LEARNING TO PROGRAM WITH PYTHON

Richard L. Halterman

Contents

Preface

Legal Notices and Information

This document is copyright ©2011 by Richard L. Halterman, all rights reserved.

Permission is hereby granted to make hardcopies and freely distribute the material herein under the following conditions:

- The copyright and this legal notice must appear in any copies of this document made in whole or in part.

- None of material herein can be sold or otherwise distributed for commercial purposes without written permission of the copyright holder.

- Instructors at any educational institution may freely use this document in their classes as a primary or optional textbook under the conditions specified above.

A local electronic copy of this document may be made under the terms specified for hardcopies:

- The copyright and these terms of use must appear in any electronic representation of this document made in whole or in part.

- None of material herein can be sold or otherwise distributed in an electronic form for commercial purposes without written permission of the copyright holder.

- Instructors at any educational institution may freely store this document in electronic form on a local server as a primary or optional textbook under the conditions specified above.

Additionally, a hardcopy or a local electronic copy must contain the uniform resource locator (URL) providing a link to the original content so the reader can check for updated and corrected content. The current URL is

```
http://python.cs.southern.edu/pythonbook/pythonbook.pdf
```

Chapter 1

The Context of Software Development

A computer program, from one perspective, is a sequence of instructions that dictate the flow of electrical impulses within a computer system. These impulses affect the computer's memory and interact with the display screen, keyboard, and mouse in such a way as to produce the "magic" that permits humans to perform useful tasks, solve high-level problems, and play games. One program allows a computer to assume the role of a financial calculator, while another transforms the machine into a worthy chess opponent. Note the two extremes here:

- at the lower, more concrete level electrical impulses alter the internal state of the computer, while

- at the higher, more abstract level computer users accomplish real-world work or derive actual pleasure.

So well is the higher-level illusion achieved that most computer users are oblivious to the lower-level activity (the machinery under the hood, so to speak). Surprisingly, perhaps, most programmers today write software at this higher, more abstract level also. An accomplished computer programmer can develop sophisticated software with little or no interest or knowledge of the actual computer system upon which it runs. Powerful software construction tools hide the lower-level details from programmers, allowing them to solve problems in higher-level terms.

The concepts of computer programming are logical and mathematical in nature. In theory, computer programs can be developed without the use of a computer. Programmers can discuss the viability of a program and reason about its correctness and efficiency by examining abstract symbols that correspond to the features of real-world programming languages but appear in no real-world programming language. While such exercises can be very valuable, in practice computer programmers are not isolated from their machines. Software is written to be used on real computer systems. Computing professionals known as *software engineers* develop software to drive particular systems. These systems are defined by their underlying hardware and operating system. Developers use concrete tools like compilers, debuggers, and profilers. This chapter examines the context of software development, including computer systems and tools.

1.1 Software

A computer program is an example of computer *software*. One can refer to a program as a *piece* of software as if it were a tangible object, but software is actually quite intangible. It is stored on a *medium*. A hard drive, a CD, a DVD, and a USB pen drive are all examples of media upon which software can reside. The CD is not the software; the software is a pattern on the CD. In order to be used, software must be stored in the computer's memory. Typically computer programs are loaded into memory from a medium like the computer's hard disk. An electromagnetic pattern representing the program is stored on the computer's hard drive. This pattern of electronic symbols must be transferred to the computer's memory before the program can be executed. The program may have been installed on the hard disk from a CD or from the Internet. In any case, the essence that was transferred from medium to medium was a pattern of electronic symbols that direct the work of the computer system.

These patterns of electronic symbols are best represented as a sequence of zeroes and ones, digits from the binary (base 2) number system. An example of a binary program sequence is

100010110110000100010000001001110

To the underlying computer hardware, specifically the processor, a zero here and three ones there might mean that certain electrical signals should be sent to the graphics device so that it makes a certain part of the display screen red. Unfortunately, only a minuscule number of people in the world would be able to produce, by hand, the complete sequence of zeroes and ones that represent the program Microsoft Word for an Intel-based computer running the Windows 7 operating system. Further, almost none of those who could produce the binary sequence would claim to enjoy the task.

The Word program for older Mac OS X computers using a PowerPC processor works similarly to the Windows version and indeed is produced by the same company, but the program is expressed in a completely different sequence of zeroes and ones! The Intel Core 2 Duo processor in the Windows machine accepts a completely different binary language than the PowerPC processor in the Mac. We say the processors have their own *machine language*.

1.2 Development Tools

If very few humans can (or want) to speak the machine language of the computers' processors and software is expressed in this language, how has so much software been developed over the years?

Software can be represented by printed words and symbols that are easier for humans to manage than binary sequences. Tools exist that automatically convert a higher-level description of what is to be done into the required lower-level code. Higher-level programming languages like Python allow programmers to express solutions to programming problems in terms that are much closer to a natural language like English. Some examples of the more popular of the hundreds of higher-level programming languages that have been devised over the past 60 years include FORTRAN, COBOL, Lisp, Haskell, C, Perl, C++, Java, and C#. Most programmers today, especially those concerned with high-level applications, usually do not worry about the details of underlying hardware platform and its machine language.

One might think that ideally such a conversion tool would accept a description in a natural language, such as English, and produce the desired executable code. This is not possible today because natural languages are quite complex compared to computer programming languages. Programs called *compilers* that translate one computer language into another have been around for 60 years, but natural language processing is still an active area of artificial intelligence research. Natural languages, as they are used

by most humans, are inherently ambiguous. To understand properly all but a very limited subset of a natural language, a human (or artificially intelligent computer system) requires a vast amount of background knowledge that is beyond the capabilities of today's software. Fortunately, programming languages provide a relatively simple structure with very strict rules for forming statements that can express a solution to any program that can be solved by a computer.

Consider the following program fragment written in the Python programming language:

```
subtotal = 25
tax = 3
total = subtotal + tax
```

These three lines do not make up a complete Python program; they are merely a piece of a program. The statements in this program fragment look similar to expressions in algebra. We see no sequence of binary digits. Three words, subtotal, tax, and total, called *variables*, are used to hold information. Mathematicians have used variables for hundreds of years before the first digital computer was built. In programming, a variable represents a value stored in the computer's memory. Familiar operators (= and +) are used instead of some cryptic binary digit sequence that instructs the processor to perform the operation. Since this program is expressed in the Python language, not machine language, it cannot be executed directly on any processor. A program called an *interpreter* translates the Python code into machine code when a user runs the program.

The higher-level language code is called *source code*. The interpreted machine language code is called the *target code*. The interpreter translates the source code into the target machine language.

The beauty of higher-level languages is this: the same Python source code can execute on different target platforms. The target platform must have a Python interpreter available, but multiple Python interpreters are available for all the major computing platforms. The human programmer therefore is free to think about writing the solution to the problem in Python, not in a specific machine language.

Programmers have a variety of tools available to enhance the software development process. Some common tools include:

- **Editors**. An *editor* allows the programmer to enter the program source code and save it to files. Most programming editors increase programmer productivity by using colors to highlight language features. The *syntax* of a language refers to the way pieces of the language are arranged to make well-formed sentences. To illustrate, the sentence

    The tall boy runs quickly to the door.

  uses proper English syntax. By comparison, the sentence

    Boy the tall runs door to quickly the.

  is not correct syntactically. It uses the same words as the original sentence, but their arrangement does not follow the rules of English.

  Similarly, programming languages have strict syntax rules that must be followed to create well-formed programs. Only well-formed programs are acceptable and can be compiled and executed. Some syntax-aware editors can use colors or other special annotations to alert programmers of syntax errors before the program is compiled.

- **Compilers**. A *compiler* translates the source code to target code. The target code may be the machine language for a particular platform or embedded device. The target code could be another source language; for example, the earliest C++ compiler translated C++ into C, another higher-level language.

The resulting C code was then processed by a C compiler to produce an executable program. (C++ compilers today translate C++ directly into machine language.)

- **Interpreters**. An *interpreter* is like a compiler, in that it translates higher-level source code into machine language. It works differently, however. While a compiler produces an executable program that may run many times with no additional translation needed, an interpreter translates source code statements into machine language as the program runs. A compiled program does not need to be re-compiled to run, but an interpreted program must be interpreted each time it is executed. In general, compiled programs execute more quickly than interpreted programs because the translation activity occurs only once. Interpreted programs, on the other hand, can run as is on any platform with an appropriate interpreter; they do not need to be recompiled to run on a different platform. Python, for example, is used mainly as an interpreted language, but compilers for it are available. Interpreted languages are better suited for dynamic, explorative development which many people feel is ideal for beginning programmers.

- **Debuggers**. A *debugger* allows programmers to simultaneously run a program and see which source code line is currently being executed. The values of variables and other program elements can be watched to see if their values change as expected. Debuggers are valuable for locating errors (also called *bugs*) and repairing programs that contain errors. (See Section 3.4 for more information about programming errors.)

- **Profilers**. A *profiler* is used to evaluate a program's performance. It indicates how many times a portion of a program is executed during a particular run, and how long that portion takes to execute. Profilers also can be used for testing purposes to ensure all the code in a program is actually being used somewhere during testing. This is known as *coverage*. It is common for software to fail after its release because users exercise some part of the program that was not executed anytime during testing. The main purpose of profiling is to find the parts of a program that can be improved to make the program run faster.

Many developers use integrated development environments (IDEs). An IDE includes editors, debuggers, and other programming aids in one comprehensive program. Examples of commercial IDEs include Microsoft's Visual Studio 2010, the Eclipse Foundation's Eclipse IDE, and Apple's XCode. IDLE is a very simple IDE for Python.

Despite the plethora of tools (and tool vendors' claims), the programming process for all but trivial programs is not automatic. Good tools are valuable and certainly increase the productivity of developers, but they cannot write software. There are no substitutes for sound logical thinking, creativity, common sense, and, of course, programming experience.

1.3 Learning Programming with Python

Guido van Rossum created the Python programming language in the late 1980s. In contrast to other popular languages such as C, C++, Java, and C#, Python strives to provide a simple but powerful syntax.

Python is used for software development at companies and organizations such as Google, Yahoo, CERN, Industrial Light and Magic, and NASA. Experienced programmers can accomplish great things with Python, but Python's beauty is that it is accessible to beginning programmers and allows them to tackle interesting problems more quickly than many other, more complex languages that have a steeper learning curve.

More information about Python, including links to download the latest version for Microsoft Windows, Mac OS X, and Linux, can be found at

```
http://www.python.org
```

The code in this book is based on Python 3.

This book does not attempt to cover all the facets of the Python programming language. Experienced programmers should look elsewhere for books that cover Python in much more detail. The focus here is on introducing programming techniques and developing good habits. To that end, our approach avoids some of the more esoteric features of Python and concentrates on the programming basics that transfer directly to other imperative programming languages such as Java, C#, and C++. We stick with the basics and explore more advanced features of Python only when necessary to handle the problem at hand.

1.4 Writing a Python Program

Python programs must be written with a particular structure. The syntax must be correct, or the interpreter will generate error messages and not execute the program. This section introduces Python by providing a simple example program.

Listing 1.1 (simple.py) is one of the simplest Python programs that does something:

Listing 1.1: simple.py

```
1  print("This is a simple Python program")
```

Note: The small numbers that appear to the left of the box containing the Python code are not part of the program; the numbers are shown to allow us to easily reference a specific line in the code if necessary.

We will consider two ways in which we can run Listing 1.1 (simple.py):

1. enter the program directly into IDLE's interactive shell and

2. enter the program into IDLE's editor, save it, and run it.

IDLE's interactive shell.

IDLE is a simple Python integrated development environment available for Windows, Linux, and Mac OS X. Figure 1.1 shows how to start IDLE from the Microsoft Windows Start menu. The IDLE interactive shell is shown in Figure 1.2. You may type the above one line Python program directly into IDLE and press enter to execute the program. Figure 1.3 shows the result using the IDLE interactive shell.

Since it does not provide a way to save the code you enter, the interactive shell is not the best tool for writing larger programs. The IDLE interactive shell is useful for experimenting with small snippets of Python code.

IDLE's editor. IDLE has a built in editor. From the IDLE menu, select *New Window*, as shown in Figure 1.4. Type the text as shown in Listing 1.1 (simple.py) into the editor. Figure 1.5 shows the resulting editor window with the text of the simple Python program. You can save your program using the *Save* option in the *File* menu as shown in Figure 1.6. Save the code to a file named simple.py. The actual name of the file is irrelevant, but the name "simple" accurately describes the nature of this program. The extension .py is the extension used for Python source code. We can run the program from within the IDLE editor by pressing the **F5** function key or from the editor's *Run* menu: *Run→Run Module*. The output appears in the IDLE interactive shell window.
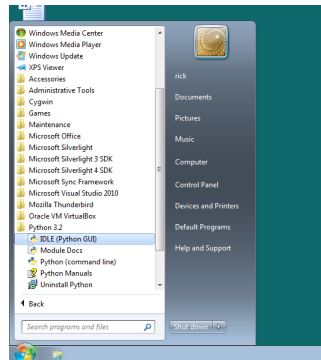
Figure 1.1: Launching IDLE from the Windows Start menu
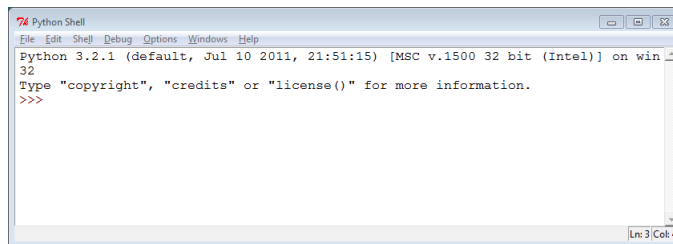
Figure 1.2: The IDLE interpreter Window
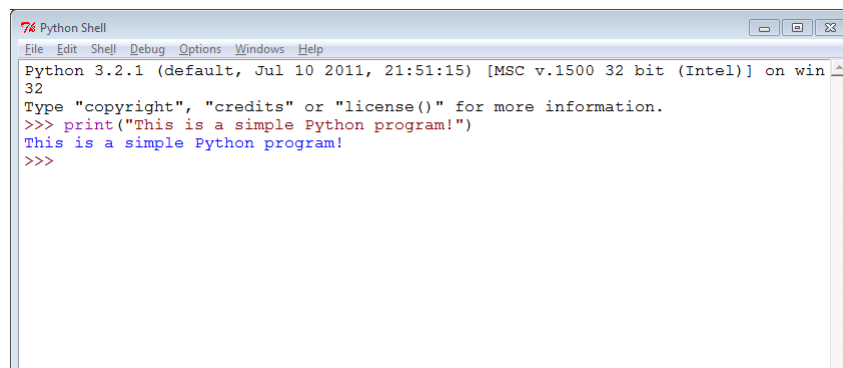
Figure 1.3: A simple Python program entered and run with the IDLE interactive shell

The editor allows us to save our programs and conveniently make changes to them later. The editor understands the syntax of the Python language and uses different colors to highlight the various components that comprise a program. Much of the work of program development occurs in the editor.
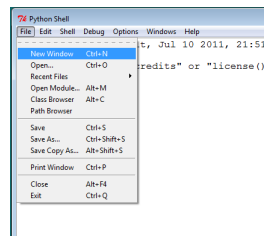
Figure 1.4: Launching the IDLE editor
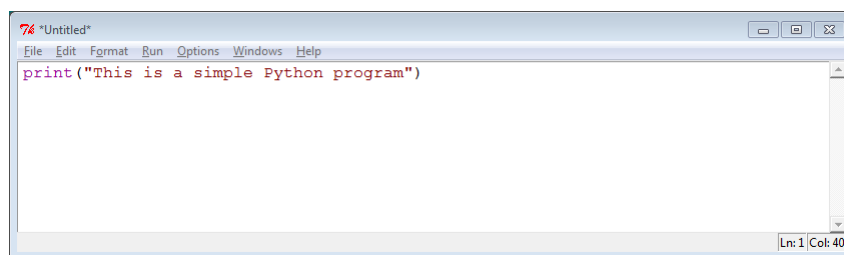
Figure 1.5: The simple Python program typed into the IDLE editor
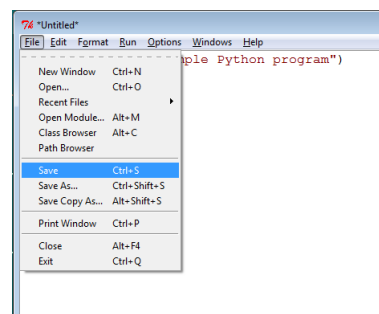
Figure 1.6: Saving a file created with the IDLE editor

Listing 1.1 (simple.py) contains only one line of code:

```
print("This is a simple Python program")
```

This is a Python statement. A statement is a command that the interpreter executes. This statement prints the message *This is a simple Python program* on the screen. A statement is the fundamental unit of execution in a Python program. Statements may be grouped into larger chunks called blocks, and blocks can make up more complex statements. Higher-order constructs such as functions and methods are composed of blocks. The statement

```
print("This is a simple Python program")
```

makes use of a built in function named print. Python has a variety of different kinds of statements that may be used to build programs, and the chapters that follow explore these various kinds of statements.
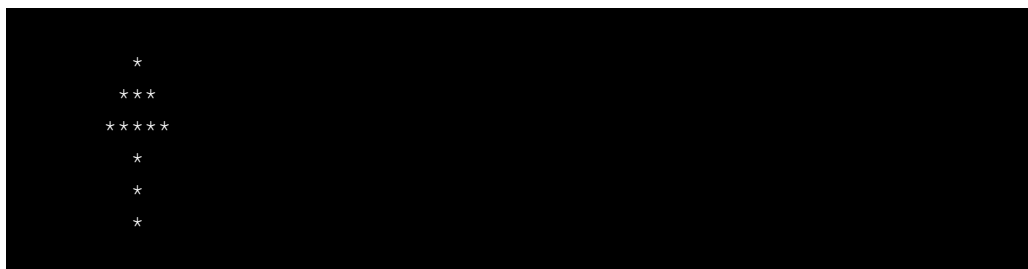
1.5   A Longer Python program

More interesting programs contain multiple statements. In Listing 1.2 (arrow.py), six print statements draw an arrow on the screen:

Listing 1.2: arrow.py

```
1  print("    *    ")
2  print("   ***   ")
3  print("  *****  ")
4  print("    *    ")
5  print("    *    ")
6  print("    *    ")
```

We wish the output of Listing 1.2 (arrow.py) to be

If you try to enter each line one at a time into the IDLE interactive shell, the program's output will be intermingled with the statements you type. In this case the best approach is to type the program into an editor, save the code you type to a file, and then execute the program. Most of the time we use an editor to enter and run our Python programs. The interactive interpreter is most useful for experimenting with small snippets of Python code.

In Listing 1.2 (arrow.py) each print statement "draws" a horizontal slice of the arrow. All the horizontal slices stacked on top of each other results in the picture of the arrow. The statements form a *block* of Python code. It is important that no *whitespace* (spaces or tabs) come before the beginning of each statement. In Python the indentation of statements is significant and must be done properly. If we try to put a single space before a statement in the interactive shell, we get

```
>>> print('hi')
  File "<stdin>", line 1
    print('hi')
    ^
IndentationError: unexpected indent
```

The interpreter reports a similar error when we attempt to run a saved Python program if the code contains such extraneous indentation.

1.6 Summary

- Computers require both hardware and software to operate. Software consists of instructions that control the hardware.

- At the lowest level, the instructions for a computer program can be represented as a sequence of zeros and ones. The pattern of zeros and ones determine the instructions performed by the processor.

- Two different kinds of processors can have different machine languages.

- Application software can be written largely without regard to the underlying hardware. Tools automatically translate the higher-level, abstract language into the machine language required by the hardware.

- A compiler translates a source file into an executable file. The executable file may be run at any time with no further translation needed.

- An interpreter translates a source file into machine language as the program executes. The source file itself is the executable file, but it must be interpreted each time a user executes it.

- Compiled programs generally execute more quickly than interpreted programs. Interpreted languages generally allow for a more interactive development experience.

- Programmers develop software using tools such as editors, compilers, interpreters, debuggers, and profilers.

- Python is a higher-level programming language. It is considered to be a higher-level language than C, C++, Java, and C#.

- An IDE is an integrated development environment—one program that provides all the tools that developers need to write software.

- Messages can be printed in the output window by using Python's `print` function.

- A Python program consists of a code block. A block is made up of statements.

1.7 Exercises

1. What is a compiler?

2. What is an interpreter?

3. How is a compiler similar to an interpreter? How are they different?

4. How is compiled or interpreted code different from source code?

5. What tool does a programmer use to produce Python source code?

6. What is necessary to execute a Python program?

7. List several advantages developing software in a higher-level language has over developing software in machine language.

8. How can an IDE improve a programmer's productivity?

9. What the "official" Python IDE?

10. What is a *statement* in a Python program?

Chapter 2

Values and Variables

In this chapter we explore some building blocks that are used to develop Python programs. We experiment with the following concepts:

- numeric values

- variables

- assignment

- identifiers

- reserved words

In the next chapter we will revisit some of these concepts in the context of other data types.

2.1 Integer Values

The number four (4) is an example of a *numeric* value. In mathematics, 4 is an *integer* value. Integers are whole numbers, which means they have no fractional parts, and they can be positive, negative, or zero. Examples of integers include 4, −19, 0, and −1005. In contrast, 4.5 is not an integer, since it is not a whole number.

Python supports a number of numeric and non-numeric values. In particular, Python programs can use integer values. The Python statement

```
print(4)
```

prints the value 4. Notice that unlike Listing 1.1 (`simple.py`) and Listing 1.2 (`arrow.py`) no quotation marks (`"`) appear in the statement. The value 4 is an example of an integer *expression*. Python supports other types of expressions besides integer expressions. An expression is part of a statement.

The number 4 by itself is not a complete Python statement and, therefore, cannot be a program. The interpreter, however, can evaluate a Python expression. You may type the enter 4 directly into the interactive interpreter shell:

```
Python 3.2.1 (default, Jul 10 2011, 21:51:15) [MSC v.1500 32 bit (Intel)] o
32
Type "help", "copyright", "credits" or "license" for more information.
>>> 4
4
>>>
```

The interactive shell attempts to evaluate both expressions and statements. In this case, the expression 4 evaluates to 4. The shell executes what is commonly called the *read, eval, print loop*. This means the interactive shell's sole activity consists of

1. *reading* the text entered by the user,

2. attempting to *evaluate* the user's input in the context of what the user has entered up that point, and

3. *printing* its evaluation of the user's input.

If the user enters a 4, the shell interprets it as a 4. If the user enters x = 10, a statement has has no overall value itself, the shell prints nothing. If the user then enters x, the shell prints the evaluation of x, which is 10. If the user next enters y, the shell reports a error because y has not been defined in a previous interaction.

Python uses the + symbol with integers to perform normal arithemtic addition, so the interactive shell can serve as a handy adding machine:

```
>>> 3 + 4
7
>>> 1 + 2 + 4 + 10 + 3
20
>>> print(1 + 2 + 4 + 10 + 3)
20
```

The last line evaluated shows how we can use the + symbol to add values within a print statement that could be part of a Python program.

Consider what happens if we use quote marks around an integer:

```
>>> 19
19
>>> "19"
'19'
>>> '19'
'19'
```

Notice how the output of the interpreter is different. The expression "19" is an example of a *string* value. A string is a sequence of characters. Strings most often contain non-numeric characters:

```
>>> "Fred"
'Fred'
>>> 'Fred'
'Fred'
```

Python recognizes both single quotes (') and double quotes (") as valid ways to delimit a string value. If a single quote marks the beginning of a string value, a single quote must delimit the end of the string. Similarly, the double quotes, if used instead, must appear in pairs. You may not mix the quotes when representing a string:

```
>>> 'ABC'
'ABC'
>>> "ABC"
'ABC'
>>> 'ABC"
  File "<stdin>", line 1
    'ABC"
        ^
SyntaxError: EOL while scanning string literal
>>> "ABC'
  File "<stdin>", line 1
    "ABC'
        ^
SyntaxError: EOL while scanning string literal
```

The interpreter's output always uses single quotes, but it accepts either single or double quotes as valid input.

Consider the following interaction sequence:

```
>>> 19
19
>>> "19"
'19'
>>> '19'
'19'
>>> "Fred"
'Fred'
>>> 'Fred'
'Fred'
>>> Fred
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Fred' is not defined
```

The expression `Fred` (without quotes) was not accepted by the interpreter because of the missing quotation marks.

It is important to note that the expressions `4` and `'4'` are different. One is an integer expression and the other is a string expression. All expressions in Python have a *type*. The type of an expression indicates the kind of expression it is. An expression's type is sometimes denoted as its *class*. At this point we have considered only integers and strings. The built in `type` function reveals the type of any Python expression:

```
>>> type(4)
<class 'int'>
>>> type('4')
<class 'str'>
```

Python associates the type name `int` with integer expressions and `str` with string expressions.

The built in `int` function converts the string representation of an integer to an actual integer, and the `str` function converts an integer expression to a string:

```
>>> 4
4
>>> str(4)
'4'
>>> '5'
'5'
>>> int('5')
5
```

The expression `str(4)` evaluates to the string value `'4'`, and `int('5')` evaluates to the integer value 5. The `int` function applied to an integer evaluates simply to the value of the integer itself, and similarly `str` applied to a string results in the same value as the original string:

```
>>> int(4)
4
>>> str('Judy')
'Judy'
```

As you might guess, there is little reason for a programmer to perform these kinds of transformations. In fact, the expression `str('4')` is more easily expressed as 4, so the utility of the `str` and `int` functions will not become apparent until we introduce variables in Section 2.2.

Any integer has a string representation, but not all strings have an integer equivalent:

```
>>> str(1024)
'1024'
>>> int('wow')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'wow'
>>> int('3.4')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '3.4'
```

In Python, neither *wow* nor 3.4 represent valid integer expressions. In short, if the contents of the string (the characters that make it up) look like a valid integer number, you safely can apply the int function to produce the represented integer.

The plus operator (+) works differently for strings; consider:

```
>>> 5 + 10
15
>>> '5' + '10'
'510'
>>> 'abc' + 'xyz'
'abcxyz'
```

As you can see, the result of the expression 5 + 10 is very different from '5' + '10'. The plus operator splices two strings together in a process known as *concatenation*. Mixing the two types directly is not allowed:

```
>>> '5' + 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> 5 + '10'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

but the int and str functions can help:

```
>>> 5 + int('10')
15
>>> '5' + str(10)
'510'
```

The `type` function can determine the type of the most complicated expressions:

```
>>> type(4)
<class 'int'>
>>> type('4')
<class 'str'>
>>> type(4 + 7)
<class 'int'>
>>> type('4' + '7')
<class 'str'>
>>> type(int('3') + int(4))
<class 'int'>
```

Commas may not appear in Python integer values. The number two thousand, four hundred sixty-eight would be written `2468`, not `2,468`.

In mathematics, integers are unbounded; said another way, the set of mathematical integers is infinite. In Python, integers may be arbitrarily large, but the larger the integer, the more memory required to represent it. This means Python integers theoretically can be as large or as small as needed, but, since a computer has a finite amount of memory (and the operating system may limit the amount of memory allowed for a running program), in practice Python integers are bounded by available memory.

2.2 Variables and Assignment

In algebra, variables represent numbers. The same is true in Python, except Python variables also can represent values other than numbers. Listing 2.1 (`variable.py`) uses a variable to store an integer value and then prints the value of the variable.

Listing 2.1: `variable.py`

```
1  x = 10
2  print(x)
```

Listing 2.1 (`variable.py`) contains two statements:

- `x = 10`

  This is an *assignment* statement. An assignment statement associates a value with a variable. The key to an assignment statement is the symbol = which is known as the *assignment operator*. The statement assigns the integer value 10 to the variable `x`. Said another way, this statement binds the variable named `x` to the value 10. At this point the type of `x` is `int` because it is bound to an integer value.

  A variable may be assigned and reassigned as often as necessary. The type of a variable will change if it is reassigned an expression of a different type.

- `print(x)`

  This statement prints the variable `x`'s current value. Note that the lack of quotation marks here is very important. If `x` has the value 10, the statement

```
print(x)
```

prints 10, the value of the variable x, but the statement

```
print('x')
```

prints x, the message containing the single letter *x*.

The meaning of the assignment operator (=) is different from equality in mathematics. In mathematics, = asserts that the expression on its left is equal to the expression on its right. In Python, = makes the variable on its left take on the value of the expression on its right. It is best to read x = 5 as "x is assigned the value 5," or "x gets the value 5." This distinction is important since in mathematics equality is symmetric: if $x = 5$, we know $5 = x$. In Python this symmetry does not exist; the statement

```
5 = x
```

attempts to reassign the value of the literal integer value 5, but this cannot be done because 5 is always 5 and cannot be changed. Such a statement will produce an error.

```
>>> x = 5
>>> x
5
>>> 5 = x
  File "<stdin>", line 1
SyntaxError: can't assign to literal
```

Variables can be reassigned different values as needed, as Listing 2.2 (multipleassignment.py) shows.

Listing 2.2: **multipleassignment.py**

```
1  x = 10
2  print('x = ' + str(x))
3  x = 20
4  print('x = ' + str(x))
5  x = 30
6  print('x = ' + str(x))
```

Observe that each print statement in Listing 2.2 (multipleassignment.py) is identical, but when the program runs (as a program, not in the interactive shell) the print statements produce different results:

```
x = 10
x = 20
x = 30
```

The variable x has type int, since it is bound to an integer value. Observe how Listing 2.2 (multipleassignment.py) uses the str function to treat x as a string so the + operator will use string concatenation:

```
print('x = ' + str(x))
```

The expression 'x = ' + x would not be legal, because, as we have seen (Section 2.1), the plus operator may not be applied with mixed string and integer operands.

Listing 2.3 (multipleassignment2.py) provides a variation of Listing 2.2 (multipleassignment.py) that produces the same output.

Listing 2.3: multipleassignment2.py

```
1  x = 10
2  print('x =',  x)
3  x = 20
4  print('x =',  x)
5  x = 30
6  print('x =',  x)
```

This version of the print statement:

```
print('x =',  x)
```

illustrates the print function accepting two parameters. The first parameter is the string 'x =', and the second parameter is the variable x bound to an integer value. The print function allows programmers to pass multiple expressions to print, each separated by commas. The elements within the parentheses of the print function comprise what is known as a *comma-separated list*. The print function prints each element in the comma-separated list of parameters. The print function automatically prints a space between each element in the list so they do not run together.

A programmer may assign multiple variables in one statement using *tuple assignment*. Listing 2.4 (tupleassign.py) shows how:

Listing 2.4: tupleassign.py

```
1  x, y, z = 100, -45, 0
2  print('x =', x, '  y =', y, '  z =', z)
```

The Listing 2.4 (tupleassign.py) program produces

```
x = 100    y = -45    z = 0
```

A *tuple* is a comma separated list of expressions. In the assignment statement

```
x, y, z = 100, -45, 0
```

x, y, z is one tuple, and 100, -45, 0 is another tuple. Tuple assignment works as follows: The first variable in the tuple on left side of the assignment operator is assigned the value of the first expression in the tuple on the left side (effectively x = 100). Similarly, the second variable in the tuple on left side of the assignment operator is assigned the value of the second expression in the tuple on the left side (in effect y = -45). z gets the value 0.

An assignment statement binds a variable name to an object. We can visualize this process with a box and arrow as shown in Figure 2.1.

We name the box with the variable's name. The arrow projecting from the box points to the object to which the variable is bound. Figure 2.2 shows how variable bindings change as the following sequence of Python executes:
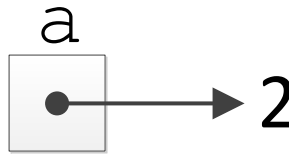
Figure 2.1: Binding a variable to an object

```
a = 2
b = 5
a = b
b = 4
```

Importantly, the statement

```
a = b
```

means that a and b both are bound to the same numeric object. Note that reassigning b does not affect a's value.

Not only may a variable's value change during its use within an executing program; the type of a variable can change as well. Consider Listing 2.5 (changeabletype.py).

Listing 2.5: changeabletype.py

```
1  a = 10
2  print('First, variable a has value', a, 'and type', type(a))
3  a = 'ABC'
4  print('Now, variable a has value', a, 'and type', type(a))
```

Listing 2.5 (changeabletype.py) produces the following output:

```
First, variable a has value 10 and type <class 'int'>
Now, variable a has value ABC and type <class 'str'>
```

Most variables maintain their original type throughout a program's execution. A variable should have a specific meaning within a program, and its meaning should not change during the program's execution. While not always the case, sometimes when a variable's type changes its meaning changes as well.

2.3 Identifiers

While mathematicians are content with giving their variables one-letter names like x, programmers should use longer, more descriptive variable names. Names such as sum, height, and sub_total are much better than the equally permissible s, h, and st. A variable's name should be related to its purpose within the
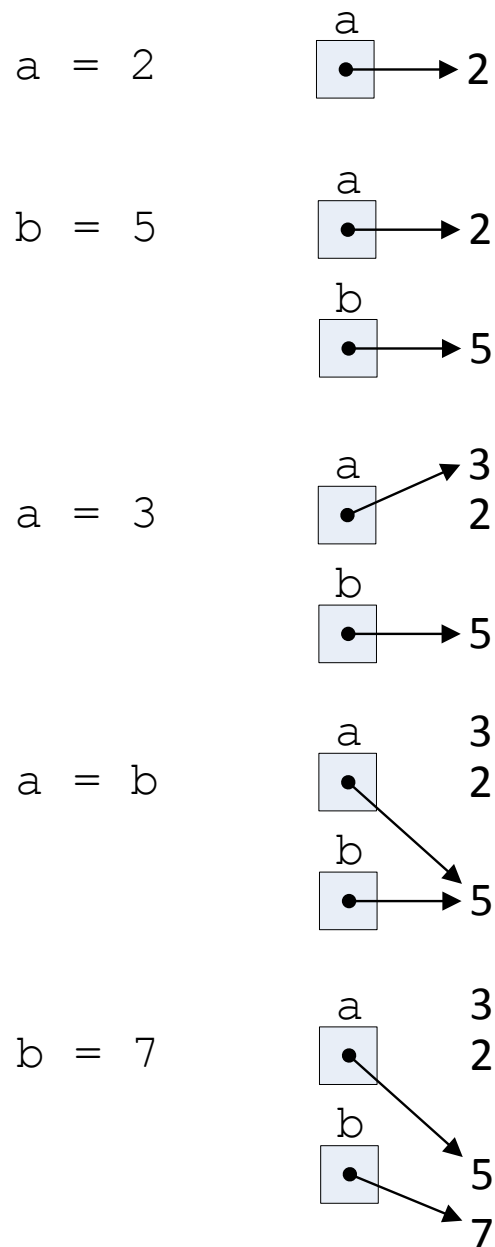
Figure 2.2: How variable bindings change as a program runs

program. Good variable names make programs more readable by humans. Since programs often contain many variables, well-chosen variable names can render an otherwise obscure collection of symbols more understandable.

Python has strict rules for variable names. A variable name is one example of an *identifier*. An identifier

and	del	from	None	try
as	elif	global	nonlocal	True
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

Table 2.1: Python keywords

is a word used to name things. One of the things an identifier can name is a variable. We will see in later chapters that identifiers name other things such as functions, classes, and methods. Identifiers have the following form:

- Identifiers must contain at least one character.

- The first character must be an alphabetic letter (upper or lower case) or the underscore

    ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_

- The remaining characters (if any) may be alphabetic characters (upper or lower case), the underscore, or a digit

    ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_0123456789

- No other characters (including spaces) are permitted in identifiers.

- A reserved word cannot be used as an identifier (see Table 2.1).

Here are some examples of valid and invalid identifiers:

- All of the following words are valid identifiers and so can be used as variable names: x, x2, total, port_22, and FLAG.

- None of the following words are valid identifiers: sub-total (dash is not a legal symbol in an identifier), first entry (space is not a legal symbol in an identifier), 4all (begins with a digit), #2 (pound sign is not a legal symbol in an identifier), and class (class is a reserved word).

Python reserves a number of words for special use that could otherwise be used as identifiers. Called *reserved words* or *keywords*, these words are special and are used to define the structure of Python programs and statements. Table 2.1 lists all the Python reserved words. The purposes of many of these reserved words are revealed throughout this book.

None of the reserved words in Table 2.1 may be used as identifiers. Fortunately, if you accidentally attempt to use one of the reserved words as a variable name within a program, the interpreter will issue an error:

```
>>> class = 15
  File "<stdin>", line 1
    class = 15
          ^
SyntaxError: invalid syntax
```

(see Section 3.4 for more on interpreter generated errors).

To this point we have avoided keywords completely in our programs. This means there is nothing special about the names `print`, `int`, `str`, or `type`, other than they happen to be the names of built-in functions. We are free to reassign these names and use them as variables. Consider the following interactive sequence that reassigns the name `print` to mean something new:

```
>>> print('Our good friend print')
Our good friend print
>>> print
<built-in function print>
>>> type(print)
<class 'builtin_function_or_method'>
>>> print = 77
>>> print
77
>>> print('Our good friend print')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
>>> type(print)
<class 'int'>
```

Here we used the name `print` as a variable. In so doing it lost its original behavior as a function to print the console. While we can reassign the names `print`, `str`, `type`, etc., it generally is not a good idea to do so.

Not only can a function name can be reassigned, but a variable can be assigned to a function.

```
>>> my_print = print
>>> my_print('hello from my_print!')
hello from my_print!
```

After binding `my_print` to `print` we can use `my_print` is exactly the same way as the built-in `print` function.

Python is a case-sensitive language. This means that capitalization matters. `if` is a reserved word, but none of `If`, `IF`, or `iF` are reserved words. Identifiers also are case sensitive; the variable called `Name` is different from the variable called `name`. Note that three of the reserved words (`False`, `None`, and `True`) are capitalized.

Variable names should not be distinguished merely by differences in capitalization because it can be confusing to human readers. For the same reason, it is considered poor practice to give a variable the same name as a reserved word with one or more of its letters capitalized.

The most important thing to remember about variables names is that they should be well chosen. A variable's name should reflect the variable's purpose within the program. For example, consider a program controlling a point-of-sale terminal (also known as an electronic cash register). The variable keeping track of the total cost of goods purchased might be named `total` or `total_cost`. Variable names such as `a67_99` and `fred` would be poor choices.

Title	Storage	Smallest Magnitude	Largest Magnitude	Minimum Precision
`float`	64 bits	2.22507×10^{-308}	$1.79769 \times 10^{+308}$	15 digits

Table 2.2: Characteristics of Floating-point Numbers on 32-bit Computer Systems

2.4 Floating-point Types

Many computational tasks require numbers that have fractional parts. For example, to compute the area of a circle given the circle's radius, the value π, or approximately 3.14159 is used. Python supports such non-integer numbers, and they are called *floating point numbers*. The name implies that during mathematical calculations the decimal point can move or "float" to various positions within the number to maintain the proper number of significant digits. The Python name for the floating-point type is `float`. Consider the following interactive session:

```
>>> x = 5.62
>>> x
5.62
>>> type(x)
<class 'float'>
```

The range of floating-points values (smallest value to largest value, both positive and negative) and precision (the number of digits available) depends of the Python implementation for a particular machine. Table 2.2 provides some information about floating point values as commonly implemented on 32-bit computer systems. Floating point numbers can be both positive and negative.

As you can see from Table 2.2, unlike Python integers which can be arbitrarily large (or, for negatives, arbitrarily small), floating-point numbers have definite bounds.

Listing 2.6 (`pi-print.py`) prints an approximation of the mathematical value π.

Listing 2.6: `pi-print.py`

```
pi = 3.14159;
print("Pi =", pi)
print("or", 3.14, "for short")
```

The first line in Listing 2.6 (`pi-print.py`) prints the value of the variable `pi`, and the second line prints a literal value. Any literal numeric value with a decimal point in a Python program automatically has the type `float`.

Floating-point numbers are an approximation of mathematical real numbers. The range of floating point numbers is limited, since each value must be stored in a fixed amount of memory. Floating-point numbers differ from integers in another, very important way. Any integer can be represented exactly. This is not true necessarily for a floating-point number. Consider the real number π. π is an irrational number which means it contains an infinite number of digits with no pattern that repeats. Since π contains an infinite number of digits, its value only can be approximated. Because of the limited number of digits available, some numbers with a finite number of digits can be only approximated; for example, the number 23.3123400654033989 contains too many digits for the `float` type and must be approximated; Python stores it as 23.312340065403397:

```
>>> x = 23.3123400654033989
>>> x
23.312340065403397
```

An example of the problems that can arise due to the inexact nature of floating-point numbers is demonstrated later in Listing 3.2 (`imprecise.py`).

Floating-point numbers can be expressed in scientific notation. Since most programming editors do not provide superscripting and special symbols like $\times$, the normal scientific notation is altered slightly. The number 6.022×10^{23} is written `6.022e23`. The number to the left of the `e` (capital `E` can be used as well) is the mantissa, and the number to the right of the `e` is the exponent of 10. As another example, -5.1×10^{-4} is expressed in Python as `-5.1e-4`. Listing 2.7 (`scientificnotation.py`) prints some scientific constants using scientific notation.

Listing 2.7: `scientificnotation.py`

```
1  avogadros_number = 6.022e23
2  c = 2.998e8
3  print("Avogadro's number =", avogadros_number)
4  print("Speed of light =", c)
```

2.5 Control Codes within Strings

The characters that can appear within strings include letters of the alphabet (`A-Z`, `a-z`), digits (`0-9`), punctuation (`.`, `:`, `,`, etc.), and other printable symbols (`#`, `&`, `%`, etc.). In addition to these "normal" characters, we may embed special characters known as *control codes*. Control codes control the way text is rendered in a console window or paper printer. The backslash symbol (\) signifies that the character that follows it is a control code, not a literal character. The string `'\n'` thus contains a single control code. The backslash is known as the *escape symbol*, and in this case we say the n symbol is *escaped*. The \n control code represents the *newline* control code which moves the text cursor down to the next line in the console window. Other control codes include \t for tab, \f for a form feed (or page eject) on a printer, \b for backspace, and \a for alert (or bell). The \b and \a do not produce the desired results in the IDLE interactive shell, but they work properly in a command shell. Listing 2.8 (`specialchars.py`) prints some strings containing some of these control codes.

Listing 2.8: `specialchars.py`

```
1  print('A\nB\nC')
2  print('D\tE\tF')
3  print('WX\bYZ')
4  print('1\a2\a3\a4\a5\a6')
```

When executed in a command shell, Listing 2.8 (`specialchars.py`) produces

```
A
B
C
D       E       F
WYZ
123456
```

On most systems, the computer's speaker beeps fives when printing the last line.

A string with a single quotation mark at the beginning must be terminated with a single quote; similarly, A string with a double quotation mark at the beginning must be terminated with a double quote. A single-quote string may have embedded double quotes, and a double-quote string may have embedded single quotes. If you wish to embed a single quote mark within a single-quote string, you can use the backslash to escape the single quote (\'). An unprotected single quote mark would terminate the string. Similarly, you may protect a double quote mark in a double-quote string with a backslash (\"). Listing 2.9 (escapequotes.py) shows the various ways in which quotation marks may be embedded within string literals.

Listing 2.9: escapequotes.py

```python
print("Did you know that 'word' is a word?")
print('Did you know that "word" is a word?')
print('Did you know that \'word\' is a word?')
print("Did you know that \"word\" is a word?")
```

The output of Listing 2.9 (escapequotes.py) is

```
Did you know that 'word' is a word?
Did you know that "word" is a word?
Did you know that 'word' is a word?
Did you know that "word" is a word?
```

Since the backslash serves as the escape symbol, in order to embed a literal backslash within a string you must use two backslashes in succession. Listing 2.10 (printpath.py) prints a string with embedded backslashes.

Listing 2.10: printpath.py

```python
filename = 'C:\\Users\\rick'
print(filename)
```

Listing 2.10 (printpath.py) displays

```
C:\Users\rick
```

2.6   User Input

The `print` function enables a Python program to display textual information to the user. Programs may use the `input` function to obtain information from the user. The simplest use of the `input` function assigns a string to a variable:

```
x = input()
```

The parentheses are empty because, the `input` function does not require any information to do its job. Listing 2.11 (`usinginput.py`) demonstrates that the `input` function produces a string value.

> **Listing 2.11: `usinginput.py`**

```python
1  print('Please enter some text:')
2  x = input()
3  print('Text entered:', x)
4  print('Type:', type(x))
```

The following shows a sample run of Listing 2.11 (`usinginput.py`):

```
Please enter some text:
My name is Rick
Text entered: My name is Rick
Type: <class 'str'>
```

The second line shown in the output is entered by the user, and the program prints the first, third, and fourth lines. After the program prints the message *Please enter some text:*, the program's execution stops and waits for the user to type some text using the keyboard. The user can type, backspace to make changes, and type some more. The text the user types is not committed until the user presses the enter (or return) key.

Quite often we want to perform calculations and need to get numbers from the user. The `input` function produces only strings, but we can use the `int` function to convert a properly formed string of digits into an integer. Listing 2.12 (`addintegers.py`) shows how to obtain an integer from the user.

> **Listing 2.12: `addintegers.py`**

```python
1  print('Please enter an integer value:')
2  x = input()
3  print('Please enter another integer value:')
4  y = input()
5  num1 = int(x)
6  num2 = int(y)
7  print(num1, '+', num2, '=', num1 + num2)
```

A sample run of Listing 2.12 (`addintegers.py`) shows

```
Please enter an integer value:
2
Please enter another integer value:
17
2 + 17 = 19
```

Lines two and four represent user input, while the program generates the other lines. The program halts after printing the first line and does continue until the user provides the input. After the program prints the second message it again pauses to accept the user's second entry.

Since user input almost always requires a message to the user about the expected input, the input function optionally accepts a string that it prints just before the program stops to wait for the user to respond. The statement

```python
x = input('Please enter some text: ')
```

prints the message *Please enter some text:* and then waits to receive the user's input to assign to x. Listing 2.12 (addintegers.py) can be expressed more compactly using this form of the input function as shown in Listing 2.13 (addintegers2.py).

Listing 2.13: addintegers2.py

```python
1  x = input('Please enter an integer value: ')
2  y = input('Please enter another integer value: ')
3  num1 = int(x)
4  num2 = int(y)
5  print(num1, '+', num2, '=', num1 + num2)
```

Listing 2.14 (addintegers3.py) is even shorter. It combines the input and int functions into one statement.

Listing 2.14: addintegers3.py

```python
1  num1 = int(input('Please enter an integer value: '))
2  num2 = int(input('Please enter another integer value: '))
3  print(num1, '+', num2, '=', num1 + num2)
```

In Listing 2.14 (addintegers3.py) the expression

```python
int(input('Please enter an integer value: '))
```

uses a technique known as *functional composition*. The result of the input function is passed directly to the int function instead of using the intermediate variables shown in Listing 2.13 (addintegers2.py). We frequently will use functional composition to make our program code simpler.

2.7 The eval Function

The input function produces a string from the user's keyboard input. If we wish to treat that input as a number, we can use the int or float function to make the necessary conversion:

```python
x = float(input('Please enter a number'))
```

Here, whether the user enters 2 or 2.0, x will be a variable with type floating point. What if we wish x to be of type integer if the user enters 2 and x to be floating point if the user enters 2.0? Python provides the eval function that attempts to evaluate a string in the same way that the interactive shell would evaluate it. Listing 2.15 (evalfunc.py) illustrates the use of eval.

Listing 2.15: `evalfunc.py`

```
1  x1 = eval(input('Entry x1? '))
2  print('x1 =', x1, ' type:', type(x1))
3
4  x2 = eval(input('Entry x2? '))
5  print('x2 =', x2, ' type:', type(x2))
6
7  x3 = eval(input('Entry x3? '))
8  print('x3 =', x3, ' type:', type(x3))
9
10  x4 = eval(input('Entry x4? '))
11  print('x4 =', x4, ' type:', type(x4))
12
13  x5 = eval(input('Entry x5? '))
14  print('x5 =', x5, ' type:', type(x5))
```

A sample run of Listing 2.15 (evalfunc.py) produces

```
Entry x1? 4
x1 = 4  type: <class 'int'>
Entry x2? 4.0
x2 = 4.0  type: <class 'float'>
Entry x3? 'x1'
x3 = x1  type: <class 'str'>
Entry x4? x1
x4 = 4  type: <class 'int'>
Entry x5? x6
Traceback (most recent call last):
  File "C:\Users\rick\Documents\Code\Other\python\changeable.py", line 13,
    x5 = eval(input('Entry x5? '))
  File "<string>", line 1, in <module>
NameError: name 'x6' is not defined
```

Notice that when the user enters 4, the variable's type is integer. When the user enters 4.0, the variable is a floating-point variable. For x3, the user supplies the string 'x3' (note the quotes), and the variable's type is string. The more interesting situation is x4. The user enters x1 (no quotes). The eval function evaluates the non-quoted text as a reference to the name x1. The program bound the name x1 to the value 4 when executing the first line of the program. Finally, the user enters x6 (no quotes). Since the quotes are missing, the eval function does not interpret x6 as a literal string; instead eval treats x6 as a name an attempts to evaluate it. Since no variable named x6 exists, the eval function prints an error message.

The eval function dynamically translates the text provided by the user into an executable form that the program can process. This allows users to provide input in a variety of flexible ways; for example, users can enter multiple entries separated by commas, and the eval function evaluates it as a Python tuple. As Listing 2.16 (addintegers4.py) shows, this makes tuple assignment (see Section 2.2) possible.

Listing 2.16: `addintegers4.py`

```
num1, num2 = eval(input('Please enter number 1, number 2: '))
print(num1, '+', num2, '=', num1 + num2)
```

The following sample run shows how the user now must enter the two numbers at the same time separated by a comma:

```
Please enter number 1, number 2: 23, 10
23 + 10 = 33
```

Listing 2.17 (`enterarith.py`) is a simple, one line Python program that behaves like the IDLE inter-active shell, except that it accepts only one expression from the user.

Listing 2.17: `enterarith.py`

```
print(eval(input()))
```

A sample run of Listing 2.17 (`enterarith.py`) shows that the user may enter an arithmetic expression, and `eval` handles it properly:

```
4 + 10
14
```

The users enters the text 4 + 10, and the program prints 14. Notice that the addition is not programmed into Listing 2.17 (`enterarith.py`); as the program runs the `eval` function compiles the user-supplied text into executable code and executes it to produce 14.

2.8 Controlling the `print` Function

In Listing 2.12 (`addintegers.py`) we would prefer that the cursor remain at the end of the printed line so when the user types a value it appears on the same line as the message prompting for the values. When the user presses the enter key to complete the input, the cursor automatically will move down to the next line. The `print` function as we have seen so far always prints a line of text, and then the cursor moves down to the next line so any future printing appears on the next line. The `print` statement accepts an additional argument that allows the cursor to remain on the same line as the printed text:

```
print('Please enter an integer value:', end='')
```

The expression `end=''` is known as a *keyword argument*. The term keyword here means something dif-ferent from the term *keyword* used to mean a *reserved word*. We defer a complete explanation of keyword arguments until we have explored more of the Python language. For now it is sufficient to know that a `print` function call of this form will cause the cursor to remain on the same line as the printed text. Without this keyword argument, the cursor moves down to the next line after printing the text.

The print statement

```
print('Please enter an integer value: ', end='')
```

means "Print the message *Please enter an integer value:*, and then terminate the line with nothing rather than the normal \n newline code." Another way to achieve the same result is

```
print(end='Please enter an integer value: ')
```

This statement means "Print nothing, and then terminate the line with the string 'Please enter an integer value:' rather than the normal \n newline code. The behavior of the two statements is indistinguishable.

The statement

```
print('Please enter an integer value:')
```

is an abbreviated form of the statement

```
print('Please enter an integer value:', end='\n')
```

that is, the default ending for a line of printed text is the string '\n', the newline control code. Similarly, the statement

```
print()
```

is a shorter way to express
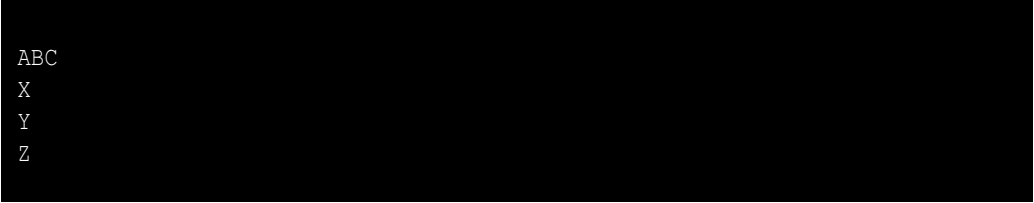
```
print(end='\n')
```

Observe closely the output of Listing 2.18 (printingexample.py).

Listing 2.18: **printingexample.py**

```
1  print('A', end='')
2  print('B', end='')
3  print('C', end='')
4  print()
5  print('X')
6  print('Y')
7  print('Z')
```

Listing 2.18 (printingexample.py) displays

```
ABC
X
Y
Z
```

The statement

```
print()
```

essentially moves the cursor down to next line.

Sometimes it is convenient to divide the output of a single line of printed text over several Python statements. As an example, we may want to compute part of a complicated calculation, print an intermediate

result, finish the calculation, and print the final answer with the output all appearing on one line of text. The `end` keyword argument allows us to do so.

Another keyword argument allows us to control how the `print` function visually separates the arguments it displays. By default, the `print` function places a single space in between the items it prints. `print` uses a keyword argument named `sep` to specify the string to use insert between items. The name `sep` stands for *separator*. The default value of `sep` is the string `' '`, a string containing a single space. Listing 2.19 (`printsep.py`) shows the `sep` keyword customizes `print`'s behavior.

Listing 2.19: `printsep.py`

```
1  w, x, y, z = 10, 15, 20, 25
2  print(w, x, y, z)
3  print(w, x, y, z, sep=',')
4  print(w, x, y, z, sep='')
5  print(w, x, y, z, sep=':')
6  print(w, x, y, z, sep='-----')
```

The output of Listing 2.19 (`printsep.py`) is

```
10 15 20 25
10,15,20,25
10152025
10:15:20:25
10-----15-----20-----25
```

The first of the output shows `print`'s default method of using a single space between printed items. The second output line uses commas as separators. The third line runs the items together with an empty string separator. The fifth line shows that the separating string may consist of multiple characters.

2.9 Summary

- Python supports both integer and floating-point kinds of numeric values and variables.

- Python does not permit commas to be used when expressing numeric literals.

- Numbers represented on a computer have limitations based on the finite nature of computer systems.

- Variables are used to store values.

- The = operator means *assignment*, not mathematical *equality*.

- A variable can be reassigned at any time.

- A variable must be assigned before it can be used within a program.

- Multiple variables can be assigned in one statement.

- A variable represents a location in memory capable of storing a value.

- The statement `a = b` copies the value stored in variable `b` into variable `a`.

- A variable name is an example of an identifier.

- The name of a variable must follow the identifier naming rules.

- All identifiers must consist of at least one character. The first symbol must be an alphabetic letter or the underscore. Remaining symbols (if any) must be alphabetic letters, the underscore, or digits.

- Reserved words have special meaning within a Python program and cannot be used as identifiers.

- Descriptive variable names are preferred over one-letter names.

- Python is case sensitive; the name X is not the same as the name x.

- Floating-point numbers approximate mathematical real numbers.

- There are many values that floating-point numbers cannot represent exactly.

- Scientific notation literals of the form 1.0×10^1 can be expressed in C++ as 1.0e1.0.

- Strings are sequences of characters.

- String literals appear within single quote marks (') or double quote marks (").

- Special non-printable control codes like newline and tab are prefixed with the backslash escape character (\).

- The \n character represents a newline.

- The literal blackslash character is a string must appear as two successive blackslash symbols.

- The input function reads in a string of text entered by the user from the keyboard during the program's execution.

- The input function accepts an optional prompt string.

- The eval function can be used to convert a string representing a numeric expression into its evaluated numeric value.

2.10 Exercises

1. Will the following lines of code print the same thing? Explain why or why not.

```
x = 6
print(6)
print("6")
```

2. Will the following lines of code print the same thing? Explain why or why not.

```
x = 7
print(x)
print("x")
```

3. What is the largest floating-point value available on your system?

4. What is the smallest floating-point value available on your system?

5. What happens if you attempt to use a variable within a program, and that variable has not been assigned a value?

6. What is wrong with the following statement that attempts to assign the value ten to variable `x`?

   10 = x

7. Once a variable has been properly assigned can its value be changed?

8. In Python can you assign more than one variable in a single statement?

9. Classify each of the following as either a *legal* or *illegal* Python identifier:

   (a) `fred`

   (b) `if`

   (c) `2x`

   (d) `-4`

   (e) `sum_total`

   (f) `sumTotal`

   (g) `sum-total`

   (h) `sum total`

   (i) `sumtotal`

   (j) `While`

   (k) `x2`

   (l) `Private`

   (m) `public`

   (n) `$16`

   (o) `xTwo`

   (p) `_static`

   (q) `_4`

   (r) `___`

   (s) `10%`

   (t) `a27834`

   (u) `wilma's`

10. What can you do if a variable name you would like to use is the same as a reserved word?

11. How is the value 2.45×10^{-5} expressed as a Python literal?

12. How is the value 0.0000000000000000000000000449 expressed as a Python literal?

13. How is the value 56992341200000000000000000000000000000 expressed as a Python literal?

14. Can a Python programmer do anything to ensure that a variable's value can never be changed after its initial assignment?

15. Is `"i"` a string literal or variable?

16. What is the difference between the following two strings? `'n'` and `'\n'`?

17. Write a Python program containing exactly one `print` statement that produces the following output:

```
A
B
C
D
E
F
```

18. Write a Python program that simply emits a beep sound when run.

Chapter 3

Expressions and Arithmetic

This chapter uses the Python numeric types introduced in Chapter 2 to build expressions and perform arithmetic. Some other important concepts are covered—user input, comments, and dealing with errors.

3.1 Expressions

A literal value like 34 and a variable like x are examples of a simple *expressions*. Values and variables can be combined with operators to form more complex expressions. In Section 2.1 we saw how we can use the + operator to add integers and concatenate strings. Listing 3.1 (adder.py) shows how the addition operator (+) can used to add two integers provided by the user.

Listing 3.1: adder.py

```
1  value1 = eval(input('Please enter a number: '))
2  value2 = eval(input('Please enter another number: '))
3  sum = value1 + value2
4  print(value1, '+', value2,  '=', sum)
```

To review, in Listing 3.1 (adder.py):

- value1 = eval(input('Please enter a number: '))

  This statement prompts the user to enter some information. After displaying the prompt string *Please enter an integer value:*, this statement causes the program's execution to stop and wait for the user to type in some text and then press the enter key. The string produced by the input function is passed off to the eval function which produces a value to assign to the variable value1. If the user types the sequence *431* and then presses the enter key, value1 is assigned the integer 431. If instead the user enters *23 + 3*, the variable gets the value 26.

- value2 = eval(input('Please enter another number: '))

  This statement is similar to the first statement.

- sum = value1 + value2;

  This is an assignment statement because is contains the assignment operator (=). The variable sum appears to the left of the assignment operator, so sum will receive a value when this statement executes. To the right of the assignment operator is an arithmetic expression involving two variables and

Expression	Meaning
x + *y*	*x* added to *y*, if *x* and *y* are numbers
	x concatenated to *y*, if *x* and *y* are strings
x − *y*	*x* take away *y*, if *x* and *y* are numbers
x * *y*	*x* times *y*, if *x* and *y* are numbers
	x concatenated with itself *y* times, if *x* is a string and *y* is an integer
	y concatenated with itself *x* times, if *y* is a string and *x* is an integer
x / *y*	*x* divided by *y*, if *x* and *y* are numbers
x // *y*	Floor of *x* divided by *y*, if *x* and *y* are numbers
x % *y*	Remainder of *x* divided by *y*, if *x* and *y* are numbers
x ** *y*	*x* raised to *y* power, if *x* and *y* are numbers

Table 3.1: Commonly used Python arithmetic binary operators

the addition operator. The expression is *evaluated* by adding together the values bound to the two variables. Once the addition expression's value has been determined, that value can be assigned to the `sum` variable.

- `print(value1, '+', value2,  '=', sum)`

  This statement prints the values of the three variables with some additional decoration to make the output clear about what it is showing.

All expressions have a value. The process of determining the expression's value is called *evaluation*. Evaluating simple expressions is easy. The literal value 54 evaluates to 54. The value of a variable named x is the value stored in the memory location bound to x. The value of a more complex expression is found by evaluating the smaller expressions that make it up and combining them with operators to form potentially new values.

The commonly used Python arithmetic operators are found in Table 3.1. The common arithmetic operations, addition, subtraction, multiplication, division, and power behave in the expected way. The // and % operators are not common arithmetic operators in everyday practice, but they are very useful in programming. The // operator is called *integer division*, and the % operator is the *modulus* or *remainder* operator. 25/3 is 8.3333. Three does not divide into 25 evenly. In fact, three goes into 25 eight times with a remainder of one. Here, eight is the quotient, and one is the remainder. 25//3 is 8 (the quotient), and 25%3 is 1 (the remainder).

All these operators are classified as *binary* operators because they operate on two operands. In the statement

x = y + z;

on the right side of the assignment operator is an addition expression y + z. The two operands of the + operator are y and z.

Two operators, + and −, can be used as *unary* operators. A unary operator has only one operand. The − unary operator expects a single numeric expression (literal number, variable, or more complicated numeric expression within parentheses) immediately to its right; it computes the *additive inverse* of its operand. If the operand is positive (greater than zero), the result is a negative value of the same magnitude; if the operand is negative (less than zero), the result is a positive value of the same magnitude. Zero is unaffected. For example, the following code sequence

```
x, y, z = 3, -4, 0
x = -x
y = -y
z = -z
print(x, y, z)
```

within a program would print

```
-3 4 0
```

The following statement

```
print(-(4 - 5))
```

within a program would print

```
1
```

The unary + operator is present only for completeness; when applied to a numeric value, variable, or expression, the resulting value is no different from the original value of its operand. Omitting the unary + operator from the following statement

```
x = +y
```

does not change its behavior.

All the arithmetic operators are subject to the limitations of the data types on which they operate; for example, consider the following interaction sequence:

```
>>> 2.0**10
1024.0
>>> 2.0**100
1.2676506002282294e+30
>>> 2.0**1000
1.0715086071862673e+301
>>> 2.0**10000
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Result too large')
```

The expression `2.0**10000` will not evaluate to the correct answer since the correct answer falls outside the range of floating point values.

When we apply the +, -, *, //, %, or ** operators to two integers, the result is an integer. The / operator applied to two integers produces a floating-point result. The statement

```python
print(10/3, 3/10, 10//3, 3//10)
```

prints

```
3.3333333333333335 0.3 3 0
```

The first two results are the same that a hand-held calculator would produce. The second two results use integer division, and three goes into 10 three times, while 10 goes into 3 zero times. The // operator produces an integer result when used with integers because in the first case 10 divided by 3 is 3 with a remainder of 1, and in the second case 3 divided by 10 is 0 with a remainder of 3. Since integers are whole numbers, any fractional part of the answer must be discarded. The process of discarding the fractional part leaving only the whole number part is called *truncation*. Truncation is not rounding; for example, 11/3 is 3.6666..., but 11//3 truncates to 3. ¡warning¿ Truncation simply removes any fractional part of the value. It does not round. Both 10.01 and 10.999 truncate to 10. ¡/warning¿

The modulus operator (%) computes the remainder of integer division; thus,

```python
print(10%3, 3%10)
```

prints

```
1 3
```

since 10 divided by 3 is 3 with a remainder of 1, and 3 divided by 10 is 0 with a remainder of 3.

The modulus operator is more useful than it may first appear. Listing 3.7 (timeconv.py) shows how it can be used to convert a given number of seconds to hours, minutes, and seconds.

Floating-point arithmetic always produces a floating-point result.

```python
print(10.0/3.0, 3.0/10.0, 10.0//3.0, 3//10.0)
```

prints

```
3.3333333333333335 0.3 3.0 0.0
```

Recall from Section 2.4 that integers can be represented exactly, but floating-point numbers are imprecise approximations of real numbers. Listing 3.2 (imprecise.py) clearly demonstrates the weakness of floating point numbers.

Listing 3.2: imprecise.py
```python
one = 1.0
one_third = 1.0/3.0
zero = one - one_third - one_third - one_third

print('one =', one, ' one_third =', one_third, ' zero =', zero)
```

```
one = 1.0   one_third = 0.3333333333333333   zero = 1.1102230246251565e-16
```

The reported result is $1.1102230246251565 \times 10^{-16}$, or 0.00000000000000011102230246251565, While this number is very small, with real numbers we get

$$1 - \frac{1}{3} - \frac{1}{3} - \frac{1}{3} = 0$$

Floating-point numbers are not real numbers, so the result of 1.0/3.0 cannot be represented exactly without infinite precision. In the decimal (base 10) number system, one-third is a repeating fraction, so it has an infinite number of digits. Even simple non-repeating decimal numbers can be a problem. One-tenth (0.1) is obviously non-repeating, so it can be expressed exactly with a finite number of digits. As it turns out, since numbers within computers are stored in binary (base 2) form, even one-tenth cannot be represented exactly with floating-point numbers, as Listing 3.3 (imprecise10.py) illustrates.

Listing 3.3: **imprecise10.py**

```python
one = 1.0
one_tenth = 1.0/10.0
zero = one - one_tenth - one_tenth - one_tenth \
          - one_tenth - one_tenth - one_tenth \
          - one_tenth - one_tenth - one_tenth \
          - one_tenth

print('one =', one, ' one_tenth =', one_tenth, ' zero =', zero)
```

The program's output is

```
one = 1.0   one_tenth = 0.1   zero = 1.3877787807814457e-16
```

Surely the reported answer ($1.3877787807814457 \times 10^{-16}$) is close to the correct answer (zero). If you round our answer to the one-hundred trillionth place (15 places behind the decimal point), it is correct.

In Listing 3.3 (imprecise10.py) lines 3–6 make up a single Python statement. If that single statement that performs nine subtractions were written on one line, it would flow well off the page or off the editing window. Ordinarily a Python statement ends at the end of the source code line. A programmer may break up a very long line over two or more lines by using the backslash (\) symbol at the end of an incomplete line. When the interpreter is processing a line that ends with a \, it automatically joins the line that follows. The interpreter thus sees a very long but complete Python statement.

Despite their inexactness, floating-point numbers are used every day throughout the world to solve sophisticated scientific and engineering problems. The limitations of floating-point numbers are unavoidable since values with infinite characteristics cannot be represented in a finite way. Floating-point numbers provide a good trade-off of precision for practicality.

Expressions may contain mixed elements; for example, in the following program fragment

```python
x = 4
y = 10.2
sum = x + y
```

x is an integer and y is a floating-point number. What type is the expression x + y? Except in the case of the / operator, arithmetic expressions that involve only integers produce an integer result. All arithmetic operators applied to floating-point numbers produce a floating-point result. When an operator has mixed operands—one operand an integer and the other a floating-point number—the interpreter treats the integer operand as floating-point number and performs floating-point arithmetic. This means x + y is a floating-point expression, and the assignment will make sum a floating-point variable.

3.2 Operator Precedence and Associativity

When different operators appear in the same expression, the normal rules of arithmetic apply. All Python operators have a *precedence* and *associativity*:

- **Precedence**—when an expression contains two different kinds of operators, which should be applied first?

- **Associativity**—when an expression contains two operators with the same precedence, which should be applied first?

To see how precedence works, consider the expression

```
2 + 3 * 4
```

Should it be interpreted as

```
(2 + 3) * 4
```

(that is, 20), or rather is

```
2 + (3 * 4)
```

(that is, 14) the correct interpretation? As in normal arithmetic, multiplication and division in Python have equal importance and are performed before addition and subtraction. We say multiplication and division have precedence over addition and subtraction. In the expression

```
2 + 3 * 4
```

the multiplication is performed before addition, since multiplication has precedence over addition. The result is 14. The multiplicative operators (*, /, //, and %) have equal precedence with each other, and the additive operators (binary + and -) have equal precedence with each other. The multiplicative operators have precedence over the additive operators.

As in standard arithmetic, in Python if the addition is to be performed first, parentheses can override the precedence rules. The expression

```
(2 + 3) * 4
```

evaluates to 20. Multiple sets of parentheses can be arranged and nested in any ways that are acceptable in standard arithmetic.

To see how associativity works, consider the expression

```
2 - 3 - 4
```

Arity	Operators	Associativity
Unary	+, –	
Binary	*, /, %	Left
Binary	+, –	Left
Binary	=	Right

Table 3.2: Operator precedence and associativity. The operators in each row have a higher precedence than the operators below it. Operators within a row have the same precedence.

The two operators are the same, so they have equal precedence. Should the first subtraction operator be applied before the second, as in

```
(2 - 3) - 4
```

(that is, -5), or rather is

```
2 - (3 - 4)
```

(that is, 3) the correct interpretation? The former (-5) is the correct interpretation. We say that the subtraction operator is *left associative*, and the evaluation is left to right. This interpretation agrees with standard arithmetic rules. All binary operators except assignment are left associative.

The assignment operator supports a technique known as *chained assignment*. The code

```
w = x = y = z
```

should be read right to left. First y gets the value of z, then y gets z's value as well. Both w and x get z's value also.

As in the case of precedence, parentheses can be used to override the natural associativity within an expression.

The unary operators have a higher precedence than the binary operators, and the unary operators are right associative. This means the statements

```
print(-3 + 2)
print(-(3 + 2))
```

which display

```
-1
-5
```

behave as expected.

Table 3.2 shows the precedence and associativity rules for some Python operators.

3.3 Comments

Good programmers annotate their code by inserting remarks that explain the purpose of a section of code or why they chose to write a section of code the way they did. These notes are meant for human readers, not

the interpreter. It is common in industry for programs to be reviewed for correctness by other programmers or technical managers. Well-chosen identifiers (see Section 2.3) and comments can aid this assessment process. Also, in practice, teams of programmers develop software. A different programmer may be required to finish or fix a part of the program written by someone else. Well-written comments can help others understand new code quicker and increase their productivity modifying old or unfinished code. While it may seem difficult to believe, even the same programmer working on her own code months later can have a difficult time remembering what various parts do. Comments can help greatly.

Any text contained within comments is ignored by the Python interpreter. The # symbol begins a comment in the source code. The comment is in effect until the end of the line of code:

```python
# Compute the average of the values
avg = sum / number
```

The first line here is a comment that explains what the statement that follows it is supposed to do. The comment begins with the # symbol and continues until the end of that line. The interpreter will ignore the # symbol and the contents of the rest of the line. You also may append a short comment to the end of a statement:

```python
avg = sum / number   # Compute the average of the values
```

Here, an executable statement and the comment appear on the same line. The interpreter will read the assignment statement, but it will ignore the comment.

How are comments best used? Avoid making a remark about the obvious; for example:

```python
result = 0   # Assign the value zero to the variable named result
```

The effect of this statement is clear to anyone with even minimal Python programming experience. Thus, the audience of the comments should be taken into account; generally, "routine" activities require no remarks. Even though the *effect* of the above statement is clear, its *purpose* may need a comment. For example:

```python
result = 0   # Ensures 'result' has a well-defined minimum value
```

This remark may be crucial for readers to completely understand how a particular part of a program works. In general, programmers are not prone to providing too many comments. When in doubt, add a remark. The extra time it takes to write good comments is well worth the effort.

3.4 Errors

Beginning programmers make mistakes writing programs because of inexperience in programming in general or because of unfamiliarity with a programming language. Seasoned programmers make mistakes due to carelessness or because the proposed solution to a problem is faulty and the correct implementation of an incorrect solution will not produce a correct program.

In Python, there are three general kinds of errors: syntax errors, run-time errors, and logic errors.

3.4.1 Syntax Errors

The interpreter is designed to execute all valid Python programs. The interpreter reads the Python source code and translates it into executable machine code. This is the *translation phase*. If the interpreter detects an invalid program during the translation phase, it will terminate the program's execution and report an

error. Such errors result from the programmer's misuse of the language. A *syntax error* is a common error that the interpreter can detect when attempting to translate a Python statement into machine language. For example, in English one can say

> The boy walks quickly.

This sentence uses correct syntax. However, the sentence

> The boy walk quickly.

is not correct syntactically: the number of the subject (singular form) disagrees with the number of the verb (plural form). It contains a syntax error. It violates a grammatical rule of the English language. Similarly, the Python statement

```
x = y + 2
```

is syntactically correct because it obeys the rules for the structure of an assignment statement described in Section 2.2. However, consider replacing this assignment statement with a slightly modified version:

```
y + 2 = x
```

If a statement like this one appears in a program, the interpreter will issue an error message; for example, if the statement appears on line 12 of an otherwise correct Python program described in a file named error.py, the interpreter reports:

```
>>> y + 2 = x
  File "error.py", line 12
SyntaxError: can't assign to operator
```

The syntax of Python does not allow an expression like `y + 2` to appear on the left side of the assignment operator.

Other common syntax errors arise from simple typographical errors like mismatched parentheses or string quotes or faulty indentation.

3.4.2 Run-time Errors

A syntactically correct Python program still can have problems. Some language errors depend on the context of the program's execution. Such errors are called *run-time errors* or *exceptions*. Run-time errors arise after the interpreter's translation phase and during its execution phase.

The interpreter may issue an error for a syntactically correct statement like

```
x = y + 2
```

if the variable `y` has yet to be assigned; for example, if the statement appears at line 12 and by that point `y` has not been assigned, we are informed:

```
>>> x = y + 2
Traceback (most recent call last):
  File "error.py", line 12, in <module>
NameError: name 'y' is not defined
```

Consider Listing 3.4 (`dividedanger.py`) which contains an error that manifests itself only in one particular situation.

Listing 3.4: `dividedanger.py`

```
1  #  File dividedanger.py
2
3  #  Get two integers from the user
4  dividend, divisor = eval(input('Please enter two numbers to divide: '))
5  #  Divide them and report the result
6  print(dividend, '/', divisor, "=", dividend/divisor)
```

The expression

dividend/divisor

is potentially dangerous. If the user enters, for example, 32 and 4, the program works nicely

```
Please enter two integers to divide: 32, 4
32 / 4 = 8.0
```

If the user instead types the numbers 32 and 0, the program reports an error and terminates:

```
Please enter two numbers to divide: 32, 0
Traceback (most recent call last):
  File "C:\Users\rick\Desktop\changeable.py", line 6, in <module>
    print(dividend, '/', divisor, "=", dividend/divisor)
ZeroDivisionError: division by zero
```

Division by zero is undefined in mathematics, and division by zero in Python is illegal.

As another example, consider Listing 3.5 (`halve.py`).

Listing 3.5: `halve.py`

```
1  #  Get a number from the user
2  value = eval(input('Please enter a number to cut in half: '))
3  #  Report the result
4  print(value/2)
```

Some sample runs of Listing 3.5 (`halve.py`) reveal

```
Please enter a number to cut in half: 100
50.0
```

and

```
Please enter a number to cut in half: 19.41
9.705
```

So far, so good, but what if the user does not follow the on-screen instructions?

```
Please enter a number to cut in half: Bobby
Traceback (most recent call last):
  File "C:\Users\rick\Desktop\changeable.py", line 122, in <module>
    value = eval(input('Please enter a number to cut in half: '))
  File "<string>", line 1, in <module>
NameError: name 'Bobby' is not defined
```

or

```
Please enter a number to cut in half: 'Bobby'
Traceback (most recent call last):
  File "C:\Users\rick\Desktop\changeable.py", line 124, in <module>
    print(value/2)
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Since the programmer cannot predict what the user will provide as input, this program is doomed eventually. Fortunately, in Chapter **??** we will examine techniques that allow programmers to avoid these kinds of problems.

The interpreter detects syntax errors immediately. The program never makes it out of the translation phase. Sometimes run-time errors do not reveal themselves immediately. The interpreter issues a run-time error only when it attempts to execute the statement with the problem. In Chapter 4 we will see how to write programs that optionally execute some statements only under certain conditions. If those conditions do not arise during testing, the faulty code is not executed. This means the error may lie undetected until a user stumbles upon it after the software is deployed. Run-time errors, therefore, are more troublesome than syntax errors.

3.4.3 Logic Errors

The interpreter can detect syntax errors during the translation phase and run-time errors during the execution phase. Both represent violations of the Python language. Such errors are the easiest to repair, because the interpreter indicates the exact location within the source code where it detected the problem.

Consider the effects of replacing the expression

dividend/divisor;

in Listing 3.4 (`dividedanger.py`) with the expression:

divisor/dividend;

The program runs, and unless a value of zero is entered for the dividend, the interpreter will report no errors. However, the answer it computes is not correct in general. The only time the correct answer is printed is when `dividend = divisor`. The program contains an error, but the interpreter is unable detect the problem. An error of this type is known as a *logic error*.

Listing 3.9 (`faultytempconv.py`) is an example of a program that contains a logic error. Listing 3.9 (`faultytempconv.py`) runs without the interpreter reporting any errors, but it produces incorrect results.

Beginning programmers tend to struggle early on with syntax and run-time errors due to their unfamiliarity with the language. The interpreter's error messages are actually the programmer's best friend. As the programmer gains experience with the language and the programs written become more complicated, the number of non-logic errors decrease or are trivially fixed and the number of logic errors increase. Unfortunately, the interpreter is powerless to provide any insight into the nature and location of logic errors. Logic errors, therefore, tend to be the most difficult to find and repair. Tools such as debuggers frequently are used to help locate and fix logic errors, but these tools are far from automatic in their operation.

Undiscovered run-time errors and logic errors that lurk in software are commonly called *bugs*. The interpreter reports execution errors only when the conditions are right that reveal those errors. The interpreter is of no help at all with logic errors. Such bugs are the major source of frustration for developers. The frustration often arises because in complex programs the bugs sometimes only reveal themselves in certain situations that are difficult to reproduce exactly during testing. You will discover this frustration as your programs become more complicated. The good news is that programming experience and the disciplined application of good programming techniques can help reduce the number logic errors. The bad news is that since software development in an inherently human intellectual pursuit, logic errors are inevitable. Accidentally introducing and later finding and eliminating logic errors is an integral part of the programming process.

3.5 Arithmetic Examples

Suppose we wish to convert temperature from degrees Fahrenheit to degrees Celsius. The following formula provides the necessary mathematics:

$$^{\circ}C = \frac{5}{9} \times (^{\circ}F - 32)$$

Listing 3.6 (`tempconv.py`) implements the conversion in Python.

Listing 3.6: `tempconv.py`

```
1   #  File tempconv.py
2   #  Author: Rick Halterman
3   #  Last modified: August 22, 2011
4   #  Converts degrees Fahrenheit to degrees Celsius
5   #  Based on the formula found at
6   #  http://en.wikipedia.org/wiki/Conversion_of_units_of_temperature
7
8   #  Prompt user for temperature to convert and read the supplied value
```

```
 9  degreesF = eval(input('Enter the temperature in degrees F: '))
10  #  Perform the conversion
11  degreesC = 5/9*(degreesF - 32);
12  #  Report the result
13  print(degreesF, "degrees F =', degreesC, 'degrees C')
```

Listing 3.6 (`tempconv.py`) contains comments that give an overview of the program's purpose and provide some details about its construction. Comments also document each step explaining the code's logic. Some sample runs show how the program behaves:

```
Enter the temperature in degrees F: 212
212 degrees F = 100.0 degrees C
```

```
Enter the temperature in degrees F: 32
32 degrees F = 0.0 degrees C
```

```
Enter the temperature in degrees F: -40
-40 degrees F = -40.0 degrees C
```

Listing 3.7 (`timeconv.py`) uses integer division and modulus to split up a given number of seconds to hours, minutes, and seconds.

Listing 3.7: `timeconv.py`

```
 1   #  File timeconv.py
 2
 3  #  Get the number of seconds
 4  seconds = eval(input("Please enter the number of seconds:"))
 5  #  First, compute the number of hours in the given number of seconds
 6  #  Note: integer division with possible truncation
 7  hours = seconds // 3600   #  3600 seconds = 1 hours
 8  #  Compute the remaining seconds after the hours are accounted for
 9  seconds = seconds % 3600
10  #  Next, compute the number of minutes in the remaining number of seconds
11  minutes = seconds // 60    #  60 seconds = 1 minute
12  #  Compute the remaining seconds after the minutes are accounted for
13  seconds = seconds % 60
14  #  Report the results
15  print(hours, "hr,", minutes, "min,", seconds, "sec")
```

If the user enters `10000`, the program prints `2 hr, 46 min, 40 sec`. Notice the assignments to the `seconds` variable, such as

seconds = seconds % 3600

The right side of the assignment operator (=) is first evaluated. The remainder of seconds divided by 3,600 is assigned back to seconds. This statement can alter the value of seconds if the current value of seconds is greater than 3,600. A similar statement that occurs frequently in programs is one like

```
x = x + 1
```

This statement increments the variable x to make it one bigger. A statement like this one provides further evidence that the Python assignment operator does not mean mathematical equality. The following statement from mathematics

$$x = x + 1$$

surely is never true; a number cannot be equal to one more than itself. If that were the case, I would deposit one dollar in the bank and then insist that I really had two dollars in the bank, since a number is equal to one more than itself. That two dollars would become $3.00, then $4.00, etc., and soon I would be rich. In Python, however, this statement simply means "add one to x's current value and update x with the result."

A variation on Listing 3.7 (timeconv.py), Listing 3.8 (enhancedtimeconv.py) performs the same logic to compute the time components (hours, minutes, and seconds), but it uses simpler arithmetic to produce a slightly different output—instead of printing 11,045 seconds as 3 hr, 4 min, 5 sec, Listing 3.8 (enhancedtimeconv.py) displays it as 3:04:05. It is trivial to modify Listing 3.7 (timeconv.py) so that it would print 3:4:5, but Listing 3.8 (enhancedtimeconv.py) includes some extra arithmetic to put leading zeroes in front of single-digit values for minutes and seconds as is done on digital clock displays.

Listing 3.8: enhancedtimeconv.py

```python
# File enhancedtimeconv.py

#  Get the number of seconds
seconds = eval(input("Please enter the number of seconds:"))
#  First, compute the number of hours in the given number of seconds
#  Note: integer division with possible truncation
hours = seconds // 3600   #  3600 seconds = 1 hours
#  Compute the remaining seconds after the hours are accounted for
seconds = seconds % 3600
#  Next, compute the number of minutes in the remaining number of seconds
minutes = seconds // 60    #  60 seconds = 1 minute
#  Compute the remaining seconds after the minutes are accounted for
seconds = seconds % 60
#  Report the results
print(hours, ":", sep="", end="")
#  Compute tens digit of minutes
tens = minutes // 10
#  Compute ones digit of minutes
ones = minutes % 10
print(tens, ones, ":", sep="", end="")
#  Compute tens digit of seconds
tens = seconds // 10
#  Compute ones digit of seconds
ones = seconds % 10
print(tens, ones, sep ="")
```

Listing 3.8 (enhancedtimeconv.py) uses the fact that if x is a one- or two-digit number, x % 10 is the tens digit of x. If x % 10 is zero, x is necessarily a one-digit number.

3.6 More Arithmetic Operators

We will see shortly that variables are often modified in a regular way as programs execute. A variable may increase by one or decrease by five. The statement

```
x = x + 1
```

increments x by one, making it one bigger than it was before this statement was executed. Python has a shorter statement that accomplishes the same effect:

```
x += 1
```

This is the *increment* statement. A similar *decrement* statement is available:

```
x -= 1     #  Same as x = x - 1;
```

Python provides a more general way of simplifying a statement that modifies a variable through simple arithmetic. For example, the statement

```
x = x + 5
```

can be shorted to

```
x += 5
```

This statement means "increase x by five." Any statement of the form

> x *op= exp*

where

- x is a variable.

- *op=* is an arithmetic operator combined with the assignment operator; for our purposes, the ones most useful to us are +=, -=, *=, /=, //=, and %=.

- *exp* is an expression compatible with the variable x.

Arithmetic reassignment statements of this form are equivalent to

> x = x *op exp*;

This means the statement

```
x *= y + z;
```

is equivalent to

```
x = x * (y + z);
```

The version using the arithmetic assignment does not require parentheses. The arithmetic assignment is especially handy if a variable with a long name must be modified; consider

```
temporary_filename_length = temporary_filename_length / (y + z);
```

versus

```
temporary_filename_length /= y + z;
```

Do not accidentally reverse the order of the symbols for the arithmetic assignment operators, like in the statement

```
x =+ 5;
```

Notice that the + and = symbols have been reversed. The compiler interprets this statement as if it had been written

```
    x = +5;
```

that is, assignment and the unary operator. This assigns exactly five to x instead of increasing it by five.

Similarly,

```
    x =- 3;
```

would assign −3 to x instead of decreasing x by three.

3.7 Algorithms

An *algorithm* is a finite sequence of steps, each step taking a finite length of time, that solves a problem or computes a result. A computer program is one example of an algorithm, as is a recipe to make lasagna. In both of these examples, the order of the steps matter. In the case of lasagna, the noodles must be cooked in boiling water before they are layered into the filling to be baked. It would be inappropriate to place the raw noodles into the pan with all the other ingredients, bake it, and then later remove the already baked noodles to cook them in boiling water separately. In the same way, the ordering of steps is very important in a computer program. While this point may be obvious, consider the following sound argument:

1. The relationship between degrees Celsius and degrees Fahrenheit can be expressed as

$$°C = \frac{5}{9} \times (°F - 32)$$

2. Given a temperature in degrees Fahrenheit, the corresponding temperature in degrees Celsius can be computed.

Armed with this knowledge, Listing 3.9 (`faultytempconv.py`) follows directly.

Listing 3.9: **faultytempconv.py**

```
1   #  File faultytempconv.py
2
3   #  Establish some variables
4   degreesF, degreesC = 0, 0
5   #  Define the relationship between F and C
6   degreesC = 5/9*(degreesF - 32)
7   #  Prompt user for degrees F
8   degreesF = eval(input('Enter the temperature in degrees F: '))
9   #  Report the result
10  print(degreesF, "degrees F =', degreesC, 'degrees C')
```

Unfortunately, when run the program always displays

```
-17.7778
```

regardless of the input provided. The English description provided above is correct. The formula is implemented faithfully. The problem lies simply in statement ordering. The statement

```
degreesC = 5/9*(degreesF - 32);
```

is an *assignment* statement, not a definition of a relationship that exists throughout the program. At the point of the assignment, degreesF has the value of zero. The variable degreesC is assigned *before* degreesF's value is received from the user.

As another example, suppose x and y are two variables in some program. How would we interchange the values of the two variables? We want x to have y's original value and y to have x's original value. This code may seem reasonable:

```
x = y
y = x
```

The problem with this section of code is that after the first statement is executed, x and y both have the same value (y's original value). The second assignment is superfluous and does nothing to change the values of x or y. The solution requires a third variable to remember the original value of one the variables before it is reassigned. The correct code to swap the values is

```
temp = x
x = y
y = temp
```

We can use tuple assignment (see Section 2.2) to make the swap even simpler:

```
x, y = y, x
```

These small examples emphasize the fact that algorithms must be specified precisely. Informal notions about how to solve a problem can be valuable in the early stages of program design, but the coded program requires a correct detailed description of the solution.

The algorithms we have seen so far have been simple. Statement 1, followed by Statement 2, etc. until every statement in the program has been executed. Chapters 4 and **??** introduce some language constructs that permit optional and repetitive execution of some statements. These constructs allow us to build programs that do much more interesting things, but more complex algorithms are required to make it happen. We must not lose sight of the fact that a complicated algorithm that is 99% correct is *not* correct. An algorithm's design and implementation can be derailed by inattention to the smallest of details.

3.8 Summary

- The literal value 4 and integer sum are examples of simple Python numeric expressions.

- 2*x + 4 is an example of a more complex Python numeric expression.

- Expressions can be printed via the print function and be assigned to variables.

- A binary operator performs an operation using two operands.

- With regard to binary operators: + represents arithmetic addition; – represents arithmetic subtraction; * represents arithmetic multiplication; / represents arithmetic division; // represents arithmetic integer division; % represents arithmetic modulus, or integer remainder after division.

- A unary operator performs an operation using one operand.

- The – unary operator represents the additive inverse of its operand.

- The + unary operator has no effect on its operand.

- Arithmetic applied to integer operands yields integer results.

- With a binary operation, floating-point arithmetic is performed if at least one of its operands is a floating-point number.

- Floating-point arithmetic is inexact and subject to rounding errors because floating-point values have finite precision.

- A mixed expression is an expression that contains values and/or variables of differing types.

- In Python, operators have both a precedence and an associativity.

- With regard to the arithmetic operators, Python uses the same precedence rules as standard arithmetic: multiplication and division are applied before addition and subtraction unless parentheses dictate otherwise.

- The arithmetic operators associate left to right; assignment associates right to left.

- Chained assignment can be used to assign the same value to multiple variables within one statement.

- The unary operators + and – have precedence over the binary arithmetic operators *, /, //, and %, which have precedence over the binary arithmetic operators + and –, which have precedence over the assignment operator.

- Comments are notes within the source code. The interpreter ignores all comments in the source code.

- Comments inform human readers about the code.

- Comments should not state the obvious, but it is better to provide too many comments rather than too few.

- A comment begins with the symbols # and continues until the end of the line.

- Source code should be formatted so that it is more easily read and understood by humans.

- Programmers introduce syntax errors when they violate the structure of the Python language.

- The interpreter detects syntax errors during its translation phase before program execution.

- Run-time errors or exceptions are errors that are detected when the program is executing.

- The interpreter detects run-time errors during its execution phase after translation.

- Logic errors elude detection by the interpreter. Improper program behavior indicates a logic error.

- In complicated arithmetic expressions involving many operators and operands, the rules pertaining to mixed arithmetic are applied on an operator-by-operator basis, following the precedence and associativity laws, not globally over the entire expression.

- The `+=` and `-=` operators can be used to increment and decrement variables.

- The family of *op=* operators (`+=`, `-=`, `*=`, `/=`, `//=` and `%=`) allow variables to be changed by a given amount using a particular arithmetic operator.

- Python programs implement algorithms; as such, Python statements do not declare statements of fact or define relationships that hold throughout the program's execution; rather they indicate how the values of variables change as the execution of the program progresses.

3.9 Exercises

1. Is the literal `4` a valid Python expression?

2. Is the variable `x` a valid Python expression?

3. Is `x + 4` a valid Python expression?

4. What affect does the unary `+` operator have when applied to a numeric expression?

5. Sort the following binary operators in order of high to low precedence: `+`, `-`, `*`, `//`, `/`, `%`, `=`.

6. Given the following assignment:

   ```
   x = 2
   ```

   Indicate what each of the following Python statements would print.

   (a) `print("x")`
   (b) `print('x')`
   (c) `print(x)`
   (d) `print("x + 1")`
   (e) `print('x' + 1)`
   (f) `print(x + 1)`

7. Given the following assignments:

   ```
   i1 = 2
   i2 = 5
   i3 = -3
   d1 = 2.0
   d2 = 5.0
   d3 = -0.5;
   ```

   Evaluate each of the following Python expressions.

   (a) `i1 + i2`
   (b) `i1 / i2`
   (c) `i1 // i2`
   (d) `i2 / i1`
   (e) `i2 // i1`
   (f) `i1 * i3`

    (g) `d1 + d2`

    (h) `d1 / d2`

    (i) `d2 / d1`

    (j) `d3 * d1`

    (k) `d1 + i2`

    (l) `i1 / d2`

    (m) `d2 / i1`

    (n) `i2 / d1`

    (o) `i1/i2*d1`

    (p) `d1*i1/i2`

    (q) `d1/d2*i1`

    (r) `i1*d1/d2`

    (s) `i2/i1*d1`

    (t) `d1*i2/i1`

    (u) `d2/d1*i1`

    (v) `i1*d2/d1`

8. What is printed by the following statement:

```
#print(5/3)
```

9. Given the following assignments:

```
i1 = 2
i2 = 5
i3 = −3
d1 = 2.0
d2 = 5.0
d3 = −0.5
```

Evaluate each of the following Python expressions.

    (a) `i1 + (i2 * i3)`

    (b) `i1 * (i2 + i3)`

    (c) `i1 / (i2 + i3)`

    (d) `i1 // (i2 + i3)`

    (e) `i1 / i2 + i3`

    (f) `i1 // i2 + i3`

    (g) `3 + 4 + 5 / 3`

    (h) `3 + 4 + 5 // 3`

    (i) `(3 + 4 + 5) / 3`

    (j) `(3 + 4 + 5) // 3`

    (k) `d1 + (d2 * d3)`

    (l) `d1 + d2 * d3`

(m) `d1 / d2 - d3`

(n) `d1 / (d2 - d3)`

(o) `d1 + d2 + d3 / 3`

(p) `(d1 + d2 + d3) / 3`

(q) `d1 + d2 + (d3 / 3)`

(r) `3 * (d1 + d2) * (d1 - d3)`

10. What symbol signifies the beginning of a comment in Python?

11. How do Python comments end?

12. Which is better, too many comments or too few comments?

13. What is the purpose of comments?

14. Why is human readability such an important consideration?

15. Consider the following program which contains some errors. You may assume that the comments within the program accurately describe the program's intended behavior.

```
#  Get two numbers from the user
n1, n2 = eval(input())                  #  1
#  Compute sum of the two numbers
print(n1 + n2)                          #  2
#  Compute average of the two numbers
print(n1+n2/2)                          #  3
#  Assign some variables
d1 = d2 = 0                             #  4
#  Compute a quotient
print(n1/d1)                            #  5
#  Compute a product
n1*n2 = d1                              #  6
#  Print result
print(d1)                               #  7
```

For each line listed in the comments, indicate whether or not an interpreter error, run-time exception, or logic error is present. Not all lines contain an error.

16. Write the shortest way to express each of the following statements.

(a) `x = x + 1`

(b) `x = x / 2`

(c) `x = x - 1`

(d) `x = x + y`

(e) `x = x - (y + 7)`

(f) `x = 2*x`

(g) `number_of_closed_cases = number_of_closed_cases + 2*ncc`

17. What is printed by the following code fragment?

```
x1 = 2
x2 = 2
x1 += 1
x2 -= 1
print(x1)
print(x2)
```

Why does the output appear as it does?

18. Consider the following program that attempts to compute the circumference of a circle given the radius entered by the user. Given a circle's radius, r, the circle's circumference, C is given by the formula:

$$C = 2\pi r$$

```
r = 0
PI = 3.14159
#  Formula for the area of a circle given its radius
C = 2*PI*r
#  Get the radius from the user
r = eval(input("Please enter the circle's radius: "))
#  Print the circumference
print("Circumference is", C)
```

(a) The program does not produce the intended result. Why?

(b) How can it be repaired so that it works correctly?

19. Write a Python program that ...

20. Write a Python program that ...

Chapter 4

Conditional Execution

All the programs in the preceding chapters execute exactly the same statements regardless of the input (if any) provided to them. They follow a linear sequence: *Statement* 1, *Statement* 2, etc. until the last statement is executed and the program terminates. Linear programs like these are very limited in the problems they can solve. This chapter introduces constructs that allow program statements to be optionally executed, depending on the context of the program's execution.

4.1 Boolean Expressions

Arithmetic expressions evaluate to numeric values; a *Boolean* expression, sometimes called a *predicate*, may have only one of two possible values: *false* or *true*. The term Boolean comes from the name of the British mathematician George Boole. A branch of discrete mathematics called Boolean algebra is dedicated to the study of the properties and the manipulation of logical expressions. While on the surface Boolean expressions may appear very limited compared to numeric expressions, they are essential for building more interesting and useful programs.

The simplest Boolean expressions in Python are `True` and `False`. In a Python interactive shell we see:

```
>>> True
True
>>> False
False
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

We see that `bool` is the name of the class representing Python's Boolean expressions. Listing 4.1 (`boolvars.py`) is a simple program that shows how Boolean variables can be used.

Expression	Meaning
$x == y$	True if $x = y$ (mathematical equality, not assignment); otherwise, false
$x < y$	True if $x < y$; otherwise, false
$x <= y$	True if $x \leq y$; otherwise, false
$x > y$	True if $x > y$; otherwise, false
$x <= y$	True if $x \geq y$; otherwise, false
$x != y$	True if $x \neq y$; otherwise, false

Table 4.1: The Python relational operators

Expression	Value
`10 < 20`	`True`
`10 >= 20`	`False`
`x < 100`	`True` if x is less than 100; otherwise, `False`
`x != y`	`True` unless x and y are equal

Table 4.2: Examples of some Simple Relational Expressions

Listing 4.1: `boolvars.py`

```python
#  Assign some Boolean variables
a = True
b = False
print('a =', a, ' b =', b)
#  Reassign a
a = False;
print('a =', a, ' b =', b)
```

Listing 4.1 (`boolvars.py`) produces

```
a = True  b = False
a = False  b = False
```

4.2 Boolean Expressions

We have seen that the simplest Boolean expressions are `False` and `True`, the Python Boolean literals. A Boolean variable is also a Boolean expression. An expression comparing numeric expressions for equality or inequality is also a Boolean expression. These comparisons are done using *relational operators*. Table 4.1 lists the relational operators available in Python.

Table 4.2 shows some simple Boolean expressions with their associated values. An expression like `10 < 20` is legal but of little use, since `10 < 20` is always true; the expression `True` is equivalent, simpler, and less likely to confuse human readers. Since variables can change their values during a program's execution, Boolean expressions are most useful when their truth values depend on the values of one or more variables.

The relational operators are binary operators and are all left associative. They all have a lower precedence than any of the arithmetic operators; therefore, the expression

```
x + 2 < y / 10
```

is evaluated as if parentheses were placed as so:

```
(x + 2) < (y / 10)
```

4.3   The Simple if Statement

The Boolean expressions described in Section 4.2 at first may seem arcane and of little use in practical programs. In reality, Boolean expressions are essential for a program to be able to adapt its behavior at run time. Most truly useful and practical programs would be impossible without the availability of Boolean expressions.

The execution errors mentioned in Section 3.4 arise from logic errors. One way that Listing 3.4 (dividedanger.py) can fail is when the user enters a zero for the divisor. Fortunately, programmers can take steps to ensure that division by zero does not occur. Listing 4.2 (betterdivision.py) shows how it might be done.

Listing 4.2: betterdivision.py

```
1  #  File betterdivision.py
2
3  #  Get two integers from the user
4  dividend, divisor = eval(input('Please enter two numbers to divide: '))
5  #  If possible, divide them and report the result
6  if divisor != 0:
7      print(dividend, '/', divisor, "=", dividend/divisor)
```

The print statement may not always be executed. In the following run

```
Please enter two numbers to divide: 32, 8
32 / 8 = 4.0
```

the print statement is executed, but if the user enters a zero as the second number:

```
Please enter two integers to divide: 32, 0
```

the program prints nothing after the user enters the values.

The last non-indented line in Listing 4.2 (betterdivision.py) begins with the reserved word if. The if statement allows code to be optionally executed. In this case, the printing statement is executed only if the variable divisor's value is not zero.

The Boolean expression

`divisor != 0`

determines if the statement in the indented block that follows is executed. If `divisor` is not zero, the message is printed; otherwise, nothing is displayed.

Figure 4.1 shows how program execution flows through the `if` statement. of Listing 4.2 (`betterdivision.py`).
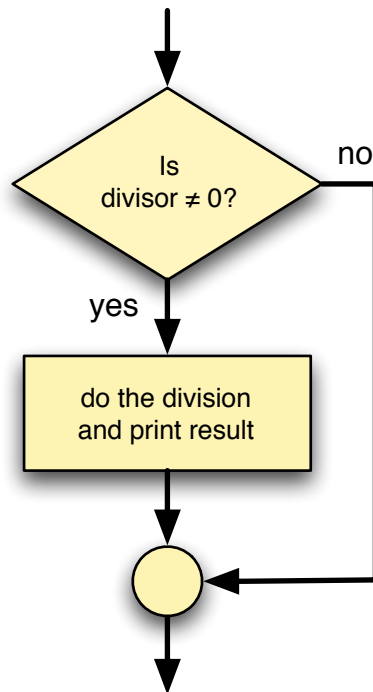
Figure 4.1: if flowchart

The general form of the `if` statement is:

$$\texttt{if} \quad condition \; :$$
$$block$$

- The reserved word `if` begins a `if` statement.

- The *condition* is a Boolean expression that determines whether or not the body will be executed. A colon (`:`) must follow the condition.

- The *block* is a block of one or more statements to be executed if the condition is true. Recall that the statements within a block must all be indented the same number of spaces from the left. The block within an `if` must be indented more spaces than the line that begins the `if` statement. The block technically is part of the `if` statement. This part of the `if` statement is sometimes called the *body* of the `if`.

Python requires the block to be indented. If the block contains just one statement, some programmers will place it on the same line as the `if`; for example, the following `if` statement that optionally assigns `y`

```python
if x < 10:
    y = x
```

could be written

```python
if  x < 10: y = x
```

but may *not* be written as

```python
if x < 10:
y = x
```

because the lack of indentation hides the fact that the assignment statement is optionally executed. Indentation is how Python determines which statements make up a block.

It is important not to mix spaces and tabs when indenting statements in a block. In many editors you cannot visually distinguish between a tab and a sequence of spaces. The number of spaces equivalent to the spacing of a tab differs from one editor to another. Most programming editors have a setting to substitute a specified number of spaces for each tab character. For Python development you should use this feature. It is best to eliminate all tabs within your Python source code.

How many spaces should you indent? Python requires at least one, some programmers consistently use two, four is the most popular number, but some prefer a more dramatic display and use eight. A four space indentation for a block is the recommended Python style. This text uses the recommended four spaces to set off each enclosed block. In most programming editors you can set the tab key to insert spaces automatically so you need not count the spaces as you type.

The `if` block may contain multiple statements to be optionally executed. Listing 4.3 (`alternatedivision.py`) optionally executes two statements depending on the input values provided by the user.

Listing 4.3: `alternatedivision.py`

```python
#  Get two integers from the user
dividend, divisor = eval(input('Please enter two numbers to divide: '))
#  If possible, divide them and report the result
if divisor != 0:
    quotient = dividend/divisor
    print(dividend, '/', divisor, "=", quotient)
print('Program finished')
```

The assignment statement and first printing statement are both a part of the block of the `if`. Given the truth value of the Boolean expression `divisor != 0` during a particular program run, either both statements will be executed or neither statement will be executed. The last statement is not indented, so it is not part of the `if` block. The program always prints *Program finished*, regardless of the user's input.

Remember when checking for equality, as in

```python
if x == 10:
    print('ten')
```

to use the relational equality operator (==), not the assignment operator (=).

As a convenience to programmers, Python's notion of true and false extends beyond what we ordinarily would consider Boolean expressions. The statement

```
if 1:
    print('one')
```

always prints *one*, while the statement

```
if 0:
    print('zero')
```

never prints anything. Python considers the integer value zero to be false and any other integer value to be true. Similarly, the floating-point value 0.0 is false, but any other floating-point value is true. The empty string ('' or "") is considered false, and any non-empty string is interpreted as true. Any Python expression can serve as the condition for an `if` statement. In later chapters we will explore additional kinds of expressions and see how they relate to Boolean conditions.

Listing 4.4 (`leadingzeros.py`) requests an integer value from the user. The program then displays the number using exactly four digits. The program prepends leading zeros where necessary to ensure all four digits are occupied. The program treats numbers less than zero as zero and numbers greater than 9,999 as 9999.

Listing 4.4: `leadingzeros.py`

```
1  #  Request input from the user
2  num = eval(input("Please enter an integer in the range 0...9999: "))
3
4  #  Attenuate the number if necessary
5  if num < 0:           # Make sure number is not too small
6      num = 0
7  if num > 9999:        # Make sure number is not too big
8      num = 9999
9
10 print(end="[")        # Print left brace
11
12 # Extract and print thousands-place digit
13 digit = num//1000     #  Determine the thousands-place digit
14 print(digit, end="")  #  Print the thousands-place digit
15 num %= 1000           #  Discard thousands-place digit
16
17 # Extract and print hundreds-place digit
18 digit = num//100      #  Determine the hundreds-place digit
19 print(digit, end="")  #  Print the hundreds-place digit
20 num %= 100            #  Discard hundreds-place digit
21
22 # Extract and print tens-place digit
23 digit = num//10       #  Determine the tens-place digit
24 print(digit, end="")  #  Print the tens-place digit
25 num %= 10             #  Discard tens-place digit
26
27 #  Remainder is the one-place digit
28 print(num, end="")    #  Print the ones-place digit
29
30 print("]")            #  Print right brace
```

In Listing 4.4 (`leadingzeros.py`), the two `if` statements at the beginning force the number to be in range. The remaining arithmetic statements carve out pieces of the number to display. Recall that the statement

```
num %= 10
```

is short for

```
num = num % 10
```

4.4 The if/else Statement

One undesirable aspect of Listing 4.2 (`betterdivision.py`) is if the user enters a zero divisor, nothing is printed. It may be better to provide some feedback to the user to indicate that the divisor provided cannot be used. The `if` statement has an optional `else` clause that is executed only if the Boolean condition is false. Listing 4.5 (`betterfeedback.py`) uses the `if`/`else` statement to provide the desired effect.

Listing 4.5: `betterfeedback.py`

```
1  #  Get two integers from the user
2  dividend, divisor = eval(input('Please enter two numbers to divide: '))
3  #  If possible, divide them and report the result
4  if divisor != 0:
5      print(dividend, '/', divisor, "=", dividend/divisor)
6  else:
7      print('Division by zero is not allowed')
```

A given run of Listing 4.5 (`betterfeedback.py`) will execute exactly one of either the `if` block or the `else` block. Unlike in Listing 4.2 (`betterdivision.py`), a message is always displayed.

```
Please enter two integers to divide: 32, 0
Division by zero is not allowed
```

The `else` clause contains an alternate block that is executed if the condition is false. The program's flow of execution is shown in Figure 4.2.

Listing 4.5 (`betterfeedback.py`) avoids the division by zero run-time error that causes the program to terminate prematurely, but it still alerts the user that there is a problem. Another application may handle the situation in a different way; for example, it may substitute some default value for `divisor` instead of zero.

The general form of an `if`/`else` statement is

$$\text{if } \textit{condition} :$$
$$\textit{if\_block}$$
$$\text{else}:$$
$$\textit{else\_block}$$

- The reserved word `if` begins the `if`/`else` statement.

- The *condition* is a Boolean expression that determines whether or not the `if` block or the `else` block will be executed. A colon (`:`) must follow the condition.
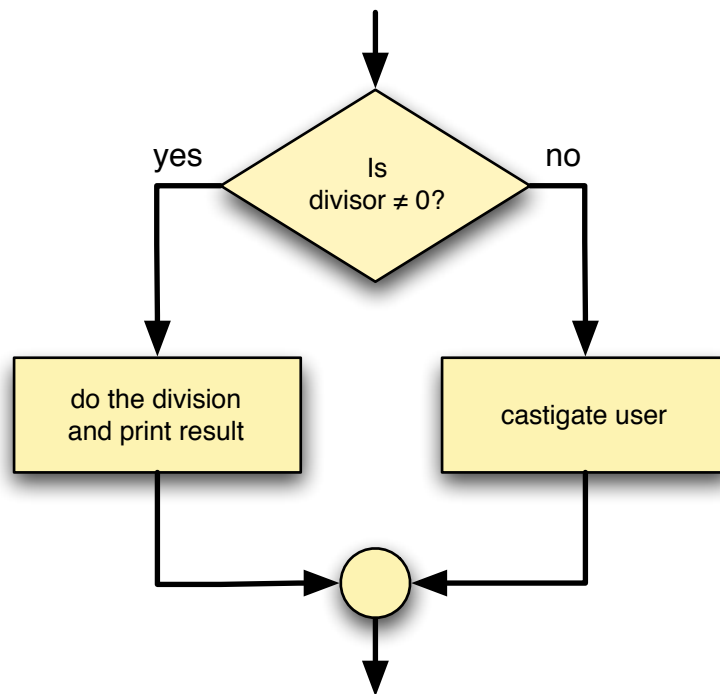
Figure 4.2: if/else flowchart

- The *if_block* is a block of one or more statements to be executed if the condition is true. As with all blocks, it must be indented more spaces than the if line. This part of the if statement is sometimes called the body of the if.

- The reserved word else begins the second part of the if/else statement. A colon (:) must follow the else.

- The *else_block* is a block of one or more statements to be executed if the condition is false. It must be indented more spaces than the else line. This part of the if/else statement is sometimes called the body of the else.

The else block, like the if block, consists of one or more statements indented to the same level.

The equality operator (==) checks for *exact* equality. This can be a problem with floating-point numbers, since floating-point numbers inherently are imprecise. Listing 4.6 (samedifferent.py) demonstrates the perils of using the equality operator with floating-point numbers.

Listing 4.6: samedifferent.py

```
1  d1 = 1.11 - 1.10
2  d2 = 2.11 - 2.10
3  print('d1 =', d1, ' d2 =', d2)
4  if d1 == d2:
5      print('Same')
6  else:
7      print('Different')
```

In mathematics, we expect the following equality to hold:

$$1.11 = 1.10 = 0.01 = 2.11 - 2.10$$

The output of the first `print` statement in Listing 4.6 (`samedifferent.py`) reminds us of the imprecision of floating-point numbers:

```
d1 = 0.010000000000000009   d2 = 0.009999999999999787
```

Since the expression

d1 == d2

checks for exact equality, the program reports the `d1` and `d2` is different. Later we will see how to determine if two floating-point numbers are "close enough" to be considered equal.

4.5 Compound Boolean Expressions

Simple Boolean expressions, each involving one relational operator, can be combined into more complex Boolean expressions using the logical operators `and`, `or`, and `not`. A combination of two or more Boolean expressions using logical operators is called a *compound Boolean expression*.

To introduce compound Boolean expressions, consider a computer science degree that requires, among other computing courses, *Operating Systems* **and** *Programming Languages*. If we isolate those two courses, we can say a student must successfully complete both *Operating Systems* and *Programming Languages* to qualify for the degree. A student that passes *Operating Systems* but not *Programming Languages* will not have met the requirements. Similarly, *Programming Languages* without *Operating Systems* is insufficient, and a student completing neither *Operating Systems* nor *Programming Languages* surely does not qualify.

Logical **AND** works in exactly the same way. If e_1 and e_2 are two Boolean expressions, e_1 and e_2 is true only if e_1 and e_2 are both true; if either one is false or both are false, the compound expression is false.

To illustrate logical **OR**, consider two mathematics courses, *Differential Equations* and *Linear Algebra*. A computer science degree requires one of those two courses. A student who successfully completes *Differential Equations* but does not take *Linear Algebra* meets the requirement. Similarly, a student may take *Linear Algebra* but not *Differential Equations*. A student that takes neither *Differential Equations* nor *Linear Algebra* certainly has not met the requirement. It is important to note the a student may elect to take both *Differential Equations* and *Linear Algebra* (perhaps on the way to a mathematics minor), but the requirement is no less fulfilled.

Logical **OR** works in a similar fashion. Given our Boolean expressions e_1 and e_2, the compound expression e_1 or e_2 is false only if e_1 and e_2 are both false; if either one is true or both are true, the compound expression is true. Note that logical **OR** is an *inclusive or*, not an *exclusive or*. In informal conversion we often imply exclusive or in a statement like "Would you like cake **or** ice cream for dessert?" The implication is one or the other, not both. In computer programming the *or* is inclusive; if both subexpressions in an *or* expression are true, the *or* expression is true.

Logical **NOT** simply reverses the truth value of the expression to which it is applied. If e is a true Boolean expression, not e is false; if e is false, not e is true.

e_1	e_2	e_1 and e_2	e_1 or e_2	not e_1
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

Table 4.3: Logical operators—e_1 and e_2 are Boolean expressions

Table 4.3 is called a *truth table*. It shows all the combinations of truth values for two Boolean expressions and the values of compound Boolean expressions built from applying the `and`, `or`, and `not` Python logical operators.

Both `and` and `or` are binary operators; that is, they require two operands, both of which must be Boolean expressions. The `not` operator is a unary operator (see Section 3.1); it requires a single Boolean operand immediately to its right.

Operator `not` has higher precedence than both `and` and `or`. `and` has higher precedence than `or`. `and` and `or` are left associative; `not` is right associative. `and` and `or` have lower precedence than any other binary operator except assignment. This means the expression

```
x <= y and x <= z
```

is evaluated as

```
(x <= y) and (x <= z)
```

Some programmers prefer to use the parentheses as shown here even though they are not required. The parentheses improve the readability of complex expressions, and the compiled code is no less efficient.

Python allows an expression like

```
x <= y and y <= z
```

which means $x \leq y \leq z$ to be expressed more naturally:

```
x <= y <= z
```

Similarly, Python allows a programmer to test the equivalence of three variables as

```
if x == y == z:
    print('They are all the same')
```

The following section of code assigns the indicated values to a `bool`:

```
x = 10
y = 20
b = (x == 10)              #  assigns True to b
b = (x != 10)              #  assigns False to b
b = (x == 10 and y == 20)  #  assigns True to b
b = (x != 10 and y == 20)  #  assigns False to b
b = (x == 10 and y != 20)  #  assigns False to b
b = (x != 10 and y != 20)  #  assigns False to b
b = (x == 10 or y == 20)   #  assigns True to b
b = (x != 10 or y == 20)   #  assigns True to b
b = (x == 10 or y != 20)   #  assigns True to b
b = (x != 10 or y != 20)   #  assigns False to b
```

Convince yourself that the following expressions are equivalent:

x != y

and

!(x == y)

and

x < y or x > y

In the expression e_1 and e_2 both subexpressions e_1 and e_2 must be true for the overall expression to be true. Since the and operator evaluates left to right, this means that if e_1 is false, there is no need to evaluate e_2. If e_1 is false, no value of e_2 can make the expression e_1 and e_2 true. The and operator first tests the expression to its left. If it finds the expression to be false, it does not bother to check the right expression. This approach is called *short-circuit evaluation*. In a similar fashion, in the expression e_1 or e_2, if e_1 is true, then it does not matter what value e_2 has—an or expression is true unless both subexpressions are false. The or operator uses short-circuit evaluation also.

Why is short-circuit evaluation important? Two situations show why it is important to consider:

- The order of the subexpressions can affect performance. When a program is running, complex expressions require more time for the computer to evaluate than simpler expressions. We classify an expression that takes a relatively long time to evaluate as an *expensive* expression. If a compound Boolean expression is made up of an expensive Boolean subexpression and an less expensive Boolean subexpression, and the order of evaluation of the two expressions does not effect the behavior of the program, then place the more expensive Boolean expression second. If the first subexpression is False and the operator and is being used, then the expensive second subexpression is not evaluated; if the first subexpression is True and the or operator is being used, then, again, the expensive second subexpression is avoided.

- Subexpressions can be ordered to prevent run-time errors. This is especially true when one of the subexpressions depends on the other in some way. Consider the following expression:

    (x != 0) and (z/x > 1)

    Here, if x is zero, the division by zero is avoided. If the subexpressions were switched, a run-time error would result if x is zero.

Suppose you wish to print the word *OK* if a variable x is 1, 2, or 3. An informal translation from English might yield:

```
if x == 1 or 2 or 3:
    print("OK")
```

Unfortunately, x's value is irrelevant; the code always prints the word *OK* regardless of the value of x. Since the == operator has lower precedence than ||, the expression

x == 1 or 2 or 3

is interpreted as

(x == 1) or 2 or 3

The expression `x == 1` is either true or false, but integer 2 is always interpreted as true, and integer 3 is interpreted as true is as well. If `x` is known to be an integer and not a floating-point number, the expression

```
1 <= x <= 3
```

also would work.

The correct statement would be

```
if x == 1 or x == 2 or x == 3:
    print("OK")
```

The revised Boolean expression is more verbose and less similar to the English rendition, but it is the correct formulation for Python.

4.6 Nested Conditionals

The statements in the block of the `if` or the `else` may be any Python statements, including other `if`/`else` statements. These nested `if` statements can be used to develop arbitrarily complex control flow logic. Consider Listing 4.7 (`checkrange.py`) that determines if a number is between 0 and 10, inclusive.

Listing 4.7: `checkrange.py`

```
1  value = eval(input("Please enter an integer value in the range 0...10: ")
2  if value >= 0:          #  First check
3      if value <= 10:     #  Second check
4          print("In range")
5  print("Done")
```

Listing 4.7 (`checkrange.py`) behaves as follows:

- The first condition is checked. If `value` is less than zero, the second condition is not evaluated and the statement following the outer `if` is executed. The statement after the outer `if` simply prints *Done*.

- If the first condition finds `value` to be greater than or equal to zero, the second condition then is checked. If the second condition is met, the *In range* message is displayed; otherwise, it is not. Regardless, the program eventually prints the *Done* message.

We say that the second `if` is *nested* within the first `if`. We call the first `if` the *outer* `if` and the second `if` the *inner* `if`. Both conditions of this nested `if` construct must be met for the *In range* message to be printed. Said another way, the first condition *and* the second condition must be met for the *In range* message to be printed. From this perspective, the program can be rewritten to behave the same way with only *one* `if` statement, as Listing 4.8 (`newcheckrange.py`) shows.

Listing 4.8: `newcheckrange.py`

```
1  value = eval(input("Please enter an integer value in the range 0...10: ")
2  if value >= 0 and value <= 10: #  Only one, more complicated check
3      print("In range")
4  print("Done")
```

Listing 4.8 (`newcheckrange.py`) uses the `and` operator to check both conditions at the same time. Its logic is simpler, using only one `if` statement, at the expense of a slightly more complex Boolean expression in its condition. The second version is preferable here, because simpler logic is usually a desirable goal.

The condition the `if` within Listing 4.8 (`newcheckrange.py`):

```
value >= 0 and value <= 10
```

and be expressed more compactly as

```
0 <= value <= 10
```

Sometimes, a program's logic cannot be simplified as in Listing 4.8 (`newcheckrange.py`). Listing 4.9 (`enhancedcheckrange.py`) would be impossible to rewrite with only one `if` statement.

Listing 4.9: enhancedcheckrange.py

```
1  value = eval(input("Please enter an integer value in the range 0...10: ")
2  if value >= 0:        #  First check
3      if value <= 10:   #  Second check
4          print(value, "is in range")
5      else:
6          print(value, "is too large")
7  else:
8      print(value, "is too small")
9  print("Done")
```

Listing 4.9 (`enhancedcheckrange.py`) provides a more specific message instead of a simple notification of acceptance. Exactly one of three messages is printed based on the value of the variable. A single `if` or `if/else` statement cannot choose from among more than two different execution paths.

Listing 4.10 (`troubleshoot.py`) implements a very simple troubleshooting program that (an equally simple) computer technician might use to diagnose an ailing computer.

Listing 4.10: troubleshoot.py

```
1  print("Help!  My computer doesn't work!")
2  print("Does the computer make any sounds (fans, etc.)")
3  choice = input("or show any lights? (y/n):")
4  #   The troubleshooting control logic
5  if choice == 'n': #  The computer does not have power
6      choice = input("Is it plugged in? (y/n):")
7      if choice == 'n': #  It is not plugged in, plug it in
8          print("Plug it in.  If the problem persists, ")
9          print("please run this program again.")
10     else:  #  It is plugged in
11         choice = input("Is the switch in the \"on\" position? (y/n):")
12         if choice == 'n':  #  The switch is off, turn it on!
13             print("Turn it on.  If the problem persists, ")
14             print("please run this program again.")
15         else:  #  The switch is on
16             choice = input("Does the computer have a fuse?  (y/n):")
17             if choice == 'n':  #  No fuse
18                 choice = input("Is the outlet OK? (y/n):")
19                 if choice == 'n':  #  Fix outlet
20                     print("Check the outlet's circuit ")
21                     print("breaker or fuse.  Move to a")
22                     print("new outlet, if necessary. ")
23                     print("If the problem persists, ")
```

```
24                          print("please run this program again.")
25                     else:  #  Beats me!
26                          print("Please consult a service technician.")
27                 else: #  Check fuse
28                     print("Check the fuse. Replace if ")
29                     print("necessary.  If the problem ")
30                     print("persists, then ")
31                     print("please run this program again.")
32  else:   # The computer has power
33       print("Please consult a service technician.")
```

This very simple troubleshooting program attempts to diagnose why a computer does not work. The potential for enhancement is unlimited, but this version deals only with power issues that have simple fixes. Notice that if the computer has power (fan or disk drive makes sounds or lights are visible), the program indicates that help should be sought elsewhere! The decision tree capturing the basic logic of the program is shown in Figure 4.3. The steps performed are:
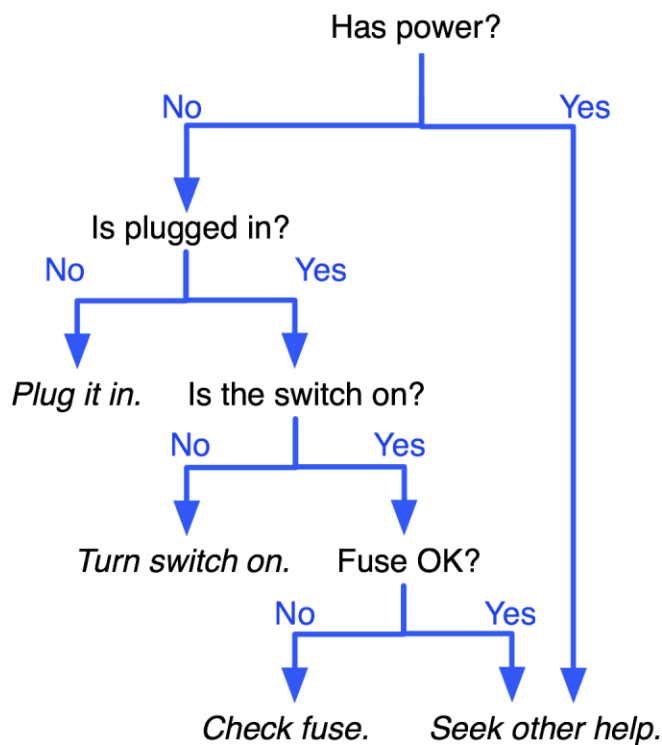
Figure 4.3: Decision tree for troubleshooting a computer system

1. Is it plugged in? This simple fix is sometimes overlooked.

2. Is the switch in the *on* position? This is another simple fix.

3. If applicable, is the fuse blown? Some computer systems have a user-serviceable fuse that can blow out during a power surge. (Most newer computers have power supplies that can handle power surges and have no user-serviceable fuses.)

4. Is there power at the receptacle? Perhaps the outlet's circuit breaker or fuse has a problem.

The easiest checks are made first. Progressively more difficult checks are introduced as the program runs. Based on your experience with troubleshooting computers that do not run properly, you may be able to think of many enhancements to this simple program.

Note the various blocks of code and how the blocks are indented within Listing 4.10 (troubleshoot.py). Visually programmers quickly can determine the logical structure of the program by the arrangement of the blocks.

4.7 Multi-way Decision Statements

A simple if/else statement can select from between two execution paths. Listing 4.9 (enhancedcheckrange.py) showed how to select from among three options. What if exactly one of many actions should be taken? Nested if/else statements are required, and the form of these nested if/else statements is shown in Listing 4.11 (digittoword.py).

Listing 4.11: digittoword.py

```
1  value = eval(input("Please enter an integer in the range 0...5: "))
2  if value < 0:
3      print("Too small")
4  else:
5      if value == 0:
6          print("zero")
7      else:
8          if value == 1:
9              print("one")
10         else:
11             if value == 2:
12                 print("two")
13             else:
14                 if value == 3:
15                     print("three")
16                 else:
17                     if value == 4:
18                         print("four")
19                     else:
20                         if value == 5:
21                             print("five")
22                         else:
23                             print("Too large")
24 print("Done")
```

Observe the following about Listing 4.11 (digittoword.py):

• It prints exactly one of eight messages depending on the user's input.

- Notice that each `if` block contains a single printing statement and each `else` block, except the last one, contains an `if` statement. The control logic forces the program execution to check each condition in turn. The first condition that matches wins, and its corresponding `if` body will be executed. If none of the conditions are true, the program prints the last `else`'s *Too large* message. printed.

As a consequence of the required formatting of Listing 4.11 (`digittoword.py`), the mass of text drifts to the right as more conditions are checked. Python provides a multi-way conditional construct called `if/elif/else` that permits a more manageable textual structure for programs that must check many conditions. Listing 4.12 (`restyleddigittoword.py`) uses the `if/elif/else` statement to avoid the rightward code drift.

Listing 4.12: `restyleddigittoword.py`

```
1  value = eval(input("Please enter an integer in the range 0...5: "))
2  if value < 0:
3      print("Too small")
4  elif value == 0:
5      print("zero")
6  elif value == 1:
7      print("one")
8  elif value == 2:
9      print("two")
10 elif value == 3:
11     print("three")
12 elif value == 4:
13     print("four")
14 elif value == 5:
15     print("five")
16 else:
17     print("Too large")
18 print("Done")
```

The word `elif` is a contraction of `else` and `if`; if you read `elif` as *else if*, you can see how the code fragment

```
else:
    if value == 2:
        print("two")
```

in Listing 4.11 (`digittoword.py`) can be transformed into

```
elif value == 2:
    print("two")
```

in Listing 4.12 (`restyleddigittoword.py`).

The `if/elif/else` statement is valuable for selecting exactly one block of code to execute from several different options. The `if` part of an `if/elif/else` statement is mandatory. The `else` part is optional. After the `if` part and before `else` part (if present) you may use as many `elif` blocks as necessary.

Listing 4.13 (`datetransformer.py`) uses an `if/elif/else` statement to transform a numeric date in month/day format to an expanded US English form and an international Spanish form; for example, `2/14` would be converted to `February 14` and `14 febrero`.

Listing 4.13: `datetransformer.py`

```python
month = eval(input("Please enter the month as a number (1-12): "))
day = eval(input("Please enter the day of the month: "))
#  Translate month into English
if month == 1:
    print("January ", end='')
elif month == 2:
    print("February ", end='')
elif month == 3:
    print("March ", end='')
elif month == 4:
    print("April ", end='')
elif month == 5:
    print("May ", end='')
elif month == 6:
    print("June ", end='')
elif month == 7:
    print("July ", end='')
elif month == 8:
    print("August ", end='')
elif month == 9:
    print("September ", end='')
elif month == 10:
    print("October ", end='')
elif month == 11:
    print("November ", end='')
else:
    print("December ", end='')
#  Add the day
print(day, 'or', day, end='')
#  Translate month into Spanish
if month == 1:
    print(" enero")
elif month == 2:
    print(" febrero")
elif month == 3:
    print(" marzo")
elif month == 4:
    print(" abril")
elif month == 5:
    print(" mayo")
elif month == 6:
    print(" junio")
elif month == 7:
    print(" julio")
elif month == 8:
    print(" agosto")
elif month == 9:
    print(" septiembre")
elif month == 10:
    print(" octubre")
elif month == 11:
    print(" noviembre")
else:
```

```
54    print(" diciembre")
```

A sample run of Listing 4.13 (`datetransformer.py`) is shown here:

```
Please enter the month as a number (1-12): 5
Please enter the day of the month: 20
May 20 or 20 mayo
```

4.8 Conditional Expressions

Consider the following code fragment:

```
if a != b:
    c = d
else:
    c = e
```

Here variable `c` is assigned one of two possible values. As purely a syntactical convenience, Python provides an alternative to the `if`/`else` construct called a *conditional expression*. A conditional expression evaluates to one of two values depending on a Boolean condition. The above code can be rewritten as

```
c = d if a != b else e
```

The general form of the conditional expression is

$$\textit{expression}_1 \; \texttt{if} \;\; \textit{condition} \; \texttt{else} \; \textit{expression}_2$$

where

- $\textit{expression}_1$ is the overall value of the conditional expression if *condition* is true.

- *condition* is a normal Boolean expression that might appear in an `if` statement.

- $\textit{expression}_2$ is the overall value of the conditional expression if *condition* is false.

Listing 4.14 (`safedivide.py`) uses our familiar `if`/`else` statement to check for division by zero.

Listing 4.14: **safedivide.py**

```
1  #  Get the dividend and divisor from the user
2  dividend, divisor = eval(input('Enter dividend, divisor: '))
3  #  We want to divide only if divisor is not zero; otherwise,
4  #  we will print an error message
5  if divisor != 0:
6      print(dividend/divisor)
7  else:
8      print('Error, cannot divide by zero')
```

Using a conditional expression, we can rewrite Listing 4.14 (safedivide.py) as Listing 4.15 (safedivideconditional.py).

Listing 4.15: safedivideconditional.py

```
1  #  Get the dividend and divisor from the user
2  dividend, divisor = eval(input('Enter dividend, divisor: '))
3  #  We want to divide only if divisor is not zero; otherwise,
4  #  we will print an error message
5  msg = dividend/divisor if divisor != 0 else 'Error, cannot divide by zero'
6  print(msg)
```

Notice that in Listing 4.15 (safedivideconditional.py) the type of the msg variable depends which expression is assigned; msg can be a floating-point value (dividend/divisor) or a string ('Error, cannot divide by zero').

As another example, the *absolute value* of a number is defined in mathematics by the following formula:

$$|n| = \begin{cases} n, & \text{when } n \geq 0 \\ -n, & \text{when } n < 0 \end{cases}$$

In other words, the absolute value of a positive number or zero is the same as that number; the absolute value of a negative number is the additive inverse (negative of) of that number. The following Python expression represents the *absolute value* of the variable n:

```
-n if n < 0 else n
```

The expression itself is not statement. Listing 4.16 (absvalueconditional.py) is a small program that provides an example of the conditional expression's use in a statement.

Listing 4.16: absvalueconditional.py

```
1  #  Acquire a number from the user and print its absolute value.
2  n = eval(input("Enter a number: "))
3  print('|', n, '| = ', (-n if n < 0 else n), sep='')
```

Some sample runs of Listing 4.16 (absvalueconditional.py) show

```
Enter a number: -34
|-34| = 34
```

and

```
Enter a number: 0
|0| = 0
```

and

```
Enter a number: 100
|100| = 100
```

Some argue that the conditional expression is not as readable as a normal `if/else` statement. Regardless, it is used sparingly because of its very specific nature. Standard `if/else` blocks can contain multiple statements, but contents in the conditional expression are limited to single, simple expressions.

4.9 Errors in Conditional Statements

Carefully consider each compound conditional used, such as

```
value > 0 and value <= 10
```

found in Listing 4.8 (`newcheckrange.py`). Confusing logical *and* and logical *or* is a common programming error. Consider the Boolean expression

```
x > 0  or  x <= 10
```

What values of x make the expression true, and what values of x make the expression false? The expression is always true, no matter what value is assigned to the variable x. A Boolean expression that is always true is known as a *tautology*. Think about it. If x is a number, what value could the variable x assume that would make this Boolean expression false? Regardless of its value, one or both of the subexpressions will be true, so the compound logical *or* expression is always true. This particular *or* expression is just a complicated way of expressing the value `True`.

Another common error is contriving compound Boolean expressions that are always false, known as *contradictions*. Suppose you wish to *exclude* values from a given range; for example, reject values in the range 0...10 and accept all other numbers. Is the Boolean expression in the following code fragment up to the task?

```
#  All but 0, 1, 2, ..., 10
if value < 0 and value > 10:
    print(value)
```

A closer look at the condition reveals it can *never* be true. What number can be both less than zero and greater than ten *at the same time*? None can, of course, so the expression is a contradiction and a complicated way of expressing `False`. To correct this code fragment, replace the `and` operator with `or`.

4.10 Summary

- Boolean expressions represents the values `True` and `False`.

- The name *Boolean* comes from *Boolean algebra*, the mathematical study of operations on truth values.

- Non-zero numbers and non-empty strings represent true Boolean values. Zero (integer or floating-point) and the empty string (`''` or `""`) represent false.

- Expressions involving the relational operators (==, !=, <, >, <=, and >=) evaluate to Boolean values.

- Boolean expressions can be combined via `and` (logical *AND*) and `or` (logical *OR*).

- `not` represents logical *NOT*.

- The `if` statement can be used to optionally execute statements.

- The block of statements that are part of the `if` statement are executed only if the `if` statement's condition is true.

- The `if` statement has an optional `else` clause to require the selection between two alternate paths of execution.

- The `if/else` statements can be nested to achieve arbitrary complexity.

- The `if/elif/else` statements allow selection of one block of code to execute from many possible options.

- The conditional expression is an expression that evaluates to one of two values depending on a given condition.

- Complex Boolean expressions require special attention, as they are easy to get wrong.

4.11 Exercises

1. What possible values can a Boolean expression have?

2. Where does the term Boolean originate?

3. What is an integer equivalent to `True` in Python?

4. What is the integer equivalent to `False` in Python?

5. Is the value `-16` interpreted as True or False?

6. Given the following definitions:

   ```
   x, y, z = 3, 5, 7
   ```

   evaluate the following Boolean expressions:

   (a) `x == 3`
   (b) `x < y`
   (c) `x >= y`
   (d) `x <= y`
   (e) `x != y - 2`
   (f) `x < 10`
   (g) `x >= 0 and x < 10`
   (h) `x < 0 and x < 10`
   (i) `x >= 0 and x < 2`
   (j) `x < 0 or x < 10`
   (k) `x > 0 or x < 10`
   (l) `x < 0 or x > 10`

7. Given the following definitions:

   ```
   b1, b2, b3, b4 = true, false, x == 3, y < 3
   ```

evaluate the following Boolean expressions:

(a) `b3`

(b) `b4`

(c) `not b1`

(d) `not b2`

(e) `not b3`

(f) `not b4`

(g) `b1 and b2`

(h) `b1 or b2`

(i) `b1 and b3`

(j) `b1 or b3`

(k) `b1 and b4`

(l) `b1 or b4`

(m) `b2 and b3`

(n) `b2 or b3`

(o) `b1 and b2 or b3`

(p) `b1 or b2 and b3`

(q) `b1 and b2 and b3`

(r) `b1 or b2 or b3`

(s) `not b1 and b2 and b3`

(t) `not b1 or b2 or b3`

(u) `not (b1 and b2 and b3)`

(v) `not (b1 or b2 or b3)`

(w) `not b1 and not b2 and not b3`

(x) `not b1 or not b2 or not b3`

(y) `not (not b1 and not b2 and not b3)`

(z) `not (not b1 or not b2 or not b3)`

8. Express the following Boolean expressions in simpler form; that is, use fewer operators. `x` is an integer.

(a) `not (x == 2)`

(b) `x < 2 or x == 2`

(c) `not (x < y)`

(d) `not (x <= y)`

(e) `x < 10 and x > 20`

(f) `x > 10 or x < 20`

(g) `x != 0`

(h) `x == 0`

9. What is the simplest tautology?

10. What is the simplest contradiction?

11. Write a Python program that requests an integer value from the user. If the value is between 1 and 100 inclusive, print "OK;" otherwise, do not print anything.

12. Write a Python program that requests an integer value from the user. If the value is between 1 and 100 inclusive, print "OK;" otherwise, print "Out of range."

13. Write a Python program that allows a user to type in an English day of the week (*Sunday*, *Monday*, etc.). The program should print the Spanish equivalent, if possible.

14. Consider the following Python code fragment:

```python
# i, j, and k are numbers
if  i < j:
    if j < k:
        i = j
    else:
        j = k
else:
    if j > k:
        j = i
    else:
        i = k
print("i =", i, " j =", j, " k =", k)
```

What will the code print if the variables i, j, and k have the following values?

    (a) i is 3, j is 5, and k is 7
    (b) i is 3, j is 7, and k is 5
    (c) i is 5, j is 3, and k is 7
    (d) i is 5, j is 7, and k is 3
    (e) i is 7, j is 3, and k is 5
    (f) i is 7, j is 5, and k is 3

15. Consider the following Python program that prints one line of text:

```python
val = eval(input())
if val < 10:
    if val != 5:
        print("wow ", end='')
    else:
        val += 1
else:
    if val == 17:
        val += 10
    else:
        print("whoa ", end='')
print(val)
```

What will the program print if the user provides the following input?

    (a) 3

      (b) 21

      (c) 5

      (d) 17

      (e) -5

16. Write a Python program that requests five integer values from the user. It then prints the maximum and minimum values entered. If the user enters the values 3, 2, 5, 0, and 1, the program would indicate that 5 is the maximum and 0 is the minimum. Your program should handle ties properly; for example, if the user enters 2, 4 2, 3 and 3, the program should report 2 as the minimum and 4 as maximum.

17. Write a Python program that requests five integer values from the user. It then prints one of two things: if any of the values entered are duplicates, it prints `"DUPLICATES"`; otherwise, it prints `"ALL UNIQUE"`.

18. Write a Python program that ...

Chapter 5

Iteration

Iteration repeats the execution of a sequence of code. Iteration is useful for solving many programming problems. Iteration and conditional execution form the basis for algorithm construction.

5.1 The while Statement

Listing 5.1 (counttofive.py) counts to five by printing a number on each output line.

Listing 5.1: counttofive.py

```
1  print(1)
2  print(2)
3  print(3)
4  print(4)
5  print(5)
```

When executed, this program displays

```
1
2
3
4
5
```

How would you write the code to count to 10,000? Would you copy, paste, and modify 10,000 printing statements? You could, but that would be impractical! Counting is such a common activity, and computers routinely count up to very large values, so there must be a better way. What we really would like to do is print the value of a variable (call it count), then increment the variable (count += 1), and repeat this process until the variable is large enough (count == 5 or maybe count == 10000). This process of executing the same section of code over and over is known as *iteration*, or *looping*. Python has two different statements, while and for, that enable iteration.

Listing 5.2 (iterativecounttofive.py) uses a while statement to count to five:

```
  Listing 5.2: iterativecounttofive.py
1 count = 1              #  Initialize counter
2 while count <= 5:      #  Should we continue?
3     print(count)       #  Display counter, then
4     count += 1         #  Increment counter
```

The `while` statement in Listing 5.2 (`iterativecounttofive.py`) repeatedly displays the variable `count`. The block of statements

```
print(count)
count += 1
```

are executed five times. After each redisplay of the variable `count`, the program increments it by one. Eventually (after five iterations), the condition `count <= 5` will no longer be true, and the block is no longer executed.

Unlike the approach taken in Listing 5.1 (`counttofive.py`), it is trivial to modify Listing 5.2 (`iterativecounttofive.py`) to count up to 10,000—just change the literal value `5` to `10000`.

The line

```
while count <= 5:
```

begins the `while` statement. The expression following the `while` keyword is the condition that determines if the statement block is executed or continues to execute. As long as the condition is true, the program executes the code block over and over again. When the condition becomes false, the loop is finished. If the condition is false initially, the code block within the body of the loop is not executed at all.

The `while` statement has the general form:

$$while \quad condition :$$
$$block$$

- The reserved word `while` begins the `while` statement.

- The *condition* determines whether the body will be (or will continue to be) executed. A colon (`:`) must follow the condition.

- *block* is a block of one or more statements to be executed as long as the condition is true. As a block, all the statements that comprise the block must be indented the same number of spaces from the left. As with the `if` statement, the block must be indented more spaces than the line that begins the `while` statement. The block technically is part of the `while` statement.

Except for the reserved word `while` instead of `if`, `while` statements look identical to `if` statements. Sometimes beginning programmers confuse the two or accidentally type `if` when they mean `while` or vice-versa. Usually the very different behavior of the two statements reveals the problem immediately; however, sometimes, especially in nested, complex logic, this mistake can be hard to detect.

Figure 5.1 shows how program execution flows through Listing 5.2 (`iterativecounttofive.py`).

The condition is checked before the body is executed, and then checked again each time after the block has been executed. If the condition is initially false the block is not executed. If the condition is initially true, the block is executed repeatedly until the condition becomes false, at which point the loop terminates.
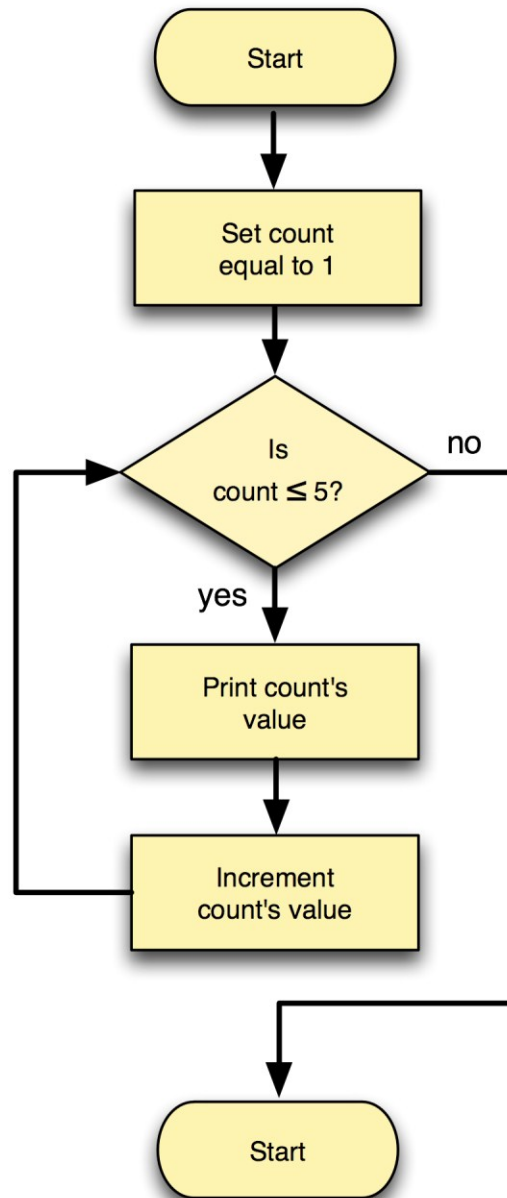
Figure 5.1: while flowchart for Listing 5.2 (`iterativecounttofive.py`)

Listing 5.3 (`addnonnegatives.py`) is a program that allows a user to enter any number of non-negative integers. When the user enters a negative value, the program no longer accepts input, and it displays the sum of all the non-negative values. If a negative number is the first entry, the sum is zero.

Listing 5.3: `addnonnegatives.py`

```
1  #  Allow the user to enter a sequence of non-negative
2  #  numbers.   The user ends the list with a negative
3  #  number.  At the end the sum of the non-negative
4  #  numbers entered is displayed.   The program prints
5  #  zero if the user provides no non-negative numbers.
6
7  entry = 0       #  Ensure the loop is entered
8  sum = 0         #  Initialize sum
9
10 #  Request input from the user
11 print("Enter numbers to sum, negative number ends list:")
12
13 while entry >= 0:            #  A negative number exits the loop
14     entry = eval(input())   #  Get the value
15     if entry >= 0:          #  Is number non-negative?
16         sum += entry        #  Only add it if it is non-negative
17 print("Sum =", sum)         #  Display the sum
```

Listing 5.3 (`addnonnegatives.py`) uses two variables, `entry` and `sum`:

- `entry`

  In the beginning we initialize `entry` to zero for the sole reason that we want the condition `entry >= 0` of the `while` statement to be true initially. If we fail to initialize `entry`, the program will produce a run-time error when it attempts to compare `entry` to zero in the `while` condition. `entry` holds the number entered by the user. Its value can change each time through the loop.

- `sum`

  The variable `sum` is known as an *accumulator*, because it accumulates each value the user enters. We initialize `sum` to zero in the beginning because a value of zero indicates that it has not accumulated anything. If we fail to initialize `sum`, the program generates a run-time when it attempts to use the `+=` operator to modify the variable. Within the loop we repeatedly add the user's input values to `sum`. When the loop finishes (because the user entered a negative number), `sum` holds the sum of all the non-negative values entered by the user.

The initialization of `entry` to zero coupled with the condition `entry >= 0` of the `while` guarantees that the body of the `while` loop will execute at least once. The `if` statement ensures that a negative entry will not be added to `sum`. (Could the `if` condition have used > instead of >= and achieved the same results?) When the user enters a negative value, `sum` will not be updated and the condition of the `while` will no longer be true. The loop then terminates and the program executes the print statement.

Listing 5.3 (`addnonnegatives.py`) shows that a `while` loop can be used for more than simple counting. The program does not keep track of the number of values entered. The program simply accumulates the entered values in the variable named `sum`.

A `while` statement can be used to make Listing 4.10 (`troubleshoot.py`) more convenient for the user. Recall that the computer troubleshooting program forces the user to rerun the program once a potential program has been detected (for example, turn on the power switch, then run the program again to see what else might be wrong). A more desirable decision logic is shown in Figure 5.2.

Listing 5.4 (`troubleshootloop.py`) incorporates a `while` statement so that the program's execution continues until the problem is resolved or its resolution is beyond the capabilities of the program.
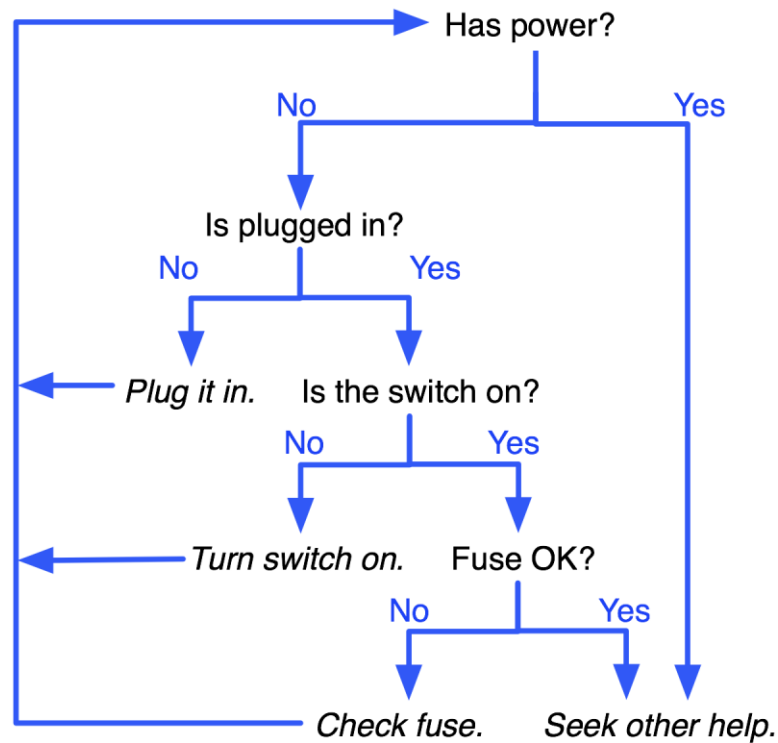
Figure 5.2: Decision tree for troubleshooting a computer system

Listing 5.4: `troubleshootloop.py`

```python
print("Help!  My computer doesn't work!")
done = False    #  Not done initially
while not done:
    print("Does the computer make any sounds (fans, etc.) ")
    choice = input("or show any lights? (y/n):")
    #    The troubleshooting control logic
    if choice == 'n': #  The computer does not have power
        choice = input("Is it plugged in? (y/n):")
        if choice == 'n': #  It is not plugged in, plug it in
            print("Plug it in.")
        else:  #  It is plugged in
            choice = input("Is the switch in the \"on\" position? (y/n):")
            if choice == 'n':  #  The switch is off, turn it on!
                print("Turn it on.")
            else:  #  The switch is on
                choice = input("Does the computer have a fuse?  (y/n):")
                if choice == 'n':  #  No fuse
                    choice = input("Is the outlet OK? (y/n):")
                    if choice == 'n':  #  Fix outlet
```

```
20                              print("Check the outlet's circuit ")
21                              print("breaker or fuse.  Move to a")
22                              print("new outlet, if necessary. ")
23                      else:  #  Beats me!
24                              print("Please consult a service technician.")
25                              done = True   #  Nothing else I can do
26                  else: #  Check fuse
27                      print("Check the fuse. Replace if ")
28                      print("necessary.")
29      else:  # The computer has power
30          print("Please consult a service technician.")
31          done = True   #  Nothing else I can do
```

The bulk of the body of the Listing 5.4 (`troubleshootloop.py`)is wrapped by a `while` statement. The Boolean variable `done` controls the loop; as long as `done` is false, the loop continues. `done` is often called a *flag*. You can think of the flag being down when the value is false and raised when it is true. In this case, when the flag is raised, it is a signal that the loop should terminate.

5.2 Definite Loops vs. Indefinite Loops

In Listing 5.5 (`definite1.py`), code similar to Listing 5.1 (`counttofive.py`), prints the integers from one to 10.

Listing 5.5: `definite1.py`

```
1  n = 1
2  while n <= 10:
3      print(n)
4      n += 1
```

We can inspect the code and determine the number of iterations the loop performs. This kind of loop is known as a *definite loop*, since we can predict exactly how many times the loop repeats. Consider Listing 5.6 (`definite2.py`).

Listing 5.6: `definite2.py`

```
1  n = 1
2  stop = int(input())
3  while n <= stop:
4      print(n)
5      n += 1
```

Looking at the source code of Listing 5.6 (`definite2.py`), we cannot predict how many times the loop will repeat. The number of iterations depends on the input provided by the user. However, at the program's point of execution after obtaining the user's input and before the start of the execution of the loop, we would be able to determine the number of iterations the `while` loop would perform. Because of this, the loop in Listing 5.6 (`definite2.py`) is considered to be a definite loop as well.

Compare these programs to Listing 5.7 (`indefinite.py`).

Listing 5.7: `indefinite.py`

```
1  done = False                    #  Enter the loop at least once
2  while not done:
3      entry = eval(input())       #  Get value from user
4      if entry == 999:            #  Did user provide the magic number?
5          done = True             #  If so, get out
6      else:
7          print(entry)            #  If not, print it and continue
```

In Listing 5.7 (indefinite.py), we cannot predict at any point during the loop's execution how many iterations the loop will perform. The value to match (999) is know before and during the loop, but the variable entry can be anything the user enters. The user could choose to enter 0 exclusively or enter 999 immediately and be done with it. The while statement in Listing 5.7 (indefinite.py) is an example of an *indefinite loop*.

Listing 5.4 (troubleshootloop.py) is another example of an indefinite loop.

The while statement is ideal for indefinite loops. Although we have used the while statement to implement definite loops, Python provides a better alternative for definite loops: the for statement.

5.3 The `for` Statement

The while loop is ideal for indefinite loops. As Listing 5.4 (troubleshootloop.py) demonstrated, a programmer cannot always predict how many times a while loop will execute. We have used a while loop to implement a definite loop, as in

```
n = 1
while n <= 10:
    print(n)
    n += 1
```

The print statement in this code executes exactly 10 times every time this code runs. This code requires three crucial pieces to manage the loop:

- initialization: n = 1

- check: n <= 10

- update: n += 1

Python provides a more convenient way to express a definite loop. The for statement iterates over a range of values. These values can be a numeric range, or, as we shall, elements of a data structure like a string, list, or tuple. The above while loop can be rewritten

```
for n in range(1, 11):
    print(n)
```

The expression range(1, 11) creates an object known as an *iterable* that allows the for loop to assign to the variable n the values 1, 2, ..., 10. During the first iteration of the loop, n's value is 1 within the block. In the loop's second iteration, n has the value of 2. The general form of the range function call is

$$\text{range}(\, begin, end, step\,)$$

where

- *begin* is the first value in the range; if omitted, the default value is 0

- *end* is **one past** the last value in the range; the *end* value may **not** be omitted

- *change* is the amount to increment or decrement; if the *change* parameter is omitted, it defaults to 1 (counts up by ones)

begin, *end*, and *step* must all be integer values; floating-point values and other types are not allowed.

The `range` function is very flexible. Consider the following loop that counts down from 21 to 3 by threes:

```python
for n in range(21, 0, -3):
    print(n, '', end='')
```

It prints

```
21 18 15 12 9 6 3
```

Thus `range(21, 0, -3)` represents the sequence $21, 18, 15, 12, 9, 3$.

The expression `range(1000)` produces the sequence $0, 1, 2, \ldots, 999$.

The following code computes and prints the sum of all the positive integers less than 100:

```python
sum = 0     #  Initialize sum
for i in range(1, 100):
    sum += i
print(sum)
```

The following examples show how `range` can be used to produce a variety of sequences:

- `range(10)` $\rightarrow 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$

- `range(1, 10)` $\rightarrow 1, 2, 3, 4, 5, 6, 7, 8, 9$

- `range(1, 10, 2)` $\rightarrow 1, 3, 5, 7, 9$

- `range(10, 0, -1)` $\rightarrow 10, 9, 8, 7, 6, 5, 4, 3, 2, 1$

- `range(10, 0, -2)` $\rightarrow 10, 8, 6, 4, 2$

- `range(2, 11, 2)` $\rightarrow 2, 4, 6, 8, 10$

- `range(-5, 5)` $\rightarrow -5, -4, -3, -2, -1, 0, 1, 2, 3, 4$

- `range(1, 2)` $\rightarrow 1$

- `range(1, 1)` $\rightarrow$ (empty)

- `range(1, -1)` $\rightarrow$ (empty)

- `range(1, -1, -1)` $\rightarrow 1, 0$

- `range(0)` $\rightarrow$ (empty)

5.4   Nested Loops

Just like with `if` statements, `while` and `for` blocks can contain arbitrary Python statements, including other loops. A loop can therefore be nested within another loop. Listing 5.8 (`timestable.py`) prints a multiplication table on the screen using nested `for` loops.

Listing 5.8: `timestable.py`

```python
 1  #  Print a multiplication table to 10 x 10
 2  #  Print column heading
 3  print("       1   2   3   4   5   6   7   8   9  10")
 4  print("     +---------------------------------------")
 5  for row in range(1, 11):          #  1 <= row <= 10, table has 10 rows
 6      if row < 10:                  #  Need to add space?
 7          print(" ", end="")
 8      print(row, "| ", end="")      #  Print heading for this row.
 9      for column in range(1, 11):   #  Table has 10 columns.
10          product = row*column;     #  Compute product
11          if product < 100:         #  Need to add space?
12              print(end=" ")
13          if product < 10:          #  Need to add another space?
14              print(end=" ")
15          print(product, end=" ")   #  Display product
16      print()                       #  Move cursor to next row
```

The output of Listing 5.8 (`timestable.py`) is

```
          1   2   3   4   5   6   7   8   9  10
        +---------------------------------------
     1 |   1   2   3   4   5   6   7   8   9  10
     2 |   2   4   6   8  10  12  14  16  18  20
     3 |   3   6   9  12  15  18  21  24  27  30
     4 |   4   8  12  16  20  24  28  32  36  40
     5 |   5  10  15  20  25  30  35  40  45  50
     6 |   6  12  18  24  30  36  42  48  54  60
     7 |   7  14  21  28  35  42  49  56  63  70
     8 |   8  16  24  32  40  48  56  64  72  80
     9 |   9  18  27  36  45  54  63  72  81  90
    10 |  10  20  30  40  50  60  70  80  90 100
```

This is how Listing 5.8 (`timestable.py`) works:

- It is important to distinguish what is done only once (outside all loops) from that which is done repeatedly. The column heading across the top of the table is outside of all the loops; therefore, it is printed once in the beginning.

- The work to print the heading for the rows is distributed throughout the execution of the outer loop. This is because the heading for a given row cannot be printed until all the results for the previous row have been printed.

- A code fragment like

```
if x < 10:
    print(end=" ")
 print(x, end=" ")
```

prints x in one of two ways: if x is a one-digit number, it prints a space before it; otherwise, it does not print the extra space. The net effect is to right justify one and two digit numbers within a two character space printing area. This technique allows the columns within the times table to be properly right aligned.

- row is the control variable for the outer loop; column controls the inner loop.

- The inner loop executes ten times on every single iteration of the outer loop. How many times is the statement

```
product = row*column    #  Compute product
```

executed? $10 \times 10 = 100$, one time for every product in the table.

- A newline is printed after the contents of each row is displayed; thus, all the values printed in the inner (column) loop appear on the same line.

With a little work, we can make our multiplication table program more flexible. Listing 5.9 (flexibletimestable.py) will print times tables of various sizes based on the value of the variable named MAX.

Listing 5.9: flexibletimestable.py

```
1  #  Print a MAX x MAX multiplication table
2  MAX = 18
3
4  # First, print heading
5  print(end="      ")
6  # Print column heading numbers
7  for column in range(1, MAX + 1):
8      print(end=" %2i " % column)
9  print()   #  Go down to the next line
10
11 #  Print line separator; a portion for each column
12 print(end="    +")
13 for column in range(1, MAX + 1):
14     print(end="----")  #  Print portion of line
15 print()   #  Go down to the next line
16
17 #  Print table contents
18 for row in range(1, MAX + 1):         #  1 <= row <= MAX, table has MAX rows
19     print(end="%2i | " % row)         #  Print heading for this row.
20     for column in range(1, MAX + 1): #  Table has 10 columns.
21         product = row*column;         #  Compute product
22         print(end="%3i " % product)   #  Display product
23     print()                           #  Move cursor to next row
```

In Listing 5.9 (flexibletimestable.py) we use loops to vary how we print the headings. Listing 5.9 (flexibletimestable.py) works just as well for 1×1 and 15×15 times tables. It is trivial to modify Listing 5.9 (flexibletimestable.py) so that the size of the times table printed is based on user input instead of a programmer-defined constant.

Nested loops are used when an iterative process itself must be repeated. Listing 5.9 (flexibletimestable.py) uses a for inner loop to print the contents of each row, but multiple rows must be printed. The inner (column) loop prints the contents of each row, while the outer (row) loop is responsible for printing all the rows.

Listing 5.10 (permuteabc.py) uses a triply-nested loop to print all the different arrangements of the letters A, B, and C. Each string printed is a *permutation* of ABC.

Listing 5.10: permuteabc.py

```
1   #  File permuteabc.py
2
3   #  The first letter varies from A to C
4   for first in 'ABC':
5       for second in 'ABC': #  The second varies from A to C
6           if second != first:  #   No duplicate letters allowed
7               for third in 'ABC':  #  The third varies from A to C
8                   #  Don't duplicate first or second letter
9                   if third != first and third != second:
10                      print(first + second + third)
```

Notice how the if statements are used to prevent duplicate letters within a given string. The output of Listing 5.10 (permuteabc.py) is all six permutations of ABC:

```
ABC
ACB
BAC
BCA
CAB
CBA
```

Listing 5.11 (permuteabcd.py) uses a four-deep nested loop to print all the different arrangements of the letters A, B, C, and D. Each string printed is a permutation of ABCD.

Listing 5.11: permuteabcd.py

```
1   #  File permuteabcd.py
2
3   #  The first letter varies from A to D
4   for first in 'ABCD':
5       for second in 'ABCD': #  The second varies from A to D
6           if second != first:  #   No duplicate letters allowed
7               for third in 'ABCD':  #  The third varies from A to D
8                   #  Don't duplicate first or second letter
9                   if third != first and third != second:
10                      for fourth in 'ABCD': The fourth varies from A to D
11                          if fourth != first and fourth != second and fourth != third:
12                              print(first + second + third + fourth)
```

5.5   Abnormal Loop Termination

Normally, a `while` statement executes until its condition becomes false. This condition is checked only at the "top" of the loop, so the loop is not immediately exited if the condition becomes false due to activity in the middle of the body. Ordinarily this behavior is not a problem because the intention is to execute all the statements within the body as an indivisible unit. Sometimes, however, it is desirable to immediately exit the body or recheck the condition from the middle of the loop instead. Python provides the `break` and `continue` statements to give programmers more flexibility designing the control logic of loops.

5.5.1   The `break` statement

Python provides the `break` statement to implement middle-exiting control logic. The `break` statement causes the immediate exit from the body of the loop. Listing 5.12 (`addmiddleexit.py`) is a variation of Listing 5.3 (`addnonnegatives.py`) that illustrates the use of `break`.

Listing 5.12: addmiddleexit.py

```
 1  #  Allow the user to enter a sequence of non-negative
 2  #  numbers.  The user ends the list with a negative
 3  #  number.  At the end the sum of the non-negative
 4  #  numbers entered is displayed.  The program prints
 5  #  zero if the user provides no non-negative numbers.
 6
 7  entry = 0       #  Ensure the loop is entered
 8  sum = 0         #  Initialize sum
 9
10  #  Request input from the user
11  print("Enter numbers to sum, negative number ends list:")
12
13  while True:                 #  Loop forever
14      entry = eval(input())   #  Get the value
15      if entry < 0:           #  Is number negative number?
16          break               #  If so, exit the loop
17      sum += entry            #  Add entry to running sum
18  print("Sum =", sum)         #  Display the sum
```

The condition of the `while` is a tautology, so the body of the loop will be entered. Since the condition of the `while` can never be false, the `break` statement is the only way to get out of the loop. The `break` statement is executed only when the user enters a negative number. When the `break` statement is encountered during the program's execution, the loop is immediately exited. Any statements following the `break` within the body are skipped. It is not possible, therefore, to add a negative number to the `sum` variable.

Listing 5.4 (`troubleshootloop.py`) uses a variable named `done` that controls the duration of the loop. Listing 5.13 (`troubleshootloop2.py`) uses `break` statements in place of the Boolean `done` variable.

Listing 5.13: troubleshootloop2.py

```
 1  print("Help!  My computer doesn't work!")
 2  while True:
 3      print("Does the computer make any sounds (fans, etc.)")
 4      choice = input(" or show any lights? (y/n):")
```

```
5        #   The troubleshooting control logic
6     if choice == 'n': #  The computer does not have power
7         choice = input("Is it plugged in? (y/n):")
8         if choice == 'n': #  It is not plugged in, plug it in
9             print("Plug it in.")
10        else:  #  It is plugged in
11            choice = input("Is the switch in the \"on\" position? (y/n):")
12            if choice == 'n':  #  The switch is off, turn it on!
13                print("Turn it on.")
14            else:  #  The switch is on
15                choice = input("Does the computer have a fuse?  (y/n):")
16                if choice == 'n':  #  No fuse
17                    choice = input("Is the outlet OK? (y/n):")
18                    if choice == 'n':  #  Fix outlet
19                        print("Check the outlet's circuit ")
20                        print("breaker or fuse.  Move to a")
21                        print("new outlet, if necessary. ")
22                    else:  #  Beats me!
23                        print("Please consult a service technician.")
24                        break   #  Nothing else I can do, exit loop
25                else: #  Check fuse
26                    print("Check the fuse. Replace if ")
27                    print("necessary.")
28     else:  # The computer has power
29         print("Please consult a service technician.")
30         break   #  Nothing else I can do, exit loop
```

The break statement should be used sparingly because it introduces an exception into the normal control logic of the loop. Ideally, every loop should have a single entry point and single exit point. While Listing 5.12 (addmiddleexit.py) has a single exit point (the break statement), programmers commonly use break statements within while statements with conditions that are not always true. In such a while loop, adding a break statement adds an extra exit point (the top of the loop where the condition is checked is one point, and the break statement is another). Using multiple break statements within a single loop is particularly dubious and should be avoided. Why have the break statement at all if its use is questionable and it is dispensable? The logic in Listing 5.3 (addnonnegatives.py) is fairly simple, so the restructuring of Listing 5.12 (addmiddleexit.py) is straightforward; in general, the effort may complicate the logic a bit and require the introduction of an additional Boolean variable. Any program that uses a break statement can be rewritten so that the break statement is not used. Any loop of the form

```
while  condition₁ :
        statement₁
        statement₂

        ...

        statementₙ
        if  condition₂:
                statementₙ₊₁
                statementₙ₊₂

                ...

                statementₙ₊ₘ
                break
        statementₙ₊ₘ₊₁
        statementₙ₊ₘ₊₂

        ...

        statementₙ₊ₘ₊ₚ
```

can be rewritten as

```
done = false
while not done and  condition₁ :
        statement₁
        statement₂

        ...

        statementₙ;
        if  condition₂ :
                statementₙ₊₁
                statementₙ₊₂

                ...

                statementₙ₊ₘ
                done = true
        else:
                statementₙ₊ₘ₊₁
                statementₙ₊ₘ₊₂

                ...

                statementₙ₊ₘ₊ₚ
```

The no-`break` version introduces a Boolean variable, and the loop control logic is a little more compli-cated. The no-`break` version uses more space (an extra variable) and more time (requires an extra check in the loop condition), and its logic is more complex. The more complicated the control logic for a given section of code, the more difficult the code is to write correctly. In some situations, even though it violates the "single entry point, single exit point" principle, a simple `break` statement can be an acceptable loop control option.

5.5.2 The continue Statement

The `continue` statement is similar to the `break` statement. During a program's execution, when the `break` statement is encountered within the body of a loop, the remaining statements within the body of the loop are skipped, and the loop is exited. When a `continue` statement is encountered within a loop, the remaining statements within the body are skipped, but the loop condition is checked to see if the loop should continue or be exited. If the loop's condition is still true, the loop is not exited, but the loop's execution continues at the top of the loop. Listing 5.14 (`continueexample.py`) shows how the `continue` statement can be used.

Listing 5.14: **continueexample.py**

```
1   sum = 0
2   done = False;
3   while not done:
4       val = eval(input("Enter positive integer (999 quits):"))
5       if val < 0:
6           print("Negative value",  val,  "ignored")
7           continue;   # Skip rest of body for this iteration
8       if val != 999:
9           print("Tallying", val)
10          sum += val
11      else:
12          done = (val == 999);   #  999 entry exits loop
13  print("sum =", sum)
```

The `continue` statement is not used as frequently as the `break` statement since it is often easy to trans-form the code into an equivalent form that does not use `continue`. Listing 5.15 (`nocontinueexample.py`) works exactly like Listing 5.14 (`continueexample.py`), but the `continue` has been eliminated.

Listing 5.15: **nocontinueexample.py**

```
1   sum = 0
2   done = False;
3   while not done:
4       val = eval(input("Enter positive integer (999 quits):"))
5       if val < 0:
6           print("Negative value",  val,  "ignored")
7       else:
8           if val != 999:
9               print("Tallying", val)
10              sum += val
11          else:
12              done = (val == 999);   #  999 entry exits loop
13  print("sum =", sum)
```

In fact any loop body of the form

```
while condition₁:
        statement₁
        statement₂
        ...
        statementₙ
        if condition₂:
                statementₙ₊₁
                statementₙ₊₂
                 ...
                statementₙ₊ₘ
                continue
        statementₙ₊ₘ₊₁
        statementₙ₊ₘ₊₂
        ...
        statementₙ₊ₘ₊ₚ
```

can be rewritten as

```
while condition₁:
        statement₁
        statement₂
        ...
        statementₙ
        if condition₂:
                statementₙ₊₁
                statementₙ₊₂
                ...
                statementₙ₊ₘ
        else:
                statementₙ₊ₘ₊₁
                statementₙ₊ₘ₊₂
                ...
                statementₙ₊ₘ₊ₚ
```

The logic of the `else` version is no more complex than the `continue` version. Therefore, unlike the `break` statement above, there is no compelling reason to use the `continue` statement. Sometimes a

`continue` statement is added at the last minute to an existing loop body to handle an exceptional condition (like ignoring negative numbers in the example above) that initially went unnoticed. If the body of the loop is lengthy, a conditional statement with a `continue` can be added easily near the top of the loop body without touching the logic of the rest of the loop. Therefore, the `continue` statement merely provides a convenient alternative to the programmer. The `else` version is preferred.

5.6 Infinite Loops

An infinite loop is a loop that executes its block of statements repeatedly until the user forces the program to quit. Once the program flow enters the loop's body it cannot escape. Infinite loops are sometimes designed. For example, a long-running server application like a Web server may need to continuously check for incoming connections. This checking can be performed within a loop that runs indefinitely. All too often for beginning programmers, however, infinite loops are created by accident and represent logical errors in their programs.

Intentional infinite loops should be made obvious. For example,

```
while True:
    #  Do something forever. . .
```

The Boolean literal `True` is always true, so it is impossible for the loop's condition to be false. The only ways to exit the loop is via a `break` statement, `return` statement (see Chapter 7), or a `sys.exit` call (see Section **??**) embedded somewhere within its body.

Intentional infinite loops are easy to write correctly. Accidental infinite loops are quite common, but can be puzzling for beginning programmers to diagnose and repair. Consider Listing 5.16 (`findfactors.py`) that attempts to print all the integers with their associated factors from 1 to 20.

Listing 5.16: `findfactors.py`

```
1  #  List the factors of the integers 1...MAX
2  MAX = 20                          #  MAX is 20
3  n = 1   #  Start with 1
4  while n <= MAX:                   #  Do not go past MAX
5      factor = 1                    #  1 is a factor of any integer
6      print(end=str(n) + ': ')      #  Which integer are we examining?
7      while factor <= n:            #  Factors are <= the number
8          if n % factor == 0:       #  Test to see if factor is a factor of n
9              print(factor, end=' ') #  If so, display it
10             factor += 1           #  Try the next number
11     print()  #  Move to next line for next n
12     n += 1
```

It displays

```
1: 1
2: 1 2
3: 1
```

and then "freezes up" or "hangs," ignoring any user input (except the key sequence Ctrl-C on most systems which interrupts and terminates the running program). This type of behavior is a frequent symptom of

an unintentional infinite loop. The factors of 1 display properly, as do the factors of 2. The first factor of 3 is properly displayed and then the program hangs. Since the program is short, the problem may be easy to locate. In some programs, though, the error may be challenging to find. Even in Listing 5.16 (`findfactors.py`) the debugging task is nontrivial since nested loops are involved. (Can you find and fix the problem in Listing 5.16 (`findfactors.py`) before reading further?)

In order to avoid infinite loops, we must ensure that the loop exhibits certain properties:

- The loop's condition must not be a tautology (a Boolean expression that can never be false). For example,

```python
while i >= 1 or i <= 10:
    #  Block of code follows ...
```

is an infinite loop since any value chosen for `i` will satisfy one or both of the two subconditions. Perhaps the programmer intended to use a `and` instead of `or` to stay in the loop as long as `i` remains in the range 1...10.

In Listing 5.16 (`findfactors.py`) the outer loop condition is

```python
n <= MAX
```

If `n` is 21 and `MAX` is 20, then the condition is false. Since we can find values for `n` and `MAX` that make this expression false, it cannot be a tautology. Checking the inner loop condition:

```python
factor <= n
```

we see that if `factor` is 3 and `n` is 2, then the expression is false; therefore, this expression also is not a tautology.

- The condition of a `while` must be true initially to gain access to its body. The code within the body must modify the state of the program in some way so as to influence the outcome of the condition that is checked at each iteration. This usually means one of the variables used in the condition is modified in the body. Eventually the variable assumes a value that makes the condition false, and the loop terminates.

In Listing 5.16 (`findfactors.py`) the outer loop's condition involves the variables `n` and `MAX`. We observe that we assign 20 to `MAX` before the loop and never change it afterward, so to avoid an infinite loop it is essential that `n` be modified within the loop. Fortunately, the last statement in the body of the outer loop increments `n`. `n` is initially 1 and `MAX` is 20, so unless the circumstances arise to make the inner loop infinite, the outer loop eventually should terminate.
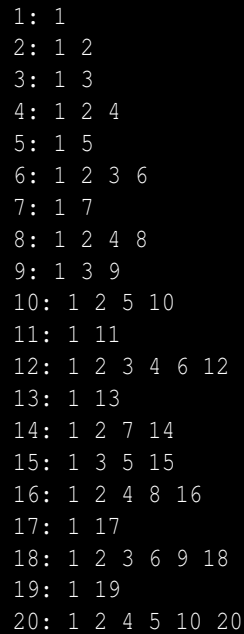
The inner loop's condition involves the variables `n` and `factor`. No statement in the inner loop modifies `n`, so it is imperative that `factor` be modified in the loop. The good news is `factor` is incremented in the body of the inner loop, but the bad news is the increment operation is protected within the body of the `if` statement. The inner loop contains one statement, the `if` statement. That `if` statement in turn has two statements in its body:

```python
while factor <= n:
    if n % factor == 0:
        print(factor, end=' ')
        factor += 1
```

If the condition of the `if` is ever false, the variable `factor` will not change. In this situation if the expression `factor <= n` was true, it will remain true. This effectively creates an infinite loop. The statement that modifies `factor` must be moved outside of the `if` statement's body:

```
while factor <= n:
    if n % factor == 0:
        print(factor, end=' ')
    factor += 1
```

This new version runs correctly:

```
1: 1
2: 1 2
3: 1 3
4: 1 2 4
5: 1 5
6: 1 2 3 6
7: 1 7
8: 1 2 4 8
9: 1 3 9
10: 1 2 5 10
11: 1 11
12: 1 2 3 4 6 12
13: 1 13
14: 1 2 7 14
15: 1 3 5 15
16: 1 2 4 8 16
17: 1 17
18: 1 2 3 6 9 18
19: 1 19
20: 1 2 4 5 10 20
```

A debugger can be used to step through a program to see where and why an infinite loop arises. Another common technique is to put print statements in strategic places to examine the values of the variables involved in the loop's control. The original inner loop can be so augmented:

```
while factor <= n:
    print('factor =', factor, '  n =', n)
    if n % factor == 0:
        print(factor, end=' ')
        factor += 1    #  <-- Note, still has original error here
```

It produces the following output:

```
1: factor = 1  n = 1
1
2: factor = 1  n = 2
1 factor = 2  n = 2
2
3: factor = 1  n = 3
1 factor = 2  n = 3
factor = 2  n = 3
factor = 2  n = 3
factor = 2  n = 3
factor = 2  n = 3
factor = 2  n = 3
   .
   .
   .
```

The program continues to print the same line until the user interrupts its execution. The output demonstrates that once `factor` becomes equal to 2 and `n` becomes equal to 3 the program's execution becomes trapped in the inner loop. Under these conditions:

1. `2 < 3` is true, so the loop continues and

2. `3 % 2` is equal to 1, so the `if` statement will not increment `factor`.

It is imperative that `factor` be incremented each time through the inner loop; therefore, the statement incrementing `factor` must be moved outside of the `if`'s guarded body. Moving it outside means removing it from the `if` statement's block, which means unindenting it.

Listing 5.17 (`findfactorsfor.py`) is a different version of our factor finder program that uses nested `for` loops instead of nested `while` loops. Not only is it slightly shorter, but it avoids the potential for the misplaced increment of the `factor` variable. This is because the `for` statement automatically handles the loop variable update.

Listing 5.17: `findfactorsfor.py`

```python
# List the factors of the integers 1...MAX
MAX = 20                               # MAX is 20
for n in range(1, MAX + 1):            # Consider numbers 1...MAX
    print(end=str(n) + ': ')           # Which integer are we examining?
    for factor in range(1, n + 1):     # Try factors 1...n
        if n % factor == 0:            # Test to see if factor is a factor of n
            print(factor, end=' ')     # If so, display it
    print()   # Move to next line for next n
```

5.7 Iteration Examples

We can implement some sophisticated algorithms in Python now that we are armed with `if` and `while` statements. This section provides several examples that show off the power of conditional execution and iteration.

5.7.1 Computing Square Root

Suppose you must write a Python program that computes the square root of a number supplied by the user. We can compute the square root of a number by using the following method:

1. Guess the square root.

2. Square the guess and see how close it is to the original number; if it is close enough to the correct answer, stop.

3. Make a new guess that will produce a better result and proceed with step 2.

Step 3 is a little vague, but Listing 5.18 (`computesquareroot.py`) implements the above algorithm in Python, providing the missing details.

Listing 5.18: `computesquareroot.py`

```
 1  #  File computesquareroot.py
 2
 3  #  Get value from the user
 4  val = eval(input('Enter number: '))
 5  #  Compute a provisional square root
 6  root = 1.0;
 7
 8  #  How far off is our provisional root?
 9  diff = root*root - val
10
11  #  Loop until the provisional root
12  #  is close enough to the actual root
13  while diff > 0.00000001 or diff < -0.00000001:
14      root = (root + val/root) / 2        # Compute new provisional root
15      print(root, 'squared is', root*root)  #  Report how we are doing
16      #  How bad is our current approximation?
17      diff = root*root - val
18
19  #  Report approximate square root
20  print('Square root of', val, "=", root)
```

The program is based on a simple algorithm that uses successive approximations to zero in on an answer that is within 0.00000001 of the true answer.

One sample run is

```
Enter number: 2
1.5 squared is 2.25
1.4166666666666665 squared is 2.006944444444444
1.4142156862745097 squared is 2.0000060073048824
1.4142135623746899 squared is 2.0000000000045106
Square root of 2 = 1.4142135623746899
```

The actual square root is approximately 1.4142135623730951 and so the result is within our accepted tolerance (0.00000001). Another run yields

```
Enter number: 100
50.5 squared is 2550.25
26.24009900990099 squared is 688.542796049407
15.025530119986813 squared is 225.76655538663093
10.840434673026925 squared is 117.51502390016438
10.032578510960604 squared is 100.6526315785885
10.000052895642693 squared is 100.0010579156518
10.000000000139897 squared is 100.00000000279795
Square root of 100 = 10.000000000139897
```

The real answer, of course, is 10, but our computed result again is well within our programmed tolerance.

While Listing 5.18 (`computesquareroot.py`) is a good example of the practical use of a loop, if we really need to compute the square root, Python has a library function that is more accurate and more efficient. We investigate it and other handy mathematical functions in Chapter 6.

5.7.2 Drawing a Tree

Suppose we wish to draw a triangular tree, and its height is provided by the user. A tree that is five levels tall would look like

```
    *
   ***
  *****
 *******
*********
```

whereas a three-level tree would look like

```
    *
   ***
  *****
```

If the height of the tree is fixed, we can write the program as a simple variation of Listing 1.2 (`arrow.py`) which uses only printing statements and no loops. Our program, however, must vary its height and width based on input from the user.

Listing 5.19 (`startree.py`) provides the necessary functionality.

Listing 5.19: `startree.py`

```python
#  Get tree height from user
height = eval(input("Enter height of tree: "))

#  Draw one row for every unit of height
row = 0
while row < height:
    #  Print leading spaces; as row gets bigger, the number of
    #  leading spaces gets smaller
    count = 0
    while count < height - row:
        print(end=" ")
        count += 1

    #  Print out stars, twice the current row plus one:
    #     1. number of stars on left side of tree
    #        = current row value
    #     2. exactly one star in the center of tree
    #     3. number of stars on right side of tree
    #        = current row value
    count = 0
    while count < 2*row + 1:
        print(end="*")
        count += 1
    #  Move cursor down to next line
    print()
    row += 1    #  Consider next row
```

When Listing 5.19 (`startree.py`) is run and the user enters, for example, `7`, the output is:

```
Enter height of tree: 7
      *
     ***
    *****
   *******
  *********
 ***********
*************
```

Listing 5.19 (`startree.py`) uses two sequential `while` loops nested within a `while` loop. The outer `while` loop is responsible for drawing one row of the tree each time its body is executed:

- As long as the user enters a value greater than zero, the body of the outer `while` loop will be executed; if the user enters zero or less, the program terminates and does nothing. This is the expected behavior.

- The last statement in the body of the outer `while`:

  `row += 1`

  ensures that the variable `row` increases by one each time through the loop; therefore, it eventually will equal `height` (since it initially had to be less than `height` to enter the loop), and the loop will terminate. There is no possibility of an infinite loop here.

The two inner loops play distinct roles:

- The first inner loop prints spaces. The number of spaces printed is equal to the height of the tree the first time through the outer loop and decreases each iteration. This is the correct behavior since each succeeding row moving down contains fewer leading spaces but more asterisks.

- The second inner loop prints the row of asterisks that make up the tree. The first time through the outer loop, `row` is zero, so no left side asterisks are printed, one central asterisk is printed (the top of the tree), and no right side asterisks are printed. Each time through the loop the number of left-hand and right-hand stars to print both increase by one and the same central asterisk is printed; therefore, the tree grows one wider on each side each line moving down. Observe how the `2*row + 1` value expresses the needed number of asterisks perfectly.

- While it seems asymmetrical, note that no third inner loop is required to print trailing spaces on the line after the asterisks are printed. The spaces would be invisible, so there is no reason to print them!

For comparison, Listing 5.20 (`startreefor.py`) uses `for` loops instead of `while` loops to draw our star trees.
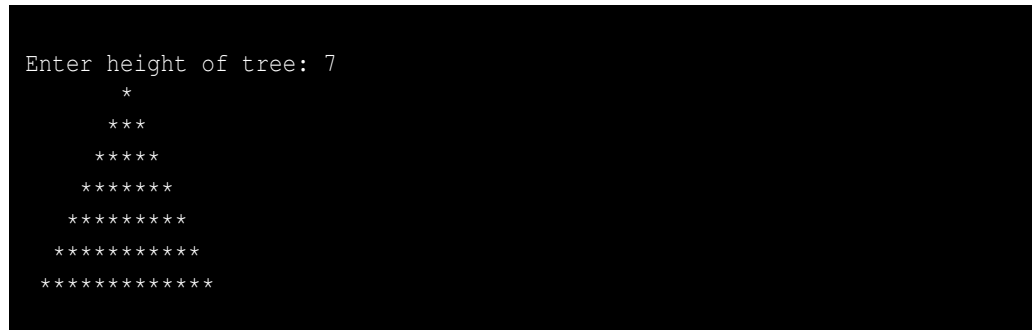
Listing 5.20: `startreefor.py`

```
1  #  Get tree height from user
2  height = eval(input("Enter height of tree: "))
3
4  #  Draw one row for every unit of height
5  for row in range(height):
6      #  Print leading spaces; as row gets bigger, the number of
7      #  leading spaces gets smaller
8      for count in range(height - row):
9          print(end=" ")
10
11     #  Print out stars, twice the current row plus one:
12     #    1. number of stars on left side of tree
13     #       = current row value
14     #    2. exactly one star in the center of tree
15     #    3. number of stars on right side of tree
16     #       = current row value
17     for count in range(2*row + 1):
18         print(end="*")
19     #  Move cursor down to next line
20     print()
```

5.7.3 Printing Prime Numbers

A *prime number* is an integer greater than one whose only factors (also called divisors) are one and itself. For example, 29 is a prime number (only 1 and 29 divide into it with no remainder), but 28 is not (2, 4, 7, and 14 are factors of 28). Prime numbers were once merely an intellectual curiosity of mathematicians, but now they play an important role in cryptography and computer security.

The task is to write a program that displays all the prime numbers up to a value entered by the user. Listing 5.21 (`printprimes.py`) provides one solution.

Listing 5.21: `printprimes.py`

```python
max_value = eval(input('Display primes up to what value? '))
value = 2  #  Smallest prime number
while value <= max_value:
    #  See if value is prime
    is_prime = True  # Provisionally, value is prime
    #  Try all possible factors from 2 to value - 1
    trial_factor = 2
    while trial_factor < value:
        if value % trial_factor == 0:
            is_prime = False;   #  Found a factor
            break               # No need to continue; it is NOT prime
        trial_factor += 1       # Try the next potential factor
    if is_prime:
        print(value, end= ' ')  #  Display the prime number
    value += 1                  # Try the next potential prime number
print()  #  Move cursor down to next line
```

Listing 5.21 (`printprimes.py`), with an input of 90, produces:

```
Display primes up to what value? 90
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89
```

The logic of Listing 5.21 (`printprimes.py`) is a little more complex than that of Listing 5.19 (`startree.py`). The user provides a value for `max_value`. The main loop (outer `while`) iterates over all the values from two to `max_value`:

- The `is_prime` variable is initialized to true, meaning `value` is assumed to be prime unless our tests prove otherwise. `trial_factor` takes on all the values from two to `value - 1` in the inner loop:

  ```python
  trial_factor = 2
  while trial_factor < value:
  if value % trial_factor == 0:
      is_prime = False;   #  Found a factor
      break               #  No need to continue; it is NOT prime
  trial_factor += 1       #  Try the next potential factor
  ```

  The expression `value % trial_factor` is zero when `trial_factor` divides into `value` with no remainder—exactly when `trial_factor` is a factor of `value`. If any of the values of `trial_factor` is determined actually to be a factor of `value`, then `is_prime` is set to false, and the loop is exited via the `break`. If the loop continues to completion, `is_prime` will never be set to false, which means no factors were found and `value` is indeed prime.

- The `if` statement after the inner loop:

  ```python
  if is_prime:
      print(value, end= ' ')  #  Display the prime number
  ```

  simply checks the status of `is_prime`. If `is_prime` is true, then `value` must be prime, so `value` is printed followed by a space to separate it from other factors that may be printed during the next iterations.

Some important questions can be asked.

1. **If the user enters a 2, will it be printed?**

   In this case max_value = value = 2, so the condition of the outer loop

   value <= max_value

   is true, since $2 \leq 2$. is_prime is set to true, but the condition of the inner loop

   trial_factor < value

   is not true (2 is not less than 2). Thus, the inner loop is skipped, is_prime is not changed from true, and 2 is printed. This behavior is correct, because 2 is the smallest prime number (and the only even prime).

2. **if the user enters a number less than 2, is anything printed?**

   The while condition ensures that values less than two are not considered. The body of the while will never be entered. Only the newline is printed, and no numbers are displayed. This behavior is correct.

3. **Is the inner loop guaranteed to always terminate?**

   In order to enter the body of the inner loop, trial_factor must be less than value. value does not change anywhere in the loop. trial_factor is not modified anywhere in the if statement within the loop, and it is incremented within the loop immediately after the if statement. trial_factor is, therefore, incremented during each iteration of the loop. Eventually, trial_factor will equal value, and the loop will terminate.

4. **Is the outer loop guaranteed to always terminate?**

   In order to enter the body of the outer loop, value must be less than or equal to max_value. max_value does not change anywhere in the loop. value is increased in the last statement within the body of the outer loop, and value is not modified anywhere else. Since the inner loop is guaranteed to terminate as shown in the previous answer, eventually value will exceed max_value and the loop will end.

The logic of the inner while can be rearranged slightly to avoid the break statement. The current version is:

```
while trial_factor < value:
    if value % trial_factor == 0:
        is_prime = False;     # Found a factor
        break                 # No need to continue; it is NOT prime
    trial_factor += 1         # Try the next potential factor
```

It can be rewritten as:

```
while is_prime and trial_factor < value:
    is_prime = (value % trial_factor != 0)  # Update is_prime
    trial_factor += 1         # Try the next potential factor
```

This version without the break introduces a slightly more complicated condition for the while but removes the if statement within its body. is_prime is initialized to true before the loop. Each time through the loop it is reassigned. trial_factor will become false if at any time value % trial_factor is zero. This is exactly when trial_factor is a factor of value. If is_prime becomes false, the loop cannot continue, and

if `is_prime` never becomes false, the loop ends when `trial_factor` becomes equal to `value`. Because of operator precedence, the parentheses in

is_prime = (value % trial_factor != 0)

are not necessary. The parentheses do improve readability, since an expression including both = and != is awkward for humans to parse. When parentheses are placed where they are not needed, as in

    x = (y + 2);

the interpreter simply ignores them, so there is no efficiency penalty in the executing program.

    We can shorten the code of Listing 5.21 (`printprimes.py`) a bit by using `for` statements instead of `while` statements as shown in Listing 5.22 (`printprimesfor.py`).

Listing 5.22: `printprimesfor.py`

```
1   max_value = eval(input('Display primes up to what value? '))
2   #  Try values from 2 (smallest prime number) to max_value
3   for value in range(2, max_value + 1):
4       #  See if value is prime
5       is_prime = True  # Provisionally, value is prime
6       #  Try all possible factors from 2 to value − 1
7       for trial_factor in range(2, value):
8           if value % trial_factor == 0:
9               is_prime = False    #  Found a factor
10              break               # No need to continue; it is NOT prime
11      if is_prime:
12          print(value, end= ' ')  #  Display the prime number
13  print()   #  Move cursor down to next line
```

5.7.4 Insisting on the Proper Input

Listing 5.23 (`betterinputonly.py`) traps the user in a loop until the user provides an acceptable integer value.

Listing 5.23: `betterinputonly.py`

```
1   #  Require the user to enter an integer in the range 1–10
2   in_value = 0    #  Ensure loop entry
3   attempts = 0    #  Count the number of tries
4
5   #  Loop until the user supplies a valid number
6   while in_value < 1 or in_value > 10:
7       in_value = int(input("Please enter an integer in the range 0–10: "))
8       attempts += 1
9
10  #  Make singular or plural word as necessary
11  tries = "try" if attempts == 1 else "tries"
12  #  in_value at this point is guaranteed to be within range
13  print("It took you", attempts, tries, "to enter a valid number")
```

A sample run of Listing 5.23 (`betterinputonly.py`) produces

```
Please enter an integer in the range 0-10: 11
Please enter an integer in the range 0-10: 12
Please enter an integer in the range 0-10: 13
Please enter an integer in the range 0-10: 14
Please enter an integer in the range 0-10: -1
Please enter an integer in the range 0-10: 5
It took you 6 tries to enter a valid number
```

We initialize the variable `in_value` at the top of the program only to make sure the loop's body executes at least one time.

5.8 Summary

- The `while` statement allows the execution of code sections to be repeated multiple times.

- The condition of the `while` controls the execution of statements within the `while`'s body.

- The statements within the body of a `while` are executed over and over until the condition of the `while` is false.

- If the `while`'s condition is initially false, the body is not executed at all.

- In an infinite loop, the `while`'s condition never becomes false.

- The statements within the `while`'s body must eventually lead to the condition being false; otherwise, the loop will be infinite.

- Do not confuse `while` statements with `if` statements; their structure is very similar (`while` reserved word instead of the `if` word), but they behave differently.

- Infinite loops are rarely intentional and usually are accidental.

- An infinite loop can be diagnosed by putting a printing statement inside its body.

- A loop contained within another loop is called a nested loop.

- Iteration is a powerful mechanism and can be used to solve many interesting problems.

- Complex iteration using nested loops mixed with conditional statements can be difficult to do correctly.

- The `break` statement immediately exits a loop, skipping the rest of the loop's body, without checking to see if the condition is true or false. Execution continues with the statement immediately following the body of the loop.

- In a nested loop, the `break` statement exits only the loop in which the `break` is found.

- The `continue` statement immediately checks the loop's condition, skipping the rest of the loop's body. If the condition is true, the execution continues at the top of the loop as usual; otherwise, the loop is terminated and execution continues with the statement immediately following the loop's body. false.

- In a nested loop, the `continue` statement affects only the loop in which the `continue` is found.

5.9 Exercises

1. In Listing 5.3 (addnonnegatives.py) could the condition of the if statement have used > instead of >= and achieved the same results? Why?

2. In Listing 5.3 (addnonnegatives.py) could the condition of the while statement have used > instead of >= and achieved the same results? Why?

3. In Listing 5.3 (addnonnegatives.py) what would happen if the statement

```
entry = eval(input())   #  Get the value
```

were moved out of the loop? Is moving the assignment out of the loop a good or bad thing to do? Why?

4. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
    print('*', end='')
    a += 1
print()
```

5. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
    print('*', end='')
print()
```

6. How many asterisks does the following code fragment print?

```
a = 0
while a > 100:
    print('*', end='')
    a += 1
print()
```

7. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
    b = 0;
    while b < 55:
        print('*', end='')
        b += 1
    print()
    a += 1
```

8. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
    if a  % 5 == 0:
```

```
                    print('*', end='')
                a += 1
        print()
```

9. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
    b = 0
    while b < 40:
        if (a + b) % 2 == 0:
            print('*', end='')
        b += 1
    print()
    a += 1
```

10. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
    b = 0
    while b < 100:
        c = 0
        while c < 100:
            print('*', end='')
            c++;
        b += 1
    a += 1
print()
```

11. How many asterisks does the following code fragment print?

```
for a in range(100):
    print('*', end='')
print()
```

12. How many asterisks does the following code fragment print?

```
for a in range(20, 100, 5):
    print('*', end='')
print()
```

13. How many asterisks does the following code fragment print?

```
for a in range(100, 0, -2):
    print('*', end='')
print()
```

14. How many asterisks does the following code fragment print?

```
for a in range(1, 1):
    print('*', end='')
print()
```

15. How many asterisks does the following code fragment print?

    ```python
    for a in range(-100, 100):
        print('*', end='')
    print()
    ```

16. How many asterisks does the following code fragment print?

    ```python
    for a in range(-100, 100, 10):
        print('*', end='')
    print()
    ```

17. Rewrite the code in the previous question so it uses a while instead of a for. Your code should behave identically.

18. How many asterisks does the following code fragment print?

    ```python
    for a in range(-100, 100, -10):
        print('*', end='')
    print()
    ```

19. How many asterisks does the following code fragment print?

    ```python
    for a in range(100, -100, 10):
        print('*', end='')
    print()
    ```

20. How many asterisks does the following code fragment print?

    ```python
    for a in range(100, -100, -10):
        print('*', end='')
    print()
    ```

21. What is printed by the following code fragment?

    ```python
    a = 0
    while a < 100:
        print(a)
        a += 1
    print()
    ```

22. Rewrite the code in the previous question so it uses a for instead of a while. Your code should behave identically.

23. What is printed by the following code fragment?

    ```python
    a = 0
    while a > 100:
        print(a)
        a += 1
    print()
    ```

24. Rewrite the following code fragment using a break statement and eliminating the done variable. Your code should behave identically to this code fragment.

```
        done = False
        n, m = 0, 100
        while not done and n != m:
            n = eval(input())
            if n < 0:
                done = true
            print("n =", n)
```

25. Rewrite the following code fragment so it does not use a `break` statement. Your code should behave identically to this code fragment.

    ```
    //  Code with break ...
    ```

26. Rewrite the following code fragment so it eliminates the `continue` statement. Your new code's logic should be simpler than the logic of this fragment.

    ```
    x = 100
    while x > 0:
        y = eval(input())
        if y == 25:
            x += 1
            continue
        x = eval(input())
        print('x =', x)
    ```

27. What is printed by the following code fragment?

    ```
    a = 0
    while a < 100:
        print(a, end='')
        a += 1
    print()
    ```

28. Modify Listing 5.9 (`flexibletimestable.py`) so that the it requests a number from the user. It should then print a multiplication table of the size entered by the user; for example, if the users enters 15, a 15×15 table should be printed. Print nothing if the user enters a value lager than 18. Be sure everything lines up correctly, and the table looks attractive.

29. Write a Python program that accepts a single integer value entered by the user. If the value entered is less than one, the program prints nothing. If the user enters a positive integer, n, the program prints an $n \times n$ box drawn with $\star$ characters. If the users enters 1, for example, the program prints

    If the user enters a 2, it prints

    An entry of three yields

```
   ***
   ***
   ***
```

and so forth. If the user enters 7, it prints

```
   *******
   *******
   *******
   *******
   *******
   *******
   *******
```

that is, a 7×7 box of `*` symbols.

30. Write a Python program that allows the user to enter exactly twenty floating-point values. The program then prints the sum, average (arithmetic mean), maximum, and minimum of the values entered.

31. Write a Python program that allows the user to enter any number of non-negative floating-point values. The user terminates the input list with any negative value. The program then prints the sum, average (arithmetic mean), maximum, and minimum of the values entered. The terminating negative value is **not** used in the computations.

32. Redesign Listing 5.19 (`startree.py`) so that it draws a sideways tree pointing right; for example, if the user enters 7, the program would print

```
*
**
***
****
*****
******
*******
******
*****
****
***
**
*
```

33. Redesign Listing 5.19 (`startree.py`) so that it draws a sideways tree pointing left; for example, if the user enters 7, the program would print

```
        *
       **
      ***
     ****
    *****
   ******
  *******
   ******
    *****
     ****
      ***
       **
        *
```
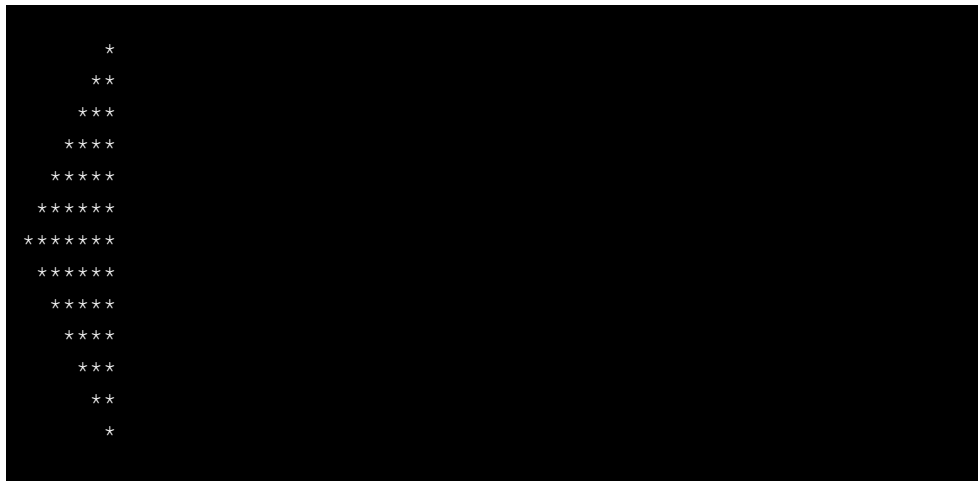
Chapter 6

Using Functions

Recall the square root code we wrote in Listing 5.18 (`computesquareroot.py`). In it we used a loop to compute the approximate square root of a value provided by the user. While this code may be acceptable for many applications, better algorithms exist that work faster and produce more precise answers. Another problem with this code is that it is not packaged in a way to be used flexibly in other programs. What if you are working on a significant scientific or engineering application and must compute square roots in various places throughout the source code? Must we copy and paste the relevant portions of this square root code to each site in our source code that requires a square root computation? Also, what if we develop other programs that use square root? Do we copy this code into every program that needs to compute square roots, or is there a better way to package the square root code and reuse it?

Code is made more reusable by packaging it in *functions*. A function is a unit of reusable code. In Chapter 7 we will see how to write our own reusable functions, but in this chapter we examine some of the functions available in the Python standard library. Python provides a collection of standard code stored in libraries called *modules*. Programmers can use parts of this library code within their own code to build sophisticated programs.

6.1 Introduction to Using Functions

We have been using functions in Python since the first chapter. These functions include `print`, `input`, `eval`, `int`, `float`, `range`, and `type`. The Python standard library includes many other functions useful for common programming tasks.

In mathematics, a *function* computes a result from a given value; for example, from the function definition $f(x) = 2x + 3$ we can compute $f(5) = 13$ and $f(0) = 3$. A function in Python works like a mathematical function. To introduce the function concept, we will look at the standard Python function that implements mathematical square root.

In Python, a function is a named block of code that performs a specific task. A program uses a function when specific processing is required. One example of a function is the mathematical square root function. Python has a function in its standard library named `sqrt` (see Section 6.2). The square root function accepts one numeric (integer or floating-point) value and produces a floating-point result; for example, $\sqrt{16} = 4$, so when presented with 16.0, `sqrt` responds with 4.0. Figure 6.1 illustrates the conceptual view of the `sqrt` function. To the user of the square root function, the function is a black box; the user is concerned more about *what* the function does, not *how* it does it.

Figure 6.1: Conceptual view of the square root function

This `sqrt` function is exactly what we need for our square root program, Listing 5.18 (`computesquareroot.py`). The new version, Listing 6.1 (`standardsquareroot.py`), uses the library function `sqrt` and eliminates the complex logic of the original code.

Listing 6.1: `standardsquareroot.py`

```
 1  from math import sqrt
 2
 3  #  Get value from the user
 4  num = eval(input("Enter number: "))
 5
 6  #  Compute the square root
 7  root = sqrt(num);
 8
 9  #  Report result
10  print("Square root of", num, "=", root)
```

The expression

sqrt(num)

is a *function invocation*, also known as a *function call*. A function provides a service to the code that uses it. Here, our code in Listing 6.1 (`standardsquareroot.py`) is the *calling code*, or *client code*. Our code is the client that uses the service provided by the `sqrt` function. We say our code *calls*, or *invokes*, `sqrt` passing it the value of `num`. The expression `sqrt(num)` evaluates to the square root of the value of the variable `num`.

The interpreter is not automatically aware of the `sqrt` function. The `sqrt` function is not part of the small collection of functions (like `type`, `int`, `str`, and `range`) always available to Python programs. The `sqrt` function is part of separate *module*. A module is a collection of Python code that can used in other programs. The statement

from math import sqrt

makes the `sqrt` function available for use in the program. The math module has many other mathematical functions. These include trigonometric, logarithmic, hyperbolic, and other mathematical functions.

When calling a function, the function's name is followed by parentheses that contain the information to pass to the function so it can perform its task. In the expression

sqrt(num)

num is the information the function needs to do its work. We say num is the *argument*, or *parameter*, passed to the function. The function cannot change the value of num as far as the caller is concerned, it simply uses the variable's value to perform the computation. It is as if we write down the value of num, hand it to sqrt, and sqrt hands us back a note with the answer. The sqrt function can be called many in other ways, as illustrated in Listing 6.2 (usingsqrt.py):

Listing 6.2: usingsqrt.py

```
1  # This program shows the various ways the
2  # sqrt function can be used.
3
4  from math import sqrt
5
6  x = 16
7  #  Pass a literal value and display the result
8  print(sqrt(16.0))
9  #  Pass a variable and display the result
10 print(sqrt(x))
11 #  Pass an expression
12 print(sqrt(2 * x - 5))
13 #  Assign result to variable
14 y = sqrt(x)
15 print(y)
16 #  Use result in an expression
17 y = 2 * sqrt(x + 16) - 4
18 print(y)
19 #  Use result as argument to a function call
20 y = sqrt(sqrt(256.0))
21 print(y)
22 print(sqrt(int('45')))
```

The sqrt function accepts a single numeric argument. As Listing 6.2 (usingsqrt.py) shows, the parameter that a client can pass to sqrt can be a literal number, a numeric variable, an arithmetic expression, or even a function invocation that produces a numeric result.

Some functions, like sqrt, compute a value that is returned to the client. The client can use this result in various ways, as shown in Listing 6.2 (usingsqrt.py). The statement

```
print(sqrt(16.0))
```

directly prints the result of computing the square root of 16. The statement

```
y = sqrt(x)
```

assigns the result of the function call to the variable y. The statement

```
y = sqrt(sqrt(256.0))
```

computes $\sqrt{\sqrt{256}} = \sqrt{16} = 4$. The statement

```
print(sqrt(int('45')))
```

prints the result of computing the square root of the integer equivalent of the string '45'.

If the client code attempts to pass a parameter to a function that is incompatible with type expected by that function, the interpreter issues an error. Consider:

```python
print(sqrt("16"))  # Illegal, a string is not a number
```

In the interactive shell we get

```
>>> from math import sqrt
>>>
>>> sqrt(16)
4.0
>>> sqrt("16")
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    sqrt("16")
TypeError: a float is required
```

The sqrt function can process only numbers: integers and floating-point numbers. Even though we know we could convert the string parameter '16' to the integer 16 (with the int function) or to the floating-point value 16.0 (with the float function), the sqrt function does not automatically do this for us.

Listing 6.2 (usingsqrt.py) shows that a program can call the sqrt function as many times and in as many places as needed. As noted in Figure 6.1, to the client of the square root function, the function is a black box; the client is concerned strictly about *what* the function does, not *how* the function accomplishes its task.

We safely can treat all functions like black boxes. We can use the service that a function provides without being concerned about its internal details. We are guaranteed that we can influence the function's behavior only via the parameters that we pass, and that nothing else we do can affect what the function does or how it does it. Furthermore, for the types of objects we have considered so far (integers, floating-point numbers, and strings), when a client passes data to a function, the function cannot affect the client's copy of that data. The client is, however, free to use the return value of function to modify any of its variables. The important distinction is that the client is modifying its variables, and a function cannot modify a client's variables.

Some functions take more than one parameter; for example, we have seen the range function that accepts one, two, or three parameters.

From the client's perspective a function has three important parts:

- **Name**. Every function has a name that identifies the code to be executed. Function names follow the same rules as variable names; a function name is another example of an identifier (see Section 2.3).

- **Parameters**. A function must be called with a certain number of parameters, and each parameter must be the correct type. Some functions, like print and range, permit clients to pass a variable number of arguments, but most functions, like sqrt, specify an exact number. If a client attempts to call a function with too many or too few parameters, the interpreter will issue an error message and refuse to run the program. Consider the following misuse of sqrt in the interactive shell:

```
>>> sqrt(10)
3.1622776601683795
>>> sqrt()
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    sqrt()
TypeError: sqrt() takes exactly one argument (0 given)
>>> sqrt(10, 20)
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    sqrt(10, 20)
TypeError: sqrt() takes exactly one argument (2 given)
```

Similarly, if the parameters the client passes during a call are not compatible with the types specified for the function, the interpreter reports appropriate error messages:

```
>>> sqrt(16)
4.0
>>> sqrt("16")
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    sqrt("16")
TypeError: a float is required
```

- **Result type**. A function returns a value to its caller. Generally a function will compute a result and return the value of the result to the client. The client's use of this result must be compatible with the function's specified result type. A function's result type and its parameter types can be completely unrelated.

Some functions do not accept any parameters; for example, the function to generate a pseudorandom floating-point number, random, requires no arguments:

```
>>> from random import random
>>> random()
0.9595266948278349
```

The random function is part of the random package. The random function returns a floating-point value, but the client does not pass the function any information to do its task. Any attempts to do so will fail:

```
>>> random.random(20)
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    random.random(20)
TypeError: random() takes no arguments (1 given)
```

Like mathematical functions that must produce a result, a Python function always produces a value to return to the client. Some functions are not designed to produce any useful results. Clients call such a function for the effects provided by the executing code within a function, not for any value that the function computes. The `print` function is one such example. The `print` function displays text in the console window; it does not compute and return a value to the client. Since Python requires that all functions return a value, `print` must return something. Functions that are not meant to return anything return the special value `None`. We can show this in the Python shell:

```
>>> print(print(4))
4
None
```

The 4 is printed by the inner `print` call, and the outer `print` displays the return value of the inner `print` call.

6.2 Standard Mathematical Functions

The standard `math` module provides much of the functionality of a scientific calculator. Table 6.1 lists only a few of the available functions.

`mathfunctions` Module
`sqrt`   Computes the square root of a number: $\texttt{sqrt}(x) = \sqrt{x}$
`exp`   Computes e raised a power: $\texttt{exp}(x) = e^x$
`log`   Computes the natural logarithm of a number: $\texttt{log}(x) = \log_e x = \ln x$
`log10`   Computes the common logarithm of a number: $\texttt{log}(x) = \log_{10} x$
`cos`   Computes the cosine of a value specified in radians: $\texttt{cos}(x) = \cos x$; other trigonometric functions include sine, tangent, arc cosine, arc sine, arc tangent, hyperbolic cosine, hyperbolic sine, and hyperbolic tangent
`pow`   Raises one number to a power of another: $\texttt{pow}(x, y) = x^y$
`degrees`   Converts a value in radians to degrees: $\texttt{degrees}(x) = \frac{\pi}{180} x$
`radians`   Converts a value in degrees to radians: $\texttt{radians}(x) = \frac{180}{\pi} x$
`fabs`   Computes the absolute value of a number: $\texttt{fabs}(x) = |x|$

Table 6.1: A few of the functions from the `math` package

The `math` package also defines the values `pi` (π) and `e` (e).

The parameter passed by the client is known as the *actual* parameter. The parameter specified by the function is called the *formal* parameter. During a function call the first actual parameter is assigned to the first formal parameter, the second actual parameter is assigned to the second formal parameter, etc. Callers must be careful to put the arguments they pass in the proper order when calling a function. The call `pow(10,2)` computes $10^2 = 100$, but the call `pow(2,10)` computes $2^{10} = 1,024$.

A Python program that uses any of these mathematical functions must import the `math` module.

Figure 6.2 shows a problem that can be solved using functions found in the `math` module. Suppose a spacecraft is at a fixed location in space some distance from a planet. A satellite is orbiting the planet in a circular orbit. We wish to compute how much farther away the satellite will be from the spacecraft when it has progressed 10 degrees along its orbital path.
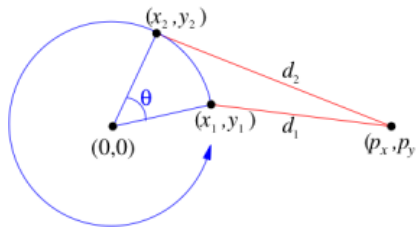

Figure 6.2: Orbital distance problem

We will let the origin of our coordinate system (0,0) be located at the center of the planet which corresponds also to the center of the circular orbital path. The satellite is initially at point (x_1, y_1) and the spacecraft is stationary at point (p_x, p_y). The spacecraft is located in the same plane as the satellite's orbit. We need to compute the difference in the distances between the moving point (satellite) and the fixed point (spacecraft) at two different times during the satellite's orbit.

Two problems must be solved, and facts from mathematics provide the answers:

1. **Problem**: The location of the moving point must be recomputed as it moves along the circle.

   Solution: Given an initial position (x_1, y_1) of the moving point, a rotation of θ degrees around the origin will yield a new point at (x_2, y_2), where

   $$\begin{aligned} x_2 &= x_1 \cos\theta - y_1 \sin\theta \\ y_2 &= x_1 \sin\theta + y_1 \cos\theta \end{aligned}$$

2. **Problem**: The distance between the moving point and the fixed point must be recalculated as the moving point moves to a new position.

   Solution: The distance d_1 in Figure 6.2 between two points (p_x, p_y) and (x_1, y_1) is given by the formula

   $$d_1 = \sqrt{(x_1 - p_x)^2 + (y_1 - p_y)^2}$$

   Similarly, the distance d_2 in Figure 6.2 is

   $$d_2 = \sqrt{(x_2 - p_x)^2 + (y_2 - p_y)^2}$$

Listing 6.3 (`orbitdist.py`) uses these mathematical results to compute the difference in the distances.

Listing 6.3: `orbitdist.py`

```
1   #  Use some functions and values from the math package
2   from math import sqrt, sin, cos, pi
3
4   #  Location of orbiting point is (x,y)
5   #  Location of fixed point is always (100, 0),
6   #  AKA (p_x, p_y).  Change these as necessary.
7   p_x = 100
8   p_y = 0
9
10  #  Radians in 10 degrees
11  radians = 10 * pi/180
12
13  #  Precompute the cosine and sine of 10 degrees
14  COS10 = cos(radians)
15  SIN10 = sin(radians)
16
17  #  Get starting point from user
18  x, y = eval(input("Enter initial satellite coordinates (x,y):"))
19
20  #  Compute the initial distance
21  d1 = sqrt((p_x - x)*(p_x - x) + (p_y - y)*(p_y - y))
22
23  #  Let the satellite orbit 10 degrees
24  x_old = x;              #  Remember x's original value
25  x = x*COS10 - y*SIN10   #  Compute new x value
26  #  x's value has changed, but y's calculate depends on
27  #  x's original value, so use x_old instead of x.
28  y = x_old*SIN10 + y*COS10
29
30  #  Compute the new distance
31  d2 = sqrt((p_x - x)*(p_x - x) + (p_y - y)*(p_y - y))
32
33  #  Print the difference in the distances
34  print("Difference in distances: %.3f" % (d2 - d1))
```

We can use the square root function to improve the efficiency of Listing 5.21 (`printprimes.py`). Instead of trying all the factors of n up to $n-1$, we need only try potential factors up to the square root of n. Listing 6.4 (`moreefficientprimes.py`) uses the `sqrt` function to reduce the number of factors that need be considered.

Listing 6.4: `moreefficientprimes.py`

```
1   from math import sqrt
2
3   max_value = eval(input('Display primes up to what value? '))
4   value = 2   #  Smallest prime number
5
6   while value <= max_value:
7       #  See if value is prime
8       is_prime = True  # Provisionally, value is prime
9       #  Try all possible factors from 2 to value - 1
10      trial_factor = 2
```

```
11      root = sqrt(value)
12      while trial_factor <= root:
13          if value % trial_factor == 0:
14              is_prime = False;    #  Found a factor
15              break                # No need to continue; it is NOT prime
16          trial_factor += 1        # Try the next potential factor
17      if is_prime:
18          print(value, end= ' ')   #  Display the prime number
19      value += 1                   # Try the next potential prime number
20
21  print()   #  Move cursor down to next line
```

6.3 `time` Functions

The `time` package contains a number of functions that relate to time. We will consider two: `clock` and `sleep`.

The `clock` function allows us measure the time of parts of a program's execution. The `clock` returns a floating-point value representing elapsed time in seconds. On Unix-like systems (Linux and Mac OS X), `clock` returns the numbers of seconds elapsed since the program began executing. Under Microsoft Windows, `clock` returns the number of seconds since the first call to `clock`. In either case, with two calls to the `clock` function we can measure *elapsed time*. Listing 6.5 (`timeit.py`) measures how long it takes a user to enter a character from the keyboard.

Listing 6.5: `timeit.py`
```
1  from time import clock
2
3  print("Enter your name: ", end="")
4  start_time = clock()
5  name = input()
6  elapsed = clock() - start_time
7  print(name, "it took you", elapsed, "seconds to respond")
```

The following represents the program's interaction with a particularly slow typist:

```
Enter your name: Rick
Rick it took you 7.246477029927183 seconds to respond
```

Listing 6.6 (`timeaddition.py`) measures the time it takes for a Python program to add up all the integers from 1 to 100,000,000.

Listing 6.6: `timeaddition.py`
```
1  from time import clock
2
3  sum = 0          #  Initialize sum accumulator
4  start = clock()  #  Start the stopwatch
5  for n in range(1, 100000001):   #   Sum the numbers
6      sum += n
```

```
7  elapsed = clock() - start   #   Stop the stopwatch
8  print("sum:", sum, "time:", elapsed)   # Report results
```

On one system Listing 6.6 (`timeaddition.py`) reports

```
sum: 5000000050000000 time: 20.663551324385658
```

Listing 6.7 (`measureprimespeed.py`) measures how long it takes a program to count all the prime numbers up to 10,000 using the same algorithm as Listing 5.22 (`printprimesfor.py`).

Listing 6.7: **measureprimespeed.py**

```
1  from time import clock
2
3  max_value = 10000
4  count = 0
5  start_time = clock()    #  Start timer
6  #  Try values from 2 (smallest prime number) to max_value
7  for value in range(2, max_value + 1):
8      #  See if value is prime
9      is_prime = True  # Provisionally, value is prime
10     #  Try all possible factors from 2 to value - 1
11     for trial_factor in range(2, value):
12         if value % trial_factor == 0:
13             is_prime = False    #  Found a factor
14             break                # No need to continue; it is NOT prime
15     if is_prime:
16         count += 1               # Count the prime number
17  print()   #  Move cursor down to next line
18  elapsed = clock() - start_time  # Stop the timer
19  print("Count:", count, "  Elapsed time:", elapsed, "sec")
```

On one system, the program took about 1.25 seconds, on average, to count all the prime numbers up to 10,000. By comparison, Listing 6.8 (`timemoreefficientprimes.py`), based on the algorithm in Listing 6.4 (`moreefficientprimes.py`) using the square root optimization runs over 10 times faster. Exact times will vary depending on the speed of the computer.

Listing 6.8: **timemoreefficientprimes.py**

```
1  from math import sqrt
2  from time import clock
3
4  max_value = 10000
5  count = 0
6  value = 2          #  Smallest prime number
7  start = clock()   #  Start the stopwatch
8  while value <= max_value:
9      #  See if value is prime
10     is_prime = True  # Provisionally, value is prime
11     #  Try all possible factors from 2 to value - 1
12     trial_factor = 2
13     root = sqrt(value)
```

```
14        while trial_factor <= root:
15            if value % trial_factor == 0:
16                is_prime = False;      #  Found a factor
17                break                  # No need to continue; it is NOT prime
18            trial_factor += 1          # Try the next potential factor
19        if is_prime:
20            count += 1                 # Count the prime number
21        value += 1                     # Try the next potential prime number
22 elapsed = clock() - start             # Stop the stopwatch
23 print("Count:", count, "  Elapsed time:", elapsed, "sec")
```

An even faster prime generator can be found in Listing 9.19 (`fasterprimes.py`); it uses a completely different algorithm to generate prime numbers.

The `sleep` function suspends the program's execution for a specified number of seconds. Listing 6.9 (`countdown.py`) counts down from 10 with one second intervals between numbers.

Listing 6.9: `countdown.py`

```
1 from time import sleep
2
3 for count in range(10, -1, -1):  #  Range 10, 9, 8, ..., 0
4     print(count)         #  Display the count
5     sleep(1)             #  Suspend execution for 1 second
```

The `sleep` function is useful for controlling the speed of graphical animations.

6.4 Random Numbers

Some applications require behavior that appears random. Random numbers are useful particularly in games and simulations. For example, many board games use a die (one of a pair of dice) to determine how many places a player is to advance. (See Figure 6.3.) A die or pair of dice are used in other games of chance. A die is a cube containing spots on each of its six faces. The number of spots range from one to six. A player rolls a die or sometimes a pair of dice, and the side(s) that face up have meaning in the game being played. The value of a face after a roll is determined at random by the complex tumbling of the die. A software adaptation of a game that involves dice would need a way to simulate the random roll of a die.
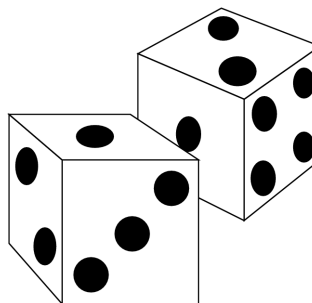

Figure 6.3: A pair of dice

All algorithmic random number generators actually produce *pseudorandom* numbers, not true random numbers. A pseudorandom number generator has a particular period, based on the nature of the algorithm used. If the generator is used long enough, the pattern of numbers produced repeats itself exactly. A sequence of true random numbers would not contain such a repeating subsequence. The good news is that all practical algorithmic pseudorandom number generators have periods that are large enough for most applications.

The Python `random` module contains a number of standard functions that programmers can use for working with pseudorandom numbers. A few of these functions are shown in Table 6.2.

`randomfunctions` Module
`random`   Returns a pseudorandom floating-point number x in the range $0 \leq x < 1$
`randrange`   Returns a pseudorandom integer value within a specified range.
`seed`   Sets the random number seed.

Table 6.2: A few of the functions from the `random` package

The `seed` function establishes the initial value from which the sequence of pseudorandom numbers is generated. Each call to `random` or `randrange` returns the next value in the sequence of pseudorandom values. Listing 6.10 (`simplerandom.py`) prints 100 pseudorandom integers in the range $1 \ldots 100$.

Listing 6.10: `simplerandom.py`

```
from random import randrange, seed

seed(23)                                 #  Set random number seed
for i in range(0, 100):                  #  Print 100 random numbers
    print(randrange(1, 1000), end=' ')   #  Range 1...1,000
print()                                  #  Print newine
```

The numbers printed by the program appear to be random. The algorithm is given a seed value to begin, and a formula is used to produce the next value. The seed value determines the sequence of numbers generated; identical seed values generate identical sequences. If you run the program again, the same sequence is displayed, because the same seed value, 23, is used. In order to allow each program run to display different sequences, the seed value must be different for each run.

If we omit the call to `seed`, the initial value in the sequence is based on the system's time. This usually is adequate for simple pseudorandom number sequences. Being able to specify a seed value is useful during development and testing when we want program executions to exhibit reproducible results.

We now have all we need to write a program that simulates the rolling of a die. Listing 6.11 (`die.py`) simulates rolling die.

Listing 6.11: `die.py`

```
from random import randrange

#  Roll the die three times
for  i in range(0, 3):
    #  Generate random number in the range 1...6
```

```
 6      value = randrange(1, 6)
 7
 8      #  Show the die
 9      print("+-------+")
10      if value == 1:
11          print("|       |")
12          print("|   *   |")
13          print("|       |")
14      elif value == 2:
15          print("| *     |")
16          print("|       |")
17          print("|     * |")
18      elif value == 3:
19          print("|     * |")
20          print("|   *   |")
21          print("| *     |")
22      elif value == 4:
23          print("| *   * |")
24          print("|       |")
25          print("| *   * |")
26      elif value == 5:
27          print("| *   * |")
28          print("|   *   |")
29          print("| *   * |")
30      elif value == 6:
31          print("| * * * |")
32          print("|       |")
33          print("| * * * |")
34      else:
35          print(" ***  Error: illegal die value ***")
36      print("+-------+")
```

The output of one run of Listing 6.11 (die.py) is

```
+-------+
| *   * |
|       |
| *   * |
+-------+
+-------+
| * * * |
|       |
| * * * |
+-------+
+-------+
|       |
|   *   |
|       |
+-------+
```

Since the values are pseudorandomly generated, actual output will vary from one run to the next.

6.5 Importing Issues

Python provides three ways to import functions from a module:

- Import one or more specific functions:

```python
from math import sqrt, log
```

  In this case, only the sqrt and log functions will be available. Even though the math module provides the atan function that computes the arctangent, this limited import statement does not provide its definition to the interpreter.

- Import everything the module has to offer:

```python
from math import *
```

  The * symbol represents "everything." This statement makes all the code in the math module available to the program. If a program needs to use many different functions from the math module, some programmers use this approach.

- Import the module itself instead of just its components:

```python
import math
```

  In this case, to use a function the client must use the following notation:

```python
y = math.sqrt(x)
print(math.log10(100))
```

  Note the math. prefix attached to sqrt and log10. We call a name like these a *qualified name*. The qualified name includes the module name and function name. Many programmers prefer this approach because the exact nature of the name is self evident.

Of the three varieties of import statements, the "import all" statement is in some ways the easiest to use. The mindset is, "Import everything, because we may need some things in the module, but we are not sure exactly what we need starting out." The source code is shorter: * is quicker to type than a list of function names, and short function names are easier to type than qualified function names. While in the short term the "import all" approach may appear to be attractive, in the long term it can lead to problems. As an example, suppose a programmer is writing a program that simulates a chemical reaction in which the rate of the reaction is related logarithmically to the temperature. The statement

```python
from math import log10
```

may cover all that this program needs from the math module. If the programmer instead uses

```python
from math import *
```

this statement imports everything, including a function named degrees which converts angle measurements in radians to degrees (from trigonometry, $360° = 2\pi$ radians). Given the nature of the program, the word degrees is a good name to use for a variable that represents temperature. The two words are the same, but their meanings are very different. The programmer is free to redefine degrees to be a floating-point variable (recall redefining the print function in Section 2.3), but then the math module's degrees function is unavailable if it is needed later. A *name collision* results if the programmer tries to use the same name for both the angle conversion and temperature representation. The same name cannot be simultaneously for both purposes.

We say that the "import everything" statement *pollutes* the program's namespace. The import adds many names (variables, functions, and other objects) to the collections of names managed by the program. This can cause name collisions as demonstrated with the name degrees, and it makes larger programs more difficult to work with and less maintainable.

To summarize, you should avoid the "import everything" statement

```python
from math import *
```

since this provides more opportunities for name collisions and makes your code less maintainable. The best approach imports the whole module

```python
import math
```

and uses qualified names for the functions the module provides. In the above example, this module import approach solves the name collision problem: math.degrees is a different name than degrees. A compromise imports only the functions needed:

```python
from math import sqrt, log
```

This does not impact the program's namespace very much, and it allows the program to use short function names. Also, by explicitly naming the functions to import, the programmer is more aware of how the names will impact the program.

You can think of a module as a toolbox. The math module is a box containing mathematics tools. The statement

```python
from math import *
```

is like bringing the math toolbox into your workroom and dumping everything out on the floor. It may be handy at times, but it makes a mess and can be dangerous (you might trip over one of the tools on the floor).

The statement

```python
import math
```

is like bringing the math toolbox into your workroom. When you need a mathematics tool you take it out of the box and use it. When you are finished with it, even if you may need it later, you put it back in the toolbox. If you need it later, you can take it out again because you know right where it is. It is a little more work, but it is more organized.

The statement

```python
from math import sqrt, log10
```

is like bringing the math toolbox into your workroom and taking out the two mathematics tools you need for a project. You don't put the tools back until you are finished with them completely. It is not as messy, and you are less likely to trip over a tool on the floor.

6.6 Summary

- The Python standard library provides a collection of functions that you can incorporate into code that you write.

- When faced with the choice of using a standard library function or writing your own code to solve the same problem, choose the library function. The standard function will be tested thoroughly, well documented, and likely more efficient than the code you would write.

- The function is a standard unit of reuse in Python.

- Code that uses a function is known as *client* code.

- A function has a name, a list of parameters (which may be empty), and a result (which may be None). A function performs some computation or action that is useful to clients. Typically a function produces a result based on the parameters passed to it.

- Clients communicate information to a function via its parameters (also known as arguments).

- Standard library functions are organized into modules.

- A module contains a collection of related functions.

- In order to use many standard functions, a client must use an import statement so that the interpreter will use function definitions from the proper module.

- The arguments passed to a function by a client consist of a comma-separated list enclosed by parentheses.

- Clients calling a function must pass the correct number and types of parameters that the function expects.

- The Python standard module math includes a variety of mathematical functions.

- The clock function from the time module may be used to measure the execution time of parts of programs.

- The sleep function suspends the program's execution for a specified number of seconds.

- The random module contains a number of functions for working with pseudorandom numbers.

- randrange(x, y) returns a pseudorandom integer in the range $x \ldots y$. random() returns a pseudorandom floating-point number x in the range $0 \leq x < 1$.

- There are three ways to import functions from modules: import certain functions only, import everything, and import the module itself as a unit.

- The complete module import is the best approach, but it requires programmers to use the longer qualified names for functions.

- You should avoid the "import everything" from a module statement. This pollutes the program's namespace and can make programs less maintainable.

- The limited import approach is a comprise between importing everything and importing the module as a unit.

6.7 Exercises

1. Suppose you need to compute the square root of a number in a Python program. Would it be a good idea to write the code to perform the square root calculation? Why or why not?

2. Which of the following values could be produced by the call `random.randrange(0, 100)` function (circle all that apply)?

   4.5   **34**   **−1**   **100**   **0**   **99**

3. Classify each of the following expressions as *legal* or *illegal*. Each expression represents a call to a standard Python library function.

   (a) `math.sqrt(4.5)`

   (b) `math.sqrt(4.5, 3.1)`

   (c) `random.rand(4)`

   (d) `random.seed()`

   (e) `random.seed(-1)`

4. From geometry: Write a computer program that, given the lengths of the two sides of a right triangle adjacent to the right angle, computes the length of the hypotenuse of the triangle. (See Figure **??**.) If you are unsure how to solve the problem mathematically, do a web search for the *Pythagorean theorem*.
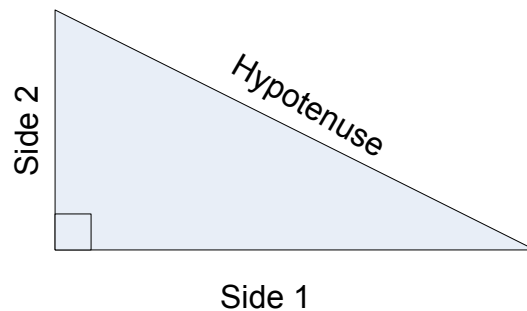
Figure 6.4: Right triangle

5. Write a guessing game program in which the computer chooses at random an integer in the range $1\ldots100$. The user's goal is to guess the number in the least number of tries. For each incorrect guess the user provides, the computer provides feedback whether the user's number is too high or too low.

6. Extend Problem 5 by keeping track of the number of guesses the user needed to get the correct answer. Report the number of guesses at the end of the game.

7. Extend Problem 6 by measuring how much time it takes for the user to guess the correct answer. Report the time and number of guesses at the end of the game.

Chapter 7

Writing Functions

As programs become more complex, programmers must structure their programs in such a way as to effectively manage their complexity. Most humans have a difficult time keeping track of too many pieces of information at one time. It is easy to become bogged down in the details of a complex problem. The trick to managing complexity is to break down the problem into more manageable pieces. Each piece has its own details that must be addressed, but these details are hidden as much as possible within that piece. The problem ultimately is solved by putting these pieces together to form the complete solution.

So far all of the code we have written has been placed within a single block of code. That single block may have contained sub-blocks for the bodies of structured statements like `if` and `while`, but the program's execution begins with the first statement in the block and ends when the last statement in that block is finished. Even though all of the code we have written has been limited to one, sometimes big, block, our programs all have executed code outside of that block. All the functions we have used—`print`, `input`, `range`, `sqrt`, `random`, etc.—represent blocks of code that some other programmers have written for us. These blocks of code have a structure that makes them reusable by any Python program.

As the number of statements within our block of code increases, the code can become unwieldy. A single block of code (like in all our programs to this point) that does all the work itself is called *monolithic code*. Monolithic code that is long and complex is undesirable for several reasons:

- **It is difficult to write correctly**. All the details in the entire piece of code must be considered when writing any statement within that code.

- **It is difficult to debug**. If the sequence of code does not work correctly, it is often difficult to find the source of the error. The effects of an erroneous statement that appears earlier in the code may not become apparent until a correct statement later uses the erroneous statement's incorrect result.

- **It is difficult to extend**. All the details in the entire sequence of code must be well understood before it can be modified. If the code is complex, this may be a formidable task.

We can write our own functions to divide our code into more manageable pieces. Using a divide and conquer strategy, a programmer can decompose a complicated block of code into several simpler functions. The original code then can do its job by delegating the work to these functions. Besides their code organization aspects, functions allow us to bundle functionality into reusable parts. In Chapter 6 we saw how library functions can dramatically increase the capabilities of our programs. While we should capitalize on library functions as much as possible, sometimes we need a function exhibiting custom behavior that is not provided by any standard function. Fortunately, we can create our own functions, and the same function

may be used (called) in numerous places within a program. If the function's purpose is general enough and we write the function properly, we may be able to reuse the function in other programs as well.

7.1 Function Basics

There are two aspects to every Python function:

- **Function definition**. The definition of a function contains the code that determines the function's behavior. Function definition is described in Section 7.2.

- **Function invocation**. A function is used within a program via a function invocation. In Chapter 6, we invoked standard functions that we did not have to define ourselves. Every function has exactly one definition but may have many invocations.

An ordinary function definition consists of three parts:

- **Name**—Most Python functions have a name. The name is an identifier (see Section 2.3). As with variable names, the name chosen for a function should accurately portray its intended purpose or describe its functionality. (Python allows specialized anonymous function called `lambda` functions, but we defer their introduction until Chapter **??**.)

- **Parameters**—every function definition specifies the parameters that it accepts from callers. The parameters appear in a parenthesized comma-separated list. The list of parameters is empty if the function requires no information from code that calls the function.

- **Body**—every function definition has a block of indented statements that constitute the function's body. The body contains the code to execute when clients invoke the function. The code within the body is responsible for producing the result, if any, to return to the client.

Figure 7.1 dissects a typical function definition.

The simplest function accepts no parameters and returns no value to the caller. The `def` keyword introduces a function definition, as shown in Listing 7.1 (`simplefunction.py`). Listing 7.1 (`simplefunction.py`) is a variation of Listing 3.1 (`adder.py`).

Listing 7.1: `simplefunction.py`

```
1  #  Print a message to prompt the user for input
2  def prompt():
3      print("Please enter an integer value: ", end="")
4
5  #  Start of program
6  print("This program adds together two integers.")
7  prompt()    #  Call the function
8  value1 = int(input())
9  prompt()    #  Call the function again
10 value2 = int(input())
11 sum = value1 + value2;
12 print(value1, "+", value2, "=", sum)
```
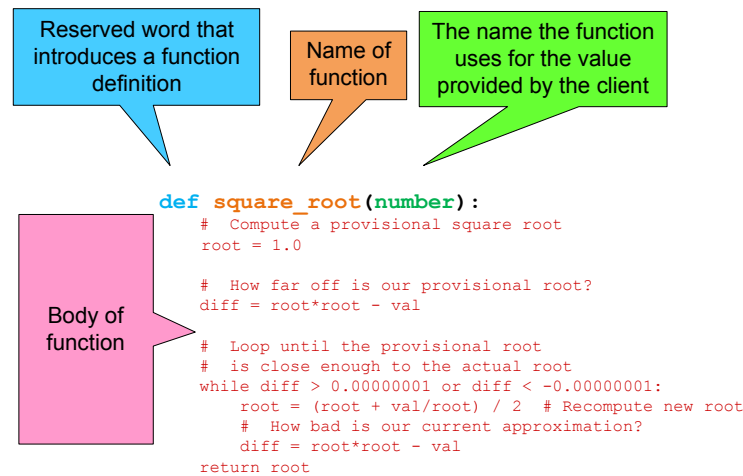
The two lines

Figure 7.1: Function definition dissection

```python
def prompt():
    print("Please enter an integer value: ", end="")
```

make up the prompt function definition. When called, the function simply prints the message *Please enter an integer value:* and leaves the cursor on the same line. The program runs as follows:
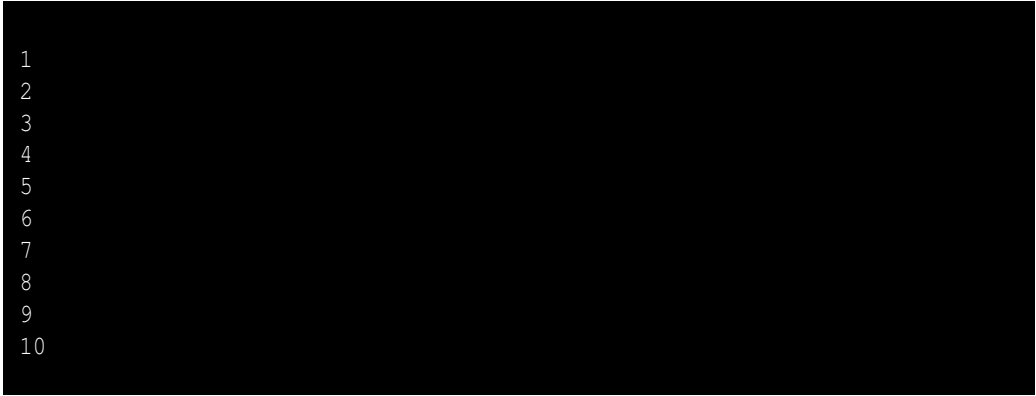
1. The program's execution begins with the first line in the "naked" block; that is, the block that is not part of the function definition.

2. The first executable statement prints the message of the program's intent.

3. The next statement is a call of the prompt function. At this point the program's execution transfers to the body of the prompt function. The code within prompt is executed until the end of its body or until a return statement is encountered. Since prompt contains no return statement, all of prompt's body (the one print statement) will be executed.

4. When prompt is finished, control is passed back to the point in the code immediately *after* the call of prompt.

5. The next action after prompt call reads the value of value1 from the keyboard.

6. A second call to prompt transfers control back to the code within the prompt function. It again prints its message.

7. When the second call to prompt is finished, control passes back to the point of the second input statement that assigns value2 from the keyboard.

8. The remaining two statements in the code, the arithmetic and printing statements, are executed, and then the program's execution terminates.

As another simple example, consider Listing 7.2 (`countto10.py`).

Listing 7.2: `countto10.py`

```
1  #  Counts to ten
2  for i in range(1, 11):
3      print(i)
```

which simply counts to ten:

```
1
2
3
4
5
6
7
8
9
10
```

If counting to ten in this way is something we want to do frequently within a program, we can write a function as shown in Listing 7.3 (`countto10func.py`) and call it as many times as necessary.

Listing 7.3: `countto10func.py`

```
1  #  Count to ten and print each number on its own line
2  def count_to_10():
3      for i in range(1, 11):
4          print(i)
5
6  print("Going to count to ten . . .")
7  count_to_10()
8  print("Going to count to ten again. . .")
9  count_to_10()
```

Our `prompt` and `countto10` functions are a bit underwhelming. The `prompt` function could be eliminated, and each call to `prompt` could be replaced with the statement in its body. The same could be said for the `countto10` function, although it is convenient to have the simple one-line statement that hides the complexity of the loop. Using the `prompt` function does have one advantage, though. If `prompt` is removed and the two calls to `prompt` are replaced with the print statement within prompt, we have to make sure that the two messages printed are identical. If we simply call `prompt`, we know the two messages printed will be identical, because only one possible message can be printed (the one in the body of `prompt`).

Our experience with using functions like `print` and `range` tells us that we can alter the behavior of some functions by passing different parameters. The following successive calls to the `print` function produces different results:

```
print('Hi')
print('Bye')
```

The two statements produce different results, of course, because we pass to the `print` function two different strings. If a function is written to accept information from the client (caller), the client must supply the information in order to use the function. The caller communicates the information via one or more parameters as required by the function. The `countto10` function does us little good if we sometimes want to count up to a different number. Listing 7.4 (`countton.py`) generalizes Listing 7.3 (`countto10func.py`) to count as high as the client needs.

Listing 7.4: `countton.py`

```
1  #  Count to n and print each number on its own line
2  def count_to_n(n):
3      for i in range(1, n + 1):
4          print(i)
5
6  print("Going to count to ten . . .")
7  count_to_n(10);
8  print("Going to count to five . . .")
9  count_to_n(5);
```

When the client code issues the call

```
count_to_n(10)
```

the argument 10 is known as the actual parameter. In the function definition, the parameter named `n` is called the formal parameter. During the call

```
count_to_n(10)
```

the actual parameter 10 is assigned to the formal parameter `n` before the function's statements begin executing.

A client must pass exactly one integer parameter to `countton` during a call. An attempt to pass no parameters or more than parameter results in a syntax error:

```
count_to_n()        #  Error, missing parameter during the call
count_to_n(3, 5)    #  Error, too many parameters during the call
```

An attempt to pass a non-integer results in a run-time exception, because the parameter is passed on to the `range` function, and the `range` function requires its parameters to be integers.

```
count_to_n(3.2)    #  Run-time error, actual parameter not an integer
```

We can enhance the `prompt` function's capabilities as shown in Listing 7.5 (`betterprompt.py`)

Listing 7.5: `betterprompt.py`

```
1  #  Definition of the prompt function
2  def prompt():
3      value = int(input("Please enter an integer value: "))
4      return value
5
6  print("This program adds together two integers.")
7  value1 = prompt()    #  Call the function
8  value2 = prompt()    #  Call the function again
9  sum = value1 + value2
10 print(value1, "+", value2, "=", sum)
```

In this version, `prompt` takes care of the input, so the client code does not have to use any input statements. The assignment statement

value1 = prompt()

implies `prompt` now produces a result we can assign to a variable or use in some other way. The last statement in the `prompt` function's definition is a `return` statement. A `return` statement specifies the exact result to return to the caller. When a `return` is encountered during a function's execution, control immediately passes back to the caller. The value of the function call is the value specified by the `return` statement, so the statement

value1 = prompt()

assigns to the variable `value1` the quantity associated with the `return` statement during `prompt`'s execution.

Note that in Listing 7.5 (`betterprompt.py`), we used a variable named `value` inside the `prompt` function. This variable is *local* to the function, meaning we cannot use this particular variable outside of `prompt`. It also means we are free to use that same name outside of the `prompt` function in a different context, and doing so will not interfere with the `value` variable within `prompt`. We say that `value` is a *local variable*.

We can further enhance our `prompt` function. Currently `prompt` always prints the same message. Using parameters, we can customize the message that `prompt` prints. Listing 7.6 (`evenbetterprompt.py`) shows how parameters are used to provide a customized message within `prompt`.

Listing 7.6: evenbetterprompt.py

```python
 1  #  Definition of the prompt function
 2  def prompt(n):
 3      value = int(input("Please enter integer #", n, ": ", sep=""))
 4      return value
 5
 6  print("This program adds together two integers.")
 7  value1 = prompt(1)      #  Call the function
 8  value2 = prompt(2)      #  Call the function again
 9  sum = value1 + value2
10  print(value1, "+", value2, "=", sum)
```

In Listing 7.6 (`evenbetterprompt.py`), the parameter influences the message that it printed. The user is now prompted to enter value #1 or value #2. The call

value1 = prompt(1)

passes the integer 1 to the `prompt` function. Since `prompt`'s parameter is named `n`, the process works as if the assignment statement

n = 1

were executed as the first action within `prompt`.

In the first line of the function definition:

def prompt(n):

`n` is the formal parameter. A formal parameter is used like a variable within the function's body, and it is local to the function.

At the point of the function call:

```
value1 = prompt(1)
```

The parameter passed in, 1, is the actual parameter. An *actual* parameter is the parameter *actually* used during a call of the function. When a client calls a function, any actual parameters provided by the client are assigned to their corresponding formal parameters, and the function begin executing. Said another way, during a function call the actual parameters are *bound* to their corresponding formal parameters.

When the call

```
value1 = prompt(1)
```

is executed by the client code, and the statement

```
value = int(input("Please enter integer #", n, ": ", sep=""))
```

within the body of prompt is executed, n will have the value 1. Similarly, when the call

```
value2 = prompt(2)
```

is executed by the client code, and the statement

```
value = int(input("Please enter integer #", n, ": ", sep=""))
```

within the body of prompt is executed, n will have the value 2. In the case of

```
value1 = prompt(1)
```

n within prompt is bound to 1, and in the case of

```
value2 = prompt(2)
```

n within prompt is bound to 2.

7.2 Using Functions

The general form of a function definition is

$$\text{def } \textit{name} \text{ (} \textit{parameter\_list} \text{) :}$$
$$\textit{block}$$

- The reserved word def signifies the beginning of a function definition.

- The *name* of the function is an identifier (see 2.3). The function's name should indicate the purpose of the function.

- The *parameter_list* is a comma separated list of names that represent formal parameters to the function. The caller of the function communicates information into the function via these parameters. The parameters specified in the parameter list of a function definition are called *formal parameters*. A parameter is also known as an *argument*. The parameter list may be empty; an empty parameter list indicates that no information may be passed into the function by the caller.

- The *body* is a block of statements. The statements define the actions that the function is to perform. The statements may include variables other than the function's formal parameters; unless specified otherwise, variables used with the function are local to that function.

A client can pass multiple pieces of information into a function via multiple parameters. A function ordinarily passes back to the client one piece of information via a `return` statement, but a function may return multiple pieces of information packed up in a tuple or other data structure.

The following code defines a function that that computes the greatest common divisor (also called greatest common factor) of two integers. It determines largest factor (divisor) common to its parameters:

```python
def gcd(num1, num2):
    #  Determine the smaller of num1 and num2
    min = num1 if num1 < num2 else num2
    #  1 is definitely a common factor to all ints
    largestFactor = 1
    for i in range(1, min + 1):
        if num1 % i == 0 and num2 % i == 0:
            largestFactor = i   # Found larger factor
    return largestFactor
```

This function is named `gcd` and expects two integer arguments. Its formal parameters are named `num1` and `num2`. It returns an integer result. The function uses three local variables: `min`, `largestFactor`, and `i`. Local variables have meaning only within their scope. The scope of a local is point within the function's block after its assignment. This means that when you write a function you can name a local variable without fear that its name may be used already in another part of the program. Two different functions can use local variables named `x`, and these are two different variables that have no influence on each other. Anything local to a function definition is hidden to all code outside that function definition.

Since a formal parameter is a local variable, you can reuse the names of formal parameters in different functions without a problem.

Another advantage of local variables is that they occupy space in the computer's memory only when the function is executing. The run-time environment allocates space in the computer's memory for local variables and parameters when the function begins executing. When the function is finished and control returns to the client, the variables and parameters go out of scope, and the run-time environment ensures that the memory the local variables held is freed up for other purposes within the running program. This process of local variable allocation and deallocation happens each time a client calls the function.

Once a function has been defined, clients can use it. A programmer-defined function is invoked in exactly the same way as a standard library function like `sqrt` (6.2) or `randrange` (6.4). If the function returns a value, then its invocation can be used anywhere an expression of that type can be used. The function `gcd` can be called as part of an assignment statement:

```python
factor = gcd(val, 24)
```

This call uses the variable `val` as its first actual parameter and the literal value 24 as its second actual parameter. As with the standard Python functions, variables, expressions, and literals can be used as actual parameters. The function then computes and returns its result. This result is assigned to the variable `factor`.

How does the function call and parameter mechanism work? It's actually quite simple. The actual parameters, in order, are assigned (bound) to each of the formal parameters in the function definition, then control is passed to the body of the function. When the function's body is finished executing, control passes back to the point in the program where the function was called. The value returned by the function, if any, replaces the function call expression. The statement

```python
factor = gcd(val, 24)
```

assigns an integer value to `factor`. The expression on the right is a function call, so the function is invoked to determine what to assign. The value of the variable `val` is assigned to the formal parameter `num1`, and the literal value 24 is assigned to the formal parameter `num2`. The body of the `gcd` function then is executed. When the `return` statement in the body is encountered, program execution returns back to where the function was called. The argument of the return statement becomes the value that is assigned to `factor`.

Note that `gcd` could be called from many different places within the same program, and, since different parameter values could be passed at each of these different invocations, `gcd` could compute a different result at each invocation.

Other invocation examples include:

- **`print(gcd(36, 24))`**

    This example simply prints the result of the invocation. The value 36 is bound to `num1` and 24 is bound to `num2` for the purpose of the function call. The value 12 will be printed, since 12 is the greatest common divisor of 36 and 24.

- **`x = gcd(x − 2, 24)`**

    The execution of this statement would evaluate `x - 2` and bind its value to `num1`. `num2` would be assigned 24. The result of the call is then assigned to `x`. Since the right side of the assignment statement is evaluated *before* being assigned to the left side, the original value of `x` is used when calculating `x - 2`, and the function return value then updates `x`.

- **`x = gcd(x − 2, gcd(10, 8))`**

    This example shows two invocations in one statement. Since the function returns an integer value its result can itself be used as an actual parameter in a function call. Passing the result of one function call as an actual parameter to another function call is called *function composition*.

7.3 Main Function

Functions help us organize our code. It is common for Python programmers to use a function named `main` to hold the statements that to this point we have not placed within a function. Listing 7.7 (`gcdwithmain.py`) illustrates the typical Python code organization.

Listing 7.7: `gcdwithmain.py`

```
1  #  Computes the greatest common divisor of m and n
2  def gcd(m, n):
3      #  Determine the smaller of m and n
4      min = m if m < n else n
5      #  1 is definitely a common factor to all ints
6      largestFactor = 1
7      for i in range(1, min + 1):
8          if m % i == 0 and n % i == 0:
9              largestFactor = i   # Found larger factor
10     return largestFactor
11
```

```
12   #  Get an integer from the user
13   def get_int():
14       return int(input("Please enter an integer: "))
15
16   #  Main code to execute
17   def main():
18       n1 = get_int()
19       n2 = get_int()
20       print("gcd(", n1, ",",  n2, ") = ", gcd(n1, n2), sep="")
21
22   # Run the program
23   main()
```

The single free statement at the end:

```
main()
```

calls the `main` function which in turn directly calls several other functions (`get_int`, `print`, `gcd`, and `str`). The `get_int` function itself directly calls `int` and `input`. In the course of its execution the `gcd` function calls `range`. Figure 7.2 contains a diagram that shows the calling relationships among the function executions during a run of Listing 7.7 (`gcdwithmain.py`).

7.4 Parameter Passing

When a client calls a function that expects a parameter, the client must pass a parameter to the function. The process behind parameter passing in Python is simple: the function call binds to the formal parameter the object referenced by the actual parameter. The kinds of objects we have considered so far—integers, floating-point numbers, and strings—are classified as *immutable* objects. This means a programmer cannot change the value of the object. For example, the assignment

```
x = 4
```

binds the variable named x to the integer 4. We may change x by reassigning it, but we cannot change the integer 4. Four is always four. Similarly, we may assign a string literal to a variable, as in

```
word = 'great'
```

but we cannot change the string object to which `word` refers. If the client's actual parameter references an immutable object, the function's activity cannot affect the value of the actual parameter. Listing 7.8 (`parampassing.py`) illustrates the consequences of passing an immutable type to an function.

Listing 7.8: **parampassing.py**

```
1    def increment(x):
2        print("Beginning execution of increment, x =", x)
3        x += 1    #  Increment x
4        print("Ending execution of increment, x =", x)
5
6    def main():
7        x = 5
8        print("Before increment, x =", x)
9        increment(x)
10       print("After increment, x =", x)
```
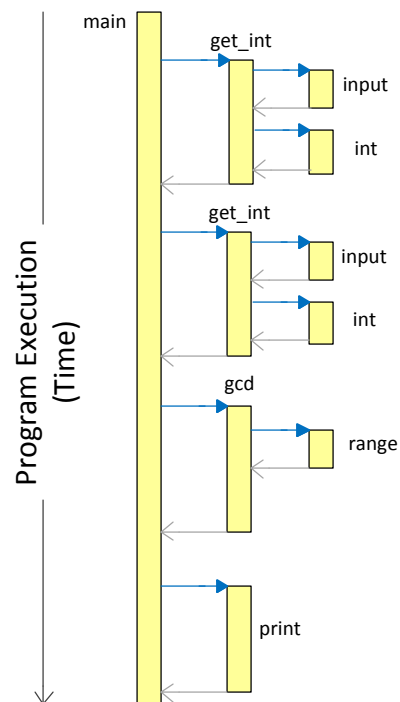
Figure 7.2: Calling relationships among functions during the execution of Listing 7.7 (`gcdwithmain.py`)

```
11
12  main()
```

For additional drama we chose to name the actual parameter the same as the formal parameter, but, of course, the names do not matter because they represent completely different contexts. Listing 7.8 (`parampassing.py`) produces

```
Before increment, x = 5
Beginning execution of increment, x = 5
Ending execution of increment, x = 6
After increment, x = 5
```

The variable x in main is unaffected by increment because x references an integer, and all integers are immutable.

7.5   Function Examples

This section contains a number of examples of code organization with functions.

7.5.1   Better Organized Prime Generator

Listing 7.9 (`primefunc.py`) is a simple enhancement of Listing 6.4 (`moreefficientprimes.py`). It uses the square root optimization and adds a separate `is_prime` function.

Listing 7.9: `primefunc.py`

```python
from math import sqrt

# is_prime(n)
#     Determines the primality of a given value
#     n an integer to test for primality
#     Returns true if n is prime; otherwise, returns false
def is_prime(n):
    result = True  #  Provisionally, n is prime
    root = sqrt(n)
    #  Try all potential factors from 2 to the square root of n
    trial_factor = 2
    while result and trial_factor <= root:
        result = (n % trial_factor != 0 )  #  Is it a factor?
        trial_factor += 1                  #  Try next candidate
    return result

# main
#     Tests for primality each integer from 2
#     up to a value provided by the user.
#     If an integer is prime, it prints it;
#     otherwise, the number is not printed.
def main():
    max_value = int(input("Display primes up to what value? "))
    for value in range(2, max_value + 1):
        if is_prime(value):          #  See if value is prime
            print(value, end=" ")    #  Display the prime number
    print()  #  Move cursor down to next line

main()    #  Run the program
```

Listing 7.9 (`primefunc.py`) illustrates several important points about well-organized programs:

- The complete work of the program is no longer limited to one block of code. The main function is responsible for generating prime candidates and printing the numbers that are prime. main delegates the task of testing for primality to the `is_prime` function. Both main and `is_prime` individually are simpler than the original monolithic code. Also, each function is more logically *coherent*. A function is coherent when it is focused on a single task. Coherence is a desirable property of functions. If a function becomes too complex by trying to do too many different things, it can be more difficult to write correctly and debug when problems are detected. A complex function usually can be decomposed into several, smaller, more coherent functions. The original function would then call these new simpler functions to accomplish its task. Here, main is not concerned about *how* to determine if a

given number is prime; `main` simply delegates the work to `is_prime` and makes use of the `is_prime` function's findings. For `is_prime` to do its job it does not need to know anything about the history of the number passed to it, nor does it need to know the client's intentions with the result it returns.

- Each function is preceded by a thorough comment that describes the nature of the function. It explains the meaning of each parameter, and it indicates what the function should return.

- While the exterior comment indicates *what* the function is to do, comments within each function explain in more detail *how* the function accomplishes its task.

A call to `is_prime` returns `True` or `False` depending on the value passed to it. The means a condition like

```python
if is_prime(value) == True:
```

can be expressed more compactly as

```python
if  is_prime(value):
```

because if `is_prime(value)` is `True`, `True == True` is `True`, and if `is_prime(value)` is `False`, `False == True` is `False`. The expression `is_prime(value)` suffices.

Just as it is better for a loop to have exactly one entry point and exactly one exit point, preferably a function will have a single `return` statement. Simple functions with a small number of `return`s are generally tolerable, however. Consider the following version of `is_prime`:

```python
def is_prime(n):
    trial_factor = 2
    root = sqrt(n)

    while trial_factor <= root:
        if n % trial_factor == 0:    #  Is trialFactor a factor?
            return False             #  Yes, return right away
        trial_factor += 1            #  Try next potential factor

    return True;                     #  Tried them all, must be prime
```

This version uses two `return` statements, but eliminates the need for a local variable (`result`). No `break` statement is necessary, because a `return` statement exits the function immediately. The two `return` statements are close enough textually in source code that the logic is easy to follow.

7.5.2 Command Interpreter

Some functions are useful even if they accept no information from the caller and return no result. Listing 7.10 (`calculator.py`) uses such a function.

Listing 7.10: `calculator.py`

```python
#     help_screen
#         Displays information about how the program works
#         Accepts no parameters
#         Returns nothing
def help_screen():
    print("Add:  Adds two numbers")
```

```
 7        print("Subtract:  Subtracts two numbers")
 8        print("Print:  Displays the result of the latest operation")
 9        print("Help:  Displays this help screen")
10        print("Quit:  Exits the program")
11
12  #  menu
13  #     Display a menu
14  #     Accepts no parameters
15  #     Returns the string entered by the user.
16  def menu():
17      #  Display a menu
18      return input("=== A)dd S)ubtract P)rint H)elp Q)uit ===")
19
20
21  #    main
22  #       Runs a command loop that allows users to
23  #       perform simple arithmetic.
24  def main():
25      result = 0.0
26      done = False;  #  Initially not done
27      while not done:
28          choice = menu()      # Get user's choice
29
30          if choice == "A" or choice == "a":   # Addition
31              arg1 = float(input("Enter arg 1: "))
32              arg2 = float(input("Enter arg 2: "))
33              result = arg1 + arg2
34              print(result)
35          elif choice == "S" or choice == "s": # Subtraction
36              arg1 = float(input("Enter arg 1: "))
37              arg2 = float(input("Enter arg 2: "))
38              result = arg1 - arg2
39              print(result)
40          elif choice == "P" or choice == "p": # Print
41              print(result)
42          elif choice == "H" or choice == "h": # Help
43              help_screen()
44          elif choice == "Q" or choice == "q": # Quit
45              done = True
46
47  main()
```

The `help_screen` function needs no information from `main`, nor does it return a result. It behaves exactly the same way each time it is called.

7.5.3 Restricted Input

Listing 5.23 (`betterinputonly.py`) forces the user to enter a value within a specified range. We now easily can adapt that concept to a function. Listing 7.11 (`betterinputfunc.py`) uses a function named `get_int_in_range` that does not return until the user supplies a proper value.

Listing 7.11: betterinputfunc.py

```
 1  # get_int_in_range(first, last)
```

```
2   #     Forces the user to enter an integer within a
3   #     specified range
4   #     first is either a minimum or maximum acceptable value
5   #     last is the corresponding other end of the range,
6   #     either a maximum or minimum value
7   #     Returns an acceptable value from the user
8   def get_int_in_range(first, last):
9       #  If the larger number is provided first,
10      #  switch the parameters
11      if  first > last:
12          first, last = last, first
13      #  Insist on values in the range first...last
14      in_value = int(input("Please enter values in the range " \
15                       + str(first) + "..." + str(last) + ": "))
16      while in_value < first or in_value > last:
17          print(in_value, "is not in the range", first, "...", last)
18          in_value = int(input("Please try again: "))
19      #  in_value at this point is guaranteed to be within range
20      return in_value;
21
22  #    main
23  #       Tests the get_int_in_range function
24  def main():
25      print(get_int_in_range(10, 20))
26      print(get_int_in_range(20, 10))
27      print(get_int_in_range(5, 5))
28      print(get_int_in_range(-100, 100))
29
30  main()    #  Run the program
```

Listing 7.11 (`betterinputfunc.py`) forces the user to enter a value within a specified range, as shown in this sample run:

```
Please enter values in the range 10...20: 4
4 is not in the range 10 ... 20
Please try again: 21
21 is not in the range 10 ... 20
Please try again: 16
16
Please enter values in the range 10...20: 10
10
Please enter values in the range 5...5: 4
4 is not in the range 5 ... 5
Please try again: 6
6 is not in the range 5 ... 5
Please try again: 5
5
Please enter values in the range -100...100: -101
-101 is not in the range -100 ... 100
Please try again: 101
101 is not in the range -100 ... 100
Please try again: 0
0
```

This functionality could be useful in many programs. In Listing 7.11 (`betterinputfunc.py`)

- The high and low values are specified by parameters. This makes the function more flexible since it could be used elsewhere in the program with a completely different range specified and still work correctly.

- The function is supposed to be called with the lower number passed as the first parameter and the higher number passed as the second parameter. The function also will accept the parameters out of order and automatically swap them to work as expected; thus,

  num = get_int_in_range(20, 50)

  will work exactly like

  num = get_int_in_range(50, 20)

- The Boolean variable `bad_entry` is used to avoid evaluating the Boolean expression twice (once to see if the bad entry message should be printed and again to see if the loop should continue).

7.5.4 Better Die Rolling Simulator

Listing 7.12 (`betterdie.py`) reorganizes Listing 6.11 (`die.py`) into functions.

Listing 7.12: `betterdie.py`

```
1  from random import randrange
2
3  #   show_die(spots)
4  #       Draws a picture of a die with number of spots
```

```
5   #       indicated spots is the number of spots on the top face
6   def show_die(spots):
7       print("+-------+")
8       if spots == 1:
9           print("|       |")
10          print("|   *   |")
11          print("|       |")
12      elif spots == 2:
13          print("| *     |")
14          print("|       |")
15          print("|     * |")
16      elif spots == 3:
17          print("|     * |")
18          print("|   *   |")
19          print("| *     |")
20      elif spots == 4:
21          print("| *   * |")
22          print("|       |")
23          print("| *   * |")
24      elif spots == 5:
25          print("| *   * |")
26          print("|   *   |")
27          print("| *   * |")
28      elif spots == 6:
29          print("| * * * |")
30          print("|       |")
31          print("| * * * |")
32      else:
33          print(" ***  Error: illegal die value ***")
34      print("+-------+")
35
36  #  roll
37  #    Returns a pseudorandom number in the range 1...6
38  def roll():
39      return randrange(1, 6)
40
41  #  main
42  #    Simulates the roll of a die three times
43  def main():
44      #  Roll the die three times
45      for i in range(0, 3):
46          show_die(roll())
47
48  main()   #  Run the program
```

In Listing 7.12 (`betterdie.py`), the `main` function is oblivious to the details of pseudorandom number generation. Also, `main` is not responsible to drawing the die. These important components of the program are now in functions, so their details can be perfected independently from `main`.

Note how the result of the call to `roll` is passed directly as an argument to `show_die`:

```
show_die(roll())
```

7.5.5 Tree Drawing Function

Listing 7.13 (`treefunc.py`) reorganizes Listing 5.19 (`startree.py`) into functions.

Listing 7.13: `treefunc.py`

```
1   #  tree(height)
2   #      Draws a tree of a given height
3   #      height is the height of the displayed tree
4   def tree(height):
5       row = 0                  #  First row, from the top, to draw
6       while row < height:  # Draw one row for every unit of height
7           #  Print leading spaces
8           count = 0
9           while count < height - row:
10              print(end=" ")
11              count += 1
12          #  Print out stars, twice the current row plus one:
13          #     1. number of stars on left side of tree
14          #        = current row value
15          #     2. exactly one star in the center of tree
16          #     3. number of stars on right side of tree
17          #        = current row value
18          count = 0
19          while count < 2*row + 1:
20              print(end="*")
21              count += 1
22          #  Move cursor down to next line
23          print()
24          #  Change to the next row
25          row += 1
26
27  #  main
28  #      Allows users to draw trees of various heights
29  def main():
30      height = int(input("Enter height of tree: "))
31      tree(height)
32
33  main()
```

Observe that the name `height` is being used as a local variable in `main` and as a formal parameter name in `tree`. There is no conflict here, and the two `height`s represent two distinct quantities. Furthermore, the fact that the statement

```
tree(height)
```

uses `main`'s `height` as an actual parameter and `height` happens to be the name as the formal parameter is simply a coincidence. The function call binds the value of `main`'s `height` variable to the formal parameter in `tree` also named `height`. The interpreter can keep track of which `height` is which based on where each is used.

7.5.6 Floating-point Equality

Recall from Listing 3.2 (`imprecise.py`) that floating-point numbers are not mathematical real numbers; a floating-point number is finite, and is represented internally as a quantity with a binary mantissa and exponent. Just as 1/3 cannot be represented finitely in the decimal (base 10) number system, 1/10 cannot be represented exactly in the binary (base 2) number system with a fixed number of digits. Often, no problems arise from this imprecision, and in fact many software applications have been written using floating-point numbers that must perform precise calculations, such as directing a spacecraft to a distant planet. In such cases even small errors can result in complete failures. Floating-point numbers can and are used safely and effectively, but not without appropriate care.

To build our confidence with floating-point numbers, consider Listing 7.14 (`simplefloataddition.py`), which adds two double-precision floating-point numbers and checks for a given value.

Listing 7.14: `simplefloataddition.py`

```
def main():
    x = 0.9
    x += 0.1
    if x == 1.0:
        print("OK")
    else:
        print("NOT OK")

main()
```

All seems well judging from the behavior of Listing 7.14 (`simplefloataddition.py`). Next, consider Listing 7.15 (`badfloatcheck.py`) which attempts to control a loop with a double-precision floating-point number.

Listing 7.15: `badfloatcheck.py`

```
def main():
    #  Count to ten by tenths
    i = 0.0
    while i != 1.0:
        print("i =", i)
        i += 0.1

main()
```

When executed, Listing 7.15 (`badfloatcheck.py`) begins as expected, but it does not end as expected:

```
i = 0
i = 0.1
i = 0.2
i = 0.3
i = 0.4
i = 0.5
i = 0.6
i = 0.7
i = 0.8
i = 0.9
i = 1
i = 1.1
i = 1.2
i = 1.3
i = 1.4
i = 1.5
i = 1.6
i = 1.7
i = 1.8
i = 1.9
i = 2
i = 2.1
```

We expect it stop when the loop variable `i` equals 1, but the program continues executing until the user types **Ctrl-C** or otherwise interrupts the program's execution. We are adding 0.1, just as in Listing 7.14 (`simplefloataddition.py`), but now there is a problem. Since 0.1 cannot be represented exactly within the constraints of the double-precision floating-point representation, the repeated addition of 0.1 leads to round off errors that accumulate over time. Whereas 0.1 + 0.9 rounded off may equal 1, 0.1 added to itself 10 times may be 1.000001 or 0.999999, neither of which is exactly 1.

Listing 7.15 (`badfloatcheck.py`) demonstrates that the `==` and `!=` operators are of questionable worth when comparing floating-point values. The better approach is to check to see if two floating-point values are *close enough*, which means they differ by only a very small amount. When comparing two floating-point numbers x and y, we essentially must determine if the absolute value of their difference is small; for example, $|x - y| < 0.00001$. We can construct an `equals` function and incorporate the `fabs` function introduced in 6.2. Listing 7.16 (`floatequals.py`) provides such an `equals` function.

Listing 7.16: `floatequals.py`

```python
from math import fabs

#   equals(a, b, tolerance)
#       Returns true if a = b or |a - b| < tolerance.
#       If a and b differ by only a small amount
#       (specified by tolerance), a and b are considered
#       "equal."  Useful to account for floating-point
#       round-off error.
#       The == operator is checked first since some special
#       floating-point values such as floating-point infinity
#       require an exact equality check.
def equals(a, b, tolerance):
```

```
13      return a == b or fabs(a - b) < tolerance;
14
15  #  Try out the equals function
16  def main():
17      i = 0.0
18      while not equals(i, 1.0, 0.0001):
19          print("i =", i)
20          i += 0.1
21
22  main()
```

The third parameter, named `tolerance`, specifies how close the first two parameters must be in order to be considered equal. The `==` operator must be used for some special floating-point values such as the floating-point representation for infinity, so the function checks for `==` equality as well. Since Python uses short-circuit evaluation for Boolean expressions involving logical *OR* (see 4.2), if the `==` operator indicates equality, the more elaborate check is not performed.

The output of Listing 7.16 (`floatequals.py`) is

```
i = 0.0
i = 0.1
i = 0.2
i = 0.30000000000000004
i = 0.4
i = 0.5
i = 0.6
i = 0.7
i = 0.7999999999999999
i = 0.8999999999999999
```

You should use a function like `equals` when comparing two floating-point values for equality.

7.6 Custom Functions vs. Standard Functions

Armed with our knowledge of function definitions, we can rewrite Listing 5.18 (`computesquareroot.py`) so the program uses a custom square root function. Listing 7.17 (`squarerootfunction.py`) shows one possibility.

Listing 7.17: `squarerootfunction.py`

```
1  #  Computes the approximate square root of val
2  #  val is an number
3  def square_root(val):
4      #  Compute a provisional square root
5      root = 1.0
6
7      #  How far off is our provisional root?
8      diff = root*root - val
9
```

```
10        #  Loop until the provisional root
11        #  is close enough to the actual root
12        while diff > 0.00000001 or diff < -0.00000001:
13            root = (root + val/root) / 2        # Compute new provisional root
14            #  How bad is our current approximation?
15            diff = root*root - val
16        return root
17
18  def main():
19        #  Get value from the user
20        num = float(input("Enter number: "))
21        #  Report square root
22        print("Square root of", num, "=", square_root(num))
23
24  main()
```

Is Listing 7.17 (`squarerootfunction.py`) better than Listing 6.1 (`standardsquareroot.py`) which uses the standard `sqrt` function from the `math` module? Generally speaking, if you have the choice of using a standard library function or writing your own custom function that provides the same functionality, choose to use the standard library routine. The advantages of using the standard library routine include:

- Your effort to produce the custom code is eliminated entirely; you can devote more effort to other parts of the application's development.

- If you write your own custom code, you must thoroughly test it to ensure its correctness; standard library code, while not immune to bugs, generally has been subjected to a complete test suite. Additionally, library code is used by many developers, and thus any lurking errors are usually exposed early; your code is exercised only by the programs you write, and errors may not become apparent immediately. If your programs are not used by a wide audience, bugs may lie dormant for a long time. Standard library routines are well known and trusted; custom code, due to its limited exposure, is suspect until it gains wider exposure and adoption.

- Standard routines typically are tuned to be very efficient; it takes a great deal of effort to make custom code efficient.

- Standard routines are well-documented; extra work is required to document custom code, and writing good documentation is hard work.

Listing 7.18 (`squarerootcomparison.py`) tests our custom square root function over a range of 10,000,000 floating point values.

Listing 7.18: **squarerootcomparison.py**

```
1   from math import fabs, sqrt
2
3   #  Consider two floating-point numbers equal when
4   #  the difference between them is very small.
5   #  equals(a, b, tolerance)
6   #     Returns true if a = b or |a - b| < tolerance.
7   #     If a and b differ by only a small amount
8   #     (specified by tolerance), a and b are considered
9   #     "equal."  Useful to account for floating-point
10  #     round-off error.
11  #     The == operator is checked first since some special
```

```
12   #      floating-point values such as floating-point infinity
13   #      require an exact equality check.
14   def equals(a, b, tolerance):
15       return a == b or fabs(a - b) < tolerance;
16
17
18   #  Computes the approximate square root of val
19   #  val is an number
20   def square_root(val):
21       #  Compute a provisional square root
22       root = 1.0;
23
24       #  How far off is our provisional root?
25       diff = root*root - val
26
27       #  Loop until the provisional root
28       #  is close enough to the actual root
29       while diff > 0.00000001 or diff < -0.00000001:
30           root = (root + val/root) / 2      # Compute new provisional root
31           #  How bad is our current approximation?
32           diff = root*root - val
33       return root
34
35   def main():
36       d = 0.0
37       while d < 100000.0:
38           if not equals(square_root(d), sqrt(d), 0.001):
39               print(d, ": Expected", sqrt(d), "but computed", square_root(d))
40           d += 0.0001  #  Consider next value
```

Listing 7.18 (squarerootcomparison.py) uses our equals method from Listing 7.16 (floatequals.py). Observe that the tolerance used within the square root computation is smaller than the tolerance main uses to check the result. The main function, therefore, uses a less strict notion of equality. The output of Listing 7.18 (squarerootcomparison.py) is

```
0.0 : Expected 0.0 but computed 6.103515625e-05
0.0006000000000000001 : Expected 0.024494897427831782 but computed 0.024495
```

shows that our custom square root function produces results outside of main's acceptable tolerance for two values. Two wrong answers out of ten million tests represents a 0.00002% error rate. While this error rate is very small, it indicates our square_root function is not perfect. Our function is not trustworthy because one of values that causes the function to fail may be very important to a particular application.

7.7 Summary

- The development of larger, more complex programs is more manageable when the program consists of multiple programmer-defined functions.

- Every function has one definition but can have many invocations.

- A function definition includes the function's name, parameters, and body.

- A function name, like a variable name, is an identifier.

- Formal parameters are the parameters as they appear in a function's definition; actual parameters are the arguments supplied by the client.

- Formal parameters essentially are variables local to the function; actual parameters passed by the client may be variables, expressions, or literal values.

- A function invocation binds the actual parameters to the formal parameters.

- Clients must pass to functions the number of parameters specified in the function definition. The types of the actual parameters must be compatible with the ways the formal parameters are used within the function definition.

- In the formal parameter is bound to an immutable type like a number or string, the function cannot affect the client's actual parameter.

- Variables defined within a function are local to that function definition. Local variables cannot be seen by code outside the function definition.

- During a program's execution, local variables live only when the function is executing. When a particular function call is finished, the space allocated for its local variables is freed up.

7.8 Exercises

1. Is the following a legal Python program?

```python
def proc(x):
    return x + 2

def proc(n):
    return 2*n + 1

def main():
    x = proc(5)

main()
```

2. Is the following a legal Python program?

```python
def proc(x):
    return x + 2

def main():
    x = proc(5)
    y = proc(4)

main()
```

3. Is the following a legal Python program?

```
def proc(x):
    print(x + 2)

def main():
    x = proc(5)

main()
```

4. Is the following a legal Python program?

```
def proc(x):
    print(x + 2)

def main():
    proc(5)

main()
```

5. Is the following a legal Python program?

```
def proc(x, y):
    return 2*x + y*y

def main():
    print(proc(5, 4))

main()
```

6. Is the following a legal Python program?

```
def proc(x, y):
    return 2*x + y*y

def main():
    print(proc(5))

main()
```

7. Is the following a legal Python program?

```
def proc(x):
    return 2*x

def main():
    print(proc(5, 4))

main()
```

8. Is the following a legal Python program?

```
def proc(x):
    print(2*x*x)
```

```
def main():
    proc(5)

main()
```

9. The programmer was expecting the following program to print 200. What does it print instead? Why does it print what it does?

```
def proc(x):
    x = 2*x*x

def main():
    num = 10
    proc(num)
    print(num)

main()
```

10. Is the following program legal since the variable x is used in two different places (proc and main)? Why or why not?

```
def proc(x):
    return 2*x*x

def main():
    x = 10
    print(proc(x))

main()
```

11. Is the following program legal since the actual parameter has a different name from the formal parameter (y vs. x)? Why or why not?

```
def proc(x):
    return 2*x*x

def main():
    y = 10
    print(proc(y))

main()
```

12. Complete the following distance function that computes the distance between two geometric points (x_1, y_1) and (x_2, y_2):

```
def distance(x1, y1, x2, y2):
    ...
```

Test it with several points to convince yourself that is correct.

13. What happens if a client passes too many parameters to a function?

14. What happens if a client passes too few parameters to a function?

15. What are the rules for naming a function in Python?

16. Consider the following function definitions:

```python
def fun1(n):
    result = 0
    while n:
        result += n
        n--
    return result

def fun2(stars):
    for i in range(stars + 1):
        print(end="*")
    print()

def fun3(x, y):
    return 2*x*x + 3*y

def fun4(n):
    return 10 <= n <= 20

def fun5(a, b, c):
    return a <= b if b <= c else false

def fun6():
    return randrange(0, 1)
```

Examine each of the following statements. If the statement is illegal, explain why it is illegal; otherwise, indicate what the statement will print.

(a) `print(fun1(5))`

(b) `print(fun1())`

(c) `print(fun1(5, 2))`

(d) `print(fun2(5))`

(e) `fun2(5)`

(f) `fun2(0)`

(g) `fun2(-2)`

(h) `print(fun3(5, 2))`

(i) `print(fun3(5.0, 2.0))`

(j) `print(fun3('A', 'B'))`

(k) `print(fun3(5.0))`

(l) `print(fun3(5.0, 0.5, 1.2))`

(m) `print(fun4(15))`

(n) `print(fun4(5))`

(o) `print(fun4(5000))`

(p) `print(fun5(2, 4, 6))`

(q) `print(fun5(4, 2, 6))`

(r) `print(fun5(2, 2, 6))`

(s) `print(fun5(2, 6))`

(t)
```
if fun5(2, 2, 6):
    print("Yes")
else:
    print("No")
```

(u) `print(fun6())`

(v) `print(fun6(4))`

(w) `print(fun3(fun1(3), 3))`

(x) `print(fun3(3, fun1(3)))`

(y) `print(fun1(fun1(fun1(3))))`

(z) `print(fun6(fun6()))`

Chapter 8

More on Functions

This chapter covers some additional aspects of functions in Python. Recursion, a key concept in computer science is introduced.

8.1 Global Variables

Variables defined within functions are local variables. Local variables have some very desirable properties:

- The memory required to store a local variable is used only when the variable is in scope. When the program execution leaves the scope of a local variable, the memory for that variable is freed up and can be used for a local variable in another function when that function is invoked.

- The same variable name can be used in different functions without any conflict. The interpreter derives all of its information about a local variable used within a function from the definition of that variable within that function. If the interpreter attempts to execute a statement that uses a variable that has not been defined, the interpreter issues a run-time error. When executing code in one function the interpreter will not look for a variable definition in another function. Thus, there is no way a local variable in one function can interfere with a local variable declared in another function.

A local variable is transitory, so its value is lost in between function invocations. Sometimes it is desirable to have a variable that lives as long as the program is running; that is, until the `main` function completes. In contrast to a local variable, a *global variable* is defined outside of all functions and is not local to any particular function. Any function can legally access and/or modify a global variable.

Any variable assigned within a function is local to that function, unless the variable is declared to be a global variable using the `global` reserved word. Listing 8.1 (`globalcalculator.py`) is a modification of Listing 7.10 (`calculator.py`) that uses a global variables named `result`, `arg1`, and `arg2` that are shared by several functions in the program.

Listing 8.1: `globalcalculator.py`

```
1  #    help_screen
2  #        Displays information about how the program works
3  #        Accepts no parameters
4  #        Returns nothing
```

```
 5  def help_screen():
 6      print("Add:  Adds two numbers")
 7      print("Subtract:  Subtracts two numbers")
 8      print("Print:  Displays the result of the latest operation")
 9      print("Help:  Displays this help screen")
10      print("Quit:  Exits the program")
11
12  #  menu
13  #     Display a menu
14  #     Accepts no parameters
15  #     Returns the string entered by the user.
16  def menu():
17      #  Display a menu
18      return input("=== A)dd S)ubtract P)rint H)elp Q)uit ===")
19
20  # Global variables used by several functions
21  result = 0.0
22  arg1 = 0.0
23  arg2 = 0.0
24
25  #  get_input
26  #     Assigns the globals arg1 and arg2 from user keyboard
27  #     input
28  def get_input():
29      global arg1, arg2  #  arg1 and arg2 are globals
30      arg1 = float(input("Enter argument #1: "))
31      arg2 = float(input("Enter argument #2: "))
32
33  #  report
34  #      Reports the value of the global result
35  def report():
36      #  Not assigning to result, global keyword not needed
37      print(result)
38
39  #  add
40  #     Assigns the sum of the globals arg1 and arg2
41  #     to the global variable result
42  def add():
43      global result   #  Assigning to result, global keyword needed
44      result = arg1 + arg2
45
46  #  subtract
47  #     Assigns the difference of the globals arg1 and arg2
48  #     to the global variable result
49  def subtract():
50      global result   #  Assigning to result, global keyword needed
51      result = arg1 - arg2
52
53
54  #    main
55  #       Runs a command loop that allows users to
56  #       perform simple arithmetic.
57  def main():
58      done = False;  #  Initially not done
59      while not done:
```

```
60            choice = menu()       # Get user's choice
61
62            if choice == "A" or choice == "a":    # Addition
63                get_input()
64                add()
65                report()
66            elif choice == "S" or choice == "s": # Subtraction
67                get_input()
68                subtract()
69                report()
70            elif choice == "P" or choice == "p": # Print
71                report()
72            elif choice == "H" or choice == "h": # Help
73                help_screen()
74            elif choice == "Q" or choice == "q": # Quit
75                done = True
76
77    main()
```

Listing 8.1 (globalcalculator.py) uses global variables result, arg1, and arg2. These names no longer appear in the main function. These global variables are accessed and/or modified in four different functions: get_input, report, add, and subtract. The global keyword within a function's block of code identifies the variables which are global variables. Notice that if a global variable is used within a function and not assigned a value, it does not need to be declared global.

When it is acceptable to use global variables, and when is it better to use local variables? In general, local variables are preferred to global variables for several reasons:

- When a function uses local variables exclusively and performs no other input operations (like calling the input function), its behavior is influenced only by the parameters passed to it. If a non-local variable appears, the function's behavior is affected by every other function that can modify that non-local variable. As a simple example, consider the following trivial function that appears in a program:

```
def increment(n):
    return n + 1
```

Can you predict what the following statement within that program will print?

```
print(increment(12))
```

If your guess is 13, you are correct. The increment function simply returns the result of adding one to its argument. The increment function behaves the same way each time it is called with the same argument.

Next, consider the following three functions that appear in some program:

```
def process(n):
    return n + m     #  m is a global integer variable

def assign_m():
    m = 5

def inc_m():
    m += 1
```

Can you predict what the following statement within the program will print?

```
print(process(12))
```

We cannot predict what this statement in isolation will print. The following scenarios all produce different results:

```
assign_m()
print(process(12))
```

prints 17,

```
m = 10
print(process(12))
```

prints 22,

```
m = 0
inc_m()
inc_m()
print(process(12))
```

prints 14, and

```
assign_m()
inc_m()
inc_m()
print(process(12))
```

prints 19. The identical printing statements print different values depending on the cumulative effects of the program's execution up to that point.

It may be difficult to locate an error if a function that uses a global variable fails because it may be the fault of *another* function that assigned an incorrect value to the global variable. The situation may be more complicated than the simple examples above; consider:

```
assign_m()
  .
  .     #  30 statements in between, some of which may change a,
  .     #  b, and m
  .
if a < 2 and b <= 10:
    m = a + b − 100
  .
  .     #  20 statements in between, some of which may change m
  .
print(process(12))
```

- A nontrivial program that uses non-local variables will be more difficult for a human reader to understand than one that does not. When examining the contents of a function, a non-local variable requires the reader to look elsewhere (outside the function) for its meaning:

```
#  Linear function
def f(x):
    return m*x + b
```

What are `m` and `b`? How, where, and when are they assigned or re-assigned?

- A function that uses only local variables can be tested for correctness in isolation from other functions, since other functions do not affect the behavior of this function. This function's behavior is only influenced only by its parameters, if it has any.

The exclusion of global variables from a function leads to *functional independence*. A function that depends on information outside of its scope to correctly perform its task is a dependent function. When a function operates on a global variable it depends on that global variable being in the correct state for the function to complete its task correctly. Nontrivial programs that contain many dependent functions are more difficult debug and extend. A truly independent function that use no global variables and uses no programmer-defined functions to help it out can be tested for correctness in isolation. Additionally, an independent function can be copied from one program, pasted into another program, and work without modification. Functional independence is a desirable quality.

The exclusion of global variables from a function's definition does not guarantee that the function always will produce the same results given the same parameter values; consider

```python
def compute(n):
    favorite = eval(input("Please enter your favorite number: "))
    return n + favorite
```

The `compute` function avoids global variables, yet we cannot predict the value of the expression `compute(12)`. Recall the `increment` function from above:

```python
def increment(n):
    return n + 1
```

Its behavior is totally predictable. Furthermore, `increment` does not modify any global variables, meaning it cannot in any way influence the overall program's behavior. We say that `increment` is a *pure function*. A pure function cannot perform any input or output (for example, use the `print` or `input` statements), nor may it use global variables. While `increment` is pure, the `compute` function is impure. The following function is impure also, since it performs output:

```python
def increment_and_report(n):
    print("Incrementing", n)
    return n + 1
```

A pure function simply computes its return value and has no other observable side effects.

A function that calls only other pure functions and otherwise would be considered pure is itself a pure function; for example:

```python
def double_increment(n):
    return increment(n) + 1
```

`double_increment` is a pure function since `increment` is pure; however, `double_increment_with_report`:

```python
def double_increment_with_report(n):
    return increment_and_report(n) + 1
```

is not a pure function since it calls `increment_and_report` which is impure.

8.2 Default Parameters

We have seen how clients may call some Python functions with differing numbers of parameters. Compare

```
a = input()
```

to

```
a = input("Enter your name: ")
```

We can define our own functions that accept a varying number of parameters by using a technique known as *default parameters*. Consider the following function that counts down:

```
def countdown(n=10):
    for count in range(n, -1, -1):  #  Count down from n to zero
        print(count)
```

The formal parameter expressed as n=10 represents a default parameter or default argument. If the client does not supply an actual parameter, the formal parameter n is assigned 10. The following call

```
countdown()
```

prints

```
10
9
8
7
6
5
4
3
2
1
0
```

but the invocation

```
countdown(5)
```

displays

```
5
4
3
2
1
0
```

As we can see, when the client does not supply a parameter specified by a function, and that parameter has a default value, the default value is used during the client's call.

Non-default and default parameters may be mixed in the parameter lists of function declarations, but all default parameters within the parameter list must appear after all the non-default parameters. This means the following definitions

```python
def sum_range(n, m=100):        #  OK, default follows non-default
    sum = 0
    for val in range(n, m + 1):
        sum += val
```

and

```python
def sum_range(n=0, m=100):      #  OK, both default
    sum = 0
    for val in range(n, m + 1):
        sum += val
```

are acceptable, but the definition

```python
def sum_range(n=0, m):     #  Illegal, non-default follows default
    sum = 0
    for val in range(n, m + 1):
        sum += val
```

is illegal, since a default parameter precedes a non-default parameter in the function's parameter list.

8.3 Recursion

The *factorial* function is widely used in combinatorial analysis (counting theory in mathematics), probability theory, and statistics. The factorial of n usually is expressed as $n!$. Factorial is defined for non-negative integers as

$$n! \ = \ n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdots 3 \cdot 2 \cdot 1$$

and 0! is defined to be 1. Thus $6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720$. Mathematicians precisely define factorial in this way:

$$n! \ = \ \begin{cases} 1 & \text{if } n \ = \ 0 \\ n \cdot (n-1)! & \text{otherwise.} \end{cases}$$

This definition is *recursive* since the ! function is being defined, but ! is used also in the definition. A Python function can be defined recursively as well. Listing 8.2 (`factorialtest.py`) includes a factorial function that exactly models the mathematical definition.

Listing 8.2: `factorialtest.py`

```python
# factorial(n)
#     Computes n!
#     Returns the factorial of n.
def factorial(n):
    if n == 0:
        return 1
    else:
```

```
 8            return n * factorial(n - 1)
 9
10  def main():
11      #  Try out the factorial function
12      print(" 0! = ",  factorial(0))
13      print(" 1! = ",  factorial(1))
14      print(" 6! = ",  factorial(6))
15      print("10! = ",  factorial(10))
16
17  main()
```

Listing 8.2 (`factorialtest.py`) produces

```
 0! =  1
 1! =  1
 6! =  720
10! =  3628800
```

Observe that the `factorial` function in Listing 8.2 (`factorialtest.py`) uses no loop to compute its result. The `factorial` function simply calls itself. The call `factorial(6)` is computed as follows:

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
             = 6 * 5 * 4 * 3 * 2 * 1 * 1
             = 6 * 5 * 4 * 3 * 2 * 1
             = 6 * 5 * 4 * 3 * 2
             = 6 * 5 * 4 * 6
             = 6 * 5 * 24
             = 6 * 120
             = 720
```

Note that the `factorial` function can be slightly optimized by changing the `if`'s condition from `n == 0` to `n < 2`. This change results in a function execution trace that eliminates two function calls at the end:

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * 1
             = 6 * 5 * 4 * 3 * 2
             = 6 * 5 * 4 * 6
             = 6 * 5 * 24
             = 6 * 120
             = 720
```

A correct simple recursive function definition is based on four key concepts:

1. The function must optionally call itself within its definition; this is the *recursive case*.

2. The function must optionally *not* call itself within its definition; this is the *base case*.

3. Some sort of conditional execution (such as an `if`/`else` statement) selects between the recursive case and the base case based on one or more parameters passed to the function.

4. Each invocation that does correspond to the base case must call itself with parameter(s) that move the execution closer to the base case. The function's recursive execution must converge to the base case.

Each recursive invocation must bring the function's execution closer to its base case. The `factorial` function calls itself in the `else` clause of the `if`/`else` statement. Its base case is executed if the condition of the `if` statement is true. Since the factorial is defined only for non-negative integers, the initial invocation of `factorial` must be passed a value of zero or greater. A zero parameter (the base case) results in no recursive call. Any other positive parameter results in a recursive call with a parameter that is closer to zero than the one before. The nature of the recursive process progresses towards the base case, upon which the recursion terminates.

Recursion is not our only option when computing a factorial. Listing 8.3 (`nonrecursfact.py`) provides a non-recursive factorial function.

Listing 8.3: `nonrecursfact.py`

```
#   factorial(n)
#       Computes n!
#       Returns the factorial of n.
def factorial(n):
    product = 1
    while n:
        product *= n
        n -= 1
    return product

def main():
    #   Try out the factorial function
    print(" 0! = ",  factorial(0))
    print(" 1! = ",  factorial(1))
    print(" 6! = ",  factorial(6))
    print("10! = ",  factorial(10))

main()
```

Which `factorial` function is better, the recursive or non-recursive version? Generally, if both the recursive and non-recursive functions implement the same basic algorithm, the non-recursive function will be more efficient. A function call is a relatively expensive operation compared to a variable assignment or comparison. The body of the non-recursive `factorial` function invokes no functions, but the recursive version calls a function—it calls itself—during all but the last recursive invocation. The iterative version of `factorial` is therefore more efficient than the recursive version.

Even though the iterative version of the factorial function is technically more efficient than the recursive version, on most systems you could not tell the difference. The reason is the factorial function "grows" fast, meaning it returns fairly large results for relatively small arguments.

Recall the `gcd` functions from 7.2. It computed he greatest common divisor (also known as greatest common factor) of two integer values. It works, but it is not very efficient. A better algorithm is used

in Listing 8.4 (`gcd.py`). It is based on one of the oldest algorithms known, developed by Euclid around 300 B.C.

Listing 8.4: `gcd.py`

```python
#   gcd(m, n)
#       Uses Euclid's method to compute
#       the greatest common divisor
#       (also called greatest common
#       factor) of  m and n.
#       Returns the GCD of m and n.
def gcd(m, n):
    if n == 0:
        return m
    else:
        return gcd(n, m % n)

def iterative_gcd(num1, num2):
    #  Determine the smaller of num1 and num2
    min = num1 if num1 < num2 else num2
    #  1 is definitely a common factor to all integers
    largestFactor = 1;
    for i in range(1, min + 1):
        if num1 % i == 0 and num2 % i == 0:
            largestFactor = i # Found larger factor
    return largestFactor

def main():
    #  Try out the gcd function
    for num1 in range(1, 101):
        for num2 in range(1, 101):
            print("gcd of", num1, "and", num2, "is", gcd(num1, num2))

main()
```

Note that this `gcd` function is recursive. The algorithm it uses is much different from our original iterative version. Because of the difference in the algorithms, this recursive version is actually much more efficient than our original iterative version. A recursive function, therefore, cannot be dismissed as inefficient just because it is recursive.

8.4 Making Functions Reusable

In a function definition we can package functionality that can be used in many different places within a program. Thus far, however, we have not seen how function definitions can be reused easily in other *programs*. For example, our `is_prime` function in Listing 7.9 (`primefunc.py`) works well within Listing 7.9 (`primefunc.py`), and it could be put to good use in other programs that need to test primality (encryption software, for example, makes heavy use of prime numbers). We could use the copy-and-paste feature of our favorite text editor to copy the `is_prime` function definition from Listing 7.9 (`primefunc.py`) into the new encryption program we are developing. It is possible to reuse a function in this way only if the function definition does not use any programmer-defined global variables nor any other programmer-defined functions. If a function does use any of these programmer-defined external entities, they must be included in the new code as well for the function to viable. Said another way, the code in the function definition ideally

will use only local variables and parameters. Such a function is a truly independent function can be reused easily in multiple programs.

The notion of copying source code from one program to another is not ideal, however. It is too easy for the copy to be incomplete or for some other error to be introduced during the copy. Furthermore, such code duplication is wasteful. If 100 programs on a particular system all need to use the is_prime function, under this scheme they must all include the is_prime code. This redundancy wastes space. Finally, in perhaps the most compelling demonstration of the weakness of this copy-and-paste approach, what if a bug is discovered in the is_prime function that all 100 programs are built around? When the error is discovered and fixed in one program, the other 99 programs will still contain the bug. Their source code must be updated, and it may be difficult to determine which files need to be fixed. The problem becomes much worse if the code has been released to the general public. It may be impossible to track down and correct all the copies of the faulty function. The situation would be the same if a correct is_prime function were updated to be made more efficient. The problem is this: all the programs using is_prime define their *own* is_prime function; while the function definitions are meant to be identical, there is no mechanism tying all these common definitions together. We really would like to reuse the function as is without copying it.

Fortunately, Python makes is easy for developers to package their functions into modules. These modules can be developed independently from the programs that use them, just as we have been using the functions in the standard math and random modules.

Consider the module Listing 8.5 (primecode.py).

Listing 8.5: `primecode.py`

```python
#  Contains the definition of the is_prime function
from math import sqrt

#  Returns True if non-negative integer n is prime;
#  otherwise, returns false
def is_prime(n):
    trial_factor = 2
    root = sqrt(n)

    while trial_factor <= root:
        if n % trialFactor == 0:    #  Is trialFactor a factor?
            return False;           #  Yes, return right away

    return True;                    #  Tried them all, must be prime
```

The code within the Listing 8.5 (primecode.py) file can be used by other Python programs. In the simplest case, this module appears in the same directory (folder) as the client code file that uses it. Listing 8.6 (usingprimecode.py) contains a sample client program that uses our packaged is_prime function.

Listing 8.6: `usingprimecode.py`

```python
from primecode import is_prime

def main():
    num = int(input("Enter an integer: "))
    if is_prime(num):
        print(num, "is prime")
    else:
        print(num, "is NOT prime")
```

The `is_prime` function now is more readily available to other programs.

If our Listing 8.5 (`primecode.py`) code is to be used widely by all users on the system, the module can be placed in a special Python library folder and be available to all users on the system.

8.5 Documenting Functions and Modules

It is good practice to document a function's definition with information that aids programmers who may need to use or extend the function. The essential information includes:

- **The purpose of the function**. The function's purpose is not always evident merely from its name. This is especially true for functions that perform complex tasks. A few sentences explaining what the function does can be helpful.

- **The role of each parameter**. The parameter names are obvious from the definition, but the type and purpose of a parameter may not be apparent merely from its name.

- **The nature of the return value**. While the function may do a number of interesting things as indicated in the function's purpose, what exactly does it return to the client? It is helpful to clarify exactly what value the function produces, if any.

We can use comments to document our functions, but Python provides a way that allows developers and tools to extract more easily the needed information. Python supports multi-line strings. Triple quotes (`'''` or `"""`). Consider Listing 8.7 (`multilinestring.py`) that uses a multi-line string.

Listing 8.7: `multilinestring.py`

```
x = '''
This is a multi-line
    string that goes on
for three lines!
'''
print(x)
```

Listing 8.7 (`multilinestring.py`) displays

```
This is a multi-line
    string that goes on
for three lines!
```

Observe that the multi-line string obeys indentation and line breaks—essentially reproducing the same formatting as in the source code.

When such a string is the first line in the block of a function definition or the first line in a module, the string is known as a *documentation string*, or *docstring* for short. Our `is_prime` function could be documented as shown in Listing 8.8 (`docprime.py`).

Listing 8.8: `docprime.py`

```python
1  '''
2  Contains the definition of the is_prime function
3  '''
4  from math import sqrt
5
6  def is_prime(n):
7      '''
8      Returns True if non-negative integer n is prime;
9      otherwise, returns false
10     '''
11     trial_factor = 2
12     root = sqrt(n)
13
14     while trial_factor <= root:
15         if n % trialFactor == 0:    #  Is trialFactor a factor?
16             return False;           #  Yes, return right away
17
18     return True;                    #  Tried them all, must be prime
```

With the `docprime` module loaded into the interactive shell we can type:

```
>>> help(is_prime)
Help on function is_prime in module docprime:

is_prime(n)
    Returns True if non-negative integer n is prime;
    otherwise, returns false
>>>
```

The normal comments serve as *internal documentation* for developers of the `is_prime` function, while the function docstring serves as *external documentation* for clients of the function.

Other information is often required in a commercial environment:

- **Author of the function**. Specify exactly who wrote the function. An email address can be included. If questions about the function arise, this contact information can be invaluable.

- **Date that the function's implementation was last modified**. An additional comment can be added each time the function is updated. Each update should specify the exact changes that were made and the person responsible for the update.

- **References**. If the code was adapted from another source, list the source. The reference may consist of a Web URL.

Some or all of this additional information may appear as internal documentation rather then appear in a docstring.

The following fragment shows the beginning of a well-commented function definition:

```
#        Author: Joe Algori (joe@eng-sys.net)
```

```
#        Last modified:  2010-01-06
#        Adapted from a formula published at
#        http://en.wikipedia.org/wiki/Distance
def distance(x1, y1, x2, y2):
    '''
      Computes the distance between two geometric points
        x1 is the x coordinate of the first point
        y1 is the y coordinate of the first point
        x2 is the x coordinate of the second point
        y2 is the y coordinate of the second point
      Returns the distance between (x1,y1) and (x2,y2)
    '''
    ...
```

From the information provided

- clients know what the function can do for them,

- clients know how to use the function,

- subsequent programmers that must maintain the function can contact the original author if questions arise about its use or implementation,

- subsequent programmers that must maintain the function can check the Wikipedia reference if questions arise about its implementation, and

- the quality of the algorithm may evaluated based upon the quality of its source of inspiration (Wikipedia).

8.6 Functions as Data

In Python, a function is special kind of object, just as integers, and strings are objects. Consider the following sequence in the interactive shell:

```
>>> type(2)
<class 'int'>
>>> type('Rick')
<class 'str'>
>>> from math import sqrt
>>> type(sqrt)
<class 'builtin_function_or_method'>
```

A function has the Python type `builtin_function_or_method`. Listing 8.9 (`arithmeticeval.py`) shows how we can treat a function as data and pass the function as a parameter to another function.

Listing 8.9: `arithmeticeval.py`

```
1  def add(x, y):
2      '''
3      Adds the parameters x and y and returns the result
```

```
 4         '''
 5         return x + y
 6
 7  def multiply(x, y):
 8         '''
 9         Multiplies the parameters x and y and returns the result
10         '''
11         return x * y
12
13  def evaluate(f, x, y):
14         '''
15         Calls the function f with parameters x and y:
16         f(x, y)
17         '''
18         return f(x, y)
19
20  def main():
21         '''
22         Tests the add, multiply, and evaluate functions
23         '''
24         print(add(2, 3))
25         print(multiply(2, 3))
26         print(evaluate(add, 2, 3))
27         print(evaluate(multiply, 2, 3))
28
29  main()   #  Call main
```

Listing 8.9 (`arithmeticeval.py`) prints

```
5
6
5
6
```

The first parameter of the `evaluate` function, `f`, represents a function. The expression

evaluate(add, 2, 3)

passes the `add` function and the literal values 2 and 3 to `evaluate`. The `evaluate` function then invokes the function specified in its first parameter, passing parameters two and three as arguments to that function.

Notice that the call

print(evaluate(add, '2', '3'))

prints

```
23
```

since the + operator applied to strings represents string concatenation instead of arithmetic addition.

We will see in Section 10.2 that the ability to pass function objects around enables us to develop flexible algorithms that can be adapted at run time.

8.7 Summary

- A global variable is defined outside of all functions and it available to all functions within its scope.

- A global variable exists for the life of the program, but local variables are created during a function call and are discarded when the function's execution has completed.

- Modifying a global variable can directly affect the behavior of any function that uses that global variable. A function that uses a global variable cannot be tested in isolation since its behavior can vary depending on how code outside the function modifies the global variable it uses.

- The behavior of an independent function is determined strictly by the parameters passed into it. An independent function will not use global variables.

- Local variables are preferred to global variables, since the indiscriminate use of global variables leads to functions that are less flexible, less reusable, and more difficult to understand.

- Programmers can define default values for functions parameters; these default parameters are substituted for parameters not supplied by clients.

- In functions that use default parameters, the default parameters must appear after all the non-default parameters in the function's parameter list.

- A recursive function must optionally call itself or not as determined by a conditional statement. The call of itself is the recursive case, and the base case does not make the recursive all. Each recursive call should move the computation closer to the base case.

- One or more functions in a file make up a module. Client programs can import these functions with an `import` statement.

- Multi-line strings are enclosed with triple quote marks (`'''` or `"""`). Such strings retain the same formatting as they appear in the source code.

- Document strings within functions and modules allow client programmers to obtain useful information about the functions and modules.

- Programmers should document each function indicating the function's purpose and the role(s) of its parameter(s) and return value. Additional information about the function's author, date of last modification, and other information may be required in some situations.

- A function can be passed as a parameter to another function. This ability enables the creation of more flexible algorithms.

8.8 Exercises

1. Consider the following Python code:

```
def sum1(n):
    s = 0
    while n > 0:
        s += 1
        n -= 1
    return s

val = 0

def sum2():
    s = 0
    while val > 0:
        s += 1
        val -= 1
    return s

def sum3():
    s = 0
    for i in range(val, 0, -1):
        s += 1
    return s

def main():
    #  See each question below for details

main()   #  Call main function
```

(a) What is printed if main is written as follows?

```
def main():
    global val
    val = 5
    print(sum1(input))
    print(sum2())
    print(sum3())
```

(b) What is printed if main is written as follows?

```
def main():
    global val
    val = 5
    print(sum1(input))
    print(sum3())
    print(sum2())
```

(c) What is printed if main is written as follows?

```
def main():
    global val
    val = 5
    print(sum2())
    print(sum1(input))
    print(sum3())
```

    (d) Which of the functions `sum1`, `sum2`, and `sum3` produce a side effect? What is the side effect?

    (e) Which function may not use the `val` variable?

    (f) What is the scope of the variable `val`? What is its lifetime?

    (g) What is the scope of the variable `i`? What is its lifetime?

    (h) Which of the functions `sum1`, `sum2`, and `sum3` demonstrate good functional independence? Why?

2. Consider the following Python code:

```python
def next_int1():
    cnt = 0
    cnt += 1
    return cnt


global_count = 0

def next_int2():
    global_count += 1
    return global_count

def main():
    for i = range(0, 5):
        print(next_int1(), next_int2())

main()
```

    (a) What does the program print?

    (b) Which of the functions `next_int1` and `next_int2` is the best function for the intended purpose? Why?

    (c) What is a better name for the function named `next_int1`?

    (d) The `next_int2` function works in this context, but why is it not a good implementation of function that always returns the next largest integer?

3. What does the following Python program print?

```python
def sum(m=0, n=0, r=0):
    return m + n + r

def main():
    print(max())
    print(max(4))
    print(max(4, 5))
    print(max(5, 4))
    print(max(1, 2, 3))
    print(max(2.6, 1.0, 3))

main()
```

4. Consider the following function:

```
def proc(n):
    if n < 1:
        return 1
    else:
        return proc(n/2) + proc(n - 1)
```

Evaluate each of the following expressions:

(a) `proc(0)`

(b) `proc(1)`

(c) `proc(2)`

(d) `proc(3)`

(e) `proc(5)`

(f) `proc(10)`

5. Rewrite the `gcd` function so that it implements Euclid's method but uses iteration instead of recursion.

6. Classify the following functions as pure or impure. `x` is a global variable.

(a)
```
def f1(m, n):
    return 2*m + 3*n
```

(b)
```
def f2(n)
    return n - 2
```

(c)
```
def f3(n):
    return n - x
```

(d)
```
def f4(n):
    print(2*n)
```

(e)
```
def f5(n):
    m = eval(input())
    return m * n
```

(f)
```
def f6(n):
    m = 2*n
    p = 2*m - 5
    return p - n
```

Chapter 9

Lists

The variables we have used to this point can assume only one value at a time. As we have seen, individual variables can be used to create some interesting and useful programs; however, variables that can represent only one value at a time do have their limitations. Consider Listing 9.1 (`averagenumbers.py`) which averages five numbers entered by the user.

Listing 9.1: averagenumbers.py

```
def main():
    print("Please enter five numbers: ")
    #  Allow the user to enter in the five values.
    n1 = eval(input("Please enter number 1: "))
    n2 = eval(input("Please enter number 2: "))
    n3 = eval(input("Please enter number 3: "))
    n4 = eval(input("Please enter number 4: "))
    n5 = eval(input("Please enter number 5: "))
    print("Numbers entered:", n1, n2, n3, n4, n5)
    print("Average:", (n1 + n2 + n3 + n4 + n5)/5)

main()
```

A sample run of Listing 9.1 (`averagenumbers.py`) looks like:

```
Please enter five numbers:
Please enter number 1: 34.2
Please enter number 2: 10.4
Please enter number 3: 18.0
Please enter number 4: 29.3
Please enter number 5: 15.1
Numbers entered: 34.2 10.4 18.0 29.3 15.1
Average: 21.4
```

The program conveniently displays the values the user entered and then computes and displays their average.

Suppose the number of values to average must increase from five to 25. If we use Listing 9.1 (`averagenumbers.py`) as a guide, twenty additional variables must be introduced, and the overall length of the program necessarily will grow. Averaging 1,000 numbers using this approach is impractical.

Listing 9.2 (`averagenumbers2.py`) provides an alternative approach for averaging numbers that uses a loop.

Listing 9.2: `averagenumbers2.py`

```
def main():
    sum = 0.0
    NUMBER_OF_ENTRIES = 5
    print("Please enter", NUMBER_OF_ENTRIES, " numbers: ")
    for i in range(0, NUMBER_OF_ENTRIES):
        num = eval(input("Enter number " + str(i) + ": ")
        sum += num;
    print("Average:", sum/NUMBER_OF_ENTRIES)

main()
```

Listing 9.2 (`averagenumbers2.py`) behaves slightly differently from Listing 9.1 (`averagenumbers.py`), as the following sample run using the same data shows:

```
Please enter 5  numbers:
Enter number 0: 34.2
Enter number 1: 10.4
Enter number 2: 18.0
Enter number 3: 29.3
Enter number 4: 15.1
Average: 21.4
```

Listing 9.2 (`averagenumbers2.py`) can be modified to average 25 values much more easily than Listing 9.1 (`averagenumbers.py`) that must use 25 separate variables—just change the value of NUMBER_OF_ENTRIES. In fact, the coding change to average 1,000 numbers is no more difficult. However, unlike the original average program, this new version does not display the numbers entered. This is a significant difference; it may be necessary to retain all the values entered for various reasons:

- All the values can be redisplayed after entry so the user can visually verify their correctness.

- The values may need to be displayed in some creative way; for example, they may be placed in a graphical user interface component, like a visual grid (spreadsheet).

- The values entered may need to be processed in a different way after they are all entered; for example, we may wish to display just the values entered above a certain value (like greater than zero), but the limit is not determined until after all the numbers are entered.

In all of these situations we must retain the values of all the variables for future recall.

We need to combine the advantages of both of the above programs; specifically we want

- the ability to retain individual values, and

- the ability to dispense with creating individual variables to store all the individual values

These may seem like contradictory requirements, but Python provides a standard data structure that simultaneously provides both of these advantages—the list.

9.1 Using Lists

A list refers to a collection of objects; it represents an ordered sequence of data. In that sense, a list is similar to a string, except a string can hold only characters. We may access the elements contained in a list via their position within the list. A list need not be homogeneous; that is, the elements of a list do not all have to be of the same type.

Like any other variable, a list variable can be local or global, and it must be defined (assigned) before it is used. The following code fragment declares a list named `lst` that holds the integer values $2, -3, 0, 4, -1$:

```
lst = [2, -3, 0, 4, -1]
```

The right-hand side of the assignment statement is a literal list. The elements of the list appear within square brackets (`[ ]`), the elements are separated by commas. The following statement:

```
a = []
```

assigns the empty list to `a`. We can print list literals and lists referenced through variables:

```
lst = [2, -3, 0, 4, -1]    # Assign the list
print([2, -3, 0, 4, -1])   # Print a literal list
print(lst)                 # Print a list variable
```

The above code prints

```
[2, -3, 0, 4, -1]
[2, -3, 0, 4, -1]
```

We can access individual elements of a list using square brackets:

```
lst = [2, -3, 0, 4, -1]    #  Assign the list
lst[0] = 5                 #  Make the first element 5
print(lst[1])              #  Print the second element
lst[4] = 12                #  Make the last element 12
print(lst)                 #  Print a list variable
print([10, 20, 30][1])     #  Print second element of literal list
```

This code prints

```
-3
[5, -3, 0, 4, 12]
20
```

The number within the square brackets indicates the distance from the beginning of the list. The expression list[0] therefore indicates the element at the very beginning (a distance of zero from the beginning), and list[1] is the second element (a distance of one away from the beginning).

If a is a list with n elements, and i is an integer such that $0 \leq i < n$, then a[n] is an element in the list.

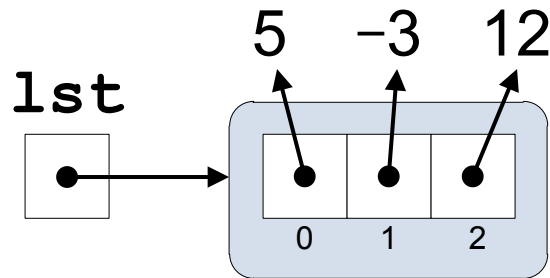Figure 9.1 visualizes the list assigned as

```
lst = [5, -3, 12]
```


Figure 9.1: A simple list with three elements. The small number below a list element represents the index of that element.

Listing 9.3 (heterolist.py) demonstrates that lists may be heterogeneous; that is, a list can hold elements of varying types.

Listing 9.3: heterolist.py

```
1  collection = [24.2, 4, 'word', eval, 19, -0.03, 'end']
2  print(collection[0])
3  print(collection[1])
4  print(collection[2])
5  print(collection[3])
6  print(collection[4])
7  print(collection[5])
8  print(collection[6])
9  print(collection)
```

Listing 9.3 (heterolist.py) prints

```
24.2
4
word
<built-in function eval>
19
-0.03
end
[24.2, 4, 'word', <built-in function eval>, 19, -0.03, 'end']
```

We clearly see that a single list can hold integers, floating-point numbers, strings, and even functions. A list can hold other lists; the following code

```
col = [23, [9.3, 11.2, 99.0], [23], [], 4, [0, 0]]
print(col)
```

prints

```
[23, [9.3, 11.2, 99.0], [23], [], 4, [0, 0]]
```

Four of the elements of the list `col` are themselves lists.

In an expression such as

```
a[3]
```

the expression within the square brackets is called an *index* or *subscript*. The subscript terminology is borrowed from mathematicians who use subscripts to reference elements in a mathematical vector (for example, V_2 represents the second element in vector *V*). Unlike the convention often used in mathematics, however, the first element in a list is at position *zero*, not one. The expression `a[2]` can be read aloud as "ay sub two." As mentioned above, the index indicates the distance from the beginning; thus, the very first element is at a distance of zero from the beginning of the list. The first element of list `a` is `a[0]`. As a consequence of a zero beginning index, if list `a` holds *n* elements, the last element in `a` is `a[n − 1]`, not `a[n]`.

The elements of a list extracted with `[]` can be treated as any other variable; for example,

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
#  Print the fourth element
print(nums[3])
#  The third element is the average of two other elements
nums[2] = (nums[0] + nums[9])/2;
#  Assign elements at indices 1 and 4 from user input
#  using tuple assignment
nums[1], nums[4] = eval(input("Enter a, b: "))
```

The expression within `[]` must evaluate to an integer; some examples include

- an integer literal: `a[34]`

- an integer variable: `a[x]` (x must be an integer)

- an integer arithmetic expression: `a[x + 3]` (x must be an integer)

- an integer result of a function call that returns an integer: `a[max(x, y)]` (max must return an integer)

- an element of another list: `a[b[3]]` (element `b[3]` must be an integer)

The action of moving through a list visiting each element is known as *traversal*. The `for` loop is made to iterate over aggregate types like lists. Listing 9.4 (`heterolistfor.py`) uses a `for` loop and behaves identically to Listing 9.3 (`heterolist.py`).

Listing 9.4: `heterolistfor.py`

```
1  collection = [24.2, 4, 'word', eval, 19, -0.03, 'end']
2  for item in collection:
3      print(item)        #  Print each element
```

The built-in function `len` returns the number of elements in a list: The code segment

```
print(len([2, 4, 6, 8]))
a = [10, 20, 30]
print(len(a))
```

prints

```
4
3
```

The name `len` stands for *length*. We can print the elements of a list in reverse order as follows:

```
nums = [2, 4, 6, 8]
#  Print last element to first (zero index) element
for i in range(len(nums) - 1, -1, -1):
    print(nums[i])
```

This fragment prints

```
8
6
4
2
```

The plus (+) operator concatenates lists in the same way it concatenates strings. The following shows some experiments in the interactive shell with list concatenation:

```
>>> a = [2, 4, 6, 8]
>>> a
[2, 4, 6, 8]
>>> a + [1, 3, 5]
[2, 4, 6, 8, 1, 3, 5]
>>> a
[2, 4, 6, 8]
>>> a = a + [1, 3, 5]
>>> a
[2, 4, 6, 8, 1, 3, 5]
>>> a += [10]
>>> a
[2, 4, 6, 8, 1, 3, 5, 10]
>>> a += 20
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    a += 20
TypeError: 'int' object is not iterable
```

The statement

a = [2, 4, 6, 8]

assigns the given list literal to the variable a. The expression

a + [1, 3, 5]

evaluates to the list [2, 4, 6, 8, 1, 3, 5], but the statement does not change the list to which a refers.
The statement

a = a + [1, 3, 5]

actually reassigns a to the new list [2, 4, 6, 8, 1, 3, 5]. The statement

a += [10]

updates a to be the new list [2, 4, 6, 8, 1, 3, 5, 10]. Observe that the + will concatenate two lists,
but it cannot join a list and a non-list. The following statement

a += 20

is illegal since a refers to a list, and 20 is an integer, not a list. If used within a program under these
conditions, this statement will produce a run-time exception.

Listing 9.5 (builduserlist.py) shows how to build lists as the program executes.

Listing 9.5: builduserlist.py

```
1   #  Build a custom list of non-negative integers specified by the user
2
3   def make_list():
4       '''
5       Builds a list from input provided by the user.
```

```
6          '''
7          result = []      #  List to return is initially empty
8          in_val = 0       #  Ensure loop is entered at least once
9          while in_val >= 0:
10             in_val = int(input("Enter integer (-1 quits): "))
11             if in_val >= 0:
12                 result = result + [in_val]   #  Add item to list
13         return result
14
15     def main():
16         col = make_list()
17         print(col)
18
19     main()
```

A sample run of Listing 9.5 (`builduserlist.py`) produces

```
Enter integer (-1 quits): 23
Enter integer (-1 quits): 100
Enter integer (-1 quits): 44
Enter integer (-1 quits): 19
Enter integer (-1 quits): 19
Enter integer (-1 quits): 101
Enter integer (-1 quits): 98
Enter integer (-1 quits): -1
[23, 100, 44, 19, 19, 101, 98]
```

If the user enters a negative number initially, we get:

```
Enter integer (-1 quits): -1
[]
```

There are several ways to build a list without explicitly listing every element in the list. We can use the `range` function to produce a regular sequence of integers. The range object returned by `range` is not itself a list, but we can make a list from a range using the `list` function, as Listing 9.6 (`makeintegerlists.py`) demonstrates.

Listing 9.6: `makeintegerlists.py`

```
1    def main():
2        a = list(range(0, 10))
3        print(a)
4        a = list(range(10, -1, -1))
5        print(a)
6        a = list(range(0, 100, 10))
7        print(a)
8        a = list(range(-5, 6))
9        print(a)
```

```
10
11   main()
```

Listing 9.6 (`makeintegerlists.py`) prints

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
```

It is easy to make a list in which all the elements are the same or a pattern of elements repeat. The *
operator, with applied to a list and an integer, "multiplies" the elements of a list. The code

```
for i in range(0, n):
    a += a
```

which effectively concatenates list `a` with itself `n` times, may be expressed more simply as

```
a * n
```

Listing 9.7 (`makeuniformlists.py`) builds several lists using the * list multiplication operator.

Listing 9.7: `makeuniformlists.py`

```
1   def main():
2       a = [0] * 10
3       print(a)
4
5       a = [3.4] * 5
6       print(a)
7
8       a = 3 * ['ABC']
9       print(a)
10
11       a = 4 * [10, 20, 30]
12       print(a)
13
14   main()
```

The output of Listing 9.7 (`makeuniformlists.py`) is

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[3.4, 3.4, 3.4, 3.4, 3.4]
['ABC', 'ABC', 'ABC']
[10, 20, 30, 10, 20, 30, 10, 20, 30, 10, 20, 30]
```

Observe that the integer multiplier may appear either to left or the right of the * operator, and the effects
are the same. This means the list multiplication * operator is *commutative*.

We now have all the tools we need to build a program that flexibly averages numbers while retaining all the values the user enters. Listing 9.8 (`listaverage.py`) uses an list and a loop to achieve the generality of Listing 9.2 (`averagenumbers2.py`) with the ability to retain all input for later redisplay.

Listing 9.8: `listaverage.py`

```python
def main():
    #  Set up variables
    sum = 0.0
    NUMBER_OF_ENTRIES = 5
    numbers = []

    #  Get input from user
    print("Please enter", NUMBER_OF_ENTRIES, "numbers: ")
    for i in range(0, NUMBER_OF_ENTRIES):
        num = eval(input("Enter number " + str(i) + ": "))
        numbers += [num]
        sum += num;

    #  Print the numbers entered
    print(end="Numbers entered: ")
    for num in numbers:
        print(num, end=" ")
    print()    #  Print newline

    #  Print average
    print("Average:", sum/NUMBER_OF_ENTRIES)

main()  #  Execute main
```

The output of Listing 9.8 (`listaverage.py`) is similar to the original Listing 9.1 (`averagenumbers.py`) program:

```
Please enter 5 numbers:
Enter number 0: 9.0
Enter number 1: 3.5
Enter number 2: 0.2
Enter number 3: 100.0
Enter number 4: 15.3
Numbers entered: 9.0 3.5 0.2 100.0 15.3
Average: 25.6
```

Unlike the original program, however, we now conveniently can extend this program to handle as many values as we wish. We need to change only the definition of the NUMBER_OF_ENTRIES variable to allow the program to handle any number of values. This centralization of the definition of the list's size eliminates duplicating a literal numeric value and leads to a program that is more maintainable. Suppose every occurrence of NUMBER_OF_ENTRIES were replaced with the literal value 5. The program would work exactly the same way, but changing the size would require touching many places within the program. When duplicate information is scattered throughout a program, it is a common mistake to update some but not all of the information when a change is to be made. If all of the duplicate information is not updated to agree, the inconsistencies result in logic errors within the program. By faithfully using the NUMBER_OF_ENTRIES

variable throughout the program instead of the literal numeric value, we can avoid the problems with this potential inconsistency.

The first loop in Listing 9.8 (`listaverage.py`) collects all five input values from the user. The second loop prints all the numbers the user entered.

9.2   List Assignment and Equivalence

Given the assignment

```
lst = [2, 4, 6, 8]
```

the expression `lst` is very different from the expression `lst[2]`. The expression `lst` is a reference to the list, while `lst[2]` is a reference to a particular element in the list, in this case the integer 6. The integer 6 is immutable (see Section 7.4); a literal integer cannot change to be another value. Six is always six. A variable, of course, can change its value and its type through assignment. Variable assignment changes the object to which the variable is bound. Recall Figure 2.2, and consider the Listing 9.9 (`listassignment.py`).

Listing 9.9: `listassignment.py`

```
1  a = [10, 20, 30, 40]
2  b = [10, 20, 30, 40]
3  print('a =', a)
4  print('b =', b)
5  b[2] = 35
6  print('a =', a)
7  print('b =', b)
```

Figure 9.2 shows the consequences of each of the assignment statements in Listing 9.9 (`listassignment.py`),
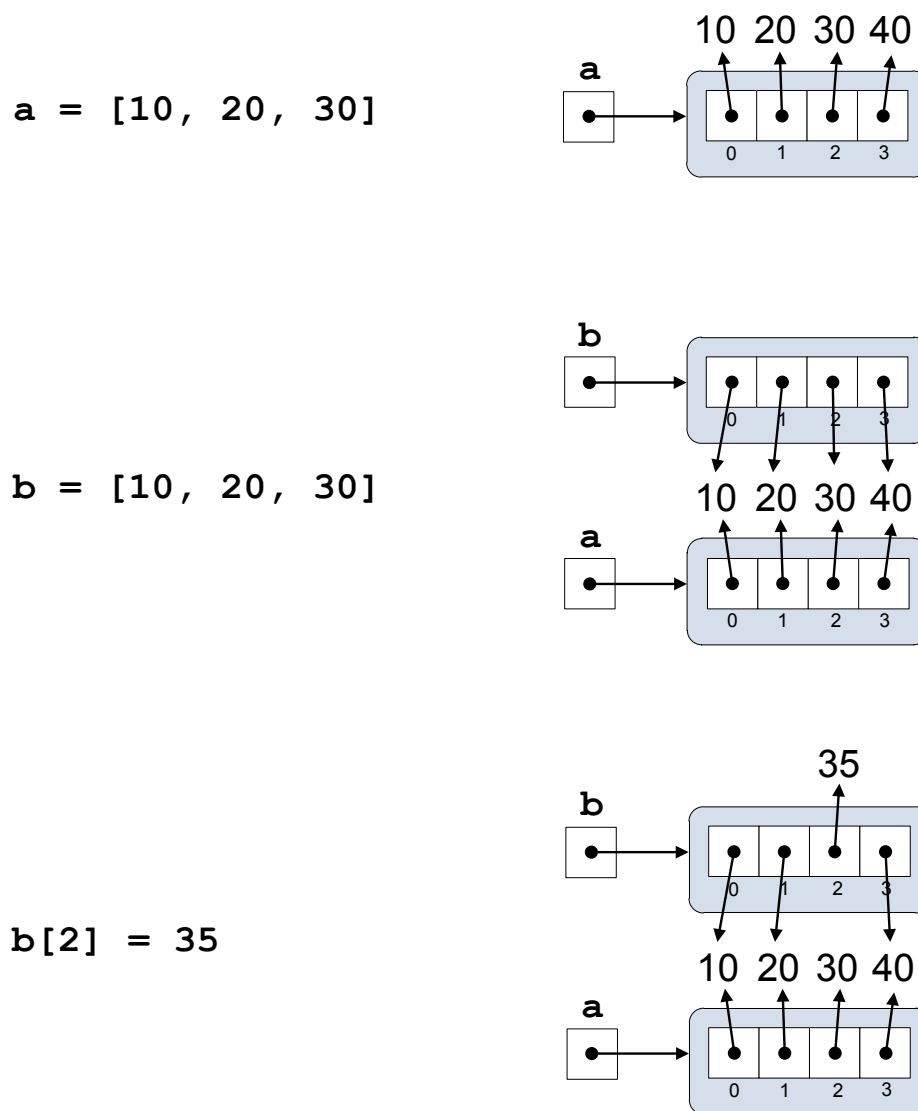
As Figure 9.2 illustrates, variables `a` and `b` refer to two different list objects; however, the elements of both lists bind to the same (immutable) values. Reassigning an element of list `b` does not affect list `a`. The output of Listing 9.9 (`listassignment.py`) verifies this analysis:

```
a = [10, 20, 30, 40]
b = [10, 20, 30, 40]
a = [10, 20, 30, 40]
b = [10, 20, 35, 40]
```

Now consider Listing 9.10 (`listalias.py`), a subtle variation of Listing 9.9 (`listassignment.py`). At first glance, the code in Listing 9.10 (`listalias.py`) looks like it may behave exactly like Listing 9.9 (`listassignment.py`).

Listing 9.10: `listalias.py`

```
1  a = [10, 20, 30, 40]
2  b = a
3  print('a =', a)
4  print('b =', b)
5  b[2] = 35
```

a = [10, 20, 30]

b = [10, 20, 30]

b[2] = 35

Figure 9.2: State of Listing 9.9 (`listassignment.py`) as the assignment statements execute

```
6  print('a =', a)
7  print('b =', b)
```

As Figure 9.3 illustrates, the second assignment statement causes variables a and b to refer to the *same* list object. We say that a and b are *aliases*. Reassigning b[2] changes a[2] as well, as Listing 9.10 (`listalias.py`)'s output shows:

a = [10, 20, 30]

b = a

b[2] = 35
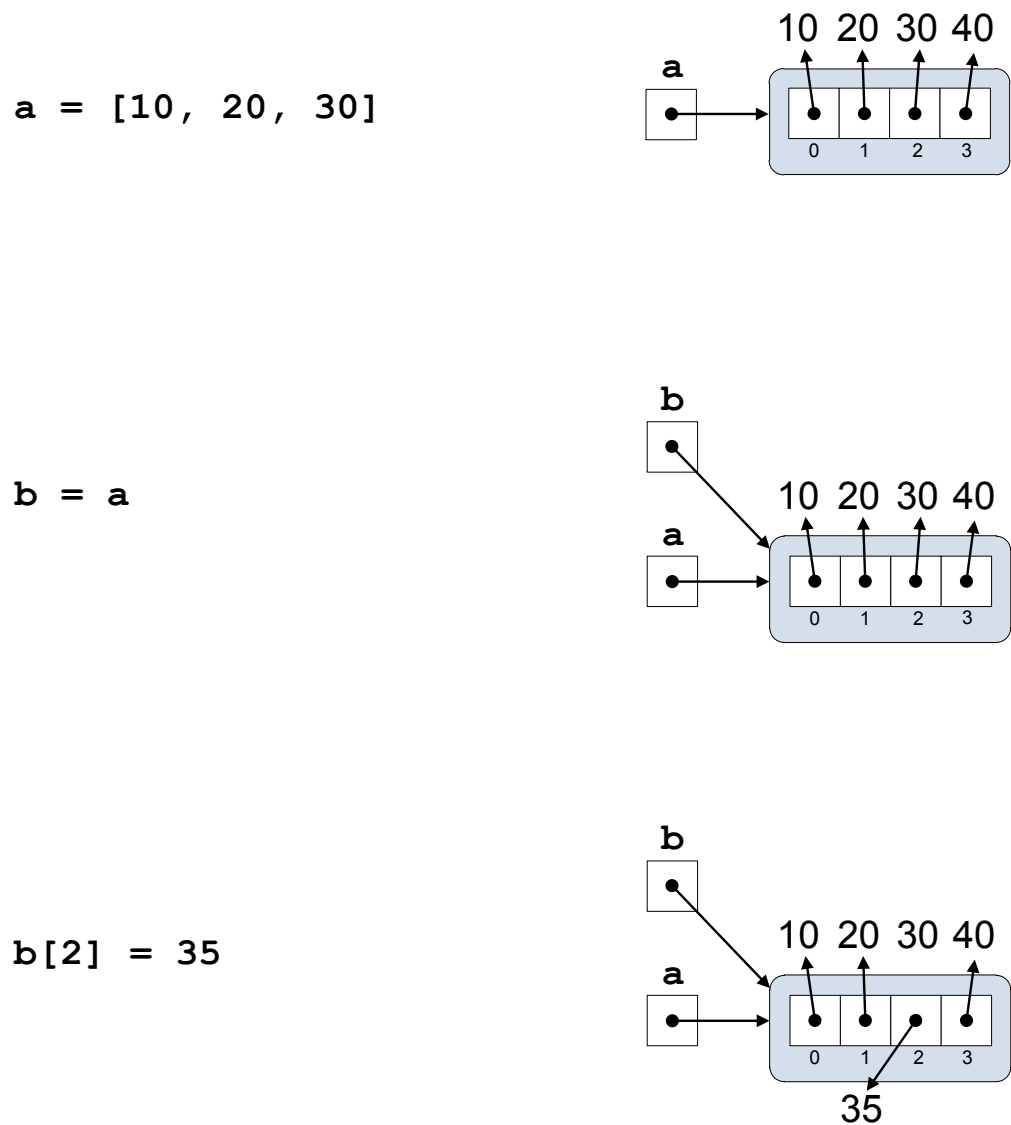
Figure 9.3: State of Listing 9.10 (listalias.py) as the assignment statements execute

```
a = [10, 20, 30, 40]
b = [10, 20, 30, 40]
a = [10, 20, 35, 40]
b = [10, 20, 35, 40]
```

If `a` refers to a list, the statement

b = a

does not make a copy of `a`'s list. Instead it makes `a` and `b` aliases to the same list. Lists are *mutable* data structures. Individual elements accessed through `[]` may be reassigned. If more than one variable is bound to the same list, any element modification through one of the variables will affect the list from the point of view of all the aliased variables.

The familiar `==` equality operator determines if two lists contain the same elements. The `is` operator determines if two variables alias the same list. Listing 9.11 (`listequivalence.py`) demonstrates the difference between the two operators.

Listing 9.11: `listequivalence.py`

```
1   #  a and b are distinct lists that contain the same elements
2   a = [10, 20, 30, 40]
3   b = [10, 20, 30, 40]
4   print('Is ', a, ' equal to ', b, '?', sep='', end=' ')
5   print(a == b)
6
7   print('Are ', a, ' and ', b, ' aliases?', sep='', end=' ')
8   print(a is b)
9
10  #  c and d alias are distinct lists that contain the same elements
11  c = [100, 200, 300, 400]
12  d = c       #  Makes d an alias of c
13  print('Is ', c, ' equal to ', d, '?', sep='', end=' ')
14  print(c == d)
15
16  print('Are ', c, ' and ', d, ' aliases?', sep='', end=' ')
17  print(c is d)
```

Listing 9.11 (`listequivalence.py`) prints

```
Is [10, 20, 30, 40] equal to [10, 20, 30, 40]? True
Are [10, 20, 30, 40] and [10, 20, 30, 40] aliases? False
Is [100, 200, 300, 400] equal to [100, 200, 300, 400]? True
Are [100, 200, 300, 400] and [100, 200, 300, 400] aliases? True
```

When comparing lists `lst1` and `lst2`, if the expression `lst1 is lst2` evaluates to `True`, the expression `lst1 == lst2` is guaranteed to be `True`.

What if we wish to make a copy of an existing list? Listing 9.12 (`listcopy.py`) shows one way to accomplish this.

Listing 9.12: `listcopy.py`

```
1   def list_copy(lst):
2       result = []
3       for item in lst:
4           result += [item]
5       return result
```

```
6
7   def main():
8       #  a and b are distinct lists that contain the same elements
9       a = [10, 20, 30, 40]
10      b = list_copy(a)       #  Make a copy of a
11      print('a =', a, '   b =', b)
12
13      print('Is ', a, ' equal to ', b, '?', sep='', end=' ')
14      print(a == b)
15
16      print('Are ', a, ' and ', b, ' aliases?', sep='', end=' ')
17      print(a is b)
18
19      b[2] = 35       #  Change an element of b
20      print('a =', a, '   b =', b)
21
22  main()
```

The output of Listing 9.12 (listcopy.py) reveals:

```
a = [10, 20, 30, 40]    b = [10, 20, 30, 40]
Is [10, 20, 30, 40] equal to [10, 20, 30, 40]? True
Are [10, 20, 30, 40] and [10, 20, 30, 40] aliases? False
a = [10, 20, 30, 40]    b = [10, 20, 35, 40]
```

The list_copy function is Listing 9.12 (listcopy.py) makes an actual copy of a. Changing an element of b does not affect list a.

In Section 9.4 we will see a more effective way to copy a list.

9.3 List Bounds

In the following code fragment:

```
a = [10, 20, 30, 40]
```

All of the following expressions are valid: a[0], a[1], a[2] and a[3]. The expression a[4] does not represent a valid element in the list. An attempt to use this expression, as in

```
a = [10, 20, 30, 40]
print(a[4])     #  Out-of-bounds access
```

results in a run-time exception. The interpreter will insist that the programmer use an integral value for an index, but in order to prevent a run-time exception the programmer must ensure that the index used is within the bounds of the list. Consider the following code:

```
#  Make a list containing 100 zeros
v = [0] * 100
#  User enters x at run time
x = int(input("Enter an integer: "))
v[x] = 1    #  Is this OK?  What is x?
```

Since the index may consist of an arbitrary integer expression whose value cannot be determined until run time, the interpreter checks every attempt to access a list. If the interpreter detects an out-of-bounds index, the interpreter raises an `IndexError` (list index out-of-bounds) exception. The programmer must ensure the provided index is in bounds to prevent such a run-time error.

The above unreliable code can be helped with conditional access:

```
#  Make a list containing 100 zeros
v = [0] * 100
#  User enters x at run time
x = int(input("Enter an integer: "))
#  Ensure index is within list bounds
if 0 <= x < len(v):
    v[x] = 1    #  This should be fine
else:
    print("Value provided is out of range")
```

Listing 9.13 (`badreverse.py`) attempts to print the list's elements in reverse order, but it fails to stay inside the bounds of the list.

Listing 9.13: **badreverse.py**

```
 1  def make_list():
 2      '''
 3      Builds a list from input provided by the user.
 4      '''
 5      result = []      #  List to return is initially empty
 6      in_val = 0       #  Ensure loop is entered at least once
 7      while in_val >= 0:
 8          in_val = int(input("Enter integer (-1 quits): "))
 9          if in_val >= 0:
10              result = result + [in_val]    #  Add item to list
11      return result
12
13  def main():
14      col = make_list()
15      #  Print the list in reverse
16      for i in range(len(col), 0, -1):
17          print(col[i], end=" ")
18      print()
19
20  main()
```

The `for` statement

```
for i in range(len(col), 0, -1):
    print(col[i], end=" ")
```

considers first the element at `col[len(col)]`, which is one index past the end of the list. The corrected `for` statement is

```
for i in range(len(col) - 1, -1, -1):
    print(col[i], end=" ")
```

A negative list index represents a negative offset from an imaginary element one past the end of the list. For list `lst`, the expression `lst[-1]` represents the last element in `lst`. The expression `lst[-2]` represents the next to last element, and so forth. The expression `lst[0]` thus corresponds to `lst[-len(lst)]`. Listing 9.14 (`negindex.py`) illustrates the use of negative indices to print a list in reverse.

Listing 9.14: `negindex.py`

```
def main():
    data = [10, 20, 30, 40, 50, 60]

    #  Print the individual elements with negative indices
    print(data[-1])
    print(data[-2])
    print(data[-3])
    print(data[-4])
    print(data[-5])
    print(data[-6])

    print('------')

    #  Print the list contents in reverse using negative indices
    for i in range(-1, -len(data) - 1, -1):
        print(data[i], end=' ')
    print()     #  Print newline

main()  #  Execute main
```

9.4 Slicing

We can make a new list from a portion of an existing list using a technique known as *slicing*. A list slice is an expression of the form

$$list\ [\ begin\ :\ end\ ]$$

where

- *list* is a list—a variable referring to a list object, a literal list, or some other expression that evaluates to a list,

- *begin* is an integer representing the starting index of a subsequence of the list, and

- *end* is an integer that is one larger than the index of the last element in a subsequence of the list.

If missing, the begin value defaults to 0. A begin value less than zero is treated as zero. If the end value is missing, it defaults to the length of the list. An end value greater than the length of the list is treated as the length of the list. The examples provided in Listing 9.15 (`listslice.py`) best illustrate how list slicing works.

Listing 9.15: `listslice.py`

```
lst = [10, 20, 30, 40, 50, 60, 70, 80]
```

```
2   print(lst)            #    [10, 20, 30, 40, 50, 60, 70, 80]
3   print(lst[0:3])       #    [10, 20, 30]
4   print(lst[4:8])       #    [50, 60, 70, 80]
5   print(lst[2:5])       #    [30, 40, 50]
6   print(lst[-5:-3])     #    [40, 50]
7   print(lst[:3])        #    [10, 20, 30]
8   print(lst[4:])        #    [50, 60, 70, 80]
9   print(lst[:])         #    [10, 20, 30, 40, 50, 60, 70, 80]
10  print(lst[-100:3])    #    [10, 20, 30]
11  print(lst[4:100])     #    [50, 60, 70, 80]
```

Slicing is the easiest way to make a copy of a list. The expression lst[:] evaluates to a copy of list lst.

Listing 9.16 (prefixsuffix.py) prints all the prefixes and suffixes of the list [1, 2, 3, 4, 5, 6, 7, 8].

Listing 9.16: prefixsuffix.py

```
1   a = [1, 2, 3, 4, 5, 6, 7, 8]
2   print('Prefixes of', a)
3   for i in range(0, len(a) + 1):
4       print('<', a[0:i], '>', sep='')
5   print('----------------------------------')
6   print('Suffixes of', a)
7   for i in range(0, len(a) + 1):
8       print('<', a[i:len(a) + 1], '>', sep='')
```

Listing 9.16 (prefixsuffix.py) prints

```
Prefixes of [1, 2, 3, 4, 5, 6, 7, 8]
<[]>
<[1]>
<[1, 2]>
<[1, 2, 3]>
<[1, 2, 3, 4]>
<[1, 2, 3, 4, 5]>
<[1, 2, 3, 4, 5, 6]>
<[1, 2, 3, 4, 5, 6, 7]>
<[1, 2, 3, 4, 5, 6, 7, 8]>
----------------------------------
Suffixes of [1, 2, 3, 4, 5, 6, 7, 8]
<[1, 2, 3, 4, 5, 6, 7, 8]>
<[2, 3, 4, 5, 6, 7, 8]>
<[3, 4, 5, 6, 7, 8]>
<[4, 5, 6, 7, 8]>
<[5, 6, 7, 8]>
<[6, 7, 8]>
<[7, 8]>
<[8]>
<[]>
```

When the slicing expression appears on the left side of the assignment operator it can modify the con-

tents of the list. This is known as *slice assignment*. A slice assignment can modify a list by removing or adding a subrange of elements in an existing list. Listing 9.17 (listslicemod.py) demonstrates how slice assignment can be used to modify a list.

Listing 9.17: `listslicemod.py`

```
1   lst = [10, 20, 30, 40, 50, 60, 70, 80]
2   print(lst)                       #  Print the list
3   lst[2:5] = ['a', 'b', 'c']    #  Replace [30, 40, 50] segment with ['a', 'b', 'c']
4   print(lst)
5   print('==================')
6   lst = [10, 20, 30, 40, 50, 60, 70, 80]
7   print(lst)                       #  Print the list
8   lst[2:6] = ['a', 'b']    #  Replace [30, 40, 50, 60] segment with ['a', 'b']
9   print(lst)
10  print('==================')
11  lst = [10, 20, 30, 40, 50, 60, 70, 80]
12  print(lst)              #  Print the list
13  lst[2:2] = ['a', 'b', 'c']    # Insert ['a', 'b', 'c'] segment at index 2
14  print(lst)
15  print('==================')
16  lst = [10, 20, 30, 40, 50, 60, 70, 80]
17  print(lst)        #  Print the list
18  lst[2:5] = []    #  Replace [30, 40, 50] segment with []
19  print(lst)
```

Listing 9.17 (listslicemod.py) displays:

```
[10, 20, 30, 40, 50, 60, 70, 80]
[10, 20, 'a', 'b', 'c', 60, 70, 80]
==================
[10, 20, 30, 40, 50, 60, 70, 80]
[10, 20, 'a', 'b', 70, 80]
==================
[10, 20, 30, 40, 50, 60, 70, 80]
[10, 20, 'a', 'b', 'c', 30, 40, 50, 60, 70, 80]
==================
[10, 20, 30, 40, 50, 60, 70, 80]
[10, 20, 60, 70, 80]
```

9.5 Lists and Functions

A list can be passed to a function, as shown in Listing 9.18 (listfunc.py)

Listing 9.18: `listfunc.py`

```
1   def sum(lst):
2       '''
3       Adds up the contents of a list of numeric
4       values
```

```
 5          lst is the list to sum
 6          Returns the sum of all the elements
 7          or zero if the list is empty.
 8          '''
 9          result = 0;
10          for item in lst:
11              result += item
12          return result
13
14   def clear(lst):
15          '''
16          Makes every element in list lst zero
17          '''
18          for i in range(len(lst)):
19              lst[i] = 0
20
21
22   def random_list(n):
23          '''
24          Builds a list of n integers, where each integer
25          is a pseudorandom number in the range 0...99.
26          Returns the random list.
27          '''
28          import random
29          result = []
30          for i in range(n):
31              rand = random.randrange(100)
32              result += [rand]
33          return result
34
35   def main():
36          a  = [2, 4, 6, 8]
37          #  Print the contents of the list
38          print(a)
39          #  Compute and display sum
40          print(sum(a))
41          #  Zero out all the elements of list
42          clear(a)
43          #  Reprint the contents of the list
44          print(a)
45          #  Compute and display sum
46          print(sum(a))
47          #  Test empty list
48          a = []
49          print(a)
50          print(sum(a))
51          #  Test pseudorandom list with 10 elements
52          a = random_list(10)
53          print(a)
54          print(sum(a))
55
56   main()
```

In Listing 9.18 (`listfunc.py`) the functions `sum` and `clear` accept a parameter of type list. Section 7.4 addressed the consequences of passing immutable types like integers and strings to functions. Since list objects are mutable, passing to a function a reference to a list object binds the formal parameter to the list object. This means the formal parameter becomes an alias of the actual parameter. The `sum` method does not attempt to modify its parameter, but the `clear` method changes every element in the list to zero. This means the `clear` function will modify the `a` list object in `main`.

9.6 Prime Generation with a List

Listing 9.19 (`fasterprimes.py`) uses an algorithm developed by the Greek mathematician Eratosthenes who lived from 274 B.C. to 195 B.C. Called the Sieve of Eratosthenes, the principle behind the algorithm is simple: Make a list of all the integers two and larger. Two is a prime number, but any multiple of two cannot be a prime number (since a multiple of two has two as a factor). Go through the rest of the list and mark out all multiples of two (4, 6, 8, ...). Move to the next number in the list (in this case, three). If it is not marked out, it must be prime, so go through the rest of the list and mark out all multiples of that number (6, 9, 12, ...). Continue this process until you have listed all the primes you want.

Listing 9.19 (`fasterprimes.py`) implements the Sieve of Eratosthenes in a Python function.

Listing 9.19: `fasterprimes.py`

```python
#  Display the prime numbers between 2 and 500

#  Largest potential prime considered
MAX = 500

def main():
    #  Each position in the Boolean list indicates
    #  if the number of that position is not prime:
    #  false means "prime," and true means "composite."
    #  Initially all numbers are prime until proven otherwise
    nonprimes = MAX * [False]  #  Initialize to all False

    #  First prime number is 2; 0 and 1 are not prime
    nonprimes[0] = nonprimes[1] = True

    #  Start at the first prime number, 2.
    for i in range(2, MAX + 1):
        #  See if i is prime
        if not nonprimes[i]:
            print(i, end=" ")
            #  It is prime, so eliminate all of its
            #  multiples that cannot be prime
            for j in range(2*i, MAX + 1, i)
                nonprimes[j] = True
    print()  #  Move cursor down to next line
```

How much better is the algorithm in Listing 9.19 (`fasterprimes.py`) than the square-root-optimized version we saw in Listing 6.8 (`timemoreefficientprimes.py`)? Listing 9.20 (`timeprimes.py`) compares the execution speed of the two algorithms.

Listing 9.20: `timeprimes.py`

```python
1   #  Count the number of prime numbers less than
2   #  2 million and time how long it takes
3   #  Compares the performance of two different
4   #  algorithms.
5
6   from time import clock
7   from math import sqrt
8
9
10  def count_primes(n):
11      '''
12      Generates all the prime numbers from 2 to n - 1.
13      n - 1 is the largest potential prime considered.
14      '''
15      start = clock()   #  Record start time
16
17      count = 0
18      for val in range(2, n):
19          result = True  #  Provisionally, n is prime
20          root = int(sqrt(val) + 1)
21          #  Try all potential factors from 2 to the square root of n
22          trial_factor = 2
23          while result and trial_factor <= root:
24              result = (val % trial_factor != 0 )  #  Is it a factor?
25              trial_factor += 1                    #  Try next candidate
26          if result:
27              count += 1
28
29      stop = clock()    #  Stop the clock
30      print("Count =", count, "Elapsed time:", stop - start, "seconds")
31
32
33
34  def seive(n):
35      '''
36      Generates all the prime numbers from 2 to n - 1.
37      n - 1 is the largest potential prime considered.
38      Algorithm originally developed by Eratosthenes.
39      '''
40
41      start = clock()   #  Record start time
42
43      #  Each position in the Boolean list indicates
44      #  if the number of that position is not prime:
45      #  false means "prime," and true means "composite."
46      #  Initially all numbers are prime until proven otherwise
47      nonprimes = n * [False]  #  Global list initialized to all False
48
49
50      count = 0
51
52      #  First prime number is 2; 0 and 1 are not prime
53      nonprimes[0] = nonprimes[1] = True
54
55      #  Start at the first prime number, 2.
```

```
56        for i in range(2, n):
57            #  See if i is prime
58            if not nonprimes[i]:
59                count += 1
60                #  It is prime, so eliminate all of its
61                #  multiples that cannot be prime
62                for j in range(2*i, n, i):
63                    nonprimes[j] = True
64        #  Print the elapsed time
65        stop = clock()
66        print("Count =", count, "Elapsed time:", stop - start, "seconds")
67
68
69  def main():
70      count_primes(2000000)
71      seive(2000000)
72
73  main()
```

Since printing to the screen takes up the majority of the time, Listing 9.20 (`timeprimes.py`) counts the number of primes rather than printing each one. This allows us to better compare the behavior of the two approaches. The square root version has been optimized slightly more: the floating-point `root` variable is not an integer. The less than comparison between two integers is faster than the floating-point equivalent.

The output of Listing 9.20 (`timeprimes.py`) on one system reveals

```
Count = 148932 Elapsed time: 56.446094827055276 seconds
Count = 148933 Elapsed time: 0.8864909583615557 seconds
```

Our previous version requires almost a minute (56 seconds) to count the number of primes less than two million, while the version based on the Sieve of Eratosthenes takes less than one second. The Sieve version is over 60 times faster than the optimized square root version.

9.7 Summary

- A list represents an ordered sequence of objects

- An element in a list may be accessed via its index using `[]`. The first element is at index 0. If the list contains n elements, the index of the last element is $n-1$.

- A positive list index is an offset from the beginning of the list. A negative list index is an offset back from an imaginary element one past the end of the list.

- A list may elements of different types.

- List literals list their elements in a comma-separated list enclosed within square brackets (`[]`).

- The `len` function returns the length of the list

- A list index is sometimes called a subscript.

- A list subscript must evaluate to an integer. Integer literals, variables, and expressions can be used as list indices.

- A `for` loop is a convenient way to traverse the contents of a list.

- Like other variables, a list variable can be local, global, or a function parameter.

- Direct list assignment produces an alias. A slice of a whole list makes an actual copy of the list.

- The `==` tests for equal contents within lists; the `is` operator tests for list aliases.

- A list may be passed to a function. The formal parameter within the function becomes an alias of the actual parameter passed by the client. This means functions may modify the contents of a list, and the modification will affect the client's copy of the list.

- It is the programmer's responsibility to stay within the bounds of a list. Venturing outside the bounds of a list results in a run-time error.

- Lists are mutable objects. Integers, floating-point, and string values are immutable.

- Parts of lists can be expressed with slices. A slice is a copy of a subrange of elements in a list.

- List slices on the right side the assignment operator can modify lists by removing or adding a subrange of elements in an existing list.

9.8 Exercises

1. Can a Python list hold a mixture of integers and strings?

2. What happens if you attempt to access an element of a list using a negative index?

3. Given the statement

    lst = [10, −4, 11, 29]

    (a) What expression represents the very first element of `lst`?
    (b) What expression represents the very last element of `lst`?
    (c) What is `lst[0]`?
    (d) What is `lst[3]`?
    (e) What is `lst[1]`?
    (f) What is `lst[-1]`?
    (g) What is `lst[-4]`?
    (h) Is the expression `lst[3.0]` legal or illegal?

4. What Python statement produces a list containing contains the values 45, −3, 16 and 8?

5. What function returns the number of elements in a list?

6. Given the list

    lst = [20, 1, −34, 40, −8, 60, 1, 3]

    evaluate the following expressions:

    (a) `lst`

    (b) `lst[0:3]`

    (c) `lst[4:8]`

    (d) `lst[4:33]`

    (e) `lst[-5:-3]`

    (f) `lst[-22:3]`

    (g) `lst[4:]`

    (h) `lst[:]`

    (i) `lst[:4]`

    (j) `lst[1:5]`

7. An assignment statement containing the expression `a[m:n]` on the left side and a list on the right side can modify list `a`. Complete the following table by supplying the m and n values in the slice assignment statement needed to produce the indicated list from the given original list.

		Slice indices	
Original List	Target List	m	n
`[2, 4, 6, 8, 10]`	`[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]`		
`[2, 4, 6, 8, 10]`	`[-10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10]`		
`[2, 4, 6, 8, 10]`	`[2, 3, 4, 5, 6, 7, 8, 10]`		
`[2, 4, 6, 8, 10]`	`[2, 4, 6, 'a', 'b', 'c', 8, 10]`		
`[2, 4, 6, 8, 10]`	`[2, 4, 6, 8, 10]`		
`[2, 4, 6, 8, 10]`	`[]`		
`[2, 4, 6, 8, 10]`	`[10, 8, 6, 4, 2]`		
`[2, 4, 6, 8, 10]`	`[2, 4, 6]`		
`[2, 4, 6, 8, 10]`	`[6, 8, 10]`		
`[2, 4, 6, 8, 10]`	`[2, 10]`		
`[2, 4, 6, 8, 10]`	`[4, 6, 8]`		

8. Complete the following function that adds up all the *positive* values in a list of integers. For example, if list `a` contains the elements $3, -3, 5, 2, -1$, and 2, the call `sum_positive(a)` would evaluate to 12, since $3 + 5 + 2 + 2 = 12$. The function returns zero if the list is empty.

```
def sum_positive(a):
    #  Add your code...
```

9. Complete the following function that counts the even numbers in a list of integers. For example, if list `a` contains the elements $3, 5, 2, -1$, and 2, the call `count_evens(a)` would evaluate to 4, since $2 + 2 = 4$. The function returns zero if the list is empty. The function does not affect the contents of the list.

```
def count_evens(a):
    #  Add your code...
```

10. Write a function named `print_big_enough` that accepts two parameters, a list of numbers and a number. The function should print, in order, all the elements in the list that are at least as large as the second parameter.

11. Write a function named `reverse` that reorders the contents of a list so they are reversed from their original order. `a` is a list. Note that your function must physically rearrange the elements within the list, not just print the elements in reverse order.

Chapter 10

List Processing

Lists, introduced in Chapter 9, are convenient structures for storing large amounts of data. In this chapter we examine several algorithms that allow us to rearrange the elements of a list in a regular way and efficiently search for elements within a list.

10.1 Sorting

Sorting—arranging the elements within a list into a particular order—is a common activity. For example, a list of integers may be arranged in ascending order (that is, from smallest to largest). A list of strings may be arranged in lexicographical (commonly called alphabetical) order. Many sorting algorithms exist, and some perform much better than others. We will consider one sorting algorithm that is relatively easy to implement.

The *selection sort* algorithm is relatively easy to implement and easy to understand how it works. If A is a list, and i represents a list index, selection sort works as follows:

1. Set n = length of list A.

2. Set $i = 0$.

3. Examine all the elements $A[j]$, where $i < j < n$. (This simply means to consider all the elements in the list from index i to the end.) If any of these elements is less than $A[i]$, then exchange $A[i]$ with the smallest of these elements. (This ensures that all elements after position i are greater than or equal to $A[i]$.)

4. If i is less than $n-1$, set i equal to $i+1$ and go to Step 2.

5. Done; list A is sorted.

The command to "go to Step 2" in Step 4 represents a loop. When the value of i in Step 3 equals n, the algorithm terminates with a sorted list.

We can begin to translate the above description into Python as follows:

```python
n = len(A)
for i in range(n - 1):
```

```
        # Examine all the elements A[j], where i < j < n.
        # If any of these A[j] is less than A[i],
        # then exchange A[i] with the smallest of these elements.
```

The directive at Step 2 beginning with "Examine all the elements $A[j]$, where $i < j < n$" also must be implemented as a loop. We continue refining our implementation with:

```
n = len(A)
for i in range(n - 1):
    # Examine all the elements A[j], where i < j < n.
    for j in range(i + 1, n):
    # If any A[j] is less than A[i],
    # then exchange A[i] with the smallest of these elements.
```

In order to determine if any of the elements is less than A[i], we introduce a new variable named small. The purpose of small is to keep track of the position of the smallest element found so far. We will set small equal to i initially, because we wish to locate any element less than the element located at position i.

```
n = len(A)
for i in range(n - 1):
    #  small is the position of the smallest value we've seen
    #  so far; we use it to find the smallest value less than A[i]
    small = i
    for j in range(i + 1, n):
       if A[j] < A[small]:
            small = j  # Found a smaller element, update small
    #  If small changed, we found an element smaller than A[i]
    if small != i:
        #  exchange A[small] and A[i]
```

Listing 10.1 (sortintegers.py) provides the complete Python implementation of the selection_sort function within a program that tests it out.

Listing 10.1: sortintegers.py

```
1  from random import randrange
2
3  def random_list():
4      '''
5      Produce a list of pseudorandom integers.
6      The list's length is chosen pseudorandomly in the
7      range 3-20.
8      The integers in the list range from -50 to 50.
9      '''
10     result = []
11     count = randrange(3, 20)
12     for i in range(count):
13         result += [randrange(-50, 50)]
14     return result
15
16 def selection_sort(lst):
17     '''
18     Arranges the elements of list lst in ascending order.
```

```
19        The contents of lst are physically rearranged.
20        '''
21    n = len(lst)
22    for i in range(n - 1):
23        #  Note: i, small, and j represent positions within lst
24        #  lst[i], lst[small], and lst[j] represent the elements at
25        #  those positions.
26        #  small is the position of the smallest value we've seen
27        #  so far; we use it to find the smallest value less
28        #  than lst[i]
29        small = i
30        #  See if a smaller value can be found later in the list
31        #  Consider all the elements at position j, where i < j < n
32        for j in range(i + 1, n):
33            if lst[j] < lst[small]:
34                small = j        #  Found a smaller value
35        #  Swap lst[i] and lst[small], if a smaller value was found
36        if i != small:
37            lst[i], lst[small] = lst[small], lst[i]

def main():
    '''
    Tests the selection_sort function
    '''
    for n in range(10):
        col = random_list()
        print(col)
        selection_sort(col)
        print(col)
        print('==============================')

main()
```

One run of Listing 10.1 (`sortintegers.py`) produces:

```
[-23, 47, -3, 4, 5, -46, 26, -27]
[-46, -27, -23, -3, 4, 5, 26, 47]
==============================
[32, -10, -4, 41, 10, -1, -31, 3, 28, -31, -33, 46, -45, -6, 37]
[-45, -33, -31, -31, -10, -6, -4, -1, 3, 10, 28, 32, 37, 41, 46]
==============================
[11, -19, 20, 43, -19, 20, -18, -17]
[-19, -19, -18, -17, 11, 20, 20, 43]
==============================
[9, -22, -41, 35, 10, 48, 9, 14, -20]
[-41, -22, -20, 9, 9, 10, 14, 35, 48]
==============================
[-38, -3, -7, 41, -8, -11, -23, 9, -47, 38]
[-47, -38, -23, -11, -8, -7, -3, 9, 38, 41]
==============================
[-47, 1, -37, 16, -40, -14, 2, 38, 43, 19, 45]
[-47, -40, -37, -14, 1, 2, 16, 19, 38, 43, 45]
==============================
[8, 39, 35, -42]
[-42, 8, 35, 39]
==============================
[-8, -22, -13, 47, -28, -46, -21, -42, 27, 14, 47, -21, 2, -47]
[-47, -46, -42, -28, -22, -21, -21, -13, -8, 2, 14, 27, 47, 47]
==============================
[37, -21, -32, -7]
[-32, -21, -7, 37]
==============================
[33, -42, -26, 35, 37, 36, -1, 47, 24, 5, 41, -6, 48, 6, 43]
[-42, -26, -6, -1, 5, 6, 24, 33, 35, 36, 37, 41, 43, 47, 48]
==============================
```

Notice than in each case the elements in the pseudorandomly generated list are rearranged into correct ascending order. To check the correctness of our sort we need to be sure that:

- the sorted list contains the same number of elements as the original, unsorted list,

- no elements in the original list are missing,

- no elements in the sorted list appear more frequently than they did in the original, unsorted list, and

- the elements appear in ascending order.

The output of Listing 10.1 (sortintegers.py) provides evidence that our selection_sort function is working correctly.

10.2 Flexible Sorting

What if want to change the behavior of the sorting function in Listing 10.1 (sortintegers.py) so that it arranges the elements in descending order instead of ascending order? It is actually an easy modification;

simply change the line

```
        if lst[j] < lst[small]:
```

to be

```
        if lst[j] > lst[small]:
```

What if instead we want to change the sort so that it sorts the elements in ascending order except that all the even numbers in the list appear before all the odd numbers? This modification would be a little more complicated, but it could accomplished in that `if` statement's conditional expression.

The next question is more intriguing: How can we rewrite the `selection_sort` function so that, by passing an additional parameter, it can sort the list in any way we want?

We can make our sort function more flexible by passing an ordering function as a parameter (see Section 8.6 for examples of functions as parameters to other functions). Listing 10.2 (`flexiblesort.py`) arranges the elements in a list two different ways using the same `selection_sort` function.

Listing 10.2: **flexiblesort.py**

```
 1  def random_list():
 2      '''
 3      Produce a list of pseudorandom integers.
 4      The list's length is chosen pseudorandomly in the
 5      range 3-20.
 6      The integers in the list range from -50 to 50.
 7      '''
 8      from random import randrange
 9      result = []
10      count = randrange(3, 20)
11      for i in range(count):
12          result += [randrange(-50, 50)]
13      return result
14
15
16  def less_than(m, n):
17      return m < n
18
19  def greater_than(m, n):
20      return m > n
21
22
23  def selection_sort(lst, cmp):
24      '''
25      Arranges the elements of list lst in ascending order.
26      The comparer function cmp is used to order the elements.
27      The contents of lst are physically rearranged.
28      '''
29      n = len(lst)
30      for i in range(n - 1):
31          #  Note: i, small, and j represent positions within lst
32          #  lst[i], lst[small], and lst[j] represent the elements at
33          #  those positions.
34          #  small is the position of the smallest value we've seen
35          #  so far; we use it to find the smallest value less
36          #  than lst[i]
```

```
37          small = i
38          #  See if a smaller value can be found later in the list
39          #  Consider all the elements at position j, where i < j < n.
40          for j in range(i + 1, n):
41              if cmp(lst[j], lst[small]):
42                  small = j        #  Found a smaller value
43          #  Swap lst[i] and lst[small], if a smaller value was found
44          if i != small:
45              lst[i], lst[small] = lst[small], lst[i]
46
47  def main():
48      '''
49      Tests the selection_sort function
50      '''
51      original = random_list()        #  Make a random list
52      working = original[:]           #  Make a working copy of the list
53      print('Original:  ', working)
54      selection_sort(working, less_than)  #  Sort ascending
55      print('Ascending: ', working)
56      working = original[:]           #  Make a working copy of the list
57      print('Original:   ', working)
58      selection_sort(working, greater_than)  #  Sort descending
59      print('Descending:', working)
60
61  main()
```

The output of Listing 10.2 (`flexiblesort.py`) is

```
Original:   [-8, 24, -46, -7, -26, -29, -44]
Ascending:  [-46, -44, -29, -26, -8, -7, 24]
Original:   [-8, 24, -46, -7, -26, -29, -44]
Descending: [24, -7, -8, -26, -29, -44, -46]
```

The comparison function passed to the sort routine customizes the sort's behavior. The basic structure of the sorting algorithm does not change, but its notion of ordering is adjustable. If the second parameter to `selection_sort` is `less_than`, the function arranges the elements ascending order. If the second parameter instead is `greater_than`, the function sorts the list in descending order. More creative orderings are possible with more elaborate comparison functions.

Selection sort is a relatively efficient simple sort, but more advanced sorts are, on average, much faster than selection sort, especially for large data sets. One such general purpose sort is *Quicksort*, devised by C. A. R. Hoare in 1962. Quicksort is the fastest known general purpose sort.

10.3 Search

Searching a list for a particular element is a common activity. We examine two basic strategies: linear search and binary search.

10.3.1 Linear Search

Listing 10.3 (`linearsearch.py`) uses a function named `locate` that returns the position of the first occurrence of a given element in a list; if the element is not present, the function returns `None`.

Listing 10.3: `linearsearch.py`

```python
def locate(lst, seek):
    '''
    Returns the index of element seek in list lst,
    if seek is present in lst.
    Returns None if seek is not an element of lst.
    lst is the lst in which to search.
    seek is the element to find.
    '''
    for i in range(len(lst)):
        if lst[i] == seek:
            return i       #  Return position immediately
    return None            #  Element not found

def format(i):
    '''
    Prints integer i right justified in a 4-space
    field.  Prints "****" if i > 9,999.
    '''
    if i > 9999:
        print("****")      #  Too big!
    else:
        print("%4d" % i)

def show(lst):
    '''
    Prints the contents of list lst
    '''
    for item in lst:
        print("%4d" % item, end='')   #  Print element right justifies in 4 spaces
    print()                           #  Print newline

def draw_arrow(value, n):
    '''
    Print an arrow to value which is an element in a list.
    n specifies the horizontal offset of the arrow.
    '''
    print(("%" + str(n) + "s") % "   ^   ")
    print(("%" + str(n) + "s") % "   |   ")
    print(("%" + str(n) + "s%i") % ("   +-- ", value))


def display(lst, value):
    '''
    Draws an ASCII art arrow showing where
    the given value is within the list.
    lst is the list.
    value is the element to locate.
    '''
```

```
49        show(lst)                      #  Print contents of the list
50        position = locate(lst, value)
51        if position != None:
52            position = 4*position + 7;  #  Compute spacing for arrow
53            draw_arrow(value, position)
54        else:
55            print("(", value, " not in list)", sep='')
56        print()
57
58
59  def main():
60      a = [100, 44, 2, 80, 5, 13, 11, 2, 110]
61      display(a, 13)
62      display(a, 2)
63      display(a, 7)
64      display(a, 100)
65      display(a, 110)
66
67  main()
```

The output of Listing 10.3 (`linearsearch.py`) is

```
100   44   2  80   5  13  11   2 110
                        ^

                        |
                        +-- 13


100   44   2  80   5  13  11   2 110
           ^

           |
           +-- 2


100   44   2  80   5  13  11   2 110
(7 not in list)


100   44   2  80   5  13  11   2 110
      ^

      |
      +-- 100


100   44   2  80   5  13  11   2 110
                                ^

                                |
                                +-- 110
```

The key function in Listing 10.3 (`linearsearch.py`) is `locate`; all the other functions simply lead to a more interesting display of `locate`'s results. If `locate` finds a match, the function immediately returns the position of the matching element; otherwise, if after examining all the elements of the list it cannot find the element sought, the function returns `None`. Here `None` indicates the function could not return a valid answer.

The client code, in this example the `display` function, must ensure that `locate`'s result is not `None` before attempting to use the result as an index into a list.

The kind of search performed by `locate` is known as *linear search*, since a straight line path is taken from the beginning of the list to the end of the list considering each element in order. Figure 10.1 illustrates linear search.
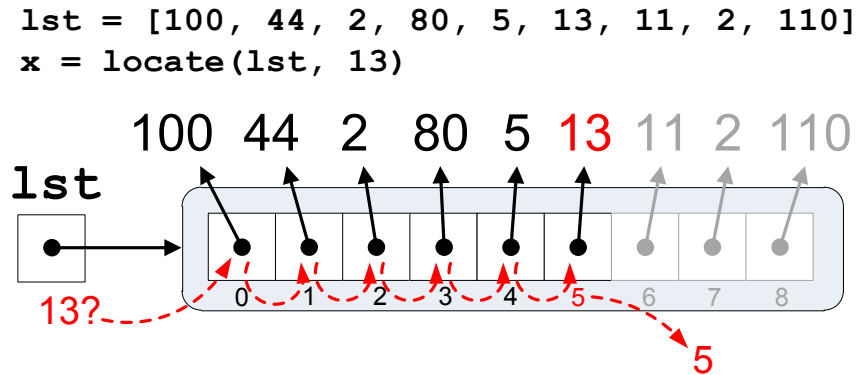
```
lst = [100, 44, 2, 80, 5, 13, 11, 2, 110]
x = locate(lst, 13)
```


Figure 10.1: Linear search

10.3.2 Binary Search

Linear search is acceptable for relatively small lists, but the process of examining each element in a large list is time consuming. An alternative to linear search is *binary search*. In order to perform binary search, a list must be in sorted order. Binary search exploits the sorted structure of the list using a clever but simple strategy that quickly zeros in on the element to find:

1. If the list is empty, return `None`.

2. Check the element in the middle of the list. If that element is what you are seeking, return its position. If the middle element is larger than the element you are seeking, perform a binary search on the first half of the list. If the middle element is smaller than the element you are seeking, perform a binary search on the second half of the list.

This approach is analogous to looking for a telephone number in the phone book in this manner:

1. Open the book at its center. If the name of the person is on one of the two visible pages, look at the phone number.

2. If not, and the person's last name is alphabetically less the names on the visible pages, apply the search to the left half of the open book; otherwise, apply the search to the right half of the open book.

3. Discontinue the search with failure if the person's name should be on one of the two visible pages but is not present.

The binary search algorithm can be implemented as a Python function as shown in Listing 10.4 (`binarysearch.py`).

Listing 10.4: binarysearch.py

```python
def binary_search(lst, seek):
    '''
    Returns the index of element seek in list lst,
    if seek is present in lst.
    Returns None if seek is not an element of lst.
    lst is the lst in which to search.
    seek is the element to find.
    '''
    first = 0              #  Initialize the first position in list
    last = len(lst) - 1   #  Initialize the last position in list
    while first <= last:
        #  mid is middle position in the list
        mid = first + (last - first + 1)//2   # Note: Integer division
        if lst[mid] == seek:
            return mid        #  Found it
        elif lst[mid] > seek:
            last = mid - 1    #  continue with 1st half
        else:  #  v[mid] < seek
            first = mid + 1  #  continue with 2nd half
    return None    #  Not there


def show(lst):
    '''
    Prints the contents of list lst
    '''
    for item in lst:
        print("%4d" % item, end='')   #  Print element right justifies in 4 spaces
    print()                          #  Print newline

def draw_arrow(value, n):
    '''
    Print an arrow to value which is an element in a list.
    n specifies the horizontal offset of the arrow.
    '''
    print(("%" + str(n) + "s") % "   ^    ")
    print(("%" + str(n) + "s") % "   |    ")
    print(("%" + str(n) + "s%i") % ("   +-- ", value))


def display(lst, value):
    '''
    Draws an ASCII art arrow showing where
    the given value is within the list.
    lst is the list.
    value is the element to locate.
    '''
    show(lst)                  #  Print contents of the list
    position = binary_search(lst, value)
    if position != None:
        position = 4*position + 7;  #  Compute spacing for arrow
        draw_arrow(value, position)
```

```
53        else:
54            print("(", value, " not in list)", sep='')
55        print()
56
57
58  def main():
59      a = [2, 5, 11, 13, 44, 80, 100, 110]
60      display(a, 13)
61      display(a, 2)
62      display(a, 7)
63      display(a, 100)
64      display(a, 110)
65
66  main()
```

In the `binary_search` function:

- The initializations of `first` and `last`:

  ```
  first = 0                # Initialize the first position in list
  last = len(lst) - 1  #  Initialize the last position in list
  ```

  ensure that `first` is less than or equal to `last` for a nonempty list. If the list is empty, `first` is zero, and `last` is equal to `len(lst) - 1 = 0 - 1 = -1`. So in the case of an empty list the function will skip the loop and return `None`. This is correct behavior because an empty list cannot possibly contain any item we seek.

- The calculation of `mid` ensures that $first \leq mid \leq last$.

- If `mid` is the location of the sought element (checked in the first `if` statement), the loop terminates, and returns the correct position.

- The `elif` and `else` clauses ensure that either `last` decreases or `first` increases each time through the loop. Thus, if the loop does not terminate for other reasons, eventually `first` will be larger than `last`, and the loop will terminate. If the loop terminates for this reason, the function returns `None`. This is the correct behavior.

- The modification to either `first` or `last` in the `elif` and `else` clauses exclude irrelevant elements from further search. The number of elements to consider is cut in half each time through the loop.

Figure 10.2 illustrates how binary search works.

The implementation of the binary search algorithm is more complicated than the simpler linear search algorithm. Ordinarily simpler is better, but for algorithms that process data structures that potentially hold large amounts of data, more complex algorithms employing clever tricks that exploit the structure of the data (as binary search does) often dramatically outperform simpler, easier-to-code algorithms.

For a fair comparison of linear vs. binary search, suppose we want to locate an element in a sorted list. That the list is ordered is essential for binary search, but it can be helpful for linear search as well. The revised linear search algorithm for ordered lists is

```
#  This version requires list lst to be sorted in
#  ascending order.
def linear_search(lst, seek):
    i = 0                    #  Start at beginning
```

```
lst = [10, 14, 20, 28, 29, 33, 34, 45, 48]
x = locate(lst, 33)
```
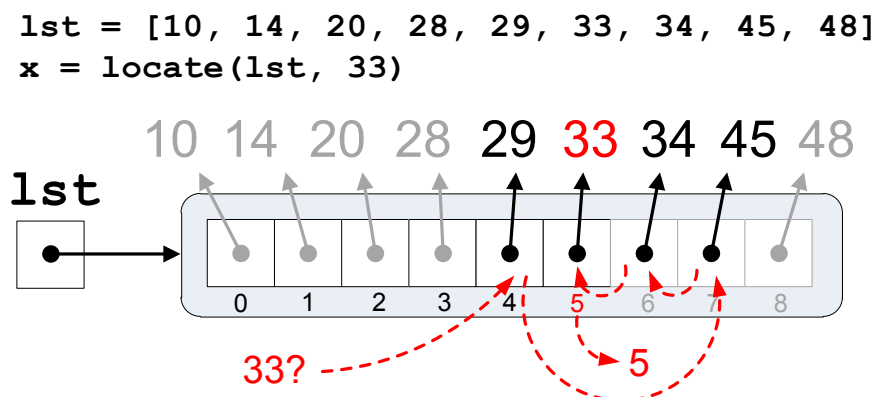

Figure 10.2: Binary search

```
n = len(lst)          #  Length of list
while i < n and lst[i] <= seek:
    if  lst[i] == seek:
        return i      #  Return position immediately
return None           #  Element not found
```

Notice that, as in the original version of linear search, the loop will terminate when all the elements have been examined, but this version will terminate early when it encounters an element larger than the sought element. Since the list is sorted, there is no need to continue the search once the search has found an element larger than the value sought; seek cannot appear after a larger element in a sorted list.

Suppose a list to search contains n elements. In the worst case—looking for an element larger than any currently in the list—the loop in linear search takes n iterations. In the best case—looking for an element smaller than any currently in the list—the function immediately returns without considering any other elements. The number of loop iterations thus ranges from 1 to n, and so on average linear search requires $\frac{n}{2}$ comparisons before the loop finishes and the function returns.

Now consider binary search. After each comparison the size of the list left to consider is one-half the original size. If the sought item is not found on the first probe, the number of remaining elements to search is $\frac{n}{2}$. The next time through the loop, the number of elements left to consider drops to $\frac{n}{4}$, then $\frac{n}{8}$, and so forth. The problem of determining how many times a set of things can be divided in half until only one element remains can be solved with a base-2 logarithm. For binary search, the worst case scenario of not finding the sought element requires the loop to make $\log_2 n$ iterations.

How does this analysis help us determine which search is better? The quality of an algorithm is judged by two key characteristics:

- How much time (processor cycles) does it take to run?

- How much space (memory) does it take to run?

In our situation, both search algorithms process the list with only a few extra local variables, so for large lists they both require essentially the same space. The big difference here is speed. Binary search performs more elaborate computations each time through the loop, and each operation takes time, so perhaps binary search is slower. Linear search is simpler (fewer operations through the loop), but perhaps its loop executes many more times than the loop in binary search, so overall it is slower.

We can deduce the faster algorithm in two ways: empirically and analytically. An empirical test is an experiment; we carefully implement both algorithms and then measure their execution times. The analytical approach analyzes the source code to determine how many operations the computer's processor must perform to run the program on a problem of a particular size.

Listing 10.5 (`searchcompare.py`) gives us some empirical results.

Listing 10.5: **searchcompare.py**

```python
def binary_search(lst, seek):
    '''
    Returns the index of element seek in list lst,
    if seek is present in lst.
    lst must be in sorted order.
    Returns None if seek is not an element of lst.
    lst is the lst in which to search.
    seek is the element to find.
    '''
    first = 0            #  Initially the first element in list
    last = len(lst) - 1  #  Initially the last element in list
    while first <= last:
        #  mid is middle of the list
        mid = first + (last - first + 1)//2  # Note: Integer division
        if lst[mid] == seek:
            return mid       #  Found it
        elif lst[mid] > seek:
            last = mid - 1   #  continue with 1st half
        else:  #  v[mid] < seek
            first = mid + 1  #  continue with 2nd half
    return None    #  Not there

def ordered_linear_search(lst, seek):
    '''
    Returns the index of element seek in list lst,
    if seek is present in lst.
    lst must be in sorted order.
    Returns None if seek is not an element of lst.
    lst is the lst in which to search.
    seek is the element to find.
    '''
    i = 0
    n = len(lst)
    while i < n and lst[i] <= seek:
        if lst[i] == seek:
            return i       #  Return position immediately
        i += 1
    return None            #  Element not found

def test_searches(lst):
    from time import clock
```

```
42
43        #  Find each element using ordered linear search
44        start = clock()     #  Start the clock
45        n = len(lst)
46        for i in range(n):
47            if ordered_linear_search(lst, i) != i:
48                print("error")
49        stop = clock()      #  Stop the clock
50        print("Linear elapsed time", stop - start)
51
52        #  Find each element using binary search
53        start = clock()     #  Start the clock
54        n = len(lst)
55        for i in range(n):
56            if binary_search(lst, i) != i:
57                print("error")
58        stop = clock()      #  Stop the clock
59        print("Binary elapsed time", stop - start)
60
61   def main():
62        SIZE = 20000
63        test_list = list(range(SIZE))
64        test_searches(test_list)
65
66   main()
```

The `test_searches` function in Listing 10.5 (`searchcompare.py`) searches for all the elements in a list using first ordered linear search and then binary search. On one system, Listing 10.5 (`searchcompare.py`) produces:

```
Linear elapsed time 72.68487246407194
Binary elapsed time 0.19088075623170653
```

The ordered linear search exercises take over one 72 seconds, while the binary search applied to the exact same searches takes less than one-fifth of a second. Binary search is almost 400 times faster than ordered linear search!

Table 10.1 lists the results for various sized lists. Empirically, binary search performs dramatically better than linear search. Figure 10.3 plots the values in Table 10.1.

In addition to using the empirical approach, we can judge which algorithm is better by analyzing the source code for each function. Each arithmetic operation, assignment, logical comparison, and list access requires time to execute. We will assume each of these activities requires one unit of processor "time." This assumption is not strictly true, but it will give good results for relative comparisons. Since we will follow the same rules when analyzing both search algorithms, the relative results for comparison purposes will be fairly accurate.

We first consider linear search. We determined that, on average, the loop makes $\frac{n}{2}$ iterations for a list of size n. The initialization of i happens only one time during each call to `linear_search`. All other activity involved with the loop except the `return` statements happens $\frac{n}{2}$ times. Either i or `None` will be returned, and only one return is executed during each call. Table 10.2 shows the breakdown for linear search. The

List Size	Linear Search	Binary Search
0	0.415	0.415
10	5.488	4.205
20	9.002	5.804
30	16.266	9.081
40	31.486	13.206
50	36.579	17.786
60	57.760	21.004
70	76.474	24.794
80	101.229	28.426
90	127.345	33.800

Table 10.1: Run-time behavior of linear and binary search on lists of different sizes. Time is listed in 10^{-5} seconds.

Action	Operation(s)	Operation Count	Times Executed	Total Cost
`i = 0`	=	1	1	1
`n = len(lst)`	=, len	2	1	2
`while i < n and lst[i] <= seek:`	<, and, <=, []	4	$\frac{n}{2}$	$2n$
`    if lst[i] == seek:`	[], ==	2	$\frac{n}{2}$	n
`        return i`	return	1	$\frac{1}{2}$	$\frac{1}{2}$
`return None`	return	1	$\frac{1}{2}$	$\frac{1}{2}$
			Total time units	$3n + 4$

Table 10.2: Analysis of Linear Search Algorithm. The $\frac{n}{2}$ loop iterations is based on average time to locate an element. The function will execute exactly one of the two `return` statements during a given call, so each is given a cost of $\frac{1}{2}$.
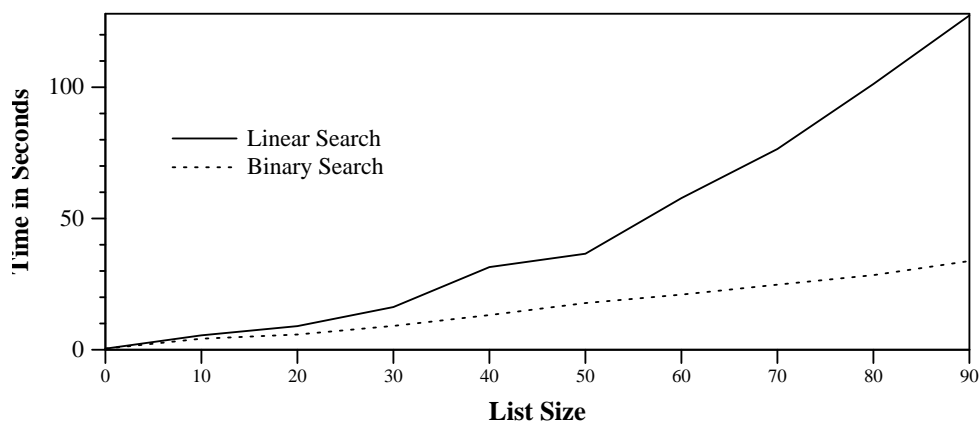
Figure 10.3: Linear vs. Binary Search

results in Table 10.2 indicate the running time of the `linear_search` function can be expressed as a simple mathematical linear function: $f(n) = 3n + 4$.

Next, we consider binary search. We determined that in the worst case the loop in `binary_search` iterates $\log_2 n$ times if the list contains n elements. The two initializations before the loop are performed once per call. Most of the actions within the loop occur $\log_2 n$ times, except that only one `return` statement can be executed per call, and in the `if/elif/else` statement only one path can be chosen per loop iteration. 10.3 shows the complete analysis of binary search. Figure 10.4 shows the plot of the two functions $3n + 4$ and $12 \log_2 n + 6$. Note the similarity of these pure function curves to the curves in Figure 10.3.
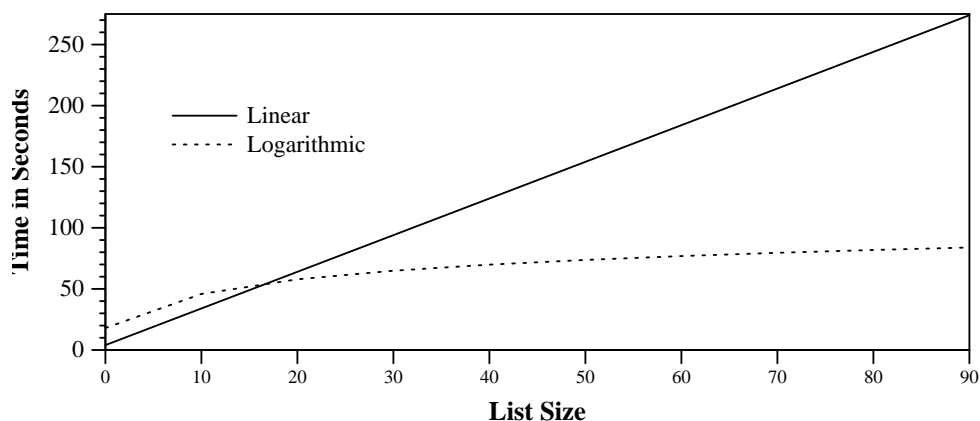

Figure 10.4: A graph of the functions derived from analyzing the linear and binary search routines

The bottom line is that binary search is fast even for large lists.

Action	Operation(s)	Operation Count	Times Executed	Total Cost
`first = 0`	=	1	1	1
`last = len(lst) - 1`	=, len, -	3	1	3
`while first <= last:`	<=	1	$\log_2 n$	$\log_2 n$
`    mid=first+(last-first+1)//2`	=, +, -, +, //	5	$\log_2 n$	$5\log_2 n$
`    if lst[mid] == seek:`	[], ==	2	$\log_2 n$	$2\log_2 n$
`        return mid`	return	1	1	1
`    elif lst[mid] > seek:`	[], >	2	$\log_2 n$	$2\log_2 n$
`        last = mid - 1`	=, -	2	$\frac{1}{2}\log_2 n$	$\log_2 n$
`    else:`		0		0
`        first = mid + 1`	=, +	2	$\frac{1}{2}\log_2 n$	$\log_2 n$
`return None`	return	1	1	1
			Total time units	$12\log_2 n + 6$

Table 10.3: Analysis of Binary Search Algorithm. Each time through the loop the function executes either the `elif` or `else` statement, so each one is charged is charged $\frac{1}{2}$ its actual cost.

10.4 List Permutations

Sometimes it is useful to consider all the possible arrangements of the elements within a list. A sorting algorithm, for example, must work correctly on any initial arrangement of elements in a list. To test a sort function, a programmer could check to see to see if it produces the correct result for all arrangements of a relatively small list. A rearrangement of a collection of ordered items is called a *permutation*. Listing 10.6 (`listpermutations.py`) generates all the permutations of a given list.

Listing 10.6: `listpermutations.py`

```python
def permute(prefix, suffix):
    '''
    Recursively shifts all the elements in suffix into
    prefix producing all the permutations of suffix.
    Prints all permutations in lexicographical order.
    '''
    suffix_size = len(suffix)
    if suffix_size == 0:  # Have we considered all the elements?
        print(prefix)
    else:
        for i in range(0, suffix_size):
            new_pre = prefix + [suffix[i]]
            new_suff = suffix[:i] + suffix[i + 1:]
            permute(new_pre, new_suff)



def print_permutations(lst):
    '''
    Calls the recursive permute function to display
    all the permutations of the elements of lst in
    lexicographical order.  The empty list is passed as
    the first parameter to permute, and the list to
```

```
24        permute is passed as the second argument.
25        '''
26        permute([], lst)
27
28  def main():
29        a = [1, 2, 3, 4]
30        print_permutations(a)
31
32  main()
```

Listing 10.6 (listpermutations.py) produces the following output:

```
[1, 2, 3, 4]
[1, 2, 4, 3]
[1, 3, 2, 4]
[1, 3, 4, 2]
[1, 4, 2, 3]
[1, 4, 3, 2]
[2, 1, 3, 4]
[2, 1, 4, 3]
[2, 3, 1, 4]
[2, 3, 4, 1]
[2, 4, 1, 3]
[2, 4, 3, 1]
[3, 1, 2, 4]
[3, 1, 4, 2]
[3, 2, 1, 4]
[3, 2, 4, 1]
[3, 4, 1, 2]
[3, 4, 2, 1]
[4, 1, 2, 3]
[4, 1, 3, 2]
[4, 2, 1, 3]
[4, 2, 3, 1]
[4, 3, 1, 2]
[4, 3, 2, 1]
```

Notice that every possible unique arrangement of the elements in the list [1, 2, 3, 4] appear in the output.

The permute function in Listing 10.6 (listpermutations.py) uses a loop and recursion to generate all the possible orderings for a given list. Recursion can be difficult to follow, but we can better understand the process by *instrumenting* the permute function as follows:

```
def permute(prefix, suffix, depth):
    '''
    Recursively shifts all the elements in suffix into
    prefix producing all the permutations of suffix.
    Prints all permutations in lexicographical order.
    '''
```

```
    suffix_size = len(suffix)
    if suffix_size == 0:  #  Have we considered all the elements?
        pass # print('>>>', prefix, '<<<')
    else:
        for i in range(0, suffix_size):
            new_pre = prefix + [suffix[i]]
            new_suff = suffix[:i] + suffix[i + 1:]
            tab(depth)
            print(new_pre, new_suff, sep=':')
            permute(new_pre, new_suff, depth + 1)
```

This version of permute includes printing statements that reveal the algorithm's process. The tab function,

```
def tab(n):
    for i in range(n):
        print(end='    ')
```

indents the output in proportion to the depth of the recursion. Notice that this version of permute accepts an additional parameter named depth. This parameter represents the depth of the recursion. The first few lines of output produced by the call

```
permute([], [1, 2, 3, 4], 0)
```

are:

```
[1]:[2, 3, 4]
   [1, 2]:[3, 4]
      [1, 2, 3]:[4]
         [1, 2, 3, 4]:[]
      [1, 2, 4]:[3]
         [1, 2, 4, 3]:[]
   [1, 3]:[2, 4]
      [1, 3, 2]:[4]
         [1, 3, 2, 4]:[]
      [1, 3, 4]:[2]
         [1, 3, 4, 2]:[]
   [1, 4]:[2, 3]
      [1, 4, 2]:[3]
         [1, 4, 2, 3]:[]
      [1, 4, 3]:[2]
         [1, 4, 3, 2]:[]
[2]:[1, 3, 4]
   [2, 1]:[3, 4]
      [2, 1, 3]:[4]
         [2, 1, 3, 4]:[]
      [2, 1, 4]:[3]
         [2, 1, 4, 3]:[]
```

(The complete output has more lines.) The initial depth is zero, and each recursive calls passes a depth parameter that is one more than the current depth. A greater indentation in an output line indicates a deeper

level of recursion. Notice that the recursion stops (indicated by the indentation going no deeper) when the suffix is empty.

While Listing 10.6 (`listpermutations.py`) is a good exercise in recursive list processing, the Python standard library provides a function named `permutations` in the `itertools` module that allows us to generate permutations with very little code. Listing 10.7 (`stdpermutations.py`) produces the same orderings as Listing 10.6 (`listpermutations.py`), but it produces the orderings in tuples instead of lists.

Listing 10.7: `stdpermutations.py`

```
1  #  Use the standard permutations function to list
2  #  the possible arrangements of elements in a list.
3
4  from itertools import permutations
5
6  def main():
7      a = [1, 2, 3, 4]
8      for p in permutations(a):
9          print(p)
10
11 main()
```

10.5 Randomly Permuting a List

Section 10.4 showed how we can generate all the permutations of a list in an orderly fashion. Often, however, we need to produce one those permutations chosen at random. For example, we may need to randomly rearrange the contents of an ordered list so that we can test a sort function to see if it will produce the original list. We could generate all the permutations, put each one in a list, and select a permutation at random from that list. This approach is inefficient, especially as the length of the list to permute grows larger. Fortunately, we can randomly permute the contents of a list easily and quickly. Listing 10.8 (`randompermute.py`) contains a function named `permute` that randomly permutes the elements of a list.

Listing 10.8: `randompermute.py`

```
1  from random import randrange
2
3  def permute(lst):
4      '''
5      Randomly permutes the contents of list lst
6      '''
7      n = len(lst)
8      for i in range(n - 1):
9          pos = randrange(i, n)     #  i <= pos < n
10         lst[i], lst[pos] = lst[pos], lst[i]
11
12 def main():
13     '''
14     Tests the permute function that randomly permutes the
15     contents of a list
16     '''
17     a = [1, 2, 3, 4, 5, 6, 7, 8]
18     print('Before:', a)
19     permute(a)
```

```
20      print('After :', a)
21
22  main()
```

Notice that the `permute` function in Listing 10.8 (`randompermute.py`) uses a simple un-nested loop and no recursion. The `permute` function varies the `i` index variable from 0 to the index of the next to last element in the list. An index greater than `i` is chosen pseudorandomly using `randrange` (see Section 6.4), and the elements at position `i` and the random position are exchanged. At this point all the elements at position `i` and smaller are fixed and will not change as the function's execution continues. The index `i` is incremented, and the process continues until all the `i` values have been considered.

Two be correct, our `permute` function must be able to generate any valid permutation of the list. It is important that our `permute` function is able produce all possible permutations with equal probability; said another way, we do not want our `permute` function to generate some permutations more often than others. The `permute` function in Listing 10.8 (`randompermute.py`) is fine, but consider a slight variation of the algorithm:

```python
def faulty_permute(lst):
    '''
    An attempt to randomly permute the contents of list lst
    '''
    n = len(lst)
    for i in range(n - 1):
        pos = randrange(0, n)     #  0 <= pos < n
        lst[i], lst[pos] = lst[pos], lst[i]
```

Do you see the difference between `faulty_permute` and `permute`? In `faulty_permute`, the random index is chosen from all valid list indices, whereas `permute` restricts the random index to valid indices greater than or equal to `i`. This means that any element within `lst` can be exchanged with the element at position `i` during any loop iteration. While this approach may superficially appear to be just as good as `permute`, it in fact produces an uneven distribution of permutations. Listing 10.9 (`comparepermutations.py`) exercises each `permutation` function 1,000,000 times on the list [1, 2, 3] and tallies each permutation. There are exactly six possible permutations of this three-element list.

Listing 10.9: `comparepermutations.py`

```python
from random import randrange

#  Randomly permute a list
def permute(lst):
    '''
    Randomly permutes the contents of list lst
    '''
    n = len(lst)
    for i in range(n - 1):
        pos = randrange(i, n)     #  i <= pos < n
        lst[i], lst[pos] = lst[pos], lst[i]

#  Randomly permute a list?
def faulty_permute(lst):
    '''
    An attempt to randomly permute the contents of list lst
    '''
    n = len(lst)
```

```python
    for i in range(n - 1):
        pos = randrange(0, n)    #  0 <= pos < n
        lst[i], lst[pos] = lst[pos], lst[i]


def classify(a):
    '''
    Classify a list as one of the six permutations
    '''
    sum = 100*a[0] + 10*a[1] + a[2]
    if sum == 123:    return 0
    elif sum == 132: return 1
    elif sum == 213: return 2
    elif sum == 231: return 3
    elif sum == 312: return 4
    elif sum == 321: return 5
    else: return -1

def report(a):
    '''
    Report the accumulated statistics
    '''
    print("1,2,3: ", a[0])
    print("1,3,2: ", a[1])
    print("2,1,3: ", a[2])
    print("2,3,1: ", a[3])
    print("3,1,2: ", a[4])
    print("3,2,1: ", a[5])


def run_test(perm, runs):
    '''
    Uses a permutation function to generate the permutations
    of the list [1,2,3]
    perm: the permutation function to test
    runs: the number permutations to perform
    '''
    #  The list to permute
    original = [1, 2, 3]

    # permutation_tally list keeps track of each permutation pattern
    # permutation_tally[0] counts {1,2,3}
    # permutation_tally[1] counts {1,3,2}
    # permutation_tally[2] counts {2,1,3}
    # permutation_tally[3] counts {2,3,1}
    # permutation_tally[4] counts {3,1,2}
    # permutation_tally[5] counts {3,2,1}
    permutation_tally = 6 * [0]   #  Clear all the counters
    for i in range(runs):     #  Run runs times
        #  working holds a copy of original is gets permuted and tallied
        working = original[:]
        #  Permute the list with the permutation algorithm
        perm(working)
        #  Count this permutation
        permutation_tally[classify(working)] += 1
```

```
74      report(permutation_tally)     #  Report results
75
76
77  def main():
78      #  Each test performs one million permutations
79      runs = 1000000
80
81      print("--- Random permute #1 -----")
82      run_test(permute, runs)
83
84      print("--- Random permute #2 -----")
85      run_test(faulty_permute, runs)
86
87
88  main()
```

In Listing 10.9 (comparepermutations.py)'s output, permute #1 corresponds to our original permute function, and permute #2 is the faulty_permute function. The output of Listing 10.9 (comparepermutations.py) reveals that the faulty permutation function favors some permutations over others:

```
--- Random permute #1 -----
1,2,3:  166576
1,3,2:  167372
2,1,3:  166117
2,3,1:  166797
3,1,2:  166925
3,2,1:  166213
--- Random permute #2 -----
1,2,3:  222789
1,3,2:  111010
2,1,3:  222458
2,3,1:  221987
3,1,2:  110690
3,2,1:  111066
```

In one million runs, the permute function provides an even distribution of the six possible permutations of [1, 2, 3]. The faulty_permute function generates the permutations [1, 2, 3], [2, 1, 3, and [2, 3, 1] twice as many times as the permutations [1, 3, 2], [3, 1, 2], and [3, 2, 1].

To see why faulty_permute misbehaves, we need to examine all the permutations it can produce during one call. Figure 10.5 shows a hierarchical structure that maps out how faulty_permute transforms its list parameter each time through the for loop. The top of the tree shows the original list, [1, 2, 3]. The second row shows the three possible resulting lists after the first iteration of the for loop. The leftmost list represents the element at index zero swapped with the element at index zero (effectively no change). The second list on the second row represents the interchange of the elements at index 0 and index 1. The third list on the second row results from the interchange of the elements at positions 0 and 2. The underlined elements represent the elements most recently swapped. If only one item in the list is underlined, the function merely swapped the item with itself.
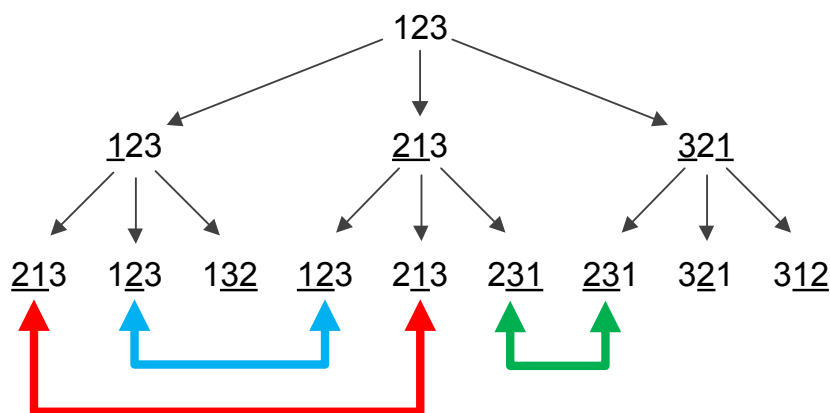
Figure 10.5: A tree mapping out the ways in which `faulty_permute` can transform the list [1, 2, 3] at each iteration of its `for` loop

As Figure 10.5 shows, the lists `[1, 2, 3]`, `[2, 1, 3`, and `[2, 3, 1]` each appear twice in the last row, while `[1, 3, 2]`, `[3, 1, 2]`, and `[3, 2, 1]` each appear only once. This means, for example, that the function is twice as likely to produce `[1, 2, 3]` as `[1, 3, 2]`.
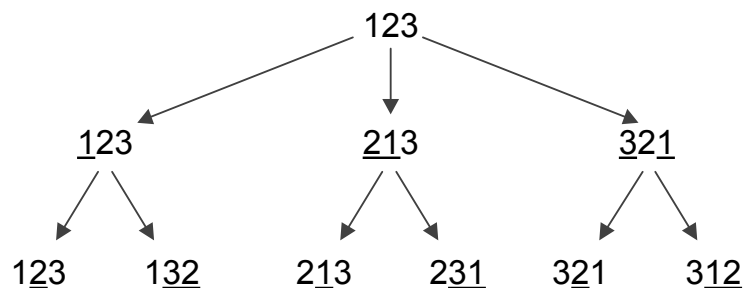

Figure 10.6: A tree mapping out the ways in which `permute` can transform the list [1, 2, 3] at each iteration of its `for` loop

Compare Figure 10.5 to Figure 10.6. The second row of the tree for `permute` is identical to the second row of the tree for `faulty_permute`, but the third rows are different. The second time through its loop the `permute` function does not attempt to exchange the element at index zero with any other elements. We see that none of the first elements in the lists in row three are underlined. The third row contains exactly one instance of each of the possible permutations of `[1, 2, 3]`. This means that the correct `permute` function is not biased towards any of the individual permutations, and so the function can generate all the permutations with equal probability.

10.6 Reversing a List

Listing 10.10 (`listreverse.py`) contains a recursive function named `rev` that accepts a list as a parameter and returns a new list with all the elements of the original list in reverse order.

Listing 10.10: `listreverse.py`

```
1  def rev(lst):
2      return [] if len(lst) == 0 else rev(lst[1:]) + lst[0:1]
3
4  print(rev([1, 2, 3, 4, 5, 6, 7]))
```

Python has a standard function, `reversed`, that accepts a list parameter. The `reversed` function does not return a list but instead returns an iterable object that can be used like the `range` function within a `for` loop (see Section 5.3). Listing 10.11 (`reversed.py`) shows how `reversed` can be used to print the contents of a list backwards.

Listing 10.11: `reversed.py`

```
1  for item in reversed([1, 2, 3, 4, 5, 6, 7]):
2      print(item)
```

In Section 11.3 we will see how to reverse the elements in a list using a special funtion-like object called a *method*.

10.7 Summary

- Various algorithms exist for sorting lists. Selection sort is a simple algorithm for sorting a list.

- A list formal parameter aliases the actual parameter passed by the client. This means any modifications a function makes to the contents of the list will affect the client's own list. This concept allows a sort or permutation routine to physically rearrange the elements in a list for the client's benefit.

- Linear search is useful for finding elements in an unordered list. Binary search can be used on ordered lists, and due to the nature of its algorithm, binary search is very fast, even on large lists.

- A permutation of a list is a reordering of its elements.

- Care must be taken when producing a random permutation of a list to ensure all the possible outcomes are equally likely.

10.8 Exercises

1. Complete the following function that reorders the contents of a list so they are reversed from their original order. For example, a list containing the elements 2, 6, 2, 5, 0, 1, 2, 3 would be transformed into 3, 2, 1, 0, 5, 2, 6, 2. Note that your function must physically rearrange the elements within the list, not just print the elements in reverse order.

```
def reverse(lst):
    #  Add your code...
```

2. Complete the following function that reorders the contents of a list of integers so that all the even numbers appear before any odd number. The even values are sorted in ascending order with respect to themselves, and the odd numbers that follow are also sorted in ascending order with respect to themselves. For example, a list containing the elements 2, 1, 10, 4, 3, 6, 7, 9, 8, 5 would be transformed into 2, 4, 6, 8, 10, 1, 3, 5, 7, 9 Note that your function must physically rearrange the elements within the list, not just print the elements in the desired order.

```python
def special_sort(lst):
    #  Add your code...
```

3. Create a special comparison function to be passed to our flexible selection sort function. The special comparison function should enable the sort function to arrange the elements of a list in the order specified in Exercise 2.

4. Complete the following function that filters negative elements out of a list. The function returns the filtered list and the original list is unchanged. For example, if a list containing the elements 2, −16, 2, −5, 0, 1, −2, −3 is passed to the function, the function would return the list containing 2, 2, 0, 1. Note the original ordering of the non-negative values is unchanged in the result.

```python
def filter(a):
    #  Add your code...
```

5. Complete the following function that shifts all the elements of a list backward one place. The last element that gets shifted off the back end of the list is copied into the first (0th) position. For example, if a list containing the elements 2, 1, 10, 4, 3, 6, 7, 9, 8, 5 is passed to the function, it would be transformed into 5, 2, 1, 10, 4, 3, 6, 7, 9, 8 Note that your function must physically rearrange the elements within the list, not just print the elements in the shifted order.

```python
def rotate(lst):
    #  Add your code...
```

6. Complete the following function that determines if the number of even and odd values in an integer list is the same. The function would return true if the list contains 5, 1, 0, 2 (two evens and two odds), but it would return false for the list containing 5, 1, 0, 2, 11 (too many odds). The function should return true if the list is empty, since an empty list contains the same number of evens and odds (0 for both). The function does not affect the contents of the list.

```python
def balanced(a):
    #  Add your code...
```

7. Complete the following function that returns true if a list lst contains duplicate elements; it returns false if all the elements in lst are unique. For example, the list [2, 3, 2, 1, 9] contains duplicates (2 appears more than once), but the list [2, 1, 0, 3, 8, 4] does not (none of the elements appear more than once).

An empty list has no duplicates. The function does not affect the contents of the list.

```python
def has_duplicates(lst):
    #  Add your code...
```

8. Can linear search be used on an unsorted list? Why or why not?

9. Can binary search be used on an unsorted list? Why or why not?

10. How many different orderings are there for the list `[4, 3, 8, 1, 10]`?

11. Complete the following function that determines if two lists contain the same elements, but not necessarily in the same order. The function would return true if the first list contains 5, 1, 0, 2 and the second list contains 0, 5, 2, 1. The function would return false if one list contains elements the other does not or if the number of elements differ. This function could be used to determine if one list is a permutation of another list. The function does not affect the contents of either list.

```python
def is_permutation(a, b):
    #  Add your code...
```

Chapter 11

Objects

In the hardware arena, a personal computer is built by assembling

- a motherboard (a circuit board containing sockets for a microprocessor and assorted support chips),

- a processor and its various support chips,

- memory boards,

- a video card,

- an input/output card (USB ports, parallel port, and mouse port),

- a disk controller,

- a disk drive,

- a case,

- a keyboard,

- a mouse, and

- a monitor.

(Some of these components like the I/O, disk controller, and video may be integrated with the motherboard.)

The video card is itself a sophisticated piece of hardware containing a video processor chip, memory, and other electronic components. A technician does not need to assemble the card; the card is used as is off the shelf. The video card provides a substantial amount of functionality in a standard package. One video card can be replaced with another card from a different vendor or with another card with different capabilities. The overall computer will work with either card (subject to availability of drivers for the operating system), because standard interfaces allow the components to work together.

Software development today is increasingly *component based*. Software components are used like hardware components. A software system can be built largely by assembling pre-existing software building blocks. Python supports various kinds of software building blocks. The simplest of these is the *function* that we investigated in Chapter 6 and Chapter 7. A more powerful technique uses software *objects*.

Python is *object oriented*. Most modern programming languages support object-oriented (OO) development to one degree or another. An OO programming language allows the programmer to define, create, and manipulate objects. Objects bundle together data and functions. Like other variables, each Python object has a type, or *class*. The terms *class* and *type* are synonymous.

In this chapter we explore some of the classes available in the Python standard library.

11.1 Using Objects

An object is an instance of a class. We have been using objects since the beginning, but we have not taken advantage of all the capabilities that objects provide. Integers, floating-point numbers, strings, lists, and functions are all objects in Python. With the exception of function objects, we have treated these objects as passive data. We can assign an integer and use its value. We can add two floating-point numbers and concatenate two strings with the + operator. We can pass objects to functions and functions can return objects.

Objects fuse data and functions together. A typical object consists of two parts: *data* and *methods*. An object's data is sometimes called its *attributes* or *fields*. Methods are like functions, and they also are known as *operations*. The data and methods of an object constitutes its *members*. Using the same terminology as functions, the code that uses an object is called the object's *client*. Just as a function provides a service to its client, an object provides a service to its client. The services provided by an object can be more elaborate that those provided by simple functions, because objects make it easy to store persistent data.

The assignment statement

x = 2

binds the variable x to an integer object with the value of 2. The name of the class of x is int. To see some of the capabilities of int objects, issue the command dir(x) or dir(int) in the Python interpreter:

```
>>> dir(x)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__',
'__class__', '__delattr__', '__divmod__', '__doc__', '__eq__',
'__float__', '__floor__', '__floordiv__', '__format__',
'__ge__', '__getattribute__', '__getnewargs__', '__gt__',
'__hash__', '__index__', '__init__', '__int__', '__invert__',
'__le__', '__lshift__', '__lt__', '__mod__', '__mul__',
'__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__',
'__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__',
'__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__',
'__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__',
'__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__',
'__xor__', 'bit_length', 'conjugate', 'denominator', 'from_bytes',
'imag', 'numerator', 'real', 'to_bytes']
```

The dir function, which is available to Python programs as well, lists the members of the class (or an object's class, if called with an object argument). Most of these names are methods and are not meant for clients to use directly. Member names that begin and end with two underscores are supposed to be reserved

for the object's own internal use, but we can experiment to see how methods work. Many of these methods are mapped to Python operators.

`__add__` is a method in the `int` class, so it is available to all integer objects. The expression `x.__add__(3)` is an example of a *method invocation*. A method invocation works like a function invocation, except we must qualify the call with an object's name (or sometimes a class name). The expression begins with the object's name, followed by a dot (`.`), and then the method name with any necessary parameters. The following interactive sequence shows how we can use the `__add__` method:

```
>>> x = 2
>>> x
2
>>> x + 3
5
>>> x.__add__(3)
5
>>> int.__add__(x, 3)
5
```

Notice that `x + 3`, `x.__add__(3)` and `int.__add__(x, 3)` all produce identical results. In the expression `x.__add__(3)` the interpreter knows that `x` is an `int`, so it calls the `__add__` method of the `int` class passing both `x` and 3 as arguments. The expression `int.__add__(x, 3)` best represents the process the interpreter uses to execute the method. The `int` class defines the `__add__` method, and the expression `int.__add__(x, 3)` indicates the `__add__` method requires both an object (`x`) and an integer (3) to do its job. The interpreter translates the expressions `x + 3` and `x.__add__(3)` into the call `int.__add__(x, 3)`. When we use the expression `x + 3` we are oblivious to details of the `__add__` method in the `int` class.

Compare the code fragment

```
s = "ABC"
print(s.__add__("DEF"))
print(str.__add__(s, "DEF"))
```

The expressions `s.__add__("DEF")` and `str.__add__(s, "DEF")` are equivalent to `s + "DEF"`, which we know is string concatenation. The interpreter translates the symbol for integer addition or string concatenation, `+`, into the appropriate method call, in this case `str.__add__`.

Clients are not meant to call directly methods that begin with two underscores (`__`). The Python language maps the binary `+` operator to the `__add__` method of the appropriate class. Most of the integer methods correspond to arithmetic operators that are easier to use; for examples, `__gt__` for `>` and `__mul__` for `*`. The `int` class does not offer too many other methods that we need to use right now. Other Python classes like `str`, `list`, and `Random` do provide methods intended for clients to use.

11.2 String Objects

Strings are like lists is some ways because they contain an ordered sequence of elements. Strings are distinguished from lists in three key ways:

- Strings must contain only characters, while lists may contain objects of any type.

- Strings are immutable. The contents of a string object may not be changed. Lists are mutable objects.

- If two strings are equal with == comparison, they automatically are aliases (equal with the `is` operator). This means two identical string literals that appear in the Python source code refer to the same string object.

Consider Listing 11.1 (`stringalias.py`).

Listing 11.1: `stringalias.py`

```
word1 = 'Wow'
word2 = 'Wow'
print('Equality:', word1 == word2, '  Alias:', word1 is word2)
```

Listing 11.1 (`stringalias.py`) assigns `word1` and `word2` to two distinct string literals. Since the two string literals contain exactly the same characters, the interpreter creates only one string object. The two variables `word1` and `word2` are bound to the same object. We say the interpreter *merges* the two strings. Since in some programs strings may be long, string merging can save space in the computer's memory.

Objects bundle data and functions together. The data that comprise strings consist of the characters that make up the string. Any string object also has available a number of methods. Listing 11.2 (`stringupper.py`) shows how a programmer can use the `upper` method available to string objects.

Listing 11.2: `stringupper.py`

```
name = input("Please enter your name: ")
print("Hello " + name.upper() + ", how are you?")
```

Listing 11.2 (`stringupper.py`) capitalizes (converts to uppercase) all the letters in the string the user enters:

```
Please enter your name: Rick
Hello RICK, how are you?
```

The expression

```
name.upper()
```

within the `print` statement represents a *method call*. The general form of a method call is

$$object.methodname\ (\ parameterlist\ )$$

- *object* is an expression that represents object. In the example in Listing 11.2 (`stringupper.py`), `name` is a reference to a string object.

- The period, pronounced *dot*, associates an object expression with the method to be called.

- *methodname* is the name of the method to execute.

- The *parameterlist* is comma-separated list of parameters to the method. For some methods the parameter list may be empty, but the parentheses always are required.

Except for the object prefix, a method works just like a function. The upper method returns a string. A method may accept parameters. Listing 11.3 (rjustprog.py), uses the rjust string method to right justify a string padded with a specified character.

Listing 11.3: `rjustprog.py`

```
word = "ABCD"
print(word.rjust(10, "*"))
print(word.rjust(3, "*"))
print(word.rjust(15, ">"))
print(word.rjust(10))
```

The output of Listing 11.3 (rjustprog.py):

```
******ABCD
ABCD
>>>>>>>>>>>ABCD
      ABCD
```

shows

- word.rjust(10, "*") right justifies the string "ABCD" within a 10-character field padded with * characters.

- word.rjust(3, "*") does not return a different string from the original "ABCD" since the specified width (3) is less than or equal to the length of the original string (4).

- word.rjust(10) shows that the default padding character is a space.

str Methods
`upper`
Returns a copy of the original string with all the characters converted to uppercase
`lower`
Returns a copy of the original string with all the characters converted to lower case
`rjust`
Returns a string right justified within a field padded with a specified character which defaults to a space
`ljust`
Returns a string left justified within a field padded with a specified character which defaults to a space
`center`
Returns a copy of the string centered within a string of a given width and optional fill characters; fill characters default to spaces
`strip`
Returns a copy of the given string with the leading and trailing whitespace removed; if provided an optional string, the `strip` function strips leading and trailing characters found in the parameter string
`startswith`
Determines if the string is a prefix of the string
`endswith`
Determines if the string is a suffix of the string
`count`
Determines the number times the string parameter is found as a substring; the count includes only non-overlapping occurrences
`find`
Returns the lowest index where the string parameter is found as a substring; returns -1 if the parameter is not a substring
`format`
Embeds formatted values in a string using 1, 2, etc. position parameters (see Listing 11.4 (`stripandcount.py`) for an example) parameter is found as a substring; returns -1 if the parameter is not a substring

Table 11.1: A few of the methods available to `str` objects

Listing 11.4 (`stripandcount.py`) demonstrates two of the string methods.

Listing 11.4: `stripandcount.py`

```
# Strip leading and trailing whitespace and count substrings
s = "     ABCDEFGHBCDIJKLMNOPQRSBCDTUVWXYZ       "
print("[", s, "]", sep="")
s = s.strip()
print("[", s, "]", sep="")

# Count occurrences of the substring "BCD"
print(s.count("BCD"))
```

Listing 11.4 (`stripandcount.py`) displays:

```
[     ABCDEFGHBCDIJKLMNOPQRSBCDTUVWXYZ     ]
[ABCDEFGHBCDIJKLMNOPQRSBCDTUVWXYZ]
3
```

The `[]` index operator applies to strings as it does lists. The `len` function returns the number of characters in a string. Listing 11.5 (`printcharacters.py`) prints the individual characters that make up a string.

Listing 11.5: `printcharacters.py`

```python
s = "ABCDEFGHIJK"
print(s)
for i in range(len(s)):
    print("[", s[i], "]", sep="", end="")
print()  # Print newline

for ch in s:
    print("<", ch, ">", sep="", end="")
print()  # Print newline
```

The expression

```python
s[i]
```

actually uses the string method `__getitem__`:

```python
s.__getitem__(i)
```

The global function `len` calls the string object's `__len__` method:

```python
s = "ABCDEFGHIJK"
print(len(s) == s.__len__())  # Prints True
```

As Listing 11.5 (`printcharacters.py`) shows, strings may be manipulated in ways similar to lists. Strings may be sliced:

```python
print("ABCDEFGHIJKL"[2:6])  # Prints CDEF
```

Since strings are immutable objects, element assignment and slice assignment is not possible:

```python
s = "ABCDEFGHIJKLMN"
s[3] = "S"      # Illegal, strings are immutable
s[3:7] = "XYX"  # Illegal, strings are immutable
```

String immutability means the `strip` method may not change a given string:

```python
s = "    ABC    "
s.strip()    # s is unchanged
print("<" + s + ">")  # Prints <    ABC    >, not <ABC>
```

In order to strip the leading and trailing whitespace as far as the string bound to the variable `s` is concerned, we must reassign `s`:

```
s = "    ABC    "
s = s.strip()      #  Note the reassignment
print("<" + s + ">")      #  Prints <ABC>
```

The `strip` method returns a new string; the string on whose behalf `strip` is called is not modified. Clients must as in this example rebind their variable to the string passed back by `strip`.

11.3 List Objects

We introduced lists in Chapter 9, but there we treated them merely as enhanced data objects. We assigned lists, passed lists to functions, returned lists from functions, and interacted with the elements of lists. List objects provide more capability than we revealed earlier.

All Python lists are instances of the `list` class. Table **??** lists some of the methods available to `list` objects.

`list` Methods
`count`
Returns the number of times a given element appears in the list. Does not modify the list.
`insert`
Inserts a new element before the element at a given index. Increases the length of the list by one. Modifies the list.
`append`
Adds a new element to the end of the list. Modifies the list.
`index`
Returns the lowest index of a given element within the list. Produces an error if the element does not appear in the list. Does not modify the list.
`remove`
Removes the first occurrence (lowest index) of a given element from the list. Produces an error if the element is not found. Modifies the list if the item to remove is in the list.
`reverse`
Physically reverses the elements in the list. The list is modified.
`sort`
Sorts the elements of the list in ascending order. The list is modified.

Table 11.2: A few of the methods available to `list` objects

Since lists are mutable data structures, the `list` class has both __getitem__ and __setitem__ methods. The statement

```
x = lst[2]
```

behind the scenes becomes the method call

```
x = list.__getitem__(lst, 2)
```

and the statement

```
lst[2] = x
```

maps to

```
list.__setitem__(lst, 2, x)
```

The str class does not have a __setitem__ method, since strings are immutable.

The code

```
lst = ["one", "two", "three"]
lst += ["four"]
```

is equivalent to

```
lst = ["one", "two", "three"]
lst.append("four")
```

but the version using append is more efficient.

11.4 Summary

- An object is an instance of a class.

- The terms *class* and *type* are synonymous.

- Integers, floating-point numbers, strings, lists, and functions are examples of objects we have seen in earlier chapters.

- Typically objects are a combination of data (attributes or fields) and methods (operations)

- An object's data and methods constitute its members.

- The code that uses the services provided by an object is known as the client of the object.

- Methods are like functions associated with a class of objects.

- Members that begin and end with two underscores __ are meant for internal use by objects; clients usually do not use these members directly.

- Methods are called (or invoked) on behalf of objects or classes.

- The dot (.) operator associates an object or class with a member.

- Clients make not call a method without its associated object or class.

- The str class represents string objects.

- String objects are immutable. You may reassign a variable to another string object, but you may not modify the contents of an existing string object. This means no str method may alter an existing string object. When client code wishes to achieve the effect of modifying a string via one of the string's methods, the client code must reassign its variable with the result passed back by the method.

- The str class contains a number methods useful for manipulating strings.

- The list class represents all list objects.

- Unlike strings, list objects are mutable. The contents of a list object may be changed, removed, or inserted.

11.5 Exercises

1. Add exercises

Chapter 12

Custom Types

Consider the task of writing a program that manages accounts for a bank. A bank account has a number of attributes:

- Every account has a unique identifier, the account number.

- Every account has an owner that can be identified by a social security number.

- Each account's owner has a name.

- Each account has a current balance.

- Each account is either active or inactive.

- Each account may have additional restrictions such as a minimum balance to remain active.

- Each account may be marked closed, meaning it will never be used again but by law information about the account must be retained for some period of time.

The list of attributes easily could be much longer.

Based on our programming experience to this point, we conclude that the information pertinent to accounts must be stored in variables. The situation gets messy when we consider that our program must be able to process thousands of accounts. The data could be stored in a list, but would we have a list of account numbers and a separate list for the customers' social security numbers? Would we need to have a separate list for every piece that makes up a bank account?

While it is possible to maintain separate lists and coordinate them somehow, it is more natural to think of having a list of *accounts*, where each account contains all the necessary attributes. Python *objects* allow us to model accounts in this more natural way.

12.1   Geometric Points

As an example to introduce simple objects, consider two-dimensional geometric points from mathematics. We consider a single point object to consist of two real number coordinates: x and y. We ordinarily represent a point by an ordered pair (x, y). In a program, we could model the point $(2.5, 1)$ as a list:

```
point = [2.5, 6]
print("In", point, "the x coordinate is", point[0])
```

or as a tuple:

```
point = 2.5, 6
print("In", point, "the x coordinate is", point[0])
```

In either case, we must remember that the element at index 0 is the *x* coordinate and the element at index 1 is the *y*. While this is not an overwhelming burden, it would be better if we could access the parts of a point through the labels *x* and *y* instead of numbers. Lists and tuples have another problem—the programmer must take care to avoid an invalid index. This can happen accidentally with a simple typographical error or when variables and expressions are used in the square brackets.

Python provides the `class` reserved word to allow the creation of new types of objects. We can create a new type, `Point`, as follows:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

This code defines a new type. This `Point` class contains a single method named __init__. This special method is known as a *constructor*, or *initializer*. The constructor code executes when the client creates an object. The first parameter of this constructor, named `self`, is a reference to the object being created. The statement

```
self.x = x
```

within the constructor establishes a field named x in the newly created `Point` object. The expression `self.x` refers to the x field in the object, and the x variable on the right side of the assignment operator refers to the parameter named x. These two x names represent different variables.

Once this new type has been defined in such a class definition, a client may create and use variables of the type `Point`:

```
#  Client code
pt = Point(2.5, 6)      #  Make a new Point object
print("(", pt.x, ",", pt.y, ")", sep="")
```

The expression `Point(2.5, 6)` creates a new `Point` object with an *x* coordinate of 2.5 and a *y* coordinate of 6. The expression `pt.x` refers to the *x* coordinate of the `Point` object named `pt`. Unlike with a list or a tuple, you do not use a numeric index to refer to a component of the object; instead you use the name of the field (like x and y) to access a part of an object.

Figure 12.1 provides a conceptual view of a point object.

A definition of the form

```
class MyName:
    #  Block of method definitions
```

creates a programmer-defined type. Once the definition is available to the interpreter, programmers can define and use variables of this custom type.
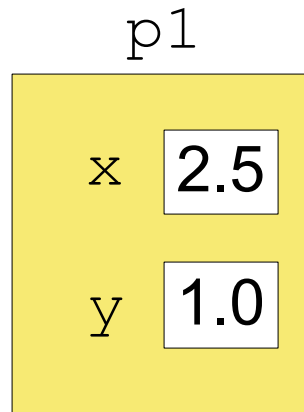
Figure 12.1: A Point object

A component data element of an object is called a *field*. Our `Point` objects have two fields, `x` and `y`. The terms *instance variable* or *attribute* sometimes are used in place of field. As with methods, Python uses the dot (`.`) notation to access a field of an object; thus,

```
pt.x = 0
```

assigns zero to the `x` field of point `pt`.

Consider a simple employee record that consists of a name (string), an identification number (integer) and a pay rate (floating-point number). Such a record can be represented by the class

```
class EmployeeRecord:
    def __init__(n, i, r):
        name = n
        id = i
        pay_rate = r
```

Such an object could be created and used as

```
rec = EmployeeRecord("Mary", 2148, 10.50)
```

Listing 12.1 (`employee.py`) uses our `EmployeeRecord` class to implement a simple database of employee records.

Listing 12.1: `employee.py`

```
1  #  Information about one employee
2  class EmployeeRecord:
3      def __init__(self, n, i, r):
4          self.name = n
5          self.id = i
6          self.pay_rate = r
7
```

```python
8
9   def open_database(filename, db):
10      """
11      Read employee information from a given file and store it
12      in the given vector.
13      Returns true if the file could be read; otherwise,
14      it returns false.
15      """
16      # Open file to read
17      lines = open(filename)
18      for line in lines:
19          name, id, rate = eval(line)
20          db.append(EmployeeRecord(name, id, rate))
21      lines.close()
22      return True
23
24
25  def print_database(db):
26      """
27      Display the contents of the database
28      """
29      for rec in db:
30          print(str.format("{:>5}: {:<10} {:>6.2f}", \
31                  rec.id, rec.name, rec.pay_rate))
32
33
34  def less_than_by_name(e1, e2):
35      """
36      Returns true if e1's name is less than e2's
37      """
38      return e1.name < e2.name
39
40
41
42  def less_than_by_id(e1, e2):
43      """
44      Returns true if e1's name is less than e2's
45      """
46      return e1.id < e2.id
47
48
49  def less_than_by_pay(e1, e2):
50      """
51      Returns true if e1's name is less than e2's
52      """
53      return e1.pay_rate < e2.pay_rate
54
55
56  def sort(db, comp):
57      """
58      Sort the database object db ordered by the given comp function.
59      """
60      n = len(db)
61      for i in range(n - 1):
62          smallest = i;
```

```
63              for j in range(i + 1, n):
64                  if comp(db[j], db[smallest]):
65                      smallest = j
66              if smallest != i:
67                  db[i], db[smallest] = db[smallest], db[i]
68
69
70  def main():
71      #  Simple "database" of employees
72      database = []
73
74      #  Open file to read
75      if open_database("data.dat", database):
76          #  Print the contents of the database
77          print("---- Unsorted:")
78          print_database(database)
79
80          #  Sort by name
81          sort(database, less_than_by_name)
82          print("---- Name order:")
83          print_database(database)
84
85          #  Sort by ID
86          sort(database, less_than_by_id)
87          print("---- ID order:")
88          print_database(database)
89
90          #  Sort by pay rate
91          sort(database, less_than_by_pay)
92          print("---- Pay order:")
93          print_database(database)
94      else:   #  Error, could not open file
95          print("Could not open database file")
96
97  main()
```

Given a text file named data.dat containing the data

```
'Fred',     324, 10.50
'Wilma',    371, 12.19
'Betty',    129, 15.45
'Barney',   120, 16.00
'Pebbles',  412,  9.34
'Bam-Bam',  420,  9.15
'George',  1038, 19.86
'Jane',     966, 19.86
'Judy',    1210, 15.61
'Elroy',   1300, 14.32
```

the program Listing 12.1 (employee.py) would produce the output

```
---- Unsorted:
  324: Fred         10.50
  371: Wilma        12.19
  129: Betty        15.45
  120: Barney       16.00
  412: Pebbles       9.34
  420: Bam-Bam       9.15
 1038: George       19.86
  966: Jane         19.86
 1210: Judy         15.61
 1300: Elroy        14.32
---- Name order:
  420: Bam-Bam       9.15
  120: Barney       16.00
  129: Betty        15.45
 1300: Elroy        14.32
  324: Fred         10.50
 1038: George       19.86
  966: Jane         19.86
 1210: Judy         15.61
  412: Pebbles       9.34
  371: Wilma        12.19
---- ID order:
  120: Barney       16.00
  129: Betty        15.45
  324: Fred         10.50
  371: Wilma        12.19
  412: Pebbles       9.34
  420: Bam-Bam       9.15
  966: Jane         19.86
 1038: George       19.86
 1210: Judy         15.61
 1300: Elroy        14.32
---- Pay order:
  420: Bam-Bam       9.15
  412: Pebbles       9.34
  324: Fred         10.50
  371: Wilma        12.19
 1300: Elroy        14.32
  129: Betty        15.45
 1210: Judy         15.61
  120: Barney       16.00
  966: Jane         19.86
 1038: George       19.86
```

Listing 12.1 (`employee.py`) uses a list of `EmployeeRecord` objects to implement a simple database. The ordering imposed by the sort function is determined by the function passed as the second argument.

The code within the `print_database` function uses the `format` of the `str` class to beautify the output of the data within a record:

```
print(str.format("{:>5}: {:<10} {:>6.2f}", \
        rec.id, rec.name, rec.pay_rate))
```

The string `"{:>5}: {:<10} {:>6.2f}"` contains formatting control codes. Each cryptic expression within the curly braces `{}` is a placeholder for a value in the list that follows. The expression within the `{}` indicates how to format its associated parameter. The first placeholder, `{:>5}`. refers to the first argument that follows the formatting string, `rec.id`. `{:<10}` refers to `rec.name`, and `{:>6.2f}` refers to `rec.pay_rate`. The colon (`:`) within the placeholder introduces the formatting code. `<` means left justify, and `>` specifies right justification. The numbers indicate field width; that is, the number of spaces allotted for the value to print. The `.2f` suffix will format a floating-point number with two explicit decimal places.

Our motivation at the beginning of the chapter was the need to build a database of bank account objects. The `class`

```
class BankAccount:
    def __init__(self):
        self.account_number = 0     #  Account number
        self.ssn = 123456789        #  Social security number
        self.name = ""              #  Customer name
        self.balance = 0.00         #  Funds available in the account
        self.min_balance = 100.00   #  Balance cannot fall below this amount
        self.active = False         #  Account is active or inactive
```

defines the structure of such account objects. Notice that the constructor of our `BankAccount` objects does not initialize any of the fields with client supplied values; instead, the constructor simply assigns default values to a new `BankAccount` objects. Clients later must assign proper values to a bank account object. A better definition would be

```
class BankAccount:
    def __init__(self, acct, ss, name, balance):
        self.account_number = acct  #  Account number
        self.ssn = ss               #  Social security number
        self.name = name            #  Customer name
        self.balance = balance      #  Funds available in the account
        self.min_balance = 100.00   #  Balance cannot fall below this amount
        self.active = False         #  Account is active or inactive
```

In this version the client can specify the account number, the customer's social security number and name, and the account's initial balance. The minimum balance and active flag are set to default values.

12.2 Methods

In modern object-oriented languages the power of objects comes from their ability to grant clients limited access. Some parts of an object are meant to be private, while other parts are meant to be public. This gives class designers the ability to hide the implementation details from clients. Knowledge of these details is not necessary for a client to use the objects in their recommended manner.

Suppose, for example, you wish to represent a mathematical rational number, or fraction. A **ratio**nal number is the **ratio** of two integers. There is a restriction, however—the number on the bottom of a fraction

cannot be zero. The number on the top of the fraction is called the numerator, and the bottom number is known as the denominator. A simple class such as

```python
class RationalNum:
    def __init__(self, num, den):
        self.numerator, self.denominator = num, den
```

There is nothing in this class definition that prevents a client from making a rational number like the following:

```python
fract = RationalNum(1, 0)
```

In this case the variable `fract` represents an undefined integer. We can help matters with a different constructor:

```python
class RationalNum:
    def __init__(self, num, den):
        self.numerator = num
        if den != 0:
            self.denominator = den
        else:
            print("Attempt to make an illegal rational number")
            from sys import exit
            exit(1)    #  Terminate program with an error code
```

While this new constructor will prevent illegal initialization, clients still can subvert our `RationalNum` objects:

```python
fract = RationalNum(1, 2)    #  This is OK
fract.denominator = 0        #  This is bad!
```

At best, the programmer made an honest mistake introducing an error into the program. Perhaps it was a careless "copy and paste" error. On the other hand, a clever programmer may be fully aware of how the program works in the larger context and intentionally write such bad code to exploit a weakness in the system that compromises its security.

Python uses a naming convention to protect a field. A field that with a name that begins with two underscores (__) is not accessible to clients using the normal dot operator.

Listing 12.2 (`rational.py`) uses protected fields.

Listing 12.2: `rational.py`

```python
 1  class Rational:
 2      """
 3      Represents a rational number (fraction)
 4      """
 5      def __init__(self, num, den):
 6          self.__numerator = num
 7          if den != 0:
 8              self.__denominator = den
 9          else:
10              print("Attempt to make an illegal rational number")
11              from sys import exit
12              exit(1)    #  Terminate program with an error code
13
```

```
14      def get_numerator(self):
15          """  Returns the numerator of the fraction.   """
16          return self.__numerator
17
18      def get_denominator(self):
19          """  Returns the denominator of the fraction.   """
20          return self.__denominator
21
22      def set_numerator(self, n):
23          """  Sets the numerator of the fraction to n.   """
24          self.__numerator = n
25
26      def set_denominator(self, d):
27          """
28          Sets the denominator of the fraction to d,
29          unless d is zero.  If d is zero, the method
30          terminates the program with an error meesage.
31          """
32          if d != 0:
33              self.__denominator = d
34          else:
35              print("Error: zero denominator!")
36              from sys import exit
37              exit(1)    #  Terminate program with an error code
38
39      def __str__(self):
40          """
41          Make a string representation of a Rational object
42          """
43          return str(self.get_numerator()) + "/" + str(self.get_denominator())
44
45  #  Client code that uses Rational objects
46  def main():
47      fract1 = Rational(1, 2)
48      fract2 = Rational(2, 3)
49      print("fract1 =", fract1)
50      print("fract2 =", fract2)
51      fract1.set_numerator(3)
52      fract1.set_denominator(4)
53      fract2.set_numerator(1)
54      fract2.set_denominator(10)
55      print("fract1 =", fract1)
56      print("fract2 =", fract2)
57
58  main()
```

Notice in Listing 12.2 (`rational.py`) in the `Rational` class that the field names begin with __. This means that client code like

```
fract = Rational(1, 2)
print(fract.__numerator)    //  Error, not possible
```

will not work. Clients no longer have direct access to the __numerator and __denominator fields of `Rational` objects.

Clients may appear to change a protected field as

```
fract = Rational(1, 2)
fract.__denominator = 0          #  Legal, but what does it do?
print(fract.get_denominator())   #  Prints 2, not 0
print(fract.__denominator)       #  Prints 0, not 2
```

Surprisingly, the second statement (assignment of `fract.__denominator`) does not affect the `__denominator` field used by the methods in the `Rational` class; it instead adds a new, unprotected field named `__denominator`. The client cannot get to the protected field by merely using the dot (`.`) operator. To avoid such confusion, a client should not attempt to use fields of an object with names that begin with two underscores.

The `__str__` method may be defined for any class. The interpreter calls an object's `__str__` method when a string representation of an object is required. For example, the `print` function converts an object into a string so it can display textual output.

In the `main` function of Listing 12.2 (`rational.py`) which contains code that uses `Rational` objects, the call

```
    fract1.set_numerator(2)
```

calls the `set_numerator` method of the `Rational` class on behalf of the object `fract1`. During the call `self` is assigned `fract1`, and `n` is assigned 2. This means the code within `set_numerator` assigns 2 to the parameter `n`, and the name `self.__numerator` within the method definition refers to `fract1`'s `__numerator` field. The method, therefore, reassigns the `__numerator` member of `fract1`.

In comparison, consider the call

```
    fract2.set_numerator(1)
```

This statement calls the `set_numerator` method of the `Rational` class on behalf of the object `fract2`. `self.__numerator` refers to `fract2`'s numerator, and parameter `n` is 1. This means the code within `set_numerator` assigns 1 to the parameter `n`, and thus the method assigns 1 to the `__numerator` field of the `fract2` object.

In OO-speak, we say the statement

```
  fract1.set_numerator(2)
```

represents the client sending a `set_numerator` *message* to object `fract1`. In this message, it provides the value 2. In this case `frac1` is the message *receiver*. In the statement

```
  fract2.set_numerator(1)
```

object `fract2` receives the `set_numerator` message with the value 1.

When the values of one or more instance variables in an object change, we say the object changes its state; for example, if we use an object to model the behavior of a traffic light, the object will contain some instance variable that represents its current color: red, yellow, or green. When that field changes, the traffic light's color is changed. In the green to yellow transition, we can say the light goes from the state of being green to the state of being yellow.

Armed with methods and protected fields, we can devise the starting point for a better bank account class:

```
class BankAccount:
    def __init__(self, number, ssn, name, balance):
```

```
        self.__account_number = number   #  Account number
        self.__ssn = ssn                 #  Social security number
        self.__name = name               #  Customer name
        self.__balance = balance         #  Funds available in the account
        self.__min_balance = 100         #  Balance cannot fall below this amount
        self.__active = True             #  Account is active or inactive

    def deposit(self, amount):
      """
      Add funds to the account, if possible
      Return true if successful, false otherwise
      """
        if self.is_active():
            self.__balance += amount
            return True #  Successful deposit
        return False    #  Unable to deposit into an inactive account

    def withdraw(self, amount)
        """
        Remove funds from the account, if possible
        Return true if successful, false otherwise
        """
        result = False;  #  Unsuccessful by default
        if self.is_active() and
             self.__balance - amount >= self.__min_balance ):
            self.__balance -= amount;
            result = True;  #  Success
        return result

    def set_active(self, act):
        """
        Activate or deactivate the account
        """
        self.__active = act

    bool is_active()
        """"
        Is the account active or inactive?
        """"
        return self.__active
```

Clients interact with these bank account objects via the methods; thus, it is only through methods that clients
may alter the state of a bank account object.

In the `BankAccount` methods

- Clients may add funds via the `deposit` method only if the account is active. Notice that the `deposit`
  method calls the `is_active` method using the parameter `self`. This means the receiver of the
  `is_active` message is the same receiver of the `deposit` call currently executing; for example, in
  the code

  ```
  acct = BankAccount(31243, 123456789, "Joe", 1000.00)
  ```

```
acct.deposit(100)
```

the `acct` object is the account object receiving the `deposit` message. Within that call to `deposit`, `acct` is the receiver of the `is_active` method call.

- The `withdraw` method prevents a client from withdrawing more money from an account than some specified minimum value. Withdrawals are not possible from an inactive account.

- The `set_active` method allows clients to activate and deactivate individual bank account objects.

- The `is_active` method allows clients to determine if an account object is currently active or inactive.

The following code will not work:

```
acct = BankAccount(31243, 123456789, "Joe", 1000.00)
acct.deposit(100)
acct.__balance -= 100;  #  Illegal
```

Clients instead must use the `withdraw` method. The `withdraw` method prevents actions such as

```
#  New bank account object with $1,000.00 balance
acct = BankAccount(31243, 123456789, "Joe", 1000.00)
acct.withdraw(2000.00); //  Method should disallow this operation
```

The operations of depositing and withdrawing funds are the responsibility of the object itself, not the client code. The attempt to withdraw the $2,000 dollars above could, for example, result in an error message.

Consider a non-programming example. If I deposit $1,000.00 dollars into a bank, the bank then has custody of my money. It is still my money, so I theoretically can reclaim it at any time. The bank stores money in its safe, and my money is in the safe as well. Suppose I wish to withdraw $100 dollars from my account. Since I have $1,000 total in my account, the transaction should be no problem. What is wrong with the following scenario:

1. Enter the bank.

2. Walk past the teller into a back room that provides access to the safe.

3. The door to the safe is open, so enter the safe and remove $100 from a stack of $20 bills.

4. Exit the safe and inform a teller that you got $100 out of your account.

5. Leave the bank.

This is not the process a normal bank uses to handle withdrawals. In a perfect world where everyone is honest and makes no mistakes, all is well. In reality, many customers might be dishonest and intentionally take more money than they report. Even though I faithfully counted out my funds, perhaps some of the bills were stuck to each other and I made an honest mistake by picking up six $20 bills instead of five. If I place the bills in my wallet with other money that already be present, I may never detect the error. Clearly a bank needs more controlled procedure for customer withdrawals.

When working with programming objects, in many situations it is better to restrict client access from the internals of an object. Client code should not be able to change directly bank account objects for various reasons, including:

- A withdrawal should not exceed the account balance.

- Federal laws dictate that deposits above a certain amount should be reported to the Internal Revenue Service, so a bank would not want customers to be able to add funds to an account in a way to circumvent this process.

- An account number should never change for a given account for the life of that account.

12.3 Custom Type Examples

This section contains a number of examples of code organization with functions.

12.3.1 Stopwatch

In 6.3 we saw how to use the `clock` function to measure elapsed time during a program's execution. The following skeleton code fragment

```python
seconds = clock()      #  Record starting time
#
#  Do something here that you wish to time
#
other = clock()        #  Record ending time
print(other − seconds, "seconds")
```

can be adapted to any program, but we can make it more convenient if we wrap the functionality into an object. We can wrap all the messy details of the timing code into a convenient package. Consider the following client code that uses an object to keep track of the time:

```python
timer = Stopwatch()  #  Declare a stopwatch object

timer.start()     #  Start timing

#
#   Do something here that you wish to time
#

timer.stop()    #  Stop the clock
print(timer.elapsed(), " seconds")
```

This code using a `Stopwatch` object is simpler. A programmer writes code using a `Stopwatch` in a similar way to using an actual stopwatch: push a button to start the clock (call the `start` method), push a button to stop the clock (call the `stop` method), and then read the elapsed time (use the result of the `elapsed` method). Programmers using a `Stopwatch` object in their code are much less likely to make a mistake because the details that make it work are hidden and inaccessible.

Given our experience designing our own types though Python classes, we now are adequately equipped to implement such a `Stopwatch` class. Listing 12.3 (`stopwatch.py`) defines the structure and capabilities of our `Stopwatch` objects.

Listing 12.3: `stopwatch.py`

```python
1  from time import clock
2
```

```
3   class Stopwatch:
4       def __init__(self):
5           self.reset()
6
7       def start(self):  #  Start the timer
8           if not self.__running:
9               self.__start_time = clock()
10              self.__running = True  #  Clock now running
11          else:
12              print("Stopwatch already running")
13
14      def stop(self):       #  Stop the timer
15          if self.__running:
16              self.__elapsed += clock() - self.__start_time
17              self.__running = False   #  Clock stopped
18          else:
19              print("Stopwatch not running")
20
21      def reset(self):      #  Reset the timer
22          self.__start_time = self.__elapsed = 0
23          self.__running = False
24
25      def elapsed(self):    #  Reveal the elapsed time
26          if not self.__running:
27              return self.__elapsed
28          else:
29              print("Stopwatch must be stopped")
30              return None
```

Four methods are available to clients: `start`, `stop`, `reset`, and `elapsed`. A client does not have to worry about the "messy" detail of the arithmetic to compute the elapsed time.

Note that our design forces clients to stop a `Stopwatch` object before calling the `elapsed` method. Failure to do so results in a programmer-defined run-time error report. A variation on this design might allow a client to read the elapsed time without stopping the watch. This implementation allows a user to stop the stopwatch and resume the timing later without resetting the time in between.

Listing 12.4 (`bettersearchcompare.py`) is a rewrite of Listing 10.5 (`searchcompare.py`) that uses our `Stopwatch` object.

Listing 12.4: bettersearchcompare.py

```
1   def binary_search(lst, seek):
2       '''
3       Returns the index of element seek in list lst,
4       if seek is present in lst.
5       lst must be in sorted order.
6       Returns None if seek is not an element of lst.
7       lst is the lst in which to search.
8       seek is the element to find.
9       '''
10      first = 0            #  Initially the first element in list
11      last = len(lst) - 1  #  Initially the last element in list
12      while first <= last:
13          #  mid is middle of the list
14          mid = first + (last - first + 1)//2  # Note: Integer division
```

```
15          if lst[mid] == seek:
16              return mid          #  Found it
17          elif lst[mid] > seek:
18              last = mid - 1    #  continue with 1st half
19          else:  #  v[mid] < seek
20              first = mid + 1  #  continue with 2nd half
21      return None     #  Not there
22
23  def ordered_linear_search(lst, seek):
24      '''
25      Returns the index of element seek in list lst,
26      if seek is present in lst.
27      lst must be in sorted order.
28      Returns None if seek is not an element of lst.
29      lst is the lst in which to search.
30      seek is the element to find.
31      '''
32      i = 0
33      n = len(lst)
34      while i < n and lst[i] <= seek:
35          if lst[i] == seek:
36              return i        #  Return position immediately
37          i += 1
38      return None             #  Element not found
39
40  def test_searches(lst):
41      from stopwatch import Stopwatch
42
43      timer = Stopwatch()
44      #  Find each element using ordered linear search
45      timer.start()     #  Start the clock
46      n = len(lst)
47      for i in range(n):
48          if ordered_linear_search(lst, i) != i:
49              print("error")
50      timer.stop()       #  Stop the clock
51      print("Linear elapsed time", timer.elapsed())
52
53      #  Find each element using binary search
54      timer.reset()       #  Reset the clock
55      timer.start()     #  Start the clock
56      n = len(lst)
57      for i in range(n):
58          if binary_search(lst, i) != i:
59              print("error")
60      timer.stop()       #  Stop the clock
61      print("Binary elapsed time", timer.elapsed())
62
63  def main():
64      SIZE = 20000
65      test_list = list(range(SIZE))
66      test_searches(test_list)
67
68  main()
```

This new, object-oriented version is simpler and more readable.

12.3.2 Automated Testing

We know that just because a program runs to completion without a run-time error does not imply that the program works correctly. We can detect logic errors in our code as we interact with the executing program. The process of exercising code to reveal errors or demonstrate the lack thereof is called *testing*. The informal testing that we have done up to this point has been adequate, but serious software development demands a more formal approach. We will see that good testing requires the same skills and creativity as programming itself.

Until relatively recently in the software development world, testing was often an afterthought. Testing was not perceived to be as glamorous as designing and coding. Poor testing led to buggy programs that frustrated users. Also, tests were written largely after the program's design and coding were complete. The problem with this approach is major design flaws may not be revealed until late in the development cycle. Changes late in the development process are invariably more expensive and difficult to deal with than changes earlier in the process.

Weaknesses in the standard approach to testing led to a new strategy: *test-driven development*. In test-driven development the testing is automated, and the design and implementation of good tests is just as important as the design and development of the actual program. In pure test-driven development, tests are developed *before* any application code is written, and any application code produced is immediately subjected to testing.

Listing 12.5 (`tester.py`) defines the structure of a rudimentary test object.

Listing 12.5: `tester.py`

```
class Tester:
    def __init__(self):
        self.__error_count = self.__total_count = 0
        print("+--------------------------------------")
        print("|  Testing                             ")
        print("+--------------------------------------")

    def check_equals(self, msg, expected, actual):
        print("[", msg, "] ")
        self.__total_count += 1   #  Count this test
        if expected == actual:
            print("OK")
        else:
            self.__error_count += 1  #  Count this failed test
            print("*** Failed!  Expected:", expected, "  actual:", actual)

    def report_results(self):
        print("+--------------------------------------")
        print("|", self.__total_count, "tests run")
        print("|", self.__total_count - self.__error_count, " passed")
        print("|", self.__error_count, " failed")
        print("+--------------------------------------")
```

A simple test object keeps track of the number of tests performed and the number of failures. The client uses the test object to check the results of a computation against a predicted result.

Listing 12.6 (`testliststuff.py`) uses our `Tester` class.

Listing 12.6: `testliststuff.py`

```python
from tester import Tester

#  sort has a bug (it has yet to be written!)
def sort(lst):
    pass    #  Sort not yet implemented

#  sum has a bug (misses first element)
def sum(lst):
    total = 0
    for i in range(1, len(lst)):
        total += lst[i]
    return total

def main():
    t = Tester()    #  Make a test object
    #  Some test cases to test sort
    col = [4, 2, 3]
    sort(col);
    t.check_equals("Sort test #1", [2, 3, 4], col)
    col = [2, 3, 4]
    sort(col);
    t.check_equals("Sort test #2", [2, 3, 4], col)
    #  Some test cases to test sum
    t.check_equals("Sum test #1", sum([0, 3, 4]), 7)
    t.check_equals("Sum test #2", sum([-3, 0, 5]), 2)

    t.report_results()

main()
```

The program's output is

```
+---------------------------------------
|  Testing
+---------------------------------------
[ Sort test #1 ]
*** Failed!  Expected: [2, 3, 4]   actual: [4, 2, 3]
[ Sort test #2 ]
OK
[ Sum test #1 ]
OK
[ Sum test #2 ]
*** Failed!  Expected: 5   actual: 2
+---------------------------------------
| 4 tests run
| 2  passed
| 2  failed
+---------------------------------------
```

Notice that the `sort` function has yet to be implemented, but we can test it anyway. The first test is bound to fail. The second test checks to see if our `sort` function will not disturb an already sorted vector, and we pass this test with no problem.

In the `sum` function, the programmer was careless and used 1 as the beginning index for the vector. Notice that the first test does not catch the error, since the element in the zeroth position (zero) does not affect the outcome. A tester must be creative and even devious to try and force the code under test to demonstrate its errors.

12.4 Class Inheritance

We can base a new class on an existing class using a technique known as *inheritance*. Recall our `Stopwatch` class we defined in Listing 12.3 (`stopwatch.py`). Our `Stopwatch` objects may be started and stopped as often as necessary without resetting the time. Support we need a stopwatch object that records the number of times the watch is started until it is reset. We can build our enhanced `Stopwatch` class from scratch, but it would more efficient to base our new class on the existing `Stopwatch` class. Listing 12.7 (`countingstopwatch.py`) defines our enhanced stopwatch objects.

Listing 12.7: `countingstopwatch.py`

```python
from stopwatch import Stopwatch

class CountingStopwatch (Stopwatch):
    def __init__(self):
        #  Allow superclass to do its initialization of the
        #   inherited fields
        super(CountingStopwatch, self).__init__()
        #  Set number of starts to zero
        self.__count = 0

    def start(self):
        #  Let superclass do its start code
        super(CountingStopwatch, self).start()
        #  Count this start message
        self.__count += 1

    def reset(self):
        #  Let superclass reset the inherited fields
        super(CountingStopwatch, self).reset()
        #  Reset new field
        self.__count = 0

    def count(self):
        return self.__count
```

The line

```python
from stopwatch import Stopwatch
```

indicates that the code in this module will somehow use the `Stopwatch` class from Listing 12.3 (`stopwatch.py`). The line

```python
class CountingStopwatch (Stopwatch):
```

defines a new class named CountingStopwatch, but this new class is based on the existing class Stopwatch. This single line means that the CountingStopwatch class *inherits* everything from the Stopwatch class. CountingStopwatch objects automatically will have start, stop, reset, and elapsed methods.

We say stopwatch is the *superclass* of CountingStopwatch. Another term for superclass is *base class*. CountingStopwatch is the *subclass* of Stopwatch, or, said another way, CountingStopwatch is a *derived class* of Stopwatch.

Even though a subclass inherits all the fields and methods of its superclass, a subclass may add new fields and methods and provide new code for an inherited method. The statement

super(CountingStopwatch, self).__init__()

in the __init__ method definition calls the constructor of the superclass. After executing the superclass constructor code, the subclass constructor defines and initializes the new __count field. The start and reset methods in CountingStopwatch similarly invoke the services of their counterparts in the superclass. The count method is a brand new method not found in the superclass.

Notice that the CountingStopwatch class has no apparent stop method. In fact, it inherits the stop method as is from Stopwatch.

Listing 12.8 (usecountingsw.py) provides some sample client code that uses the CountingStopwatch class.

Listing 12.8: usecountingsw.py

```
1  from countingstopwatch import CountingStopwatch
2  from time import sleep
3
4  timer = CountingStopwatch()
5  timer.start()
6  sleep(10)   #  Pause program for 10 seconds
7  timer.stop()
8  print("Time:", timer.elapsed(), "   Number:", timer.count())
9
10 timer.start()
11 sleep(5)    #  Pause program for 5 seconds
12 timer.stop()
13 print("Time:", timer.elapsed(), "   Number:", timer.count())
14
15 timer.start()
16 sleep(20)   #  Pause program for 20 seconds
17 timer.stop()
18 print("Time:", timer.elapsed(), "   Number:", timer.count())
```

Listing 12.8 (usecountingsw.py) produces

```
Time: 10.010378278632945    Number: 1
Time: 15.016618866378108    Number: 2
Time: 35.02881993198008    Number: 3
```

12.5 Summary

- The `class` reserved word introduces a programmer-defined type.

- Variables of a class are called *objects* or *instances* of that class.

- The dot (`.`) operator is used to access elements of an object.

- A data member of a class is known as a *field*. Equivalent terms include *data member*, *instance variable*, and *attribute*.

- A function defined in a class that operates on objects of that class is called a *method*. Equivalent terms include *member function* and *operation*.

- Encapsulation and data hiding offers several benefits to programmers:

  - Flexibility—class authors are free to change the private details of a class. Existing client code need not be changed to work with the new implementation.

  - Reducing programming errors—if client code cannot touch directly the hidden details of an object, the internal state of that object is completely under the control of the class author. With a well-designed class, clients cannot place the object in an ill-defined state (thus leading to incorrect program execution).

  - Hiding complexity—the hidden internals of an object might be quite complex, but clients cannot see and should not be concerned with those details. Clients need to know *what* an object can do, not *how* it accomplishes the task.

- A field with a name that begins with two underscores (`__`) is no meant to be used directly by clients.

12.6 Exercises

1. Given the definition of the `Rational` number class Listing 12.2 (`rational.py`), complete the function named `add`:

   ```
   def add(r1, r2):
       # Details go here
   ```

   that returns the rational number representing the sum of its two parameters.

2. Given the definition of the geometric `Point` class, complete the function named `distance`:

   ```
   def distance(r1, r2):
       # Details go here
   ```

   that returns the distance between the two points passed as parameters.

3. Given the definition of the `Rational` number class, complete the following function named `reduce`:

   ```
   def reduce(r):
       # Details go here
   ```

   that returns the rational number that represents the parameter reduced to lowest terms; for example, the fraction 10/20 would be reduced to 1/2.

4. What is the purpose of the __init__ method in a class?

5. What is the parameter named `self` that appears as the first parameter of a method?

6. Given the definition of the `Rational` number class, complete the following method named `reduce`:

```python
class Rational:
    #  Other details omitted here ...

    #  Returns an object of the same value reduced
    #  to lowest terms
    def reduce(self):
        #  Details go here
```

that returns the rational number that represents the object reduced to lowest terms; for example, the fraction 10/20 would be reduced to 1/2.

7. Given the definition of the `Rational` number class, complete the following method named `reduce`:

```python
class Rational:
    #  Other details omitted here ...

    #  Reduces the object to lowest terms
    def reduce(self):
        #  Details go here
```

that reduces the object on whose behalf the method is called to lowest terms; for example, the fraction 10/20 would be reduced to 1/2.

8. Given the definition of the geometric `Point` class, add a method named `distance`:

```python
class Point:
    #  Other details omitted

    #  Returns the distance from this point to the
    #  parameter p
    double distance(self, p):
        #  Details go here
```

that returns the distance between the point on whose behalf the method is called and the parameter `p`.

Chapter 13

Handling Exceptions

In our programming experience so far we have encountered several kinds of run-time errors, such as integer division by zero, accessing a list with an out-of-range index, using an object reference set to `None`, and attempting to convert a non-number to an integer. To this point, all of our run-time errors have resulted in the program's termination. Python provides a standard mechanism called *exception handling* that allows programmers to deal with these kinds of run-time errors and many more. Rather than always terminating the program's execution, a program can detect the problem and execute code to correct the issue or manage it in other ways. This chapter explores Python's exception handling mechanism.

13.1 Motivation

Algorithm design can be tricky because the details are crucial. It may be straightforward to write an algorithm to solve a problem in the general case, but there may be a number of special cases that must all be addressed within the algorithm for the algorithm to be correct. Some of these special cases might occur rarely under the most extraordinary circumstances. For the code implementing the algorithm to be robust, these exceptional cases must be handled properly; however, adding the necessary details to the algorithm may render it overly complex and difficult to construct correctly. Such an overly complex algorithm would be difficult for others to read and understand, and it would be harder to debug and extend.

Ideally, a developer would write the algorithm in its general form including any common special cases. Exceptional situations that should arise rarely, along with a strategy to handle them, could appear elsewhere, perhaps as an annotation to the algorithm. Thus, the algorithm is kept focused on solving the problem at hand, and measures to deal with exceptional cases are handled elsewhere.

Python's exception handling infrastructure allows programmers to cleanly separate the code that implements the focused algorithm from the code that deals with exceptional situations that the algorithm may face. This approach is more modular and encourages the development of code that is cleaner and easier to maintain and debug.

An *exception* is a special object that the executing program can create when it encounters an extraordinary situation. Such a situation almost always represents a problem, usually some sort of run-time error. Examples of exceptional situations include:

- attempting to read past the end of a file

- evaluating the expression `lst[i]` where `lst` is a list, and `i > len(lst)`.

- attempting to convert a non-numeric string to a number, as in `int("Fred")`

- attempting to read data from the network when the connection is lost (perhaps due to a server crash or the wire being unplugged from the port).

Many of these potential problems can be handled by the algorithm itself. For example, an `if` statement can test to see if a list index is within the bounds of the list. However, if the list is accessed at many different places within a function, the large number of conditionals in place to ensure the list access safety can quickly obscure the overall logic of the function. Other problems such as the network connection problem are less straightforward to address directly in the algorithm. Fortunately, specific Python exceptions are available to cover problems such as these.

Exceptions represent a standard way to deal with run-time errors. In programming languages that do not support exception handling, programmers must devise their own ways of dealing with exceptional situations. One common approach is for functions to return an integer code that represents success or failure. For example, consider a function named `ReadFile` that is to open a file and read its contents. It returns an integer that is interpreted as follows:

- 0: Success; the function successfully opened and read the contents of the file

- 1: File not found error; the requested file does not exist

- 2: Permissions error; the program is not authorized to read the file

- 3: Device not ready error; for example, a DVD is not present in the drive

- 4: Media error; the program encountered bad sectors on the disk while reading the file

- 5: Some other file error

Notice that zero indicates success, and nonzero indicates failure. Client code that uses the function may look like

```python
if ReadFile("stats.data") == 0:
    # Code to execute if the file was read properly
else:
    # Code to execute if an error occurred while reading the file
```

The developers of `ReadFile` were looking toward the future, since any value above 4 represents some unspecified file error. New codes can be specified (for example, 5 may mean illegal file format). Existing client code that uses the updated class containing `ReadFile` will still work (5 > 4 just represents some kind of file error), but new client code can explicitly check for a return value of 5 and act accordingly.

This kind of error handling has its limitations, however. The primary purpose of some functions is to return an integer result that is not an indication of an error (for example, the `int` function). Perhaps a string could be returned instead? Unfortunately, some functions naturally return strings (like the `str` function). Also, returning a string would not work for a function that naturally returns an integer as its result. A completely different type of exception handling technique would need to be developed for functions such as these.

The return-value-as-error-status approach can be cumbersome to use for complicated programming situations. Consider the situation where function `A` calls function `B` which calls function `C` which calls function `D` which calls `ReadFile`:

```
A → B → C → D → ReadFile
```

Suppose function `A` is concerned about the file being opened correctly and read. The `ReadFile` function returns an error status, but this value is returned to function `D`, the function that calls `ReadFile` directly. If `A` really needs to know about how `ReadFile` worked, then all the functions in between in the call chain (`B`, `C`, and `D`) must also return an error status. The process essentially passes the error status of `ReadFile` back up the call chain to `A`. While this is inconvenient at best, it may be impossible in general. Suppose `D`'s job is to read the data in the file (via `ReadFile`) and then pass each piece of data read to another function called `Process`. Now `Process` also returns an integer value that indicates its error status. If the data passed to `Process` is not of the proper format, it returns 1; otherwise, it returns 0. If function `A` needs to know specifics about why the data file was not properly read in and processed (was it a problem reading the file with `ReadFile` or a problem with the data format with `Process`?), it cannot distinguish the cause from the single error indication passed up the call chain.

The main problem with these ad hoc approaches to exception handling is that the error handling facilities developed by one programmer may be incompatible with those used by another. A comprehensive, uniform exception handling mechanism is needed. Python's exceptions provide such a framework. Python's exception handling infrastructure leads to code that is logically cleaner and less prone to programming errors. Exceptions are used in the standard Python API, and programmers can create new exceptions that address issues specific to their particular problems. These exceptions all use a common mechanism and are completely compatible with each other.

13.2 Exception Examples

The following small Python program certainly will cause a run-time error if the user enters the word "five" instead of typing the digit 5.

```python
x = int(input("Please enter a small positive integer: "))
print("x =", x)
```

If the user enters "five," this code results in the run-time environment reporting a `ValueError` exception before killing the program.

We can wrap this code in a `try`/`except` construct as

```python
try:
    x = int(input("Please enter a small positive integer: "))
    print("x =", x)
except ValueError:
    print("Input cannot be parsed as an integer")
```

Now if the user enters "five" when this section of code is executed, the program displays

```
Input cannot be parsed as an integer
```

Notably, the program does not crash. The `try` block

```python
try:
    #  Code that might raise an exception goes here . . .
```

wraps the code segment that has the potential to produce an exception. The `except` block

```
except ValueError:
    #  Code to execute if the except block produced an exception goes here . . .
```

provides the code to be executed only if the code within the `try` block does indeed produce a `ValueError` exception. We say code within the `except` block *handles* the exception that code within the `try` block *raises*. Code within the `exception` block constitutes "Plan B;" that is, what to do if the code in the `try` block fails.

Consider Listing 13.1 (`pitfalls.py`) which contains a common potential problem and two real problems.

Listing 13.1: `pitfalls.py`

```
1  #  I hope the user enters a valid Python integer!
2  x = int(input("Please enter a small positive integer: "))
3  print("x =", x)
4  if x < 5:
5      a = None
6      a[3] = 2     #  Using None as a populated list!
7  elif x < 10:
8      a = [0, 1]
9      a[2] = 3     #  Exceeding the list's bounds
```

Here are the problems with Listing 13.1 (`pitfalls.py`):

- If the user enters a non-integer, the program crashes with a `ValueError` run-time error. We have tolerated this behavior for too long enough, and it is time to defend against this possibility.

- If the user enters an integer less than five, the program attempts to use `None` as a list. The program thus crashes with a `TypeError` error.

- If the user enters an integer in the range 6...9, the program attempts to access a list with an index outside the range of the list. This results in an `IndexError` run-time error.

Consider Listing 13.2 (`handlepitfalls.py`) shows how to handle multiple exceptions in a section of code.

Listing 13.2: `handlepitfalls.py`

```
1  x = 0
2  while x < 100:
3      try:
4          #  I hope the user enters a valid Python integer!
5          x = int(input("Please enter a small positive integer: "))
6          print("x =", x)
7          if x < 5:
8              a = None
9              a[3] = 2     #  Using None as a populated list!
10         elif x < 10:
11             a = [0, 1]
12             a[2] = 3     #  Exceeding the list's bounds
13     except ValueError:
14         print("Input cannot be parsed as an integer")
15     except TypeError:
```

```
16            print("Trying to use a None as a valid object")
17        except IndexError:
18            print("Straying from the bounds of the list")
19        print("Program continues")
20 print("Program finished")
```

In Listing 13.2 (handlepitfalls.py), we finally address the issue of robust user numeric input. Up to this point, if we wished to obtain an integer from the user, we wrote code such as

```
value = int(input("Enter an integer: "))
```

and hoped the user does not enter 2.45 or the word *fred*. Bad input in Listing 13.2 (handlepitfalls.py) causes the program to scold the user but does not terminate the program.

13.3   Using Exceptions

Exceptions should be reserved for uncommon errors. For example, the following code adds up all the elements in a list of numbers named lst:

```
sum = 0
for elem in range(len(lst)):
    sum += elem
print("Sum =", sum)
```

This loop is fairly typical. Another approach uses exceptions:

```
sum = 0
int i = 0
try:
    while True:
        sum += lst[i]
        i += 1
except IndexError:
    pass
print("Sum =", sum)
```

Both approaches compute the same result. In the second approach the loop is terminated when the list access is out of bounds. The statement is interrupted in midstream so sum's value is not incorrectly incremented. However, the second approach *always* throws and catches an exception. The exception definitely is **not** an uncommon occurrence.

Exceptions should not be used to dictate normal logical flow. While very useful for its intended purpose, the exception mechanism adds some overhead to program execution, especially when an exception is raised. This overhead is reasonable when exceptions are rare but not when exceptions are part of the program's normal execution.

Exceptions are valuable aids for careless or novice programmers. A careful programmer ensures that code accessing a list does not exceed the list's bounds. Another programmer's code may accidentally attempt to access a[len(a)]. A novice may believe a[len(a)] is a valid element. Since no programmer is perfect, exceptions provide a nice safety net.

As you develop more sophisticated classes you will find exceptions more compelling. You should analyze your classes and methods carefully to determine their limitations. Exceptions can be valuable for

covering these limitations. Exceptions are used extensively throughout the Python standard class library. Programs that make use of these classes must properly handle the exceptions they can throw.

13.4 Custom Exceptions

13.5 Summary

- Add summary items

13.6 Exercises

1. Add exercises

Index