images in TIFF format can be stored

The ultimate file size of a given image depends not only on the image format used, but also on the image itself. In addition, formats such as JPEG allow the user to select the degree of compression for the image quality desired. For this program, we assume the image compression ratios given below. Also assume that all the images have a resolution of 800 3 600 pixels.

Thus, for example, a 800 3 600 resolution image with 16-bit (2 bytes) color depth would have a total number of bytes of 800 3 600 3 2 5 960,000. For a compression rate of 25:1, the total number of bytes needed to store the image would be 960000/25 5 38400.

Finally, assume that a GB (gigabyte) equals 1,000,000,000 bytes, as given in Figure 2.1.

| Format | Full Name | Color Depth | | Compression | |
|---|---|---|---|---|---|
| GIF | Graphics Interchange Format | 256 colors | 8 bits | lossless | 5:1 |
| JPEG | Joint Photographic Experts Group | 16 million colors | 24 bits | lossy | 25:1 |
| PNG | Portable Network Graphics | 16 million colors | 24 bits | lossless | 8:1 |
| TIFF | Tagged Image File Format | 280 trillion colors | 48 bits | lossless | n/a |

Note that a “lossless” compression is one in which no information is lost. A “lossy” compression does lose some of the original information.

**D4.** Life Signs
Develop and test a program that prompts the user for their age and determines approximately how many breaths and how many heartbeats the person has had in their life. The average respiration (breath) rate of people changes during different stages of development. Use the breath rates given below for use in your program:

Breaths per Minute
Infant 30–60 1–4 years 20–30
5–14 years 15–25
adults 12–20

For heart rate, use an average of 67.5 beats per second.

## Control Structures CHAPTER 3

*In Chapter 2 we looked at the “nuts and bolts” of programming. In this chapter, we discuss the three fundamental means of controlling the order of execution of instructions within a program, referred to as sequential, selection, and iterative control.*

OBJECTIVES
After reading this chapter and completing the exercises, you will be able to: ♦ Explain what a control structure is
♦ Explain the difference between sequential, selection, and iterative control ♦ Describe and use Boolean operators
♦ Explain the notion of logically equivalent Boolean expressions ♦ Explain what is meant by an infinite loop
♦ Explain the difference between a definite and indefinite loop ♦ Explain the use of indentation in Python
♦ Effectively use if statements in Python for selection control ♦ Effectively implement multi-way selection in Python
♦ Effectively use while statements in Python for iterative control

CHAPTER CONTENTS
Motivation

Fundamental Concepts
3.1 What Is a Control Structure? 3.2 Boolean Expressions (Conditionals) 3.3 Selection Control
3.4 Iterative Control
Computational Problem Solving 3.5 Calendar Month Program

79
MOTIVATION

The first electronic computers over sixty years ago were referred to as “Electronic Brains.” This gave the misleading impression that computers could “think.” Although very complex in their design, computers are machines that simply do, step-by-step (instruction-by-instruction), what they are told. Thus, there is no more intelligence in a computer than what it is instructed to do.

What computers can do, however, is to exe cute a series of instructions very quickly and very

reliably. It is the speed in which instructions can be executed that gives computers their power (see Figure 3-1), since the execution of many simple instructions can result in very complex behavior. And thus this is the enticement of computing. A computer can accomplish any task for which there is an algorithm for doing so. The instructions could be for something as simple as sorting lists, or as ambitious as

performing intelligent tasks that as of now only humans are capable of performing.

In this chapter, we look at how to control the order that instructions are executed in Python.

| Term | Number of Floating Point Operations / Second | | Device |
|---|---|---|---|
| Megaflops | $10^6$ | 1 million FLOPS | Supercomputers (1970s) |
| Gigaflops | $10^9$ | 1 billion FLOPS | CPU (single core) |
| Teraflops | $10^{12}$ | 1 trillion FLOPS | CPU (multi-core) |
| Petaflops | $10^{15}$ | 1 quadrillion FLOPS | Supercomputers (current) |
| Exaflops | $10^{18}$ | 1 quintillion FLOPS | Supercomputers in 2020 (projected) |

FIGURE 3-1 Processing Speed—Floating-Point Operations per Second (FLOPS)

FUNDAMENTAL CONCEPTS

3.1 What Is a Control Structure?

C *ontrol flow* is the order that instructions are executed in a program. A **control statement** is a statement that determines the control flow of a set of instructions. There are three fundamental forms of control that programming languages provide— *sequential control*, *selection control*, and *iterative control.*

**Sequential control** is an implicit form of control in which instructions are executed in the order that they are written. A program consisting of only sequential control is referred to as a “straight-line program.” The program examples in Chapter 2 are all straight-line programs. **Selection control** is provided by a control statement that *selectively executes* instructions, while **iterative control** is provided by an iterative control statement that *repeatedly executes* instructions. Each is based on a given condition. Collectively a set of instructions and the control statements controlling their execution is called a **control structure**.

Few programs are straight-line programs. Most use all three forms of control, depicted in Figure 3-2. We look at selection control and iterative control next.

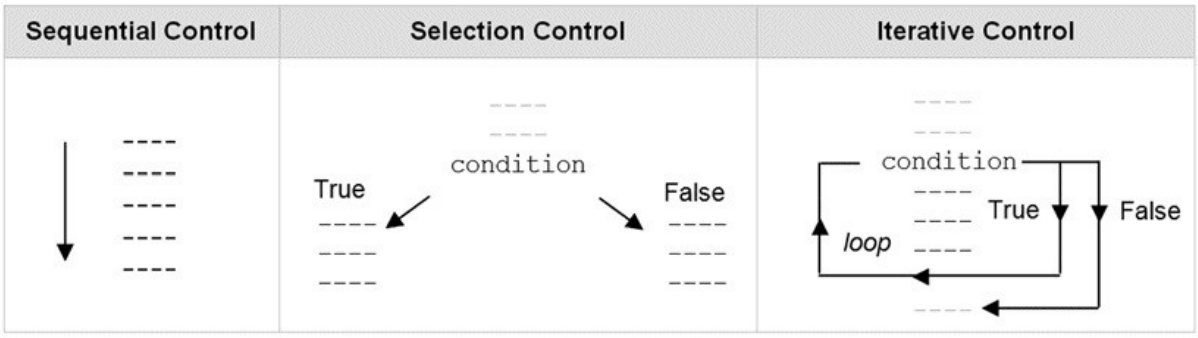


FIGURE 3-2 if Statement

A **control statement** is a statement that determines the control flow of a set of instructions. A **control structure** is a set of instructions and the control statements controlling their execution. Three fundamental forms of control in programming are **sequential**, **selection**, and **iterative control**.

3.2 Boolean Expressions (Conditions)

The **Boolean data type** contains two Boolean values, denoted as **True** and **False** in Python. A **Boolean expression** is an expression that evaluates to a Boolean value. Boolean expressions are used to denote the conditions for selection and iterative control statements. We look at the use of Boolean expressions next.

The **Boolean data type** contains two Boolean values, denoted as **True** and **False** in Python. A **Boolean expression** is an expression that evaluates to a Boolean value.

3.2.1 Relational Operators

The relational operators in Python perform the usual comparison operations, shown in Figure 3-3. Relational expressions are a type of Boolean expression, since they evaluate to a Boolean result. These operators not only apply to numeric values, but to any set of values that has an ordering, such as strings.

Note the use of the **comparison operator**, 55, for determining if two values are equal. This, rather than the (single) equal sign, 5, is used since the equal sign is used as the assignment operator. This is often a source of confusion for new programmers,

num 510 variable num is assigned the value 10
num 55 10 variable num is compared to the value 10

| Relational Operators | Example | Result |
|---|---|---|
| == equal | 10 == 10 | True |
| != not equal | 10 != 10 | False |
| < less than | 10 < 20 | True |
| > greater than | 'Alan' > 'Brenda' | False |
| <= less than or equal to | 10 <= 10 | True |
| >= greater than or equal to | 'A' >= 'D' | False |

FIGURE 3-3 The Relational Operators

Also, ! 5 is used for inequality simply because there is no keyboard character for the fi symbol. String values are ordered based on their character encoding, which normally follows a
**lexographical** (**dictionary**) **ordering**. For example, 'Alan' is less than 'Brenda' since the
Unicode (ASCII) value for 'A' is 65, and 'B' is 66. However, 'alan' is greater than (comes
*after*) 'Brenda' since the Unicode encoding of lowercase letters (97, 98, . . .) comes *after* the
encoding of uppercase letters (65, 66, . . .). Recall from Chapter 2 that the encoding of any character can be obtained by use of the ord function.

LET’S TRY IT From the Python Shell, enter the following and observe the results.

... 10 55 20 ... '2' , '9' ... 'Hello' 55 "Hello" ??? ??? ???
... 10 !5 20 ... '12' , '9' ... 'Hello' , 'Zebra' ??? ??? ???
... 10 ,5 20 ... '12' . '9' ... 'hello' , 'ZEBRA' ??? ??? ???

The relational operators 55, !5, ,, ., ,5, .5 can be applied to any set of values that has an ordering.
3.2.2 Membership Operators

Python provides a convenient pair of **membership operators**. These operators can be used to easily determine if a particular value occurs within a specified list of values. The membership operators are given in Figure 3-4.

The in operator is used to determine if a specific value is in a given list, returning True if found, and False otherwise. The not in operator returns the opposite result. The list of values surrounded by matching parentheses in the figure are called *tuples* in Python. Tuples (and lists) are covered in Chapter 4.

| Membership Operators | Examples | Result |
|---|---|---|
| in | 10 in (10, 20, 30) | True |
| | red in ('red','green','blue') | True |
| not in | 10 not in (10, 20, 30) | False |

FIGURE 3-4 The Membership Operators

The membership operators *can also be used to check if a given string occurs within another string*,

```
... 'Dr.' in 'Dr. Madison'
True
```

As with the relational operators, the membership operators can be used to construct Boolean expressions.

LET'S TRY IT From the Python Shell, enter the following and observe the results.

```
... 10 in (40, 20, 10) ... grade 5 'A'
??? ... grade in ('A','B','C','D','F')

???
... 10 not in (40, 20, 10)
??? ... city 5 'Houston'

... city in ('NY', 'Baltimore', 'LA') ... .25 in (.45, .25, .65) ???
???
```

Python provides membership operators in and not in for determining if a specific value is in (or not in) a given list of values.

3.2.3 Boolean Operators

George Boole, in the mid-1800s, developed what we now call *Boolean algebra*. His goal was to develop an algebra based on true/false rather than numerical

values. Boolean algebra contains a set of **Boolean** ( **logical**) **operators**, denoted by and, or, and not in Python. These logical operators can be used to construct arbitrarily complex Boolean expressions. The Boolean operators are shown in Figure 3-5.

Logical and is true only when *both* its operands are true—otherwise, it is false. Logical or is true when *either or both* of its operands are true, and thus false only when both operands are false. Logical not simply reverses truth values—not False equals True, and not True equals False.

| x | y | x and y | x or y | not x |
|---|---|---|---|---|
| False | False | False | False | True |
| True | False | False | True | False |
| False | True | False | True | |
| True | True | True | True | |

FIGURE 3-5
Boolean Logic Truth Table

One must be cautious when using Boolean operators. For example, in mathematics, to denote that a value is within a certain range is written as
1 ,5 num ,5 10
In most programming languages, however, this expression does not make sense. To see why, let's assume that num has the value 15. The expression would then be evaluated as follows, 1 ,5 num ,5 10 → 1 ,5 15 ,5 10 → True ,5 10 → **?!?**

It does not make sense to check if True is less than or equal to 10. (Some programming languages would generate a mixed-type expression error for this.) The correct way of denoting the condition is by use of the Boolean and operator,

1 ,5 num and num ,5 10

In some languages (such as Python), Boolean values True and False have integer values 1 and 0, respectively. In such cases, the expression 1 ,5num ,510 would evaluate to True ,510 would evaluate to 1 ,510, which equals True. This would not be the correct result for this expression, however. Let's see what we get when we do evaluate this expression in the Python shell,

. .. num 5 15

... 1 ,5 num ,5 10
False

We actually get the correct result, False. So what is going on here? The answer is that Python is playing a trick here. For Boolean expressions of the particular form,
*value1* ,5 *var* ,5 *value2*
Python automatically rewrites this before performing the evaluation,
*value1* ,5 *var* and *var* ,5 *value2*
Thus, it is important to note that expressions of this form are handled in a special way in Python, and would not be proper to use in most other programming languages.

One must also be careful in the use of and/or Boolean operators. For example, not(num 55 0 and num 5 5 1) is True for any value of num, as is (num ! 5 0) or (num ! 5 1), and therefore are not useful expressions. The Boolean expression num , 0 and num . 10 is also useless since it is always False.

Finally, Boolean literals True and False are never quoted. Doing so would cause them to be taken as string values ('True'). And as we saw, Boolean expressions do not necessarily contain Boolean operators. For example, 10 5 20 is a Boolean expression. By definition, Boolean literals True and False are Boolean expressions as well.

LET’S TRY IT From the Python Shell, enter the following and observe the results.

... True and False ... (10 , 0) and (10 . 2) ??? ???
... True or False ... (10 , 0) or (10 . 2) ??? ???
... not(True) and False ... not(10 , 0) or (10 . 2) ??? ???
... not(True and False) ... not(10 , 0 or 10 . 2) ??? ???

**Boolean operators** in Python are denoted by and, or, and not.
3.2.4 Operator Precedence and Boolean Expressions

The operator precedence (and operator associativity) of arithmetic operators was given in Chapter 2. Operator precedence also applies to Boolean operators. Since Boolean expressions can contain arithmetic as well as relational and Boolean operators, the precedence of all operators needs to be collectively applied. An updated operator precedence table is given in Figure 3-6.

| Operator | Associativity |
|---|---|
| ** (exponentiation) | right-to-left |
| - (negation) | left-to-right |
| * (mult), / (div), // (truncating div), % (modulo) | left-to-right |
| + (addition), - (subtraction) | left-to-right |
| <, >, <=, >=, !=, == (relational operators) | left-to-right |
| not | left-to-right |
| and | left-to-right |
| or | left-to-right |

FIGURE 3-6 Operator Precedence of Arithmetic, Relational, and Boolean Operators

As before, in the table, higher-priority operators are placed above lower-priority ones. Thus, we see that *all arithmetic operators are performed before any relational or Boolean operator*, 10 20 1 30 → 30 50 → True

In addition, *all of the relational operators are performed before any Boolean operator*, 10 20 and 30 , 20 → True and False → False

10 20 or 30 , 20 → True or False → True

And as with arithmetic operators, Boolean operators have various levels of precedence. Unary Boolean operator not has higher precedence than and, and Boolean operator and has higher precedence than or.

10 , 20 and 30 , 20 or 30 , 40 → True and False or True
→ False or True → True not 10 , 20 or 30 , 20 → not True or False
→ False or False → False

As with arithmetic expressions, it is good programming practice to use parentheses, even if not needed, to add clarity and enhance readability. Thus, the above expressions would be better written by denoting at least some of the subexpressions,

(10 , 20 and 30 , 20) or (30 , 40)
(not 10 , 20) or (30 , 20)
if not all subexpressions,
((10 , 20) and (30 , 20)) or (30 , 40)
(not (10 , 20)) or (30 , 20)

Finally, note from Figure 3.6 above that all relational and Boolean operators associate from left to right.

LET’S TRY IT
From the Python Shell, enter the following and observe the results.

... not True and False ... 10 , 0 and not 10 . 2 ??? ???
... not True and False or True ... not (10 , 0 or 10 , 20) ??? ???

3.2.5 Short-Circuit (Lazy) Evaluation

There are differences in how Boolean expressions are evaluated in different programming languages. For logical and, if the first operand evaluates to false, then regardless of the value of the second operand, the expression is false. Similarly, for logical or, if the first operand evaluates to true, regardless of the value of the second operand, the expression is true. Because of this, some programming languages do not evaluate the second operand when the result is known by the first operand alone, called **shortcircuit (lazy) evaluation**. Subtle errors can result if the programmer is not aware of this. For example, the expression

if n ! 5 0 and 1/n , tolerance:

would evaluate without error for all values of n when short-circuit evaluation is used. If programming in a language not using short-circuit evaluation, however, a “divide by zero” error would result when n is equal to 0. In such cases, the proper construction would be,

if n ! 5 0:
if 1/n , tolerance:
In the Python programming language, short-circuit evaluation is used.
In **short-circuit (lazy) evaluation**, the second operand of Boolean operators and and or is not evaluated if the value of the Boolean expression can be determined from the first operand alone.
3.2.6 Logically Equivalent Boolean Expressions

In numerical algebra, there are arithmetically equivalent expressions of different form. For example, x(y 1 z) and xy 1 xz are equivalent for any numerical values x, y, and z. Similarly, there are *logically* equivalent **Boolean expressions** of different form. We give some examples in Figure 3-7.

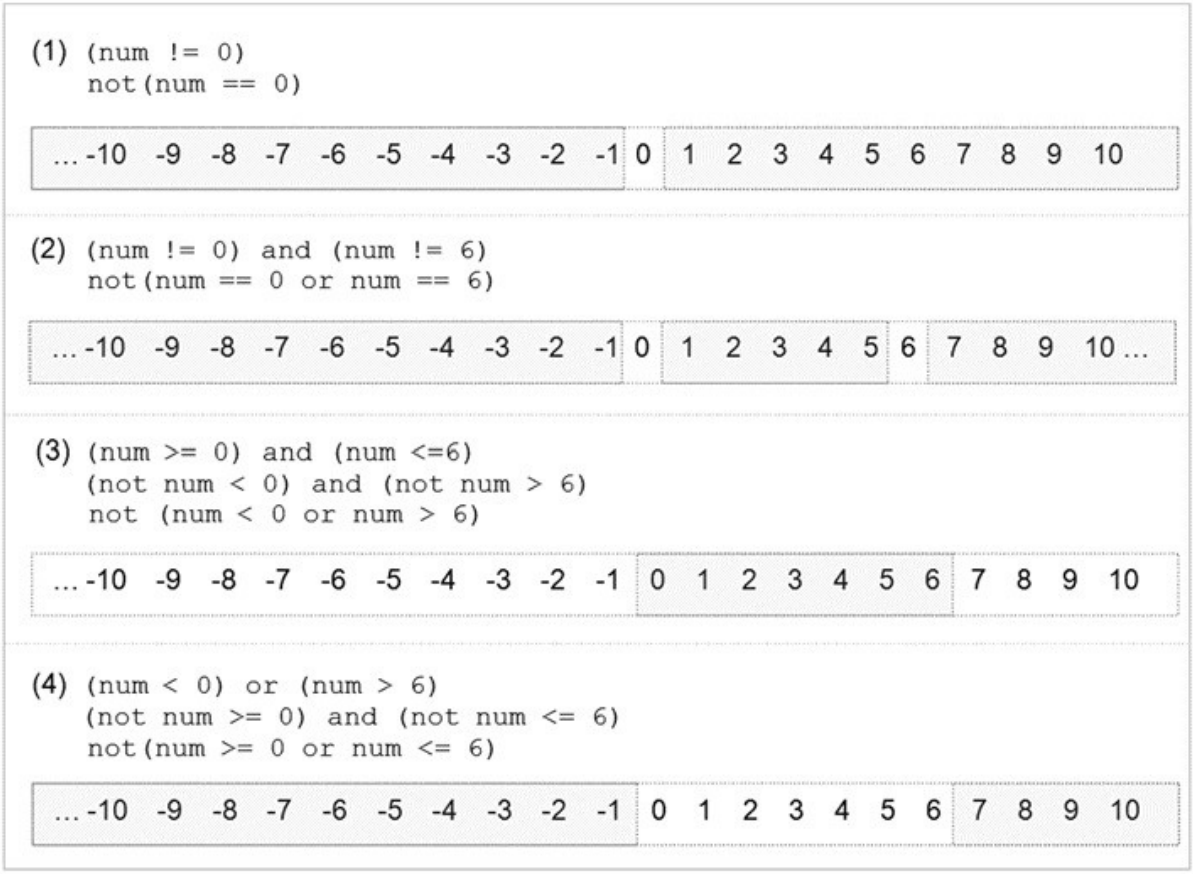


FIGURE 3-7 Logically Equivalent Conditional Expressions

The range of values satisfying each set of expressions is shaded in the figure. Both expressions in (1) are true for any value except 0. The expressions in (2) are true for any value except 0 and 6. The expressions in (3) are only true for values in the range 0 through 6, inclusive. The expressions in (4) are true for all values *except* 0 through 6, inclusive. Figure 3-8 lists common forms of logically equivalent expressions.

| Logically Equivalent Boolean Expressions | | |
|---|---|---|
| x < y | is equivalent to | not(x >= y) |
| x <= y | is equivalent to | not(x > y) |
| x == y | is equivalent to | not(x != y) |
| x != y | is equivalent to | not(x == y) |
| not(x and y) | is equivalent to | (not x) or (not y) |
| not(x or y) | is equivalent to | (not x) and (not y) |

FIGURE 3-8 Forms of Logically Equivalent Boolean Expressions
The last two equivalences above are referred to as De Morgan's Laws.
LET'S TRY IT
From the Python Shell, enter the following and observe the results.

... 10 , 20 ... not(10 , 20 and 10 , 30) ??? ???
... not(10 .5 20) ... (not 10 , 20) or (not 10 , 30) ??? ???
... 10 !5 20 ... not(10 , 20 or 10 , 30) ??? ???
... not (10 55 20) ... (not 10 , 20) and (not 10 , 30) ??? ???

There are logically equivalent **Boolean expressions** of different form.
Self-Test Questions
**1.** Three forms of control in programming are sequential, selection, and _____ control.
**2.** Which of the following expressions evaluate to True?
**(a)**10 .5 8 **(b)**8 ,5 10 **(c)**10 55 8 **(d)**10 !5 8 **(e)**'8' , '10' **3.** Which of the following Boolean expressions evaluate to True?
**(a)**'Dave' , 'Ed' **(b)**'dave' , 'Ed' **(c)**'Dave' , 'Dale'

**4.** What is the value of variable num after the following is executed?
... num 5 10
... num 5 num 1 5
... num 55 20
... num 5 num 1 1

**5.** What does the following expression evaluate to for name equal to 'Ann'? name in ('Jacob', 'MaryAnn', 'Thomas')
**6.** Evaluate the following Boolean expressions using the operator precedence

rules of Python. **(a)**10 .5 8 and 5 !5 3 **(b)** 10 .5 8 and 5 55 3 or 14 , 5
**7.** Which one of the following Boolean expressions is not logically equivalent to the other two? **(a)**not(num , 0 or num . 10)
**(b)**num . 0 and num , 10
**(c)**num .5 0 and num ,5 10

ANSWERS: 1. Iterative, 2. (a,b,d), 3. (a), 4. 16, 5. False, 6. (a) True, (b) False, 7. (b)

## 3.3 Selection Control

A **selection control statement** is a control statement providing selective execution of instructions. A *selection control structure* is a given set of instructions and the selection control statement(s) controlling their execution. We look at the *if statement* providing selection control in Python next.

A **selection control statement** is a control statement providing selective execution of instructions.

### 3.3.1 If Statement

An **if statement** is a selection control statement based on the value of a given Boolean expression. The if statement in Python is depicted in Figure 3-9.

| if statement | Example use | |
|---|---|---|
| `if condition:`<br>`  statements`<br>`else:`<br>`  statements` | `if grade >= 70:`<br>`  print('passing grade')`<br>`else:`<br>`  print('failing grade')` | `if grade == 100:`<br>`  print('perfect score!')` |

FIGURE 3-9 if Statement

Note that if statements may omit the “else” part. A version of the temperature conversion program from Chapter 2 using an if statement is given in Figure 3-10.

This program extends the original program by converting Celsius to Fahrenheit, as well as Fahrenheit to Celsius. The if statement ( **line 13**) selects the appropriate set of instructions to execute based on user input ('F' for Fahrenheit to Celsius, and 'C' for Celsius to Fahrenheit). A statement that contains other statements, such as the if statement, is called a **compound statement**. We look at Python’s use of indentation in compound statements next.

```
1  # Temperature Conversion Program (Celsius-Fahrenheit / Fahrenheit-Celsius)
2
3  # Display program welcome
4  print('This program will convert temperatures (Fahrenheit/Celsius)')
5  print('Enter (F) to convert Fahrenheit to Celsius')
6  print('Enter (C) to convert Celsius to Fahrenheit')
7
8  # Get temperature to convert
9  which = input('Enter selection: ')
10 temp = int(input('Enter temperature to convert: '))
11
12 # Determine temperature conversion needed and display results
13 if which == 'F':
14     converted_temp = (temp - 32) * 5/9
15     print(temp, 'degrees Fahrenheit equals', converted_temp, 'degrees Celsius')
16 else:
17     converted_temp = (9/5 * temp) + 32
18     print(temp, 'degrees Celsius equals', converted_temp, 'degrees Fahrenheit')
```

FIGURE 3-10 Temperature Conversion (Two-Way Conversion)

An **if statement** is a selection control statement based on the value of a given Boolean expression. Statements that contain other statements are referred to as a **compound statement**.

3.3.2 Indentation in Python

One fairly unique aspect of Python is that the amount of indentation of each program line is significant. In most programming languages, indentation has no affect on program logic—it is simply used to align program lines to aid readability. In Python, however, indentation is used to associate and group statements, as shown in Figure 3-11.

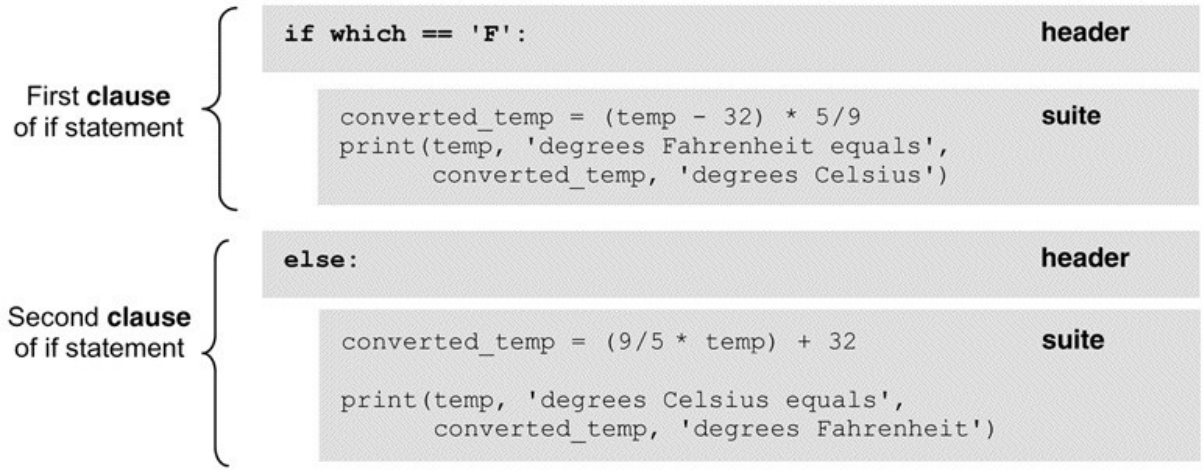



FIGURE 3-11 Compound Statement in Python

A **header** in Python is a specific keyword followed by a colon. In the figure, the

if-else statement contains two headers, “if which 5 5 'F':” containing keyword if, and “else:” consisting only of the keyword else. Headers that are part of the same compound statement must be indented the same amount—otherwise, a syntax error will result.

The set of statements following a header in Python is called a **suite** (commonly called a **block**). The statements of a given suite must all be indented the same amount. A header and its associated suite are together referred to as a **clause**. A compound statement in Python may consist of one or more clauses. While four spaces is commonly used for each level of indentation, any number of spaces may be used, as shown in Figure 3-12.

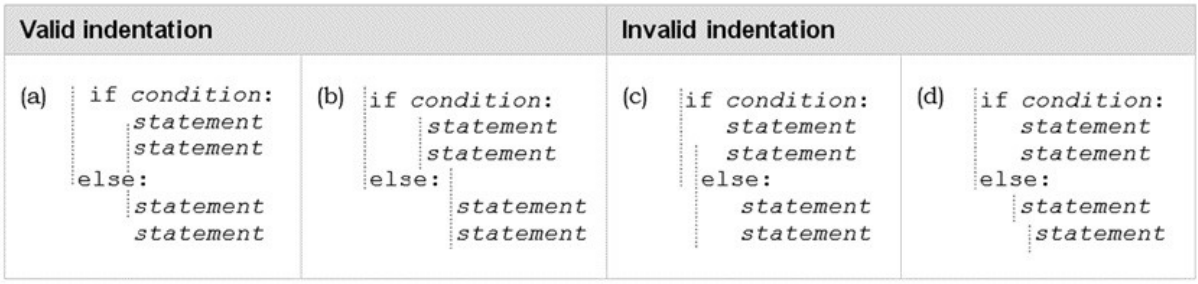



FIGURE 3-12 Compound Statements and Indentation in Python

Both (a) and (b) in the figure are properly indented. In (a), both suites have the same amount of indentation. In (b), each suite has a different amount of indentation. This is syntactically correct (although not good practice) since the amount of indentation within each suite is consistent. Both (c) and (d) are examples of invalid indentation, and thus syntactically incorrect. In (c), the if and else headers of the if statement are not indented the same amount. In (d), the headers are indented the same amount. However, the statements within the second suite are not properly aligned. Finally, note that the suite following a header can itself be a compound statement (another if statement, for example). Thus, compound statements may be nested one within another. We look at nested compounded statements next.

LET’S TRY IT

From IDLE, create and run a Python program containing the code on the left and observe the results. Modify and run the code to match the version on the right and again observe the results. Make sure to indent the code exactly as shown.

grade 5 90

if grade .5 70:
print('passing grade')

else:
print('failing grade') grade 5 90
if grade .5 70:

print('passing grade') else:
print('failing grade')

A **header** in Python starts with a keyword and ends with a colon. The group of statements following a header is called a **suite**. A header and its associated suite are together referred to as a **clause**.

3.3.3 Multi-Way Selection

In this section, we look at the two means of constructing multi-way selection in Python—one involving multiple nested if statements, and the other involving a single if statement and the use of elif headers.

Nested if Statements

There are often times when selection among more than two sets of statements (suites) is needed. For such situations, if statements can be nested, resulting in **multi-way selection**. An example of this is given in Figure 3-13.

| Nested if statements | Example use |
|---|---|
| `if condition:`<br>`    statements`<br>`else:`<br>`    if condition:`<br>`        statements`<br>`    else:`<br>`        if condition:`<br>`            statements`<br><br>`            etc.` | `if grade >= 90:`<br>`    print('Grade of A')`<br>`else:`<br>`    if grade >= 80:`<br>`        print('Grade of B')`<br>`    else:`<br>`        if grade >= 70:`<br>`            print('Grade of C')`<br>`        else:`<br>`            if grade >= 60:`<br>`                print('Grade of D')`<br>`            else:`<br>`                print('Grade of F')` |

FIGURE 3-13 Multi-way Selection Using if Statements

The nested if statements on the right result in a 5-way selection. In the first if statement, if variable grade is greater than or equal to 90, then 'Grade of A' is displayed. Therefore, its else suite is not executed, containing the remaining if statements. If grade is less than 90, the else suite is executed. If grade is greater than or equal to 80, 'Grade of B' is displayed and the rest of the if statements in *its* else suite are skipped, and so on. The final else clause is executed only if all the previous conditions fail, displaying 'Grade of F'. This is referred to as a *catch-all* case. As an example use of nested if statements and a check for invalid input in a program, we give a revised version of the temperature conversion program from Figure 3-10 in Figure 3-14.

```
1  # Temperature Conversion Program (Celsius-Fahrenheit / Fahrenheit-Celsius)
2
3  # Display program welcome
4  print('This program will convert temperatures (Fahrenheit/Celsius)')
5  print('Enter (F) to convert Fahrenheit to Celsius')
6  print('Enter (C) to convert Celsius to Fahrenheit')
7
8  # Get temperature to convert
9  which = input('Enter selection: ')
10 temp = int(input('Enter temperature to convert: '))
11
12 # Determine temperature conversion needed and display results
13 if which == 'F':
14     converted_temp = format((temp - 32) * 5.0/9.0, '.1f')
15     print(temp, 'degrees Fahrenheit equals', converted_temp, 'degrees Celsius')
16 else:
17     if which == 'C':
18        converted_temp = format((9.0/5.0 * temp) + 32, '.1f')
19        print(temp,'degrees Celsius equals',converted_temp,'degrees Fahrenheit')
20     else:
21         print('INVALID INPUT')
```

FIGURE 3-14 Temperature Conversion Program (Input Error Dectection)

In this version, there is a catch-all clause ( **line 20**) for handling invalid input. We next look at a more concise means of denoting multi-way selection in Python.

LET’S TRY IT

From IDLE, create and run a simple program containing the code below and observe the results. Make sure to indent the code exactly as shown.

credits 5 45
if credits .5 90:
print('Senior')
else:
if credits 60:
print('Junior')

else:
if credits 30:
print('Sophomore')
else:
if credits 1:
print('Freshman')
else:
print('* No Earned Credits *')
If statements can be nested in Python, resulting in **multi-way selection**.
The **elif** Header in Python

If statements may contain only one else header. Thus, if-else statements must be nested to achieve multi-way selection. Python, however, has another header called elif ("else-if") that provides multi-way selection in a *single* if statement, shown in Figure 3-15.

All the headers of an if-elif statement are indented the same amount, thus avoiding the deeply nested levels of indentation with the use of if-else statements. A final else clause may be used for "catch-all" situations. We next look at iterative control in Python.

```
if grade >= 90:
    print('Grade of A')
elif grade >= 80:
    print('Grade of B')
elif grade >= 70:
    print('Grade of C')
elif grade >= 60:
    print('Grade of D')
else:
    print('Grade of F')
```

FIGURE 3-15 The elif Header in Python
LET'S TRY IT From IDLE, create and run a Python program containing the code below and observe the results. Make sure to indent the code exactly as shown.
credits 5 45

if credits .5 90:
print('Senior')

```
elif credits .5 60:
print('Junior')
elif grade .5 30:
print('Sophomore')
elif grade .5 1:
print('Freshman')
else:
print('* No Earned Credits *')
```

If statements may contain any number of **elif headers**, providing for multi-way selection.

3.3.4 Let's Apply It—Number of Days in Month Program

The following Python program (Figure 3-16) prompts the user for a given month (and year for February), and displays how many days are in the month. This program utilizes the following programming features:

➤ if statement ➤ elif header

**Lines 1–4** provide the program header and program greeting. On **line 7**, variable valid_input is initialized to True for the input error-checking performed. **Line 10** prompts the user for the month, read as an integer value (1–12), and stores in variable month. On **line 15** the month of February is checked for. February is the only month that may have a different number of days—28 for a regular year, and 29 for leap years. Thus, when February (2) is entered, the user is also prompted for the year ( **line 16**). If the year is a leap year, then variable num_days is set to 29— otherwise, it is set to 28.

Generally, if a year is (evenly) divisible by 4, then it is a leap year. However, there are a couple of exceptions. If the year is divisible by 4 but is also divisible by 100, then it is *not* a leap year— unless, it is also divisible by 400, then it is. For example, 1996 and 2000 were leap years, but 1900 was not. This condition is given below.

```
(year % 4 55 0) and (not (year % 100 55 0) or (year % 400 55 0))
Thus, the conditions for which this Boolean expression is true are,
(year % 4 55 0) and not (year % 100 55 0) and
(year % 4 55 0) and (year % 400 55 0)
```

Program Execution ...

```
This program will determine the number of days in a given month

Enter the month (1-12): 14
* Invalid Value Entered - 14 '*'
>>>

This program will determine the number of days in a given month

Enter the month (1-12): 2
Please enter the year (e.g., 2010): 2000
There are 29 days in the month
```

```
1  # Number of Days in Month Program
2
3  # program greeting
4  print('This program will display the number of days in a given month\n')
5
6  # init
7  valid_input = True
8
9  # get user input
10 month = int(input('Enter the month (1-12): '))
11
12 # determine num of days in month
13
14 # february
15 if month == 2:
16     year = int(input('Please enter the year (e.g., 2010): '))
17
18     if (year % 4 == 0) and (not (year % 100 == 0) or (year % 400 == 0)):
19         num_days = 29
20     else:
21         num_days = 28
22
23 # january, march, may, july, august, october, december
24 elif month in (1, 3, 5, 7, 8, 10, 12):
25     num days = 31
26
27 # april, june, september, november
28 elif month in (4,6,9,11):
29     num_days = 30
30
31 # invalid input
32 else:
33     print('* Invalid Value Entered - ', month, '*')
34     valid_input = False
35
36 # output result
37 if valid_input:
38     print('There are', num_days, 'days in the month')
```

FIGURE 3-16 Number of Days in Month Program

**Line 24** checks if month is equal to 1, 3, 5, 7, 8, 10, or 12. If true, then

num_days is assigned to 31. If not true, **line 28** checks if month is equal to 4, 6, 9, or 11 (all the remaining months except February). If true, then num_days is assigned to 30. If not true, then an invalid month (number) was entered, and valid_input is set to False. Finally, the number of days in the month is displayed only if the input is valid ( **line 38**).

Self-T est Questions
**1.** All if statements must contain either an else or elif header. (TRUE/FALSE)

**2.** A compound statement is,
**(a)** A statement that spans more than one line
**(b)** A statement that contains other statements
**(c)** A statement that contains at least one arithmetic expression

**3.** Which of the following statements are true regarding headers in Python?
**(a)** Headers begin with a keyword and end with a colon.
**(b)** Headers always occur in pairs.
**(c)** All headers of the same compound statement must be indented the same amount.

**4.** Which of the following statements is true?
**(a)** Statements within a suite can be indented a different amount.
**(b)** Statements within a suite can be indented a different amount as long as all headers in the

statement that it occurs in are indented the same amount.
**(c)** All headers must be indented the same amount as all other headers in the same statement,
and all statements in a given suite must be indented the same amount.

**5 .** The elif header allows for,
**(a)** Multi-way selection that cannot be accomplished otherwise
**(b)** Multi-way selection as a single if statement
**(c)** The use of a “catch-all” case in multi-way selection

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

## 3.4 Iterative Control

An **iterative control statement** is a control statement providing the repeated execution of a set of instructions. An *iterative control structure* is a set of

instructions and the iterative control statement(s) controlling their execution. Because of their repeated execution, iterative control structures are commonly referred to as “loops.” We look at one specific iterative control statement next, the while statement.

An **iterative control statement** is a control statement that allows for the repeated execution of a set of statements.

3.4.1 While Statement

A **while statement** is an iterative control statement that repeatedly executes a set of statements based on a provided Boolean expression (condition). All iterative control needed in a program can be achieved by use of the while statement. Figure 3-17 contains an example of a while loop in Python that sums the first n integers, for a given (positive) value n entered by the user.

| while statement | Example use |
| --- | --- |
| `while condition:`<br>`    suite` | `sum = 0`<br>`current = 1`<br><br>`n = int(input('Enter value: '))`<br><br>`while current <= n:`<br>`    sum = sum + current`<br>`    current = current + 1` |

FIGURE 3-17 The while Statement in Python

As long as the condition of a while statement is true, the statements within the loop are (re)executed. Once the condition becomes false, the iteration terminates and control continues with the first statement after the while loop. Note that it is possible that the first time a loop is reached, the condition may be false, and therefore the loop would never be executed.

Suppose, for the example in the figure, that the user enters the value 3. Since variable current is initialized to 1 (referred to as a *counter variable*), the first time the while statement is reached, current ,53 is true. Thus, the statements within the loop are executed and sum is u pdated to sum  1current. Since sum is initialized to 0, sum becomes 1. Similarly, current is updated and assigned to 2. After the first time through the loop, control returns to the “top” of the loop. The condition is again found to be true and thus the loop is executed a second time.

In this iteration, both sum and current become 3. In the next iteration, the condition is still true, and therefore, the loop is executed a third time. This time, sum becomes 6 and current becomes 4. Thus, when control returns to the top of the loop, the condition is False and the loop terminates. The final value of sum therefore is 6(1 1 2 1 3). This process is summarized in Figure 3-18.

| Iteration | sum | current | current <= 3 | sum = sum + current | current = current + 1 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | True | sum = 0 + 1 (1) | current = 1 + 1 (2) |
| 2 | 1 | 2 | True | sum = 1 + 2 (3) | current = 2 + 1 (3) |
| 3 | 3 | 3 | True | sum = 3 + 3 (6) | current = 3 + 1 (4) |
| 4 | 6 | 4 | False | loop termination | |

FIGURE 3-18 Iterative Steps for Adding First Three Integers

A **while statement** is an iterative control statement that repeatedly executes a set of statements based on a provided Boolean expression.

3.4.2 Input Error Checking

The while statement is well suited for input error checking in a program. This is demonstrated in the revised version of the temperature conversion program from Figure 3-14, reproduced in Figure 3-19.

```
# Temperature Conversion Program (Celsius-Fahrenheit / Fahrenheit-Celsius)

# Display program welcome
print('This program will convert temperatures (Fahrenheit/Celsius)')
print('Enter (F) to convert Fahrenheit to Celsius')
print('Enter (C) to convert Celsius to Fahrenheit')

# Get temperature to convert
which = input('Enter selection: ')

while which != 'F' and which != 'C':
   which = input("Please enter 'F' or 'C': ")

temp = int(input('Enter temperature to convert: '))

# Determine temperature conversion needed and display results
if which == 'F':
    converted_temp = format((temp - 32) * 5/9, '.1f')
    print(temp, 'degrees Fahrenheit equals', converted_temp, 'degrees Celsius')
else:
    converted_temp = format((9/5 * temp) + 32, '.1f')
    print(temp, 'degrees Celsius equals', converted_temp, 'degrees Fahrenheit')
```

FIGURE 3-19 Temperature Conversion Program (Invalid Input Checking)

The difference in this program from the previous version is that rather than terminating on invalid input, the program continues to prompt the user until a valid temperature conversion, 'F' or 'C', is entered. Thus, the associated input

statement is contained within a while loop that keeps iterating as long as variable which contains an invalid value. Once the user enters a proper value, the loop terminates allowing the program to continue.

LET'S TRY IT
In IDLE, create and run a simple program containing the code below and observe the results. Make sure to indent the code exactly as shown.

n 5 10
sum 5 0
current 5 1

while current < 5 n:
sum sum 1 current current current 1 1

print(sum)
???
n 5 10
sum 5 0
current 5 1
while current <5 n:

sum sum 1 current current current 1 1 print(sum)
???
The while statement is well suited for input error checking.
3.4.3 Infinite loops

An **infinite loop** is an iterative control structure that never terminates (or eventually terminates with a system error). Infinite loops are generally the result of programming errors. For example, if the condition of a while loop can never be false, an infinite loop will result when executed. Consider if the program segment in Figure 3-17, reproduced in Figure 3-20, omitted the statement incrementing variable current. Since current is initialized to 1, it would remain 1 in all iterations, causing the expression current ,5n to be always be true. Thus, the loop would never terminate.

```
# add up first n integers
sum = 0
current = 1

n = int(input('Enter value: '))

while current <= n:
    sum = sum + current
```

FIGURE 3-20 Infinite Loop

Such infinite loops can cause a program to “hang,” that is, to be unresponsive to the user. In such cases, the program must be terminated by use of some special keyboard input (such as ctrl-C) to interrupt the execution.

LET’S TRY IT From IDLE, create and run a simple program containing the code below and observe the results. Make sure to indent the code exactly as shown. To terminate an executing loop, hit ctrl-C.
while True:
print ('Looping') ???

n 5 10
sum 5 0
current 5 1

while current ,5 n: sum sum 1 current print(sum)

???
n 5 10
sum 5 0
current 5 1

while current ,5 n: sum sum 1 current n n 2 1

print(sum)
???
An **infinite loop** is an iterative control structure that never terminates (or eventually terminates with a system error).
3.4.4 Definite vs. Indefinite Loops
A **definite loop** is a program loop in which the number of times the loop will iterate can be determined before the loop is executed. For example, the while

loop introduced in Figure 3-17 is a definite loop,

sum 5 0
current 5 1
n 5 input('Enter value: ')
while current ,5 n:

sum 5 sum 1 current
current 5 current 1 1
Although it is not known what the value of n will be until the input statement is executed, its value *is* known by the time the while loop is reached. Thus, it will execute “n times.”

An **indefinite loop** is a program loop in which the number of times that the loop will iterate cannot be determined before the loop is executed. Consider the while loop in the temperature conversion program of Figure 3-19.

which 5 input("Enter selection: ")
while which ! 5 'F' and which ! 5 'C':
which 5 input("Please enter 'F' or 'C': ")

In this case, the number of times that the loop will be executed depends on how many times the user mistypes the input. Thus, a while statement can be used to construct both definite and indefinite loops. In the next chapter we look at the for statement, specifically suited for the construction of definite loops.

A **definite loop** is a program loop in which the number of times the loop will iterate can be determined before the loop is executed. A **indefinite loop** is a program loop in which the number of times the loop will iterate is not known before the loop is executed.

3.4.5 Boolean Flags and Indefinite Loops
Often the condition of a given while loop is denoted by a single Boolean variable, called a **Boolean flag**. This is shown in Figure 3-21.

Boolean variable valid_entries is a Boolean flag, controlling the while loop at **line 12**. If the mileage of the last oil change is greater than the current mileage, an error message is displayed ( **lines 17–18**), and the while loop is re-executed. If the current mileage is greater than (or equal to) the mileage of the last oil change, miles_traveled is set to this difference and valid_ entries is set to True,

causing the loop to terminate. Thus, **lines 23–28** display either that they are due for an oil change, an oil change will soon be needed, or there is no immediate need for an oil change.

A single Boolean variable used as the condition of a given control statement is called a **Boolean flag**.

```
1  # Oil Change Notification Program
2
3  # display program welcome
4  print('This program will determine if your car is in need of an oil change')
5
6  # init
7  miles_between_oil_change = 7500  # num miles between oil changes
8  miles_warning = 500      # how soon to warn of needed oil change
9  valid_entries = False
10
11 # get mileage of last oil change and current mileage and display
12 while not valid_entries:
13     mileage_last_oilchange = int(input('Enter mileage of last oil change: '))
14     current_mileage = int(input('Enter current mileage: '))
15
16     if current_mileage < mileage_last_oilchange:
17         print('Invalid entry - current mileage entered is less than')
18         print('mileage entered of last oil change')
19     else:
20         miles_traveled = current_mileage - mileage_last_oilchange
21         valid_entries = True
22
23 if miles_traveled >= miles_oil_change:
24     print('You are due for an oil change')
25 elif miles_traveled >= miles_oil_change - miles_warning:
26     print('You will soon be due for an oil change')
27 else:
28     print('You are not in immediate need of an oil change')
```

FIGURE 3-21 Indefinite Loop Using a Boolean Flag

### 3.4.6 Let's Apply It—Coin Change Exercise Program

The Python program in Figure 3-22 implements an exercise for children learning to count change. It displays a random value between 1 and 99 cents, and asks the user to enter a set of coins that sums exactly to the amount shown. The program utilizes the following programming features:

➤ while loop ➤ if statement ➤ Boolean flag ➤ random number generator

On **line 3**, the random module is imported for use of function randint. This function is called (on **line 18**) to randomly generate a coin value for the user to match, stored in variable amount. **Lines 6–10** provide the program greeting. On **line 13** variable terminate is initialized to False, used to control when the main loop (and thus the program) terminates. On **line 14**, empty_str is initialized to

the empty string literal '', used to determine when the user has entered an empty line to end the coin entries. These two variables need only be initialized once, and therefore are assigned before the main while loop.

The game begins on **line 17**. Since Boolean flag terminate is initialized to False, the while loop is executed. Besides variable amount, game_over is initialized to False, and total is initialized to 0. Variable game_over serves as another Boolean flag to determine if the current game is to continue or not. The coin entry ends if either the user enters a blank line (indicating that they are done entering coins) in which case the result is displayed and game_over is set to True ( **line 37–43**), or if the total amount accumulated exceeds the total amount to be matched (on **line 45–48**).

At the top of the while loop, a third Boolean flag is used, valid_entry. The value of this flag determines whether the user should be prompted again because of an invalid input—a value

```
Program Execution ...

The purpose of this exercise is to enter a number of coin values
that add up to a displayed target value.

Enter coins values as 1-penny, 5-nickel, 10-dime and 25-quarter.
Hit return after the last entered coin value.
---------------
Enter coins that add up to 63 cents, one per line.

Enter first coin: 25
Enter next coin: 25
Enter next coin: 10
Enter next coin:
Sorry - you only entered 60 cents.

Try again (y/n)?: y
Enter coins that add up to 21 cents, one per line.

Enter first coin: 11
Invalid entry
Enter next coin: 10
Enter next coin: 10
Enter next coin: 5
Sorry - total amount exceeds 21 cents.

Try again (y/n)?: y
Enter coins that add up to 83 cents, one per line.

Enter first coin: 25
Enter next coin: 25
Enter next coin: 25
Enter next coin: 5
Enter next coin: 1
Enter next coin: 1
Enter next coin: 1
Enter next coin:
Correct!

Try again (y/n)?: n
Thanks for playing ... goodbye
```

FIGURE 3-22 Coin Change Exercise Program (*Continued*)

other than '1', '5', '10', or '25', or the empty string. Note the use of the membership operator in ( **line 32**). Thus, once the user inputs an appropriate value, valid_entry is set to True ( **line 33**)—otherwise, the message 'Invalid entry' is displayed and valid_entry remains False, causing the loop to execute again. The list of valid entered values on line 32 includes variable empty_str

since this is the value input when the user hits return to terminate their entry of coin values. When the empty string is found ( **line 37**), the total coin value entered in variable total is compared with variable amount (the amount to be matched). If equal, the message 'Correct!' is displayed ( **line 39**)—otherwise, a message is displayed indicating how much they entered. This amount is always less than the required amount, since whenever variable total exceeds amount, the current game ends ( **lines 46–48**).

```
# Coin Change Exercise Program

import random

# program greeting
print('The purpose of this exercise is to enter a number of coin values')
print('that add up to to a displayed target value.\n')
print('Enter coins values as 1-penny, 5-nickel, 10-dime and 25-quarter')
print("Hit return after the last entered coin value.")
print('----------------')

# init
terminate = False
empty_str = ''

# start game
while not terminate:
    amount = random.randint(1,99)
    print('Enter coins that add up to', amount, 'cents, one per line.\n')
    game_over = False
    total = 0

    while not game_over:
        valid_entry = False

        while not valid_entry:
            if total == 0:
                entry = input('Enter first coin: ')
            else:
                entry = input('Enter next coin: ')

            if entry in (empty_str,'1','5','10','25'):
                valid_entry = True
            else:
                print('Invalid entry')

        if entry == empty_str:
            if total == amount:
                print('Correct!')
            else:
                print('Sorry - you only entered', total, 'cents.')

            game_over = True
        else:
            total = total + int(entry)
            if total > amount:
                print('Sorry - total amount exceeds', amount, 'cents.')
                game_over = True

        if game_over:
            entry = input('\nTry again (y/n)?: ')

            if entry == 'n':
                terminate = True

print('Thanks for playing ... goodbye')
```

FIGURE 3-22 Coin Change Exercise Program

When **line 50** is reached, Boolean flag game_over may be either True or False. It is True when the user has indicated that they have entered all their coin values (by hitting return), or if the total of the coin values entered exceeds the value in variable amount—it is False otherwise. Therefore, flag variable game_over is used to determine whether the user should be prompted to play another game ( **line 51**). If they choose to quit the program when prompted, then Boolean variable terminate is set to True. This causes the encompassing while loop at **line 17** to terminate, leaving only the final "goodbye" message on **line 56** to be executed before the program terminates.

Self-T est Questions
**1.** A while loop continues to iterate until its condition becomes false. TRUE/FALSE
**2.** A while loop executes zero or more times. TRUE/FALSE
**3.** All iteration can be achieved by a while loop. TRUE/FALSE

**4.** An infinite loop is an iterative control structures that,
**(a)** Loops forever and must be forced to terminate
**(b)** Loops until the program terminates with a system error
**(c)** Both of the above

**5.** The terms *definite loop* and *indefinite loop* are used to indicate whether,
**(a)** A given loop executes at least once
**(b)** The number of times that a loop is executed can be determined before the loop is executed. **(c)** Both of the above

**6.** A Boolean flag is,
**(a)** A variable
**(b)** Has the value True or False
**(c)** Is used as a condition for control statements
**(d)** All of the above

ANSWERS: 1. True, 2. True, 3. True, 4. (c), 5. (b), 6. (d)

COMPUTATIONAL PROBLEM SOLVING
3.5 Calendar Month Program

3.5.1 The Problem

The problem is to display a calendar month for any given month between

January 1800 and December 2099. The format of the month should be as shown in Figure 3-23.

3.5.2 Problem Analysis

T wo specific algorithms are needed for this problem. First, we need an algorithm for computing the first day of a given month for years 1800 through 2099. This algorithm is given in Chapter 1. The second needed algorithm is for appropriately displaying the calendar month, given the day of the week that

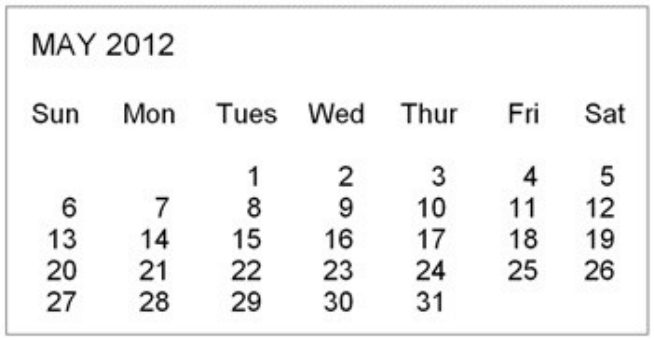



FIGURE 3-23 Calendar Month Display

the first day falls on, and the number of days in the month. We shall develop this algorithm. The data representation issues for this problem are straight forward.

3.5.3 Program Design
Meeting the Program Requirements

We will develop and implement an algorithm that displays the month as given. There is no requirement of how the month and year are to be entered. We shall therefore request the user to enter the month and year as integer values, with appropriate input error checking.

Data Description

What needs to be represented in the program is the month and year entered, whether the year is a leap year or not, the number of days in the month, and which day the first of the month falls on. Given that information, the calendar month can be displayed. The year and month will be entered and stored as integer values, represented by variables year and month,

year 52012 month 5 5

The remaining values will be computed by the program based on the given year and month, as given below,
leap_year num_days_in_month day_of_week
Variable leap_year holds a Boolean (True/False) value. Variables num_days_in_month and day_of_week each hold integer values.
Algorithmic Approach
First, we need an algorithm for determining the day of the week that a given date falls on. The algorithm for this from Chapter 1 is reproduced in Figure 3-24.

We also need to determine how many days are in a given month, which relies on an algorithm for determining leap years for the month of February. The code for this has already been developed in the “Number of Days in Month” program in section 3.3.4. We shall also reuse the portion of code from that program for determining leap years, reproduced below.

if (year % 4 55 0) and (not (year % 100 55 0) or year % 400): leap_year 5 True else:
leap_Year 5 False

Let’s review how this algorithm works, and try to determine the day of the week on which May 24, 2025 falls. First, variable century_digits (holding the first two digits of the year) is set to 20 and year_digits (holding the last two digits of the year) is set to 25 ( **steps 1 and 2**). Variable value, in **step 3**, is then set to

value 5 year_digits 1 floor(year_digits / 4)
5 25 1 floor(25/4) → 25 1 floor(6.25) → 25 1 6 → 31

To determine the day of the week for a given **month**, **day**, and **year**:

1. Let **century_digits** be equal to the first two digits of the year.
2. Let **year_digits** be equal to the last two digits of the year.
3. Let **value** be equal to **year_digits** + floor(**year_digits** / 4)
4. If **century_digits** equals 18, then add 2 to **value**, else
   if **century_digits** equals 20, then add 6 to **value**.
5. If the **month** is equal to January and the **year** is not a leap year,
   then add 1 to **value**, else,

   if the **month** is equal to February and the **year** is a leap year, then
   add 3 to **value**; if not a leap year, then add 4 to **value,** else,

   if the **month** is equal to March or November, then add 4 to **value**, else,

   if the **month** is equal to April or July, then add 0 to **value**, else,

   if the **month** is equal to May, then add 2 to **value**, else,

   if the **month** is equal to June, then add 5 to **value**, else,

   if the **month** is equal to August, then add 3 to **value**, else,

   if the **month** is equal to October, then add 1 to **value**, else,

   if the **month** is equal to September or December, then add 6 to **value**,
6. Set **value** equal to (**value** + **day**) mod 7.
7. If **value** is equal to 1, then the day of the week is Sunday; else

   if **value** is equal to 2, day of the week is Monday; else

   if **value** is equal to 3, day of the week is Tuesday; else

   if **value** is equal to 4, day of the week is Wednesday; else

   if **value** is equal to 5, day of the week is Thursday; else

   if **value** is equal to 6, day of the week is Friday; else

   if **value** is equal to 0, day of the week is Saturday

FIGURE 3-24 Day of the Week Algorithm (from Chapter 1)

In **step 4**, since century_digits is equal to 20, value is incremented by 6,
value → 6 → 37

In **step 5**, since the month is equal to May, value is incremented by 2,
value → 2 → 39

In **step 6**, value is updated based on the day of the month. Since we want to determine the day of the week for the 24th (of May), value is updated as follows,

value 5 (value 1 *day of the month*) mod 7
5 (39 1 24) mod 7
5 63 mod 7

5 0

Therefore, by **step 7** of the algorithm, the day of the week for May 24, 2025 is a Saturday. A table for the interpretation of the day of the week for the final computed value is given in Figure 3-25.

| 1 | 2 | 3 | 4 | 5 | 6 | 0 |
|---|---|---|---|---|---|---|
| Sunday | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday |

FIGURE 3-25 Interpretation of the Day of the Week Algorithm Results
Overall Program Steps
The overall steps in this program design are given in Figure 3-26.

3.5.4 Program Implementation and Testing
Stage 1—Determining the Number of Days in the Month/Leap Years

We develop and test the program in three stages. First, we implement and test the code that determines, for a given month and year, the number of days in the month and whether the year is a leap year or not, given in Figure 3-27.

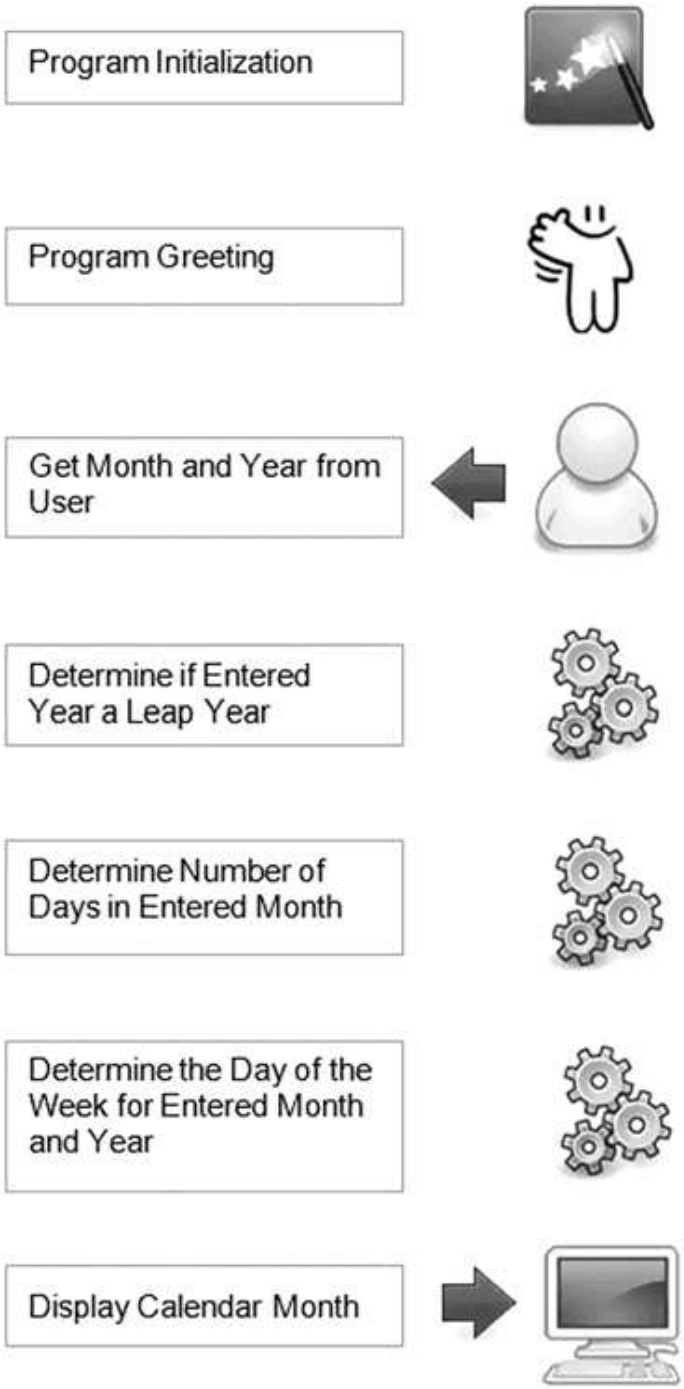



FIGURE 3-26 Overall Steps of Calendar Month Program

```
 # Calendar Month Program (stage 1)

# init
terminate = False

# program greeting
print('This program will display a calendar month between 1800 and 2099')

while not terminate:
    # get month and year
    month = int(input('Enter month 1-12 (-1 to quit): '))

    if month == -1:
        terminate = True
    else:
        while month < 1 or month > 12:
            month = int(input('INVALID INPUT - Enter month 1-12: '))

        year = int(input('Enter year (yyyy): '))

        while year < 1800 or year > 2099:
            year = int(input('INVALID - Enter year (1800-2099): '))

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

        # determine num of days in month
        if month in (1, 3, 5, 7, 8, 10, 12):
            num_days_in_month = 31
        elif month in (4, 6, 9, 11):
            num_days_in_month = 30
        elif leap_year:  # February
            num_days_in_month = 29
        else:
            num_days_in_month = 28

        print ('\n', month, ',', year, 'has', num_days_in_month, 'days')

        if leap_year:
            print (year, 'is a leap year\n')
        else:
            print (year, 'is NOT a leap year\n')
```

FIGURE 3-27 First Stage of Calendar Month Program

The month and year entered by the user are stored in variables month and year. While loops are used at **lines 16** and **21** to perform input error checking. **Lines 25–28** are adapted from the previous Number of Days in Month program for determining leap years. **Lines 31–38** are similar to the previous program for determining the number of days in a month, stored in variable num_days_in_month. **Lines 42– 45** contain added code for the purpose of testing. These instructions will not be part of the final program. The program

continues to prompt for another month until 21 is entered. Thus, Boolean flag terminate is initialized to False ( **line 4**) and set to True ( **line 14**) when the program is to terminate.

Stage 1 Testing
We give output from the testing of this version of the program in Figure 3-28.

```
Enter month (1-12): 14
INVALID INPUT
Enter month (1-12): 1
Enter year (yyyy): 1800

1 , 1800 has 31 days
1800 is NOT a leap year
```

FIGURE 3-28
Example Output of First Stage Testing

The set of test cases for this stage of the program is given in Figure 3-29. The test cases are selected such that each month is tested within the 1800s, 1900s, and 2000s. The month of February has a number of test cases to ensure that the program is working for non-leap years (1985), “typical” leap years (1984), and exception years (1900 and 2000). The test plan also includes the “extreme” cases of January 1800 and December 2099 (the beginning and end of the range of valid months). All test cases are shown to have passed, and thus we can move on to stage 2 of the program development.

| Calendar Month | Expected Results | | Actual Results | | Evaluation |
|---|---|---|---|---|---|
| | num days | leap year | num days | leap year | |
| January 1800 | 31 | no | 31 | no | Passed |
| February 1900 | 28 | no | 28 | no | Passed |
| February 1984 | 29 | yes | 29 | yes | Passed |
| February 1985 | 28 | no | 28 | no | Passed |
| February 2000 | 29 | yes | 29 | yes | Passed |
| March 1810 | 31 | no | 31 | no | Passed |
| April 1912 | 30 | yes | 30 | yes | Passed |
| May 2015 | 31 | no | 31 | no | Passed |
| June 1825 | 30 | no | 30 | no | Passed |
| July 1928 | 31 | yes | 31 | yes | Passed |
| August 2031 | 31 | no | 31 | no | Passed |
| September 1845 | 30 | no | 30 | no | Passed |
| October 1947 | 31 | no | 31 | no | Passed |
| November 2053 | 30 | no | 30 | no | Passed |
| December 2099 | 31 | no | 31 | no | Passed |

FIGURE 3-29 Results of Execution of Test Plan for Stage 1

Stage 2—Determining the Day of the Week

We give the next stage of the program in Figure 3-30. This version includes the code for determining the day of the week for the first day of a given month and year ( **lines 40–71**), with the final print statement ( **line 74**) displaying the test results. Note that for testing purposes, there is no need to convert the day number into the actual name (e.g., “Monday”)—this “raw output” is good enough. Also, for this program, we will need to determine only the day of the week for the first day of any

```
# Calendar Month Program (stage 2)

# init
terminate = False

# program greeting
print('This program will display a calendar month between 1800 and 2099'

while not terminate:
    # get month and year
    month = int(input('Enter month (1-12): '))

    if month == -1:
        terminate = True
    else:
        while month < 1 or month > 12:
            month = int(input('INVALID INPUT - Enter month 1-12: '))

        year = int(input('Enter year (yyyy): '))

        while year < 1800 or year > 2099:
            year = int(input('INVALID - Enter year (1800-2099): '))

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

        # determine num of days in month
        if month in (1,3,5,7,8,10,12):
            num_days_in_month = 31
        elif month in (4,6,9,11):
            num_days_in_month = 30
        elif leap_year:  # February
            num_days_in_month = 29
        else:
            num_days_in_month = 28

        # determine day of the week
        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

```

FIGURE 3-30 Second Stage of Calendar Month Program (*Continued*)

given month, since all remaining days follow sequentially. Therefore, the day value in the day of the week algorithm part of the code is hard-coded to 1 (on **line 71**). Let's look at the code that implements the day of the week algorithm.

The algorithm operates separately on the first two digits and last two digits of the year. On **line 41**, integer division is used to extract the first two digits of the year (for example, 1860 // 100

```
51          if month == 1 and not leap_year:
52              value = value + 1
53          elif month == 2:
54              if leap_year:
55                  value = value + 3
56              else:
57                  value = value + 4
58          elif month == 3 or month == 11:
59              value = value + 4
60          elif month == 5:
61              value = value + 2
62          elif month == 6:
63              value = value + 5
64          elif month == 8:
65              value = value + 3
66          elif month == 9 or month == 12:
67              value = value + 6
68          elif month == 10:
69              value = value + 1
70
71          day_of_week = (value + 1) % 7  # 1-Sunday, 2-Monday, ...,
72
73          # display results
74          print('Day of the week is', day_of_week)
```

FIGURE 3-30 Second Stage of Calendar Month Program

equals 18). On **line 42**, the modulus operator, %, is used to extract the last two digits (for example, 1860 % 100 equals 60). The rest of the program (through **line 71**) follows the day of the week algorithm given above.

Stage 2 Testing

We give a sample test run of this version of the program in Figure 3-31.

```
1
Enter month (1-12): 4
Enter year (yyyy): 1860
Day of the week is 1
Enter month (1-12): -1
>>>
```

FIGURE 3-31 Example Output of Second Stage Testing

Figure 3-32 shows the results of the execution of the test plan for this version of the program. It includes the same months as in the test plan for the first stage. Since all test cases passed, we can move on to the final stage of program development.

Final Stage—Displaying the Calendar Month

In the final stage of the program (Figure 3-33), we add the code for displaying the calendar month. The corresponding name for the month number is determined on **lines 74–97** and displayed
( **line 100**). The while loop at **line 113** moves the cursor to the proper starting column by “printing”

| Calendar Month | Expected Results first day of month | Actual Results first day of month | Evaluation |
|---|---|---|---|
| January 1800 | 4 (Wednesday) | 4 | Passed |
| February 1900 | 5 (Thursday) | 5 | Passed |
| February 1984 | 4 (Wednesday) | 4 | Passed |
| February 1985 | 6 (Friday) | 6 | Passed |
| February 2000 | 3 (Tuesday) | 3 | Passed |
| March 1810 | 5 (Thursday) | 5 | Passed |
| April 1912 | 2 (Monday) | 2 | Passed |
| May 2015 | 6 (Friday) | 6 | Passed |
| June 1825 | 4 (Wednesday) | 4 | Passed |
| July 1928 | 1 (Sunday) | 1 | Passed |
| August 2031 | 6 (Friday) | 6 | Passed |
| September 1845 | 2 (Monday) | 2 | Passed |
| October 1947 | 4 (Wednesday) | 4 | Passed |
| November 2053 | 0 (Saturday) | 0 | Passed |
| December 2099 | 3 (Tuesday) | 3 | Passed |

FIGURE 3-32 Results of Execution of Test Plan for Stage 2

```
# Calendar Month Program

# init
terminate = False

# program greeting
print('This program will display a calendar month between 1800 and 2099')

while not terminate:
    # get month and year
    month = int(input('Enter month 1-12 (-1 to quit): '))

    if month == -1:
        terminate = True
    else:
        while month < 1 or month > 12:
            month = int(input('INVALID - Enter month (1-12): '))

        year = int(input('Enter year (yyyy): '))

        while year < 1800 or year > 2099:
            year = int(input('INVALID - Enter year (1800-2099): '))

```

FIGURE 3-33 Final Stage of Calendar Month Program (*Continued*)

```
24          # determine if leap year
25          if (year % 4 == 0) and (not(year % 100 == 0) or (year % 400 == 0)):
26              leap_year = True
27          else:
28              leap_year = False
29
30          # determine num of days in month
31          if month in (1,3,5,7,8,10,12):
32              num_days_in_month = 31
33          elif month in (4,6,9,11):
34              num_days_in_month = 30
35          elif leap_year:  # February
36              num_days_in_month = 29
37          else:
38              num_days_in_month = 28
39
40          # determine day of the week
41          century_digits = year // 100
42          year_digits = year % 100
43
44          value = year_digits + (year_digits // 4)
45
46          if century_digits == 18:
47              value = value + 2
48          elif century_digits == 20:
49              value = value + 6
50
51          if month == 1 and not leap_year:
52              value = value + 1
53          elif month == 2:
54              if leap_year:
55                  value = value + 3
56              else:
57                  value = value + 4
58          elif month == 3 or month == 11:
59              value = value + 4
60          elif month == 5:
61              value = value + 2
62          elif month == 6:
63              value = value + 5
64          elif month == 8:
65              value = value + 3
66          elif month == 9 or month == 12:
67              value = value + 6
68          elif month == 10:
69              value = value + 1
70
71          day_of_week = (value + 1) % 7  # 1-Sun, 2-Mon, ..., 0-Sat
72
```

FIGURE 3-33 Final Stage of Calendar Month Program (*Continued*)

the column_width number of blank characters (4) for each column to be skipped. The while loop at **line 119** displays the dates. Single-digit dates are output ( **line 121**) with three leading spaces, and two-digit dates with two ( **line 123**) so that the columns line up. Each uses the **newline suppression form of print**, print(..., end 5") to prevent the cursor from moving to the next screen line until it is time to do so.

Variable current_day is incremented from 1 to the number of days in the month. Variable current_col is also incremented by 1 to keep track of what column the current date is being

```
        # determine month name
        if month == 1:
            month_name = 'January'
        elif month == 2:
            month_name = 'February'
        elif month == 3:
            month_name = 'March'
        elif month == 4:
            month_name = 'April'
        elif month == 5:
            month_name = 'May'
        elif month == 6:
            month_name = 'June'
        elif month == 7:
            month_name = 'July'
        elif month == 8:
            month_name = 'August'
        elif month == 9:
            month_name = 'September'
        elif month == 10:
            month_name = 'October'
        elif month == 11:
            month_name = 'November'
        else:
            month_name = 'December'

        # display month and year heading
        print('\n', ' ' + month_name, year)

        # display rows of dates
        if day_of_week == 0:
            starting_col = 7
        else:
            starting_col = day_of_week

        current_col = 1
        column_width = 4
        blank_char = ' '
        blank_column = format(blank_char, str(column_width))

        while current_col <= starting_col:
            print(blank_column, end='')
            current_col = current_col + 1

        current_day = 1

        while current_day <= num_days_in_month:
            if current_day < 10:
                print (format(blank_char, '3') + str(current_day), end='')
            else:
                print (format(blank_char, '2') + str(current_day), end='')

            if current_col <= 7:
                current_col = current_col + 1
            else:
                current_col = 1
                print()

            current_day = current_day + 1

        print('\n')
```

FIGURE 3-33 Final Stage of Calendar Month Program

displayed in. When current_col equals 7, it is reset to 1 ( **line 128**) and print() moves the cursor to the start of the next line ( **line 129**). Otherwise, current_col is simply incremented by 1 ( **line 126**).

An example test run of this final version of the program is given in Figure 3-34.

```
This program will display a calendar month between 1800 and 2099

Enter month 1-12 (-1 to quit): 1
Enter year (yyyy): 1800

  January   1800
                    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

Enter month (1-12): -1
>>>
```

FIGURE 3-34 Example Output of Final Stage Testing

*Something is obviously wrong* . The calendar month is displayed with eight columns instead of seven. The testing of all other months produces the same results. Since the first two stages of the program were successfully tested, the problem must be in the code added in the final stage. The code at **line 74** simply assigns the month name. Therefore, we reflect on the logic of the code starting on **line 103**.

**Lines 128–129** is where the column is reset back to column 1 and a new screen line is started, based on the current value of variable current_col,

if current_col ,5 7:
current_col 5 current_col 1 1
else:
**current_col** 5 **1**
**print()**

V ariable current_col is initialized to 1 at **line 108**, and is advanced to the proper starting column on **lines 113–115**. Variable starting_col is set to the value (0-6) for the day of the week for the particular month being displayed. Since the day

of the week results have been successfully tested, we can assume that current_col will have a value between 0 and 6. With that assumption, we can step though **lines 125–129** and see if this is where the problem is. Stepping through a program on paper by tracking the values of variables is referred to as **deskchecking**. We check what happens as the value of current_col approaches 7, as shown in Figure 3-35.

Now it is clear what the problem is—the classic *"off by one" error*! The condition of the while loop should be current_col ,7, *not*current_col ,57. Current_col should be reset to 1 once the seventh column has been displayed (when current_col is 7). Using the

| Current value of `current_col` | Value of condition `current_col <= 7` | Updated value of `current_col` |
|---|---|---|
| 5 | True | 6 |
| 6 | True | 7 |
| 7 | True | 8 |
| 8 | False | 1 |
| 1 | True | 2 |
| etc. | | |

FIGURE 3-35 Deskchecking the Value of Variable current_col

,5 operator causes current_col to be reset to 1 only after an *eighth* column is displayed. Thus, we make this correction in the program,

**if current_col , 7:**
current_col 5 current_col 1 1
else:
current_col 5 1
print()

After re-executing the program with this correction we get the current output, depicted in Figure 3-36.

```
This program will display a calendar month between 1800 and 2099

Enter month 1-12 (-1 to quit): 1
Enter year (yyyy): 1800

  January   1800
                    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

Enter month (1-12): -1
>>>
```

FIGURE 3-36 Display Output of Final Stage of Calendar Month Program

Although the column error has been corrected, we find that the first of the month appears under the wrong column—the month should start on a Wednesday (fourth column), not a Thursday column (fifth column). The problem must be in how the first row of the month is displayed. Other months are tested, each found to be off by one day. We therefore look at **lines 113–115** that are responsible for moving over the cursor to the correct starting column,

while current_col ,5 starting_col:
print(blank_column, end 5")
current_col 5 current_col 1 1



We consider whether there is another “off by one” error. Reconsidering the condition of the while loop, we realize that, in fact, this is the error. If the correct starting column is 4 (Wednesday), then the cursor should move past three columns and place a 1 in the fourth column. The current condition, however, would move the cursor past *four* columns, thus placing a 1 in the fifth column (Thursday). The corrected code is given below.

**while current_col , starting_col:** print(' ', end ")
current_col 5 current_col 1 1

The month is now correctly displayed. We complete the testing by executing the program on a set of test cases (Figure 3-37). Although the test plan is not as complete as it could be, it includes test cases for months from each century,

including both leap years and non-leap years.

| Calendar Month | Expected Results | | Evaluation |
|---|---|---|---|
| | first day of month | num days | |
| April 1912 | Sunday | 30 | Passed |
| February 1985 | Monday | 28 | Passed |
| May 2015 | Tuesday | 31 | Passed |
| January 1800 | Wednesday | 31 | Passed |
| February 1900 | Thursday | 28 | Passed |
| February 1984 | Friday | 29 | Passed |
| January 2011 | Saturday | 31 | Passed |

FIGURE 3-37 Results of Execution of Test Plan for Final Stage

CHAPTER SUMMARY General Topics

Control Statement/Control Structure
Sequential, Selection, and Iterative Control Relational Operators/Boolean Operators/

Boolean Expressions
Operator Precedence and Boolean Expressions Logically Equivalent Boolean Expressions Short-Circuit (Lazy) Evaluation
Selection Control Statements/if Statement Compound Statement
Multi-way Selection
While Statements
Input Error Checking
Infinite Loops
Definite vs. Indefinite Loops
Boolean Flags and Indefinite Loops
Deskchecking

Python-Specific Programming Topics

Membership Operators in, not in
if Statement in Python/else and elif headers Indentation in Python

Multi-way Selection in Python
while Statement in Python

CHAPTER EXERCISES

Section 3.1
**1.** Which of the three forms of control is an implicit form of control?
**2.** What is meant by a “straight-line” program?
**3.** What is the difference between a control statement and a control structure?

Section 3.2

**4.** The Boolean data type contains two literal values, denoted as _____ and _____ in Python.
**5.** Which of the following relational expressions evaluate to True?
**(a)** 5 , 8 **(c)** '10' , '8'
**(b)**'5' , '8' **(d)** 'Jake' , 'Brian'
**6.** Which of the following relational expressions evaluate to False?
**(a)** 5 ,5 5 **(c)** 5 55 5 **(e)** 5 !5 10
**(b)** .5 5 **(d)** 5 !5 5
**7.** Give an appropriate expression for each of the following.
**(a)** To determine if the number 24 does *not* appear in a given list of numbers assigned to variable nums. **(b)** To determine if the name 'Ellen' appears in a list of names assigned to variable names. **(c)** To determine if a single last name stored in variable last_name is either 'Morris' or 'Morrison'.
**8.** Evaluate the following Python expressions.
**(a)** (12 * 2) 55 (3 * 8)
**(b)**(14 * 2) !5 (3 * 8)
**9.** What value for x makes each of the following Boolean expressions true?
**(a)** x or False
**(b)**x and True
**(c)** not (x or False)
**(d)**not (x and True)
**10.** Evaluate the Boolean expressions below for n 5 10 and k 5 20.
**(a)** (n . 10) and (k 55 20)
**(b)**(n . 10) or (k 55 20)
**(c)** not((n . 10) and (k 55 20))
**(d)**not(n . 10) and not(k 55 20)
**(e)** (n .10) or (k 5510 or k !5 5)

**11.** Give an appropriate Boolean expression for each of the following.
**(a)** Determine if variable num is greater than or equal to 0, and less than 100.
**(b)** Determine if variable num is less than 100 and greater than or equal to 0, or it is equal to 200. **(c)** Determine if either the name 'Thompson' or 'Wu' appears in a list of names assigned to variable last_names.
**(d)** Determine if the name 'Thomson' appears and the name 'Wu' does not appear in a list of last names assigned to variable last_names.
**12.** Evaluate the following Boolean expressions for num1 5 10 and num2 5 20.
**(a)** not (num1 ,1) and num2 , 10
**(b)**not (num1 , 1) and num2 , 10 or num1 1 num3 , 100

Chapter Exercises 119

**13.** Give a logically equivalent expression for each of the following. **(a)** num !5 25 or num 55 0
**(b)**1 ,5 num and num ,5 50
**(c)** not num . 100 and not num , 0
**(d)**(num , 0 or num . 100)

Section 3.3
**14.** Give an appropriate if statement for each of the following.
**(a)** An if statement that displays 'within range' if num is between 0 and 100, inclusive. **(b)** An if statement that displays 'within range' if num is between 0 and 100, inclusive, and displays 'out of range' otherwise.

**15.** Rewrite the following if-else statements using a single if statement and elif headers.

```
if temperature . 5 85 and humidity . 60: print('muggy day today')
else:
if temperature . 5 85:
print('warm, but not muggy today') else:
if temperature . 5 65:
print('pleasant today')
else:
if temperature , 5 45:
print('cold today')
else:
print('cool today')
```

**16.** Regarding proper indentation,
**(a)** Explain the change in indentation needed in order for the following code to be syntactically correct. **(b)** Indicate other changes in the indentation of the code that is not strictly needed, but would make the

code more readable.

if level ,5 1:
print('Value is well within range')
print('Recheck in one year')

elif level ,5 2:
print('Value is within range')
print('Recheck within one month')

elif level ,5 3:
print('Value is slightly high)
print('Recheck in one week')

elif level ,5 4:
print('Value abnormally high')
print('Shut down system immediately')

Section 3.4
**17.** Write a program segment that uses a while loop to add up all the even numbers between 100 and 200, inclusive.
**18.** The following while loop is meant to multiply a series of integers input by the user, until a sentinel value of 0 is entered. Indicate any errors in the code given.

product 5 1
num 5 input('Enter first number: ') while num ! 5 0:

num 5 input('Enter first number: ') product 5 product * num
print('product 5 ', product)
**19.** For each of the following, indicate which is a definite loop, and which is an indefinite loop. **(a)** num 5 input('Enter a non-zero value: ')
while num 5 5 0:
num 5 input('Enter a non-zero value: ')

**(b)** num 5 0
while n , 10:
print 2 ** n
n 5 n 1 1

PYTHON PROGRAMMING EXERCISES

**P1.** Write a Python program in which the user enters either 'A', 'B', or 'C'. If 'A' is entered, the program should display the word 'Apple'; if 'B' is entered, it displays 'Banana'; and if 'C' is entered, it displays 'Coconut'. Use nested if statements for this as depicted in Figure 3-13.

**P2.** Repeat question P1 using an if statement with elif headers instead.

**P3.** Write a Python program in which a student enters the number of college credits earned. If the number of credits is greater than 90, 'Senior Status' is displayed; if greater than 60, 'Junior Status' is displayed; if greater than 30, 'Sophomore Status' is displayed; else, 'Freshman Status' is displayed.

**P4.** Write a program that sums a series of (positive) integers entered by the user, excluding all numbers that are greater than 100.
**P5.** Write a program, in which the user can enter any number of positive and negative integer values, that displays the number of positive values entered, as well as the number of negative values.
**P6.** Write a program containing a pair of nested while loops that displays the integer values 1–100, ten numbers per row, with the columns aligned as shown below,

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
.
.
91 92 93 94 95 96 97 98 99 100

**P7.** Display the integer values 1–100 as given in question P6 using only *one* while loop.
Program Modification Problems 121
PROGRAM MODIFICATION PROBLEMS

**M1.** Temperature Conversion Program: Input Error Checking
Modify the Temperature Conversion program in Figure 3-19 to perform input error checking of entered temperatures. On the Fahrenheit scale, absolute zero is 2459.67. Therefore, all valid Fahrenheit temperatures start at that value (with no upper limit). On the Celsius scale, absolute zero is 2273.15. The program should reprompt the user for any invalid entered temperatures.

**M 2.** Temperature Conversion Program: Addition of Kelvin Scale
Modify the Temperature Conversion program in Figure 3-19 to add an additional option of converting to and from degrees Kelvin. The formula for conversion to Kelvin (K) from Celsius (C) is K 5 C 1 273.15.

**M3.** Number of Days in Month Program: Input Error Checking
Modify the Number of Days in Month Program of section 3.3.4 so that the program prompts the user to re-enter any month (not in the range 1–12) or year that is an invalid value.

**M4.** Number of Days in Month Program: Indication of Leap Years
Modify the Number of Days in Month program of section 3.3.4 so that the program displays, in addition to the number of days in the month, that the year is a leap year or not as shown below.

Enter the month (1-12): 2
Please enter the year (e.g., 2010): 2000 There are 29 days in the month (a leap year)

**M5.** Oil Change Notification Program: Number of Miles before Change
Modify the Oil Change Notification program in Figure 3-21 so that the program displays the number of miles left before the next oil change, or the number of miles overdue for an oil change, as appropriate.

**M6.** Coin Change Exercise Program: Addition of Half-Dollar Coins
Modify the Coin Change Exercise program in section 3.4.6 to allow for the use of half-dollar coins. Make all necessary changes in the program.

**M7.** Coin Change Exercise Program: Raising the Challenge
Modify the Coin Change Exercise program in section 3.4.6 so that the least possible number of coins must be entered. For example, the least number of coins that total to 43 cents is 6 (one quarter, one dime, one nickel, and three pennies).

**M8.** Calendar Month Program: Indication of Leap Year
Modify the final version of the Calendar Month program in section 3.5 so that for leap years, the month heading is displayed as in the following,

```
June   1984 (leap year)

                        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
```

**M9.** Calendar Month Program: User Entry of Month Name
Modify the final version of the Calendar Month program to allow the user to enter a month's name (e.g., 'January') rather than a number (e.g., 1). Make all appropriate changes in the program as a result of this change.
**M10.** Calendar Month Program: Day of the Week Headings

Modify the final version of the Calendar Month program in section 3.5 so that there is day heading for each of the columns as shown below.

```
January   1800

 Su  Mo  Tu  We  Th  Fr  Sa
             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
```

**M11.** Sage Program Modification
Following is the output of an "all knowing" Sage program that replies with random responses to questions posed by the user. The responses generated have no meaningful connection to the questions asked.

```
What is your question?:
When will I finally finish the book?

You ask me "When will I finally finish the book?" . . .
You know the answer to that already, don't you?

Do you have another question? (y/n): y
What is your question?:
Do you really know what I am asking?

You ask me "Do you really know what I am asking?" . . .
I would focus my thoughts on something else.

Do you have another question? (y/n): y
What is your question?:
Can't you answer my questions directly?

You ask me "Can't you answer my questions directly?" . . .
The probabilities are in your favor.

Do you have another question? (y/n): y
What is your question?:
So, what's the difference between an orange?

You ask me "So, what's the difference between an orange?" . . .
It is close to certainty.

Do you have another question? (y/n): y
What is your question?:
I think you are bogus, aren't you?

You ask me "I think you are bogus, aren't you?" . . .
Someone you would not expect can be most helpful about this.

Do you have another question? (y/n): n
Goodbye ... hope I was of some help!
>>>
```

Program Development Problems 123

In the 1960s, a program called Eliza was developed that was able to behave as a psychotherapist. It did not really understand anything, it only looked for certain words to turn the patient's comments or questions back to the patient. For example, if a patient said, "My mom drives me crazy," it might reply with "Tell me more about your mom." Modify this program so that it appears to have understanding by similar means of word recognition as used in the Eliza program. Specifically, incorporate a set of "trigger" words that, if found, causes a specific response to be given. For example, if the word "I" appears in the question (for example, "Will I ever be rich?" or "Am I always going to be

happy?"), the response may be "You are in charge of your own destiny." If the word "new" appears in the question (for example, "Will I find a new boyfriend soon?" or "Will I find a new life?," the response may be "Changes are up to you and the unpredictable events in life."

Be creative. In order to determine if a given word (or phrase) appears in a given question, make use of the in membership operator.

```
# Fortune Teller Program
import random

have_question = 'y'

while have_question == 'y':
    question = input('What is your question?:\n')
    print('\nYou ask me', '"' + question + '" . . .')

    rand_num = random.randint(1,8)

    if rand_num == 1:
        print ('The probabilities are in your favor')
    elif rand_num == 2:
        print ("I wouldn't make any definite plans")
    elif rand_num == 3:
        print ('The outlook is dim')
    elif rand_num == 4:
        print ('I would focus my thoughts on something else')
    elif rand_num == 5:
        print ('You are the only one that can answer that!')
    elif rand_num == 6:
        print ("You know the answer to that already, don't you?")
    elif rand_num == 7:
        print ('Someone unexpected can be most helpful with this')
    elif rand_num == 8:
        print ('It is close to certainty')

    have_question = input('\nDo you have another question? (y/n): ')

print('Goodbye ... hope I was of some help!')
```

PROGRAM DEVELOPMENT PROBLEMS

**D 1.** Metric Conversion
Develop and test a Python program that converts pounds to grams, inches to centimeters, and kilometers to miles. The program should allow conversions both ways.

**D2.** Leap Years to Come
Develop and test a Python program that displays future leap years, starting with

the first occurring leap year from the current year, until a final year entered by the user. (HINT: Module datetime used in the Age in Seconds Program of Chapter 2 will be needed here.)

**D3.** The First-Time Home Buyer Tax Credit
Develop and test a Python program that determines if an individual qualifies for a government First-Time Home Buyer Tax Credit of $8,000. The credit was only available to those that (a) bought a house that cost less than $800,000, (b) had a combined income of under $225,000 and (c) had not owned a primary residence in the last three years.

**D4.** Home Loan Amortization
Develop and test a Python program that calculates the monthly mortgage payments for a given loan amount, term (number of years) and range of interest rates from 3% to 18%. The fundamental formula for determining this is A/D, where A is the original loan amount, and D is the discount factor. The discount factor is calculated as,

D 5 ((1 1 r) 2 1) / r(1 1 r)

where is the number of total payments (12 times the number of years of the loan) and r is the interest rate, expressed in decimal form (e.g., .05), divided by 12. A monthly payment table should be generated as shown below,

Loan Amount: $350,000 Term: 30 years Interest Rate Monthly Payment 3% 1475.61 4% 1670.95 5% 1878.88 6% 2098.43 . .
. .
18% 5274.80

Check your results with an online mortgage calculator.
**D5.** Life Signs
Develop and test a program that determines how many breaths and how many heartbeats a person has had
in their life. The average respiration (breath) rate of people varies with age. Use the breath rates given
below for use in your program,

Breaths per Minute
Infant 25–60 1–4 years 20–30
5–14 years 15–25

15–18 years 11–23

For heart rate, use an average of 67.5 beats per second.

## Lists CHAPTER 4

*In this chapter, we look at a means of structuring and accessing a collection of data. In particular, we look at a way of organizing data in a linear sequence, generally referred to as a list.*

OBJECTIVES
After reading this chapter and completing the exercises, you will be able to: ♦ Explain what a list is in programming
♦ Describe the typical operations performed on lists
♦ Explain what is meant by list traversal
♦ Effectively create and use lists in Python
♦ Explain the difference between lists and tuples in Python
♦ Explain what a sequence is in Python
♦ Describe the sequence operations common to lists, tuples, and strings in Python ♦ Effectively use nested lists and tuples in Python
♦ Effectively iterate over lists (sequences) in Python
♦ Effectively use for statements for iterative control in Python
♦ Use the range function in Python
♦ Explain how list representation relates to list assignment in Python ♦ Effectively use list comprehensions in Python
♦ Write Python programs using sequences

CHAPTER CONTENTS

MOTIVATION

The way that data is organized has a significant impact on how effec tively it can be used. One of the most obvious and useful ways to organize data is as a list. We use lists in our everyday lives—we make shopping lists, to-do lists, and mental checklists. Various forms of lists are provided by programming languages, differing in the ele ments they can store (mixed type?), their size (variable size?), whether they can be altered (mutable?), and the operations that can be per formed on them (see Figure 4-1).

Lists also occur in nature. Our DNA is essentially a long list of molecules in the form of a double helix, found in the nucleus of all human cells and all living organisms. Its purpose is also to store information—specifically, the instructions that are used to construct all other cells in the body—that we call *genes*. Given the 2.85 billion nucleotides that make up the human genome, determining their sequencing (and thus understanding our genetic makeup) is fundamentally a computational problem.

In this chapter, we look at the use of lists and other sequences in Python.

| List Characteristics | Elements |
|---|---|
| **Element Type** | All elements of the same type |
| | Elements of different types |
| **Length** | Fixed length |
| | Varying length |
| **Modifiability** | Mutable (alterable) |
| | Immutable (unalterable) |
| **Common Operations** | Determine if a list is empty |
| | Determine the length of a list |
| | Access (retrieve) elements of a list |
| | Insert elements into a list |
| | Replace elements of a list |
| | Delete elements of a list |
| | Append elements to (the end of) a list |

FIGURE 4-1 List Properties and Common Operations

## 4.1 List Structures

In this section we introduce the use of lists in programming. The concept of a list is similar to our everyday notion of a list. We read off (access) items on our to-do list, add items, cross off (delete) items, and so forth. We look at the use of lists next.

### 4.1.1 What Is a List?

A **list** is a *linear data structure*, meaning that its elements have a linear ordering. That is, there is a first element, a second element, and so on. Figure 4-2 depicts a list storing the average temperature for each day of a given week, in which each item in the list is identified by its *index value*.

The location at index 0 stores the temperature for Sunday, the location at index 1 stores the temperature for Monday, and so on. It is customary in programming languages to begin numbering sequences of items with an index value of 0 rather than 1. This is referred to as *zero-based indexing*. This is important to keep in mind to avoid any “off by one” errors in programs, as we shall see. We next look at some common operations performed on lists.

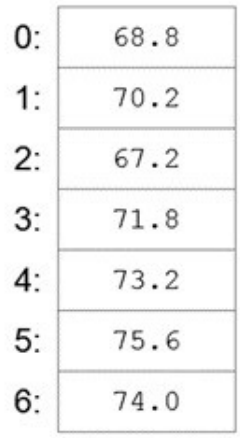



FIGURE 4-2
Indexed Data
Structure daily_temperatures

A **list** is a linear data structure, thus its elements have a linear ordering.

### 4.1.2 Common List Operations

Operations commonly performed on lists include retrieve, update, insert, delete

(remove) and append. Figure 4-3 depicts these operations on a list of integers.

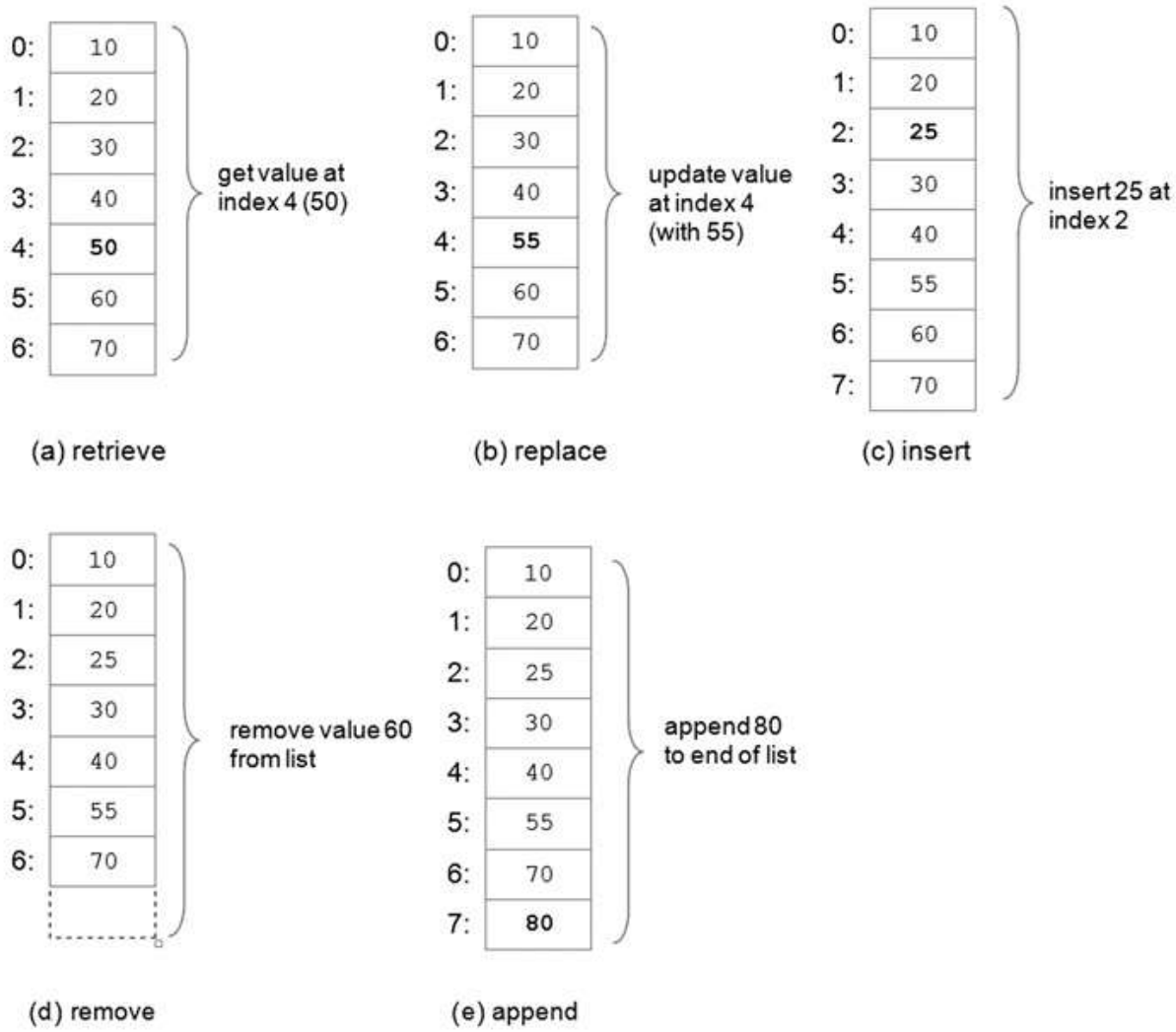



FIGURE 4-3 Common List Operations

The operation depicted in (a) retrieves elements of a list by index value. Thus, the value 50 is retrieved at index 4 (the fifth item in the list). The replace operation in (b) updates the current value at index 4, 50, with 55. The insert operation in (c) inserts the new value 25 at index 2, thus shifting down all elements below that point and lengthening the list by one. In (d), the remove operation deletes the element at index 6, thus shifting up all elements below that point and shortening the list by one. Finally, the append operation in (e) adds a new value, 80, to the end of the list. In the following sections we will see how these operations are accomplished in Python. First, we look at what is called *list traversal*, a way of accessing each of the elements of a given list.

Operations commonly performed on lists include retrieve, update, insert, remove, and append.

4.1.3 List Traversal

A **list traversal** is a means of accessing, one-by-one, the elements of a list. For example, to add up all the elements in a list of integers, each element can be accessed one-by-one, starting with the first, and ending with the last element. Similarly, the list could be traversed starting with the last element and ending with the first. To find a particular value in a list also requires traversal. We depict the tasks of summing and searching a list in Figure 4-4.

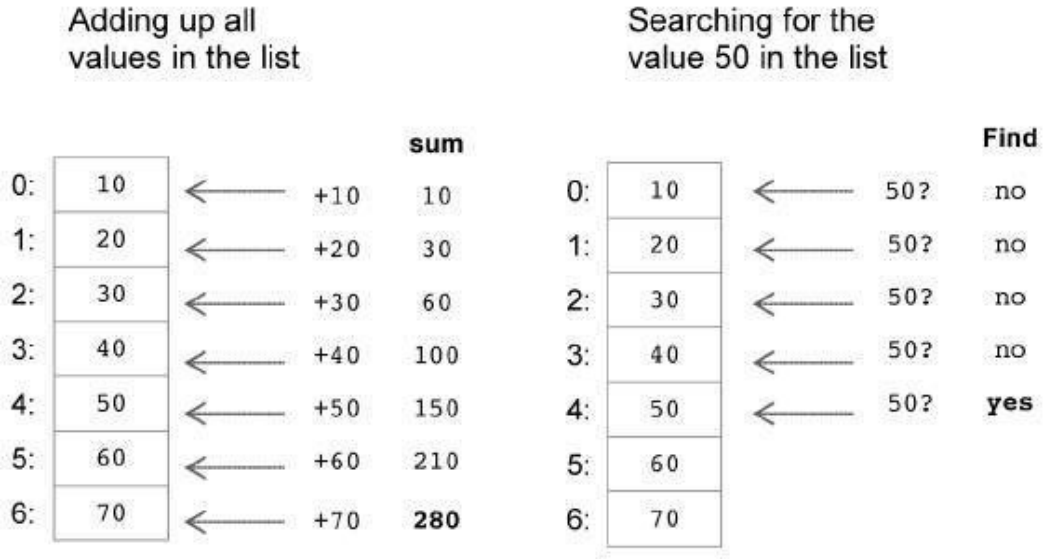



FIGURE 4-4 List Traversal

A **list traversal** is a means of accessing, one-by-one, the elements of a list.

Self-Test Questions

**1.** What would be the range of index values for a list of 10 elements?
**(a)** 0–9 **(b)** 0–10 **(c)** 1–10

**2.** Which one of the following is NOT a common operation on lists?
**(a)** access **(b)** replace **(c)** interleave **(d)** append **(e)** insert **(f )** delete

**3.** Which of the following would be the resulting list after inserting the value 50 at index 2?

0: 35
1: 15
2: 45

3: 28

(a) 0: 35 (b) 0: 35 (c) 0: 50
1: 50 1: 15 1: 35
2: 15 2: 50 2: 15
3: 45 3: 45 3: 45
4: 28 4: 28 4: 28

ANSWERS: 1. a, 2. c, 3. b

## 4.2 Lists (Sequences) in Python

Next, we look at lists (and other sequence types) in Python.

### 4.2.1 Python List Type

A **list** in Python is a mutable, linear data structure of variable length, allowing mixed-type elements. *Mutable* means that the contents of the list may be altered. Lists in Python use zerobased indexing. Thus, all lists have index values 0 ... n-1, where n is the number of elements in the list. Lists are denoted by a comma-separated list of elements within square brackets as shown below,

[1, 2, 3] ['one', 'two', 'three'] ['apples', 50, True]

An **empty list** is denoted by an empty pair of square brackets, []. (We shall later see the usefulness of the empty list.) Elements of a list are accessed by using an index value within square brackets,

lst 5 [1, 2, 3] lst[0] → 1 access of first element lst[1] → 2 access of second element lst[2] → 3 access of third element

Thus, for example, the following prints the first element of list lst,
print(lst[0])
The elements in list lst can be summed as follows,
sum 5 lst[0] 1 lst[1] 1 lst[2]

For longer lists, we would want to have a more concise way of traversing the elements. We discuss this below. Elements of a list can be updated (replaced) or deleted (removed) as follows (for lst 5 [1, 2, 3]),

lst[2] 5 4 [1, 2, 4] replacement of 3 with 4 at index 2 del lst[2] [1, 2] removal of 4 at index 2
Methods insert and append also provide a means of altering a list,

lst.insert(1, 3) [1, 3, 2] insertion of 3 at index 1 lst.append(4) [1, 3, 2, 4] appending of 4 to end of list
In addition, methods sort and reverse reorder the elements of a given list. These list modifying operations are summarized in Figure 4-5.

| Operation | fruit = ['banana', 'apple, 'cherry'] | |
|---|---|---|
| Replace | `fruit[2] = 'coconut'` | `['banana', 'apple', 'coconut']` |
| Delete | `del fruit[1]` | `['banana', 'cherry']` |
| Insert | `fruit.insert(2, 'pear')` | `['banana', 'apple', 'pear', 'cherry']` |
| Append | `fruit.append('peach')` | `['banana', 'apple', 'cherry', 'peach']` |
| Sort | `fruit.sort()` | `['apple', 'banana', 'cherry']` |
| Reverse | `fruit.reverse()` | `['cherry', 'banana', 'apple']` |

FIGURE 4-5 List Modification Operations in Python

*Methods* , and the associated *dot notation* used, are fully explained in Chapter 6 on Objects and Their Use. We only mention methods here for the sake of completeness in covering the topic of list operations.

LET'S TRY IT From the Python Shell, enter the following and observe the results.

... lst 5 [10, 20, 30] ... del lst[2] ... lst ... lst
??? ???

... lst[0] ... lst.insert(1, 15) ??? ... lst
???

... lst[0] 5 5 ... lst.append(40) ... lst ... lst
??? ???

A **list** in Python is a mutable linear data structure, denoted by a comma-separated list of elements within square brackets, allowing mixed-type elements.
4.2.2 Tuples

A **tuple** is an *immutable* linear data structure. Thus, in contrast to lists, once a tuple is defined, it cannot be altered. Otherwise, tuples and lists are essentially the same. To distinguish tuples from lists, tuples are denoted by parentheses instead of square brackets as given below,

nums 5 (10, 20, 30)
student 5 ('John Smith', 48, 'Computer Science', 3.42)

Another difference between tuples and lists is that *tuples of one element must include a comma following the element.* Otherwise, the parenthesized element will not be made into a tuple, as shown below,

CORRECT WRONG ... (1,) ... (1)
(1) 1

An **empty tuple** is represented by a set of empty parentheses, (). (We shall later see the usefulness of the empty tuple.) The elements of tuples are accessed the same as lists, with square brackets,

... nums[0] ... student[0]
10 'John Smith'

Any attempt to alter a tuple is invalid. Thus, delete, update, insert, and append operations are not defined on tuples. For now, we can consider using tuples when the information to represent should not be altered. We will see additional uses of tuples in the coming chapters.

LET'S TRY IT From the Python Shell, enter the following and observe the results.

... t 5 (10, 20, 30) ... t.insert(1, 15) ... t[0] ... ???
??? ???

... del t[2] ... t.append(40) ??? ???
A **tuple** in Python is an *immutable* linear data structure, denoted by a comma-separated list of elements within parentheses, allowing mixed-type elements.
4.2.3 Sequences

A **sequence** in Python is a linearly ordered set of elements accessed by an index number. Lists, tuples, and strings are all sequences. Strings, like tuples, are

immutable ; therefore, they cannot be altered. We give sequence operations common to strings, lists, and tuples in Figure 4-6.

For any sequence s, len(s)gives its length, and s[k] retrieves the element at index k. The slice operation, s[index1:index2], returns a subsequence of a sequence, starting with the first index location up to *but not including* the second. The s[index:] form of the slice operation returns a string containing all the list elements starting from the given index location to the end of the sequence. The count method returns how many instances of a given value occur within a sequence, and the find method returns the index location of the *first occurrence* of a specific item, returning 21 if not found. For determining only if a given value occurs within a

| **Operation** | | **String**<br>s = 'hello' w = '!' | **Tuple**<br>s = (1,2,3,4) w = (5,6) | **List**<br>s = [1,2,3,4] w = [5,6] |
|---|---|---|---|---|
| Length | `len(s)` | `5` | `4` | `4` |
| Select | `s[0]` | `'h'` | `1` | `1` |
| Slice | `s[1:4]`<br>`s[1:]` | `'ell'`<br>`'ello'` | `(2, 3, 4)`<br>`(2, 3 ,4)` | `[2 ,3, 4]`<br>`[2, 3, 4]` |
| Count | `s.count('e')`<br>`s.count(4)` | `1`<br>*error* | `0`<br>`1` | `0`<br>`1` |
| Index | `s.index('e')`<br>`s.index(3)` | `1`<br>`--` | `--`<br>`2` | `--`<br>`2` |
| Membership | `'h' in s` | `True` | `False` | `False` |
| Concatenation | `s + w` | `'hello!'` | `(1, 2, 3, 4, 5, 6)` | `[1, 2, 3, 4, 5, 6]` |
| Minimum Value | `min(s)` | `'e'` | `1` | `1` |
| Maximum Value | `max(s)` | `'o'` | `4` | `4` |
| Sum | `sum(s)` | *error* | `10` | `10` |

FIGURE 4-6 Sequence Operations in Python

sequence, without needing to know where, the in operator (introduced in Chapter 3) can be used instead.

The 1 operator is used to denote concatenation. Since the plus sign also denotes addition, Python determines which operation to perform based on the operand types. Thus the plus sign, 1, is referred to as an **overloaded operator**. If both operands are numeric types, addition is performed. If both operands are

sequence types, concatenation is performed. (If a mix of numeric and sequence operands is used, an “unsupported operand type(s) for 1” error message will occur.) Operations min/max return the smallest/largest value of a sequence, and sum returns the sum of all the elements (when of numeric type). Finally, the comparison operator, 55, returns True if the two sequences are the same length, and their corresponding elements are equal to each other.

LET’S TRY IT From the Python Shell, enter the following and observe the results.

... s 5 'coconut' ... s 5 (10, 30, 20, 10) ... s 5 [10, 30, 20, 10] ... s[4:7] ... s[1:3] ... s[1:3] ??? ??? ???

... s.count('o') ... s.count(10) ... s.count(10) ??? ??? ???
... s.index('o') ... s.index(10) ... s.index(10) ??? ??? ???
... s 1 ' juice' ... s 1 (40, 50) ... s 1 (40, 50) ??? ??? ???
In Python, a **sequence** is a linearly ordered set of elements accessed by index value. Lists, tuples, and strings are sequence types in Python.
4.2.4 Nested Lists

Lists and tuples can contain elements of any type, including other sequences. Thus, lists and tuples can be nested to create arbitrarily complex data structures. Below is a list of exam grades for each student in a given class,

class_grades 5 [ [85, 91, 89], [78, 81, 86], [62, 75, 77], ...]

In this list, for example, class_grades[0] equals [85, 91, 89], and class_grades[1] equals [78, 81, 86]. Thus, the following would access the first exam grade of the first student in the list,

student1_grades 5 class_grades[0] student1_exam1 5 student1_grades[0]
However, there is no need for intermediate variables student1_grades and student1_ exam1. The exam grade can be directly accessed as follows,
class_grades[0][0] → [85, 91, 89][0] → 85
To calculate the class average on the first exam, a while loop can be constructed that iterates over the first grade of each student’s list of grades,

sum 5 0
k 5 0
while k , len(class_grades):

sum 5 sum 1 class_grades[k][0]
k 5 k 1 1
average_exam1 5 sum / float(len(class_grades)) If we wanted to produce a new list containing the exam average for each student in the class, we could do the following,

exam_avgs 5 []
k 5 0
while k , len(class_grades):

avg 5 (class_grades[k][0] 1 class_grades[k][1] 1 \ class_grades[k][2]) / 3.0
exam_avgs.append(avg)
k 5 k 1 1

Each time through the loop, the average of the exam grades for a student is computed and appended to list exam_avgs. When the loop terminates, exam_avgs will contain the corresponding exam average for each student in the class.
LET'S TRY IT

From the Python Shell, enter the following and observe the results.

... lst 5 [[1, 2 ,3], [4, 5, 6], [7, 8, 9]]
... lst[0] ... lst[1] ??? ??? ... lst[0][1] ... lst[1][1] ??? ???

Lists and tuples can be nested within each other to construct arbitrarily complex data structures.
4.2.5 Let's Apply It—A Chinese Zodiac Program
The following program (Figure 4-8) determines the animal and associated characteristics from the Chinese Zodiac for a given year of birth. This program utilizes the following programming features:
➤ tuples ➤ datetime module
Example execution of the program is given in Figure 4-7.

```
This program will display your Chinese zodiac sign and associated
personal characteristics.

Enter your year of birth (yyyy): 1984
Your Chinese zodiac sign is the Rat

Your personal characteristics ...
Forthright, industrious, sensitive, intellectual, sociable

Would you like to enter another year? (y/n): y
Enter your year of birth (yyyy): 1986
Your Chinese zodiac sign is the Tiger

Your personal characteristics ...
Unpredictable, rebellious, passionate, daring, impulsive

Would you like to enter another year? (y/n): n
>>>
```
FIGURE 4-7 Execution of the Chinese Zodiac Program

**Line 3** imports the datetime module. It provides the current year ( **line 31**), used to check for invalid years of birth (only years between 1900 and the current year are considered valid). **Lines 9–24** perform the initialization for the program. The variables on lines **9–20** are assigned the characteristics of each animal. The set of characteristics is represented as a tuple

```
 # Chinese Zodiac Program

 import datetime

# init
zodiac_animals = ('Rat', 'Ox', 'Tiger', 'Rabbit', 'Dragon', 'Snake', 'Horse',
                  'Goat', 'Monkey', 'Rooster', 'Dog', 'Pig')

rat = 'Forthright, industrious, sensitive, intellectual, sociable'
ox = 'Dependable, methodical, modest, born leader, patient'
tiger = 'Unpredictable, rebellious, passionate, daring, impulsive'
rabbit = 'Good friend, kind, soft-spoken, cautious, artistic'
dragon = 'Strong, self-assured, proud, decisive, loyal'
snake = 'Deep thinker, creative, responsible, calm, purposeful'
horse = 'Cheerful, quick-witted, perceptive, talkative, open-minded'
goat = 'Sincere, sympathetic, shy, generous, mothering'
monkey = 'Motivator, inquisitive, flexible, innovative, problem solver'
rooster = 'Organized, self-assured, decisive, perfectionist, zealous'
dog = 'Honest, unpretentious, idealistic, moralistic, easy going'
pig = 'Peace-loving, hard-working, trusting, understanding, thoughtful'

characteristics = (rat, ox, tiger, rabbit, dragon, snake, horse, goat, monkey,
                   rooster, dog, pig)
terminate = False

# program greeting
print('This program will display your Chinese zodiac sign and associated')
print('personal characteristics.\n')

# get current year from module datetime
current_yr = datetime.date.today().year

while not terminate:

    # get year of birth
    birth_year = int(input('Enter your year of birth (yyyy): '))

    while birth_year < 1900 or birth_year > current_yr:
        print('Invalid year. Please re-enter\n')
        birth_year = int(input('Enter your year of birth (yyyy): '))

    # output results
    cycle_num = (birth_year - 1900) % 12

    print('Your Chinese zodiac sign is the', zodiac_animals[cycle_num],'\n')
    print('Your personal characteristics ...')
    print(characteristics[cycle_num])

    # continue?
    response = input('\nWould you like to enter another year? (y/n): ')

    while response != 'y' and response != 'n':
        response = input("Please enter 'y' or 'n': ")

    if response == 'n':
        terminate = True
```

FIGURE 4-8 Chinese Zodiac Program

( **line 22**), and not a list type, since the information is not meant to be altered. It

associates each set of characteristics with the corresponding year of the twelve-year cycle of the zodiac based on their position in the tuple. (We could have defined characteristics to contain each of the twelve string descriptions, without the use of variables rat, ox, and so on. It was written this way for the sake of readability.) Variable terminate, initialized to False, is a Boolean flag used to quit the program once set to True (in response to the user being asked to continue with another month or not at **line 50**). **Lines 27–28** display the program greeting.

**Lines 33–56** comprise the main loop of the program. The while loop at **line 38** ensures that the entered year is valid. On **line 43**, the cycle_num for the individual is assigned a value between 0–11, based on their year of birth. Since the year 1900 was the year of the rat in the Chinese Zodiac, the value of cycle_num is (birth_year — 1900) % 12. **Lines 45–47** then use the cycle_num as an index into tuple zodiac_animals (to get the animal for that birth year) and tuple characteristics (to get the associated personal characteristics) to display the results.

Self-Test Questions
**1.** Which of the following sequence types is a mutable type?
**(a)** strings **(b)** lists **(c)** tuples

**2.** Which of the following is true?
**(a)** Lists and tuples are denoted by the use of square brackets.
**(b)** Lists are denoted by use of square brackets and tuples are denoted by the use of

parentheses.
**(c)** Lists are denoted by use of parentheses and tuples are denoted by the use of square
brackets.

**3.** Lists and tuples must each contain at least one element. (TRUE/FALSE)

**4.** For lst 5 [4, 2, 9, 1], what is the result of the following operation, lst.insert(2, 3)?
**(a)**[4, 2, 3, 9, 1] **(b)**[4, 3 ,2, 9, 1] **(c)** [4, 2, 9, 2, 1]

**5.** Which of the following is the correct way to denote a tuple of one element?
**(a)** [6] **(b)** (6) **(c)** [6,] **(d)** (6,)

**6.** Which of the following set of operations can be applied to any sequence?
**(a)**len(s), s[i], s 1w (concatenation)
**(b)**max(s), s[i], sum(s)
**(c)**len(s), s[i], s.sort()

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

## 4.3 Iterating Over Lists (Sequences) in Python

Python's for statement provides a convenient means of iterating over lists (and other sequences). In this section, we look at both for loops and while loops for list iteration.

### 4.3.1 For Loops

A **for statement** is an iterative control statement that iterates once for each element in a specified sequence of elements. Thus, for loops are used to construct definite loops. For example, a for loop is given in Figure 4-9 that prints out the values of a specific list of integers.

| for statement | Example use |
|---|---|
| `for k in sequence:`<br>`  suite` | `nums = [10, 20, 30, 40, 50, 60]`<br><br>`for k in nums:`<br>`  print(k)` |

FIGURE 4-9 The for Statement in Python

V ariable k is referred to as a **loop variable**. Since there are six elements in the provided list, the for loop iterates exactly six times. To contrast the use of for loops and while loops for list iteration, the same iteration is provided as a while loop below,

k 5 0
while k , len(nums):
print(nums[k])
k 5 k 1 1

In the while loop version, loop variable k must be initialized to 0 and incremented by 1 each time through the loop. In the for loop version, loop variable k *automatically* iterates over the provided sequence of values.

The for statement can be applied to all sequence types, including strings. Thus, iteration over a string can be done as follows (which prints each letter on a separate line).
for ch in 'Hello':
print(ch)
Next we look at the use of the built-in range function with for loops.
LET'S TRY IT From the Python Shell, enter the following and observe the results.

... for k in [4, 2 ,3, 1]: ... for k in ['Apple', 'Banana', 'Pear']: print(k) print(k)
??? ???

... for k in (4, 2, 3, 1): ... for k in 'Apple':
print(k) print(k)
??? ???

A **for statement** is an iterative control statement that iterates once for each element in a specified sequence of elements.
4.3.2 The Built-in **range** Function
Python provides a built-in **range function** that can be used for generating a sequence of integers that a for loop can iterate over, as shown below.

sum 5 0
for k in range(1, 11):
sum 5 sum 1 k

The values in the generated sequence include the starting value, up to *but not including* the ending value. For example, range(1, 11) generates the sequence [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Thus, this for loop adds up the integer values 1–10.

The range function is convenient when long sequences of integers are needed. Actually, range does not create a sequence of integers. It creates a *generator function* able to produce each next item of the sequence when needed. This saves memory, especially for long lists. Therefore, typing range(0, 9) in the Python shell does not produce a list as expected—it simply "echoes out" the call to range.

By default, the range function generates a sequence of consecutive integers. A "step" value can be provided, however. For example, range(0, 11, 2) produces the sequence [0, 2, 4, 6, 8, 10], with a step value of 2. A sequence can also be

generated “backwards” when given a negative step value. For example, range(10, 0, 2 1) produces the sequence [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]. Note that since the generated sequence always begins with the provided starting value, “up to” but not including the final value, the final value here is 0, and not 1.

LET’S TRY IT From the Python Shell, enter the following and observe the results.

... for k in range(0, 11): ... for k in range(2, 102, 2): print(k) print(k)
??? ???

... for k in range[0, 11]: ... for k in range(10, 21, 22): print(k) print(k)
??? ???

Python provides a built-in **range** function that can be used for generating a sequence of integers that a for loop can iterate over.
4.3.3 Iterating Over List Elements vs. List Index Values

When the elements of a list need to be accessed, but not altered, a loop variable that iterates over each list element is an appropriate approach. However, there are times when the loop variable must iterate over the *index values* of a list instead. A comparison of the two approaches is shown in Figure 4-10.

| Loop variable iterating over the elements of a sequence | Loop variable iterating over the index values of a sequence |
|---|---|
| `nums = [10, 20, 30, 40, 50, 60]`<br><br>`for k in nums:`<br>`  sum = sum + k` | `nums = [10, 20, 30, 40, 50, 60]`<br><br>`for k in range(len(nums)):`<br>`  sum = sum + nums[k]` |

FIGURE 4-10 Iterating Over the Elements vs. the Index Values of a Given Sequence
Suppose the average of a list of class grades named grades needs to be computed. In this case, a for loop can be constructed to iterate over the grades,
for k in grades:
sum 5 sum 1 k
print('Class average is', sum/len(grades))

However, suppose that the instructor made a mistake in grading, and a point

needed to be added to each student's grade? In order to accomplish this, the index value (the location) of each element must be used to update each grade value. Thus, the loop variable of the for loop must iterate over the index values of the list,

for k in range(len(grades)): grades[k] 5 grades[k] 1 1

In such cases, the loop variable k is also functioning as an *index variable*. An **index variable** is a variable whose *changing value is used to access elements of an indexed data structure*. Note that the range function may be given only one argument. In that case, the starting value of the range defaults to 0. Thus, range(len(grades)) is equivalent to range(0,len(grades)).

LET'S TRY IT From the Python Shell, enter the following and observe the results. ... nums 5 [10, 20, 30]

... for k in range(len(nums)): print(nums[k])
???
... for k in range(len(nums)21, 21, 21): print(nums[k])
???

An **index variable** is a variable whose changing value is used to access elements of an indexed data structure.
4.3.4 While Loops and Lists (Sequences)

There are situations in which a sequence is to be traversed while a given condition is true. In such cases, a while loop is the appropriate control structure. (Another approach for the partial traversal of a sequence is by use of a for loop containing break statements. We avoid the use of break statements in this text, favoring the more structured while loop approach.)

Let's say that we need to determine whether the value 40 occurs in list nums (equal to [10, 20, 30]). In this case, once the value is found, the traversal of the list is terminated. An example of this is given in Figure 4-11.

V ariable k is initialized to 0,and used as an index variable. Thus, the first time through the loop, k is 0, and nums[0] (with the value 10) is compared to item_to_find. Since they are not equal, the second clause of the if statement is executed, incrementing k to 1. The loop continues until either the item is found, or the complete list has been traversed. The final if statement determines

```
k = 0
item_to_find = 40
found_item = False

while k < len(nums) and not found_item:
    if nums[k] == item_to_find:
        found_item = True
    else:
        k = k + 1

if found_item:
    print('item found')
else:
    print('item not found')
```

FIGURE 4-11 List Search in Python

which of the two possibilities for ending the loop occurred, displaying either 'item found' or 'item not found'. Finally, note that the correct loop condition is k , len(nums), and not k ,5 len(nums). Otherwise, an “index out of range” error would result.

LET’S TRY IT
Enter and execute the following Python code and observe the results.

k 5 0
sum 5 0
nums 5 range(100)

while k , len(nums) and sum , 100:
sum 5 sum 1 nums[k]
k 5 k 1 1

print('The first', k, 'integers sum to 100 or greater')
For situations in which a sequence is to be traversed while a given condition is true, a while loop is the appropriate control structure to use.
4.3.5 Let’s Apply It—Password Encryption/Decryption Program

The following program (Figure 4-13) allows a user to encrypt and decrypt passwords containing uppercase/lowercase characters, digits, and special characters. This program utilizes the following programming features:

➤ for loop ➤ nested sequences (tuples)

Example program execution is given in Figure 4-12.

**Lines 4–9** perform the initialization needed for the program. Variable password_out is used to hold the encrypted or decrypted output of the program. Since the output string is created by appending to it each translated character one at a time, it is initialized to the empty string.

Program Execution …

```
This program will encrypt and decrypt user passwords

Enter (e) to encrypt a password, and (d) to decrypt: e
Enter password: Pizza2Day!
Your encrypted password is: Njaam2Fmc!
>>>
```

Program Execution …

```
This program will encrypt and decrypt user passwords

Enter (e) to encrypt a password, and (d) to decrypt: d
Enter password: Njaam2Fmc!
Your decrypted password is: Pizza2Day!
```

FIGURE 4-12 Execution of Password Encryption/Decryption Program

Variable encryption_key holds the tuple (of tuples) used to encrypt/decrypt passwords. This tuple contains as elements tuples of length two, encryption_key 5 (('a', 'm'), ('b', 'h'), etc.

The first tuple, ('a', 'm'), for example, is used to encode the letter 'a'. Thus, when encrypting a given file, each occurrence of 'a' is replaced by the letter 'm'. When decrypting, the reverse is done—all occurrences of letter 'm' are replaced by the letter 'a'.

**Line 12** contains the program greeting. **Line 15** inputs from the user whether they wish to encrypt or decrypt a password. Based on the response, variable encrypting is set to either True or False ( **line 20**).

The program section in **lines 26–47** performs the encryption and decryption. If variable encrypting is equal to True, then from_index is set to 0 and to_index is set to 1, causing the “direction” of the substitution of letters to go from the first in the pair to the second ('a' replaced by 'm'). When encrypting is False (and thus

decryption should be performed), the direction of the substitution is from the second of the pair to the first ('m' replaced by 'a').

V ariable case_changer ( **line 33**) is set to the difference between the encoding of the lowercase and the uppercase letters (recall that the encoding of the lowercase letters is greater than that of the uppercase letters). The for loop at **line 38** performs the iteration over the pairs of letters in the encryption key. The first time through the loop, t 5('a', 'm'). Thus, t[from_index] and t[to_index] refer to each of the characters in the pair. Since all characters in the encryption key are in lowercase, when uppercase letters are found in the password, they are converted to lowercase by use of variable case_changer ( **line 43**) before being compared to the (lowercase) letters in the encryption key. This works because the character encoding of all lowercase letters is greater than the corresponding uppercase version,

... ord('A') ... ord('a') ... ord('a') 2 ord('A') 65 97 32
A similar approach is used for converting from lowercase back to uppercase. Finally, on **lines 50–53**, the encrypted and decrypted versions of the password are displayed to the user.

```
1  # Password Encryption/Decryption Program
2
3  # init
4  password_out = ''
5  case_changer = ord('a') - ord('A')
6  encryption_key = (('a','m'), ('b','h'), ('c','t'), ('d','f'), ('e','g'),
7    ('f','k'), ('g','b'), ('h','p'), ('i','j'), ('j','w'), ('k','e'),('l','r'),
8    ('m','q'), ('n','s'), ('o','l'), ('p','n'), ('q','i'), ('r','u'), ('s','o'),
9    ('t','x'), ('u','z'), ('v','y'), ('w','v'), ('x','d'), ('y','c'), ('z','a'))
10
11 # program greeting
12 print('This program will encrypt and decrypt user passwords\n')
13
14 # get selection (encrypt/decrypt)
15 which = input('Enter (e) to encrypt a password, and (d) to decrypt: ')
16
17 while which != 'e' and which != 'd':
18     which = input("\nINVALID - Enter 'e' to encrypt, 'd' to decrypt: ")
19
20 encrypting = (which == 'e')  # assigns True or False
21
22 # get password
23 password_in = input('Enter password: ')
24
25 # perform encryption / decryption
26 if encrypting:
27     from_index = 0
28     to_index = 1
29 else:
30     from_index = 1
31     to_index = 0
32
33 case_changer = ord('a') - ord('A')
34
35 for ch in password_in:
36     letter_found = False
37
38     for t in encryption_key:
39         if ('a' <= ch and ch <= 'z') and ch == t[from_index]:
40             password_out = password_out + t[to_index]
41             letter_found = True
42         elif ('A' <= ch and ch <= 'Z') and chr(ord(ch) + 32) == t[from_index]:
43             password_out = password_out + chr(ord(t[to_index]) - case_changer)
44             letter_found = True
45
46     if not letter_found:
47          password_out = password_out + ch
48
49 # output
50 if encrypting:
51     print('Your encrypted password is:', password_out)
52 else:
53     print('Your decrypted password is:', password_out)
```

FIGURE 4-13 Password Encryption/Decryption Program

The substitution occurs in the nested for loops in **lines 35–47**. The outer for loop iterates variable ch over each character in the entered password (to be encrypted or decrypted). The first step of the outer for loop is to initialize letter_found to

False. This variable is used to indicate if each character is a (uppercase or lowercase) letter. If so, it is replaced by its corresponding encoding character. If not, it must be a digit or special character, and thus appended as is ( **line 47**). The code on **lines 39–41** and **lines 42–46** is similar to each other. The only difference is that since the letters in the encryption key are all lowercase, any uppercase letters in the password need to be converted to lowercase before being compared to the letters in the key.

Self-Test Questions **1.** For nums 5 [10,30,20,40], what does the following for loop output? for k in nums:
print(k)

**(a)** 10 **(b)**10 **(c)**10
20 30 30
30 20 20
40 40

**2.** For nums 5 [10, 30, 20, 40], what does the following for loop output? for k in range(1, 4):
print(nums[k])

**(a)** 10 **(b)**30 **(c)**10
30 20 30
20 40 20
40

**3.** For fruit 5 'strawberry', what does the following for loop output? for k in range(0, len(fruit), 2):
print(fruit[k], end='')
**(a)**srwer **(b)**tabry
**4.** For nums 5 [12, 4, 11, 23, 18, 41, 27], what is the value of k when the while loop terminates?
k 5 0
while k , len(nums) and nums[k] ! 5 18:
k 5 k 1 1
**(a)**3 **(b)**4 **(c)**5
ANSWERS: 1. (b), 2. (b), 3. (a), 4. (b)
4.4 More on Python Lists

In this section, we take a closer look at the assignment of lists. We also introduce

a useful and convenient means of generating lists that the range function cannot produce, called *list comprehensions*.

### 4.4.1 Assigning and Copying Lists

Because of the way that lists are represented in Python, when a variable is assigned to another variable holding a list, list2 5list1, each variable ends up referring to the *same instance* of the list in memory. This is depicted in Figure 4-14.



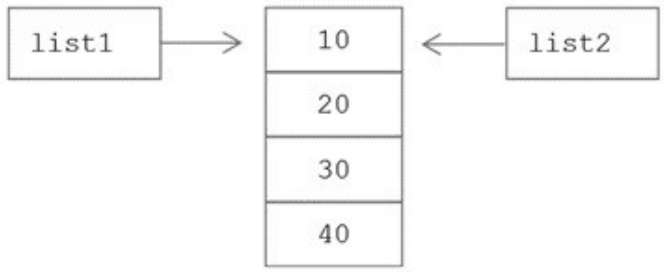



FIGURE 4-14 Assignment of Lists

This has important implications. For example, if an element of list1 is changed, then the corresponding element of list2 will change as well,

... list1 5 [10, 20, 30, 40]
... list2 5 list1
... list1[0] 5 5
... list1
[5, 20, 30, 40] change made in list1
... list2
[5, 20, 30, 40] change in list1 causes a change in list2

Knowing that variables list1 and list2 refer to the same list explains this behavior. This issue does not apply to strings and tuples, since they are immutable and therefore cannot be modified. When needed, a copy of a list can be made as given below,
list2 5 list(list1)
In this case, we get the following results,

... list1 5 [10, 20, 30, 40]
... list2 5 **list(list1)**
... list1[0] 5 5
... list1

[5, 20, 30, 40] change made in list1
... list2
[10, 20, 30, 40] change in list1 does NOT cause any change in list2

When copying lists that have sublists, another means of copying, called *deep copy*, may be needed. We will discuss this further in Chapter 6 when discussing objects in Python.

LET'S TRY IT
From the Python Shell, enter the following and observe the results.
... list1 5 ['red', 'blue', 'green'] ... list1 5 ['red', 'blue', 'green']

... list2 5 list1 ... list2 5 list(list1) ... list1[2] 5 'yellow' ... list1[2] 5 'yellow' ... list1
... list1
??? ???
... list2 ... list2
??? ???
When a variable is assigned to another variable holding a list, each variable ends up referring to the *same instance* of the list in memory.
4.4.2 List Comprehensions

The range function allows for the generation of sequences of integers in fixed increments. **List comprehensions** in Python can be used to generate more varied sequences. Example list comprehensions are given in Figure 4-15.

| Example List Comprehensions | Resulting List |
|---|---|
| (a) `[x**2 for x in [1, 2, 3]]` | `[1, 4, 9]` |
| (b) `[x**2 for x in range(5)]` | `[0, 1, 4, 9, 16]` |
| (c) `nums = [-1, 1, -2, 2, -3, 3, -4, 4]`<br>`[x for x in nums if x >= 0]` | `[1, 2, 3, 4]` |
| (d) `[ord(ch) for ch in 'Hello']` | `[72, 101, 108, 108, 111]` |
| (e) `vowels = ('a', 'e', 'i', 'o', 'u')`<br>`w = 'Hello'`<br>`[ch for ch in w if ch in vowels]` | `['e', 'o']` |

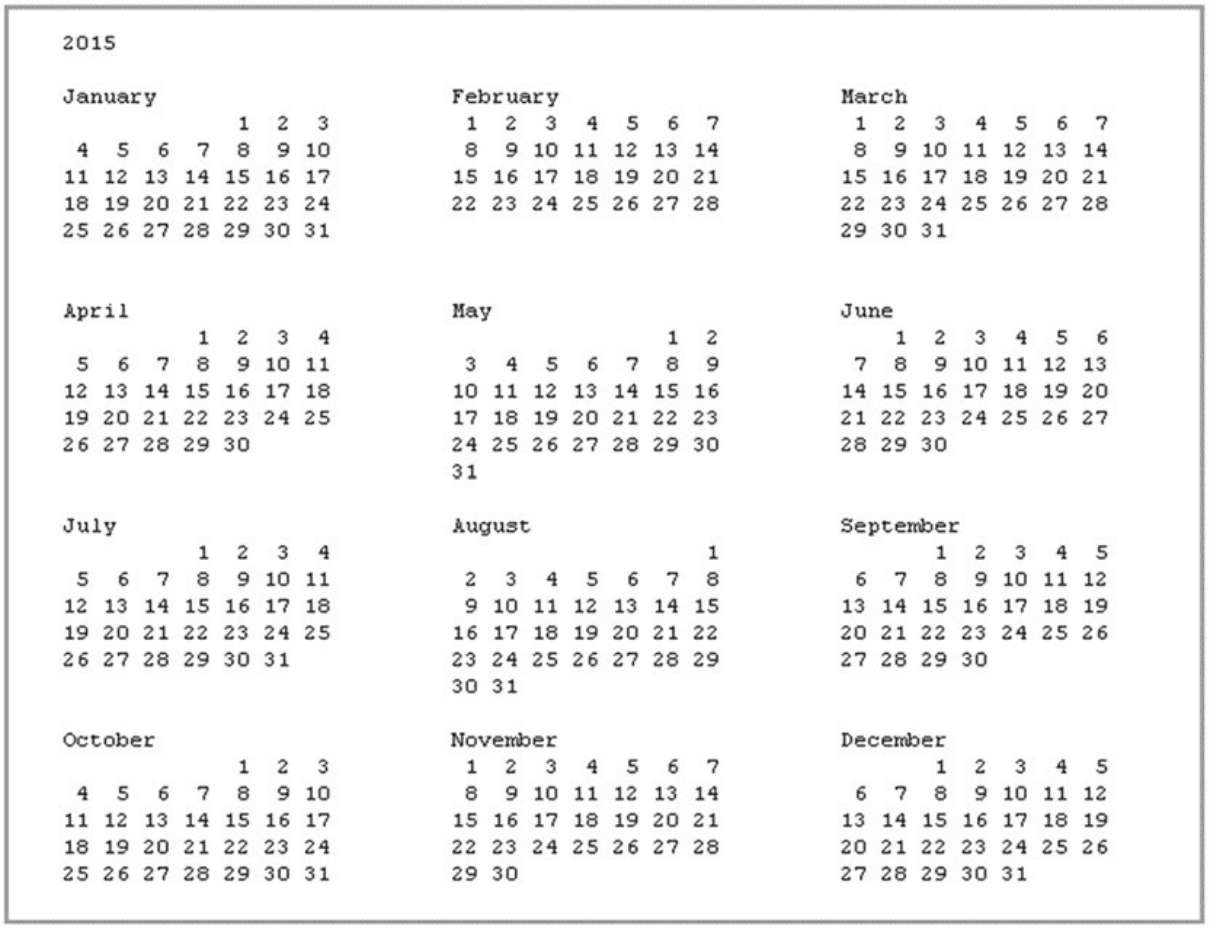
FIGURE 4-15 List Comprehensions

In the figure, (a) generates a list of squares of the integers in list [1, 2, 3]. In (b), squares are generated for each value in range(5). In (c), only positive elements of list nums are included in the resulting list. In (d), a list containing the character encoding values in the string 'Hello' is created. Finally, in (e), tuple vowels is used for generating a list containing only the vowels in string w. List comprehensions are a very powerful feature of Python.

LET'S TRY IT
From the Python Shell, enter the following and observe the results.

... temperatures 5 [88, 94, 97, 89, 101, 98, 102, 95, 100]
... [t for t in temperatures if t .5 100]
???

... [(t 2 32) * 5/9 for t in temperatures]
???
**List comprehensions** in Python provide a concise means of generating a more varied set of sequences than those that can be generated by the range function.
COMPUTATIONAL PROBLEM SOLVING
4.5 Calendar Year Program
In this section, we extend the calendar month program given in the Computational Problem Solving section of Chapter 3 to display a complete calendar year.
4.5.1 The Problem
The problem is to display a calendar year for any year between 1800 and 2099, inclusive. The format of the displayed year should be as depicted in Figure 4-16.



```
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 4-16 Calendar Month Display

### 4.5.2 Problem Analysis

The computational issues for this problem are similar to the calendar month program of Chapter 3. We need an algorithm for computing the first day of a given month for years 1800–2099. However, since the complete year is being displayed, only the day of the week for January 1st of the given year needs be computed—the rest of the days follow from knowing the number of days in each month (including February for leap years). The algorithm previously developed to display a calendar month, however, is not relevant for this program. Instead, the information will first be stored in a data structure allowing for the months to be displayed three across.

### 4.5.3 Program Design

#### Meeting the Program Requirements

We will develop and implement an algorithm that displays the calendar year as

shown in Figure 4-16. We shall request the user to enter the four-digit year to display, with appropriate input error checking.

Data Description

The program needs to represent the year entered, whether it is a leap year, the day of the week for January 1st of the year, and the number of days in each month (accounting for leap years). The names of each of the twelve months will also be stored for display in the calendar year. Given this information, the calendar year can be appropriately constructed and displayed.

We make use of nested lists for representing the calendar year. The data structure will start out as an empty list and will be built incrementally as each new calendar month is computed. The list structures for the calendar year and calendar month are given below,

calendar_year 5 [ [ *calendar_month*], [ *calendar_month*], etc.] ]
calendar_month 5 [ *week_1, week_2, . . ., week_k* ]
Each italicized month is represented as a list of four to six strings, with each string storing a week of the month to be displayed (or a blank line for alignment purposes).
FEBRUARY 2015 MAY 2015
Sun Mon Tues Wed Thur Fri Sat Sun Mon Tues Wed Thur Fri Sat 1 2 3 4 5 6 7 1 2

8 9 10 11 12 13 14 4 3 4 5 6 7 8 9 615 16 17 18 19 20 21 lines 10 11 12 13 14 15 16 22 23 24 25 26 27 28 17 18 19 20 21 22 23 lines 24 25 26 27 28 29 30 31

The strings are formatted to contain all the spaces needed for proper alignment when displayed. For example, since the first week of May 2015 begins on a Friday, the string value for this week would be,

' 1 2'
The complete representation for the calendar year 2015 is given below, with the details shown for the months of February and May.
[[ *January* ],
[ ' 1 2 3 4 5 6 7', ' 8 9 10 11 12 13 14', February ' 15 16 17 18 19 20 21', ' 22 23 24 25 26 27 28'],
*[ March ],*

*[ April ],*

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

*[ June ],*
*[ July ],*
*[ August ],*
*[ September ],*
*[ October ],*
*[ November ],*
*[ December ] ]*

(Typically, yearly calendars combine the one or two remaining days of the month on the sixth line of a calendar month onto the previous week. We shall not do that in this program, however.) Algorithmic Approach

We make use of the algorithm for determining the day of the week previously used. For this program, however, the only date for which the day of the week needs to be determined is January 1 of a given year. Thus, the original day of the week algorithm can be simplified by removing variable day and replacing its occurrence on line 6 with 1, given in Figure 4-17.

To determine the day of the week for January 1 of a given **year**:

1. Let **century_digits** be equal to the first two digits of the year.
2. Let **year_digits** be equal to the last two digits of the year.
3. Let **value** be equal to **year_digits** + floor(**year_digits** / 4)
4. If **century_digits** equals 18, then add 2 to **value**, else
   if **century_digits** equals 20, then add 6 to **value**.
5. If **year** is not a leap year then add 1 to **value**.
6. Set **value** equal to (**value** + 1) mod 7.
7. If **value** is equal to 1 (Sunday), 2 (Monday), … 0 ( Saturday).

FIGURE 4-17 Simplified Day of the Week Algorithm
Overall Program Steps
The overall steps in this program design are given in Figure 4-18.

4.5.4 Program Implementation and Testing
Stage 1—Determining the Day of the Week (for January 1st)

Wefirst develop and test the code for determining the day of the week for January 1st of a given year. This modified code from the calendar month program is given in Figure 4-19.

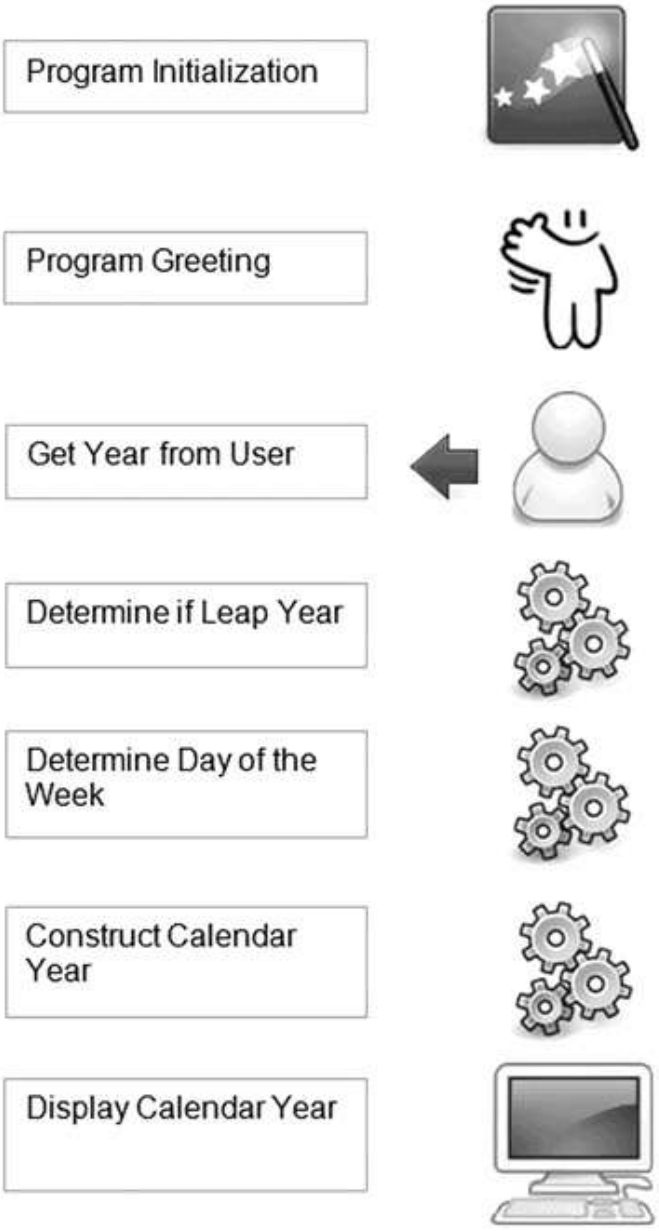



FIGURE 4-18 Overall Steps of Calendar Year Program

**Line 4** initializes Boolean flag terminate to False. If the user enters −1 for the

year (in **lines 10–13**), terminate is set to True and the while loop at **line 7** terminates, thus terminating the program. If a valid year is entered, **lines 19–42** are executed.

**Lines 19–22** determine if the year is a leap year using the same code as in the calendar month program, assigning Boolean variable leap_year accordingly. **Lines 25–40** implement the simplified day of the week algorithm for determining the day of the week for January 1 of a given year in Figure 4.17, with the result displayed on **line 42**.

Stage 1—Testing

We show a sample test run of this stage of the program in Figure 4-20.
Figure 4-21 displays the test cases used for this program.
Since all test cases passed, we can move on to the next stage of program development.

Stage 2—Constructing the Calendar Year Data Structure

Next we develop the part of the program that constructs the data structure holding all of the calendar year information to be displayed. The data structure begins empty and is incrementally built, consisting of nested lists, as previously discussed.

```
# Calendar Year Program (Stage 1)

# initialization
terminate = False

# prompt for years until quit
while not terminate:

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

    while (year < 1800 or year > 2099) and year != -1:
        year = int(input('INVALID - Enter year(1800-2099): '))

    if year == -1:
        terminate = True
    else:
        # determine if leap year
        if (year % 4 == 0) and (not (year % 100 == 0) or (year % 400 == 0)):
            leap_year = True
        else:
            leap_year = False

        # determine day of the week
        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

        # leap year check
        if not leap_year:
            value = value + 1

        # determine first day of month for Jan 1
        first_day_of_month = (value + 1) % 7

        print('Day of week is:', first_day_of_month)
```

FIGURE 4-19 Stage 1 of the Calendar Year Program

```
Enter year (yyyy) (-1 to quit): 1800
Day of week is: 4
Enter year (yyyy) (-1 to quit): 1900
Day of week is: 2
Enter year (yyyy) (-1 to quit): 1984
Day of week is: 1
Enter year (yyyy) (-1 to quit): 1985
Day of week is: 3
Enter year (yyyy) (-1 to quit): 3000
INVALID - Enter year(1800-2099): 2000
Day of week is: 0
Enter year (yyyy) (-1 to quit): 2009
Day of week is: 5
Enter year (yyyy) (-1 to quit): -1
>>>
```

FIGURE 4-20 Example Output of First Stage Testing

| Calendar Month | Expected Results first day of month | Actual Results first day of month | Evaluation |
|---|---|---|---|
| January 1800 | 4 (Wednesday) | 4 | Passed |
| January 1900 | 2 (Monday) | 2 | Passed |
| January 1984 | 1 (Sunday) | 1 | Passed |
| January 1985 | 3 (Tuesday) | 3 | Passed |
| January 2000 | 0 (Saturday) | 0 | Passed |
| January 2009 | 5 (Thursday) | 5 | Passed |

FIGURE 4-21 Test Cases for Stage 1 of the Calendar Year Program

Figure 4-22 shows an implementation of this stage of the program.

**Lines 4–14** perform the required initialization. Tuples days_in_month and month_ names have been added to the program to store the number of days for each month (with February handled as an exception) and the month names. On **line 11**, calendar_year is initialized to the empty list. It will be constructed month-by-month for the twelve months of the year. There is the need for strings of blanks of various lengths in the program, initialized as month_ separator, blank_week, and blank_col ( **lines 12–14**). The calendar_ year data structure will contain all the space characters needed for the calendar months to be properly displayed. Therefore, there will be no need to develop code that determines how

each month should be displayed as in the calendar month program. The complete structure will simply be displayed row by row.

**Lines 17–49** are the same as the first stage of the program for determining the day of the week of a given date. Once the day of the week for January 1st of the given year is known, the days of the week for all remaining dates simply follow. Thus, there is no need to calculate the day of the week for any other date.

**Line 52** begins the for loop for constructing each of the twelve months. On **line 53**, the month name is retrieved from tuple month_names and assigned to month_name. Variable current_ day, holding the current day of the month, is initialized to 1 for the new month ( **line 56**). In **lines 57–60**, first_day_of_current_month, determined by the day of the week algorithm, is converted to the appropriate column number. Thus, since 0 denotes Saturday, if first_day_of_current_month equals 0, starting_col is set to 7. Otherwise, starting_col is set to first_day_of_current_month (e.g., if first_day_of_ current_month is 1, then starting_col is set to 1).

In **lines 62–64**, the initialization for a new month finishes with the reassignment of current_col, calendar_week, and calendar_month. Each calendar week of a given month is initially assigned to the empty string, with each date appended one-by-one. Variable current_col is used to keep track of the current column (day) of the week, incremented from 0 to 6. Since the first day of the month can fall on any day of the week, the first week of any month may contain blank (“skipped”) columns. This includes the columns from current_col up to but not including starting_col. The while loop in **lines 67–69** appends any of these skipped columns to empty string calendar_week.

Lines **72–75** assign num_days_this_month to the number of days stored in tuple days_ in_month. The exception for February, based on whether the year is a leap year or not, is handled as a special case. The while loop at **line 77** increments variable current_day from 1 to the number of

```
# Calendar Year Program (Final Version)

# initialization
terminate = False
days_in_month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

month_names = ('January', 'February', 'March', 'April', 'May', 'June'
               'July', 'August', 'September', 'October', 'November',
               'December')

calendar_year = []
month_separator = format(' ', '8')
blank_week = format(' ', '21')
blank_col = format(' ', '3')

# prompt for years until quit
while not terminate:

    # get year
    year = int(input('Enter year (yyyy) (-1 to quit): '))
    while (year < 1800 or year > 2099) and year != -1:
        year = int(input('INVALID - Enter year(1800-2099): '))

    if year == -1:
        terminate = True
    else:
        # determine if leap year
        if (year % 4 == 0) and (not (year % 100 == 0) or
            (year % 400 == 0)):
            leap_year = True
        else:
            leap_year = False

        # determine day of the week
        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

        # leap year check
        if not leap_year:
            value = value + 1

        # determine first day of month for Jan 1
        first_day_of_current_month = (value + 1) % 7

        # construct calendar for all 12 months
        for month_num in range(12):
            month_name = month_names[month_num]

            # init for new month
            current_day = 1
            if first_day_of_current_month == 0:
                starting_col = 7
            else:
                starting_col = first_day_of_current_month

            current_col = 1
            calendar_week = ''
            calendar_month = []

```

FIGURE 4-22 Stage 2 of the Calendar Year Program (*Continued*)

```
66              # add any needed leading space for first week of month
67              while current_col < starting_col:
68                  calendar_week = calendar_week + blank_col
69                  current_col = current_col + 1
70
71              # store month as separate weeks
72              if (month_name == 'February') and leap_year:
73                  num_days_this_month = 29
74              else:
75                  num_days_this_month = days_in_month[month_num]
76
77              while current_day <= num_days_this_month:
78
79                  # store day of month in field of length 3
80                  calendar_week = calendar_week + \
81                     format(str(current_day),'>3')
82
83                  # check if at last column of displayed week
84                  if current_col == 7:
85                      calendar_month = calendar_month + [calendar_week]
86                      calendar_week = ''
87                      current_col = 1
88                  else:
89                      current_col = current_col + 1
90
91                  # increment current day
92                  current_day = current_day + 1
93
94              # fill out final row of month with needed blanks
95              calendar_week = calendar_week + \
96               blank_week[0:(7-current_col+1) * 3]
97              calendar_month = calendar_month + [calendar_week]
98
99              # reset values for next month
100             first_day_of_current_month = current_col
101             calendar_year = calendar_year + [calendar_month]
102             calendar_month = []
103
104     print(calendar_year)
105
106     #reset for another year
107     calendar_year = []
```

FIGURE 4-22 Stage 2 of the Calendar Year Program

days in the month. In **lines 80–81** each date is appended to calendar_week right-justified as a string of length three by use of the format function. Thus, a single-digit date will be appended with two leading blanks, and a double-digit date with one leading blank so that the columns of dates align.

For each new date appended to calendar_week, a check is made on **line 84** as to whether the end of the week has been reached. If the last column of the calendar week has been reached (when column_col equals 7) then the constructed calendar_week string is appended to the calendar_month ( **line 85**). In addition,

calendar_week is re-initialized to the empty string, and current_col is reset to 1 ( **lines 86–87**). If the last column of the calendar week has not yet been reached, then current_col is simply incremented by 1 ( **line 89**). Then, on **line 92**, variable current_day is incremented by 1, whether or not a new week is started.

When the while loop (at **line 77**) eventually terminates, variable current_week holds the last week of the constructed month. Therefore, as with the first week of the month, the last week may contain empty columns. This is handled by **lines 95–97**. Before appending calendar_week to calendar_month, any remaining unfilled columns are appended to it (the reason that these final columns must be blank-filled is because months are displayed side-by-side, and therefore are needed to keep the whole calendar properly aligned),

calendar_week 5 calendar_week 1 blank_week[0:(7-current_col 11) * 3]

Thus, the substring of blank_week produced will end up as an empty string if the value of current_col is 6 (for Saturday, the last column) as it should. **Line 100** sets variable first_day_ of_current_month to current_col since current_col holds the column value of the next column that *would have been* used for the current month, and thus is the first day of the following month. On **line 101**, the completed current month is appended to list calendar_year. And on **line 102**, calendar_month is reset to an empty list in anticipation of the next month to be constructed.

Finally, on **line 104**, the complete calendar_year list is displayed. Because the program prompts the user for other years to be constructed and displayed, the calendar_year list is reset to the empty list (**line 107**).

Stage 2—Testing
The program terminates with an error on line 53,
Enter year (yyyy) (-1 to quit): 2015
Traceback (most recent call last):
File “C:\My Python Programs\CalendarYearStage2.py”, line 54, in ,module .
month_name 5 month_names[month_num]
IndexError: tuple index out of range
This line is within the for loop at line 52,
for month_num in range(12):
month_name 5 month_names[month_num]
For some reason, index variable month_num is out of range for tuple

month_names. We look at the final value of month_num by typing the variable name into the Python shell, ... month_num
11

Since month_names has index values 0–11 (since of length 12), an index value of 11 should not be out of range. How, then, can this index out of range error happen? Just to make sure that month_ names has the right values, we display its length,

.. len(month_names)
11

This is not right! The tuple month_names should contain all twelve months of the year. That is the way it was initialized on line 7, and tuples, unlike lists, cannot be altered, they are immutable. This does not seem to make sense. To continue our investigation, we display the value of the tuple,

.. month_names
('January', 'February', 'March', 'April', 'May', 'JuneJuly', 'August', 'September', 'October', 'November', 'December')
...

Now we see something that doesn't look right. Months June and July are concatenated into one string value 'JuneJuly' making the length of the tuple 11, and not 12 (as we discovered). *That* would explain why the index out of range error occurred.

What, then, is the problem. Why were the strings 'June' and 'July' concatenated? We need to look at the line of code that creates this tuple,
month_names 5 ('January', 'February', 'March', 'April', 'May', 'June' 'July', 'August', 'September', 'October', 'November', 'December')

It looks OK. Strings 'June' and 'July' were written as separate strings. We then decide to count the number of items in the tuple. Since items in tuples and lists are separated by commas, we count the number of items between the commas. We count the items up to 'May', which is five items as it should be, then 'June', which is six items . . . ah, there is no comma after the string 'June'! *That* must be why strings 'June' and 'July' were concatenated, and thus the source of the index out of range error. We try to reproduce this in the shell,

... 'June' 'July'
'JuneJuly' That’s it! We have found the problem and should feel good about it. After making the correction and re-executing the program, we get the following results,

```
Enter year (yyyy) (-1 to quit): 2015
[[' 1 2 3', ' 4 5 6 7 8 9 10', ' 11 12 13 14 15 16 17', ' 1 8 19 20,21 22 23 24', ' 25 26 27 28 29 30 31',' '], [' 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', ' '], [' 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

'], [' 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 '], [' 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 '], [' 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 '], [' 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 '], ['
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 '], [' 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 '], [' 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', ' '], [' 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 '], [' 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 ']]
Enter year (yyyy) (-1 to quit):
```

We can see if the output looks like the structure that we expect. The first item in the list, the structure for the month of January, is as follows,

```
[[' 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', ' ']
```

In checking against available calendar month calculators, we see that the first day of the month for January 2015 is a Thursday. Thus, the first week of the month should have four skipped days, followed by 1, 2, and 3 each in a column width of 3. We find that there are fourteen blank characters in the first line. The first twelve are for the four skipped columns, and the last two are for the right-justified string ‘1’ in the column of the first day of the month,

' 1 2 3'
\|||/
12 blank 2 blank
chars chars

Since there are five weeks in the month, there should be one extra “blank week” at the end of the list to match the vertical spacing of all other months. We see, in fact, that the last (sixth) string is a string of blanks.

Since the calendar_year structure looks correct, we now develop the final stage of the program that displays the complete calendar year.

Stage 3—Displaying the Calendar Year Data Structure

We now give the complete calendar year program in Figure 4-23. In this final version, the only change at the start of the program is that a program greeting is added on **line 19**. The rest of the program is the same up to **line 105**, the point where the calendar year has been constructed. (The print(calendar_year) line and re-initialization of calendar_year to the empty list have been removed from the previous version, since they were only there for testing purposes.)

The new code in this version of the program is in **lines 107–141**, which displays the calendar year. The calendar year output is given in Figure 4-24.

On **line 108** the year is displayed. Because the months are displayed three across, as shown in Figure 4-16, the for loop on **line 111** iterates variable month_num over the values [0, 3, 6, 9]. Thus, when month_num is 0, months 0-2 (January–March) are displayed. When month_num is 3, months 3-5 (April–June) are displayed, and so forth.

The for loop at **line 114** displays the month names for each row (for example, January, February, and March). Each is displayed left-justified in a field width of 19. A leading blank character is appended to the formatting string to align with the first column of numbers displayed for each month. The print(. . ., end 5") form of print is used, which prevents the cursor from moving to the next line. Thus, the months can be displayed side-by-side. Variable month_ separator contains the appropriate number of blank spaces (initialized at the top of the program) to provide the required amount of padding between the months, as shown below,

**January February March** 1 2 3 1 2 3 4 5 6 7 1 2 3 4 5 6 7
\|||||/ \|||||||/
month_separator month_separator

**Lines 119–120** perform the initialization needed for the following while loop (at **line 122**), which displays each week, one-by-one, of the current three months. Variable week is initialized to zero for each month and is used to keep count of the number of weeks displayed. Variable lines_to_print is initialized to True to start the execution of the following while loop.

At **line 125** within the while loop, lines_to_print is initialized to False. It is then set to True by any (or all) of the current three months being displayed only if they still have more calendar lines (weeks) to print, thus causing the while loop to continue with another

```
# Calendar Year Program (Final Version)

# initialization
terminate = False
days_in_month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

month_names = ('January', 'February', 'March', 'April', 'May', 'June',
               'July', 'August', 'September', 'October', 'November',
               'December')

calendar_year = []
month_separator = format(' ', '8')
blank_week = format(' ', '21')
blank_col = format(' ', '3')

# prompt for years until quit
while not terminate:

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

    # get year
    year = int(input('Enter year (yyyy) (-1 to quit): '))
    while (year < 1800 or year > 2099) and year != -1:
        year = int(input('INVALID - Enter year(1800-2099): '))

    if year == -1:
        terminate = True
    else:
        # determine if leap year
        if (year % 4 == 0) and (not (year % 100 == 0) or
            (year % 400 == 0)):
            leap_year = True
        else:
            leap_year = False

        # determine day of the week
        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

        # leap year check
        if not leap_year:
            value = value + 1

        # determine first day of month for Jan 1
        first_day_of_current_month = (value + 1) % 7

        # construct calendar for all 12 months
        for month_num in range(12):
            month_name = month_names[month_num]

            # init for new month
            current_day = 1
            if first_day_of_current_month == 0:
                starting_col = 7
            else:
                starting_col = first_day_of_current_month

```

```
            current_col = 1
            calendar_week = ''
            calendar_month = []

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

            # store month as separate weeks
            if (month_name == 'February') and leap_year:
                num_days_this_month = 29
            else:
                num_days_this_month = days_in_month[month_num]

            while current_day <= num_days_this_month:

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

                # check if at last column of displayed week
                if current_col == 7:
                    calendar_month = calendar_month + [calendar_week]
                    calendar_week = ''
                    current_col = 1
                else:
                    current_col = current_col + 1

                # increment current day
                current_day = current_day + 1

            # fill out final row of month with needed blanks
            calendar_week = calendar_week + \
                            blank_week[0:(7-current_col+1) * 3]
            calendar_month = calendar_month + [calendar_week]

            # reset values for next month
            first_day_of_current_month = current_col
            calendar_year = calendar_year + [calendar_month]
            calendar_month = []

        # print calendar year
        print('\n', year,'\n')

        # each row starts with January, April, July, or October
        for month_num in [0,3,6,9]:

            # displays three months in each row
            for i in range(month_num, month_num + 3):
                  print(' ' + format(month_names[i],'19'),
                        month_separator, end='')

            # display each week of months on separate lines
            week = 0
            lines_to_print = True

```

```
122             while lines_to_print:
123
124                 # init
125                 lines_to_print = False
126
127                 # another week to display for first month in row?
128                 for k in range(month_num, month_num + 3):
129                     if week < len(calendar_year[k]):
130                         print(calendar_year[k][week], end='')
131                         lines_to_print = True
132                     else:
133                         print(blank_week, end='')
134
135                     print(month_separator, end='')
136
137                 # move to next screen line
138                 print()
139
140                 # increment week
141                 week = week + 1
```

FIGURE 4-23 Final Stage of the Calendar Year Program

iteration. This occurs within the for loop at **lines 128–135**. Since variable month_num indicates the current month being displayed, the number of weeks in the month is determined by the length of the tuple of strings for the current month k.

len(calendar_year[k])
Note that some months may have no more weeks to display, whereas others may. This is the case for the first three months of 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

In this case, the while loop needs to continue to iterate in order to display the last lines of January and March even though the last line of February has been displayed. Therefore, in cases where a given month has a line to print but another month doesn’t, a blank line is displayed in order to maintain the correct alignment of month weeks. After the week of dates (or blank week) is output for each of the three months, the cursor is moved to the start of the next line (on **line 138**) and variable week is incremented by one ( **line 141**) before the loop begins

the next iteration for displaying the next row of calendar weeks.

Chapter Summary 161

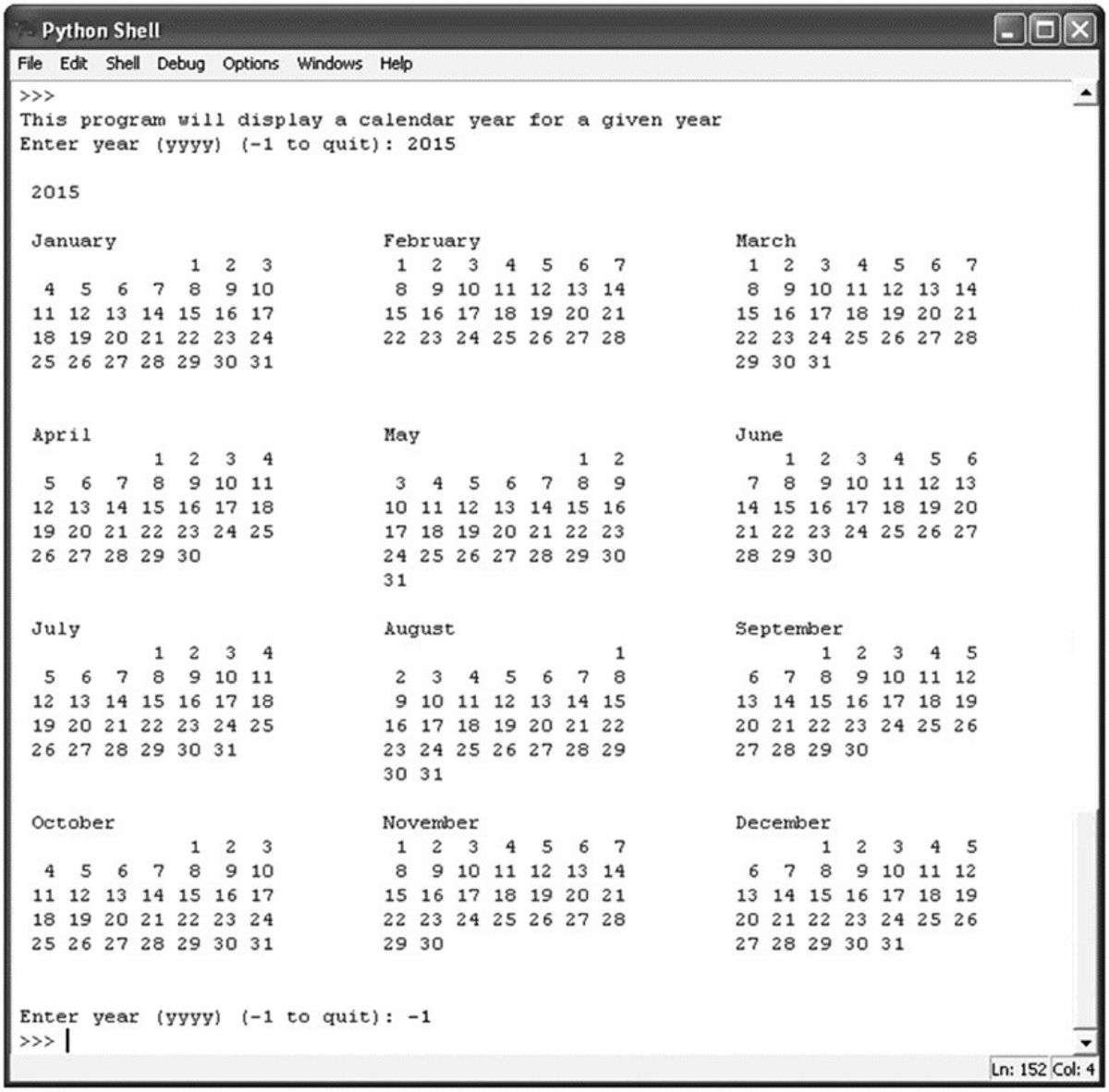



FIGURE 4-24 Calendar Year Program Output

Finally, the while loop at line 122 continues to iterate until there are no more lines to display for all of the three months currently being displayed—that is, until lines_to_print is False. Figure 4-25 displays the results of testing this final version by using the test plan from the Calendar Month program of Chapter 3. The test plan passed for all test cases.

CHAPTER SUMMARY General Topics

Linear Data Structures List Operations
List Traversal
The Empty List and Its Use Nested Lists
List Iteration
Loop Variable/Index Variable

| Calendar Month | Expected Results | | Actual Results | | Evaluation |
|---|---|---|---|---|---|
| | first day | num days | first day | num days | |
| January 1800 | Wednesday | 31 | Wednesday | 31 | Pass |
| February 1800 | Saturday | 28 | Saturday | 28 | Pass |
| April 1860 | Sunday | 30 | Sunday | 30 | Pass |
| July 1887 | Friday | 31 | Friday | 31 | Pass |
| February 1904 | Monday | 29 | Monday | 29 | Pass |
| March 1945 | Thursday | 31 | Thursday | 31 | Pass |
| September 1960 | Thursday | 30 | Thursday | 30 | Pass |
| October 1990 | Monday | 31 | Monday | 31 | Pass |
| November 1992 | Sunday | 30 | Sunday | 30 | Pass |
| February 2000 | Tuesday | 29 | Tuesday | 29 | Pass |
| August 2006 | Tuesday | 31 | Tuesday | 31 | Pass |
| May 2014 | Thursday | 31 | Thursday | 31 | Pass |
| December 2019 | Sunday | 31 | Sunday | 31 | Pass |
| June 2084 | Thursday | 30 | Thursday | 30 | Pass |

FIGURE 4-25 Final Calendar Year Program Testing

Python-Specific Programming Topics

Lists in Python
List Operations in Python
Empty Lists and Tuples in Python
Lists, Tuples, and Strings as Sequences in Python Additional Sequence Operations

Nested Lists and Tuples in Python
For/ While Loops and List Iteration in Python Built-in Range Function in Python
Iterating over List (Sequence) Elements vs.

Iterating over Index Values in Python Assigning Lists in Python
List Comprehensions in Python

CHAPTER EXERCISES Section 4.1
**1. (a)** Give the index values of all the odd numbers in the following list representation, assuming zero-based indexing.

23
16
14
33
19
6
11


**(b)** How many elements would be looked at when the list is traversed (from top to bottom) until the value 19 was found?
Section 4.2
**2.** Which of the following lists are syntactically correct in Python?
**(a)** [1, 2, 3, 'four'] **(b)**[1, 2, [3, 4]] **(c)**[[1, 2, 3]['four']] **3.** For lst 5[4, 2, 9, 1], what is the result of each of the following list operations? **(a)** lst[1] **(b)** lst.insert(2, 3) **(c)** del lst[3] **(d)** lst.append(3)

**4.** For fruit 5['apple', 'banana', 'pear', 'cherry'], use a list operation to change the list to ['apple', 'banana', 'cherry'] .
**5.** For a list of integers, lst, give the code to retrieve the maximum value of the second half of the list.
**6.** For variable product_code containing a string of letters and digits,
**(a)** Give an if statement that outputs “Verified” if product_code contains both a “Z” and a “9”, and outputs “Failed” otherwise.
**(b)** Give a Python instruction that prints out just the last three characters in product_code.
**7.** Which of the following are valid operations on tuples (for tuples t1 and t2)?
**(a)**len(t1) **(b)**t1 1 t2 **(c)**t1.append(10) **(d)**t1.insert(0, 10)

**8.** For str1 5'Hello World', answer the following,
**(a)** Give an instruction that prints the fourth character of the string.
**(b)** Give an instruction that finds the index location of the first occurrence of the letter 'o' in the string.
**9.** For a nested list lst that contains sublists of integers of the form [n1, n2, n3],
**(a)** Give a Python instruction that determines the length of the list.
**(b)** Give Python code that determines how many total integer values there are in list lst. **(c)** Give Python code that totals all the values in list lst.
**(d)** Given an assignment statement that assigns the third integer of the fourth element (sublist) of lst to the value 12.

Section 4.3

**10.** For a list of integers named nums,
**(a)** Write a while loop that adds up all the values in nums.
**(b)** Write a for loop that adds up all the values in nums in which the loop variable is assigned each value

in the list.
**(c)** Write a for loop that adds up all the elements in nums in which the loop variable is assigned to the
index value of each element in the list.
**(d)** Write a for loop that displays the elements in nums backwards.
**(e)** Write a for loop that displays every other element in nums, starting with the first element.

Section 4.4
**11.** For list1 5 [1, 2, 3, 4] and list2 5[5, 6, 7, 8], give the values of list1[0] and list2[0] where indicated after the following assignments.
**(a)** list1[0] 510

list2[0] 550 list1[0] _____ list2[0] _____ **(b)** list2 5list1 list1[0] _____ list2[0] _____ **(c)** list2[0] 515 list1[0] _____ list2[0] _____ **(d)** list1[0] 50 list1[0] _____ list2[0] _____

**12.** Give an appropriate list comprehension for each of the following.
**(a)** Producing a list of consonants that appear in string variable w.
**(b)** Producing a list of numbers between 1 and 100 that are divisible by 3.
**(c)** Producing a list of numbers, zero_values, from a list of floating-point values,

data_values,

that are within some distance, epsilon, from 0.

## PYTHON PROGRAMMING EXERCISES

**P1.** Write a Python program that prompts the user for a list of integers, stores in another list only those values between 1–100, and displays the resulting list.
**P2.** Write a Python program that prompts the user for a list of integers, stores in another list only those values that are in tuple valid_values, and displays the resulting list.
**P3.** Write a Python program that prompts the user for a list of integers and stores them in a list. For all values that are greater than 100, the string 'over' should be stored instead. The program should display the resulting list.
**P4.** Write a Python program that prompts the user to enter a list of first names and stores them in a list. The program should display how many times the letter 'a' appears within the list.
**P5.** Write a Python program that prompts the user to enter a list of words and stores in a list only those words whose first letter occurs again within the word (for example, 'Baboon'). The program should display the resulting list.
**P6.** Write a Python program that prompts the user to enter types of fruit, and how many pounds of fruit there are for each type. The program should then display the information in the form *fruit, weight* listed in alphabetical order, one fruit type per line as shown below,

Apple, 6 lbs. Banana, 11 lbs. etc.

**P7.** Write a Python program that prompts the user to enter integer values for each of two lists. It then should displays whether the lists are of the same length, whether the elements in each list sum to the same value, and whether there are any values that occur in both lists.

## PROGRAM MODIFICATION PROBLEMS

**M1.** Chinese Zodiac Program: Japanese and Vietnamese Variations
Modify the Chinese Zodiac program in the chapter to allow the user to select the Chinese Zodiac, the Japanese Zodiac, or the Vietnamese Zodiac. The Japanese Zodiac is the same as the Chinese Zodiac, except that “Pig” is substituted with “Wild Boar.” The Vietnamese Zodiac is also the same except that the “Ox” is substituted with “Water Buffalo” and “Rabbit” is replaced with “Cat.”

**M2.** Chinese Zodiac Program: Improved Accuracy
The true Chinese Zodiac does not strictly follow the year that a given person was born. It also depends on the month and date as well, which vary over the years. Following are the correct range of dates for each of the Zodiac symbols for the years 1984 to 2007 (which includes two full cycles of the zodiac). Modify

| | |
|---|---|
| 鼠 Rat | Feb 02, **1984** – Feb 19, 1985 |
| 牛 Ox | Feb 20, **1985** – Feb 08, 1986 |
| 虎 Tiger | Feb 09, **1986** – Jan 28, 1987 |
| 兔 Rabbit | Jan 29, **1987** – Feb 16, 1988 |
| 龍 Dragon | Feb 17, **1988** – Feb 05, 1989 |
| 蛇 Snake | Feb 06, **1989** – Jan 26, 1990 |
| 馬 Horse | Jan 27, **1990** – Feb 14, 1991 |
| 羊 Sheep | Feb 15, **1991** – Feb 03, 1992 |
| 猴 Monkey | Feb 04, **1992** – Jan 22, 1993 |
| 鷄 Rooster | Jan 23, **1993** – Feb 09, 1994 |
| 狗 Dog | Feb 10, **1994** – Jan 30 1995 |
| 猪 Pig | Jan 31, **1995** – Feb 18, 1996 |

| | |
|---|---|
| 鼠 Rat | Feb 19, **1996** – Feb 06, 1997 |
| 牛 Ox | Feb 07, **1997** – Jan 27, 1998 |
| 虎 Tiger | Jan 28, **1998** – Feb 15, 1999 |
| 兔 Rabbit | Feb 16, **1999** – Feb 04, 2000 |
| 龍 Dragon | Feb 05, **2000** – Jan 23, 2001 |
| 蛇 Snake | Jan 24, **2001** – Feb 11, 2002 |
| 馬 Horse | Feb 12, **2002** – Jan 31, 2003 |
| 羊 Sheep | Feb 01, **2003** – Jan 21, 2004 |
| 猴 Monkey | Jan 22, **2004** – Feb 08, 2005 |
| 鷄 Rooster | Feb 09, **2005** – Jan 28, 2006 |
| 狗 Dog | Jan 29, **2006** – Feb 17, 2007 |
| 猪 Pig | Feb 18, **2007** – Feb 06, 2008 |

the Chinese Zodiac program in the chapter so that the user is prompted to enter their date of birth, including month and day, and displays the name and characteristics of the corresponding Chinese Zodiac symbol based on the more accurate zodiac provided here.

**M3.** Password Encryption/Decryption Program: Multiple Executions
Modify the Password Encryption/Decryption program in the chapter so that it allows the user to continue to encrypt and decrypt passwords until they quit.

**M4.** Password Encryption/Decryption Program: Secure Password Check
Modify the Password Encryption/Decryption program in the chapter so that the program rejects any entered password for encryption that is not considered "secure" enough. A password is considered secure if it contains at least eight characters, with at least one digit and one special character (!, #, etc).

**M5.** Password Encryption/Decryption Program: Random Key Generation
Modify the Encryption/Decryption program in the chapter so that a new

encryption key is randomly generated each time the program is executed. (See the Python 3 Programmers' Reference for information on the Random module.)

**M6.** Calendar Year Program: Multilingual Version
Modify the Calendar Year program so that the user can select the language with which the calendar months are labeled. Give the user the choice of at least three different languages from which to select. Find the month names for the other languages online.

**M7.** Calendar Year Program: Flexible Calendar Format
The program as is always displays three months per row. Modify the Calendar Year program so that the user can select how many months are displayed per row. Allow the user to select either two, three, or four months per row.

PROGRAM DEVELOPMENT PROBLEMS

**D1.** Morse Code Encryption/Decryption Program
Develop and test a Python program that allows a user to type in a message and have it converted into Morse code, and also enter Morse code and have it converted back to the original message. The encoding of Morse code 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 | ––•• |

Format the original message (containing English words) so that there is one sentence per line. Format the Morse code file (containing dots and dashes) so that there is one letter per line, with a blank line following the last letter of each word, and two blank lines following the end of each sentence (except the last).

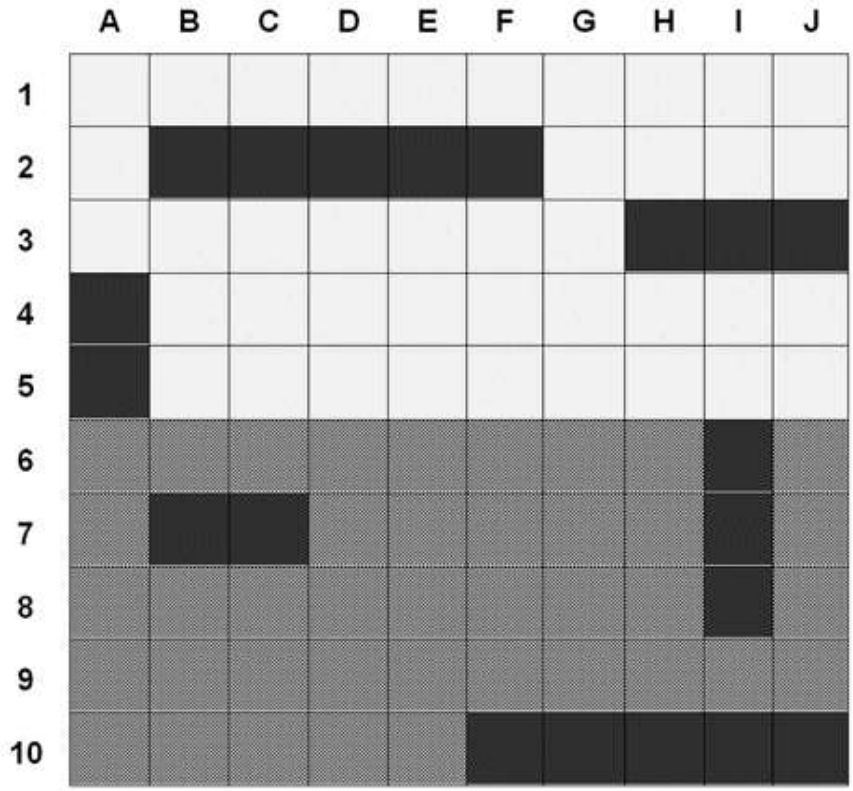
**D2.** Holidays Calendar
Develop and test a Python program that displays the day of the week that the following holidays fall on for a year entered by the user,

♦ New Year's Eve

♦ Valentine’s Day
♦ St. Patrick’s Day
♦ April Fool’s Day
♦ Fourth of July
♦ Labor Day
♦ Halloween
♦ User’s Birthday

Note that Labor Day, as opposed to the other holidays above, does not fall on the same date each year. It occurs each year on the first Monday of September.

**D3.** The Game of Battleship
Battleship is a game involving ships at sea for each of two players. The ships are located in a grid in which each column of the grid is identified by a letter, and each row by a number, as shown below.




The top half of the board contains the ships of player 1, and the bottom half the ships of player 2. The darkened areas indicate the size and location of ships. Each player starts with the same number and types of ships. The location of each

player's ships is determined by the player. Players take turns taking a shot at the opponent's ships by "calling out" a particular grid location. For example, if player 1 calls out "C10," no ship would be hit in this example. If, however, they were to call out "G10," then player 2's ship (on the bottom half of the board) would be hit. Each player calls out "hit" or "miss" when they are shot at by the other player. When all grid locations of a given ship have been hit, the ship is sunk, and the opponent gets the number of points based on the ship's size (given below).

The number of grid locations that a given ship takes up indicates its type and point value. A typical set of ships is given below.

| Type of Ship | Size of Ships |
|---|---|
| aircraft carrier | 5 |
| battleship | 4 |
| cruiser | 3 |
| submarine | 3 |
| destroyer | 2 |

Develop and test a Python program that can play the game of battleship. The user should be able to select the skill level. The higher the skill level, the larger the grid that is created for play. All games start with exactly one of each type of ship for each player. The locations of the computer's ships will be randomly placed. The user, however, must be able to enter the location of each of their ships. The computer's shots into the opponent's grid area should be randomly generated.

**D4.** Heuristic Play for the Game of Battleship
A heuristic is a general "rule of thumb" for solving a problem. Modify the Game of Battleship program from the previous problem so that the locations of the shots that the computer makes into the opponent's grid area are based on heuristics, rather than being randomly generated. Include an explanation of the heuristics developed.

# CHAPTER 5 Functions

*Up until this point, we have viewed a computer program as a single series of instructions. Most programs, however, consist of distinct groups of instructions,*

*each of which accomplishes a specific task. Such a group of instructions is referred to as a "routine." Program routines, called "functions" in Python, are fundamental building blocks in software development. We take our first look at functions in this chapter.*

OBJECTIVES
After reading this chapter and completing the exercises, you will be able to: ♦ Explain the concept of a program routine
♦ Explain the concept of parameter passing
♦ Explain the concept of value-returning and non-value-returning functions ♦ Explain the notion of the side-effects of a function call
♦ Differentiate between local scope and global scope
♦ Define and use functions in Python
♦ Explain the concept of keyword and default arguments in Python ♦ Write a Python program using programmer-defined functions ♦ Effectively use trace statements for program testing

CHAPTER CONTENTS



MOTIVATION

So far, we have limited ourselves to using only the most fundamental features of Python—variables, expressions, control structures, input/print, and lists. In theory, these are the only instructions needed to write any pro gram (that is, to perform any computation). From a *practical* point-of-view, however, these instructions alone are not enough.

The problem is one of complexity. Some smart phones, for example, contain over 10 million lines of code (see Figure 5-1). Imagine

the effort needed to develop and debug software of that size. It certainly cannot be implemented by

any one person, it takes a team of programmers to develop such a project.

In order to manage the complexity of a large problem, it is broken down into smaller subproblems. Then, each subproblem can be focused on and solved separately. In programming, we do the same thing. Programs are divided into manageable pieces called *program routines* (or simply *routines* ). Doing so is a form of *abstraction* in which a more general, less detailed view of a system can be achieved. In addition, program routines provide the opportunity for code reuse, so that systems do not have to be created from "scratch." Routines, therefore, are a fundamental building block in software development.

In this chapter, we look at the definition and use of program routines in Python.

| Term | Number of Lines of Code (LOC) | Equivalent Storage |
|---|---|---|
| KLOC | 1,000 | Application programs |
| MLOC | 1,000,000 | Operating systems / smart phones |
| GLOC | 1,000,000,000 | Number of lines of code in existence for various programming languages |

FIGURE 5-1 Measures of Lines of Program Code

FUNDAMENTAL CONCEPTS

## 5.1 Program Routines

We first introduce the notion of a program routine. We then look in particular at program routines in Python, called *functions*. We have already been using Python's built-in functions such as len, range, and others. We now look more closely at how functions are used in Python, as well as how to define our own.

### 5.1.1 What Is a Function Routine?

A **routine** is a named group of instructions performing some task. A routine can be **invoked**( *called*) as many times as needed in a given program, as shown in Figure 5-2.

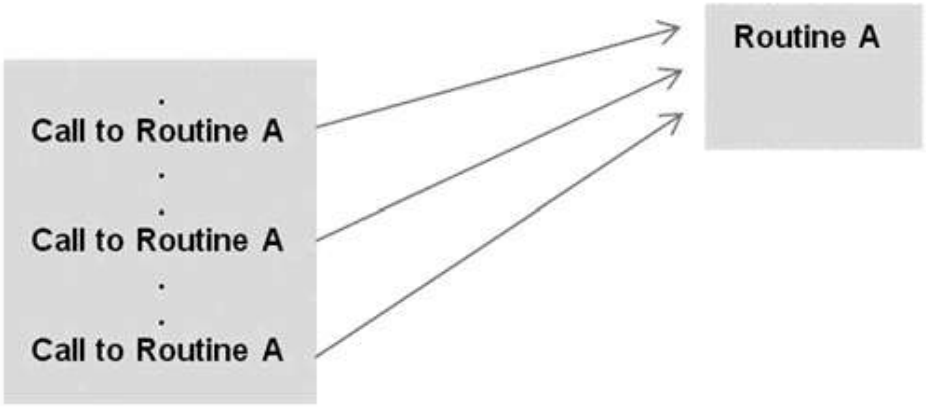



FIGURE 5-2
Program Routine

When a routine terminates, execution automatically returns to the point from which it was called. Such routines may be predefined in the programming language, or designed and implemented by the programmer.

A **function** is Python's version of a program routine. Some functions are designed to return a value, while others are designed for other purposes. We look at these two types of functions next.

A program **routine** is a named group of instructions that accomplishes some task. A routine may be **invoked** (called) as many times as needed in a given program. A **function** is Python's version of a program routine.

### 5.1.2 Defining Functions

In addition to the built-in functions of Python, there is the capability to define

new functions. Such functions may be generally useful, or specific to a particular program. The elements of a function definition are given in Figure 5-3.

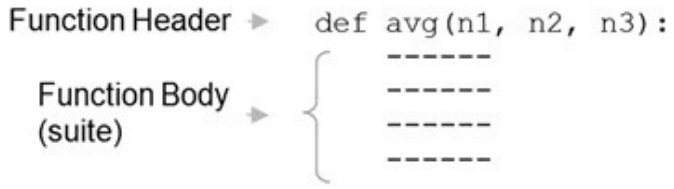



FIGURE 5-3 Example of Python Function Definition

The first line of a function definition is the *function header*. A function header starts with the keyword def, followed by an identifier (avg), which is the function's name. The function name is followed by a comma-separated (possibly empty) list of identifiers (n1, n2, n3) called **formal parameters**, or simply "parameters." Following the *parameter list* is a colon ( ). Following the function header is the body of the function, a suite (program block) containing the function's instructions. As with all suites, the statements must be indented at the same level, relative to the function header.

The number of items in a parameter list indicates the number of values that must be passed to the function, called **actual arguments** (or simply "arguments"), such as the variables num1, num2, and num3 below.

... num1 5 10
... num2 5 25
... num3 5 16

... avg(num1,num2,num3)
Functions are generally defined at the top of a program. However, *every function must be defined before it is called*.
We discuss more about function definition and use in the following sections.

**Actual arguments** , or simply "arguments," are the values passed to functions to be operated on. **Formal parameters**, or simply "parameters," are the "placeholder" names for the arguments passed.

Value-Returning Functions

A **value-returning function** is a program routine called for its return value, and is therefore similar to a mathematical function. Take the simple mathematical

function f(x) 52x. In this notation, “x” stands for any numeric value that function f may be applied to, for example, f(2)5 2x 54. Program functions are similarly used, as illustrated in Figure 5-4.

Function avg takes three arguments (n1, n2, and n3) and returns the average of the three. The *function call*avg(10, 25, 16), therefore, is an expression that evaluates to the returned function value. This is indicated in the function’s *return statement* of the form return *expr*, where *expr* may be any expression. Next, we look at a second form of program routine called for a purpose other than a returned function value.

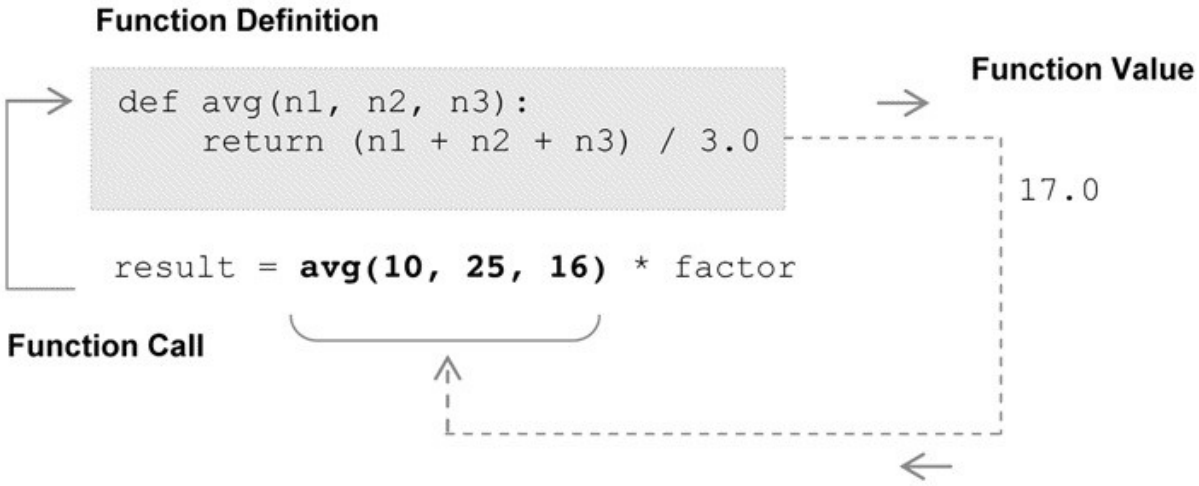



FIGURE 5-4 Call to Value-Returning Function
LET’S TRY IT

From the Python Shell, first enter the following function, making sure to indent the code as given. Hit return twice after the last line of the function is entered. Then enter the following function calls and observe the results.

... def avg(n1, n2, n3): ... avg(40, 10, 25) return (n1 1 n2 1 n3) / 3.0 ???
... avg(10, 25, 40) ... avg(40, 25, 10) ??? ???
A **value-returning function** in Python is a program routine called for its return value, and is therefore similar to a mathematical function.
Non-Value-Returning Functions

A **non-value-returning function** is called not for a returned value, but for its *side effects*. A **side effect** is an action other than returning a function value, such as displaying output on the screen. There is a fundamental difference in the way that value-returning and non-value-returning functions are called. A call to a

value-returning function is an expression, as for the call to function avg: result 5 **avg(10, 25, 16)** * factor.

When non-value-returning functions are called, however, the function call is a *statement*, as shown in Figure 5-5. Since such functions do not have a return value, it is incorrect to use a call to a non-value-returning function as an expression.

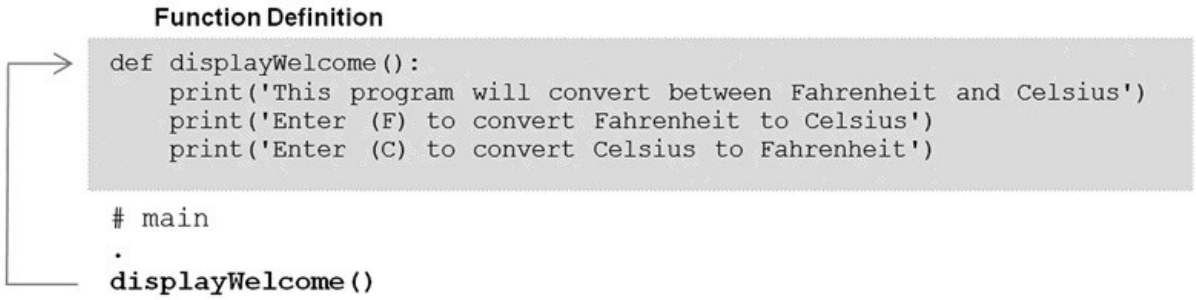


FIGURE 5-5 Call to Non-Value-Returning Function

In this example, function displayWelcome is called only for the side-effect of the screen output produced. Finally, every function in Python is technically a value-returning function since *any function that does not explicitly return a function value (via a return statement) automatically returns the special value*None. We will, however, consider such functions as non-value-returning functions.

LET'S TRY IT
From the Python Shell, first enter the following function, making sure to indent the code as given. Then enter the following function calls and observe the results.
... def hello(name): ... name 5 'John' print('Hello', name 1 '!') ... hello(name)
???
A **non-value-returning function** is a function called for its *side effects*, and not for a returned function value.
5.1.3 Let's Apply It—Temperature Conversion Program
(Function Version)

The following is a program (Figure 5-7) that allows a user to convert a range of values from Fahrenheit to Celsius, or Celsius to Fahrenheit, as presented in Chapter 3. In this version, however, the program is designed with the use of functions. This program utilizes the following programming features.

➤ value-returning functions ➤ non-value-returning functions
Example execution of the program is given in Figure 5-6.

```
This program will convert a range of temperatures
Enter (F) to convert Fahrenheit to Celsius
Enter (C) to convert Celsius to Fahrenheit

Enter selection: F
Enter starting temperature to convert: 65
Enter ending temperature to convert: 95

  Degrees  Degrees
Fahrenheit Celsius
   65.0     18.3
   66.0     18.9
   67.0     19.4
   68.0     20.0
   69.0     20.6
   70.0     21.1
   71.0     21.7
   72.0     22.2
   73.0     22.8
   74.0     23.3
   75.0     23.9
   76.0     24.4
   77.0     25.0
   78.0     25.6
   79.0     26.1
   80.0     26.7
   81.0     27.2
   82.0     27.8
   83.0     28.3
   84.0     28.9
   85.0     29.4
   86.0     30.0
   87.0     30.6
   88.0     31.1
   89.0     31.7
   90.0     32.2
   91.0     32.8
   92.0     33.3
   93.0     33.9
   94.0     34.4
   95.0     35.0
```

FIGURE 5-6 Execution of Temperature Conversion Program

```
1  # Temperature Conversion Program (Celsius-Fahrenheit / Fahrenheit-Celsius)
2
3  def displayWelcome():
4
5     print('This program will convert a range of temperatures')
6     print('Enter (F) to convert Fahrenheit to Celsius')
7     print('Enter (C) to convert Celsius to Fahrenheit\n')
8
9  def getConvertTo():
10
11    which = input('Enter selection: ')
12    while which != 'F' and which != 'C':
13       which = input('Enter selection: ')
14
15    return which
16
17 def displayFahrenToCelsius(start, end):
18
19     print('\n  Degrees', ' Degrees')
20     print('Fahrenheit', 'Celsius')
21
22     for temp in range(start, end + 1):
23        converted_temp = (temp - 32) * 5/9
24        print('  ', format(temp, '4.1f'), '    ', format(converted_temp, '4.1f'))
25
26 def displayCelsiusToFahren(start, end):
27
28     print('\n  Degrees', ' Degrees')
29     print('  Celsius', 'Fahrenheit')
30
31     for temp in range(start, end + 1):
32        converted_temp = (9/5 * temp) + 32
33        print('  ', format(temp, '4.1f'), '    ', format(converted_temp, '4.1f'))
34
35 # ---- main
36
37 # Display program welcome
38 displayWelcome()
39
40 # Get which converion from user
41 which = getConvertTo()
42
43 # Get range of temperatures to convert
44 temp_start = int(input('Enter starting temperature to convert: '))
45 temp_end = int(input('Enter ending temperature to convert: '))
46
47 # Display range of converted temperatures
48 if which == 'F':
49    displayFahrenToCelsius(temp_start, temp_end)
50 else:
51    displayCelsiusToFahren(temp_start, temp_end)
```

FIGURE 5-7 Temperature Conversion Program (Function Version)

In **lines 3–29** are defined functions displayWelcome, getConvertTo, displayFahrenToCelsius, and displayCelsiusToFahren. The functions are directly called from the main module of the program in **lines 32–48**.

On **line 35**, the non-value-returning function displayWelcome is called. Its job is to display information about the program to the user. It does not need to be passed any arguments since it performs the same output each time it is called. Next, on **line 38**, value-returning function getConvertTo is called. This function also is not passed any arguments. It simply asks the user to enter either 'F' or 'C' to indicate whether they want to convert from Fahrenheit to Celsius, or Celsius to Fahrenheit. The input value entered is returned as the function value.

The instructions on **line 41–42** then prompt the user for the start and end range of temperatures to be converted. This task does not warrant the construction of a function since there are only two input instructions to accomplish this.

The final part of the program displays the converted range of temperatures. Two non-valuereturning functions are defined for accomplishing this task —displayFahrenToCelsius and displayCelsiusToFahren. Each is passed two arguments, temp_start and temp_end, which indicate the range of temperature values to be converted.

What is left to look at is the implementation of each of the individual functions. The implementation of function displayWelcome ( **lines 3–6**) is very straightforward. It simply contains three print instructions. Function getConvertTo ( **lines 8–13**) contains a call to input followed by a while loop that performs input validation. The user is forced to enter either 'F' or 'C', and is continually prompted to re-enter as long as a value other than these two values is entered. When the loop terminates, variable which is returned by the return statement in **line 13**.

Function displayFahrenToCelsius ( **lines 15–21**) and function displayCelsiusToFahren ( **lines 23–29**) are similar in design. Each contains two parameters—start and end (which are each passed actual arguments temp_start and temp_end in the main section of the program). Each first prints the appropriate column headings followed by a for statement that iterates variable temp over the requested temperature range. The conversion formula is different in each, however. Each has the same final print instruction to print out the original temperature and the converted temperature in each of the columns.

Self-Test Questions

**1.** The values passed in a given function call in Python are called,

**(a)** formal parameters
**(b)** actual arguments

**2.** The identifiers of a given Python function providing names for the values passed to it are called, **(a)** formal parameters
**(b)** actual arguments

**3.** Functions can be called as many times as needed in a given program. (TRUE/FALSE) **4.** When a given function is called, it is said to be,
**(a)** subrogated **(b)** invoked **(c)** activated
**5.** Which of the following types of functions must contain a return statement, **(a)** value-returning functions **(b)** non-value-returning functions
**6.** Value-returning function calls are,
**(a)** expressions **(b)** statements
**7.** Non-value-returning function calls are,
**(a)** expressions **(b)** statements
**8.** Which of the following types of routines is meant to produce side effects? **(a)** value-returning functions **(b)** non-value-returning functions
ANSWERS: 1. (b), 2. (a), 3. TRUE, 4. (b), 5. (a), 6. (a), 7. (b), 8. (b)

## 5.2 More on Functions

In this section we further discuss issues related to function use, including more on function invocation and parameter passing.

### 5.2.1 Calling Value-Returning Functions

Calls to value-returning functions can be used anywhere that a function's return value is appropriate,

result 5 max(num_list) * 100

Here, we apply built-in function max to a list of integers, num_list. Examples of additional allowable forms of function calls are given below.

(a) result 5 max(num_list1) * max(num_list2)
(b) result 5 abs(max(num_list))
(c) if max(num_list) , 10:...
(d) print('Largest value in num_list is ', max(num_list))

The examples demonstrate that an expression may contain multiple function calls, as in (a); a function call may contain function calls as arguments, as in (b); conditional expressions may contain function calls, as in (c); and the arguments in print function calls may contain function calls, as in (d).

What if a function is to return more than one value, such as function maxmin to

return *both* the maximum and minimum values of a list of integers? In Python, we can do this by returning the two values as a single tuple,

function definition
def maxmin(num_list):
return (max(num_list), min(num_list))
function use
weekly_temps 5 [45, 30, 52, 58, 62, 48, 49] (a) highlow_temps 5
maxmin(weekly_temps) (b) high, low 5 maxmin(weekly_temps)

In (a) above, the returned tuple is assigned to a single variable, highlow_temps. Thus, highlow_ temps[0] contains the maximum temperature, and highlow_temps[1] contains the minimum temperature. In (b), however, a *tuple assignment* is used. In this case, variables high and low are each assigned a value of the tuple based on the order that they appear. Thus, high is assigned to the tuple value at index 0, and low the tuple value at index 1 of the returned tuple.

Note that it does not make sense for a call to a value-returning function to be used as a statement, for example,
max(num_list)
Such a function call does not have any utility because the expression would evaluate to a value that is never used and thus is effectively "thrown away."

Finally, we can design value-returning functions that do not take any arguments, as we saw in the getConvertTo function of the previous temperature conversion program. Empty parentheses are used in both the function header and the function call. This is needed to distinguish the identifier as denoting a function name and not a variable.

LET'S TRY IT Enter the definitions of functions avg (from section 5.1.2) and minmax given above. Then enter the following function calls and observe the results. ...avg(10,25,40)
???
... num_list 5 [10,20,30]

... avg(10,25,40) 1 10 ???
... if avg(10,25,240) , 0:

print 'Invalid avg' ???
... max_min 5 maxmin(num_list) ... max_min[0]

???
... max_min[1]
???

... avg(avg(2,4,6),8,12) ???
... avg(1,2,3) * avg(4,5,6) ???
... max, min 5 maxmin(num_list) ... max
???
... min
???

Function calls to value-returning functions can be used anywhere that a function's return value is appropriate.

5.2.2 Calling Non-Value-Returning Functions

As we have seen, non-value-returning functions are called for their side effects, and not for a returned function value. Thus, such function calls are statements, and therefore can be used anywhere that an executable statement is allowed. Consider such a function call to displayWelcome from Figure 5-7,

displayWelcome()

It would not make sense to treat this function call as an expression, since no meaningful value is returned (only the default return value None). Thus, for example, the following assignment statement would not serve any purpose,

welcome_displayed 5 displayWelcome()

Finally, as demonstrated by function displayWelcome(), functions called for their side effects can be designed to take no arguments, the same as we saw for value-returning functions. Parentheses are still included in the function call to indicate that identifier displayWelcome is a function name, and not a variable.

LET'S TRY IT

Enter the definition of function hello given below, then enter the following function calls and observe the results.

... def sayHello(): print('Hello!')
... sayHello()
???

```
... t 5 sayHello()
???
... t
???
... t 55 None
???
... def buildHello(name):
return 'Hello' 1 name 1 '!'
... greeting 5 buildHello('Charles')
... print(greeting)
???
... buildHello('Charles')
???
... buildHello()
???
```

Function calls to non-value-returning functions can be used anywhere that an executable statement is allowed.

### 5.2.3 Parameter Passing

Now that we have discussed how functions are called, we take a closer look at the passing of arguments to functions.

#### Actual Arguments vs. Formal Parameters

Parameter passing is the process of passing arguments to a function. As we have seen, actual arguments are the values passed to a function’s formal parameters to be operated on. This is illustrated in Figure 5-8.

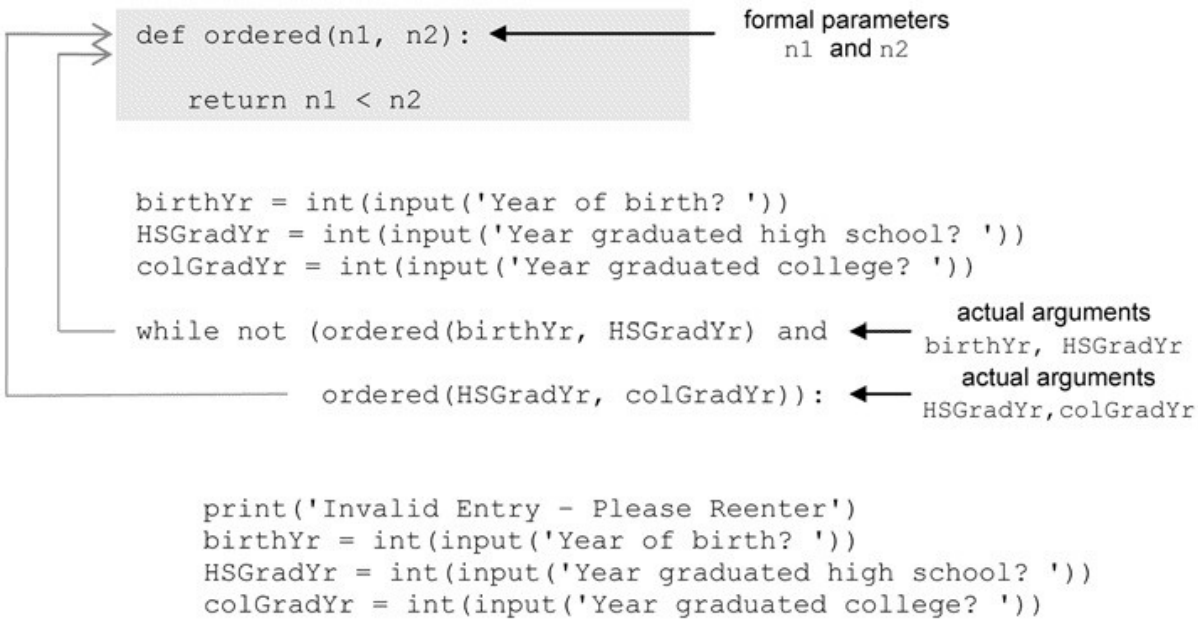


FIGURE 5-8 Parameter Passing

Here, the values of birthYr (the user's year of birth) and HSGradYr (the user's year of high school graduation) are passed as the actual arguments to formal parameters n1 and n2. Each call is part of the same Boolean expression ordered(birthYr, HSGradYr) and ordered(HSGradYr, colGradYr). In the second function call of the expression, a *different* set of values HSGradYr and colGradYr are passed. Formal parameter names n1 and n2, however, remain the same.

Note that the correspondence of actual arguments and formal parameters is determined by the *order* of the arguments passed, and not their names. Thus, for example, it is perfectly fine to pass an actual argument named num2 to formal parameter n1, and actual argument num1 to formal parameter n2, as given in Figure 5-9.

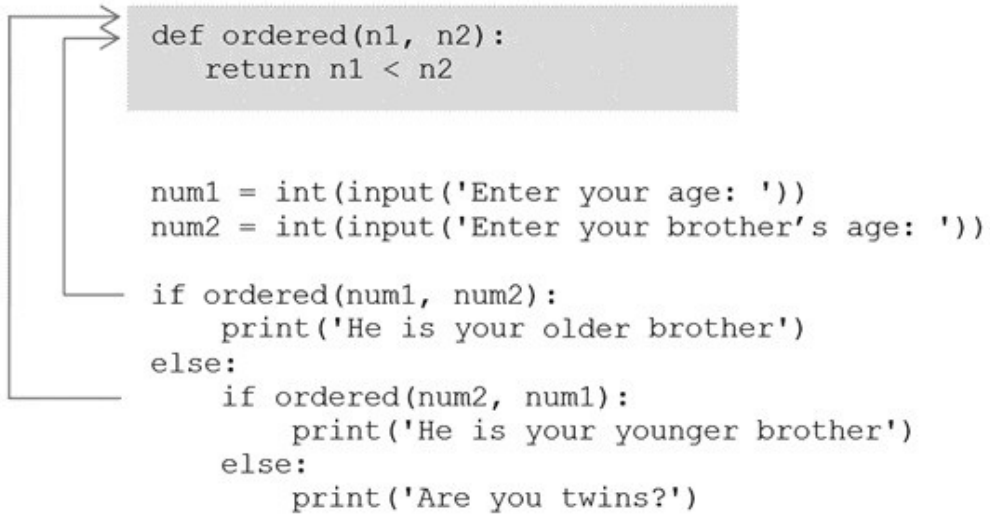



FIGURE 5-9

Parameter Passing and Argument Names

In this example, function ordered is called once with arguments num1, num2 and a second time with arguments num2, num1. Each is a proper function call and each is what is logically needed in this instance.

LET'S TRY IT Enter the definition of function ordered given above into the Python Shell. Then enter the following and observe the results.
... nums_1 5 [5,2,9,3] ... ordered(max(nums_1), max(nums_2)) ... nums_2 5 [8,4,6,1] ... ???
... ordered(min(nums_1), max(nums_2)) ???
The correspondence of actual arguments and formal parameters is determined by the *order* of the arguments passed, and not their names.
Mutable vs. Immutable Arguments

There is an issue related to parameter passing that we have yet to address. We know that when a function is called, the current values of the arguments passed become the initial values of their corresponding formal parameters,

def avg(n1, n2, n3):
**avg(10, 25, 40)**
In this case, literal values are passed as the arguments to function avg. When variables are passed as actual arguments, however, as shown below,
def avg( **n1**, **n2**, **n3**):
avg( **num1**, **num2**, **num3**)

there is the question as to whether any changes to formal parameters n1, n2, and

n3 in the function result in changes to the corresponding actual arguments num1, num2, and num3. In this case, function avg doesn’t assign values to its formal parameters, so there is no possibility of the actual arguments being changed. Consider, however, the following function,

def countDown(n):
while n .5 0:
if (n != 0):
print(n, '..', end='')

else:
print(n)
n 5 n 2 1

This function simply displays a countdown of the provided integer parameter value. For example, function call countDown(4) produces the following output,
4 . . 3 . . 2 . . 1 . . 0

What if the function call contained a variable as the argument, for example, countDown(num_ tics)? Since function countDown alters the value of formal parameter n, decrementing it until it reaches the value − 1, does the corresponding actual argument num_tics have value − 1 as well?

... num_tics 5 10
... countDown(num_tics) ... num_tics
???

If you try this, you will see that num_tics is unchanged. Now consider the following function,
def sumPos(nums): ... nums_1 5 [5, 22, 9, 4, 26, 1] for k in range(0, len(nums)):
... total 5 sumPos(nums_1) if nums[k] , 0: ... total
nums[k] 5 0 19
... nums_1
return sum(nums) [5,0,9,4,0,1]

Function sumPos returns the sum of only the positive numbers in the provided argument. It does this by first replacing all negative values in parameter nums with 0, then summing the list using built-in function sum. We see above that the corresponding actual argument nums_1 has been altered in this case, with all of the original negative values set to 0.

The reason that there was no change in integer argument num_tics above but there was in list argument nums_1 has to do with their types. Lists are mutable. Thus, arguments of type list will be altered if passed to a function that alters its value. Integers, floats, Booleans, strings, and tuples, on the other hand, are immutable. Thus, arguments of these types cannot be altered as a result of any function call.

It is generally better to design functions that do not return results through their arguments. In most cases, the result should be returned as the function's return value. What if a function needs to return more than one function value? The values can be returned in a tuple, as discussed above.

LET'S TRY IT Enter the following and observe the results. ... num 5 10

... def incr(n): n 5 n 1 1
... incr(num)
... num
??? ??? ... nums_1 5 [1,2,3] ... nums_2 5 (1,2,3) ... def update(nums): ... update(nums_2) nums[1] 5 nums[1] 1 1 ... ???
... update(nums_1)
... nums_1

Only arguments of mutable type can be altered when passed as an argument to a function. In general, function results should be through a function's return value, and not through altered parameters.

5.2.4 Keyword Arguments in Python

The functions we have looked at so far were called with a fixed number of positional arguments. A **positional argument** is an argument that is assigned to a particular parameter based on its position in the argument list, as illustrated below.

def mortgage_rate(amount, rate, term)
monthly_payment 5 mortgage_rate(350000, 0.06, 20)
This function computes and returns the monthly mortgage payment for a given loan amount (amount), interest rate (rate), and number of years of the loan (term).

Python provides the option of calling any function by the use of keyword arguments. A **keyword argument** is an argument that is specified by parameter

name, rather than as a positional argument as shown below (note that keyword arguments, by convention, do not have a space before or after the equal sign),

def mortgage_rate(amount, rate, term)
monthly_payment 5 mortgage_rate(rate 50.06, term 520, amount 5350000)

This can be a useful way of calling a function if it is easier to remember the parameter names than it is to remember their order. It is possible to call a function with the use of both positional and keyword arguments. However, all positional arguments must come before all keyword arguments in the function call, as shown below.

def mortgage_rate(amount, rate, term)
monthly_payment 5 mortgage_rate(35000, term 520, rate 50.06)
This form of function call might be useful, for example, if you remember that the first argument is the loan amount, but you are not sure of the order of the last two arguments rate and term.
LET'S TRY IT
Enter the following function definition in the Python Shell. Execute the statements below and observe the results.
... def addup(first, last): ... addup(1,10) ???
if first . last:
sum 5 21 1, last510)

else: ???
sum 5 0
for i in range(first, last11): ... addup(last510, first51)

sum 5 sum 1 i ???
return sum
A **positional argument** is an argument that is assigned to a particular parameter based on its position in the argument list. A **keyword argument** is an argument that is specified by parameter name.
5.2.5 Default Arguments in Python

Python also provides the ability to assign a default value to any function parameter allowing for the use of default arguments. A **default argument** is an argument that can be optionally provided, as shown here.

def mortgage_rate(amount, rate, term 520)

monthly_payment 5 mortgage_rate(35000, 0.62)

In this case, the third argument in calls to function mortgage_rate is optional. If omitted, parameter term will default to the value 20 (years) as shown. If, on the other hand, a third argument is provided, the value passed replaces the default parameter value. All positional arguments must come before any default arguments in a function definition.

LET'S TRY IT Enter the following function definition in the Python Shell. Execute the statements below and observe the results.
... def addup(first, last, incr51): addup(1,10)

???
if first . last:
sum 5 21

else:
sum 5 0
for i in range(first, last11, incr):

sum 5 sum 1 i
return sum
... addup(1,10,2) ???
... addup(first=1, last=10) ???
... addup(incr=2, first=1,

last=10) ???
A **default argument** is an argument that can be optionally provided in a given function call. When not provided, the corresponding parameter provides a default value.
5.2.6 Variable Scope

Looking back at the temperature conversion program in section 5.1.3, we see that functions displayFahrenToCelsius and displayCelsiusToFahren each contain variables named temp and converted_temp. We ask, "Do these identifiers refer to common entities, or does each function have its own distinct entities?" The answer is based on the concept of identifier *scope*, which we discuss next.

Local Scope and Local Variables

A **local variable** is a variable that is only accessible from within a given function. Such variables are said to have **local scope**. In Python, any variable assigned a value in a function becomes a local variable of the function. Consider the example in Figure 5-10.

```
def func1():
    n = 10
    print('n in func1 = ', n)

def func2():
    n = 20
    print('n in func2 before call to func1 = ', n)
    func1()
    print('n in func2 after call to func1 = ', n)

>>> func2()
n in func2 before call to func1 =  20
n in func1 =  10
n in func2 after call to func1 =  20
```

FIGURE 5-10 Defining Local Variables

Both func1 and func2 contain identifier n. Function func1 assigns n to 10, while function func2 assigns n to 20. Both functions display the value of n when called —func2 displays the value of n both *before* and *after* its call to func1. If identifier n represents the same variable, then shouldn't its value change to 10 after the call to func1? However, as shown by the output, the value of n remains 20. This is because there are *two* distinct instances of variable n, each local to the function assigned in and inaccessible from the other.

Now consider the example in Figure 5-11. In this case, the functions are the same as above except that the assignment to variable n in func1 is commented out.

```
def func1():
    # n = 10
    print('n in func1 = ', n)

def func2():
    n = 20
    print('n in func2 before call to func1 = ', n)
    func1()
    print('n in func2 after call to func1 = ', n)
```

```
>>> func2()
n in func2 before call to func1 =  20
Traceback (most recent call last):
  .
  .
    print('n in func1 = ', n)
NameError: global name 'n' is not defined
```

FIGURE 5-11 Inaccessibility of Local Variables

In this case, we get an error indicating that variable n is not defined within func1. This is because variable n defined in func2 is inaccessible from func1. (In this case, n is expected to be a *global* variable, discussed next.)

The period of time that a variable exists is called its **lifetime**. Local variables are automatically created (allocated memory) when a function is called, and destroyed (deallocated) when the function terminates. Thus, the lifetime of a local variable is equal to the duration of its function's execution. Consequently, the values of local variables are not retained from one function call to the next.

The concept of a local variable is an important one in programming. It allows variables to be defined in a function without regard to the variable names used in other functions of the program. It also allows previously written functions to be easily incorporated into a program. The use of global variables, on the other hand, brings potential havoc to programs, discussed next.

LET'S TRY IT Enter the following function definition in the Python Shell. Execute the statements below and observe the results.
... def func1(): ... func1() some_var 5 10 ... some_var ???
A **local variable** is a variable that is only accessible from within the function it resides. Such variables are said to have **local scope**.

Global Variables and Global Scope

A **global variable** is a variable that is defined outside of any function definition. Such variables are said to have **global scope**. This is demonstrated in Figure 5-12.

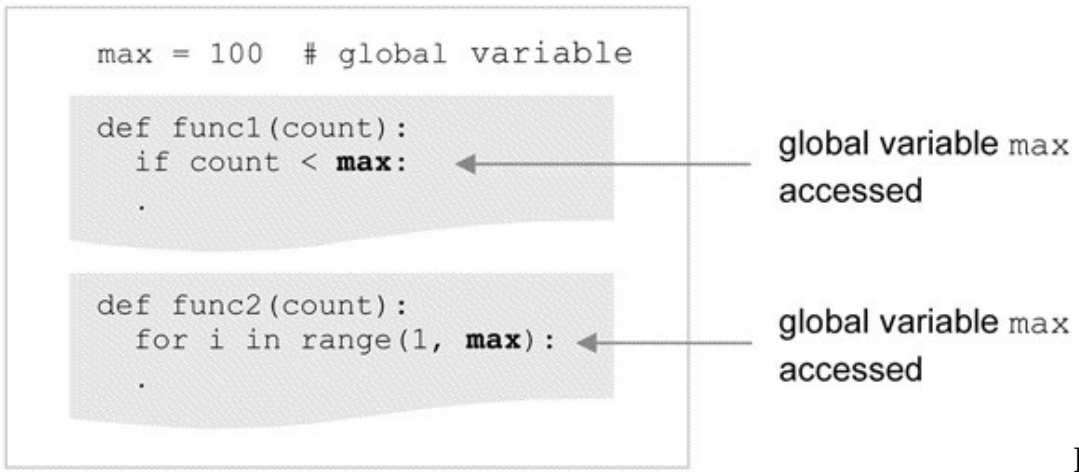



FIGURE 5-12 Access to Value of Global Variable

V ariable max is defined outside func1 and func2 and therefore “global” to each. As a result, it is directly accessible by both functions. For this reason, *the use of global variables is generally considered to be bad programming style*. Although it provides a convenient way to share values among functions, *all* functions within the scope of a global variable can access and alter it. This may include functions that have no need to access the variable, but none-the-less may unintentionally alter it.

Another reason that the use of global variables is bad practice is related to code reuse. If a function is to be reused in another program, the function will not work properly if it is reliant on the existence of global variables that are nonexistent in the new program. Thus, it is good programming practice to design functions so all data needed for a function (other than its local variables) are explicitly passed as arguments, and not accessed through global variables.

A **global variable** is a variable defined outside of any function definition. Such variables are said to have **global scope**. The use of global variables is considered bad programming practice.

5.2.7 Let’s Apply It—GPA Calculation Program

The following program (Figure 5-14) computes a semester GPA and new cumulative GPA for a given student. This program utilizes the following programming features:

tuple assignment ➤
Figure 5-13 illustrates an example execution of the program.

```
This program calculates semester and cumulative GPAs

Enter total number of earned credits: 30
Enter your current cumulative GPA: 3.25

Enter grade (hit Enter if done): A
Enter number of credits: 4
Enter grade (hit Enter if done): A
Enter number of credits: 3
Enter grade (hit Enter if done): B
Enter number of credits: 3
Enter grade (hit Enter if done): B
Enter number of credits: 3
Enter grade (hit Enter if done): A
Enter number of credits: 3
Enter grade (hit Enter if done):

Your semester GPA is 3.62
Your new cumulative GPA is 3.38
>>>
```

FIGURE 5-13 Execution of GPA Calculation Program

```
1  # Semester GPA Calculation
2
3  def convertGrade(grade):
4      if grade == 'F':
5          return 0
6      else:
7          return 4 - (ord(grade) - ord('A'))
8
9  def getGrades():
10     semester_info = []
11     more_grades = True
12     empty_str = ''
13
```

FIGURE 5-14 GPA Calculation Program (*Continued*)

```
    while more_grades:
        course_grade = input('Enter grade (hit Enter if done): ')
        while course_grade not in ('A','B','C','D','F',empty_str):
            course_grade = input('Enter letter grade received: ')
            if course_grade == empty_str:
            more_grades = False
        else:
            num_credits = int(input('Enter number of credits: '))
            semester_info.append([num_credits, course_grade])

def calculateGPA(sem_grades_info, cumm_gpa_info):
    sem_quality_pts = 0
    sem_credits = 0
    current_cumm_gpa, total_credits = cumm_gpa_info

    for k in range(len(sem_grades_info)):
        num_credits, letter_grade = sem_grades_info[k]

        sem_quality_pts = sem_quality_pts + \
                          num_credits * convertGrade(letter_grade)

        sem_credits = sem_credits + num_credits

    sem_gpa = sem_quality_pts / sem_credits
    new_cumm_gpa = (current_cumm_gpa * total_credits + sem_gpa * \
                    sem_credits) /(total_credits + sem_credits)

    return (sem_gpa, new_cumm_gpa)

# ---- main

# program greeting
print('This program calculates new semester and cummulative GPAs\n')

# get current GPA info
total_credits = int(input('Enter total number of earned credits: '))
cumm_gpa = float(input('Enter your current cummulative GPA: '))
cumm_gpa_info = (cumm_gpa, total_credits)

# get current semester grade info
print()
semester_grades = getGrades()

# calculate semester gpa and new cummulative gpa
semester_gpa, cumm_gpa = calculateGPA(semester_grades, cumm_gpa_info)

# display semester gpa and new cummulative gpa
print('\nYour semester GPA is', format(semester_gpa, '.2f'))
print('Your new cummulative GPA is', format(cumm_gpa, '.2f'))
```

FIGURE 5-14 GPA Calculation Program

The program begins with the display of the program greeting on **line 49**. **Lines 49–50** get the number of earned credits (total_credits) and current cumulative

GPA (cumm_gpa) from the user. These two variables are bundled into a tuple named cumm_gpa_info on **line 51**. Since they are always used together, bundling these variables allows them to be passed to functions as one parameter rather than as separate parameters.

Function getGrades is called on **line 55**, which gets the semester grades from the user and assigns it to variable semester_grades. The value returned by function getGrades is a list of sublists, in which each sublist contains the letter grade for a given course, and the associated number of credits,

[['A', 3], ['B', 4], ['A', 3], ['C', 3]]

On **line 58**, function calculateGPA is called with arguments semester_grades and cumm_ gpa_info. The function returns a tuple containing the semester GPA and new cumulative GPA of the user. A tuple assignment is used to unpack the two values into variables semester_gpa and cumm_gpa. Finally, these values are displayed on **lines 61** and **62**.

Function calculateGPA is defined in **lines 24–41** with parameters sem_grades_ info and cumm_gpa_info. A GPA is calculated as the total quality points earned for a given set of courses, divided by the total number of credits the courses are worth. The number of *quality points* for a given course is defined as a course grade times the number of credits the course is worth. Thus, assuming a grade of A is worth 4 points, B worth 3 points, and grades of C, D, and F worth 2, 1 and 0 points, respectively, to calculate the semester GPA for a student receiving A's in two four-credit courses, B's in two three-credit courses, and a C in a one-credit course would be,

(4 * 4 1 4 * 4 1 3 * 3 1 3 * 3 1 2 * 1) / 15 5 3.47
A A B B C
where 15 is the total number of credits of all courses.

Similarly, in order to calculate a new cumulative GPA, the total quality points of the current cumulative GPA plus the total quality points of the new semester GPA is divided by the total number of credits the student has earned to date. Thus, to calculate a new cumulative GPA for a current cumulative GPA of 3.25 earning thirty credits, and a new semester GPA as given above (3.47 earning fifteen credits) would be,

(3.25 * 30 1 3.47 * 15) / 45 5 3.32

with 45 total earned credits. Thus, in function calculateGPA, local variables sem_quality_ pts and sem_credits are initialized to zero. Their values for the courses provided in parameter sem_grades_info are computed in the for loop on **lines 29–35**. This loop also calculates the semester quality points and the number of credits of the current semester, assigned to local variables sem_quality_pts and sem_credits, respectively (at **lines 32** and **35**). Note that in the calculation of the semester quality points, function convertGrade is called to convert each letter grade to its corresponding numerical value. Finally, at the end of function calculateGPA, local variable sem_gpa is assigned to the total semester quality points divided by the total semester credits. Similarly, local variable new_cumm_gpa is assigned to the total quality points to date (current_cumm_gpa * total_credits 1sem_gpa * sem_credits) divided by the total number of credits earned to date (num_credits 1sem_credits). Finally, on **line 41**, a tuple is returned containing both of these computed values.

The remaining functions defined in this program are convertGrade and getGrades. Function convertGrade is passed a letter grade, and returns the corresponding numerical value. Since the ordinal value (via the ord function) of letters in Python are sequential integers, determining the difference between the ordinal value of A and the ordinal value of a given letter grade allows the numerical value of the letter grade to be determined. For example, for a letter grade of A through D, its numerical value is determined and returned as,

return 4 2 (ord('A') 2 ord('A')) → return 4
return 4 2 (ord('B') 2 ord('A')) → return 3
return 4 2 (ord('C') 2 ord('A')) → return 2
return 4 2 (ord('D') 2 ord('A')) → return 1

Since there is no letter of grade E used, a grade of F has to be handled separately.

Finally , function getGrades returns a list of sublists of grades and credits entered by the user, as mentioned above. Thus, local variable semester_info is initialized to an empty list on **line 10**. The while loop at **line 14** iterates until Boolean variable more_grades is False, initialized to True in **line 11**. The loop continues to iterate and append another pair of grade/credits to the list until the user hits the Enter key when prompted for a course grade ( **line 15**).

Self-Test Questions **1.** A function call can be made anywhere within a program in which the return type of the function is appropriate. (TRUE/FALSE)

**2.** An expression may contain more than one function call. (TRUE/FALSE)
**3.** Function calls may contain arguments that are function calls. (TRUE/FALSE)
**4.** All value-returning functions must contain at least one parameter. (TRUE/FALSE)
**5.** Every function must have at least one mutable parameter. (TRUE/FALSE)

**6.** A local variable in Python is a variable that is,
**(a)** defined inside of every function in a given program
**(b)** local to a given program
**(c)** only accessible from within the function it is defined

**7.** A global variable is a variable that is defined outside of any function definition. (TRUE/FALSE)
**8.** The use of global variables is a good way to allow different functions to access and modify the same variables. (TRUE/FALSE)

ANSWERS: 1. True, 2. True, 3. True, 4. False, 5. False, 6. (c), 7. True, 8. False

COMPUTATIONAL PROBLEM SOLVING

5.3 Credit Card Calculation Program

In this section, we design, implement, and test a program that will allow us to determine the length of time needed to pay off a credit card balance, as well as the total interest paid.

5.3.1 The Problem

The problem is to generate a table showing the decreasing balance and accumulating interest paid on a credit card account for a given credit card balance, interest rate, and monthly payment, as shown in Figure 5-15.

| Year | Balance | Interest Paid |
|---|---|---|
| 1 | 330.18 | 5.17 |
| | 315.13 | 10.13 |
| | 299.85 | 14.85 |
| | 284.35 | 19.35 |
| | 268.62 | 23.62 |
| | 252.65 | 27.65 |
| | 236.44 | 31.44 |
| | 219.98 | 34.98 |
| | 203.28 | 38.28 |
| | 186.33 | 41.33 |
| | 169.13 | 44.13 |
| 2 | 151.66 | 46.66 |
| | 133.94 | 48.94 |
| | 115.95 | 50.95 |
| | 97.69 | 52.69 |
| | 79.15 | 54.15 |
| | 60.34 | 55.34 |
| | 41.25 | 56.25 |
| | 21.86 | 56.86 |
| | 2.19 | 57.19 |
| | 0.00 | 57.22 |

FIGURE 5-15 Example Execution of the Credit Card Calculation Program

### 5.3.2 Problem Analysis

The factors that determine how quickly a loan is paid off are the amount of the loan, the interest rate charged, and the monthly payments made. For a fixed-rate home mortgage, the monthly payments are predetermined so that the loan is paid off within a specific number of years. Therefore, the total interest that will be paid on the loan is made evident at the time the loan is signed.

For a credit card, there is only a minimum payment required each month. It is not always explicitly stated by the credit card company, however, how long it would take to pay off the card by making only the minimum payment. The minimum payment for a credit card is dependent on the particular credit card

company. However, it is usually around 2–3% of the outstanding loan amount each month, and no less than twenty dollars. Thus, calculating this allows us to project the amount of time that it would take before the account balance becomes zero, as well as the total interest paid.

### 5.3.3 Program Design

#### Meeting the Program Requirements

No particular format is specified for how the output is to be displayed. All that is required is that the user be able to enter the relevant information and that the length of time to pay off the loan and the total interest paid is displayed. The user will also be given the choice of assuming the monthly payment to be the required minimum payment, or a larger specified amount.

#### Data Description

All that needs to be represented in this program are numerical values for the loan amount, the interest rate, and the monthly payment made. There is no need to create a data structure as the table of payments can be generated as it is displayed.

#### Algorithmic Approach

The only algorithm needed for this problem is the calculation of the required minimum payment. The minimum payment is usually calculated at 2% or 3% of the outstanding balance, with a lower limit of around $20. Therefore, we will assume a worst case scenario of a minimum payment calculated at 2%, with a minimum payment of $20.

#### Overall Program Steps

The overall steps in this program design are given in Figure 5-16.

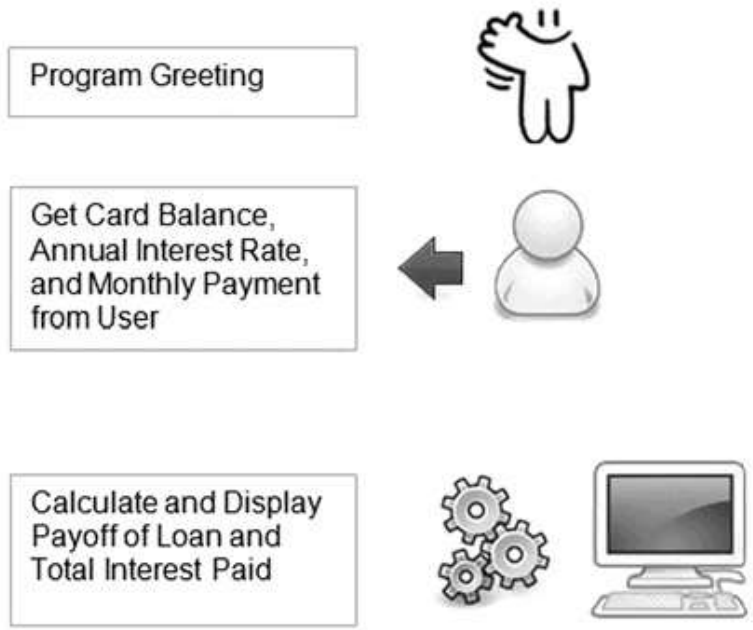



FIGURE 5-16 Overall Steps of the Credit Card Calculation Program

5.3.4 Program Implementation and Testing
Stage 1—Developing the Overall Program Structure

We first develop and test the overall program structure given in Figure 5-17.

The program begins on **line 15** with a call to function displayWelcome(). Next, the current credit card balance and annual interest rate (APR) are input from the user ( **lines 18–19**), each read as an integer value. Since the monthly interest rate is what will be used in the calculations, the value in apr is divided by 1200 (on **line 21**). This converts the value to a monthly interest rate, as well as converting it to decimal form (for example, 18% as 0.18).

The final value input from the user is the monthly payments that they wish to have the payoff calculated with. They have a choice of either going with the minimum required monthly payment, assumed to be $20 for testing purposes ( **line 28**), or a specified monthly payment ( **line 31**). The credit card balance, annual percentage rate, and the assumed monthly payments are passed to function displayPayments (on **line 34**) to calculate and display the pay down of the balance as well as the interest paid over each month of the payoff period.

```
1  # Credit Card Calculation Program (Stage 1)
2
3  def displayWelcome():
4      print('\n.... Entering function display welcome')
5
6  def displayPayments(balance, int_rate, monthly_payment):
7      print('\n.... Entering function displayPayments')
8      print('parameter balance =', balance)
9      print('parameter int_rate =', int_rate)
10     print('parameter monthly_payment =', monthly_payment)
11
12 # ---- main
13
14 # display welcome screen
15 displayWelcome()
16
17 # get current balance and APR
18 balance = int(input('\nEnter the balance on your credit card: '))
19 apr = int(input('Enter the interest rate (APR) on the card: '))
20
21 monthly_int_rate = apr/1200
22
23 # determine monthly payment
24 response = input('Use the minimum monthly payment? (y/n): ')
25
26 if response in ('y','Y'):
27     print('Minimum payment selected')
28     monthly_payment = 20
29 else:
30     print('User-entered monthly payments selected')
31     monthly_payment = input('Enter monthly payment: ')
32
33 # display monthly payoff
34 displayPayments(balance, monthly_int_rate, monthly_payment)
```

FIGURE 5-17 Credit Card Calculation Program (Stage 1)

Functions displayWelcome ( **line 3**) and displayPayments ( **line 6**) consist only of **trace statements**. A trace statement prints, for testing purposes, a message indicating that a certain point in the program has been reached. Trace statements are also used to display the value of certain variables. Once this part of the program is working, we can focus on implementing the functions and further developing the main program section.

Stage 1 Testing
We show sample test runs of this version of the program in Figure 5-18.

From the test results, we see that the appropriate values are being input and passed to function displayPayments. So it looks like the overall structure of this

stage of the program is working correctly.

Stage 2—Generating an Unformatted Display of Payments

We next implement function displayWelcome, and develop an initial implementation of function displayPayments,given in Figure 5-19. We remove the two print instructions that were included only for test purposes in stage 1 of the program (previously on **lines 27** and **30**).

```
.... Entering function display welcome

Enter the balance on your credit card: 1500
Enter the interest rate (APR) on the card: 18
Use the minimum monthly payment? (y/n): n
User-entered monthly payments selected
Enter monthly payment: 100

.... Entering function displayPayments
parameter balance = 1500
parameter int_rate = 0.015
parameter monthly_payment = 100
>>>
```

```
.... Entering function display welcome

Enter the balance on your credit card: 1500
Enter the interest rate (APR) on the card: 18
Use the minimum monthly payment? (y/n): y
Minimum payment selected

.... Entering function displayPayments
parameter balance = 1500
parameter int_rate = 0.015
parameter monthly_payment = 20
>>>
```

FIGURE 5-18 Output of First Stage Testing

Also, the minimum required monthly payment is computed ( **lines 45–48**) rather than being set to 20.

Function displayPayments is where most of the work is done in the program. Therefore, we shall develop this function in stages as well. At this point, we develop the function to display, for each month during the loan payoff, the year, the current balance, and the total interest paid to date. We delay issues of screen formatting for the alignment of numbers, and only include formatting for

rounding numeric values to two decimal places.

The while loop on **line 21** iterates while balance, passed as an argument to the function, is greater than zero. The function will keep count of the number of months (lines) displayed, as well as the total interest paid. Variables num_months and total_int_paid are used for this purpose, and are therefore initialized before the loop to 0 ( **lines 11–12**). On **lines 15–18** the initial information for the calculation is displayed. Within the while loop, on **line 22**, the monthly interest paid (monthly_int) is computed as the current balance of that month during the payoff period (balance), times the monthly interest rate (int_rate). The total interest paid is then updated on **line 23**. On **line 24**, the new balance is computed as the current balance, plus the interest for the month, minus the monthly payment.

The next step is to display these computed values. Since time is kept track of in terms of months, the current year to be displayed is computed using integer division ( **line 26**), adding one so that the first year is displayed as 1, and not 0. Then on **line 27**, the line representing the payment for the current month is displayed. Formatting is used so that all numerical values are displayed with two decimal places. Finally, variable num_months is incremented by one for the next iteration of the loop.

Stage 2 Testing

We test this program once for a specified monthly payment amount, and once for the option of minimum monthly payments. The results are given in Figures 5-20 and 5-21.

```
# Credit Card Calculation Program (Stage 2)

def displayWelcome():
    print('This program will determine the time to pay off a credit')
    print('card and the interest paid based on the current balance,')
    print('the interest rate, and the monthly payments made.')

def displayPayments(balance, int_rate, monthly_payment):

    # init
    num_months = 0
    total_int_paid = 0

    # display loan info
    print('PAYOFF SCHEDULE')
    print('\nCredit card balance: $' + format(balance,'.2f'))
    print('Annual interest rate:', str(1200 * int_rate)+'%')
    print('Monthly payment: $', format(monthly_payment,'.2f'))

    # display year-by-year account status
    while balance > 0:
        monthly_int = balance * int_rate
        total_int_paid = total_int_paid + monthly_int
        balance = balance + monthly_int - monthly_payment

        year = (num_months // 12) + 1
        print(year, format(balance,'.2f'), format(total_int_paid,'.2f'))

        num_months = num_months + 1

# ---- main

# display welcome screen
displayWelcome()

# determine current balance and APR
balance = int(input('\nEnter the balance on your credit card: '))
apr = int(input('Enter the interest rate (APR) on the card: '))

monthly_int_rate = apr/1200

# determine monthly payment
response = input('Use the minimum monthly payment? (y/n): ')

if response in ('y','Y'):
    if balance < 1000:
        monthly_payment = 20
    else:
        monthly_payment = balance * .02
else:
    monthly_payment = input('Enter monthly payment: ')

# display monthly payoff
displayPayments(balance, monthly_int_rate, monthly_payment)
```

FIGURE 5-19 Credit Card Calculation Program (Stage 2)

```
This program will determine the time to pay off a credit
card and the interest paid based on the current balance,
the interest rate, and the monthly payments made.

Enter the balance on your credit card: 1500
Enter the annual interest rate (APR) on the card: 18
Use the minimum monthly payment? (y/n): n
Enter monthly payment: 100
PAYOFF SCHEDULE

Credit card balance: $1500.00
Annual interest rate: 18.0%
Traceback (most recent call last):
  File "C:\My Python Programs\CreditCardCalc-Stage2 with
error.py", line 53, in <module>
    displayPayments(balance, monthly_int_rate, monthly_payment)
  File "C:\My Python Programs\CreditCardCalc-Stage2 with
error.py", line 18, in displayPayments
    print('Monthly payment: $', format(monthly_payment,'.2f'))
ValueError: Unknown format code 'f' for object of type 'str'
>>>
```

FIGURE 5-20 Output of Second Stage Testing (User-Entered Payment)

```
This program will determine the time to pay off a credit
card and the interest paid based on the current balance,
the interest rate, and the monthly payments made.

Enter the balance on your credit card: 350
Enter the annual interest rate (APR) on the card: 18
Use the minimum monthly payment? (y/n): y
PAYOFF SCHEDULE

Credit card balance: $350.00
Annual interest rate: 18.0%
Monthly payment: $ 20.00
1 335.25 5.25
1 320.28 10.28
1 305.08 15.08
1 289.66 19.66
1 274.00 24.00
1 258.11 28.11
1 241.99 31.99
1 225.62 35.62
1 209.00 39.00
1 192.13 42.13
1 175.02 45.02
1 157.64 47.64
2 140.01 50.01
2 122.11 52.11
2 103.94 53.94
2 85.50 55.50
2 66.78 56.78
2 47.78 57.78
2 28.50 58.50
2 8.93 58.93
2 -10.94 59.06
>>>
```

FIGURE 5-21 Output of Second Stage Testing (Minimum Payment)

Clearly, there is something wrong with this version of the program. The ValueError generated in Figure 5-20 indicates that the format specifier .2f is an unknown format code for a string type value, referring to **line 18**. Thus, this must be referring to variable monthly_payment. But that should be a numeric value, and not a string value! How could it have become a string type? Let's check if the problem also occurs when selecting the minimum payment option (Figure 5-21).

In this case the program works. Since the problem only occurred when the user entered the monthly payment (as opposed to the minimum payment option), we

next try to determine what differences there are in the program related to the assignment of variable monthly_payment.

```
# determine monthly payment
response 5 input('Use the minimum monthly payment? (y/n): ') if response in
('y', 'Y'):

if balance < 1000:
monthly_payment 5 20
else:
monthly_payment = balance * .02
else:
monthly_payment = input('Enter monthly payment: ')
```

When the user selects the minimum monthly payment option, variable monthly_payment is set to integer value 20 (or 2% of the current balance if balance is greater than 1000). Otherwise, its value is input from the user. This variable is not redefined anywhere else in the program. Since the variable monthly_payment is not a local variable, we can display its value directly from the Python shell,

```
... monthly_payment '140'
```

It *is* a string value. We immediately realize that the input value for variable monthly_payment was not converted to an integer type, and was thus left as a string type! We fix this problem by replacing the line with the following,

```
monthly_payment 5 int(input('Enter monthly payment: ')
```

This explains why the problem did not appear in the testing of stage 1 of the program. In that version, variable monthly_payment was never formatted as a numeric value, and also never used in a numerical calculation (both of which would have generated an error).

At this point, we execute a number of test cases for various initial balances, interest rates, and monthly payments. The result is given in Figure 5-22, checked against online loan payoff calculator tools. We next move on to the final stage of program development.

Stage 3—Formatting the Displayed Output

In this final stage of the program, input error checking is added. The program is also modified to allow the user to continue to enter various monthly payments for recalculating a given balance payoff. Output formatting is added to make the displayed information more readable. Finally, we correct the display of a negative balance at the end of the payoff schedule, as appears in Figure 5-21. The final version of the program is given in Figure 5-23.

| **Payoff Information** | | | **Expected Results** | | **Actual Results** | | **Evaluation** |
|---|---|---|---|---|---|---|---|
| Balance | Interest Rate | Monthly Payment | Num Months | Interest Paid | Num Months | Interest Paid | |
| 250 | 18% | $20 (min) | 14 | 28.93 | 14 | 28.93 | Passed |
| 600 | 14% | $20 (min) | 38 | 142.80 | 38 | 142.80 | Passed |
| 12,000 | 20% | $240 (min) | 109 | 14,016.23 | 109 | 14,016.23 | Passed |
| 250 | 18% | $40 | 7 | 14.54 | 7 | 14.54 | Passed |
| 600 | 14% | $50 | 14 | 50.15 | 14 | 50.15 | Passed |
| 12,000 | 20% | $400 | 42 | 4,773.98 | 42 | 4,773.98 | Passed |

FIGURE 5-22 Test Cases for Stage 2 of the Credit Card Calculation Program

```
# Credit Card Calculation Program (Final Version)

def displayWelcome():
    print('This program will determine the time to pay off a credit')
    print('card and the interest paid based on the current balance,')
    print('the interest rate, and the monthly payments made.')

def displayPayments(balance, int_rate, monthly_payment):

    # init
    num_months = 0
    total_int_paid = 0
    payment_num = 1

    empty_year_field = format(' ', '8')

    # display heading
    print('\n', format('PAYOFF SCHEDULE','>20'))
    print(format('Year','>10') + format('Balance','>10') +
          format('Payment Num', '>14') + format('Interest Paid','>16'))

    # display year-by-year account status
    while balance > 0:
        monthly_int = balance * int_rate
        total_int_paid = total_int_paid + monthly_int

        balance = balance + monthly_int - monthly_payment

        if balance < 0:
            balance = 0

        if num_months % 12 == 0:
            year_field = format(num_months // 12 + 1, '>8')
        else:
            year_field = empty_year_field

        print(year_field + format(balance, '>12,.2f') +
              format(payment_num, '>9') +
              format(total_int_paid, '>17,.2f'))

        payment_num = payment_num + 1
        num_months = num_months + 1

```

```
# ---- main

# display welcome screen
displayWelcome()

# get current balance and APR
balance = int(input('\nEnter the balance on your credit card: '))
apr = int(input('Enter the interest rate (APR) on the card: '))

monthly_int_rate = apr/1200

yes_response = ('y','Y')
no_response = ('n','N')

calc = True
while calc:

    # calc minimum monthly payment
    if balance < 1000:
        min_monthly_payment = 20
    else:
        min_monthly_payment = balance * .02

    # get monthly payment
    print('\nAssuming a minimum payment of 2% of the balance ($20 min)')
    print('Your minimum payment would be',
          format(min_monthly_payment, '.2f'),'\n')

    response = input('Use the minimum monthly payment? (y/n): ')
    while response not in yes_response + no_response:
        response = input('Use the minimum monthly payment? (y/n): ')

    if response in yes_response:
        monthly_payment = min_monthly_payment
    else:
        acceptable_payment = False

        while not acceptable_payment:
            monthly_payment = int(input('\nEnter monthly payment: '))

            if monthly_payment < balance * .02:
                print('Minimum payment of 2% of balance required ($' +
                      str(balance * .02) + ')')

            elif monthly_payment < 20:
                print('Minimum payment of $20 required')
            else:
                acceptable_payment = True

```

```
93      # check if single payment pays off balance
94      if monthly_payment >= balance:
95          print('* This payment amount would pay off your balance *')
96      else:
97          # display month-by-month balance payoff
98          displayPayments(balance, monthly_int_rate, monthly_payment)
99
100         # calculate again with another monthly payment?
101         again = input('\nRecalculate with another payment? (y/n): ')
102         while again not in yes_response + no_response:
103             again = input('Recalculate with another payment? (y/n): ')
104
105         if again in yes_response:
106             calc = True    # continue program
107             print('\n\nFor your current balance of $' + str(balance))
108         else:
109             calc = False   # terminate program
```

FIGURE 5-23 Final Stage of the Credit Card Calculation Program

The first set of changes in the program provides some input error checking. (We will address means of more complete error checking in Chapter 7.) In **lines 55–56**, tuples yes_response and no_ response are defined. These are used to check if input from the user is an appropriate yes/no response. For example, the while statement on **line 73** checks that the input from **line 72** is either 'y', 'Y' ,'n', or 'N',

while response not in yes_response 1 no_response:
by checking if response is in the concatenation of tuples yes_response and no_response. For determining the specific response, the tuples can be used individually ( **line 76**),
if response in yes_response:
Similar input error checking is done on **line 101**.

The next set of changes allows a number of payoff schedules for an entered balance to be calculated. A while statement is added at **line 59**, with its condition based on the value of Boolean variable calc (initialized to True on **line 58**). To accommodate the recalculation of payoff schedules, variables num_months, total_int_paid and payment_num are each reset to 0 in function displayPayments ( **lines 11–13**).

Output formatting is added in function displayPayments. On **line 18**, 'PAYOFF SCHEDULE' is displayed right-justified within a field of twenty. On **lines 19–20**, the column headings are displayed with appropriate field widths. **Lines 37–39** display the balance, payment number and interest of each month, aligned

under the column headings. **Lines 32–35** ensure that each year is displayed only once. Finally, in **lines 29–30** variable balance is set to zero if it becomes negative so that negative balances are not displayed.

Stage 3 Testing
We give example output of this version of the program for both a payoff using the required minimum monthly payment, and for a user-entered monthly payment in Figures 5-24 and 5-25.

Figure 5-25 depicts a portion of the output for the sake of space. We run the same set of test cases used in the testing of the previous (stage 2) version of the program, given in Figure 5-26.
Based on these results, we can assume that the program is functioning properly.

```
This program will determine the time to pay off a credit
card and the interest paid based on the current balance,
the interest rate, and the monthly payments made.

Enter the balance on your credit card: 1500
Enter the interest rate (APR) on the card: 18

Assuming a minimum payment of 2% of the balance ($20 min)
Your minimum payment would be 30.00

Use the minimum monthly payment? (y/n): n

Enter monthly payment: 100

      PAYOFF SCHEDULE
      Year    Balance    Payment Num    Interest Paid
       1     1,422.50          1             22.50
             1,343.84          2             43.84
             1,264.00          3             64.00
             1,182.95          4             82.95
             1,100.70          5            100.70
             1,017.21          6            117.21
               932.47          7            132.47
               846.45          8            146.45
               759.15          9            159.15
               670.54         10            170.54
               580.60         11            180.60
               489.31         12            189.31
       2       396.65         13            196.65
               302.60         14            202.60
               207.13         15            207.13
               110.24         16            210.24
                11.89         17            211.89
                 0.00         18            212.07

Recalculate with another payment? (y/n): n
>>>
```

FIGURE 5-24 Output of Third Stage Testing (User-Entered Payment)

```
This program will determine the time to pay off a credit
card and the interest paid based on the current balance,
the interest rate, and the monthly payments made.

Enter the balance on your credit card: 1500
Enter the interest rate (APR) on the card: 18

Assuming a minimum payment of 2% of the balance ($20 min)
Your minimum payment would be 30.00

Use the minimum monthly payment? (y/n): y

      PAYOFF SCHEDULE
      Year    Balance    Payment Num    Interest Paid
       1     1,492.50         1             22.50
             1,484.89         2             44.89
             1,477.16         3             67.16
             1,469.32         4             89.32
                 .            .              .
                 .            .              .
       7       517.51        73           1,207.51
               495.28        74           1,215.28
               472.70        75           1,222.70
               449.79        76           1,229.79
                 .            .              .
                 .            .              .
       8       227.51        85           1,277.51
               200.92        86           1,280.92
               173.94        87           1,283.94
               146.55        88           1,286.55
               118.74        89           1,288.74
                90.53        90           1,290.53
                61.88        91           1,291.88
                32.81        92           1,292.81
                 3.30        93           1,293.30
                 0.00        94           1,293.35

Recalculate with another payment? (y/n): n
>>>
```

FIGURE 5-25 Output of Third Stage Testing (Minimum Payment)

| Payoff Information | | | Expected Results | | Actual Results | | Evaluation |
|---|---|---|---|---|---|---|---|
| Balance | Interest Rate | Monthly Payment | Num Months | Interest Paid | Num Months | Interest Paid | |
| 250 | 18% | $20 (min) | 14 | 28.93 | 14 | 28.93 | Passed |
| 600 | 14% | $20 (min) | 38 | 142.80 | 38 | 142.80 | Passed |
| 12,000 | 20% | $240 (min) | 109 | 14,016.23 | 109 | 14,016.23 | Passed |
| 250 | 18% | $40 | 7 | 14.54 | 7 | 14.54 | Passed |
| 600 | 14% | $50 | 14 | 50.15 | 14 | 50.15 | Passed |
| 12,000 | 20% | $400 | 42 | 4,773.98 | 42 | 4,773.98 | Passed |

FIGURE 5-26 Test Cases for Stage 3 of the Credit Card Calculation Program



CHAPTER SUMMARY
General Topics

Program Routines
Value-Returning vs. Non-Value-Returning Functions Side Effects (of Function Calls)
Parameter Passing: Actual Arguments vs.

Formal Parameters
Local Scope and Local Variables
Global Scope and Global Variables
Variable Lifetime
Python-Specific Programming Topics

Defining Functions in Python
Built-in Functions of Python
Value-Returning and Non-Value-Returning

Functions in Python
Tuple Assignment in Python
Mutable vs. Immutable Arguments in Python Local vs. Global Variables in Python

CHAPTER EXERCISES Section 5.1

**1.** Function avg returns the average of three values, as given in the chapter.

Which of the following statements, each making calls to function avg, are valid? (Assume that all variables are of numeric type.) **(a)** result 5 avg(n1, n2)
**(b)** result 5 avg(n1, n2, avg(n3, n4, n5))
**(c)** result 5 avg(n1 1 n2, n3 1 n4, n5 1 n6)
**(d)**print(avg(n1, n2, n3))
**(e)** avg(n1, n2, n3)

**2.** Which of the following statements, each involving calls to function displayWelcome displaying a welcome message on the screen as given in the chapter, are valid?
**(a)** print(displayWelcome)
**(b)** displayWelcome
**(c)** result 5 displayWelcome()
**(d)** displayWelcome()

Section 5.2
**3.** Suppose there are nine variables, each holding an integer value as shown below, for which the average of the largest value in each line of variables is to be computed.

num1 5 10 num2 5 20 num3 5 25 max1 5 25
num4 5 5 num5 5 15 num6 5 35 max2 5 35
num7 5 20 num8 5 30 num9 5 25 max3 5 30
average 5 (max1 1 max2 1 max3) / 3.0
5 (25 1 35 1 30) / 3.0
5 30.0

Using functions avg and max, give an expression that computes the average as shown above. **4.** Assume that there exists a Boolean function named isLeapYear that determines if a given year is a
leap year or not. Give an appropriate if statement that prints “Year is a Leap Year” if the year passed is a
leap year, and “Year is Not a Leap Year” otherwise, for variable year.

Python Programming Exercises 203
**5.** For the following function definition and associated function calls,

def somefunction(n1, n2): .
.

# main
num1 5 10
somefunction(num1, 15)

**(a)** List all the formal parameters.
**(b)** List all the actual arguments.
**6.** For the following function, indicate whether each function call is proper or not. If improper, explain why.

def gcd(n1, n2): function gcd calculates the greatest common divisor of n1 and n2, with the requirement that n1 be less than or equal to n2, and n1 and n2 are integer values.

**(a)** a 5 10
b 5 20
result 5 gcd(a, b)

**(b)** a 5 10.0
b 5 20
result 5 gcd(a, b)

**(c)** a 5 20
b 5 10
result 5 gcd(b, a)
**(d)** a 5 10
b 5 20
c 5 30
result 5 gcd(gcd(a, b), c) **(e)** a 5 10
b 5 20
c 5 30
print(gcd(a, gcd(c, b)))

PYTHON PROGRAMMING EXERCISES

**P1.** Write a Python function named zeroCheck that is given three integers, and returns true if any of the integers is 0, otherwise it returns false.
**P2.** Write a Python function named ordered3 that is passed three integers, and returns true if the three integers are in order from smallest to largest, otherwise it returns false.
**P3.** Write a Python function named modCount that is given a positive integer, n,

and a second positive integer, m <= n, and returns how many numbers between 1 and n are evenly divisible by m.
**P4.** Write a Python function named helloWorld that displays "Hello World, my name is *name*", for any given name passed to the routine.
**P5.** Write a Python function named printAsterisks that is passed a positive integer value n, and prints out a line of n asterisks. If n is greater than 75, then only 75 asterisks should be displayed.
**P6.** Write a Python function named getContinue that displays to the user “Do you want to continue (y/n): ”, and continues to prompt the user until either uppercase or lowercase 'y' or 'n' is entered, returning (lowercase) 'y' or 'n' as the function value.
**P7.** Implement a Python function that is passed a list of numeric values and a particular threshold value, and returns the list with all values above the given threshold value set to 0. The list should be altered as a side effect to the function call, and not by function return value.
**P8.** Implement the Python function described in question P7 so that the altered list is returned as a function value, rather than by side effect.

PROGRAM MODIFICATION PROBLEMS

**M1.** Temperature Conversion Program: Adding Kelvin Scale
Modify the Temperature Conversion program in section 5.1.3 so that it allows the user to select temperature conversion to include degrees Kelvin, in addition to degrees Fahrenheit and degrees Celsius. Include input error checking for inappropriate temperature values. (NOTE: Refer to questions M1 and M2 from Chapter 3.)

**M2.** GPA Calculation Program: Accommodating First-Semester Students
Modify the GPA Calculation program in section 5.2.7 so that it asks the student if this is their first semester. If so, the program should only prompt for their current semester grades, and not their cumulative GPA and total earned credits, and display their semester GPA and cumulative GPA accordingly.

**M3.** GPA Calculation Program: Allowing for Plus/Minus Grading
Modify the GPA Calculation program in section 5.2.7 so that it is capable of calculating a GPA for plus/ minus letter grades: A, A2, B 1, B, B2, and so forth.

**M4.** Credit Card Calculation Program: Summarized Output
Modify the Credit Card Calculation program in section 5.3 so that the user is

given the option of either displaying the balance and interest paid month-by-month as currently written, or to simply have the total number of months and the total interest paid without the month-by-month details.

**M5.** Credit Card Calculation Program: Adjustable Minimum Payment
Modify the Credit Card Calculation program in section 5.3 so that the user can enter the percentage from which the minimum monthly payment is calculated. Also modify the program so that this minimum payment percentage is displayed along with the other credit card related information.

**M6.** Credit Card Calculation Program: Recalculation with New Balance
Modify the Credit Card Calculation program in section 5.3 so that the program will allow the user to recalculate a new payoff schedule for a new entered balance.

PROGRAM DEVELOPMENT PROBLEMS

**D1.** Metric Conversion Program
Develop and test a Python program that allows the user to convert between the metric measurements of millimeter, centimeter, meter, kilometer, and inches, feet, yards, and miles. The program should be written so that any one measurement can be converted to the other.

**D2.** GPA Projection Program
Develop and test a Python program that lets the user enter their current cumulative GPA, their total credits earned, and the number of credits they are currently taking. The program should then request from the


user a target cumulative GPA that they wish to achieve, and display the GPA of the current semester needed to achieve it.

**D3.** Tic-Tac-Toe Two-Player Program
Develop and test a Python program that lets two players play tic-tac-toe. Let player 1 be X and player 2 be O. Devise a method for each player to indicate where they wish to place their symbol. The program should terminate if either there is a winner, or if the game results in a tie. The tic-tac-toe board should be displayed after every move as shown below.

X --- X
O O --
X X O

**D4.** Tic-Tac-Toe Automated Play
Develop and test a Python program that plays tic-tac-toe against the user. Develop an appropriate strategy of play and implement it in your program. The program should be designed to allow the user to continue to play new games until they decide to quit. The program should display the total number of wins by the computer versus the player at the start of each new game.

# CHAPTER 6 Objects and Their Use

*In procedural programming, functions are the primary building blocks of program design. In object-oriented programming, objects are the fundamental building blocks in which functions (methods) are a component. We first look at the use of individual software objects in this chapter, and in Chapter 10 look at the use of objects in object-oriented design.*

OBJECTIVES
After reading this chapter and completing the exercises, you will be able to: ♦ Explain the concept of an object
♦ Explain the difference between a reference and dereferenced value ♦ Describe the use of object references
♦ Explain the concept of memory allocation and deallocation ♦ Describe automatic garbage collection
♦ Explain the fundamental features of turtle graphics
♦ Effectively use objects in Python
♦ Develop simple turtle graphics programs in Python

CHAPTER CONTENTS

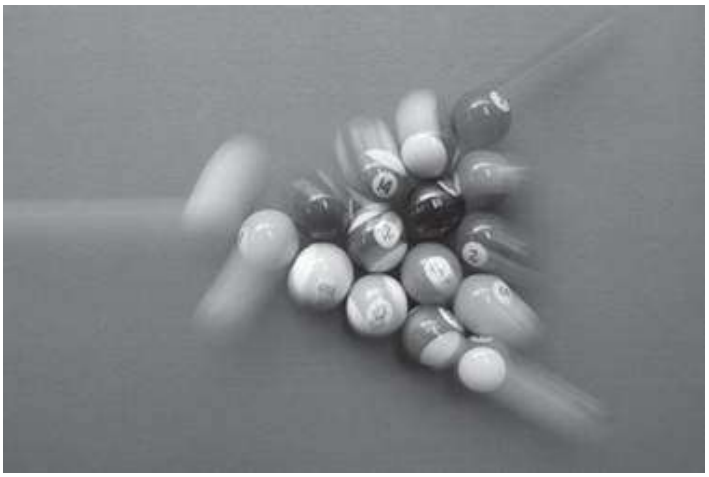


MOTIVATION

An object is one of the first concepts that a baby understands during its development. They understand an object as something that has a set of *attributes* (“big,” “red” ball) and a related set of *behaviors* (it rolls, it bounces).

The idea of incorporating “objects” into a programming language came out of work in computer simulation. Given the prevalence of objects in the world, it was natural to provide the corresponding notion of an object within a simulation program.

In the early 1970s, Alan Kay at Xerox PARC (Palo Alto Research Center) fully evolved the notion of object-oriented programming with the development of a programming language called Smalltalk. The language became the inspiration for the development of graphical user interfaces (GUIs)—the primary means of interacting with computers today. Before that, all interaction was through typed text. In fact, it was a visit to Xerox PARC by Steve Jobs of Apple Computers that led to the development of the first commercially successful GUI-based computer, the Apple Macintosh in 1984. Figure 6-1 lists some of the most commonly used programming languages and whether they support procedural (imperative) programming, object-oriented programming, or both. In this chapter, we look at the creation and use of objects in Python.



| Programming Language | Programming Paradigm Supported | |
|---|---|---|
| | Procedural | Object-oriented |
| C (early 1970s) | X | |
| Smalltalk (1980) | | X |
| C++ (mid 1980s) | X | X |
| Python (early 1990s) | X | X |
| Java (1995) | | X |
| Ruby (mid 1990s) | X | X |
| C # (2000) | X | X |

FIGURE 6-1
Common Programming Languages Supporting Procedural and/or Object-Oriented Programming

FUNDAMENTAL CONCEPTS

## 6.1 Software Objects

Objects are the fundamental component of object-oriented programming. Although we have not yet stated it, *all* values in Python are represented as objects. This includes, for example, lists, as well as numeric values. We discuss object-oriented programming in Chapter 10. In this chapter, we discuss what objects are and how they are used.

### 6.1.1 What Is an Object?

The notion of software objects derives from objects in the real world. All objects have certain *attributes* and *behavior*. The attributes of a car, for example, include its color, number of miles driven, current location, and so on. Its behaviors include driving the car (changing the number of miles driven attribute) and painting the car (changing its color attribute), for example.

Similarly, an **object** contains a set of attributes, stored in a set of **instance variables**, and a set of functions called **methods** that provide its behavior. For example, when sorting a list in procedural programming, there are two distinct entities—a sort function and a list to pass it, as depicted in Figure 6-2.

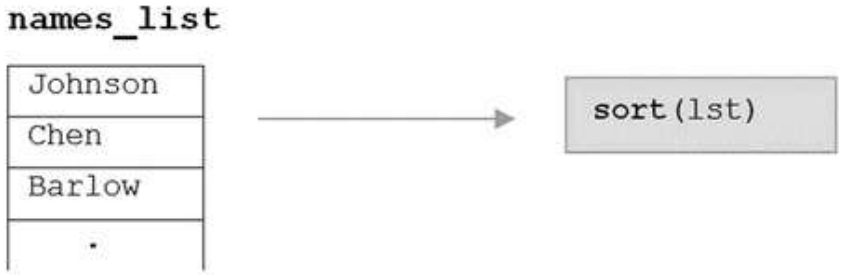



FIGURE 6-2

Procedural Programming Approach

In object-oriented programming, the sort routine would be *part of* the object containing the list, depicted in Figure 6-3.

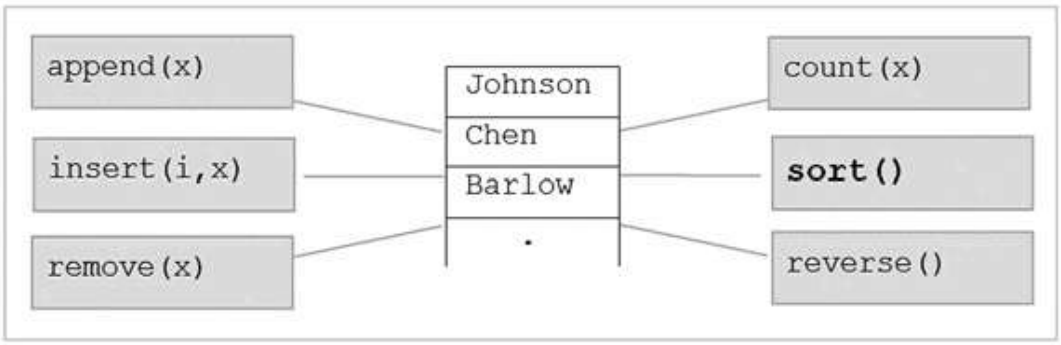



FIGURE 6-3 Object names_list

Here, names_list is an object instance of the Python built-in list type. All list objects contain the same set of methods. Thus, names_list is sorted by simply calling that object's sort method, names_list.sort()

The period is referred to as the *dot operator*, used to select a member of a given object—in this case, the sort method. Note that no arguments are passed to sort. That is because methods operate on the data of the object that they are part of. Thus, the sort method does not need to be told which list to sort.

Suppose there were another list object called part_numbers, containing a list of automobile part numbers. Since all list objects behave the same, part_numbers would contain the identical set of methods as names_list. The data that they would operate on, however, would be different. Thus, two objects of the same type differ only in the particular set of values that each holds. This is depicted in Figure 6-4.

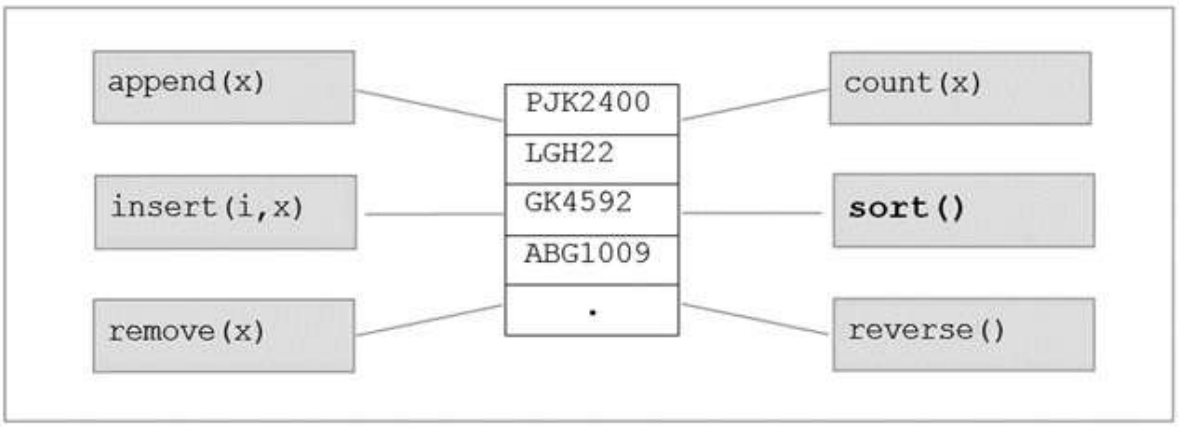


FIGURE 6-4 Object part_numbers

In order to sort *this* list, therefore, the sort method of object part_numbers is called, part_numbers.sort()

The sort routine is the same as the sort routine of object names_list. In this case, however, the list of part numbers is sorted instead. Methods append, insert, remove, count, and reverse also provide additional functionality for lists, as was discussed in Chapter 4. We next discuss the way that objects are represented in Python.

An **object** contains a set of attributes, stored in a set of **instance variables**, and a set of functions called **methods** that provide its behavior.

6.1.2 Object References

In this section we look at how objects are represented (which all values in Python are), and the effect it has on the operations of assignment and comparison, as well as parameter passing. References in Python

In Python, objects are represented as a *reference* to an object in memory, as shown in Figure 6-5.

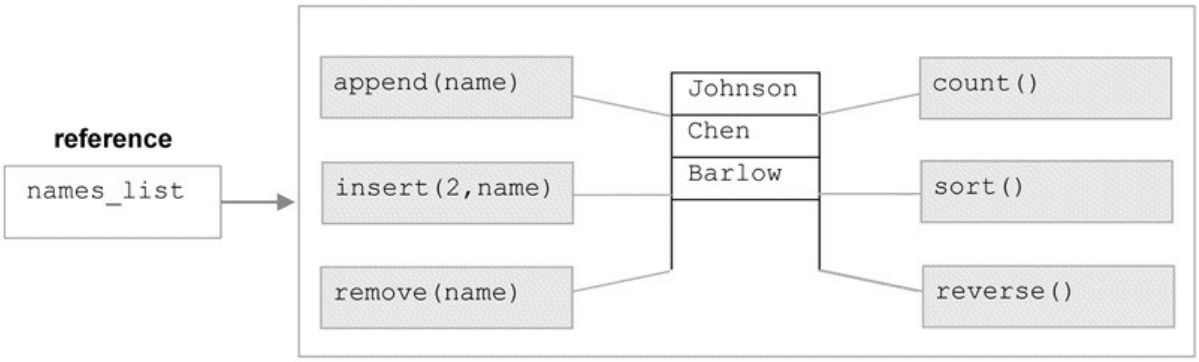


FIGURE 6-5 Object Reference

A **reference** is a value that references, or "points to," the location of another entity. Thus, when a new object in Python is created, *two* entities are stored—the object, and a variable holding a reference to the object. All access to the object is through the reference value. This is depicted in Figure 6-6.

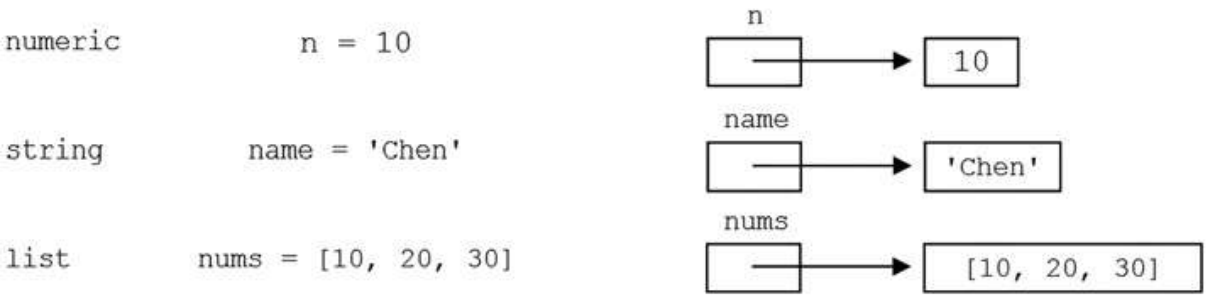



FIGURE 6-6 Object References to Python Values

The value that a reference points to is called the **dereferenced value**. This is the value that the variable represents, as shown in Figure 6-7.

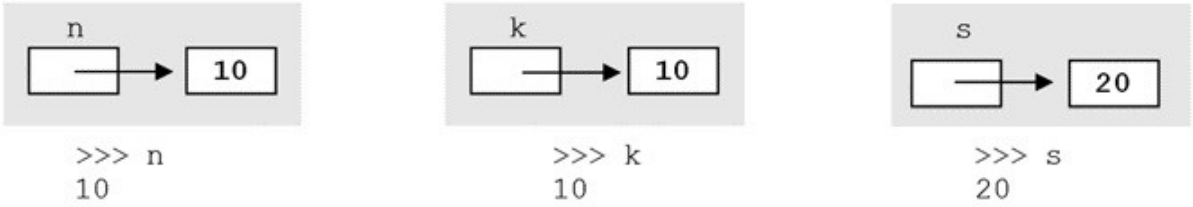



FIGURE 6-7 Variables' Dereferenced Values

We can get the reference value of a variable (that is, the location in which the corresponding object is stored) by use of **built-in function id**.

...id(n) id(k) id(s) 505498136 505498136 505498296

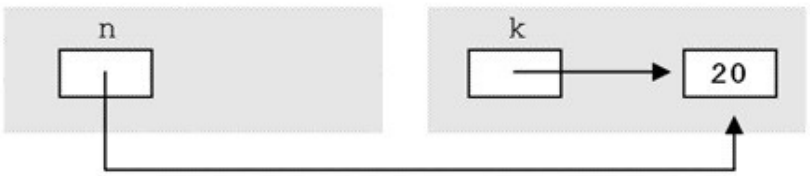
We see that the dereferenced values of n and k, 10, is stored in the same memory location (505498136), whereas the dereferenced value of s, 20,is stored in a different location (505498296). Even though n and k are each separately assigned literal value 10, they reference the *same instance* of 10 in memory (505498136). We would expect there to be separate instances of 10 stored. Python is using a little cleverness here. Since integer values are immutable, it assigned both n and k to the same instance. This saves memory and reduces the number of reference locations that Python must maintain. From the programmer's perspective however, they can be treated as if they are separate instances.

LET'S TRY IT From the Python Shell, first enter the following and observe the results. ... n 5 10 ... n 5 20 ... k 5 20 ... k 5 20

... id(n) ... id(n) ??? ??? ... id(k) ... id(k) ??? ???

A **reference** is a value that references, or “points to,” the location of another entity. The value that a reference points to is called the **dereferenced value**. A variable’s reference value can be determined with **built-in function** id.

The Assignment of References
With our current understanding of references, consider what happens when variable n is assigned to variable k, depicted in Figure 6-8.




```
k = 20
n = k
n
>>> 20
```

FIGURE 6-8
The Assignment of References

When variable n is assigned to k, it is the *reference value*of k that is assigned, not the dereferenced value 20, as shown in Figure 6-8. This can be determined by use of the built-in id function, as demonstrated below.

... id(k) ... id(k) 55 id(n) 505498136 True
... id(n) ... n is k 505498136 True

Thus, to verify that two variables refer to the same object instance, we can either compare the two id values by use of the comparison operator, or make use of the provided is operator (which performs id(k) 55id(n)).

Thus, both n and k reference the same instance of literal value 20. This occurred in the above example when n and k were *separately* assigned 20 because integers are an immutable type, and Python makes attempts to save memory. In this case, however, n and k reference the same instance of 20 because assignment in Python assigns reference values. We must be aware of the fact, therefore, that when assigning variables referencing mutable values, such as lists, both variables reference the same list instance as well. We will discuss the implication of this next.

Finally, we look at what happens when the value of one of the two variables n or k is changed, as depicted in Figure 6-9.

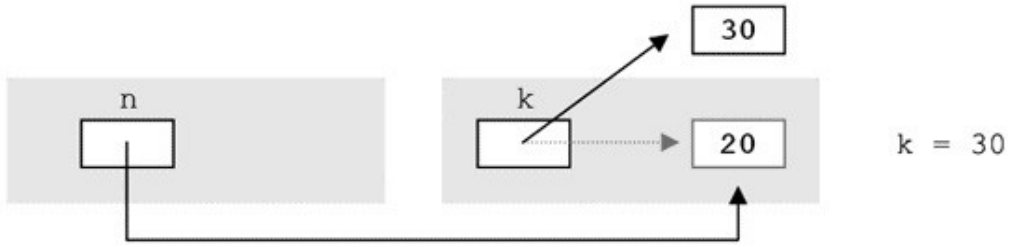



FIGURE 6-9 Reassignment of Reference Value

Here, variable k is assigned a reference value to a *new* memory location holding the value 30. The previous memory location that variable k referenced is retained since variable n is still referencing it. As a result, n and k point to different values, and therefore are no longer equal. LET'S TRY IT

From the Python Shell, first enter the following and observe the results.

... k 5 10 ... k 5 30 ... n 5 k ... id(k) ... id(k) ???
??? ... id(n) ... id(n) ???
??? ... id(k) 5 5 id(n) ... id(k) 5 5 id(n) ???
??? ... n is k ... n is k ???
???

When one variable is assigned to another, it is the *reference value* that is assigned, not the dereferenced value.
Memory Deallocation and Garbage Collection
Next we consider what happens when in addition to variable k being reassigned, variable n is reassigned as well. The result is depicted in Figure 6-10.

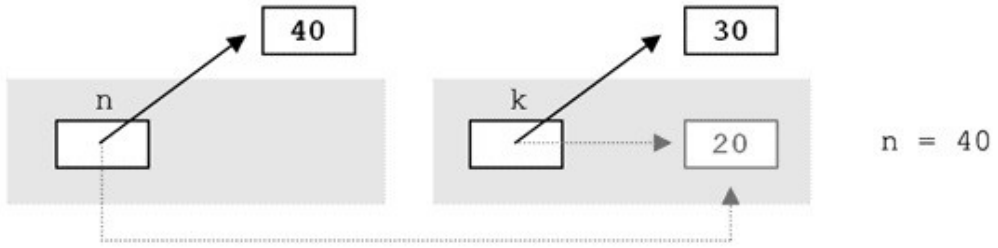



FIGURE 6-10 Inaccessible Values

After n is assigned to 40, the memory location storing integer value 20 is no longer referenced— thus, it can be *deallocated*. To **deallocate** a memory location means to change its status from "currently in use" to "available for reuse." In Python, memory deallocation is automatically performed by a process

called *garbage collection*. **Garbage collection** is a method of automatically determining which locations in memory are no longer in use and deallocating them. The garbage collection process is ongoing during the execution of a Python program.

**Garbage collection** is a method of determining which locations in memory are no longer in use, and deallocating them.
List Assignment and Copying

Now that we understand the use of references in Python, we can revisit the discussion on copying lists from Chapter 4. We know that when a variable is assigned to another variable referencing a list, each variable ends up referring to the *same instance* of the list in memory, depicted in Figure 6-11.

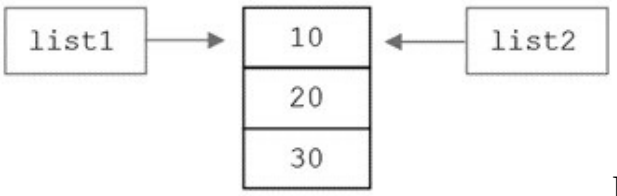



FIGURE 6-11 List Assignment
Thus, any changes to the elements of list1 results in changes to list2,

... list1[0] 5 5
... list2[0]
5

We also learned that a copy of a list can be made as follows,
... list2 5 list(list1)
list() is referred to as a *list constructor*. The result of the copying is depicted in Figure 6-12.


FIGURE 6-12 Copying of Lists by Use of the List Constructor
A copy of the list structure has been made. Therefore, changes to the list elements of list1 will *not* result in changes in list2.

... list1[0] 5 5
... list2[0]
10

The situation is different if a list contains sublists, however.
... list1 5 [[10, 20], [30, 40], [50, 60]] ... list2 5 list(list1)
The resulting list structure after the assignment is depicted in Figure 6-13.

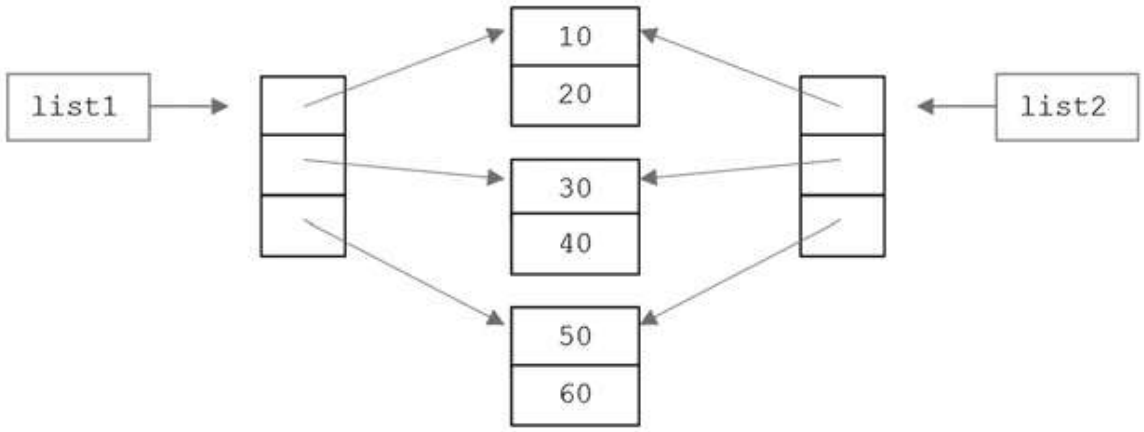



FIGURE 6-13 Shallow Copy List Structures

We see that although copies were made of the top-level list structures, the elements *within* each list were not copied. This is referred to as a **shallow copy**. Thus, if a top-level element of one list is reassigned, for example list1[0] 5 [70, 80], the other list would remain unchanged, as shown in Figure 6-14.

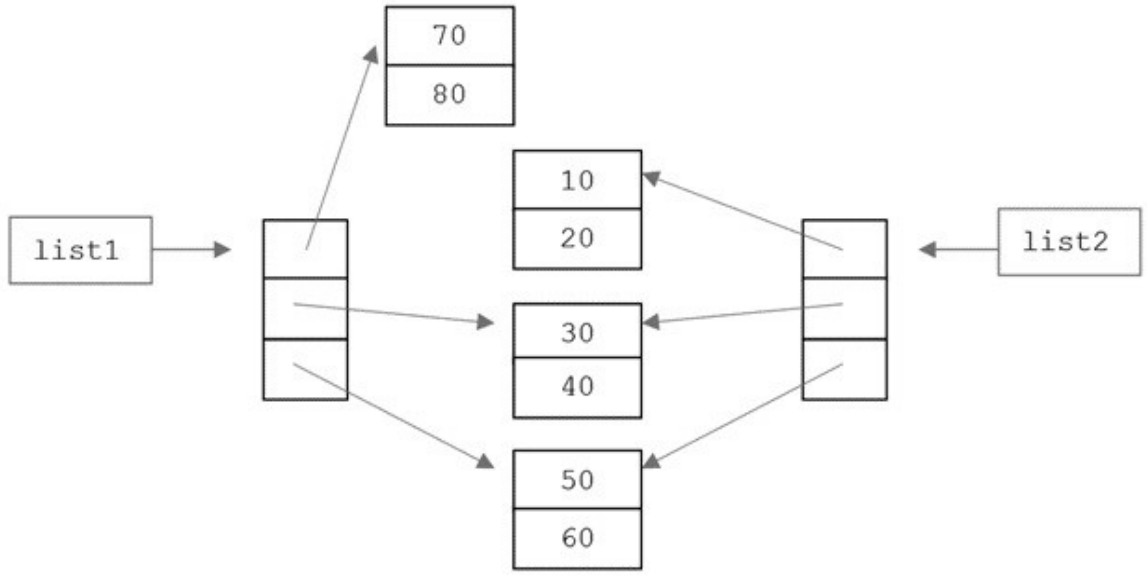



FIGURE 6-14 Top-Level Reassignment of Shallow Copies

If, however, a change to one of the sublists is made, for example, list1[0][0] 5 70, the corresponding change would be made in the other list. That is, list2[0][0] would be equal to 70 also, as depicted in Figure 6-15.

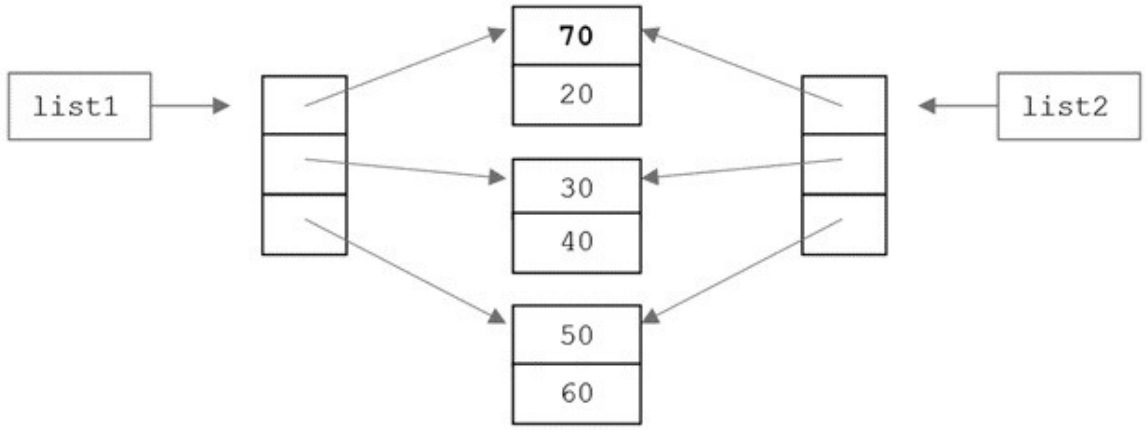



FIGURE 6-15 Sublevel Reassignment of Shallow Copies

A **deep copy** operation of a list (structure) makes a copy of the *complete* structure, including sublists. (Since immutable types cannot be altered, immutable parts of the structure may not be copied.) Such an operation can be performed with the deepcopy method of the copy module,

... import copy
... list2 5 copy.deepcopy(list1)
The result of this form of copying is given in Figure 6-16.

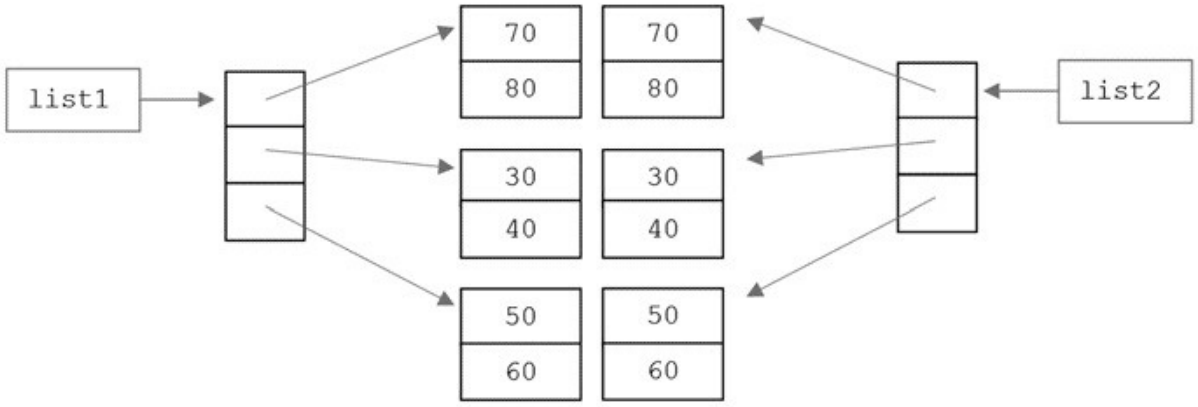



FIGURE 6-16 Deep Copy List Structures

Thus, the reassignment of any part (top level or sublist) of one list will not result in a change in the other. It is up to you as the programmer to determine which form of copy is needed for lists, and other mutable types, such as dictionaries

and sets covered in Chapter 9.

LET’S TRY IT
From the Python Shell, enter the following and observe the results.
... import copy

... list1 5 [10, 20, 30, 40] ... list2 5 list1
... id(list1) 55 id(list2) ???
... list1[0] 5 60
... list1
???
... list2
???

... list1 5 [10, 20, 30, [40]] ... list2 5 list(list1)
... id(list1) 55 id(list2) ???
... list1[0] 5 60
... list1[3][0] 5 90
... list1
???
... list2
???
... list1 5 [10, 20, 30, [40]] ... list2 5 copy.deepcopy(list1) ... id(list1) 55 id(list2)
???
... list1[0] 5 60
... list1[3][0] 5 90
... list1
???
... list2
???

... list1 5 [10, 20, 30, (40)] ... list2 5 copy.deepcopy(list1) ... list1[3][0] 5 90
???
... list1[3] 5 (100,)
... list1
???
... list2
???

The list constructor list() makes a copy of the top level of a list, in which the sublist (lower-level) structures are shared is referred to as a **shallow copy**. A **deep copy** operation makes a complete copy of a list. A deep copy operator is provided by method deepcopy of the copy module in Python.

Self-Test Questions

**1.** All objects have a set of _____ and _____.
**2.** The _____ operator is used to select members of a given object.
**3.** Functions that are part of an object are called _____.

**4.** There are two values associated with every object in Python, the _____ value and the _____ value.

**5.** When memory locations are *deallocated*, it means that,
**(a)** The memory locations are marked as unusable for the rest of the program execution. **(b)** The memory locations are marked as available for reuse during the remaining program

execution.
**6.** Garbage collection is the process of automatically identifying which areas of memory can be deallocated. (TRUE/FALSE)
**7.** Indicate which of the following is true,

**(a)** When one variable is assigned to another holding an integer value, if the second variable is assigned a new value, the value of the first variable will change as well.
**(b)** When one variable is assigned to another holding a list of integer values, if the second variable assigns a new integer value to an element in the list, the list that the first variable is assigned to will be changed as well.

ANSWERS: 1. attributes/behavior, 2. dot, 3. methods, 4. reference/dereferenced, 5. (b), 6. True, 7. (b)

## 6.2 Turtle Graphics

*Turtle graphics* refers to a means of controlling a graphical entity (a “turtle”) in a graphics window with x,y coordinates. A turtle can be told to draw lines as it travels, therefore having the ability to create various graphical designs. Turtle graphics was first developed for a language named Logo in the 1960s for teaching children how to program. Remnants of Logo still exist today.

Python provides the capability of turtle graphics in the turtle Python standard library module. There may be more than one turtle on the screen at once. Each turtle is represented by a distinct object. Thus, each can be individually controlled by the methods available for turtle objects. We introduce turtle graphics here for two reasons—first, to provide a means of better understanding objects in programming, and second, to have some fun!

**Turtle graphics** refers to a means of controlling a graphical entity (a "turtle") in a graphics window with x,y coordinates.

6.2.1 Creating a Turtle Graphics Window

The first step in the use of turtle graphics is the creation of a turtle graphics window (a *turtle screen*). Figure 6-17 shows how to create a turtle screen of a certain size with an appropriate title bar. Assuming that the import turtle form of import is used, each of the turtle graphics methods must be called in the form turtle. *methodname*. The first method called, setup,

```
import turtle

# set window size
turtle.setup(800, 600)

# get reference to turtle window
window = turtle.Screen()

# set window title bar
window.title('My First Turtle Graphics Program')
```

FIGURE 6-17
Creating a Turtle Graphics Window

creates a graphics window of the specified size (in pixels). In this case, a window of size 800 pixels wide by 600 pixels high is created. The center point of the window is at coordinate (0,0). Thus, x-coordinate values to the right of the center point are positive values, and those to the left are negative values. Similarly, y-coordinate values above the center point are positive values, and those below are negative values. The top-left, top-right, bottom-left, and bottom-left coordinates for a window of size (800, 600) are as shown in Figure 6-18. A turtle graphics window in Python is also an object. Therefore, to set the title of this window, we need the reference to this object. This is done by call to method Screen.

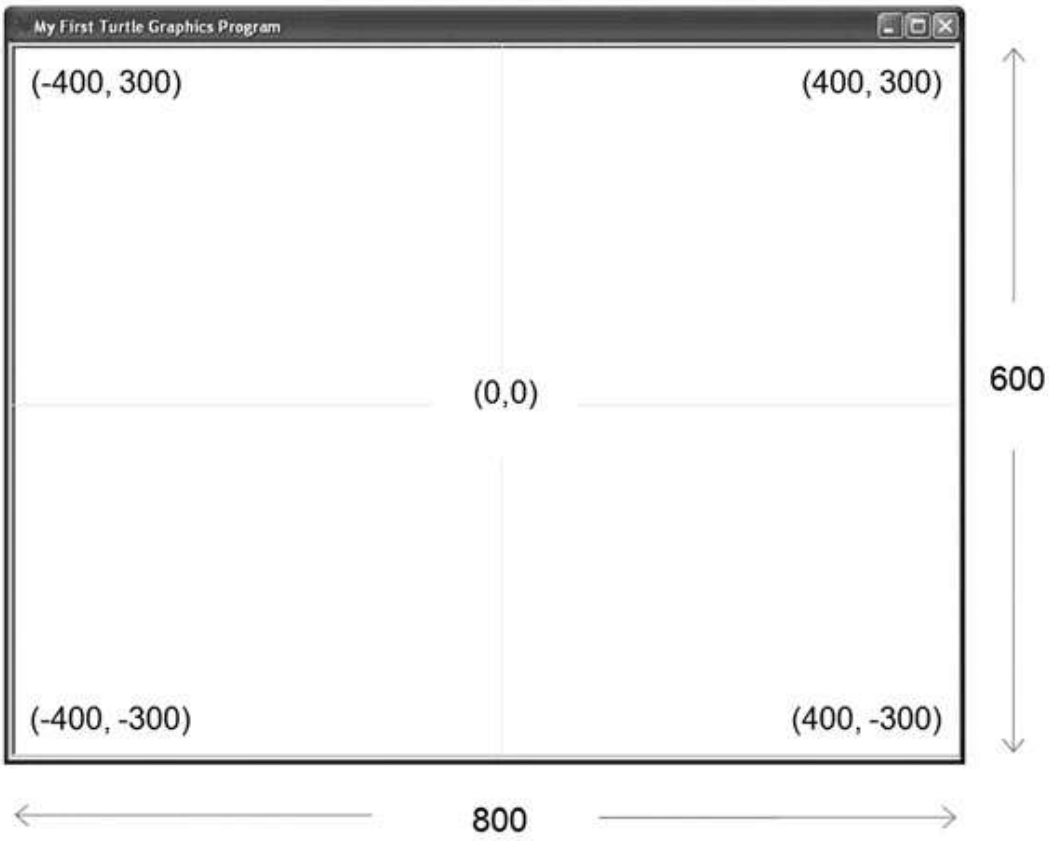



FIGURE 6-18 Python Turtle Graphics Window (of size 800 3 600)

The background color of the turtle window can be changed from the default white background color. This is done using method bgcolor,

window 5 turtle.Screen()
window.bgcolor('blue')

See the discussion about pen color below for details on the specification of color values. The first step in the use of turtle graphics is to create a **turtle graphics window** of a specific size with an appropriate title.

6.2.2 The “Default” Turtle

A “turtle” is an entity in a turtle graphics window that can be controlled in various ways. Like the graphics window, turtles are objects. A “default” turtle is created when the setup method is called. The reference to this turtle object can be obtained by,

the_turtle 5 turtle.getturtle()

A call to getturtle returns the reference to the default turtle and causes it to appear on the screen. The initial position of all turtles is the center of the screen at coordinate (0,0), as shown in Figure 6-19.

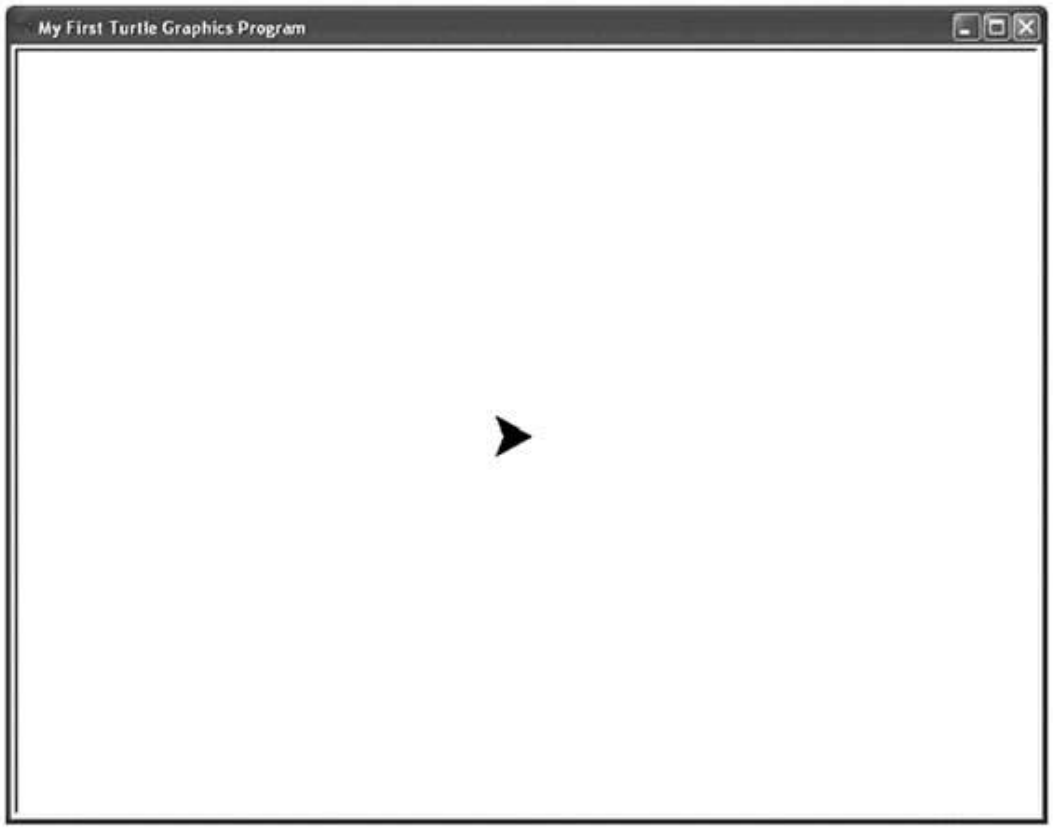



FIGURE 6-19 The Default Turtle

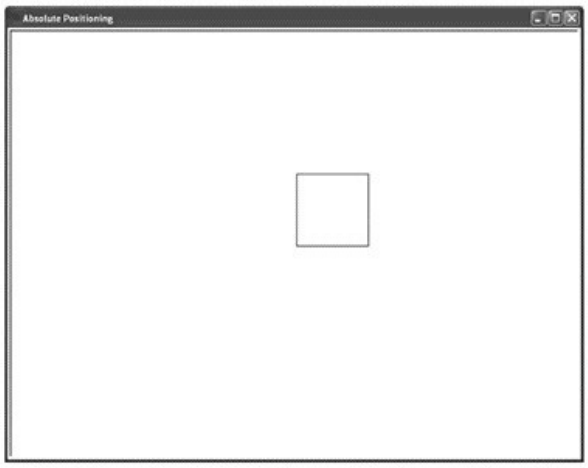
The default turtle shape is an arrowhead. (The size of the turtle shape was enlarged from its default size for clarity.) A turtle's shape can be set to basic geometric shapes, or even made from a provided image file (shown in section 6.2.4).

A **default turtle** is created when the setup method is called. A call to method getturtle returns the reference to the default turtle and causes it to appear on the screen.

6.2.3 Fundamental Turtle Attributes and Behavior

Recall that objects have both attributes and behavior. Turtle objects have three fundamental attributes: *position, heading* (orientation), and *pen* attributes. We discuss each of these attributes next. Absolute Positioning

Method position returns a turtle's current position. For newly created turtles, this returns the tuple (0, 0). A turtle's position can be changed using *absolute positioning* by moving the turtle to a specific x,y coordinate location by use of method setposition. An example of this is given in Figure 6-20.




```
# set window title
window = turtle.Screen()
window.title('Absolute Positioning')

# get default turtle and hide
the_turtle = turtle.getturtle()
the_turtle.hideturtle()

# create square (absolute positioning)
the_turtle.setposition(100, 0)
the_turtle.setposition(100, 100)
the_turtle.setposition(0, 100)
the_turtle.setposition(0, 0)

# exit on close window
turtle.exitonclick()
```

FIGURE 6-20 Absolute Positioning of Turtle

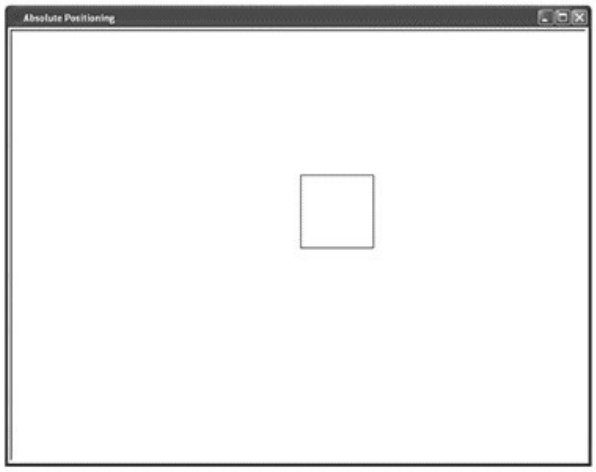
The turtle is made invisible by a call to method hideturtle. Since newly created turtles are positioned at coordinates (0, 0), the square will be displayed near the middle of the turtle window. To draw the square, the turtle is first positioned at coordinates (100, 0), 100 pixels to the right of its current position. Since the turtle's pen is down, a line will be drawn from location (0, 0) to location (100, 0). The turtle is then positioned at coordinates (100, 100), which draws a line from the bottom-right corner to the top-right corner of the square. Positioning the turtle to coordinates (0, 100) draws a line from the top-right corner to the top-left corner. Finally, positioning the turtle back to coordinates (0, 0) draws the final line from the top-left corner to the bottom-left corner.

A turtle's position can be changed using *absolute positioning* by use of method setposition.

Turtle Heading and Relative Positioning

A turtle's position can also be changed through *relative positioning*. In this case, the location that a turtle moves to is determined by its second fundamental attribute, its heading. A newly created turtle's heading is to the right, at 0 degrees. A turtle with heading 90 degrees moves up; with a heading 180 degrees moves left; and with a heading 270 degrees moves down. A turtle's heading can be changed by turning the turtle a given number of degrees left, left(90), or right,

right(90). The forward method moves a turtle in the direction that it is currently heading. An example of relative positioning is given in Figure 6-21.




```
# set window title
window = turtle.Screen()
window.title('Relative Positioning')

# get default turtle and hide
the_turtle = turtle.getturtle()
the_turtle.hideturtle()

# create box (relative positioning)
the_turtle.forward(100)
the_turtle.left(90)
the_turtle.forward(100)
the_turtle.left(90)
the_turtle.forward(100)
the_turtle.left(90)
the_turtle.forward(100)

# exit on close window
turtle.exitonclick()
```

FIGURE 6-21 Relative Positioning of Turtle

In this example, the turtle is controlled using relative positioning, drawing the same square as in Figure 6-20 above. Since turtles are initially positioned at coordinates (0, 0) with an initial heading of 0 degrees, the first step is to move the turtle forward 100 pixels. That draws the bottom line of the square. The turtle is then turned left 90 degrees and again moved forward 100 pixels. This draws the line of the right side of the square. These steps continue until the turtle arrives back at the original coordinates (0, 0), completing the square.

Methods left and right change a turtle's heading relative to its current heading. A turtle's heading can also be set to a specific heading by use of method setheading: the_turtle.setheading(90). In addition, method heading can be used to determine a turtle's current heading.

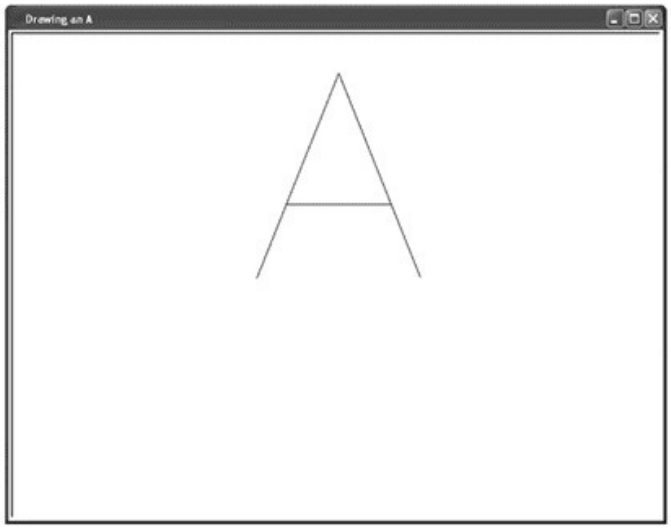
A turtle's position can be changed using *relative positioning* by use of methods setheading, left, right, and forward.

Pen Attributes

The pen attribute of a turtle object is related to its drawing capabilities. The most fundamental of these attributes is whether the pen is currently "up" or "down," controlled by methods penup() and pendown(). When the pen attribute value is "up," the turtle can be moved to another location without lines being drawn. This

is especially needed when drawing graphical images with disconnected segments. Example use of these methods is given in Figure 6-22.

In this example, the turtle is hidden so that only the needed lines appear. Since the initial location of the turtle is at coordinate (0, 0), the pen is set to “up” so that the position of the turtle can be set to (2100, 0) without a line being drawn as it moves. This puts the turtle at the bottom of the left side of the letter. The pen is then set to “down” and the turtle is moved to coordinate (0, 250), drawing as it moves. This therefore draws a line from the bottom of the left side to the top of the “A.” The turtle is then moved (with its pen still down) to the location of the bottom of the right side of the letter, coordinate (100, 0). To cross the “A,” the pen is again set to “up” and the turtle is moved to the location of the left end of the crossing line, coordinate (264, 90). The pen is then set to “down” and moved to the end of the crossing line, at coordinate (64, 90), to finish the letter.




```
turtle.setup(800, 600)
window = turtle.Screen()
window.title('Drawing an A')

the_turtle = turtle.getturtle()
the_turtle.hideturtle()
the_turtle.penup()
the_turtle.setposition(-100, 0)
the_turtle.pendown()
the_turtle.setposition(0, 250)
the_turtle.setposition(100, 0)
the_turtle.penup()
the_turtle.setposition(-64, 90)
the_turtle.pendown()
the_turtle.setposition(64, 90)

turtle.exitonclick()
```

FIGURE 6-22 Example Use of Methods penup and pendown

The pen size of a turtle determines the width of the lines drawn when the pen attribute is “down.” The pensize method is used to control this: the_turtle.pensize(5). The width is given in pixels, and is limited only by the size of the turtle screen. Example pen sizes are depicted in Figure 6-23.

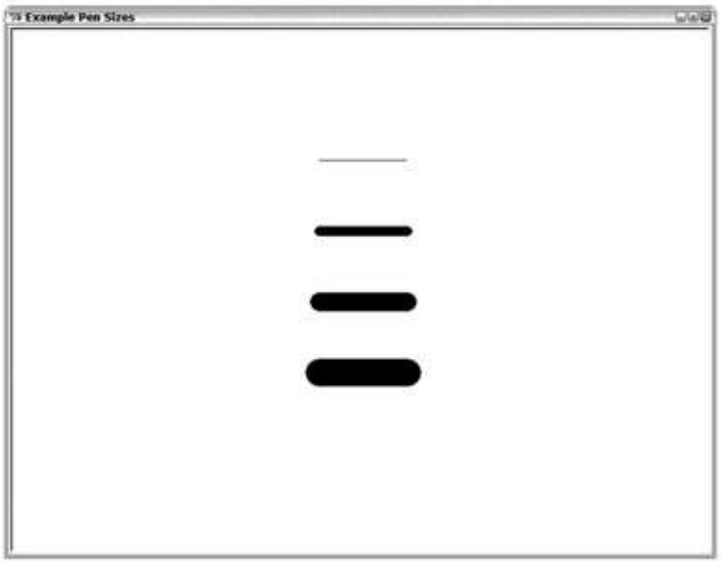



FIGURE 6-23 Example Turtle Pen Sizes

The pen color can also be selected by use of the pencolor method: the_turtle. pencolor('blue'). The name of any common color can be used, for example 'white', 'red', 'blue', 'green', 'yellow', 'gray', and 'black'. Colors can also be specified in RGB (red/green/blue) component values. These values can be specified in the range 0–255 if the color mode attribute of the turtle window is set as given below,

```
turtle.colormode(255)
the_turtle.pencolor(238, 130, 238) # violet
```

This provides a means for a full spectrum of colors to be displayed.
The pen attributes that can be controlled include whether the pen is down or up (using methods penup and pendown), the pen size (using method pensize), and the pen color (using method pencolor).

### 6.2.4 Additional Turtle Attributes

In addition to the fundamental turtle attributes already discussed, we provide details on other attributes of a turtle that may be controlled. This includes whether the turtle is visible or not, the size (both demonstrated above), shape, and fill color of the turtle, the turtle's speed, and the tilt of the turtle. We will discuss each of these attributes next.

#### Turtle Visibility

As we saw, a turtle's visibility can be controlled by use of methods hideturtle() and showturtle() (in which an invisible turtle can still draw on the screen). There are various reasons for doing this. A turtle may be made invisible while being repositioned on the screen. In gaming, a turtle might be made invisible when it meets its "demise." Or maybe a given turtle needs to blink, as we will see at the end of the chapter.

Methods showturtle() and hideturtle()control a turtle's visibility.

Turtle Size

The size of a turtle shape can be controlled with methods resizemode and turtlesize as shown in Figure 6-24.

```
# set to allow user to change turtle size
the_turtle.resizemode('user')

# set a new turtle size
the_turtle.turtlesize(3, 3)
```

FIGURE 6-24 Changing the Size of a Turtle

The first instruction sets the resize attribute of a turtle to 'user'. This allows the user (programmer) to change the size of the turtle by use of method turtlesize. Otherwise, calls to turtlesize will have no effect. The call to method turtlesize in the figure is passed two parameters. The first is used to change the *width* of the shape (perpendicular to its orientation), and the second changes its *length* (parallel to its orientation). Each value provides a factor by which the size is to be changed. Thus, the_turtle.turtlesize(3, 3)stretches both the width and length of the current turtle shape by a factor of 3. (A third parameter can also be added that determines the thickness of the shape's outline.)

There are two other values that method resizemode may be set to. An argument value of 'auto' causes the size of the turtle to change with changes in the pen size, whereas a value of 'noresize' causes the turtle shape to remain the same size.

The size of a given turtle shape can be controlled with methods resizemode and turtlesize.

Turtle Shape

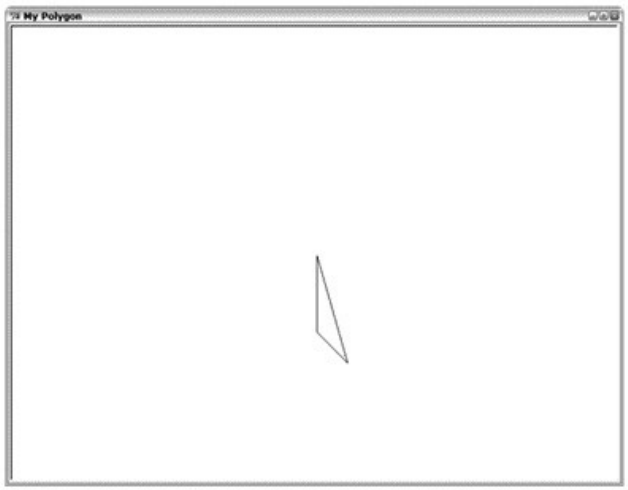
There are a number of ways that a turtle's shape (and fill color) may be defined to something other than the default shape (the arrowhead) and fill color (black). First, a turtle may be assigned one of the following provided shapes: 'arrow', 'turtle', 'circle', 'square', 'triangle', and 'classic' (the default arrowhead shape), as

shown in Figure 6-25.


FIGURE 6-25 Available Turtle Shapes

The shape and fill colors are set by use of the shape and fillcolor methods, the_turtle.shape('circle') the_turtle.fillcolor('white')

New shapes may be created and registered with (added to) the turtle screen's *shape dictionary*. One way of creating a new is shape by providing a set of coordinates denoting a polygon, as shown in Figure 6-26.



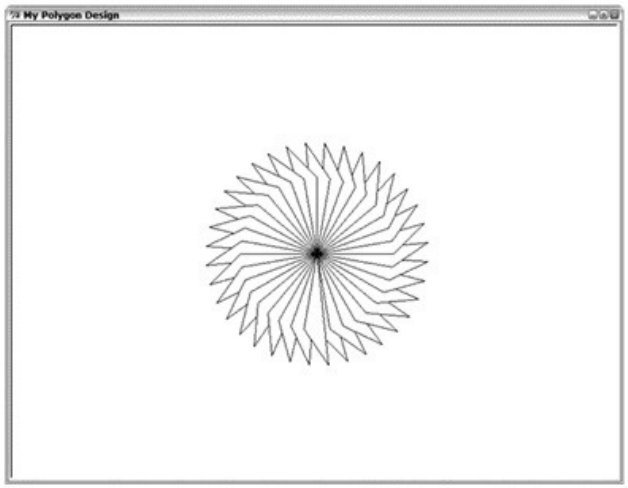

```
turtle.setup(800, 600)
window = turtle.Screen()
window.title('My Polygon')
the_turtle = turtle.getturtle()

turtle.register_shape('mypolygon',
((0, 0), (100, 0), (140, 40)))

the_turtle.shape('mypolygon')
the_turtle.fillcolor('white')
```

FIGURE 6-26 Creating a New Polygon Turtle Shape

In the figure, method register_shape is used to register the new turtle shape with the name mypolygon. The new shape is provided by the tuple of coordinates in the second argument. These coordinates define the polygon shown in the figure. Once the new shape is defined, a turtle can be set to that shape by calling the shape method with the desired shape's name. The fillcolor method is then called to make the fill color of the polygon white (with the edges remaining black). It is also possible to create turtle shapes composed of various individual polygons called *compound shapes*. We refer the reader to the official online Python documentation of the turtle module for details (see http://docs.python.org/py3k/library/turtle. html#module-turtle).

The creation of this polygon may not seem too exciting, but the orientation of a turtle can be changed. In addition, a turtle is able to *stamp* its shape on the screen, which remains there even after the turtle is repositioned (or relocated).

That means that we can create all sorts interesting graphic patterns by appropriately repositioning the turtle, as shown in Figure 6-27.




```
turtle.setup(800, 600)
window = turtle.Screen()
window.title('My Polygon Design')
the_turtle = turtle.getturtle()

turtle.register_shape('mypolygon',
((0, 0), (100, 0), (140, 40)))
the_turtle.shape('mypolygon')
the_turtle.fillcolor('white')

for angle in range(0, 360, 10):
    the_turtle.setheading(angle)
    the_turtle.stamp()
```

FIGURE 6-27 Creating a Design from a Turtle using a Polygon Shape

Only a few lines of code are needed to generate this design. The for loop in the figure iterates variable angle over the complete range of degrees, 0 to 360, by increments of 10 degrees. Within the loop the turtle's heading is set to the current angle, and the stamp() method is called to stamp the polygon shape at the turtle's current position. By varying the shape of the polygon and the angles that the turtle is set to, a wide range of such designs may be produced.

Another way that a turtle shape can be created is by use of an image. The image file used must be a "gif file" (with file extension .gif). The name of the file is then registered and the shape of the turtle set to the registered name,

register_shape('image1.gif') the_turtle.shape('image1.gif')

The final program of this chapter gives an example of the use of image shapes. A turtle's shape may be set to one of the provided shapes, a described polygon (or collection of polygons), or an image.

Turtle Speed

At times, you may want to control the speed at which a turtle moves. A turtle's speed can be set to a range of speed values from 0 to 10, with a "normal" speed being around 6. To set the speed of the turtle, the speed method is used, the_turtle.speed(6). The following speed values can be set using a descriptive

rather than a numeric value,

10: 'fast' 6: 'normal' 3: 'slow' 1: 'slowest' 0: 'fastest'

Thus, a normal speed can also be set by the_turtle.speed('normal'). When using the turtle for line drawing only, the turtle will move more quickly if it is made invisible (by use of the hideturtle method).

The speed of a turtle can be controlled by use of the speed method.

### 6.2.5 Creating Multiple Turtles

So far, we have seen examples in which there is only one turtle object, the default turtle created with a turtle window. However, it is possible to create and control any number of turtle objects. To create a new turtle, the Turtle() method is used,

turtle1 5 turtle.Turtle() turtle2 5 turtle.Turtle() etc.

By storing turtle objects in a list, any number of turtles may be maintained,

turtles 5 []
turtles.append(turtle.Turtle()) turtles.append(turtle.Turtle()) etc.

An example of using multiple turtle objects is given in the following “Let’s Apply It” section.
Any number of turtle objects can be created by use of method Turtle().

### 6.2.6 Let’s Apply It—Bouncing Balls Program

Following is a program (Figure 6-29) that displays one or more bouncing balls within a turtle screen. This program utilizes the following programming features.
➤ turtle module ➤ time module ➤ random module
Example execution of the program is given in Figure 6-28.

Program Execution ...

```
This program simulates one or more bouncing balls on a turtle screen.
Enter number of seconds to run: 30
Enter number of balls in simulation: 5
```

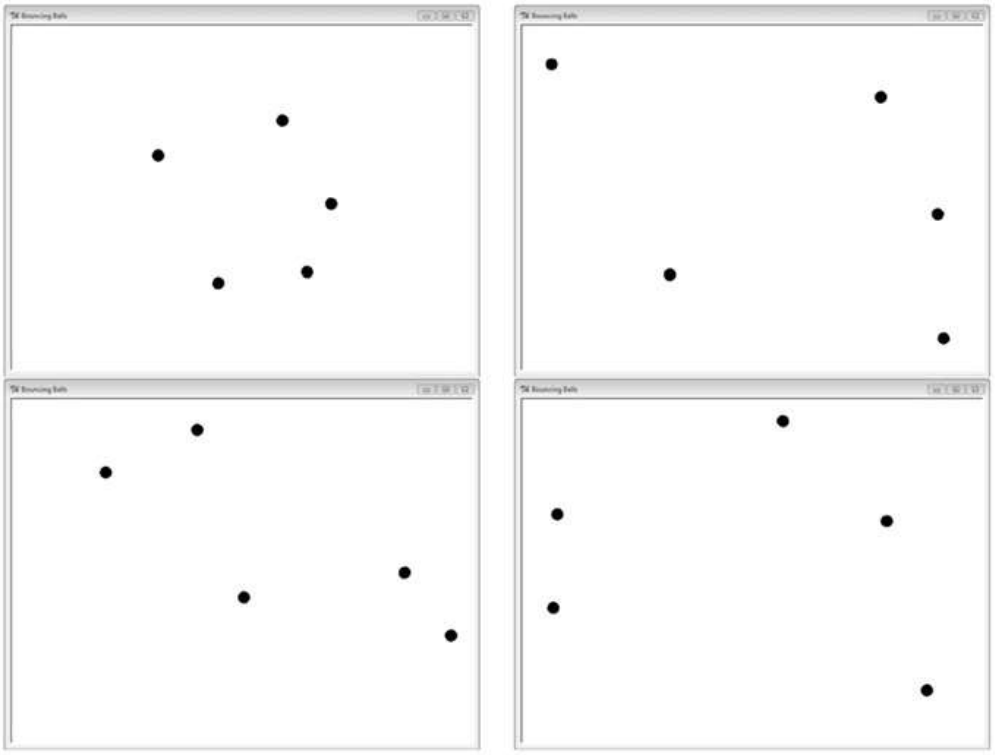


FIGURE 6-28 Execution of Bouncing Balls Program

In addition to the turtle graphics module, this program makes use of the time and random Python standard library modules to allow control of how long (in seconds) the simulation is executed, as indicated by the user, and to generate the random motion of the bouncing balls.

The main section of the program begins on **line 52** with the programming greeting. On **lines 58–60**, the size of the turtle screen (in pixels) is hard-coded into the program, assigned to variables screen_width and screen_height. Since all references to the screen size are through these variables, the desired window size can be altered by simply altering these variables.

```
# Bouncing Balls Simulation Program

import turtle
import random
import time

def atLeftEdge(ball, screen_width):
    if ball.xcor() < -screen_width / 2:
        return True
    else:
        return False

def atRightEdge(ball, screen_width):
    if ball.xcor() > screen_width / 2:
        return True
    else:
        return False

def atTopEdge(ball, screen_height):
    if ball.ycor() > screen_height / 2:
        return True
    else:
        return False

def atBottomEdge(ball, screen_height):
    if ball.ycor() < -screen_height / 2:
        return True
    else:
        return False

def bounceBall(ball, new_direction):
    if new_direction == 'left' or new_direction == 'right':
        new_heading = 180 - ball.heading()
    elif new_direction == 'down' or new_direction == 'up':
        new_heading =  360 - ball.heading()

    return new_heading

def createBalls(num_balls):
    balls = []
    for k in range(0, num_balls):
        new_ball = turtle.Turtle()
        new_ball.shape('circle')
        new_ball.fillcolor('black')
        new_ball.speed(0)
        new_ball.penup()
        new_ball.setheading(random.randint(1, 359))
        balls.append(new_ball)

    return balls

```

FIGURE 6-29 Bouncing Balls Program (*Continued*)

On **lines 63–64**, the turtle screen is created and its reference value assigned to variable window. The title of the window is assigned through a call to the title method. Following that, the user is prompted to enter the number of seconds for

the simulation, as well as the number of simultaneously bouncing balls.

```
52 # ---- main
53 # program greeting
54 print('This program simulates bouncing balls in a turtle screen')
55 print('for a specified number of seconds')
56
57 # init screen size
58 screen_width = 800
59 screen_height = 600
60 turtle.setup(screen_width, screen_height)
61
62 # create turtle window
63 window = turtle.Screen()
64 window.title('Bouncing Balls')
65
66 # prompt user for execution time and number of balls
67 num_seconds = int(input('Enter number of seconds to run: '))
68 num_balls = int(input('Enter number of balls in simulation: '))
69
70 # create balls
71 balls = createBalls(num_balls)
72
73 # set start time
74 start_time = time.time()
75
76 # begin simulation
77 terminate = False
78
79 while not terminate:
80     for k in range(0, len(balls)):
81         balls[k].forward(15)
82
83         if atLeftEdge(balls[k], screen_width):
84             balls[k].setheading(bounceBall(balls[k], 'right'))
85         elif atRightEdge(balls[k], screen_width):
86             balls[k].setheading(bounceBall(balls[k], 'left'))
87         elif atTopEdge(balls[k], screen_height):
88             balls[k].setheading(bounceBall(balls[k], 'down'))
89         elif atBottomEdge(balls[k], screen_height):
90             balls[k].setheading(bounceBall(balls[k], 'up'))
91
92         if time.time() - start_time > num_seconds:
93             terminate = True
94
95 # exit on close window
96 turtle.exitonclick()
```

FIGURE 6-29 Bouncing Balls Program

Function createBalls is called (on **line 71**) to create and return a list of turtle objects with a ball shape. The function definition ( **lines 39–50**) initializes an empty list named balls and creates the requested number of balls one-by-one, each appended to the list, by use of the for loop at **line 41**. Each ball is created with shape 'circle', fill color of 'black', speed of 0 (fastest speed), and with pen

attribute 'up'. In addition, the initial heading of each turtle is set to a random angle between 1 and 359 ( **line 47**).

Back in the main program section at **line 74**, the current time (in seconds) is obtained from a call to method time of the time module: time.time(). The current time value is stored in variable start_time. (The current time is the number of seconds since the "epoch," which is January 1, 1970. This will be discussed further in the Horse Racing program that follows.) The while loop beginning on **line 79** begins the simulation. The loop iterates as long as Boolean variable terminate is False (initialized to False on **line 77**). The for loop at **line 80** moves each of the specified number of balls a small distance until reaching one of the four edges of the window (left, right, top, or bottom edge). Boolean functions atLeftEdge, atRightEdge, atTopEdge, and atBottomEdge are used to determine when a ball is at an edge (defined in **lines 7–29**). Function bounceBall is called to bounce the ball in the opposite direction it is heading, and returns the new heading of the ball, passed as the argument to that ball's setheading method. Finally, on **line 92** a check is made to determine whether the user-requested simulation time has been exceeded. If so, Boolean variable terminate is set to True, and the program terminates. Because of the call to exitonclick() on **line 96**, the program will properly shut down when the close button of the turtle window is clicked.

Self-Test Questions **1.** A turtle screen is an 800-pixel wide by 600-pixel high graphics window. (TRUE/FALSE).
**2.** The three main attributes of a turtle object are _____, _____, and _____.

**3.** A turtle can be moved using either _____ or _____ positioning.
**4.** A turtle can only draw lines when it is not hidden. (TRUE/FALSE)

**5.** A turtle shape is limited to an arrow, turtle, circle, square, triangle, or classic (default) shape. (TRUE/FALSE)

**6.** What attribute of a turtle determines the size of the lines it draw?
**(a)** Pen size
**(b)** Turtle size

**7.** A turtle can draw in one of seven colors. (TRUE/FALSE)