Fundamentals of Python Programming

DRAFT

Richard L. Halterman
Southern Adventist University

July 9, 2019

Fundamentals of Python Programming

See the preface for the terms of use of this document.

Contents

1 The Context of Software Development **1**

1.1 Software . . . 2
1.2 Development Tools . . . 2
1.3 Learning Programming with Python . . . 4
1.4 Writing a Python Program . . . 5
1.5 The Python Interactive Shell . . . 9
1.6 A Longer Python program . . . 11
1.7 Exercises . . . 12

2 Values and Variables **13**

2.1 Integer and String Values . . . 13
2.2 Variables and Assignment . . . 17
2.3 Identifiers . . . 24
2.4 Floating-point Numbers . . . 26
2.5 Control Codes within Strings . . . 30
2.6 User Input . . . 31
2.7 Controlling the `print` Function . . . 34
2.8 String Formatting . . . 35
2.9 Multi-line Strings . . . 39
2.10 Exercises . . . 40

3 Expressions and Arithmetic **43**

3.1 Expressions . . . 43
3.2 Mixed Type Expressions . . . 49
3.3 Operator Precedence and Associativity . . . 49
3.4 Formatting Expressions . . . 51

3.5 Comments . . . 52
3.6 Errors . . . 53
3.6.1 Syntax Errors . . . 53
3.6.2 Run-time Exceptions . . . 54
3.6.3 Logic Errors . . . 56
3.7 Arithmetic Examples . . . 57
3.8 More Arithmetic Operators . . . 59
3.9 Algorithms . . . 61
3.10 Exercises . . . 62

4 Conditional Execution 67
4.1 Boolean Expressions . . . 67
4.2 Boolean Expressions . . . 68
4.3 The Simple if Statement . . . 69
4.4 The if/else Statement . . . 75
4.5 Compound Boolean Expressions . . . 77
4.6 The `pass` Statement . . . 80
4.7 Floating-point Equality . . . 82
4.8 Nested Conditionals . . . 83
4.9 Multi-way Decision Statements . . . 93
4.10 Multi-way Versus Sequential Conditionals . . . 97
4.11 Conditional Expressions . . . 99
4.12 Errors in Conditional Statements . . . 102
4.13 Logic Complexity . . . 105
4.14 Exercises . . . 107

5 Iteration 113
5.1 The while Statement . . . 113
5.2 Definite Loops vs. Indefinite Loops . . . 121
5.3 The `for` Statement . . . 122
5.4 Nested Loops . . . 126
5.5 Abnormal Loop Termination . . . 132
5.5.1 The `break` statement . . . 133
5.5.2 The continue Statement . . . 136
5.6 while/else and for/else . . . 137

5.7 Infinite Loops . . . 139
5.8 Iteration Examples . . . 143
5.8.1 Computing Square Root . . . 143
5.8.2 Drawing a Tree . . . 144
5.8.3 Printing Prime Numbers . . . 146
5.8.4 Insisting on the Proper Input . . . 150
5.9 Exercises . . . 150

6 Using Functions **157**
6.1 Introduction to Using Functions . . . 158
6.2 Functions and Modules . . . 162
6.3 The Built-in Functions . . . 164
6.4 Standard Mathematical Functions . . . 167
6.5 `time` Functions . . . 170
6.6 Random Numbers . . . 173
6.7 System-specific Functions . . . 176
6.8 The `eval` and `exec` Functions . . . 176
6.9 Turtle Graphics . . . 179
6.10 Other Techniques for Importing Functions and Modules . . . 185
6.11 Exercises . . . 191

7 Writing Functions **193**
7.1 Function Basics . . . 194
7.2 Parameter Passing . . . 209
7.3 Documenting Functions . . . 211
7.4 Function Examples . . . 213
7.4.1 Better Organized Prime Generator . . . 213
7.4.2 Command Interpreter . . . 215
7.4.3 Restricted Input . . . 216
7.4.4 Better Die Rolling Simulator . . . 218
7.4.5 Tree Drawing Function . . . 219
7.4.6 Floating-point Equality . . . 220
7.5 Refactoring to Eliminate Code Duplication . . . 222
7.6 Custom Functions vs. Standard Functions . . . 224
7.7 Exercises . . . 227

8 More on Functions **233**

8.1 Global Variables . . . 233
8.2 Default Parameters . . . 238
8.3 Introduction to Recursion . . . 241
8.4 Making Functions Reusable . . . 246
8.5 Functions as Data . . . 249
8.6 Separating Concerns with Pluggable Modules . . . 253
8.7 Lambda Expressions . . . 273
8.8 Generators . . . 278
8.9 Local Function Definitions . . . 286
8.10 Decorators . . . 292
8.11 Partial Application . . . 303
8.12 Exercises . . . 306

9 Objects **311**

9.1 Using Objects . . . 311
9.2 String Objects . . . 312
9.3 File Objects . . . 316
9.4 `Fraction` Objects . . . 323
9.5 Turtle Graphics Objects . . . 325
9.6 Graphics with `tkinter` Objects . . . 326
9.7 Other Standard Python Objects . . . 332
9.8 Object Mutability and Aliasing . . . 332
9.9 Garbage Collection . . . 336
9.10 Exercises . . . 337

10 Lists **339**

10.1 Using Lists . . . 341
10.2 List Traversal . . . 345
10.3 Building Lists . . . 346
10.4 List Membership . . . 351
10.5 List Assignment and Equivalence . . . 352
10.6 List Bounds . . . 357
10.7 Slicing . . . 358
10.8 List Element Removal . . . 361

10.9 Lists and Functions . . . 362
10.10 List Methods . . . 363
10.11 Prime Generation with a List . . . 366
10.12 Command-line Arguments . . . 368
10.13 List Comprehensions . . . 369
10.14 Multidimensional Lists . . . 375
10.15 Summary of List Creation Techniques . . . 384
10.16 Lists vs. Generators . . . 385
10.17 Exercises . . . 385

11 Tuples, Dictionaries, and Sets 389

11.1 Tuples . . . 389
11.2 Arbitrary Argument Lists . . . 393
11.3 Dictionaries . . . 398
11.4 Using Dictionaries . . . 402
11.5 Counting with Dictionaries . . . 404
11.6 Grouping with Dictionaries . . . 408
11.7 Keyword Arguments . . . 411
11.8 Sets . . . 414
11.9 Set Quantification with `all` and `any` . . . 415
11.10 Enumerating the Elements of a Data Structure . . . 419
11.11 Exercises . . . 421

12 Handling Exceptions 425

12.1 Motivation . . . 425
12.2 Common Standard Exceptions . . . 426
12.3 Handling Exceptions . . . 428
12.4 Handling Multiple Exceptions . . . 431
12.5 The Catch-all Handler . . . 432
12.6 Catching Exception Objects . . . 436
12.7 Exception Handling Scope . . . 437
12.8 Raising Exceptions . . . 446
12.9 The `try` Statement's Optional `else` Block . . . 451
12.10 `finally` block . . . 453
12.11 Using Exceptions Wisely . . . 457

12.12 Exercises 458

13 Custom Types **463**

13.1 Circle Objects 463
13.2 Restricting Access to Members 473
13.3 Rational Numbers 474
13.4 Bank Account Objects 479
13.5 Timing Objects 483
13.6 Traffic Light Objects 486
13.7 Automated Testing 492
13.8 Plotting Data 497
13.9 Dynamic Content 502
13.10 Class Variables 506
13.11 Exercises 507

14 Class Design: Composition and Inheritance **513**

14.1 Composition 513
14.2 Class Inheritance 518
14.3 Composition vs. Inheritance 537
14.4 Multiple Inheritance 540
14.5 Unit Testing 558
14.6 Custom Exceptions 563
14.7 Exercises 565

15 Algorithm Quality **567**

15.1 Good Algorithms Versus Bad Algorithms 567
15.2 Sorting 576
15.3 Flexible Sorting 579
15.4 Search 582
15.4.1 Linear Search 582
15.4.2 Binary Search 584
15.5 Recursion Revisited 593
15.6 List Permutations 600
15.7 Randomly Permuting a List 609
15.8 Reversing a List 614

15.9 Memoization 615
15.10 Exercises 627

16 Representing Relationships with Graphs **631**
16.1 Introduction to Graphs 631
16.2 Implementing Graphs in Python 635
16.3 Path Finding 635
16.4 Breadth-first Search 637
16.5 Depth-first Search 647
16.6 Exercises 654

Index **655**

Preface

Legal Notices and Information

If you are an instructor using this book in one or more of your courses, please let me know. Keeping track of how and where this book is used helps me justify to my employer that it is providing a useful service to the community and worthy of the time I spend working on it. Simply send a message to `halterman@southern.edu` with your name, your institution, and the course(s) in which you use it.

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, mouse, and perhaps even other computers across a network 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

10001011011000010001000001001110

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 8.1 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 i7 in the Windows machine accepts a completely different binary language than the PowerPC processor in the older 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 over 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 problem 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
```

While these three lines do constitute a proper Python program, they more likely are merely a small piece of a larger program. The lines of text 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*, represent 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. Instead of some cryptic binary instructions meant only for the processor, we see familiar-looking mathematical operators (= and +). Since this program is expressed in the Python language, not machine language, no computer processor can execute the program directly. 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 corresponding 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 programmers must follow to create well-formed programs. Only well-formed programs are acceptable for translation into executable machine code. Some syntax-aware editors can use colors or other special annotations to alert programmers of syntax errors during the editing process.

- **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.) Compilers translate the contents of a source file and produce a file containing all the target code. Popular compiled languages include C, C++, Java, C#.

- **Interpreters**. An *interpreter* is like a compiler, in that it translates higher-level source code into target code (usually 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 each time a user runs the program. A compiled program does not need to be recompiled to run, but an interpreted program must be reinterpreted each time it executes. For this reason interpreted languages are often refered to as *scripting languages*. The interpreter in essence reads the script, where the script is the source code of the program. 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. Popular scripting languages include Python, Ruby, Perl, and, for web browsers, Javascript.

- **Debuggers**. A *debugger* allows a programmer to more easily trace a program's execution in order to locate and correct errors in the program's implementation. With a debugger, a developer can simultaneously run a program and see which line in the source code is responsible for the program's current actions. The programmer can watch the values of variables and other program elements 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 3.6 for more information about programming errors.)

- **Profilers**. A *profiler* collects statistics about a program's execution allowing developers to tune appropriate parts of the program to improve its overall performance. A profiler indicates how many times a portion of a program is executed during a particular run, and how long that portion takes to execute. Developers also can use profilers 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. Python IDEs include Wingware, Enthought, and IDLE.

Despite the wide variety 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, Facebook, 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`.

In late 2008, Python 3.0 was released. Commonly called Python 3, the current version of Python is incompatible with earlier versions of the language. Currently the Python world still is in transition between Python 2 and Python 3. Many existing published books cover Python 2, but more Python 3 resources now are becoming widely available. 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

The text that makes up a Python program has 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.

A program consists of one or more *statements*. A statement is an instruction that the interpreter executes. The following statement invokes the `print` function to display a message:

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

We can use the statement in a program. Listing 1.1 (simple.py) is one of the simplest Python programs that does something:

Listing 1.1: `simple.py`

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

We will use Wingware's *WingIDE 101* to develop our Python programs. This integrated development environment is freely available from `http://wingware.com/downloads/wingide-101`, and its target audience is beginning Python programmers. Its feature set and ease of use make *WingIDE 101* an ideal platform for exploring programming in Python.

The way you launch *WingIDE 101* depends on your operating system and how it was installed. Figure 1.1 shows a screenshot of *WingIDE 101* running on a Windows 8.1 computer. The IDE consists of a menu bar at the top, along with a tool bar directly underneath it, and several sub-panes within the window. The large, unlabeled pane in the upper left portion of the window is the editor pane in which we type in our program's source code. The versions of *WingIDE 101* for Apple Mac OS X and Linux are similar in appearance.

To begin entering our program, we will choose the *New* item from the *File* menu (*File→New* menu sequence), as shown in Figure 1.2. This action produces a new editor pane for a file named Unititled-1.py.

Figure 1.1 *WingIDE 101* running under Microsoft Windows

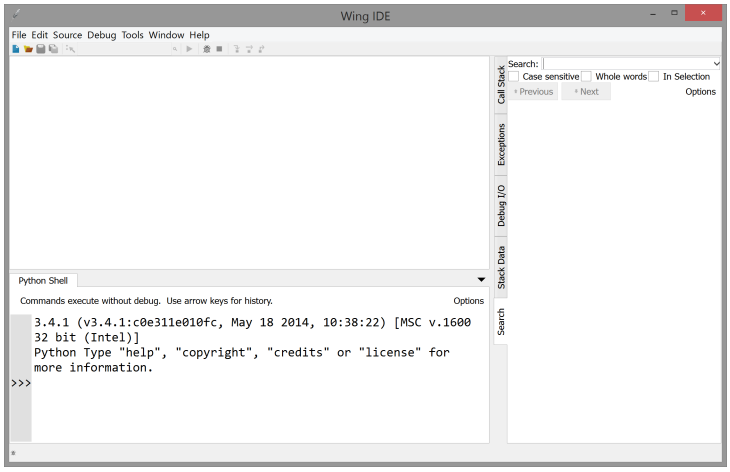


Figure 1.2 The menu selection to create a new Python program.

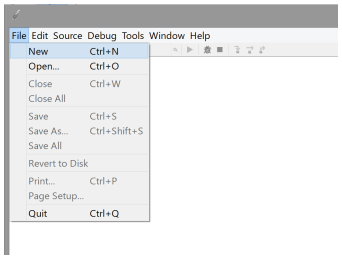


Figure 1.3 The new, untitled editor pane ready for code.

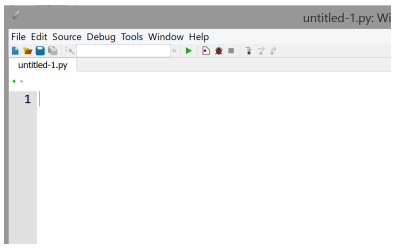
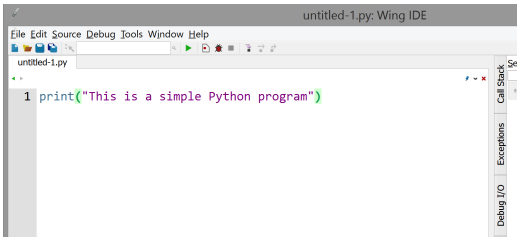


Figure 1.4 The code for the simple program after typed into the editor pane.



As Figure 1.3 shows, the file's name appears in the editor's tab at the top. (We will save the file with a different name later.)

We now are ready to type in the code that constitutes the program. Figure 1.4 shows the text to type.

Next we will save the file. The menu sequence *File→Save*, also shown in Figure 1.5, produces the dialog box shown in Figure 1.6 that allows us to select a folder and filename for our program. You should name all Python programs with a .py extension.

The *WingIDE-101* IDE provides two different ways to execute the program. We can *run* the program by selecting the little green triangle under the menu bar, as shown in Figure 1.7. The pane labeled *Python Shell* will display the program's output. Figure 1.8 shows the results of running the program.

Another way to execute the program is via the *Debug* button on the menu, as shown in Figure 1.9. When debugging the program, the executing program's output appears in the *Debug I/O* pane as shown in Figure 1.10.

Which the better choice, the *Run* option or the *Debug* option? As we will see later (see Section 5.7), the debugging option provides developers more control over the program's execution, so, during development,

Figure 1.5 Save the Python program

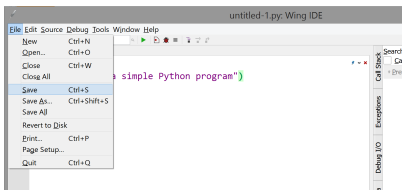


Figure 1.6 The file save dialog allows the user to name the Python file and locate the file in a particular folder.

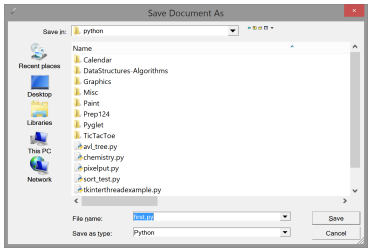


Figure 1.7 Running the program

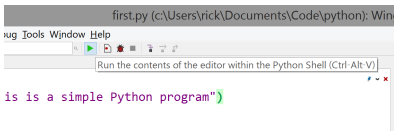


Figure 1.8 *WingIDE 101* running under Microsoft Windows

Figure 1.9 Debugging the program

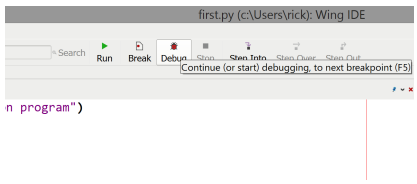


Figure 1.10 Debugger output

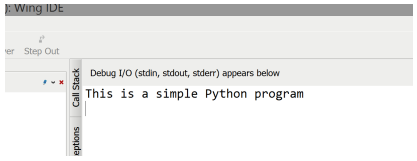
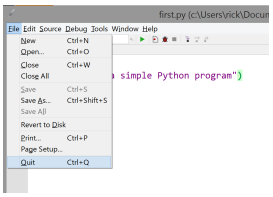


Figure 1.11 *WingIDE 101* running under Microsoft Windows



we prefer the *Debug* option to the *Run* option.

When you are finished programming and wish to quit the IDE, follow the menu sequence *File→Quit* as shown in Figure 1.11.

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 we can use to build programs, and the chapters that follow explore these various kinds of statements.

While integrated development environments like Wingware's *WingIDE-101* are useful for developing Python programs, we can execute Python programs directly from a command line. In Microsoft Windows, the command console (cmd.exe) and PowerShell offer command lines. In Apple Mac OS X, Linux, and Unix, a terminal provides a command line. Figure 1.12 shows the Windows command shell running a Python program. In all cases the user's `PATH` environment variable must be set properly in order for the operating system to find the Python interpreter to run the program.

Figure 1.8 shows that *WingIDE 101* displays a program's output as black text on a white background. In order to better distinguish visually in this text program source code from program output, we will render the program's output with white text on a black background, as it would appear in the command line interpreter under Windows as shown in Figure 1.12. This means we would show the output of Listing 1.1 (simple.py) as

```
This is a simple Python program
```

1.5 The Python Interactive Shell

We created the program in Listing 1.1 (simple.py) and submitted it to the Python interpreter for execution. We can interact with the interpreter directly, typing in Python statements and expressions for its immediate execution. As we saw in Figure 1.8, the *WingIDE 101* pane labeled *Python Shell* is where the executing program directs its output. We also can type commands into the *Python Shell* pane, and the interpreter

Figure 1.12 Running a Python program from the command line

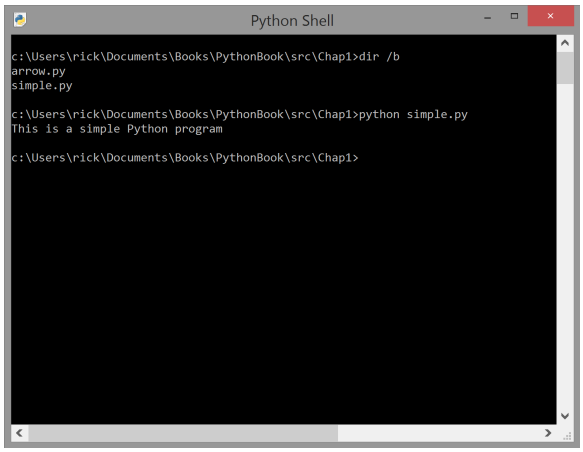


Figure 1.13 The interactive shell allows us to submit Python statements and expressions directly to the interpreter

will attempt to execute them. Figure 1.13 shows how the interpreter responds when we enter the program statement directly into the shell. The interpreter prompts the user for input with three greater-than symbols (>>>). This means the user typed in the text on the line prefixed with >>>. Any lines without the >>> prefix represent the interpreter's output, or feedback, to the user.

We will find Python's interactive interpreter invaluable for experimenting with various language constructs. We can discover many things about Python without ever writing a complete program.

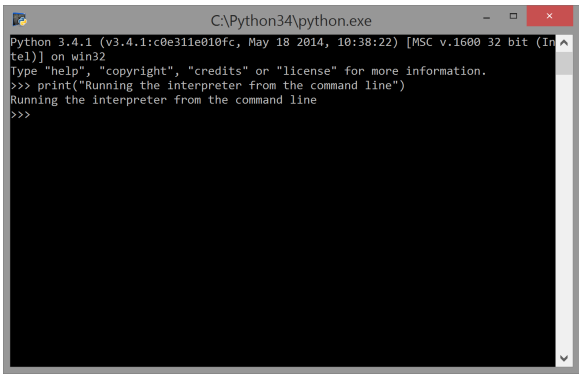
We can execute the interactive Python interpreter directly from the command line, as Figure 1.14 demonstrates. This means not only can we execute Python programs apart from the *WingIDE 101* developer environment, we also we can access Python's interactive interpreter separately from *WingIDE 101* if we so choose.

Figure 1.13 shows that the *WingIDE 101* interpreter pane displays black text on a white background. In order for readers of this text to better distinguish visually program source code from program output, we will render the user's direct interaction with the Python interpreter as black text on a light-gray background. As an example, the following shows a possible interactive session with a user:

```
>>> print("Hello!")
Hello!
```

The interpreter prompt (>>>) prefixes all user input in the interactive shell. Lines that do not begin with the

Figure 1.14 Running the Python interpreter from the command line



>>> prompt represent the interpreter's response.

1.6 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`

```
print("    *    ")
print("   ***   ")
print("  *****  ")
print("    *    ")
print("    *    ")
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 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 the interpreter generates error messages for improper indentation. 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.

1.7 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 is 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
- strings
- 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 and String 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 nonnumeric 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 a basic building block of a Python 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 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40)
[MSC v.1600 64 bit (AM
D64)] on win32
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 arithmetic 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 nonnumeric characters:

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

Python recognizes both single quotes (`'`) and double quotes (`"`) as valid ways to delimit a string value. The word *delimit* means to determine the boundaries or limits of something. The left `'` symbol determines the

beginning of a string, and the right ' symbol that follows specifies the end of the string. 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 two kinds of quotation marks when used to delimit a particular string, as the following interactive sequence shows:

```
>>> '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
```

Notice that with the missing quotation marks the interpreter does not accept the expression `Fred`.

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 creates an actual integer object from a string that looks like an integer, and the `str` function creates a string object from the digits that make up an integer:

```
>>> 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 transform an object into itself—the expression `int(4)` is more easily expressed as `4`, so the utility of the `str` and `int` functions will not become apparent until we introduce variables (Section 2.2) and need to process user input (Section 2.6).

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`

```
x = 10
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.

We may assign and reassign a variable 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
```

We can reassign different values to a variable as needed, as Listing 2.2 (multipleassignment.py) shows.

Listing 2.2: `multipleassignment.py`

```
x = 10
print(x)
x = 20
print(x)
x = 30
print(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:

```
10
20
30
```

This program demonstrates that can we cannot always predict the behavior of a statement in isolation, especially if that statement involves a variable. This is because the statement's behavior may be dependent on the assigned values of one or more variables that it uses.

The variable x in Listing 2.2 (multipleassignment.py) has type `int`, since x is bound to an integer value. Listing 2.3 (multipleassignment2.py) is an enhancement of Listing 2.2 (multipleassignment.py) that provides more descriptive `print` statements.

Listing 2.3: `multipleassignment2.py`

```
x = 10
print('x = ' + str(x))
x = 20
print('x = ' + str(x))
x = 30
print('x = ' + str(x))
```

Listing 2.3 (multipleassignment2.py) outputs

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

Listing 2.3 (multipleassignment2.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; as indicated in Section 2.1, the plus (+) operator may not applied with mixed string and integer operands.

Listing 2.4 (multipleassignment3.py) provides a variation of Listing 2.3 (multipleassignment2.py) that produces the same output.

Listing 2.4: `multipleassignment3.py`

```
x = 10
print('x =',  x)
x = 20
print('x =',  x)
x = 30
print('x =',  x)
```

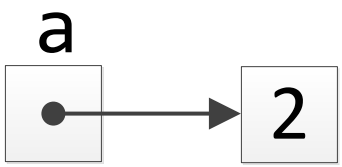
This version of the `print` statement:

```
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 the printed text elements do not run together.

A programmer may assign multiple variables in one statement using *tuple assignment*. Listing 2.5 (tupleassign.py) shows how:

Figure 2.1 Binding a variable to an object



Listing 2.5: tupleassign.py

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

The Listing 2.5 (tupleassign.py) program produces

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

A *tuple* is a comma-separated list of expressions. If the variables `total` and `s` are defined, the expression `total, 45, s, 0.3` represents a 4-tuple; that is, a tuple with composed of four elements. 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.

Tuple assignment works only if the tuple on the left side of the assignment operator contains the same number of elements as the tuple on the right side, as the following example illustrates:

```
>>> x, y, z = 45, 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
>>> x, y, z = 45, 3, 23, 8
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 3)
```

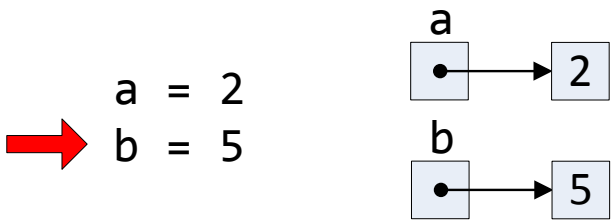
A tuple is a kind of Python type, like `int` or `float`, and we explore tuples in more detail in Chapter 11.

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

One box represents the variable, so 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. In this case the arrow points to another box that contains the value 2. The second box represents a memory location that holds the internal binary representation of the value 2.

Figure 2.2 How variable bindings change as a program runs: step 1

Figure 2.3 How variable bindings change as a program runs: step 2



To see how variable bindings can change as the computer executes a sequence of assignment statements, consider the following sequence of Python statements:

```
a = 2
b = 5
a = 3
a = b
b = 7
```

Figures 2.2–2.6 illustrate how the variable bindings change as the Python interpreter executes each of the above statements. Importantly, the statement

```
a = b
```

means that a and b both are bound to the same numeric object. Observe that later 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

Figure 2.4 How variable bindings change as a program runs: step 3

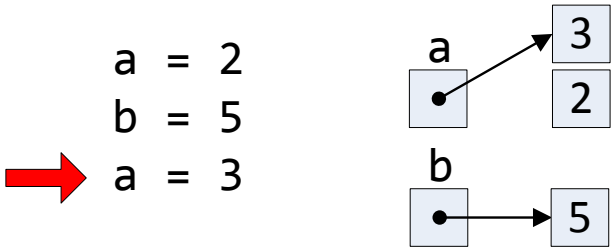


Figure 2.5 How variable bindings change as a program runs: step 4

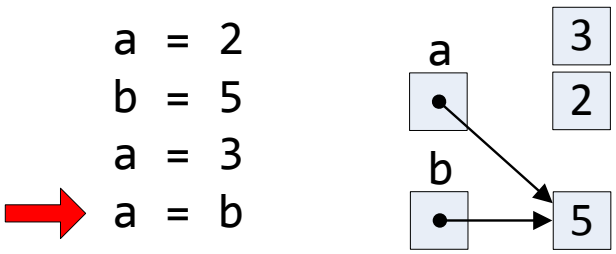
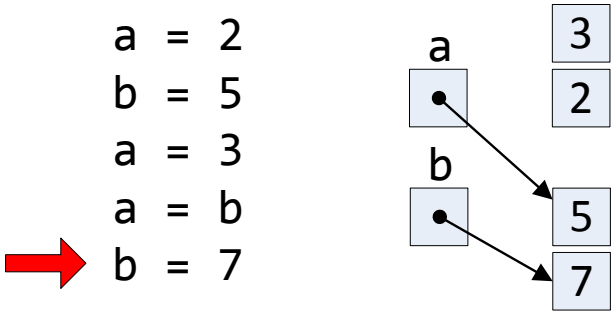


Figure 2.6 How variable bindings change as a program runs: step 5



can change as well. Consider Listing 2.6 (changeabletype.py).

Listing 2.6: changeabletype.py

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

Listing 2.6 (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'>
```

Programmers infrequently perform assignments that change a variable's type. 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.

A variable that has not been assigned is an *undefined variable* or *unbound variable*. Any attempt to use an undefined variable is an error, as the following sequence from Python's interactive shell shows:

```
>>> x = 2
>>> x
2
>>> y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
```

The assignment statement binds 2 to the variable x, and after that the interpreter can evaluate x. The interpreter cannot evaluate the variable y, so it reports an error.

In rare circumstances we may want to undefine a previously defined variable. The del statement does that, as the following interactive sequence illustrates:

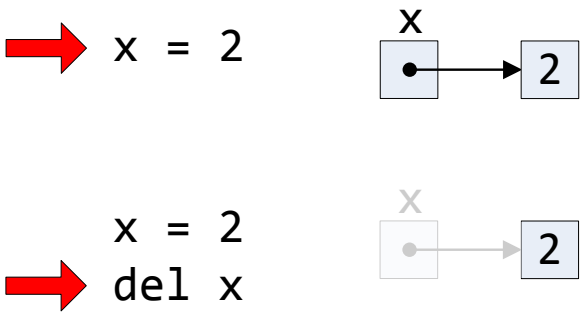
```
>>> x = 2
>>> x
2
>>> del x
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

The del keyword stands for *delete*, and so del deletes or removes a variable's definition from the current interpreter session or from an executing Python program. Figure 2.7 illustrates the definition and subsequent deletion of variable x. If variables a, b, and c currently are defined, the statement

```
del a, b, c
```

undefines all three variables in one statement.

Figure 2.7 Definition and subsequent deletion of variable x



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 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 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:

- An identifiers must contain at least one character.
- The first character of an identifiers 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).

Examples of valid Python identifiers include

- `x`
- `x2`
- `total`
- `port_22`

Table 2.1 Python keywords

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	

- 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` (the asterisk is not a legal symbol in an identifier)
- `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 3.6 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 we reassign a function name, but Python allows us to assign a variable to a function.

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

After binding the variable `my_print` to `print` we can use `my_print` in exactly as we would use 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` is a reserved word. 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.

Programmers generally avoid distinguishing between two variables in the same context merely by differences in capitalization. Doing so is more likely to confuse 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 a variable's name is that it 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 for such an application.

2.4 Floating-point Numbers

Many computational tasks require numbers that have fractional parts. For example, to compute the area of a circle given the circle's radius, we use the value π, or approximately 3.14159. 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'>
```


Table 2.2 Characteristics of Floating-point Numbers

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

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 most 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 2.7 (pi-print.py) prints an approximation of the mathematical value π.

Listing 2.7: `pi-print.py`

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

The first line in Listing 2.7 (pi-print.py) assigns an approximation of π to the variable named `pi`, and the second line prints its value. The last line prints some text along with a literal floating-point value. Any literal numeric value with a decimal point in a Python program automatically has the type `float`. This means the Python literal `2.0` is a `float`, not an `int`, even though mathematically we would classify it as an integer.

Floating-point numbers are an approximation of mathematical real numbers. The range of floating-point numbers is limited, since each value requires a fixed amount of memory. Floating-point numbers differ from integers in another, very important way. An integer has an exact representation. This is not true necessarily for a floating-point number. Consider the real number π. The mathematical constant 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, a Python program can only approximate π's value. Because of the limited number of digits available to floating-point numbers, Python cannot represent exactly even some numbers with a finite number of digits; for example, the number 23.3123400654033989 contains too many digits for the `float` type. As the following interaction sequence shows, Python stores 23.3123400654033989 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).

We can express floating-point numbers in scientific notation. Since most programming editors do not provide superscripting and special symbols like $\times$, Python slightly alters the normal scientific notation. The number 6.022×10^{23} is written `6.022e23`. The number to the left of the `e` (we can use capital `E` 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 2.8 (scientificnotation.py) prints some scientific constants using scientific notation.

Listing 2.8: scientificnotation.py

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

The type of any literal expressed scientific notation always has type `float`; for example, the Python expression `2e3` is a `float`, even though conceptually we may consider it the same as integer the 2,000.

Unlike floating-point numbers, integers are whole numbers and cannot store fractional quantities. We can convert a floating-point to an integer in two fundamentally different ways:

- Rounding adds or subtracts a fractional amount as necessary to produce the integer closest to the original floating-point value.
- Truncation simply drops the fractional part of the floating-point number, thus keeping whole number part that remains.

We can see how rounding and truncation differ in Python's interactive shell:

```
>>> 28.71
28.71
>>> int(28.71)
28
>>> round(28.71)
29
>>> round(19.47)
19
>>> int(19.47)
19
```

As we can see, truncation always "rounds down," while rounding behaves as we would expect.

We also can use the `round` function to round a floating-point number to a specified number of decimal places. The `round` function accepts an optional argument that produces a floating-point rounded to fewer decimal places. The additional argument must be an integer and specifies the desired number of decimal places to round. In the shell we see

```
>>> x
93.34836
>>> round(x)
93
>>> round(x, 2)
93.35
>>> round(x, 3)
93.348
>>> round(x, 0)
93.0
>>> round(x, 1)
93.3
>>> type(round(x))
<class 'int'>
>>> type(round(x, 1))
<class 'float'>
```



```
>>> type(round(x, 0))
<class 'float'>
```

As we can see, the single-argument version of `round` produces an integer result, but the two-argument version produces a floating-point result.

The second argument to the `round` function may be a negative integer:

```
>>> x = 28793.54836
>>> round(x)
28794
>>> round(x, 1)
28793.5
>>> round(x, 2)
28793.55
>>> round(x, 0)
28794.0
>>> round(x, 1)
28793.5
>>> round(x, -1)
28790.0
>>> round(x, -2)
28800.0
>>> round(x, -3)
29000.0
```

The expression `round(`n`, `r`)` rounds floating-point expression n to the 10^{-r} decimal digit; for example, `round(n, -2)` rounds floating-point value `n` to the hundreds place (10^2). Similarly, `round(n, 3)` rounds floating-point value `n` to the thousandths place (10^{-3}).

The `round` function can be useful for integer values as well. If the first argument to `round` is an integer, and the second argument to round is a negative integer, the second argument specifies the number decimal places to the *left* of the decimal point to round. Consider the following experiments:

```
>>> round(65535)
65535
>>> round(65535, 0)
65535
>>> round(65535, 1)
65535
>>> round(65535, 2)
65535
>>> round(65535, -1)
65540
>>> round(65535, -2)
65500
>>> round(65535, -3)
66000
>>> round(65535, -4)
70000
>>> round(65535, -5)
100000
>>> round(65535, -6)
0
```


In all of these cases the round function produced an integer result. As you can see, if the second argument is a nonnegative integer, the round function evaluates to the original value.

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 the console window or a printer renders text. 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 interactive shell, but they work properly in a command shell. Listing 2.9 (specialchars.py) prints some strings containing some of these control codes.

Listing 2.9: specialchars.py

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

When executed in a command shell, Listing 2.9 (specialchars.py) produces

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

On many systems, the computer's speaker beeps five times 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 2.10 (escapequotes.py) shows the various ways in which quotation marks may be embedded within string literals.

Listing 2.10: escapequotes.py

```
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 2.10 (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 2.11 (printpath.py) prints a string with embedded backslashes.

Listing 2.11: `printpath.py`

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

Listing 2.11 (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 2.12 (usinginput.py) demonstrates that the `input` function produces a string value.

Listing 2.12: `usinginput.py`

```
print('Please enter some text:')
x = input()
print('Text entered:', x)
print('Type:', type(x))
```

The following shows a sample run of Listing 2.12 (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 2.13 (addintegers.py) shows how to obtain an integer from the user.

Listing 2.13: addintegers.py

```
print('Please enter an integer value:')
x = input()
print('Please enter another integer value:')
y = input()
num1 = int(x)
num2 = int(y)
print(num1, '+', num2, '=', num1 + num2)
```

A sample run of Listing 2.13 (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 not 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

```
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`. We can express Listing 2.13 (addintegers.py) more compactly using this form of the `input` function as shown in Listing 2.14 (addintegers2.py).

Listing 2.14: addintegers2.py

```
x = input('Please enter an integer value: ')
y = input('Please enter another integer value: ')
num1 = int(x)
num2 = int(y)
print(num1, '+', num2, '=', num1 + num2)
```

Listing 2.15 (addintegers3.py) is even shorter. It combines the `input` and `int` functions into one statement.

Listing 2.15: addintegers3.py

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

In Listing 2.15 (addintegers3.py) the expression

```
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 2.14 (addintegers2.py). We frequently will use functional composition to make our program code simpler.

Be careful about code such as

```
num = int(input('Please enter a number: '))
```

This statement expects the user to enter an integer value. If the user types 3, for example, all is well. The variable num then will refer to the integer object 3. The int function can convert the string '3' to the integer value 3. The word *number* is ambiguous, however, so the user might attempt to enter 3.4. In this case the input statement would return the string '3.4'. The int function cannot convert the string '3.4' to an integer directly, even though it can convert the floating-point number 3.4 to the integer 3. The following interactive sequence demonstrates:

```
>>> num = int(input('Please enter a number: '))
Please enter a number: 3
>>> num
3
>>> num = int(input('Please enter a number: '))
Please enter a number: 3.4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '3.4'
>>> int(3.4)
3
>>> int('3.4')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '3.4'
```

This example reveals that the int can convert a string to an integer only if the string looks exactly like an integer.

We could be more specific and simply request an integer value from the user. If we really want an integer from the user but want to tolerate decimal places, we could use an additional function in our composition of functions, as in the following:

```
>>> num = int(float(input('Please enter a number: ')))
Please enter a number: 3
>>> num
3
>>> num = int(float(input('Please enter a number: ')))
Please enter a number: 3.4
>>> num
3
```

The assignment statement here uses the input function to obtain the number from the user as a string. It then uses the float function to convert the received string to a floating-point number. Finally it converts the floating-point number to an integer via the int function. What if you wish to round the user's input value instead of truncating it? The following function composition would work in that case:

```
>>> num = round(float(input('Please enter a number: ')))
Please enter a number: 3.7
>>> num
4
```

2.7 Controlling the `print` Function

In Listing 2.13 (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 different 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 2.16 (printingexample.py).

Listing 2.16: `printingexample.py`

```
print('A', end='')
print('B', end='')
print('C', end='')
print()
print('X')
print('Y')
print('Z')
```


Listing 2.16 (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 2.17 (printsep.py) shows the `sep` keyword customizes `print`'s behavior.

Listing 2.17: `printsep.py`

```
w, x, y, z = 10, 15, 20, 25
print(w, x, y, z)
print(w, x, y, z, sep=',')
print(w, x, y, z, sep='')
print(w, x, y, z, sep=':')
print(w, x, y, z, sep='-----')
```

The output of Listing 2.17 (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.

2.8 String Formatting

Consider Listing 2.18 (powers10left.py) which prints the first few powers of 10.

Listing 2.18: `powers10left.py`

```
print(0, 10**0)
print(1, 10**1)
```



```
print(2, 10**2)
print(3, 10**3)
print(4, 10**4)
print(5, 10**5)
print(6, 10**6)
print(7, 10**7)
print(8, 10**8)
print(9, 10**9)
print(10, 10**10)
print(11, 10**11)
print(12, 10**12)
print(13, 10**13)
print(14, 10**14)
print(15, 10**15)
```

Listing 2.18 (powers10left.py) prints

```
0 1
1 10
2 100
3 1000
4 10000
5 100000
6 1000000
7 10000000
8 100000000
9 1000000000
10 10000000000
11 100000000000
12 1000000000000
13 10000000000000
14 100000000000000
15 1000000000000000
```

Observe that each number is left justified.

Next, consider Listing 2.19 (powers10left2.py) which again prints the first few powers of 10, albeit in most complicated way.

Listing 2.19: powers10left2.py

```
print('{0} {1}'.format(0, 10**0))
print('{0} {1}'.format(1, 10**1))
print('{0} {1}'.format(2, 10**2))
print('{0} {1}'.format(3, 10**3))
print('{0} {1}'.format(4, 10**4))
print('{0} {1}'.format(5, 10**5))
print('{0} {1}'.format(6, 10**6))
print('{0} {1}'.format(7, 10**7))
print('{0} {1}'.format(8, 10**8))
print('{0} {1}'.format(9, 10**9))
print('{0} {1}'.format(10, 10**10))
print('{0} {1}'.format(11, 10**11))
print('{0} {1}'.format(12, 10**12))
print('{0} {1}'.format(13, 10**13))
```



```
print('{0} {1}'.format(14, 10**14))
print('{0} {1}'.format(15, 10**15))
```

Listing 2.19 (powers10left2.py) produces output identical to Listing 2.18 (powers10left.py):

```
0 1
1 10
2 100
3 1000
4 10000
5 100000
6 1000000
7 10000000
8 100000000
9 1000000000
10 10000000000
11 100000000000
12 1000000000000
13 10000000000000
14 100000000000000
15 1000000000000000
```

The third print statement in Listing 2.19 (powers10left2.py) prints the expression

```
'{0} {1}'.format(2, 10**2)
```

This expression has two main parts:

- `'{0} {1}'`: This is known as the *formatting string*. It is a Python string because it is a sequence of characters enclosed with quotes. Notice that the program at no time prints the literal string {0} {1}. This formatting string serves as a pattern that the second part of the expression will use. `{0}` and `{1}` are placeholders, known as *positional parameters*, to be replaced by other objects. This formatting string, therefore, represents two objects separated by a single space.
- `format(2, 10**2)`: This part provides arguments to be substituted into the formatting string. The first argument, 2, will take the position of the `{0}` positional parameter in the formatting string. The value of the second argument, `10**2`, which is 100, will replace the `{1}` positional parameter.

The `format` operation matches the `2` with the position marked by `{0}` and the `10**2` with the position marked by `{1}`. This somewhat complicated expression evaluates to the simple string `'2 100'`. The `print` function then prints this string as the first line of the program's output.

In the statement

```
print('{0} {1}'.format(7, 10**7))
```

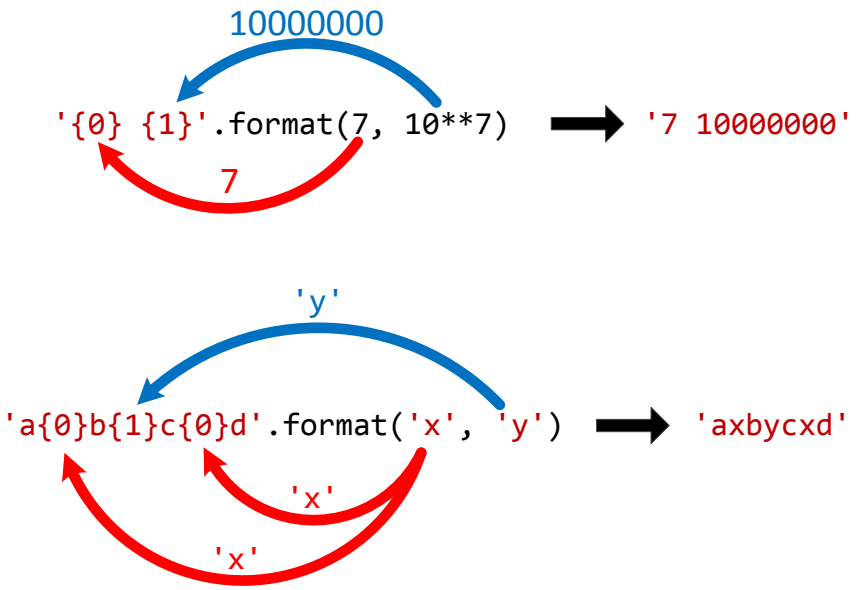
the expression to print, namely

```
'{0} {1}'.format(7, 10**7)
```

becomes '7 10000000', since 7 replaces `{0}` and $10^7 = 10000000$ replaces `{1}`. Figure 2.8 shows how the arguments of `format` substitute for the positional parameters in the formatting string.

Listing 2.19 (powers10left2.py) provides no advantage over Listing 2.18 (powers10left.py), and it is more complicated. Is the extra effort of string formatting ever useful? Observe that in both programs each

Figure 2.8 Placeholder substitution within a formatting string



number printed is left justified. Ordinarily we want numeric values appearing in a column to be right-justified so they align on the right instead of the left. A positional parameter in the format string provides options for right-justifying the object that takes its place. Listing 2.20 (powers10right.py) uses a *string formatter* with enhanced positional parameters to right justify the values it prints.

Listing 2.20: powers10right.py

```
print('{0:>3} {1:>16}'.format(0, 10**0))
print('{0:>3} {1:>16}'.format(1, 10**1))
print('{0:>3} {1:>16}'.format(2, 10**2))
print('{0:>3} {1:>16}'.format(3, 10**3))
print('{0:>3} {1:>16}'.format(4, 10**4))
print('{0:>3} {1:>16}'.format(5, 10**5))
print('{0:>3} {1:>16}'.format(6, 10**6))
print('{0:>3} {1:>16}'.format(7, 10**7))
print('{0:>3} {1:>16}'.format(8, 10**8))
print('{0:>3} {1:>16}'.format(9, 10**9))
print('{0:>3} {1:>16}'.format(10, 10**10))
print('{0:>3} {1:>16}'.format(11, 10**11))
print('{0:>3} {1:>16}'.format(12, 10**12))
print('{0:>3} {1:>16}'.format(13, 10**13))
print('{0:>3} {1:>16}'.format(14, 10**14))
print('{0:>3} {1:>16}'.format(15, 10**15))
```

Listing 2.20 (powers10right.py) prints

```
  0                1
  1               10
```

```
 2             100
 3            1000
 4           10000
 5          100000
 6         1000000
 7        10000000
 8       100000000
 9      1000000000
10     10000000000
11    100000000000
12   1000000000000
13  10000000000000
14 100000000000000
15 1000000000000000
```

The positional parameter `{0:>3}` means "right-justify the first argument to `format` within a width of three characters." Similarly, the `{1:>16}` positional parameter indicates that `format`'s second argument is to be right justified within 16 places. This is exactly what we need to properly align the two columns of numbers.

The format string can contain arbitrary text amongst the positional parameters. Consider the following interactive sequence:

```
>>> print('$${0}//{1}&&{0}^ ^ ^{2}abc'.format(6, 'Fred', 4.7))
$$6//Fred&&6^ ^ ^4.7abc
```

Note how the resulting string is formatted exactly like the format string, including spaces. The only difference is the `format` arguments replace all the positional parameters. Also notice that we may repeat a positional parameter multiple times within a formatting string.

2.9 Multi-line Strings

A Python string ordinarily spans a single line of text. The following statement is illegal:

```
x = 'This is a long string with
several words'
```

A string literal that begins with a `'` or `"` must be terminated with its matching `'` or `"` on the same line in which it begins. As we saw in Section 2.5), we can add newline control codes to produce line breaks within the string:

```
x = 'This is a long string with\nseveral words'
```

This technique, however, obscures the programmer's view of the string within the source code. Python provides way to represent a string's layout more naturally within source code, using *triple quotes*. The triple quotes (`'''` or `"""`) delimit strings that can span multiple lines in the source code. Consider Listing 2.21 (multilinestring.py) that uses a multi-line string.

Listing 2.21: `multilinestring.py`

```
x = '''
This is a multi-line
```



```
    string that goes on
for three lines!
'''
print(x)
```

Listing 2.21 (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. For a fancier example, consider the following two-dimensional rendition of a three-dimensional cube, rendered with characters:

Listing 2.22: charactercube.py

```
x = '''
    A cube has 8 corners:

           7------8
          /|     /|
         3------4 |
         | |    | |
         | 5----|-6
         |/     |/
         1------2
'''
print(x)
```

Listing 2.22 (charactercube.py) displays

```
    A cube has 8 corners:

           7------8
          /|     /|
         3------4 |
         | |    | |
         | 5----|-6
         |/     |/
         1------2
```

The "picture" in the source code looks like the picture on the screen.

We will see in Section 7.3 how Python's multi-line strings play a major role in source code documentation.

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 can you express the literal value 0.00000000000000000000000000449 as a much more compact Python literal?

13. How can you express the literal value 5699234120000000000000000000000000000 as a much more compact 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 simple *expressions*. We can use operators to combine values and variables and 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 we can use the addition operator (+) to add two integers provided by the user.

Listing 3.1: `adder.py`

```
value1 = int(input('Please enter a number: '))
value2 = int(input('Please enter another number: '))
sum = value1 + value2
print(value1, '+', value2,  '=', sum)
```

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

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

  This statement prompts the user to enter some information. After displaying the prompt string *Please enter a number:*, 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 `int` function which produces an integer 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.

- `value2 = int(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

Table 3.1 Commonly used Python arithmetic binary operators

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

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 is 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.

Table 3.1 contains the most commonly used Python arithmetic operators. 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 Python's floating point values.

When we apply the +, -, *, //, %, or ** operators to two integers, the result is an integer. The statement

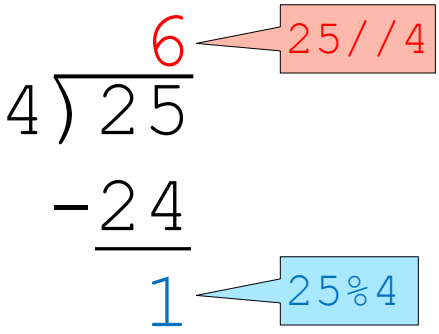
```
print(25//4, 4//25)
```

prints

```
6 0
```

The // operator produces an integer result when used with integers. In the first case above 25 divided by 4 is 6 with a remainder of 1, and in the second case 4 divided by 25 is 0 with a remainder of 4. Since integers are whole numbers, the // operator discards any fractional part of the answer. The process of discarding the fractional part of a number leaving only the whole number part is called *truncation*. Truncation is not rounding; for example, 13 divided by 5 is 2.6, but 2.6 truncates to 2.

Figure 3.1 Integer division and modulus



Truncation simply removes any fractional part of the value. It does not round. Both 10.01 and 10.999 truncate to 10.

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

```
print(25%4, 4%25)
```

prints

```
1 4
```

since 25 divided by 4 is 6 with a remainder of 1, and 4 divided by 25 is 0 with a remainder of 4. Figure 3.1 shows the relationship between integer division and modulus.

The modulus operator is more useful than it may first appear. Listing 3.9 (timeconv.py) shows how it can be used to convert a given number of seconds to hours, minutes, and seconds.

The / operator applied to two integers produces a floating-point result. The statement

```
print(25/4, 4/25)
```

prints

```
6.25 0.16
```

These results are what we would expect from a hand-held calculator. Floating-point arithmetic always produces a floating-point result.

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`

```
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 nonrepeating decimal numbers can be a problem. One-tenth (0.1) is obviously nonrepeating, so we can express it 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`

```
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.

The Python interpreter also automatically joins long statements spread over multiple lines in the source code if it dectects an opening parenthesis (, square bracket [, or curly brace { that is unmatched by its corresponding closing symbol. The following is a legal Python statement spread over two lines in the source code:

```
x = (int(input('Please enter an integer'))
    + (y - 2) + 16) * 2
```


The last closing parenthesis on the second line matches the first opening parenthesis on the first line. No backslash symbol is required at the end of the first line. The interpreter will begin scanning the first line, matching closing parentheses with opening parentheses. When it gets to the end of the line and has not detected a closing parenthesis to match an earlier opening parenthesis, the interpreter assumes it must appear on a subsequent line, and so continues scanning until it completes the long statement. If the interpreter does not find expected closing parenthesis in a program, it issues a error. In the Python interactive shell, the interpreter keeps waiting until the user complies or otherwise types something that causes an error:

```
>>> y = 10
>>> x = (int(input('Please enter an integer: '))
... + (y - 2) + 16
...
...
...
...
...
... ) * 2
Please enter an integer: 3
>>> x
54
```

Since computers represent floating-point values internally in binary form, if we choose a binary fractional power, the mathematics will work out precisely. Python can represent the fraction $\frac{1}{4} = 0.25 = 2^{-2}$ exactly. Listing 3.4 (precise4.py) illustrates.

Listing 3.4: precise4.py

```
one = 1.0
one_fourth = 1.0/4.0
zero = one - one_fourth - one_fourth - one_fourth - one_fourth
print('one =', one, '  one-fourth =', one_fourth, '  zero =', zero)
```

Listing 3.4 (precise4.py) behaves much better than the previous examples:

```
ne = 1.0   one-fourth = 0.25   zero = 0.0
```

Our computed zero actually is zero.

When should you use integers and when should you use floating-point numbers? A good rule of thumb is this: use integers to count things and use floating-point numbers for quantities obtained from a measuring device. As examples, we can measure length with a ruler or a laser range finder; we can measure volume with a graduated cylinder or a flow meter; we can measure mass with a spring scale or triple-beam balance. In all of these cases, the accuracy of the measured quantity is limited by the accuracy of the measuring device and the competence of the person or system performing the measurement. Environmental factors such as temperature or air density can affect some measurements. In general, the degree of inexactness of such measured quantities is far greater than that of the floating-point values that represent them.

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.

3.2 Mixed Type Expressions

Expressions may contain mixed integer and floating-point elements; for example, in the following program fragment

```
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 the variable `sum` bind to a floating-point value.

3.3 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, a Python programmer can use parentheses to override the precedence rules and force addition to be performed before multiplication. The expression

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.

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

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

evaluates to 20. The parentheses in a Python arithmetic expression may be arranged and nested in any ways that are acceptable in standard arithmetic.

To see how associativity works, consider the expression

```
2 - 3 - 4
```

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, $(-1)-4=-5$), or rather is

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

(that is, $2-(-1)=2+1=3$) 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 and exponentiation are left associative.

As in the case of precedence, we can use parentheses 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.

The assignment operator is a different kind of operator from the arithmetic operators. Programmers use the assignment operator only to build assignment *statements*. Python does not allow the assignment operator to be part of a larger expression or part of another statement. As such, the notions of precedence

and associativity do not apply in the context of the assignment operator. Python does, however, support a special kind of assignment statement called *chained assignment*. The code

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

assigns the value of the rightmost variable (in this case `z`) to all the other variables (`w`, `x`, and `y`) to its left. To initialize several variables to zero in one statement, you can write

```
sum = count = 0
```

which is slightly shorter than tuple assignment:

```
sum, count = 0, 0
```

3.4 Formatting Expressions

Python offers considerable flexibility for formatting arithmetic expressions; for example, suppose a program is using the variables `x` and `y` to hold data, and we wish to use those variables within a mathematical expression like the following from algebra:

$$3x + 2y - 5$$

Unlike algebra, Python has no implicit multiplication. This means we must write $3x$ as `3*x`. We may not omit the `*` operator.

We can print the value of the complete expression above as follows:

```
print(3*x + 2*y - 5)
```

This formatting is preferred, as it follows closely the style used in mathematics. The following is equally acceptable to the Python interpreter:

```
print(3*x+2*y-5)
```

Note the lack of spaces. Most agree that this formatting, while legal Python, is less readable by humans. Since people develop and read source code, human readability is important. Some suggest that spaces be used around *all* binary operators, as in the following:

```
print(3 * x + 2 * y - 5)
```

In this case such a practice reduces the readability, since it make it appear as the operators have equal precedence (which they do not—spacing in an expression does not influence operator precedence). The first way we wrote the expression natrually grouped together the factors within a term, just as is common in algebra. The following is computationally equivalent in Python but misleading and should be avoided:

```
print(3 * x+2 * y-5)
```

Again, spacing does not affect operator precedence, but this formatting makes it appear that the addition and the subtraction will take place *before* the multiplication. Psychologically, the lack of space makes it appear that `+` and `-` "bind" their operands more tightly than does `*`. This is not true, and this statement is easily misinterpreted by a human reader.

The bottom line is that you should strive to format your Python source code in a way that enhances human readability. Why is this important? Teams of programmers develop commercial software. They

must be able to review and revise code written by others. Any coding techniques that make it easier for people to read and understand each other's code greatly facilitates the development process.

If you really need to add and subtract in the expression above before performing the multiplications, you can use parentheses to override the normal precedence rules, as in:

```
print(3 * (x + 2) * (y - 5))
```

The following formatting:

```
print(3*(x + 2)*(y - 5))
```

better presents the expression as it you would find it in a algebra book.

3.5 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:

```
# 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:

```
  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:

```
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:

```
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.

3.6 Errors

Beginning programmers make mistakes writing programs because of inexperience in programming in general or due to 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 exceptions, and logic errors.

3.6.1 Syntax Errors

The interpreter is designed to execute all valid Python programs. The interpreter reads the Python source file and translates it into a executable form. This is the *translation phase*. If the interpreter detects an invalid program statement 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; Listing 3.5 (error.py) attempts such an assignment.

Listing 3.5: `error.py`

```
y = 5
x = y + 2
y + 2 = x
```

When runningListing 3.5 (error.py) the interpreter displays

```
  File "error.py", line 3
    y + 2 = x
    ^
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

```
>>> x = )3 + 4)
  File "<stdin>", line 1
    x = )3 + 4)
        ^
SyntaxError: invalid syntax
```

or mismatched string quotes

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

or faulty indentation.

```
>>> x = 2
>>>  y = 5
  File "<stdin>", line 1
    y = 5
    ^
IndentationError: unexpected indent
```

These examples illustrate just a few of the ways programmers can write ill-formed code.

The interpreter detects syntax errors before it begins running the program, and so it will not execute any parts of a program that contains syntax errors.

3.6.2 Run-time Exceptions

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 exceptions* or *run-time errors*. We say the interpreter *raises* an exception. Run-time exceptions arise after the interpreter's translation phase and during the program's execution phase.

The interpreter may issue an exception 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 3.6 (dividedanger.py) which contains an error that manifests itself only in one particular situation.

Listing 3.6: dividedanger.py

```
#  File dividedanger.py

#  Get two integers from the user
print('Please enter two numbers to divide.')
dividend = int(input('Please enter the dividend: '))
divisor = int(input('Please enter the divisor: '))
#  Divide them and report the result
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 numbers to divide.
Please enter the dividend: 32
Please enter the divisor: 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.
Please enter the dividend: 32
Please enter the divisor: 0
Traceback (most recent call last):
  File "C:\Users\rick\Desktop\changeable.py", line 8, 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 3.7 (halve.py).

Listing 3.7: halve.py

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

Some sample runs of Listing 3.7 (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 = int(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 12 we will examine techniques that allow programmers to avoid these kinds of problems.

The interpreter detects syntax errors immediately. Syntax errors never make it out of the translation phase. Sometimes run-time exceptions do not reveal themselves immediately. The interpreter issues a run-time exception only when it attempts to execute the faulty statement. 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 does not get a chance to execute. This means the error may lie undetected until a user stumbles upon it after the software is deployed. Run-time exceptions, therefore, are more troublesome than syntax errors.

3.6.3 Logic Errors

The interpreter can detect syntax errors during the translation phase and uncover run-time exceptions during the execution phase. Both kinds of problems 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 3.6 (dividedanger.py) with the expression:

```
divisor/dividend
```

The program runs, and unless the user enters a value of zero for the dividend, the interpreter will report no errors. However, the answer it computes is not correct in general. The only time the program will print the correct answer is when `dividend` equals `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 3.11 (faultytempconv.py) is an example of a program that contains a logic error. Listing 3.11 (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. Programmers frequently use tools such as debuggers to help them 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 (exceptions) 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 reveal themselves only 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 of 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.

3.7 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 3.8 (tempconv.py) implements the conversion in Python.

Listing 3.8: tempconv.py

```
#  File tempconv.py
#  Author: Rick Halterman
#  Last modified: August 22, 2014
#  Converts degrees Fahrenheit to degrees Celsius
#  Based on the formula found at
#  http://en.wikipedia.org/wiki/Conversion_of_units_of_temperature

#  Prompt user for temperature to convert and read the supplied value
degreesF = float(input('Enter the temperature in degrees F: '))
#  Perform the conversion
degreesC = 5/9*(degreesF - 32)
#  Report the result
print(degreesF, 'degrees F =', degreesC, 'degrees C')
```

Listing 3.8 (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 3.9 (timeconv.py) uses integer division and modulus to split up a given number of seconds to hours, minutes, and seconds.

Listing 3.9: timeconv.py

```
#  File timeconv.py

#  Get the number of seconds
seconds = int(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, "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 statement assigns back to the `seconds` variable the remainder of `seconds` divided by 3,600. 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 3.9 (timeconv.py), Listing 3.10 (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 3.10 (enhancedtimeconv.py) displays it as `3:04:05`. It is trivial to modify Listing 3.9 (timeconv.py) so that it would print `3:4:5`, but Listing 3.10 (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 3.10: enhancedtimeconv.py

```
#  File enhancedtimeconv.py

#  Get the number of seconds
seconds = int(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 3.10 (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.

3.8 More Arithmetic Operators

As Listing 3.10 (enhancedtimeconv.py) demonstrates, an executing program can alter a variable's value by performing some arithmetic on its current value. 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 we need to modify a variable with a long name; 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.

3.9 Algorithms

Have you ever tried to explain to someone how to perform a reasonably complex task? The task could involve how to make a loaf of bread from scratch, how to get to the zoo from city hall, or how to factor an algebraic expression. Were you able to explain all the steps perfectly without omitting any important details critical to the task's solution? Were you frustrated because the person wanting to perform the task obviously was misunderstanding some of the steps in the process, and you believed you were making everything perfectly clear? Have you ever attempted to follow a recipe for your favorite dish only to discover that some of the instructions were unclear or ambiguous? Have you ever faithfully followed the travel directions provided by a friend and, in the end, found yourself nowhere near the intended destination?

Often it is easy to envision the steps to complete a task but hard to communicate precisely to someone else how to perform those steps. We may have completed the task many times, or we even may be an expert on completing the task. The problem is that someone who has never completed the task requires exact, detailed, unambiguous, and complete instructions to complete the task successfully.

Because many real-world tasks involve a number of factors, people sometimes get lucky and can complete a complex task given less-than-perfect instructions. A person often can use experience and common sense to handle ambiguous or incomplete instructions. If fact, humans are so good at dealing with "fuzzy" knowledge that in most instances the effort to produce excruciatingly detailed instructions to complete a task is not worth the effort.

When a computer executes the instructions found in software, it has no cumulative experience and no common sense. It is a slave that dutifully executes the instructions it receives. While executing a program a computer cannot fill in the gaps in instructions that a human naturally might be able to do. Further, unlike with humans, executing the same program over and over does not improve the computer's ability to perform the task. The computer has no *understanding*.

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

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

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

Armed with this knowledge, Listing 3.11 (faultytempconv.py) follows directly.

Listing 3.11: `faultytempconv.py`

```
#  File faultytempconv.py

#  Establish some variables
degreesF, degreesC = 0, 0
#  Define the relationship between F and C
```

```
degreesC = 5/9*(degreesF - 32)
#  Prompt user for degrees F
degreesF = float(input('Enter the temperature in degrees F: '))
#  Report the result
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 program assigns variable `degreesC` *before* it receives `degreesF`'s value 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 we must specify algorithms 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 5 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 the algorithms that take advantage of them are more complex. 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.

3.10 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. What circumstances can cause each of the following run-time errors to arise?

    - `NameError`
    - `ValueError`
    - `ZeroDivisionError`
    - `IndentationError`
    - `OverflowError`
    - `SyntaxError`
    - `TypeError`

    Hint: Try some of following activities in the interpreter or within a Python program:

    - print a variable that has not been assigned
    - convert the string `'two'` to an integer
    - add an integer to a string
    - assign to a variable named `end-point`
    - experiment adding spaces and tabs at various places in the code of an error-free Python program
    - compute raise a floating-point number to a large power, as in $1.5^{10,000}$.

16. 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 = float(input())                  # 1
n2 = float(input())                  # 2
#  Compute sum of the two numbers
print(n1 + n2)                       # 3
#  Compute average of the two numbers
print(n1+n2/2)                       # 4
#  Assign some variables
d1 = d2 = 0                          # 5
#  Compute a quotient
print(n1/d1)                         # 6
#  Compute a product
n1*n2 = d1                           # 7
#  Print result
print(d1)                            # 8
```

    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.

17. 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`

18. 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?

19. 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 = float(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?

20. Write a Python program that ...

21. 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.

Listing 4.1: `boolvars.py`

```
#  Assign some Boolean variables
a = True
```


Table 4.1 The Python relational operators

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 4.2 Examples of some Simple Relational Expressions

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

```
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. The simplest kinds of Boolean expressions use *relational operators* to compare two expressions. 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.

In the Python interactive shell we see:

```
>>> x = 10
>>> x
10
>>> x < 10
```



```
False
>>> x <= 10
True
>>> x == 10
True
>>> x >= 10
True
>>> x > 10
False
>>> x < 100
True
>>> x < 5
False
```

The first input in the shell binds the variable x to the value 10. The other expressions experiment with the relational operators. Exactly matching their mathematical representations, the following expressions all are equivalent:

- `x < 10`
- `10 > x`
- `!(x >= 10)`
- `!(10 <= x)`

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

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

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 3.6 arise from logic errors. One way that Listing 3.6 (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`

```
#  File betterdivision.py

#  Get two integers from the user
```

```
print('Please enter two numbers to divide.')
dividend = int(input('Please enter the first number to divide: '))
divisor = int(input('Please enter the second number to divide: '))
#  If possible, divide them and report the result
if divisor != 0:
    print(dividend, '/', divisor, "=", dividend/divisor)
```

The program may not always execute the `print` statement. In the following run

```
Please enter two numbers to divide.
Please enter the first number to divide: 32
Please enter the second number to divide: 8
32 / 8 = 4.0
```

the program executes the `print` statement, but if the user enters a zero as the second number:

```
Please enter two numbers to divide.
Please enter the first number to divide: 32
Please enter the second number to divide: 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 optionally executes the indented section of code. In this case, the `if` statement executes the `print` statement only if the variable `divisor`'s value is not zero.

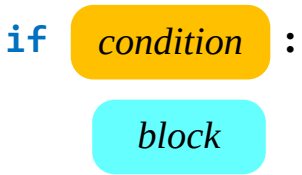
The Boolean expression

```
divisor != 0
```

determines whether or not the program will execute the statement in the indented block. If `divisor` is not zero, the program prints the message; otherwise, the program displays nothing after the provides the input.

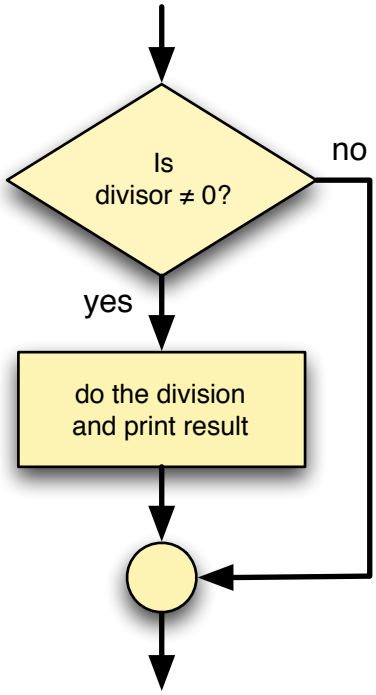
Figure 4.1 shows how program execution flows through the `if` statement. of Listing 4.2 (betterdivision.py).

The general form of the `if` statement is:



- 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. The statements within the block must all be indented the same number of spaces from the left. The block within an

Figure 4.1 if flowchart



`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`:

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

could be written

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

but may *not* be written as

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

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

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. Whichever indent distance you choose, you must use this same distance consistently throughout a Python program.

It is important not to mix spaces and tabs when indenting statements within a block. Python 3 does not allow tabs and spaces to be mixed when indenting statements. 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. Generally speaking, a *tab stop* represents a fixed distance from the left side of the editing window. Most editors support multiple tab stops; for example, the first tab stop might be eight characters from the start of the line, followed by a series of tab stops, each one at a distance of eight characters from the previous tab stop (at columns 8, 16, 24, 32, etc. within the line of text). Furthermore, most editors allow the user to reconfigure in some way the locations of the tab stops. Pressing the Tab key in the editor causes the cursor within the editing window to jump to the next tab stop within the line. The space bar, on the other hand, always moves the cursor exactly one character to the right.

Why is indentation that mixes tabs and spaces a problem and thus forbidden in Python 3? Consider creating a Python source file in one editor and then viewing it in a different editor with tab stops set differently. Lines that appear perfectly indented in the original editor would be misaligned in the new editor. Instead, code indented with four spaces within one editor would appear exactly the same in any other editor.

Python 3 does allow the use of tabs for indentation—you just cannot mix them with spaces within the same source file. Most programming editors have a setting to substitute automatically a specified number of spaces when the user presses the Tab key. For Python development you should use this feature. Python best practice prefers using spaces over tabs for indentation within Python source.

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`

```
#  Get two integers from the user
dividend = int(input('Please enter the number to divide: '))
divisor = int(input('Please enter dividend: '))
#  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

```
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 treats every other integer value, positive and negative, 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 nonempty 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

```
#  Request input from the user
num = int(input("Please enter an integer in the range 0...9999: "))

#  Attenuate the number if necessary
if num < 0:          # Make sure number is not too small
    num = 0
if num > 9999:       # Make sure number is not too big
    num = 9999

print(end="[")       # Print left brace

# Extract and print thousands-place digit
digit = num//1000    # Determine the thousands-place digit
print(digit, end="") # Print the thousands-place digit
num %= 1000          # Discard thousands-place digit

# Extract and print hundreds-place digit
digit = num//100     # Determine the hundreds-place digit
print(digit, end="") # Print the hundreds-place digit
num %= 100           # Discard hundreds-place digit

# Extract and print tens-place digit
digit = num//10      # Determine the tens-place digit
print(digit, end="") # Print the tens-place digit
num %= 10            # Discard tens-place digit

#  Remainder is the one-place digit
print(num, end="")   # Print the ones-place digit

print("]")           # Print right brace
```

A sample run of Listing 4.4 (leadingzeros.py) produces

```
Please enter an integer in the range 0...9999: 38
[0038]
```

Another run demonstrates the effects of a user entering a negative number:

```
Please enter an integer in the range 0...9999: -450
[0000]
```

The program attenuates numbers that are too large:

```
Please enter an integer in the range 0...9999: 3256670
[9999]
```

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, the program prints nothing. 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` block 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`

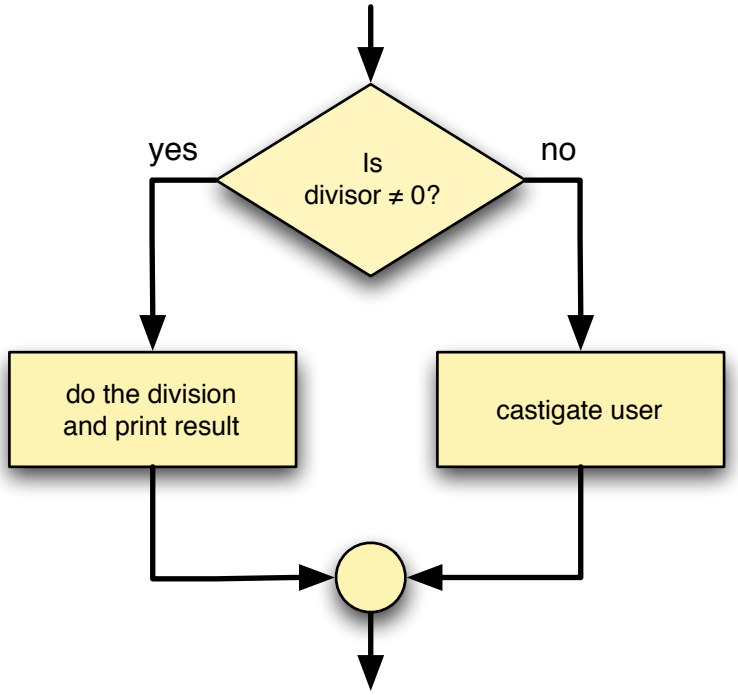
```
#  Get two integers from the user
dividend = int(input('Please enter the number to divide: '))
divisor = int(input('Please enter dividend: '))
#  If possible, divide them and report the result
if divisor != 0:
    print(dividend, '/', divisor, "=", dividend/divisor)
else:
    print('Division by zero 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 Listing 4.2 (betterdivision.py), this program always displays a message:

```
Please enter the number to divide: 32
Please enter dividend: 0
Division by zero is not allowed
```

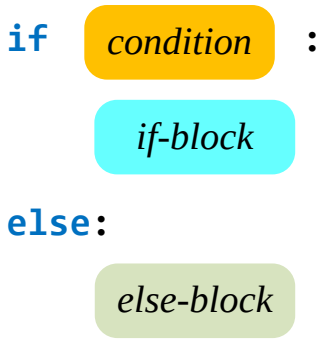
The `else` block contains an alternate block of code that the program executes when the condition is false. Figure 4.2 illustrates the program's flow of execution.

Figure 4.2 if/else flowchart



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



- 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.
- 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 one level deeper 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 one level deeper than the line with the `else`. 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.

4.5 Compound Boolean Expressions

We can combine simple Boolean expressions, each involving one relational operator, 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.

Table 4.3 Logical operators—e_1 and e_2 are Boolean expressions

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

The Python logical `and` operator works in exactly the same way. Suppose 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.

Related to the logical `and` operator is the logical `or` operator. To illustrate the logical `or` operator, consider two mathematics courses, *Differential Equations* and *Linear Algebra*. A computer science degree requires at least 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 the `or` operator 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 logical `not` operator 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. In mathematics, if the expression $x = y$ is false, it must be true that $x \neq y$. In Python, the expression `not (x == y)` is equivalent to the expression `x != y`. If also is the case that the Python expresion `not (x != y)` is just a more complicated way of expressing `x == y`. In mathematics, if the expression $x < y$ is false, it must be the case that $x \geq y$. In Python, `not (x < y)` has the same truth value as `x >= y`. The expression `not (x >= y)` is equivalent to `x < y`. You may be able to see from these examples that if e is a Boolean expression, it always is true that `not not` e is equivalent to e (this is known as the *double negative* property of mathematical logic).

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. The `not` operator is a unary operator (see Section 3.1); it requires a single truth expression immediately to its right.

Table 4.4 lists the Python operators we have seen so far. Table 4.4 shows that operator `not` has higher precedence than both `and` and `or`. The `and` operator has higher precedence than `or`. Both the `and` and `or` operators are left associative; `not` is right associative. The `and` and `or` operators 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)
```


Table 4.4 Precedence of Some Python Operators. Higher precedence operators appear above lower precedence operators.

Arity	Operators	Associativity
binary	`**`	
unary	`+, -`	
binary	`*, /, //, %`	left
binary	`+, -`	left
binary	`>, <, >=, <=, ==, !=`	left
unary	`not`	
binary	`and`	left
binary	`or`	left

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 interpreted 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

```
not (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 e_2's value is irrelevant—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. In the context of the `and` operator, if its left operand is `False`, the more more expensive right operand need not be evaluated. In the context of the `or` operator, if the left operand is `True`, the more expensive right operand may be ignored.

  As a simple example, consider the following Python code snippet that could be part of a larger program:

  ```
  if x < 10 and input("Print value (y/n)?") == 'y':
      print(x)
  ```

  If `x` is a numeric value less than 10, this statement will query the user to print or not print the value of `x`. If `x` $\geq$ 10, the program need not stop and wait for the user's input. If `x` $\geq$ 10, the user's input is superfluous anyway. Now consider the statement with the Boolean expressions ordered the other way:

  ```
  if input("Print value (y/n)?") == 'y' and x < 10:
      print(x)
  ```

  In this case as well, both subconditions must be true to print the value of `x`. The difference here is that the program always pauses its execution to accept the user's input regardless of `x`'s value. This statement bothers the user for input even when the second subcondition ensures the user's answer will make no difference.

- Subexpressions may 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.

4.6 The pass Statement

Some beginning programmers attempt to use an `if/else` statement when a simple `if` statement is more appropriate; for example, in the following code fragment the programmer wishes to do nothing if the value of the variable `x` is less than zero; otherwise, the programmer wishes to print `x`'s value:

```
if x < 0:
    # Do nothing    (This will not work!)
else:
    print(x)
```

If the value of x is less than zero, this section of code should print nothing. Unfortunately, the code fragment above is not legal Python. The if/else statement contains an else block, but it does not contain an if block. The comment does not count as a Python statement. Both if and if/else statements require an if block that contains at least one statement. Additionally, an if/else statement requires an else block that contains at least one statement.

Python has a special statement, pass, that means *do nothing*. We may use the pass statement in our code in places where the language requires a statement to appear but we wish the program to take no action whatsoever. We can make the above code fragment legal by adding a pass statement:

```
if x < 0:
    pass  # Do nothing
else:
    print(x)
```

While the pass statement makes the code legal, we can express its logic better by using a simple if statement. In mathematics, if the expression $x < y$ is false, it must be the case that $x \geq y$. If we invert the truth value of the relation within the condition, we can express the above code more succinctly as

```
if x >= 0:
    print(x)
```

So, if you ever feel the need to write an if/else statement with an empty if body, do the following instead:

1. invert the truth value of the condition
2. make the proposed else body the if body
3. eliminate the else

In situations where you may be tempted to use a non-functional else block, as in the following:

```
if x == 2:
    print(x)
else:
    pass     # Do nothing if x is not equal to 2
```

do not alter the condition but simply eliminate the else and the else block altogether:

```
if x == 2:
    print(x)   # Print only if x is equal to 2
```

The pass statement in Python is useful for holding the place for code to appear in the future; for example, consider the following code fragment:

```
if x < 0:
    pass  # TODO: print an appropriate warning message to be determined
else:
    print(x)
```


In this code fragment the programmer intends to provide an `if` block, but the exact nature of the code in the `if` block is yet to be determined. The `pass` statement serves as a suitable placeholder for the future code. The included comment documents what is expected to appear eventually in place of the `pass` statement.

We will see other uses of the `pass` statement as we explore Python more deeply.

4.7 Floating-point Equality

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

```
d1 = 1.11 - 1.10
d2 = 2.11 - 2.10
print('d1 =', d1, ' d2 =', d2)
if d1 == d2:
    print('Same')
else:
    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 that `d1` and `d2` are different.

The solution is not to check floating-point numbers for exact equality, but rather see if the values "close enough" to each other to be considered the same. If `d1` and `d2` are two floating-point numbers, we need to check if the absolute value of the `d1 - d2` is a very small number. Listing 4.7 (floatequals.py) adapts Listing 4.6 (samedifferent.py) using this approximately equal concept.

Listing 4.7: floatequals.py

```
d1 = 1.11 - 1.10
d2 = 2.11 - 2.10
print('d1 =', d1, ' d2 =', d2)
diff = d1 - d2          # Compute difference
if diff < 0:            # Compute absolute value
    diff = -diff
if diff < 0.0000001:  # Are the values close enough?
    print('Same')
else:
    print('Different')
```


Listing 4.8 (floatequals2.py) is a variation of Listing 4.7 (floatequals.py) that does not compute the absolute value but instead checks to see if the difference is between two numbers that are very close to zero: one negative and the other positive.

Listing 4.8: floatequals2.py

```
d1 = 1.11 - 1.10
d2 = 2.11 - 2.10
print('d1 =', d1, ' d2 =', d2)
if -0.0000001 < d1 - d2 < 0.0000001:
    print('Same')
else:
    print('Different')
```

In Section 7.4.6 we will see how to encapsulate this floating-point equality code within a function to make it more convenient for general use.

4.8 Nested Conditionals

The statements in the block of the `if` or the `else` may be any Python statements, including other `if`/`else` statements. We can use these nested `if` statements to develop arbitrarily complex program logic. Consider Listing 4.9 (checkrange.py) that determines if a number is between 0 and 10, inclusive.

Listing 4.9: checkrange.py

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

Listing 4.9 (checkrange.py) behaves as follows:

- The executing program checks first condition. If `value` is less than zero, the program does not evaluate the second condition and it continues its execution with the statement following the outer `if`. The statement after the outer `if` simply prints *Done*.
- If the executing program finds the `value` variable to be greater than or equal to zero, it executes the statement within the `if`-block. This statement is itself an `if` statement. The program thus checks the second (inner) condition. If the second condition is satisfied, the program displays the *In range* message; otherwise, it does not. Regardless, the program eventually prints the *Done* message.

We say that the second `if` (with the comment *Second check*) is *nested* within the first `if` (*First check*). We call the first `if` the *outer* `if` and the second `if` the *inner* `if`. Notice the entire inner `if` statement is indented one level relative to the outer `if` statement. This means the inner `if`'s block, the `print("In range")` statement, is indented two levels deeper than the outer `if` statement. Remember that if you use four spaces as the distance for a indentation level, you must consistently use this four space distance for each indentation level throughout the program.

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 program to print the *In range*

message. From this perspective, the program can be rewritten to behave the same way with only *one* `if` statement, as Listing 4.10 (newcheckrange.py) shows.

Listing 4.10: newcheckrange.py

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

Listing 4.10 (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.

We may express the condition the `if` within Listing 4.10 (newcheckrange.py):

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

more compactly as

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

Sometimes we cannot simplify a program's logic as readily as in Listing 4.10 (newcheckrange.py). Listing 4.11 (enhancedcheckrange.py) would be impossible to rewrite with only one `if` statement.

Listing 4.11: enhancedcheckrange.py

```
value = int(input("Please enter an integer value in the range 0...10: ")
if value >= 0:        # First check
    if value <= 10:   # Second check
        print(value, "is in range")
    else:
        print(value, "is too large")
else:
    print(value, "is too small")
print("Done")
```

Listing 4.11 (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.

Computers store all data internally in binary form. The binary (base 2) number system is much simpler than the familiar decimal (base 10) number system because it uses only two digits: 0 and 1. The decimal system uses 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Despite the lack of digits, every decimal integer has an equivalent binary representation. Binary numbers use a place value system not unlike the decimal system. Figure 4.3 shows how the familiar base 10 place value system works.

With 10 digits to work with, the decimal number system distinguishes place values with powers of 10. Compare the base 10 system to the base 2 place value system shown in Figure 4.4.

With only two digits to work with, the binary number system distinguishes place values by powers of two. Since both binary and decimal numbers share the digits 0 and 1, we will use the subscript 2 to indicate a binary number; therefore, 100 represents the decimal value *one hundred*, while 100_2 is the binary number *four*. Sometimes to be very clear we will attach a subscript of 10 to a decimal number, as in 100_{10}.

Figure 4.3 The base 10 place value system

$$\begin{array}{cccccccc} \cdots & \boxed{4} & \boxed{7} & \boxed{3} & \boxed{4} & \boxed{0} & \boxed{6} \\ \cdots & 10^5 & 10^4 & 10^3 & 10^2 & 10^1 & 10^0 \\ \cdots & 100{,}000 & 10{,}000 & 1{,}000 & 100 & 10 & 1 \end{array}$$

$$\begin{aligned} 473{,}406 &= 4\times10^5+7\times10^4+3\times10^3+4\times10^2+0\times10^1+6\times10^0 \\ &= 400{,}000+70{,}000+3{,}000+400+0+6 \\ &= 473{,}406 \end{aligned}$$

Figure 4.4 The base 2 place value system

$$\begin{array}{cccccccc} \cdots & \boxed{1} & \boxed{0} & \boxed{0} & \boxed{1} & \boxed{1} & \boxed{1} \\ \cdots & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ \cdots & 32 & 16 & 8 & 4 & 2 & 1 \end{array}$$

$$\begin{aligned} 100111_2 &= 1\times2^5+0\times2^4+0\times2^3+1\times2^2+1\times2^1+1\times2^0 \\ &= 32+0+0+4+2+1 \\ &= 39 \end{aligned}$$

Listing 4.12 (binaryconversion.py) uses an `if` statement containing a series of nested `if` statements to print a 10-bit binary string representing the binary equivalent of a decimal integer supplied by the user. We use `if`/`else` statements to print the individual digits left to right, essentially assembling the sequence of bits that represents the binary number.

Listing 4.12: `binaryconversion.py`

```
#  Get number from the user
value = int(input("Please enter an integer value in the range 0...1023: "))
#  Create an empty binary string to build upon
binary_string = ''
#  Integer must be less than 1024
if 0 <= value < 1024:
    if value >= 512:
        binary_string += '1'
        value %= 512
    else:
        binary_string += '0'
    if value >= 256:
        binary_string += '1'
        value %= 256
    else:
        binary_string += '0'
    if value >= 128:
```

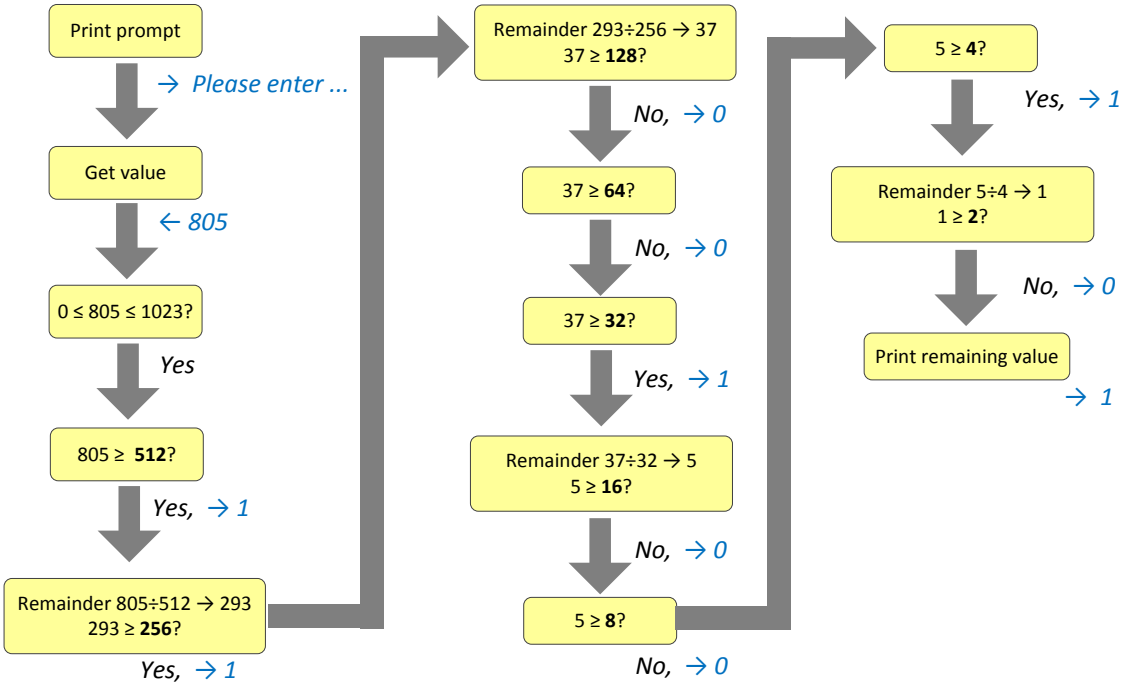
```
        binary_string += '1'
        value %= 128
    else:
        binary_string += '0'
    if value >= 64:
        binary_string += '1'
        value %= 64
    else:
        binary_string += '0'
    if value >= 32:
        binary_string += '1'
        value %= 32
    else:
        binary_string += '0'
    if value >= 16:
        binary_string += '1'
        value %= 16
    else:
        binary_string += '0'
    if value >= 8:
        binary_string += '1'
        value %= 8
    else:
        binary_string += '0'
    if value >= 4:
        binary_string += '1'
        value %= 4
    else:
        binary_string += '0'
    if value >= 2:
        binary_string += '1'
        value %= 2
    else:
        binary_string += '0'
    binary_string += str(value)

#  Display the results
if binary_string != '':
    print(binary_string)
else:
    print('Cannot convert')
```

In Listing 4.12 (binaryconversion.py):

- The outer `if` checks to see if the value the user provides is in the proper range. The program works only for numbers in the range $0 \leq$ `value` $< 1{,}024$.
- Each inner `if` compares the user-supplied entered integer against decreasing powers of two. If the number is large enough, the program:
  - prints appends the digit (actually character) 1 to the binary string under construction, and
  - removes via the remainder operator that power of two's contribution to the value.

  If the number is not at least as big as the given power of two, the program concatenates a 0 instead and moves on without modifying the input value.

Figure 4.5 The process of the binary number conversion program when the user supplies 805 as the input value.



- For the ones place at the end no check is necessary—the remaining value will be 0 or 1 and so the program appends the string version of 0 or 1.

The following shows a sample run of Listing 4.12 (binaryconversion.py):

```
Please enter an integer value in the range 0...1023: 805
1100100101
```

Figure 4.5 illustrates the execution of Listing 4.12 (binaryconversion.py) when the user enters 805.

Listing 4.13 (simplerbinaryconversion.py) simplifies the logic of Listing 4.12 (binaryconversion.py) at the expense of some additional arithmetic. It uses only one `if` statement.

Listing 4.13: simplerbinaryconversion.py

```
#  Get number from the user
value = int(input("Please enter an integer value in the range 0...1023: "))
#  Initial binary string is empty
binary_string = ''
#  Integer must be less than 1024
if 0 <= value < 1024:
    binary_string += str(value//512)
    value %= 512
    binary_string += str(value//256)
```

```
    value %= 256
    binary_string += str(value//128)
    value %= 128
    binary_string += str(value//64)
    value %= 64
    binary_string += str(value//32)
    value %= 32
    binary_string += str(value//16)
    value %= 16
    binary_string += str(value//8)
    value %= 8
    binary_string += str(value//4)
    value %= 4
    binary_string += str(value//2)
    value %= 2
    binary_string += str(value)
#  Report results
if binary_string != '':
    print(binary_string)
else:
    print('Unable to convert')
```

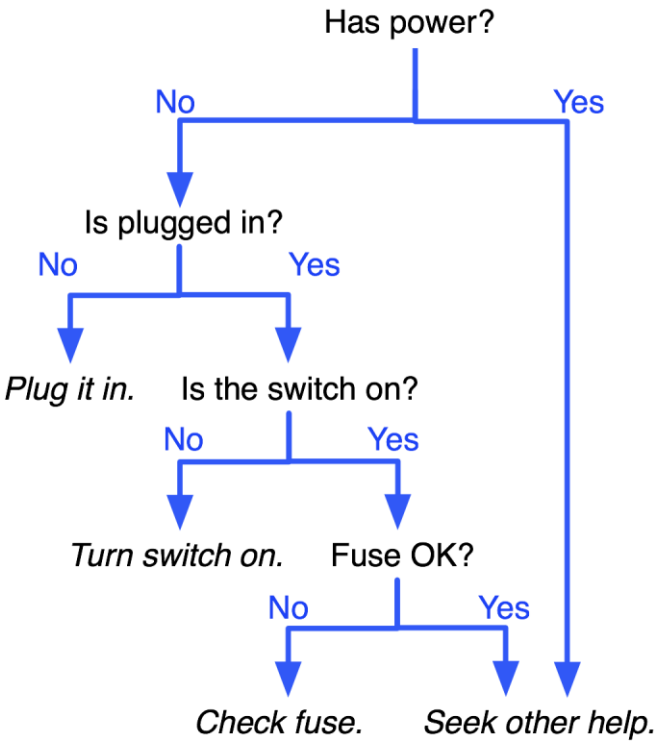
The sole `if` statement in Listing 4.13 (simplerbinaryconversion.py) ensures that the user provides an integer in the proper range. The other `if` statements that originally appeared in Listing 4.12 (binaryconversion.py) are gone. A clever sequence of integer arithmetic operations replace the original conditional logic. The two programs—binaryconversion.py and simplerbinaryconversion.py—behave identically but simplerbinaryconversion.py's logic is simpler.

Listing 4.14 (troubleshoot.py) implements a very simple troubleshooting program that an (equally simple) computer technician might use to diagnose an ailing computer.

Listing 4.14: `troubleshoot.py`

```
print("Help!  My computer doesn't work!")
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.  If the problem persists, ")
        print("please run this program again.")
    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.  If the problem persists, ")
            print("please run this program again.")
        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
                    print("Check the outlet's circuit ")
```

Figure 4.6 Decision tree for troubleshooting a computer system




```
                    print("breaker or fuse.  Move to a")
                    print("new outlet, if necessary. ")
                    print("If the problem persists, ")
                    print("please run this program again.")
                else:  # Beats me!
                    print("Please consult a service technician.")
            else: # Check fuse
                print("Check the fuse. Replace if ")
                print("necessary.  If the problem ")
                print("persists, then ")
                print("please run this program again.")
else:  # The computer has power
    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 4.6. The steps performed are:

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.

This algorithm performs the easiest checks first. It adds progressively more difficult checks as the program continues. 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 4.14 (troubleshoot.py). Visually programmers quickly can determine the logical structure of the program by the arrangement and indentation of the blocks.

Recall the time conversion program in Listing 3.9 (timeconv.py). If the user enters `10000`, the program runs as follows:

```
Please enter the number of seconds:10000
2 hr 46 min 40 sec
```

Suppose we wish to improve the English presentation by not using abbreviations. If we spell out *hours*, *minutes*, and *seconds*, we must be careful to use the singular form *hour*, *minute*, or *second* when the corresponding value is one. Listing 4.15 (timeconvcond1.py) uses `if/else` statements to express to time units with the correct number.

Listing 4.15: `timeconvcond1.py`

```
#  File timeconvcond1.py

#  Some useful conversion factors
seconds_per_minute = 60
seconds_per_hour   = 60*seconds_per_minute  # 3600

#  Get user input in seconds
seconds = int(input("Please enter the number of seconds:"))

#  First, compute the number of hours in the given number of seconds
hours = seconds // seconds_per_hour  # 3600 seconds = 1 hour
#  Compute the remaining seconds after the hours are accounted for
seconds = seconds % seconds_per_hour
#  Next, compute the number of minutes in the remaining number of seconds
minutes = seconds // seconds_per_minute  # 60 seconds = 1 minute
#  Compute the remaining seconds after the minutes are  accounted for
seconds = seconds % seconds_per_minute
#  Report the results
print(hours, end='')
# Decide between singular and plural form of hours
if hours == 1:
    print(" hour ", end='')
else:
   print(" hours ", end='')
```

```
print(minutes, end='')
# Decide between singular and plural form of minutes
if minutes == 1:
    print(" minute ", end='')
else:
    print(" minutes ", end='')
print(seconds, end='')
# Decide between singular and plural form of seconds
if seconds == 1:
    print(" second")
else:
    print(" seconds")
```

The `if`/`else` statements within Listing 4.15 (timeconvcond1.py) are responsible for printing the correct version—singular or plural—for each time unit. One run of Listing 4.15 (timeconvcond1.py) produces

```
Please enter the number of seconds:10000
2 hours 46 minutes 40 seconds
```

All the words are plural since all the value are greater than one. Another run produces

```
Please enter the number of seconds:9961
2 hours 46 minutes 1 second
```

Note the word *second* is singular as it should be.

```
Please enter the number of seconds:3601
1 hour 0 minutes 1 second
```

Here again the printed words agree with the number of the value they represent.

An improvement to Listing 4.15 (timeconvcond1.py) would not print a value and its associated time unit if the value is zero. Listing 4.16 (timeconvcond2.py) adds this feature.

Listing 4.16: `timeconvcond2.py`

```
#  File timeconvcond2.py

#  Some useful conversion constants
seconds_per_minute = 60
seconds_per_hour   = 60*seconds_per_minute  # 3600
seconds = int(input("Please enter the number of seconds:"))
#  First, compute the number of hours in the given number of seconds
hours = seconds // seconds_per_hour  # 3600 seconds = 1 hour
#  Compute the remaining seconds after the hours are accounted for
seconds = seconds % seconds_per_hour
#  Next, compute the number of minutes in the remaining number of seconds
minutes = seconds // seconds_per_minute  # 60 seconds = 1 minute
#  Compute the remaining seconds after the minutes are accounted for
seconds = seconds % seconds_per_minute
#  Report the results
if hours > 0:      # Print hours at all?
    print(hours, end='')
    # Decide between singular and plural form of hours
    if hours == 1:
```

```
        print(" hour ", end='')
    else:
        print(" hours ", end='')
if minutes > 0:   # Print minutes at all?
    print(minutes, end='')
    # Decide between singular and plural form of minutes
    if minutes == 1:
        print(" minute ", end='')
    else:
        print(" minutes ", end='')
#  Print seconds at all?
if seconds > 0 or (hours == 0 and minutes == 0 and seconds == 0):
    print(seconds, end='')
    # Decide between singular and plural form of seconds
    if seconds == 1:
        print(" second", end='')
    else:
        print(" seconds", end='')
print()  # Finally print the newline
```

In Listing 4.16 (timeconvcond2.py) each code segment responsible for printing a time value and its English word unit is protected by an `if` statement that only allows the code to execute if the time value is greater than zero. The exception is in the processing of seconds: if all time values are zero, the program should print *0 seconds*. Note that each of the `if`/`else` statements responsible for determining the singular or plural form is nested within the `if` statement that determines whether or not the value will be printed at all.

One run of Listing 4.16 (timeconvcond2.py) produces

```
Please enter the number of seconds:10000
2 hours 46 minutes 40 seconds
```

All the words are plural since all the value are greater than one. Another run produces

```
Please enter the number of seconds:9961
2 hours 46 minutes 1 second
```

Note the word *second* is singular as it should be.

```
Please enter the number of seconds:3601
1 hour 1 second
```

Here again the printed words agree with the number of the value they represent.

```
Please enter the number of seconds:7200
2 hours
```

Another run produces:

```
Please enter the number of seconds:60
1 minute
```

Finally, the following run shows that the program handles zero seconds properly:

```
Please enter the number of seconds:0
0 seconds
```

4.9 Multi-way Decision Statements

A simple `if/else` statement can select from between two execution paths. Listing 4.11 (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 4.17 (digittoword.py).

Listing 4.17: digittoword.py

```
value = int(input("Please enter an integer in the range 0...5: "))
if value < 0:
    print("Too small")
else:
    if value == 0:
        print("zero")
    else:
        if value == 1:
            print("one")
        else:
            if value == 2:
                print("two")
            else:
                if value == 3:
                    print("three")
                else:
                    if value == 4:
                        print("four")
                    else:
                        if value == 5:
                            print("five")
                        else:
                            print("Too large")
print("Done")
```

Observe the following about Listing 4.17 (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.

As a consequence of the required formatting of Listing 4.17 (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 4.18 (restyleddigittoword.py) uses the `if/elif/else` statement to avoid the rightward code drift.

Listing 4.18: restyleddigittoword.py

```
value = int(input("Please enter an integer in the range 0...5: "))
if value < 0:
```

```
    print("Too small")
elif value == 0:
    print("zero")
elif value == 1:
    print("one")
elif value == 2:
    print("two")
elif value == 3:
    print("three")
elif value == 4:
    print("four")
elif value == 5:
    print("five")
else:
    print("Too large")
print("Done")
```

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

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

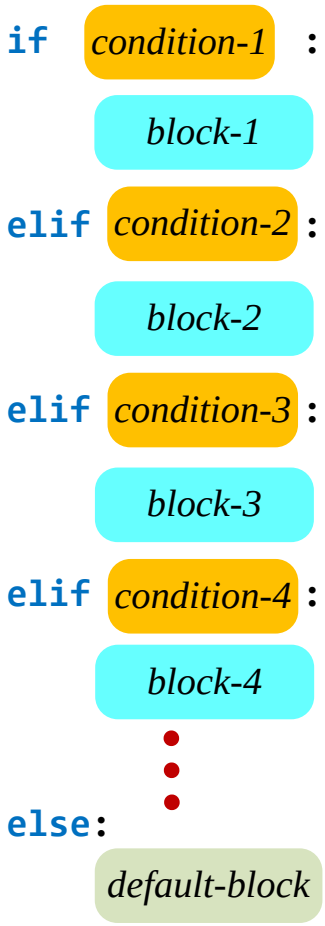
in Listing 4.17 (digittoword.py) into

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

in Listing 4.18 (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.

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



Listing 4.19 (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 4.19: `datetransformer.py`

```
month = int(input("Please enter the month as a number (1-12): "))
day = int(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(" de enero")
elif month == 2:
    print(" de febrero")
elif month == 3:
    print(" de marzo")
elif month == 4:
    print(" de abril")
elif month == 5:
    print(" de mayo")
elif month == 6:
    print(" de junio")
elif month == 7:
    print(" de julio")
elif month == 8:
    print(" de agosto")
elif month == 9:
    print(" de septiembre")
elif month == 10:
    print(" de octubre")
elif month == 11:
    print(" de noviembre")
else:
    print(" de diciembre")
```

A sample run of Listing 4.19 (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 de mayo
```

An `if`/`elif`/`else` statement that includes the optional `else` will execute exactly one of its blocks. The first condition that evaluates to true selects the block to execute. An `if`/`elif`/`else` statement that omits the `else` block may fail to execute the code in any of its blocks if none of its conditions evaluate to `True`.

4.10 Multi-way Versus Sequential Conditionals

Beginning programmers sometimes confuse multi-way `if`/`elif`/`else` statement with a sequence of simple statements. Consider Listing 4.20 (simplerdigittoword.py). It does not print anything if the user enters an integer out of range; otherwise, it prints the equivalent English word.

Listing 4.20: simplerdigittoword.py

```
# Use a mult-way conditional statement
value = int(input())
if value == 0:
    print("zero")
elif value == 1:
    print("one")
elif value == 2:
    print("two")
elif value == 3:
    print("three")
elif value == 4:
    print("four")
elif value == 5:
    print("five")
print("Done")
```

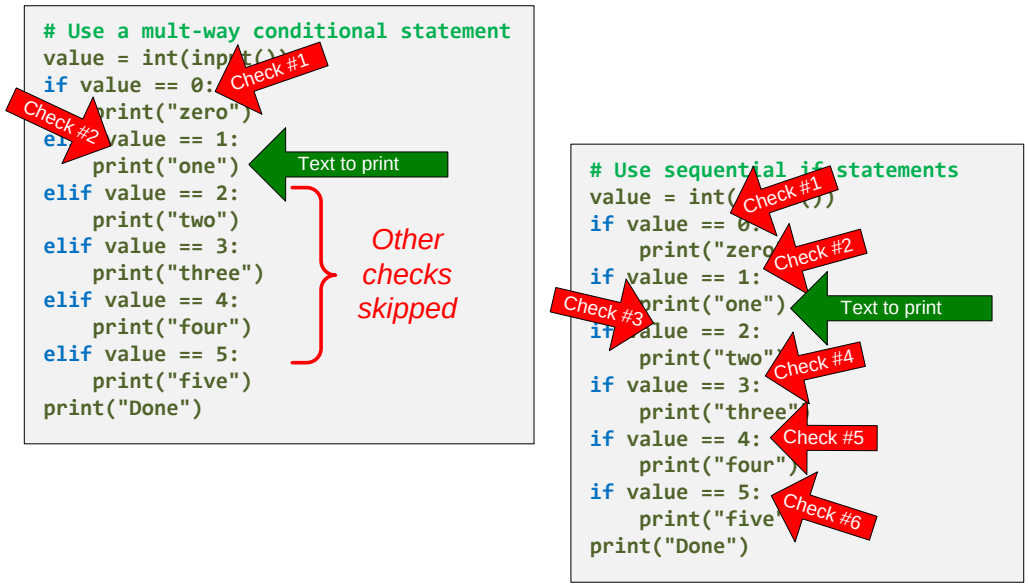
Next, consider Listing 4.21 (simplerdigittoword2.py). It replaces the `elif`s with `if`s. It behaves identically to Listing 4.20 (simplerdigittoword.py).

Listing 4.21: simplerdigittoword2.py

```
# Use sequential conditional statements
value = int(input())
if value == 0:
    print("zero")
if value == 1:
    print("one")
if value == 2:
    print("two")
if value == 3:
    print("three")
if value == 4:
    print("four")
if value == 5:
    print("five")
print("Done")
```

Despite their equivalent behavior, Listing 4.20 (simplerdigittoword.py) is the better program. Figure 4.7 traces the two programs' progress when the user enters the value 1. Listing 4.20 (simplerdigittoword.py) performs only two comparisons before making its choice and then finally printing the message *Done*. In constrast, Listing 4.21 (simplerdigittoword2.py) must check every conditional expression in the program before it prints *Done*. For a program of this size and complexity, the difference is imperceptible. In some more complex situations, however, the extra computation may noticeably affect the application. The bigger issue is the impression that it can make on new programmers that the sequential `if`s are equivalent to

Figure 4.7 A trace of the execution of Listing 4.20 (simplerdigittoword.py) (left) and Listing 4.21 (simplerdigittoword2.py) (right) when the user enters the value 1.



the multi-way `if`/`elif`s. To dispell this idea, compare the behavior of Listing 4.22 (multivsseq1.py) to Listing 4.23 (multivsseq2.py).

Listing 4.22: `multivsseq1.py`

```
num = int(input("Enter a number: "))
if num == 1:
    print("You entered one")
elif num == 2:
    print("You entered two")
elif num > 5:
    print("You entered a number greater than five")
elif num == 7:
    print("You entered seven")
else:
    print("You entered some other number")
```

In Listing 4.22 (multivsseq1.py) if the user enters 7, the program prints *You entered a number greater than five*. Listing 4.23 (multivsseq2.py) merely changes all the `elif`s to `if`s. In this case if the user enters 7, the program will print the same message as does Listing 4.22 (multivsseq1.py), but it additionally will print the message *You entered seven*.

Listing 4.23: `multivsseq2.py`

```
num = int(input("Enter a number: "))
if num == 1:
    print("You entered one")
if num == 2:
    print("You entered two")
if num > 5:
    print("You entered a number greater than five")
if num == 7:
    print("You entered seven")
else:
    print("You entered some other number")
```

These two programs clearly demonstrate that multi-way conditional statements behave differently from sequential `if`s.

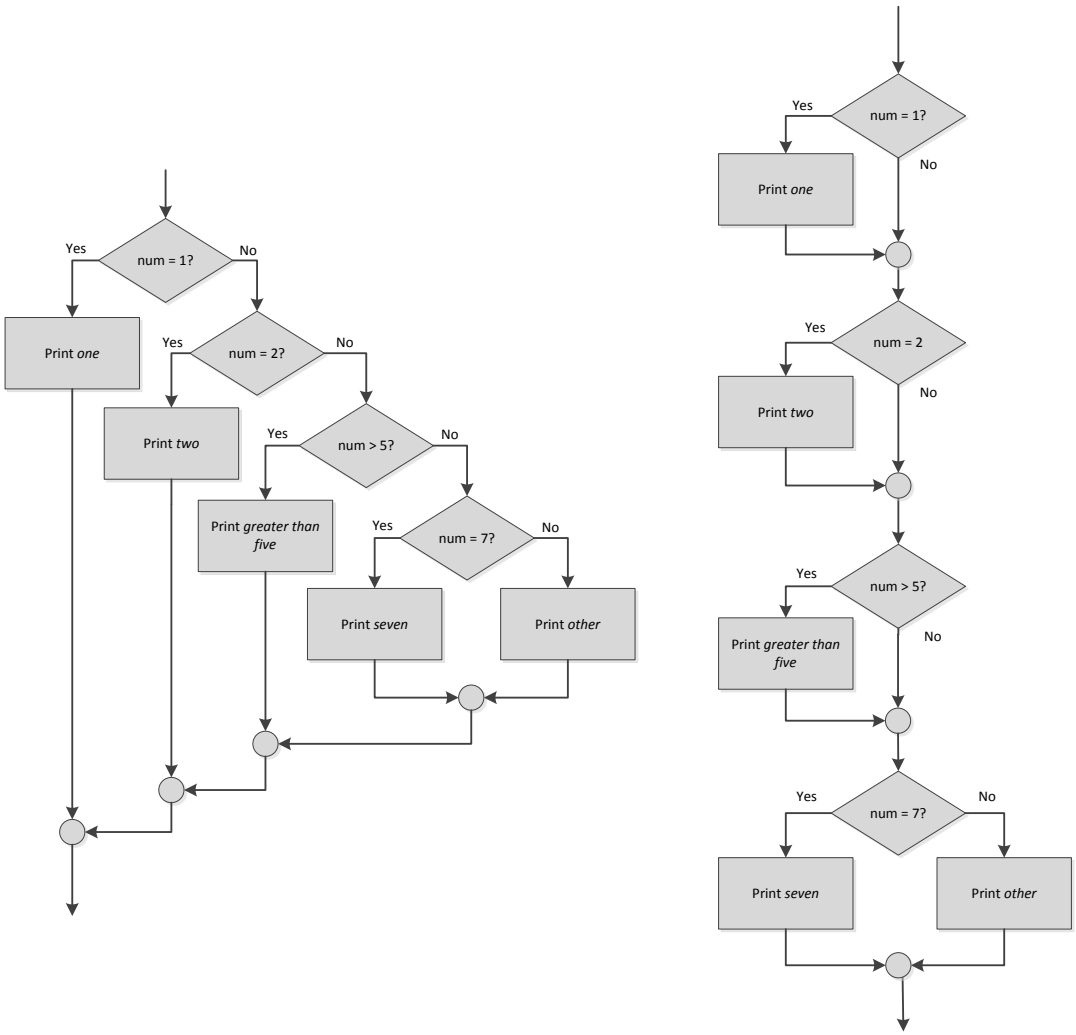
Figure 4.8 compares the structure of the conditional statements in Listing 4.22 (multivsseq1.py) to Listing 4.23 (multivsseq2.py).

4.11 Conditional Expressions

Consider the following code fragment:

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


Figure 4.8 The structure of the conditional statements in Listing 4.22 (multivsseq1.py) (left) versus Listing 4.23 (multivsseq2.py) (right).



This code assigns to variable c 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. We can rewrite the above code as

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

The general form of the conditional expression is

where

- *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.
- *expression-2* is the overall value of the conditional expression if *condition* is false.

In the above code fragment, *expression-1* is the variable d, *condition* is `a != b`, and *expression-2* is e.

Listing 4.24 (safedivide.py) uses our familiar `if`/`else` statement to check for division by zero.

Listing 4.24: safedivide.py

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

Using a conditional expression, we can rewrite Listing 4.24 (safedivide.py) as Listing 4.25 (safedivideconditional.py).

Listing 4.25: safedivideconditional.py

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

Notice that in Listing 4.25 (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
```

An equally valid way to express it is

```
n if n >= 0 else -n
```

The expression itself is not statement. Listing 4.26 (absvalueconditional.py) is a small program that provides an example of the conditional expression's use in a statement.

Listing 4.26: absvalueconditional.py

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

Some sample runs of Listing 4.26 (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, many Python programmers use it 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.

4.12 Errors in Conditional Statements

Carefully consider each compound conditional used, such as

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

found in Listing 4.10 (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? This 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 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.

Table 4.4 indicates that the logical operators and, or, and not have lower precedence than the comparison operators. 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 higher precedence than or, Python interprets the expression

```
x == 1 or 2 or 3
```

as if it were expressed 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. Any appearance of a true value within a sequence of subexpressions all connected by or will make the overall expression true.

The the most correct way express the original 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. Note that each of the subexpressions involves a Boolean expression, not an integer value. If x is known to be an integer and not a floating-point number, the expression

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

also would be equivalent.

Consider Listing 4.27 (digittoworderror.py), which is a variation of Listing 4.18 (restyleddigittoword.py). Listing 4.27 (digittoworderror.py) assigns a word to a string variable instead of printing it immediately.

Listing 4.27: digittoworderror.py

```
value = int(input("Please enter an integer in the range 0...5: "))
answer = "not in range"  # Default answer
if value == 0:
    answer = "zero"
elif value == 1:
    answer = "one"
elif value == 2:
    answer = "two"
elif value == 3:
    amswer = "three"
elif value == 4:
    answer = "four"
elif value == 5:
    answer = "five"
print("The number you entered was", answer)
```

The following shows a sample run of Listing 4.27 (digittoworderror.py):

```
Please enter an integer in the range 0...5: 2
The number you entered was two
```

Another run shows

```
Please enter an integer in the range 0...5: 7
The number you entered was not in range
```

Of course the program will crash with an error if the user enters a noninteger value (the `int` function will raise an exception), but the program should accept any integer value and always do the right thing. In fact, Listing 4.27 (digittoworderror.py) always works as expected *except* when the user enters 3:

```
Please enter an integer in the range 0...5: 3
The number you entered was not in range
```

The program has a subtle spelling error; can you find it?

On an English keyboard the M and N keys are next to each other. It is a common typographical error to press one in place of the other by mistake. Note the variable named `amswer` in the condition that checks for `value` equaling 3. The program reassigns the original `answer` variable for each option except when the user submits the value 3. When the user enters 3 the program creates a new variable which goes unreported in the final printing statement. This kind of error is not restricted to conditional statements but is due to the way Python manages variables. A programmer can create a new variable at any time using the simple assignment operator. The statement

```
amswer = "three"
```

cannot reassign the original `answer` variable but only create a brand new variable with a different name. Unfortunately the interpreter cannot understand what we *mean* by the code we write; it only can dutifully *execute* the code we provide to it.

Listing 4.27 (digittoworderror.py) is small and focussed, and so the logic error introduced by the misspelled variable was easy to track down. As our programs grow in size and our logic becomes more complex, a misspelling error that creates a new variable sometimes can be difficult to track down without

assistance. Fortunately tools exist that can help. Programmers can use *Pylint* (`http://www.pylint.org/`) to analyze Python source code. Among other useful information, Pylint can detect variables defined within a program that have no later use.

4.13 Logic Complexity

Python provides the tools to construct some very complicated conditional statements. It is important to resist the urge to make things overly complex. Composing Boolean expressions with `and`, `or`, and `not` allows us to build conditions with arbitrarily complex logic. There often are many ways to achieve the same effect; for example, the following four Boolean expressions are equivalent:

```
not (a == b and c != d)
not (a == b and not (c == d))
not (a == b) or not (c != d)
a != b or c == d
```

While the four expressions are logically equivalent, they exhibit various degrees of complication.

Some situations require complicated logic to achieve the desired program behavior, but, in general, the simplest logic that works is better than more complex equivalent logic for several reasons:

- Simpler logic usually is easier to understand.
- Simpler logic is easier to write and get working. Longer, more complicated expressions increase the risk of typographical errors that can manifest themselves as logic errors. These errors are more difficult to find and correct in complex code.
- Simpler logic is can be more efficient. Every relational comparison and Boolean operation executed by a running program requires machine cycles. The expression

  ```
  not (a == b and not (c == d))
  ```

  performs up to five separate operations: `==` twice, `not` twice, and `and` once. If `a` does not equal `b`, the expression will perform only two operations (`==` and the outer `not`) due to short-circuit Boolean evaluation of the `and`. Compare this expression to the following equivalent expression:

  ```
  a != b or c == d
  ```

  An executing program can evaluate this expression faster, as it involves at most three operations: `!=`, `==`, and `or`. If `a` is not equal to `b`, the execution short-circuits the `or`, and evaluates only the `!=` operation.
- Simpler logic is easier to modify and extend.

Consider the task of computing the maximum value of four integers entered by the user. Listing 4.28 (max4a.py) provides one solution.

Listing 4.28: `max4a.py`

```
# Get input values from user
print("Please enter four integer values.")
num1 = int(input("Enter number 1: "))
num2 = int(input("Enter number 2: "))
```



```
num3 = int(input("Enter number 3: "))
num4 = int(input("Enter number 4: "))

# Compute the maximum value
if num1 >= num2 and num1 >= num3 and num1 >= num4:
    max = num1
elif num2 >= num1 and num2 >= num3 and num2 >= num4:
    max = num2
elif num3 >= num1 and num3 >= num2 and num3 >= num4:
    max = num3
else:  # The maximum must be num4 at this point
    max = num4

# Report result
print("The maximum number entered was:", max)
```

Listing 4.28 (max4a.py) uses a multiway `if`/`elif`/`else` construct to select the correct value to assign to its `max` variable.

Listing 4.28 (max4a.py) works correctly for any four integers a user may enter, but its logic is somewhat complicated. We will consider a different strategy for our selection logic. All variables have a meaning, and their names should reflect their meaning in some way. We'll let our `max` variable mean "maximum I have determined so far." The following outlines our new approach to the solution:

1. Set `max` equal to `n1`. This means as far as we know at the moment, `n1` is the biggest number because `max` and `n1` have the same value.
2. Compare `max` to `n2`. If `n2` is larger than `max`, change `max` to have `n2`'s value to reflect the fact that we determined `n2` is larger; if `n2` is not larger than `max`, we have no reason to change `max`, so do not change it.
3. Compare `max` to `n3`. If `n3` is larger than `max`, change `max` to have `n3`'s value to reflect the fact that we determined `n3` is larger; if `n3` is not larger than `max`, we have no reason to change `max`, so do not change it.
4. Compare `max` to `n4`. If `n4` is larger than `max`, change `max` to have `n4`'s value to reflect the fact that we determined `n4` is larger; if `n4` is not larger than `max`, we have no reason to change `max`, so do not change it.

In the end the meaning of the `max` variable remains the same–"maximum I have determined so far," but, after comparing `max` to all the input variables, we now know it is *the* maximum value of all four input numbers. The extra variable `max` is not strictly necessary, but it makes thinking about the problem and its solution easier.

Listing 4.29 (max4b.py) uses our revised selection logic to provide an alternative to Listing 4.28 (max4a.py).

Listing 4.29: `max4b.py`

```
# Get input values from user
print("Please enter four integer values.")
num1 = int(input("Enter number 1: "))
num2 = int(input("Enter number 2: "))
num3 = int(input("Enter number 3: "))
num4 = int(input("Enter number 4: "))
```



```
# Compute the maximum value
max = num1
if num2 > max:
    max = num2
if num3 > max:
    max = num3
if num4 > max:
    max = num4

# Report result
print("The maximum number entered was:", max)
```

Listing 4.29 (max4b.py) uses simple, sequential `if` statements to eventually assign the correct value to its `max` variable.

Listing 4.29 (max4b.py) always performs three > comparisons and at most four assignments regardless of the values provided by the user. The source code for Listing 4.28 (max4a.py) contains many more Boolean operations than Listing 4.29 (max4b.py), but, due to short-circuit evaluation, Listing 4.28 (max4a.py) actually can outperform Listing 4.29 (max4b.py) in some situations. Simpler logic is no guarantee of better performance. Both programs are so simple, however, that users will perceive no difference in execution speeds.

What changes would we need to make to both Listing 4.28 (max4a.py) and Listing 4.29 (max4b.py) if we must extend them to handle five input values instead of four? Adding this capability to Listing 4.28 (max4a.py) forces us to modify **every** condition in the program, adding a check against a new `num5` variable. We also must provide an additional `elif` check since we will need to select from among five possible assignments to the `max` variable. In Listing 4.29 (max4b.py), however, we need only add an extra sequential `if` statement with a new simple condition to check.

Chapter 5 introduces loops, the ability to execute statements repeatedly. You easily can adapt the sequential `if` approach to allow users to type in as many numbers as they like and then have the program report the maximum number the user entered. The multiway `if`/`elif`/`else` approach with the more complex logic cannot be adapted in this manner. Not only is our sequential `if` version cleaner and simpler; we more easily can extend its capabilities.

4.14 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:

```
x, y = 3, 5
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 or fewer symbols. `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. Express the following Boolean expressions in an equivalent form *without* the `not` operator. `x` and `y` are integers.

   (a) `not (x == y)`

   (b) `not (x > y)`

   (c) `not (x < y)`

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

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

   (f) `not (x != y)`

   (g) `not (x != y)`

   (h) `not (x == y and x < 2)`

   (i) `not (x == y or x < 2)`

   (j) `not (not (x == y))`

10. What is the simplest tautology?

11. What is the simplest contradiction?

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, do not print anything.

13. 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."

14. 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.

15. Consider the following Python code fragment:

```
# 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

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

```
val = int(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

17. Consider the following two Python programs that appear very similar:

```
n = int(input())
if n < 1000:
    print('*', end='')
if n < 100:
    print('*', end='')
if n < 10:
    print('*', end='')
if n < 1:
    print('*', end='')
print()
```

```
n = int(input())
if n < 1000:
    print('*', end='')
elif n < 100:
    print('*', end='')
elif n < 10:
    print('*', end='')
elif n < 1:
    print('*', end='')
print()
```

How do the two programs react when the user provides the following inputs?

(a) 0

(b) 1

(c) 5

(d) 50

(e) 500

(f) 5000

Why do the two programs behave as they do?

18. 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.

19. 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"`.

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

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


```
count = 1             # Initialize counter
while count <= 5:     # Should we continue?
    print(count)      # Display counter, then
    count += 1        # Increment counter
```

The `while` statement in Listing 5.2 (iterativecounttofive.py) repeatedly displays the variable `count`. The program executes the following block of statements five times:

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

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.

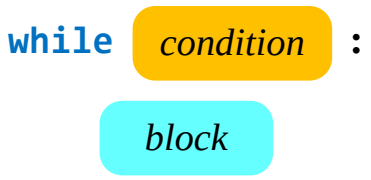
Unlike the approach taken in Listing 5.1 (counttofive.py), it is trivial to modify Listing 5.2 (iterativecounttofive.py) to count up to 10,000—just change the literal value `5` to `10000`.

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 program will not execute the code block within the body of the loop at all.

The `while` statement has the general form:

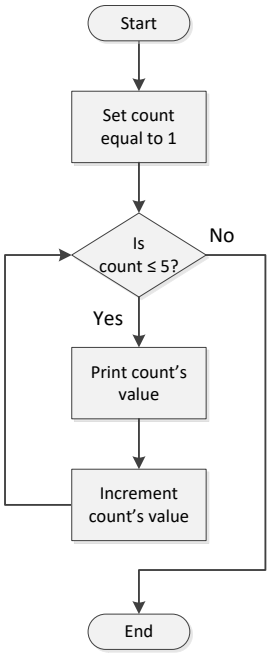


- The reserved word `while` begins the `while` statement.
- The *condition* determines whether the body will be (or will continue to be) executed. A colon (`:`) must follow the condition.
- *block* is a block of one or more statements to be executed as long as the condition is true. As a block, all the statements that comprise the block must be indented one level deeper 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).

Figure 5.1 while flowchart for Listing 5.2 (iterativecounttofive.py)



The executing program checks the condition before executing the `while` block and then checks the condition again after executing the `while` block. As long as the condition remains truth, the program repeatedly executes the code in the `while` block. If the condition initially is false, the program will not execute the code within the `while` block. If the condition initially is true, the program executes the block repeatedly until the condition becomes false, at which point the loop terminates.

Listing 5.3 (countup.py) counts up from zero as long as the user wishes to do so.

Listing 5.3: countup.py

```
#  Counts up from zero.  The user continues the count by entering
#  'Y'.  The user discontinues the count by entering 'N'.

count = 0     # The current count
entry = 'Y'   # Count to begin with

while entry != 'N' and entry != 'n':
    # Print the current value of count
    print(count)
    entry = input('Please enter "Y" to continue or "N" to quit: ')
    if entry == 'Y' or entry == 'y':
        count += 1    # Keep counting
    # Check for "bad" entry
    elif entry != 'N' and entry != 'n':
        print('"' + entry + '" is not a valid choice')
    # else must be 'N' or 'n'
```

A sample run of Listing 5.3 (countup.py) produces

```
0
Please enter "Y" to continue or "N" to quit: y
1
Please enter "Y" to continue or "N" to quit: y
2
Please enter "Y" to continue or "N" to quit: y
3
Please enter "Y" to continue or "N" to quit: q
"q" is not a valid choice
3
Please enter "Y" to continue or "N" to quit: r
"r" is not a valid choice
3
Please enter "Y" to continue or "N" to quit: W
"W" is not a valid choice
3
Please enter "Y" to continue or "N" to quit: Y
4
Please enter "Y" to continue or "N" to quit: y
5
Please enter "Y" to continue or "N" to quit: n
```

In Listing 5.3 (countup.py) the expression

```
entry != 'N' and entry != 'n'
```

is true if entry is neither *N* not *n*.

Listing 5.4 (addnonnegatives.py) is a program that allows a user to enter any number of nonnegative integers. When the user enters a negative value, the program no longer accepts input, and it displays the sum of all the nonnegative values. If a negative number is the first entry, the sum is zero.

Listing 5.4: addnonnegatives.py

```
#  Allow the user to enter a sequence of nonnegative
#  integers.  The user ends the list with a negative
#  integer.  At the end the sum of the nonnegative
#  numbers entered is displayed.  The program prints
#  zero if the user provides no nonnegative numbers.

entry = 0      # Ensure the loop is entered
sum = 0        # Initialize sum

#  Request input from the user
print("Enter numbers to sum, negative number ends list:")

while entry >= 0:          # A negative number exits the loop
    entry = int(input())  # Get the value
    if entry >= 0:         # Is number nonnegative?
        sum += entry       # Only add it if it is nonnegative
print("Sum =", sum)        # Display the sum
```

Listing 5.4 (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. The entry variable holds the number entered by the user. Its value can change each time through the loop.

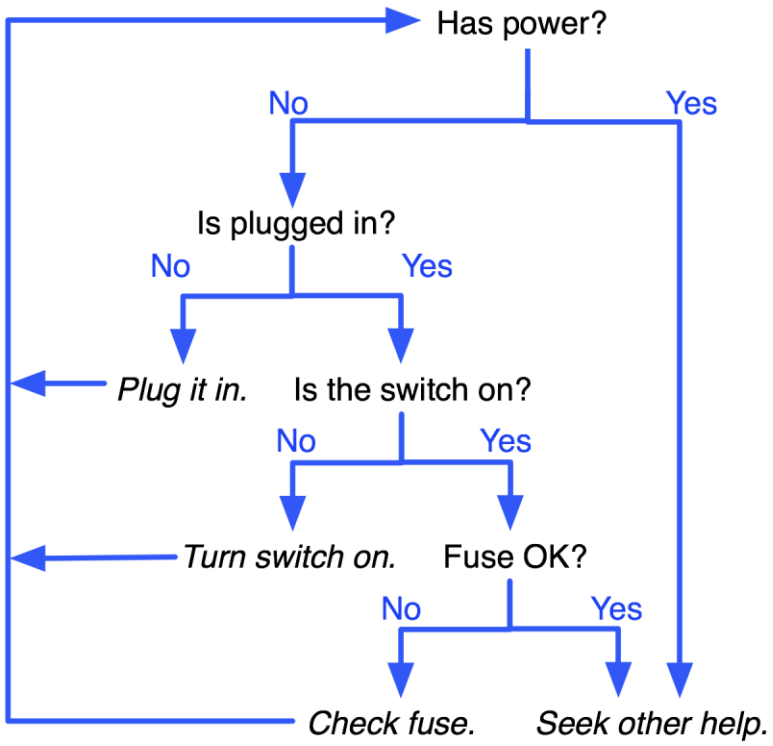
- sum

  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 will generate a run-time error when it attempts to use the += operator to modify the (non-existent) 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 nonnegative values entered by the user.

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

Listing 5.4 (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.

Figure 5.2 Decision tree for troubleshooting a computer system



We can use a `while` statement to make Listing 4.14 (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 5.5 (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.

Listing 5.5: troubleshootloop.py

```
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
                        print("Check the outlet's circuit ")
                        print("breaker or fuse.  Move to a")
                        print("new outlet, if necessary. ")
                    else:  # Beats me!
                        print("Please consult a service technician.")
                        done = True   # Nothing else I can do
                else: # Check fuse
                    print("Check the fuse. Replace if ")
                    print("necessary.")
    else:  # The computer has power
        print("Please consult a service technician.")
        done = True   # Nothing else I can do
```

A `while` block makes up the bulk of Listing 5.5 (troubleshootloop.py). The Boolean variable done controls the loop; as long as done is false, the loop continues. A Boolean variable like done used in this fashion 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.

It is important to note that the expression

```
not done
```

of the `while` statement's condition evaluates to the opposite truth value of the variable done; the expression does *not* affect the value of done. In other words, the `not` operator applied to a variable does not modify the variable's value. In order to actually change the variable done, you would need to reassign it, as in

```
done = not done  # Invert the truth value
```

For Listing 5.5 (troubleshootloop.py) we have no need to invert done's value. We ensure that done's value is `False` initially and then make it `True` when the user has exhausted the program's options.

In Python, sometimes it is convenient to use a simple value as conditional expression in an `if` or `while` statement. Python interprets the integer value 0 and floating-point value 0.0 both as `False`. All other integer and floating-point values, both positive and negative, are considered `True`. This means the following code:

```
x = int(input())  # Get integer from user
while x:
    print(x)   # Print x only if x is nonzero
    x -= 1     # Decrement x
```

is equivalent to

```
x = int(input())  # Get integer from user
while x != 0:
    print(x)   # Print x only if x is nonzero
    x -= 1     # Decrement x
```


Speaking of floating-point values, Section 3.1 demonstrated how the limited precision of floating-point numbers can be a problem. You should use caution when using a floating-point number to control a loop, as Listing 5.6 (stopatone.py) shows

Listing 5.6: stopatone.py

```
x = 0.0
while x != 1.0:
    print(x)
    x += 0.1
print("Done")
```

The programmer's intention is for Listing 5.6 (stopatone.py) to begin at 0.0 and count up to 1.0 by tenths. In other words, the program should stop looping when the variable x becomes 1.0. With true mathematical real numbers, if we add $\frac{1}{10}$ to 0 ten times, the result equals 1. Unfortunately, Listing 5.6 (stopatone.py) bypasses 1 and keeps going! The first few lines of its execution are

```
0.0
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
1.0999999999999999
1.2
1.3
1.4000000000000001
1.5000000000000002
```

The program never stops printing numbers. Note that $\frac{1}{10}$ does not have an exact internal binary floating-point representation. If you change the statement

```
x += 0.1
```

to be

```
x += 0.125
```

the program stops as expected. This is because $\frac{1}{8}$ happens to have an exact internal binary floating-point representation. Since such exact representations of floating-point values are rare, you should avoid using the == and != operators with floating-point numbers to control the number of loop iterations. Listing 5.7 (stopatonefixed.py) uses <= to control the loop instead of !=, and it behaves as expected.

Listing 5.7: stopatonefixed.py

```
x = 0.0
```

```
while x != 1.0:
    print(x)
    x += 0.1
print("Done")
```

5.2 Definite Loops vs. Indefinite Loops

In Listing 5.8 (definite1.py), code similar to Listing 5.1 (counttofive.py), prints the integers from one to 10.

Listing 5.8: definite1.py

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

We can inspect the code and determine the exact number of iterations the loop will perform. This kind of loop is known as a *definite loop*, since we can predict exactly how many times the loop repeats. Consider Listing 5.9 (definite2.py).

Listing 5.9: definite2.py

```
n = 1
stop = int(input())
while n <= stop:
    print(n)
    n += 1
```

Looking at the source code of Listing 5.9 (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 5.9 (definite2.py) is considered to be a definite loop as well.

Compare these programs to Listing 5.10 (indefinite.py).

Listing 5.10: indefinite.py

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

In Listing 5.10 (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 5.10 (indefinite.py) is an example of an *indefinite loop*.

Listing 5.5 (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 5.5 (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 sequence of values. One way to express a sequence is via a tuple, as shown here:

```
for n in 1, 2, 3, 4, 5, 6, 7, 8, 9, 10:
    print(n)
```

This code behaves identically to the while loop above. The print statement here executes exactly 10 times. The code first prints 1, then 2, then 3, etc. The final value it prints is 10. During the iteration the variable n thus assumes, in order, all the values that make up the tuple.

It usually is cumbersome to explicitly list all the elements of a tuple, and often it is impractical. Consider iterating over all the integers from 1 to 1,000—writing out all the tuple's elements would be unwieldy. Fortunately, Python provides a convenient way to express a sequence of integers that follow a regular pattern. The following code uses a range expression to print the integers 1 through 10:

```
for n in range(1, 11):
    print(n)
```

The expression range(1, 11) creates a range object that allows the for loop to assign to the variable n the values 1, 2, ..., 10. Conceptually, the expression range(1, 11) represents the sequence of integers $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$.

The line

```
for n in range(1, 11):
```


is best read as "For each integer n in the range $1 \leq n < 11$." 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. Each time through the loop, n's value increases by one. The code within the block will use the values of n up to 10.

The general form of the range expression is

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 is always required and may **not** be omitted
- *step* is the amount to increment or decrement; if the *step* parameter is omitted, it defaults to 1 (counts up by ones)

begin, *end*, and *step* must all be integer expressions; floating-point expressions and other types are not allowed. The arguments in the range expression may be literal numbers (like 10), variables (like x, if x is bound to an integer), and arbitrarily complex integer expressions.

The range expression is very flexible. Consider the following loop that counts down from 21 to 3 by threes:

```
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:

```
sum = 0    # Initialize sum
for i in range(1, 100):
   sum += i
print(sum)
```

The following examples show how to use range 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)` → $-5,-4,-3,-2,-1,0,1,2,3,4$
- `range(1, 2)` → 1
- `range(1, 1)` → (empty)
- `range(1, -1)` → (empty)
- `range(1, -1, -1)` → $1,0$
- `range(0)` → (empty)

In a `range` expression with one argument, as in `range(x)`, the `x` represents the *end* of the range, with 0 being the implied *begin* value, and 1 being the *step* value.

In a `range` expression with two arguments, as in `range(x, y)`, the `x` represents the *begin* value, and `y` represents the *end* of the range. The implied *step* value is 1.

In a `range` expression with three arguments, as in `range(x, y, z)`, the `x` represents the *begin* value, `y` represents the *end* of the range, and `z` is the step value.

Loops allow us to rewrite an expanded form of Listing 2.20 (powers10right.py) more compactly. Listing 5.11 (powers10loop.py) uses a `for` loop to print the first 16 powers of 10.

Listing 5.11: powers10loop.py

```
for i in range(16):
    print('{0:3} {1:16}'.format(i, 10**i))
```

Listing 5.11 (powers10loop.py) prints

```
  0                1
  1               10
  2              100
  3             1000
  4            10000
  5           100000
  6          1000000
  7         10000000
  8        100000000
  9       1000000000
 10      10000000000
 11     100000000000
 12    1000000000000
 13   10000000000000
 14  100000000000000
 15 1000000000000000
```

In a `for` loop the `range` object has complete control over determining the loop variable each time through the loop. To prove this, Listing 5.12 (abusefor.py) attempts to thwart the `range`'s loop variable by changing its value inside the loop.

Listing 5.12: abusefor.py

```
#  Abuse the for statement
```

```
for i in range(10):
    print(i, end=' ')   # Print i as served by the range object
    if i == 5:
        i = 20   # Change i inside the loop?
    print('({})'.format(i), end=' ')
print()
```

Listing 5.12 (abusefor.py) prints the following:

```
0 (0) 1 (1) 2 (2) 3 (3) 4 (4) 5 (20) 6 (6) 7 (7) 8 (8) 9 (9)
```

The first number is i's value at the beginning of the block, and the parenthesized number is i's value at the end of the block before the next iteration. The code within the block can reassign i, but this binds i to a different integer object (20). The next time through the loop the for statement obtains the next integer served by the range object and binds i to this new integer.

If you look in older Python books or at online examples of Python code, you probably will encounter the xrange expression. Python 2 has both range and xrange, but Python 3 (the version we use in this text) does not have the xrange expresson. The range expression in Python 3 is equivalent to the xrange expression in Python 2. The range expression in Python 2 creates a data structure called a *list*, and this process can involve considerable overhead for an executing program. The xrange expression in Python 2 avoids this overhead, making it more efficient than range, especially for a large sequence. When building loops with the for statement, Python 2 programmers usually use xrange rather than range to improve their code's efficiency. In Python 3, we can use range without compromising run-time performance. In Chapter 10 we will see it is easy to make a list out of a Python 3 range expression, so Python 3 does not need two different range expressions that do almost exactly the same thing.

We initially emphasize the for loop's ability to iterate over integer sequences because this is a useful and common task in software construction. The for loop, however, can iterate over any iterable object. As we have seen, a tuple is an iterable object, and a range object is an iterable object. A string also is an iterable object. We can use a for loop to iterate over the characters that comprise a string. Listing 5.13 (stringletters.py) uses a for loop to print the individual characters of a string.

Listing 5.13: stringletters.py

```
word = input('Enter a word: ')
for letter in word:
    print(letter)
```

In the following sample execution of Listing 5.13 (stringletters.py) shows how the program responds when the user enters the word *tree*:

```
Enter a word: tree
t
r
e
e
```


At each iteration of its `for` loop Listing 5.13 (stringletters.py) assigns to the `letter` variable a string containing a single character.

Listing 5.14 (stringliteralletters.py) uses a `for` loop to iterate over a literal string.

Listing 5.14: `stringliteralletters.py`

```
for c in 'ABCDEF':
    print('[', c, ']', end='', sep='')
print()
```

Listing 5.14 (stringliteralletters.py) prints

```
[A][B][C][D][E][F]
```

Listing 5.15 (countvowels.py) counts the number of vowels in the text provided by the user.

Listing 5.15: `countvowels.py`

```
word = input('Enter text: ')
vowel_count = 0
for c in word:
    if c == 'A' or c == 'a' or c == 'E' or c == 'e' \
       or c == 'I' or c == 'i' or c == 'O' or c == 'o':
        print(c, ', ', sep='', end='')  # Print the vowel
        vowel_count += 1                # Count the vowel
print(' (', vowel_count, ' vowels)', sep='')
```

Listing 5.14 (stringliteralletters.py) prints vowels it finds and then reports how many it found:

```
Enter text: Mary had a little lamb.
a, a, a, i, e, a,  (6 vowels)
```

Chapter 10 and beyond use the `for` statement to traverse data structures such as lists and dictionaries.

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. To see how nested loops work, consider a program that prints out a *multiplication table*. Elementary school students use multiplication tables, or times tables, as they learn the products of integers up to 10 or even 12. Figure 5.3 shows a 10×10 multiplication table. We want our multiplication table program to be flexible and allow the user to specify the table's size. We will begin our development work with a simple program and add features as we go. First, we will not worry about printing the table's row and column titles, nor will we print the lines separating the titles from the contents of the table. Initially we will print only the contents of the table. We will see we need a nested loop to print the table's contents, but that still is too much to manage in our first attempt. In our first attempt we will print the rows of the table in a very rudimentary manner. Once we are satisfied that our simple program works we can add more features. Listing 5.16 (timestable1.py) shows our first attempt at a muliplication table.

Listing 5.16: `timestable1.py`

Figure 5.3 A 10 × 10 multiplication table

×	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

```
#  Get the number of rows and columns in the table
size = int(input("Please enter the table size: "))
#  Print a size x size multiplication table
for row in range(1, size + 1):
    print("Row #", row)
```

The output of Listing 5.16 (timestable1.py) is somewhat underwhelming:

```
Please enter the table size: 10
Row #1
Row #2
Row #3
Row #4
Row #5
Row #6
Row #7
Row #8
Row #9
Row #10
```

Listing 5.16 (timestable1.py) does indeed print each row in its proper place—it just does not supply the needed detail for each row. Our next step is to refine the way the program prints each row. Each row should contain `size` numbers. Each number within each row represents the product of the current row and current column; for example, the number in row 2, column 5 should be $2 \times 5 = 10$. In each row, therefore, we must vary the column number from from 1 to `size`. Listing 5.17 (timestable2.py) contains the needed refinement.

Listing 5.17: `timestable2.py`

```
#  Get the number of rows and columns in the table
size = int(input("Please enter the table size: "))
#  Print a size x size multiplication table
for row in range(1, size + 1):
```



```
    for column in range(1, size + 1):
        product = row*column     # Compute product
        print(product, end=' ')  # Display product
    print()                      # Move cursor to next row
```

We use a loop to print the contents of each row. The outer loop controls how many total rows the program prints, and the inner loop, executed in its entirety each time the program prints a row, prints the individual elements that make up a row.

The result of Listing 5.17 (timestable2.py) is

```
Please enter the table size: 10
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

The numbers within each column are not lined up nicely, but the numbers are in their correct positions relative to each other. We can use the string formatter introduced in Listing 2.20 (powers10right.py) to right justify the numbers within a four-digit area. Listing 5.18 (timestable3.py) contains this alignment adjustment.

Listing 5.18: timestable3.py

```
#  Get the number of rows and columns in the table
size = int(input("Please enter the table size: "))
#  Print a size x size multiplication table
for row in range(1, size + 1):
    for column in range(1, size + 1):
        product = row*column                    # Compute product
        print('{0:4}'.format(product), end='')  # Display product
    print()                                     # Move cursor to next row
```

Listing 5.18 (timestable3.py) produces the table's contents in an attractive form:

```
Please enter the table size: 10
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100
```

Notice that the table presentation adjusts to the user's input:

```
Please enter the table size: 5
  1   2   3   4   5
  2   4   6   8  10
  3   6   9  12  15
  4   8  12  16  20
  5  10  15  20  25
```

A multiplication table of size 15 looks like

```
Please enter the table size: 15
  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
  2   4   6   8  10  12  14  16  18  20  22  24  26  28  30
  3   6   9  12  15  18  21  24  27  30  33  36  39  42  45
  4   8  12  16  20  24  28  32  36  40  44  48  52  56  60
  5  10  15  20  25  30  35  40  45  50  55  60  65  70  75
  6  12  18  24  30  36  42  48  54  60  66  72  78  84  90
  7  14  21  28  35  42  49  56  63  70  77  84  91  98 105
  8  16  24  32  40  48  56  64  72  80  88  96 104 112 120
  9  18  27  36  45  54  63  72  81  90  99 108 117 126 135
 10  20  30  40  50  60  70  80  90 100 110 120 130 140 150
 11  22  33  44  55  66  77  88  99 110 121 132 143 154 165
 12  24  36  48  60  72  84  96 108 120 132 144 156 168 180
 13  26  39  52  65  78  91 104 117 130 143 156 169 182 195
 14  28  42  56  70  84  98 112 126 140 154 168 182 196 210
 15  30  45  60  75  90 105 120 135 150 165 180 195 210 225
```

All that is left is to add the row and column titles and the lines that bound the edges of the table. Listing 5.19 (timestable4.py) adds the necessary code.

Listing 5.19: `timestable4.py`

```
#  Get the number of rows and columns in the table
size = int(input("Please enter the table size: "))
#  Print a size x size multiplication table
#  First, print heading:  1  2  3  4   5   etc.
print("     ", end='')
#  Print column heading
for column in range(1, size + 1):
    print('{0:4}'.format(column), end='')  # Display column number
print()   # Go down to the next line
#  Print line separator:   +------------------
print("    +", end='')
for column in range(1, size + 1):
    print('----', end='')  # Display line
print()     # Drop down to next line

#  Print table contents
for row in range(1, size + 1):
    print('{0:3} |'.format(row), end='')       # Print heading for this row
    for column in range(1, size + 1):
        product = row*column                    # Compute product
        print('{0:4}'.format(product), end='')  # Display product
    print()                                     # Move cursor to next row
```


When the user supplies the value 10, Listing 5.19 (timestable4.py) produces

```
Please enter the table size: 10
       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
```

An input of 15 yields

```
Please enter the table size: 15
       1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
    +-----------------------------------------------------------
  1 |   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
  2 |   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30
  3 |   3   6   9  12  15  18  21  24  27  30  33  36  39  42  45
  4 |   4   8  12  16  20  24  28  32  36  40  44  48  52  56  60
  5 |   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75
  6 |   6  12  18  24  30  36  42  48  54  60  66  72  78  84  90
  7 |   7  14  21  28  35  42  49  56  63  70  77  84  91  98 105
  8 |   8  16  24  32  40  48  56  64  72  80  88  96 104 112 120
  9 |   9  18  27  36  45  54  63  72  81  90  99 108 117 126 135
 10 |  10  20  30  40  50  60  70  80  90 100 110 120 130 140 150
 11 |  11  22  33  44  55  66  77  88  99 110 121 132 143 154 165
 12 |  12  24  36  48  60  72  84  96 108 120 132 144 156 168 180
 13 |  13  26  39  52  65  78  91 104 117 130 143 156 169 182 195
 14 |  14  28  42  56  70  84  98 112 126 140 154 168 182 196 210
 15 |  15  30  45  60  75  90 105 120 135 150 165 180 195 210 225
```

If the user enters 7, the program prints

```
Please enter the table size: 7
       1   2   3   4   5   6   7
    +----------------------------
  1 |   1   2   3   4   5   6   7
  2 |   2   4   6   8  10  12  14
  3 |   3   6   9  12  15  18  21
  4 |   4   8  12  16  20  24  28
  5 |   5  10  15  20  25  30  35
  6 |   6  12  18  24  30  36  42
  7 |   7  14  21  28  35  42  49
```

The user even can enter a 1:

```
Please enter the table size: 1
       1
    +----
```



```
1 |   1
```

As we can see, the table automatically adjusts to the size and spacing required by the user's input.

This is how Listing 5.19 (timestable4.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, the program uses a loop to print it one time.
- 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.
- The printing statement

  ```
  print('{0:4}'.format(product), end='')  # Display product
  ```

  right justifies the value of product in field that is four characters wide. This technique properly aligns the columns within the times table.
- In the nested loop, `row` is the control variable for the outer loop; `column` controls the inner loop.
- The inner loop executes `size` times on every single iteration of the outer loop. This means the innermost statement

  ```
  print('{0:4}'.format(product), end='')  # Display product
  ```

  executes `size` $\times$ `size` times, one time for every product in the table.
- The program prints a newline after it displays the contents of each row; thus, all the values printed in the inner (`column`) loop appear on the same line.

Nested loops are necessary when an iterative process itself must be repeated. In our times table example, a `for` loop prints the contents of each row, and an enclosing `for` loop prints out each row.

Listing 5.20 (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*. A permutation, therefore, is a possible ordering of a sequence.

Listing 5.20: `permuteabc.py`

```
#  File permuteabc.py

#  The first letter varies from A to C
for first in 'ABC':
    for second in 'ABC': # The second varies from A to C
        if second != first:  #  No duplicate letters allowed
            for third in 'ABC':  # The third varies from A to C
                # Don't duplicate first or second letter
                if third != first and third != second:
                    print(first + second + third)
```

Notice how the `if` statements prevent duplicate letters within a given string. The output of Listing 5.20 (permuteabc.py) is all six permutations of `ABC`:

```
ABC
ACB
BAC
BCA
CAB
CBA
```

Listing 5.21 (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 5.21: permuteabcd.py

```
#  File permuteabcd.py

#  The first letter varies from A to D
for first in 'ABCD':
    for second in 'ABCD': # The second varies from A to D
        if second != first:  #  No duplicate letters allowed
            for third in 'ABCD':  # The third varies from A to D
                # Don't duplicate first or second letter
                if third != first and third != second:
                    for fourth in 'ABCD':   # The fourth varies from A to D
                        if fourth != first and fourth != second and fourth != third:
                            print(first + second + third + fourth)
```

Nested loops are powerful, and some novice programmers attempt to use nested loops where a single loop is more appropriate. Before you attempt to solve a problem with a nested loop, make sure that there is no way you can do so with a single loop. Nested loops are more difficult to write correctly and, when not necessary, they are less efficient than a simple loop.

5.5 Abnormal Loop Termination

Normally, a `while` statement executes until its condition becomes false. A running program checks this condition first to determine if it should execute the statements in the loop's body. It then re-checks this condition only after executing all the statements in the loop's body. Ordinarily a `while` loop will not immediately exit its body if its condition becomes false before completing all the statements in its body. The `while` statement is designed this way because usually the programmer intends 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. Said another way, a `while` statement checks its condition only at the "top" of the loop. It is not the case that a `while` loop finishes immediately whenever its condition becomes true. Listing 5.22 (whileexitattop.py) demonstrates this top-exit behavior.

Listing 5.22: whileexitattop.py

```
x = 10
while x == 10:
    print('First print statement in the while loop')
    x = 5    # Condition no longer true; do we exit immediately?
    print('Second print statement in the while loop')
```

Listing 5.22 (whileexitattop.py) prints

```
First print statement in the while loop
Second print statement in the while loop
```

Even though the condition for continuing in the loop (x being equal to 10) changes in the middle of the loop's body, the `while` statement does not check the condition until it completes all the statements in its body and execution returns to the top of the loop.

Sometimes it is more convenient to exit a loop from the middle of its body; that is, quit the loop before all the statements in its body execute. This means if a certain condition becomes true in the loop's body, exit right away.

Similarly, a `for` statement typically iterates over all the values in its range or over all the characters in its string. Sometimes, however, it is desirable to exit the `for` loop prematurely. Python provides the `break` and `continue` statements to give programmers more flexibility designing the control logic of loops.

5.5.1 The `break` statement

As we noted above, sometimes it is necessary to exit a loop from the middle of its body; that is, quit the loop before all the statements in its body execute. This means if a certain condition becomes true in the loop's body, exit right away. This "middle-exiting" condition could be the same condition that controls the `while` loop (that is, the "top-exiting" condition), but it does not need to be.

Python provides the `break` statement to implement middle-exiting loop control logic. The `break` statement causes the program's execution to immediately exit from the body of the loop. Listing 5.23 (addmiddleexit.py) is a variation of Listing 5.4 (addnonnegatives.py) that illustrates the use of `break`.

Listing 5.23: `addmiddleexit.py`

```
#  Allow the user to enter a sequence of nonnegative
#  numbers.  The user ends the list with a negative
#  number.  At the end the sum of the nonnegative
#  numbers entered is displayed.  The program prints
#  zero if the user provides no nonnegative numbers.

entry = 0      # Ensure the loop is entered
sum = 0        # Initialize sum

#  Request input from the user
print("Enter numbers to sum, negative number ends list:")

while True:                # Loop forever?  Not really
    entry = int(input())   # Get the value
    if entry < 0:          # Is number negative number?
        break              # If so, exit the loop
    sum += entry           # Add entry to running sum
print("Sum =", sum)        # Display the sum
```

The condition of the `while` statement in Listing 5.23 (addmiddleexit.py) is a tautology, so when the program runs it is guaranteed to begin executing the statements in its `while` block at least once. Since the condition

of the while can never be false, the break statement is the only way to get out of the loop. Here, the break statement executes only when it determines that the number the user entered is negative. When the program encounters the break statement during its execution, it skips any statements that follow in the loop's body and exits the loop immediately. The keyword break means "break out of the loop." The placement of the break statement in Listing 5.23 (addmiddleexit.py) makes it impossible to add a negative number to the sum variable.

Listing 5.5 (troubleshootloop.py) uses a variable named done that controls the duration of the loop. Listing 5.24 (troubleshootloop2.py) uses break statements in place of the Boolean done variable.

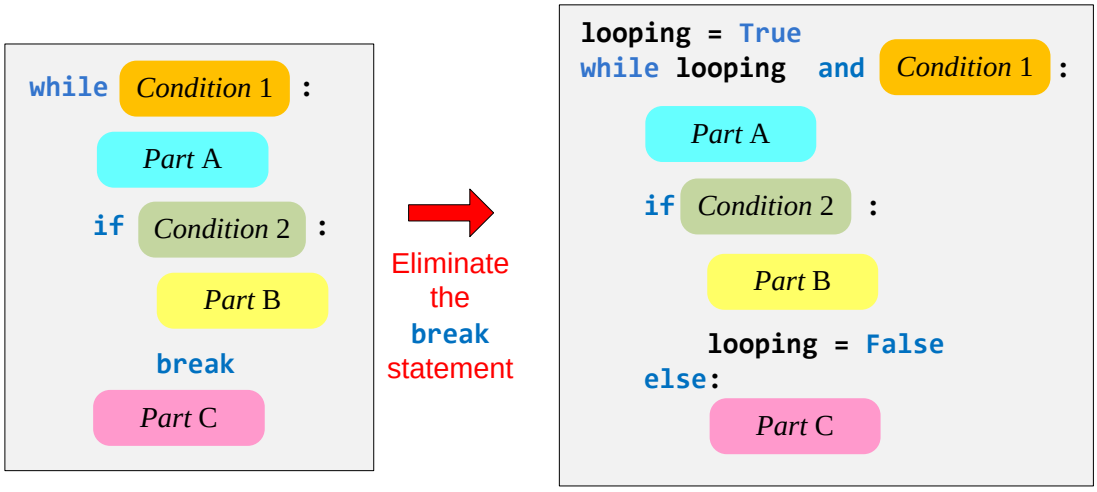
Listing 5.24: troubleshootloop2.py

```
print("Help!  My computer doesn't work!")
while True:
    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
                        print("Check the outlet's circuit ")
                        print("breaker or fuse.  Move to a")
                        print("new outlet, if necessary. ")
                    else:  # Beats me!
                        print("Please consult a service technician.")
                        break   # Nothing else I can do, exit loop
                else: # Check fuse
                    print("Check the fuse. Replace if ")
                    print("necessary.")
    else:  # The computer has power
        print("Please consult a service technician.")
        break   # Nothing else I can do, exit loop
```

Some software designers believe that programmers should use the break statement sparingly because it deviates from the normal loop control logic. Ideally, every loop should have a single entry point and single exit point. While Listing 5.23 (addmiddleexit.py) has a single exit point (the break statement), some programmers commonly use break statements within while statements in the which the condition for the while is not a tautology. Adding a break statement to such a loop adds an extra exit point (the top of the loop where the condition is checked is one point, and the break statement is another). Most programmers find two exits point perfectly acceptable, but much above two break points within a single loop is particularly dubious and you should avoid that practice.

The break statement is not absolutely required for full control over a while loop; that is, we can rewrite any Python program that contains a break statement within a while loop so that it behaves the same way but

Figure 5.4 The code on the left generically represents any `while` loop that uses a `break` statement. The code on the right shows how we can transform the loop into a functionally equivalent form that does not use `break`.



does not use a `break`. Figure 5.4 shows how we can transform any `while` loop that uses a `break` statement into a beak-free version. The no-`break` version introduces a Boolean variable (`looping`), and the loop control logic is a little more complicated. The no-`break` version uses more memory (an extra variable) and more time to execute (requires an extra check in the loop condition during every iteration of the loop). This extra memory is insignificant, and except for rare, specialized applications, the extra execution time is imperceptible. In most cases, the more important issue is that 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 is a desirable loop control option.

We can use the `break` statement inside a `for` loop as well. Listing 5.25 (countvowelsnox.py) shows how we can use a `break` statement to exit a `for` loop prematurely. provided by the user.

Listing 5.25: `countvowelsnox.py`

```
word = input('Enter text (no X\'s, please): ')
vowel_count = 0
for c in word:
    if c == 'A' or c == 'a' or c == 'E' or c == 'e' \
       or c == 'I' or c == 'i' or c == 'O' or c == 'o':
        print(c, ', ', sep='', end='')  # Print the vowel
        vowel_count += 1                # Count the vowel
    elif c == 'X' or c =='x':
        break
print(' (', vowel_count, ' vowels)', sep='')
```

If the program detects an *X* or *x* anywhere in the user's input string, it immediately exits the `for` even though it may not have considered all the characters in the string. Consider the following sample run:

```
Enter text (no X's, please): Mary had a lixtle lamb.
a, a, a, i,  (4 vowels)
```

The program breaks out of the loop when it attempts to process the *x* in the user's input.

The `break` statement is handy when a situation arises that requires immediate exit from a loop. The `for` loop in Python behaves differently from the `while` loop, in that it has no explicit condition that it checks to continue its iteration. We must use a `break` statement if we wish to prematurely exit a `for` loop before it has completed its specified iterations. The `for` loop is a *definite loop*, which means programmers can determine up front the number of iterations the loop will perform. The `break` statement has the potential to disrupt this predictability. For this reason, programmers use `break` statements in `for` loops less frequently, and they often serve as an escape from a bad situation that continued iteration might make worse.

5.5.2 The continue Statement

When a program's execution encounters a `break` statement inside a loop, it skips the rest of the body of the loop and exits the loop. The `continue` statement is similar to the `break` statement, except the `continue` statement does not necessarily exit the loop. The `continue` statement skips the rest of the body of the loop and immediately checks the loop's condition. If the loop's condition remains true, the loop's execution resumes at the top of the loop. Listing 5.26 (continueexample.py) shows the `continue` statement in action.

Listing 5.26: `continueexample.py`

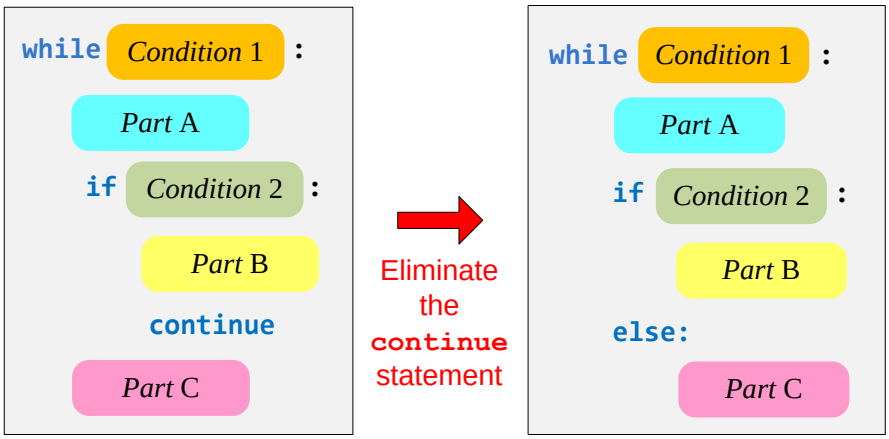
```
sum = 0
done = False
while not done:
    val = int(input("Enter positive integer (999 quits):"))
    if val < 0:
        print("Negative value",  val,  "ignored")
        continue  # Skip rest of body for this iteration
    if val != 999:
        print("Tallying", val)
        sum += val
    else:
        done = (val == 999)   # 999 entry exits loop
print("sum =", sum)
```

Programmers do not use the `continue` statement as frequently as the `break` statement since it is very easy to transform the code that uses `continue` into an equivalent form that does not. Listing 5.27 (nocontinueexample.py) works exactly like Listing 5.26 (continueexample.py), but it avoids the `continue` statement.

Listing 5.27: `nocontinueexample.py`

```
sum = 0
done = False
while not done:
    val = int(input("Enter positive integer (999 quits):"))
    if val < 0:
        print("Negative value",  val,  "ignored")
    else:
        if val != 999:
```

Figure 5.5 The code on the left generically represents any loop that uses a `continue` statement. It is possible to transform the code on the left to eliminate the `continue` statement, as the code on the right shows.




```
            print("Tallying", val)
            sum += val
        else:
            done = (val == 999)   # 999 entry exits loop
print("sum =", sum)
```

Figure 5.5 shows how we can rewrite any program that uses a `continue` statement into an equivalent form that does not use `continue`. The transformation is simpler than for `break` elimination (see Figure 5.4) since the loop's condition remains the same, and no additional variable is needed. 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 programmer may add a `continue` statement 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, the programmer can add a conditional statement with a `continue` near the top of the loop body without touching the logic of the rest of the loop. The `continue` statement thus merely provides a convenient alternative for the programmer. The `else` version is preferred.

5.6 while/else and for/else

Python loops support an optional `else` block. The `else` block in the context of a loop provides code to execute when the loop exits normally. Said another way, the code in a loop's `else` block does not execute if the loop terminates due to a `break` statement.

When a `while` loop exits due to its condition being false during its normal check, its associated `else` block executes. This is true even if its condition is found to be false before its body has had a chance to execute. Listing 5.28 (whileelse.py) shows how the `while`/`else` statement works.

Listing 5.28: `whileelse.py`


```
#  Add five nonnegative numbers supplied by the user
count = sum = 0
print('Please provide five nonnegative numbers when prompted')
while count < 5:
    # Get value from the user
    val = float(input('Enter number: '))
    if val < 0:
        print('Negative numbers not acceptable!  Terminating')
        break
    count += 1
    sum += val
else:
    print('Average =', sum/count)
```

When the user behaves and supplies only nonnegative values to Listing 5.28 (whileelse.py), it computes the average of the values provided:

```
Please provide five nonnegative numbers when prompted
Enter number: 23
Enter number: 12
Enter number: 14
Enter number: 10
Enter number: 11
Average = 14.0
```

When the user does not comply with the instructions, the program will print a corrective message and not attempt to compute the average:

```
Please provide five nonnegative numbers when prompted
Enter number: 23
Enter number: 12
Enter number: -4
Negative numbers not acceptable!  Terminating
```

It may be more natural to read the `else` keyword for the `while` statement as "if no `break`," meaning execute the code in the `else` block if the program's execution of code in the `while` block did not encounter the `break` statement.

The `else` block is not essential; Listing 5.29 (whilenoelse.py) uses `if`/`else` statement to achieve the same effect as Listing 5.28 (whileelse.py).

Listing 5.29: `whilenoelse.py`

```
#  Add five nonnegative numbers supplied by the user
count = sum = 0
print('Please provide five nonnegative numbers when prompted')
while count < 5:
    # Get value from the user
    val = float(input('Enter number: '))
    if val < 0:
        break
    count += 1
    sum += val
if count < 5:
```

```
    print('Negative numbers not acceptable!  Terminating')
else:
    print('Average =', sum/count)
```

Listing 5.29 (whilenoelse.py) uses two distinct Python constructs, the `while` statement followed by an `if/else` statement, whereas Listing 5.28 (whileelse.py) uses only one, a `while/else` statement. Listing 5.29 (whilenoelse.py) also must check the `count < 5` condition twice, once in the `while` statement and again in the `if/else` statement.

A `for` statement with an `else` block works similarly to the `while/else` statement. When a `for/else` loop exits because it has considered all the values in its range or all the characters in its string, it executes the code in its associated `else` block. If a `for/else` statement exits prematurely due to a `break` statement, it does not execute the code in its `else` block. Listing 5.30 (countvowelselse.py) shows how the `else` block works with a `for` statement.

Listing 5.30: countvowelselse.py

```
word = input('Enter text (no X\'s, please): ')
vowel_count = 0
for c in word:
    if c == 'A' or c == 'a' or c == 'E' or c == 'e' \
       or c == 'I' or c == 'i' or c == 'O' or c == 'o':
        print(c, ', ', sep='', end='')  # Print the vowel
        vowel_count += 1                # Count the vowel
    elif c == 'X' or c =='x':
        print('X not allowed')
        break
else:
    print(' (', vowel_count, ' vowels)', sep='')
```

Unlike Listing 5.15 (countvowels.py), Listing 5.30 (countvowelselse.py), does not print the number of vowels if the user supplies text containing and *X* or *x*.

5.7 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 sometimes are by design. For example, a long-running server application like a Web server may need to continuously check for incoming connections. The Web server can perform this checking within a loop that runs indefinitely. Beginning programmers, unfortunately, all too often create infinite loops by accident, and these infinite loops represent logic 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 Chapter 6) 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 5.31 (findfactors.py) that attempts to print all the integers with their associated factors from 1 to 20.

Listing 5.31: findfactors.py

```
#  List the factors of the integers 1...MAX
MAX = 20                            # MAX is 20
n = 1   # Start with 1
while n <= MAX:                     # Do not go past MAX
    factor = 1                      # 1 is a factor of any integer
    print(end=str(n) + ': ')        # Which integer are we examining?
    while factor <= n:              # Factors are <= the number
        if n % factor == 0:         # Test to see if factor is a factor of n
            print(factor, end=' ')  # If so, display it
            factor += 1             # Try the next number
    print()  # Move to next line for next n
    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 program displays the first factor of 3 properly and then 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 5.31 (findfactors.py) the debugging task is nontrivial since the program involves nested loops. (Can you find and fix the problem in Listing 5.31 (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, the statement

  ```
  while i >= 1 or i <= 10:
      # Block of code follows ...
  ```

  would produce an infinite loop since any value chosen for `i` will satisfy one or both of the two subconditions. Perhaps the programmer intended to use `and` instead of `or` to stay in the loop as long as `i` remains in the range 1...10.

  In Listing 5.31 (findfactors.py) the outer loop condition is

  ```
  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:

  ```
  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 the body must be able to modify one of the variables used in the condition. Eventually the variable assumes a value that makes the condition false, and the loop terminates.

  In Listing 5.31 (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:

```
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
```


We can use 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. We can augment the original inner loop in this way:

```
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 the program increment `factor` 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 5.32 (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 5.32: `findfactorsfor.py`

```
#  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
```


As a final note on infinite loops, Section 1.4 mentioned the preference for using the *Debug* option under the *WingIDE-101* integrated development environment when running our programs. When executing the program under the *Run* option, the IDE can become unresponsive if the program encounters an infinite loop. At that point, terminating the IDE is the only solution. Under the debugger, we very easily can interrupt a wayward program's execution via *WingIDE-101*'s *Stop* action.

5.8 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.

5.8.1 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 simple strategy:

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 5.33 (computesquareroot.py) implements the above strategy in Python, providing the missing details.

Listing 5.33: computesquareroot.py

```
#  File computesquareroot.py

#  Get value from the user
val = float(input('Enter number: '))
#  Compute a provisional square root
root = 1.0

#  How far off is our provisional root?
diff = root*root - val

#  Loop until the provisional root
#  is close enough to the actual root
while diff > 0.00000001 or diff < -0.00000001:
    print(root, 'squared is', root*root)  # Report how we are doing
    root = (root + val/root) / 2          # Compute new provisional root
    # How bad is our current approximation?
    diff = root*root - val

#  Report approximate square root
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.

The following shows the program's output when the user enters the value 2:

```
Enter number: 2
1.0 squared is 1.0
1.5 squared is 2.25
1.4166666666666665 squared is 2.006944444444444
1.4142156862745097 squared is 2.0000060073048824
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
1.0 squared is 1.0
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
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 5.33 (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 explore it and other handy mathematical functions in Chapter 6.

5.8.2 Drawing a Tree

Suppose we wish to draw a triangular tree with its height 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 5.34 (startree.py) provides the necessary functionality.

Listing 5.34: startree.py

```
#  Get tree height from user
height = int(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
```

The following shows a sample run of Listing 5.34 (startree.py) where the user enters 7:

```
Enter height of tree: 7
       *
      ***
     *****
    *******
   *********
  ***********
 *************
```

Listing 5.34 (startree.py) uses two sequential `while` loops nested within a `while` loop. The outer `while` loop draws one row of the tree each time its body executes:

- As long as the user enters a value greater than zero, the body of the outer `while` loop will execute; 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 it prints 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 it prints no left side asterisks, one central asterisk, and no right side asterisks. Each time through the loop the number of left-hand and right-hand stars to print both increase by one, but there remains just one central asterisk to print. This means the tree grows one wider on each side for 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 5.35 (startreefor.py) uses `for` loops instead of `while` loops to draw our star trees. The `for` loop is a better choice for this program since once the user provides the height, the program can calculate exactly the number of iterations required for each loop. This number will not change during the rest of the program's execution, so the definite loop (`for`) is better a better choice than the indefinite loop (`while`).

Listing 5.35: `startreefor.py`

```
#  Get tree height from user
height = int(input('Enter height of tree: '))

#  Draw one row for every unit of height
for row in range(height):
    # Print leading spaces; as row gets bigger, the number of
    # leading spaces gets smaller
    for count in range(height - row):
        print(end=' ')

    # 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
    for count in range(2*row + 1):
        print(end='*')
    # Move cursor down to next line
    print()
```

5.8.3 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 29 with no remainder), but 28 is not (1, 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 5.36 (printprimes.py) provides one solution.

Listing 5.36: printprimes.py

```
max_value = int(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 5.36 (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 5.36 (printprimes.py) is a little more complex than that of Listing 5.34 (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 program initializes the `is_prime` variable to true, meaning it assumes `value` is a prime number unless later tests prove otherwise. `trial_factor` takes on all the values from two to `value - 1` in the inner loop:

  ```
  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 the program discovers a value of `trial_factor` that actually is a factor of `value`, then it sets `is_prime` false and exits the loop via the `break` statement. If the loop continues to completion, the program will not set `is_prime` to false, which means it found no factors, and, so, `value` is indeed prime.

- The `if` statement after the inner loop:

  ```
  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 the program prints `value` followed by a space to separate it from other factors that it may print during the remaining iterations.

Some important questions must be answered:

1. **If the user enters a 2, what will the program print?**

   In this case `max_value` = `value` = 2, so the condition of the outer loop

   ```
   value <= max_value
   ```

   is true, since $2 \leq 2$. The executing program sets `is_prime` to true, but the condition of the inner loop

   ```
   trial_factor < value
   ```

   is not true (2 is not less than 2). Thus, the program skips the inner loop, it does not change `is_prime` from true, and so it prints 2. 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, what will the program print?**

   The `while` condition ensures that values less than two are not considered. The program will never enter the body of the `while`. The program prints only the newline, and it displays no numbers. This behavior is correct, as there are no primes numbers less than 2.

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. The last statement within the body of the outer loop increases `value`, and no where else does the program modify `value`. 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.

We can rearrange slightly the logic of the inner `while` 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
```

We can be rewrite it 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 5.36 (printprimes.py) a bit by using `for` statements instead of `while` statements as shown in Listing 5.37 (printprimesfor.py).

Listing 5.37: printprimesfor.py

```
max_value = int(input('Display primes up to what value? '))
#  Try values from 2 (smallest prime number) to max_value
for value in range(2, max_value + 1):
    # See if value is prime
    is_prime = True  # Provisionally, value is prime
    # Try all possible factors from 2 to value - 1
    for trial_factor in range(2, value):
        if value % trial_factor == 0:
            is_prime = False    # Found a factor
            break               # No need to continue; it is NOT prime
    if is_prime:
        print(value, end= ' ')  # Display the prime number
print()  # Move cursor down to next line
```

We can simplify Listing 5.37 (printprimesfor.py) even further by using the `for`/`else` statement as Listing 5.38 (printprimesforelse.py) illustrates.

Listing 5.38: printprimesforelse.py

```
max_value = int(input('Display primes up to what value? '))
#  Try values from 2 (smallest prime number) to max_value
for value in range(2, max_value + 1):
    # See if value is prime: try all possible factors from 2 to value - 1
    for trial_factor in range(2, value):
        if value % trial_factor == 0:
            break   # Found a factor, no need to continue; it is NOT prime
    else:
        print(value, end= ' ')  # Display the prime number
print()  # Move cursor down to next line
```

If the inner `for` loop completes its iteration over all the values in its range, it will execute the print statement in its `else` block. The only way the inner `for` loop can be interrupted is if it discovers a factor of `value`. If it does find a factor, the premature exit of the inner `for` loop prevents the execution of its `else` block. This logic enables it to print only prime numbers—exactly the behavior we want.

5.8.4 Insisting on the Proper Input

Listing 5.39 (betterinputonly.py) traps the user in a loop until the user provides an acceptable integer value.

Listing 5.39: betterinputonly.py

```
#  Require the user to enter an integer in the range 1-10
in_value = 0    # Ensure loop entry
attempts = 0    # Count the number of tries

#  Loop until the user supplies a valid number
while in_value < 1 or in_value > 10:
    in_value = int(input("Please enter an integer in the range 0-10: "))
    attempts += 1

#  Make singular or plural word as necessary
tries = "try" if attempts == 1 else "tries"
#  in_value at this point is guaranteed to be within range
print("It took you", attempts, tries, "to enter a valid number")
```

A sample run of Listing 5.39 (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. A definite loop (`for`) is inappropriate for a program like Listing 5.39 (betterinputonly.py) because the program cannot determine ahead of time how many attempts the user will make before providing a value in range.

5.9 Exercises

1. In Listing 5.4 (addnonnegatives.py) could the condition of the `if` statement have used > instead of >= and achieved the same results? Why?

2. In Listing 5.4 (addnonnegatives.py) could the condition of the `while` statement have used > instead of >= and achieved the same results? Why?

3. In Listing 5.4 (addnonnegatives.py) what would happen if the statement

   ```
   entry = int(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 += 1
        b += 1
    a += 1
print()
```

11. What is minimum number of arguments acceptable to the `range` expression?

12. What is maximum number of arguments acceptable to the `range` expression?

13. Provide the exact sequence of integers specified by each of the following `range` expressions.

    (a) `range(5)`
    (b) `range(5, 10)`
    (c) `range(5, 20, 3)`
    (d) `range(20, 5, -1)`
    (e) `range(20, 5, -3)`
    (f) `range(10, 5)`
    (g) `range(0)`
    (h) `range(10, 101, 10)`
    (i) `range(10, -1, -1)`
    (j) `range(-3, 4)`
    (k) `range(0, 10, 1)`

14. What is a shorter way to express `range(0, 5, 1)`?

15. Provide an equivalent Python `range` expression for each of the following integer sequences.

    (a) $1, 2, 3, 4, 5$
    (b) $5, 4, 3, 2, 1$
    (c) $5, 10, 15, 20, 25, 30$
    (d) $30, 25, 20, 15, 10, 5$
    (e) $-3, -2, -1, 0, 1, 2, 3$
    (f) $3, 2, 1, 0, -1, -2, -3$
    (g) $-50, -40, -30, -20, -10$
    (h) *Empty sequence*

16. If `x` is bound to the integer value 2, what integer sequence does `range(x, 10*x, x)` represent?

17. If `x` is bound to the integer value 2 and `y` is bound to the integer 5, what integer sequence does `range(x, x + y)` represent?

18. Is it possible to represent the following sequence with a Python `range` expression: $1, -1, 2, -2, 3, -3, 4, -4$?

19. How many asterisks does the following code fragment print?

```
for a in range(100):
    print('*', end='')
print()
```

20. How many asterisks does the following code fragment print?

```
for a in range(20, 100, 5):
    print('*', end='')
print()
```

21. How many asterisks does the following code fragment print?

```
for a in range(100, 0, -2):
    print('*', end='')
print()
```

22. How many asterisks does the following code fragment print?

```
for a in range(1, 1):
    print('*', end='')
print()
```

23. How many asterisks does the following code fragment print?

```
for a in range(-100, 100):
    print('*', end='')
print()
```

24. How many asterisks does the following code fragment print?

```
for a in range(-100, 100, 10):
    print('*', end='')
print()
```

25. Rewrite the code in the previous question so it uses a `while` instead of a `for`. Your code should behave identically.

26. What does the following code fragment print?

```
a = 0
while a < 100:
    print(a)
    a += 1
print()
```

27. Rewrite the code in the previous question so it uses a `for` instead of a `while`. Your code should behave identically.

28. What is printed by the following code fragment?

```
a = 0
while a > 100:
    print(a)
    a += 1
print()
```

29. 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 = int(input())
    if n < 0:
        done = True
    print("n =", n)
```

30. 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 = 5
while x > 0:
    y = int(input())
    if y == 25:
        continue
    x -= 1
    print('x =', x)
```

31. What is printed by the following code fragment?

```
a = 0
while a < 100:
    print(a, end=' ')
    a += 1
print()
```

32. Modify Listing 5.19 (timestable4.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.

33. 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 $*$ 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.

34. 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.

35. Write a Python program that allows the user to enter any number of nonnegative 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.

36. Redesign Listing 5.34 (startree.py) so that it draws a sideways tree pointing right; for example, if the user enters 7, the program would print

```
*
**
***
****
*****
******
*******
******
*****
****
***
**
*
```

37. Redesign Listing 5.34 (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 5.33 (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 the code is this: What if you are working on a significant scientific or engineering application and must use different formulas in various parts of the source code, and each of these formulas involve square roots in some way? In mathematics, for example, we use square root to compute the distance between two geometric points (x_1, y_1) and (x_2, y_2) as

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

and, using the quadratic formula, the solution to the equation $ax^2 + bx + c = 0$ is

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In electrical engineering and physics, the root mean square of a set of values $\{a_1, a_2, a_3, \ldots, a_n\}$ is

$$\sqrt{\frac{a_1^2 + a_2^2 + a_3^2 + \ldots + a_n^2}{n}}$$

Suppose we are writing one big program that, among many other things, needs compute distances and solve quadratic equations. Must we copy and paste the relevant portions of our square root code found in Listing 5.33 (computesquareroot.py) to each location in our source code that requires a square root computation? Also, what if we develop another program that requires computing a root mean square? Will we need to copy the code from Listing 5.33 (computesquareroot.py) into every program that needs to compute square roots, or is there a better way to package the square root code and reuse it?

One way to make code more reusable is 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 functions stored in libraries called *modules*. Programmers can use the functions from these libraries within their own code to build sophisticated programs.

Figure 6.1 Conceptual view of the square root function. The function is like a black box—callers do not need to know the details of the code inside the function in order to use it.

6.1 Introduction to Using Functions

We have been using functions in Python since the first chapter. These functions include `print`, `input`, `int`, `float`, `str`, 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 $f(5) = 2 \cdot 5 + 13 = 13$ and $f(0) = 2 \cdot 0 + 3 = 3$. 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. If an executing program needs to perform such a task, it calls upon the function to do the work. One example of a function is the mathematical square root function. Python has a function in its standard library named `sqrt` (see Section 6.4). 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. The square root function is like a black box to the code that uses it. Callers do not need to know the details of the code inside the function in order to use it. Programmers are concerned more about *what* the function does, not *how* it does it.

This `sqrt` function is exactly what we need for our square root program, Listing 5.33 (computesquareroot.py). The new version, Listing 6.1 (standardsquareroot.py), uses the library function `sqrt`, eliminating the complex logic of the original code.

Listing 6.1: `standardsquareroot.py`

```
from math import sqrt

#  Get value from the user
num = float(input("Enter number: "))

#  Compute the square root
root = sqrt(num)

#  Report result
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`.

Unlike the other functions we have used earlier, 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`, and `str`) always available to Python programs. The `sqrt` function is part of separate *module* within the standard library. A module is a collection of Python code that can used in other programs. The `import` keyword makes a module available to the interpreter. The first statement in Listing 6.1 (standardsquareroot.py) shows one way to use the `import` keyword:

```
from math import sqrt
```

This statement 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. This `import` statement will make *only* the `sqrt` function available to the program.

When calling a function, a pair of parentheses follow the function's name. Information that the function requires to perform its task must appear within these parentheses. 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. We also can say "we are passing `num` to the `sqrt` function." The function uses the variable `num`'s value to perform the computation. Parameters enable callers to communicate information to a function during the function's execution.

The program could call the `sqrt` function in many other ways, as Listing 6.2 (usingsqrt.py) illustrates.

Listing 6.2: `usingsqrt.py`

```
# This program shows the various ways the
# sqrt function can be used.

from math import sqrt

x = 16
#  Pass a literal value and display the result
print(sqrt(16.0))
#  Pass a variable and display the result
print(sqrt(x))
#  Pass an expression
print(sqrt(2 * x - 5))
#  Assign result to variable
y = sqrt(x)
print(y)
#  Use result in an expression
y = 2 * sqrt(x + 16) - 4
print(y)
#  Use result as argument to a function call
y = sqrt(sqrt(256.0))
print(y)
print(sqrt(int('45')))
```


The sqrt function accepts a single numeric argument. As Listing 6.2 (usingsqrt.py) shows, the parameter that a caller 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 it returns to the caller. The caller can use this returned value 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 $\sqrt{256}$ via the inner sqrt call and immediately passes the result to the outer sqrt call. This composition of function calls computes $\sqrt{\sqrt{256}} = \sqrt{16} = 4$, and the assignment operator binds the variable y to 4. The statement

```
print(sqrt(int('45')))
```

prints the result of computing the square root of the integer created from the string '45'.

If the calling code attempts to pass a parameter to a function that is incompatible with type expected by that function, the interpreter issues an error. Consider:

```
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, the square root function is a black box to the caller. The caller 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. Ordinarily 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 caller passes data to a function, the function cannot affect the caller's copy of that data. The caller is, however, free to

use the return value of function to modify any of its variables. The important distinction is that the caller is modifying its own variables—the function is not modifying the caller's variables. The following interactive sequence demonstrates that the `sqrt` function does not affect the object passed to it:

```
>>> from math import sqrt
>>> x = 2
>>> sqrt(x)
1.4142135623730951
>>> x
2
>>> x = sqrt(x)
>>> x
1.4142135623730951
```

Observe that passing `x` to the `sqrt` function did not change `x`, it still refers to the integer object 2. We must reassign `x` to the value that `sqrt` returns if we really wish to change `x` to be its square root.

Some functions take more than one parameter; for example, `print` can accept multiple parameters separated by commas.

From the caller'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` permit callers to pass a variable number of arguments, but most functions, like `sqrt`, specify an exact number. If a caller 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 caller passes 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 caller. The caller'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. The `sqrt` function computes and returns a floating-point value; the interactive shell reports

```
>>> type(sqrt(16.0))
<class 'float'>
```

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` module. The `random` function returns a floating-point value, but the caller does not pass the function any information to do its task. Any attempts to do so will fail:

```
>>> random(20)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
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 caller. 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 caller. Since Python requires that all functions return a value, `print` must return something. Functions that are not meant to return anything return the special object `None`. We can show this in the Python shell:

```
>>> print(print(4))
4
None
```

The inner `print` call prints 4, and the outer `print` displays the return value of the inner `print` call.

We can assign the value `None` to any variable. It represents "nothing" or "no object."

6.2 Functions and Modules

A Python *module* is simply a file that contains Python code. The name of the file dictates the name of the module; for example, a file named math.py contains the functions available from the standard `math` module. The modules of immediate interest to us are the standard modules that contain functions that our programs can use. The Python standard library contains thousands of functions distributed throughout more than 230 modules. These modules cover a wide range of application domains. One of the modules, known as the built-ins module (actual name `__builtins__`), contains all the functions we have been using in earlier chapters: `print`, `input`, etc. These built-in functions make up only a very small fraction of all the functions the standard library provides. Programmers must use one or more `import` statements within a program or within the interactive interpreter to gain access to the remaining standard functions.

Figure 6.2 General form of a statement that imports a subset of a module's available functions. The *function list* is a comma-separated list of function names to import.

The Python distribution for a given platform stores these standard modules somewhere on the computer's hard drive. The interpreter knows where to locate these standard modules when an executing program needs to import them. It is not uncommon for a complex program to import a dozen separate modules to obtain all functions it needs to do its job.

Python provides a number of ways to import functions from a module. We will concentrate on the two most commonly used techniques. Section 6.10 briefly examines other, more specialized `import` statements.

Listing 6.1 (standardsquareroot.py) imported the square root function as follows:

```
from math import sqrt
```

A program that needs to compute square roots, common logarithms, and the trigonometric cosine function could use the following import statement:

```
from math import sqrt, log10, cos
```

This statement makes only the `sqrt`, `log10`, and `cos` functions from the `math` module available to the program. The `math` module offers many other mathematical functions—for example, the `atan` function that computes the arctangent—but this limited import statement does not provide these other definitions to the interpreter. Figure 6.2 shows the general form of this kind of import statement. Such an import statement is appropriate for smaller Python programs that use a small number of functions from a module. This kind of import statement allows callers to use an imported function's simple name, as in

```
y = sqrt(x)
```

If a program requires many different functions from a module, listing them all individually can become unwieldy. Python provides a way to import everything a module has to offer, as we will see in Section 6.10, but we also will see why this practice is not desirable.

Rather than importing one or more components of a module, we can import the entire module, as shown here:

```
import math
```

Figure 6.3 shows the general form of this kind of import statement. This import statement makes all of the functions of the module available to the program, but in order to use a function the caller must attach the module's name during the call. The following code demonstrates the call notation:

```
y = math.sqrt(x)
print(math.log10(100))
```

Note the `math.` prefix attached to the calls of the `sqrt` and `log10` functions. We call a composite name (*module-name*.*function-name*) like this a *qualified name*. The qualified name includes the module name

Figure 6.3 General form of a statement that imports an entire module. The *module list* is a comma-separated list of module names to import.

```
import module list
```

and function name. Many programmers prefer this approach because the complete name unambiguously identifies the function with its module. A large, complex program could import the `math` module and a different, third-party module called `extramath`. Suppose the `extramath` module provided its own `sqrt` function. There can be no mistaking the fact that the `sqrt` being called in the expression `math.sqrt(16)` is the one provided by the `math` module. It is impossible for a program to import the `sqrt` functions separately from both modules and use their simple names simultaneously within a program. Does

```
y = sqrt(x)
```

intend to use `math`'s `sqrt` or `extramath`'s `sqrt`?

Note that a statement such as

```
from math import sqrt
```

does not import the entire module; specifically, code under this import statement may use only the simple name, `sqrt`, and cannot use the qualified name, `math.sqrt`.

As programs become larger and more complex, the import entire module approach becomes more compelling. The qualified function names improve the code's readability and avoids name clashes between modules that provide functions with identical names. Soon we will be writing our own, custom functions. Qualified names ensure that names we create ourselves will not clash with any names that modules may provide.

6.3 The Built-in Functions

Section 6.1 observed that we have been using functions in Python since the first chapter. These functions include `print`, `input`, `int`, `float`, `str`, and `type`. These functions and many others reside in a module named `__builtins__`. The `__builtins__` module is special because its components are automatically available to any Python program with—no `import` statement is required. The full name of the `print` function is `__builtins__.print`, although chances are you will never see its full name written in a Python program. We can verify its fully qualified name in the interpreter:

```
>>> print('Hi')
Hi
>>> __builtins__.print('Hi')
Hi
>>> print
<built-in function print>
>>> __builtins__.print
<built-in function print>
>>> id(print)
```



```
9506056
>>> id(__builtins__.print)
9506056
```

This interactive sequence verifies that the names `print` and `__builtins__.print` refer to the same function object. The `id` function is another `__builtins__` function. The expression `id(x)` evaluates to the address in memory of object named `x`. Since `id(print)` and `id(__builtins__.print)` evaluate to the same value, we know both names correspond to the same function object.

The `dir` function, which stands for *directory*, reveals all the components that a module has to offer. The following interactive sequence prints the `__builtins__` components:

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError', 'BufferError', 'Bytes Warning',
'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError', 'Depre cationWarning',
'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
'FileExistsError', 'FileNotFoundError', 'FloatingP ointError',
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
'ImportWarning', 'IndentationError', 'IndexError', 'Interrup tedError',
'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
'MemoryError', 'NameError', 'None', 'NotADirectoryEr ror',
'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'PermissionError', 'ProcessL ookupError',
'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEn codeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'WindowsError', 'Zero DivisionError',
'__build_class__', '__debug__', '__doc__', '__import__', '__loader__',
'__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii',
'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod',
'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir',
'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float',
'format', 'frozenset', 'getattr ', 'globals', 'hasattr', 'hash', 'help',
'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len',
'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next',
'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',
'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type',
'vars', 'zip']
>>>
```

A module can contain other things besides functions. Most of the names in the first 18 lines or so of the `__builtins__` module's directory listing are types the module defines, and most of the names in the last 11 lines are functions it provides. The list contains many recognizable names: `dir`, `bool`, `float`, `id`, `input`, `int`, `print`, `range`, `round`, `str`, and `type` functions.

The `__builtins__` module provides a common core of general functions useful to any Python program regardless of its application area. The other standard modules that Python provides are aimed at specific application domains, such as mathematics, text processing, file processing, system administration, graphics, and Internet protocols, and multimedia. Programs that require more domain-specific functionality must

import the appropriate modules that provide the needed services.

The `__builtins__` module includes a `help` function. In the interactive interpreter we can use the `help` function to print human-readable information about specific functions in the current namespace. The following interactive sequence shows how `help` works:

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.

>>> help(input)
Help on built-in function input in module builtins:

input(...)
    input([prompt]) -> string

    Read a string from standard input.  The trailing newline is stripped.
    If the user hits EOF (Unix: Ctl-D, Windows: Ctl-Z+Return), raise EOFError.
    On Unix, GNU readline is used if enabled.  The prompt string, if given,
    is printed without a trailing newline before reading.

>>> help(sqrt)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
>>> help(math.sqrt)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
>>> import math
>>> help(math.sqrt)
Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.
```

Notice that `help` was powerless to provide information about `math.sqrt` function until we imported the `math` module.

6.4 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.

Table 6.1 A few of the functions from the `math` module

`math` Module
`sqrt` Computes the square root of a number: `sqrt(`x`)` = $\sqrt{x}$
`exp` Computes e raised a power: `exp(`x`)` = e^x
`log` Computes the natural logarithm of a number: `log(`x`)` = $\log_e x = \ln x$
`log10` Computes the common logarithm of a number: `log(`x`)` = $\log_{10} x$
`cos` Computes the cosine of a value specified in radians: `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: `pow(`x, y`)` = x^y
`degrees` Converts a value in radians to degrees: `degrees(`x`)` = $\frac{\pi}{180}x$
`radians` Converts a value in degrees to radians: `radians(`x`)` = $\frac{180}{\pi}x$
`fabs` Computes the absolute value of a number: `fabs(`x`)` = $\lvert x \rvert$

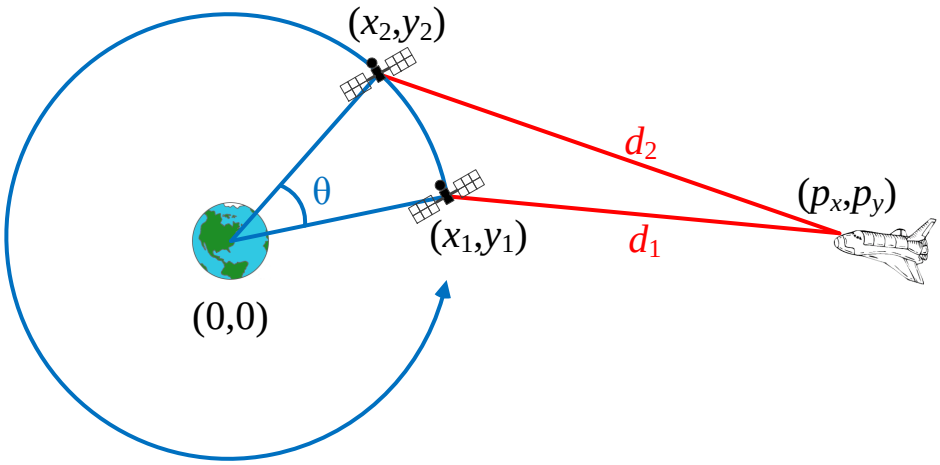
The `math` module also defines the values `pi` (π) and `e` (e). The following interactive sequence reveals the `math` module's full directory of components:

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'g amma', 'hypot',
'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'log2', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'trunc']
>>>
```

Most of the names in the directory represent functions.

The parameter passed by the caller 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; for example, the call `math.pow(10,2)` computes $10^2 = 100$, but the call `math.pow(2,10)` computes $2^{10} = 1,024$.

Figure 6.4 Orbital distance problem. In this diagram, the satellite begins at point (x_1, y_1), a distance of d_1 from the spacecraft. The satellite's orbit takes it to point (x_2, y_2) after an angle of θ rotation. The distance to its new location is d_2.



A Python program that uses any of these mathematical functions must import the `math` module.

The functions in the `math` module are ideal for solving problems like the one shown in Figure 6.4. 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 θ degrees along its orbital path.

We will let the origin of our coordinate system (0,0) be located at the center of the planet. This location corresponds also to the center of the satellite's circular orbital path. The satellite is located as some point, (x, y) 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 wish to compute the distances between the moving point (satellite) and the fixed point (spacecraft) as the satellite orbits the planet.

Facts from mathematics provide solutions to the following two problems:

1. **Problem**: We must recompute the location of the moving point as it moves along the circle.

   Solution: Given an initial position (x, y) of a point, a rotation of θ degrees around the origin will yield a new point at (x', y'), where

   $$\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \end{aligned}$$

2. **Problem**: We must recalculate the distance between the moving point and the fixed point as the moving point moves to a new position.

   Solution: The distance d in Figure 6.4 between the two points (p_x, p_y) and (x, y) is given by the formula

   $$d = \sqrt{(x - p_x)^2 + (y - p_y)^2}$$

Listing 6.3 (orbitdist.py) uses these mathematical results to compute a table of distances that span a complete orbit of the satellite.

Listing 6.3: orbitdist.py

```
#  Use some functions and values from the math module
from math import sqrt, sin, cos, pi, radians

#  Get coordinates of the stationary spacecraft, (px, py)
px = float(input("Enter x coordinate of spacecraft: "))
py = float(input("Enter y coordinate of spacecraft: "))

#  Get starting coordinates of satellite, (x1, y1)
x = float(input("Enter initial satellite x coordinate: "))
y = float(input("Enter initial satellite y coordinate: "))

#  Convert 60 degrees to radians to be able to use the trigonometric functions
rads = radians(60)

#  Precompute the cosine and sine of the angle
COS_theta = cos(rads)
SIN_theta = sin(rads)

#  Make a complete revolution (6*60 = 360 degrees)
for increment in range(0, 7):
    # Compute the distance to the satellite
    dist = sqrt((px - x)*(px - x) + (py - y)*(py - y))
    print('Distance to satellite {0:10.2f}  km'.format(dist))
    # Compute the satellite's new (x, y) location after rotating by 60 degrees
    x, y = x*COS_theta - y*SIN_theta, x*SIN_theta + y*COS_theta
```

Listing 6.3 (orbitdist.py) prints the distances from the spacecraft to the satellite in 60-degree orbit increments. A sample run of Listing 6.3 (orbitdist.py) looks like

```
Enter x coordinate of spacecraft: 100000
Enter y coordinate of spacecraft: 0
Enter initial satellite x coordinate: 20000
Enter initial satellite y coordinate: 0
Distance to satellite   80000.00  km
Distance to satellite   91651.51  km
Distance to satellite  111355.29  km
Distance to satellite  120000.00  km
Distance to satellite  111355.29  km
Distance to satellite   91651.51  km
Distance to satellite   80000.00  km
```

Here, the user first enters the point $(100,000, 0)$ and then the tuple $(20,000, 0)$. Observe that the satellite begins 80,000 km away from the spacecraft and the distance increases to a maximum of 120,000 km when it is at the far side of its orbit. Eventually the satellite returns to its starting place ready for the next orbit.

Listing 6.3 (orbitdist.py) uses tuple assignment to update the x and y variables:

```
#  Uses tuple assignment
x, y = x*COS_theta - y*SIN_theta, x*SIN_theta + y*COS_theta
```


If we instead used two separate assignment statements, we must be careful—the following code does *not* work the same way:

```
#  Does not work correctly
x = x*COS_theta - y*SIN_theta
y = x*SIN_theta + y*COS_theta
```

This is because the value of x used in the second assignment statement is the *new* value of x computed by the first assignment statement. The tuple assignment version uses the original x value in both computations. If we really wanted to use two assignment statements rather than a single tuple assignment, we would need to introduce an extra variable so we do not lose x's original value:

```
new_x = x*COS_theta - y*SIN_theta  # Compute new x value
y = x*SIN_theta + y*COS_theta      # Compute new y value using original x
x = new_x                          # Update x
```

We can use the square root function to improve the efficiency of Listing 5.36 (printprimes.py). Instead of trying all the potential factors of n up to $n-1$, we need only try potential factors up to $\sqrt{n}$. Listing 6.4 (moreefficientprimes.py) uses the `sqrt` function to reduce the number of potential factors the program needs to consider.

Listing 6.4: moreefficientprimes.py

```
from math import sqrt

max_value = int(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
    root = sqrt(value)  # Compute the square root of value
    while trial_factor <= root:
        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
```

6.5 `time` Functions

The `time` module contains a number of functions that relate to time. We will consider two: `perf_counter` and `sleep`.

`time.perf_counter`. The `time.perf_counter` function allows us measure elapsed time as if we were using a stopwatch or looking at a clock. The `time.perf_counter` function returns a floating-point

value representing a point in time. Its return value by itself is not useful; it is only after a second call to `time.perf_counter` that we can extract any useful information. The difference between the first call to `time.perf_counter` and the second call to `time.perf_counter` represents an elapsed time in seconds; thus, with two calls to the `time.perf_counter` 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

```
from time import perf_counter

print("Enter your name: ", end="")
start_time = perf_counter()
name = input()
elapsed = perf_counter() - start_time
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

```
from time import perf_counter

sum = 0          # Initialize sum accumulator
start = perf_counter()  # Start the stopwatch
for n in range(1, 100000001):   #  Sum the numbers
    sum += n
elapsed = perf_counter() - start  #  Stop the stopwatch
print("sum:", sum, "time:", elapsed)  # Report results
```

On one system Listing 6.6 (timeaddition.py) reports

```
sum: 5000000050000000 time: 24.922694830903826
```

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 5.37 (printprimesfor.py).

Listing 6.7: measureprimespeed.py

```
from time import perf_counter

max_value = 10000
count = 0
start_time = perf_counter()   # Start timer
#  Try values from 2 (smallest prime number) to max_value
for value in range(2, max_value + 1):
    # See if value is prime
    is_prime = True  # Provisionally, value is prime
```

```
    # Try all possible factors from 2 to value - 1
    for trial_factor in range(2, value):
        if value % trial_factor == 0:
            is_prime = False    # Found a factor
            break               # No need to continue; it is NOT prime
    if is_prime:
        count += 1              # Count the prime number
print()  # Move cursor down to next line
elapsed = perf_counter() - start_time  # Stop the timer
print("Count:", count, "  Elapsed time:", elapsed, "sec")
```

On one system, the program produces

```
Count: 1229   Elapsed time: 1.6250698114336175 sec
```

Repeated runs consistently report an execution time of approximately 1.6 seconds 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 on average over 20 times faster. A sample run shows

```
Count: 1229   Elapsed time: 0.07575643612557352 sec
```

Exact times will vary depending on the speed of the computer.

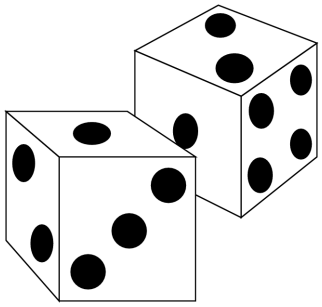
Listing 6.8: timemoreefficientprimes.py

```
from math import sqrt
from time import perf_counter

max_value = 10000
count = 0
value = 2        # Smallest prime number
start = perf_counter()  # Start the stopwatch
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
    root = sqrt(value)
    while trial_factor <= root:
        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:
        count += 1              # Count the prime number
    value += 1                  # Try the next potential prime number
elapsed = perf_counter() - start       # Stop the stopwatch
print("Count:", count, "  Elapsed time:", elapsed, "sec")
```

An even faster prime generator appears in Listing 10.24 (fasterprimes.py); it uses a completely different algorithm to generate prime numbers.

time.sleep. The time.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.

Figure 6.5 A pair of dice



Listing 6.9: countdown.py

```
from time import sleep

for count in range(10, -1, -1):  # Range 10, 9, 8, ..., 0
    print(count)      # Display the count
    sleep(1)          # Suspend execution for 1 second
```

The `time.sleep` function is useful for controlling the speed of graphical animations.

6.6 Random Numbers

Some applications require behavior that appears random. Random numbers are particularly useful in games and simulations. For example, many board games use a die (one of a pair of dice—see Figure 6.5) to determine how many places a player is to advance. 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.

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. All practical algorithmic pseudorandom number generators have periods that are large enough for most applications.

In addition to a long period, a good pseudorandom generator would be equally likely to generate any number in its range; that is, it would not be biased toward a subset of its possible values. Ideally, the numbers the generator produces will be uniformly distributed across its range of values.

The good news is that the Python standard library has a very good pseudorandom number generator based the *Mersenne Twister* algorithm. See `http://en.wikipedia.org/wiki/Mersenne_twister` for more information about the algorithm.

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.

Table 6.2 A few of the functions from the `random` module

`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.
`choice` Selects an element at random from a collection of elements.

The `random.seed` function establishes the initial value from which the sequence of pseudorandom numbers is generated. Each call to `random.random` or `random.randrange` returns the next value in the sequence of pseudorandom values. Listing 6.10 (simplerandom.py) prints 100 pseudorandom integers in the range 1...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, 1001), end=' ')      # Range 1...1,000, inclusive
print()                                     # Print newine
```

The numbers Listing 6.10 (simplerandom.py) prints appear to be random. The program begins its pseudorandom number generation with a seed value, 23. The seed value determines the exact sequence of numbers the program generates; identical seed values generate identical sequences. If you run the program again, it displays the same sequence. In order for the program to display different sequences, the seed value must be different for each run.

If we omit the call to the `random.seed` function, the program derives its initial value in the sequence from the time kept by the operating system. 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...7
    value = randrange(1, 7)

    # Show the die
    print("+-------+")
    if value == 1:
```

```
        print("|       |")
        print("|   *   |")
        print("|       |")
    elif value == 2:
        print("| *     |")
        print("|       |")
        print("|     * |")
    elif value == 3:
        print("|     * |")
        print("|   *   |")
        print("| *     |")
    elif value == 4:
        print("| *   * |")
        print("|       |")
        print("| *   * |")
    elif value == 5:
        print("| *   * |")
        print("|   *   |")
        print("| *   * |")
    elif value == 6:
        print("| * * * |")
        print("|       |")
        print("| * * * |")
    else:
        print(" ***  Error: illegal die value ***")
    print("+-------+")
```

The output of one run of Listing 6.11 (die.py) is

```
+-------+
| *   * |
|       |
| *   * |
+-------+
+-------+
| * * * |
|       |
| * * * |
+-------+
+-------+
|       |
|   *   |
|       |
+-------+
```

Since the program generates the values pseudorandomly, actual output will vary from one run to the next.

The `random` module provides a `randint` function that works similarly to `random.randrange`. The call `random.randint(a, b)` is equivalent to `random.randrange(a, b + 1)`.

We will find the `choice` function more useful after we examine Python's more advanced data structures like lists. As a preview, Listing 6.12 (randomstring.py) shows how we can select a random element from a tuple of strings.

Listing 6.12: `randomstring.py`


```
from random import choice

for i in range(10):
    print(choice(("one", "two", "three", "four", "five", "six",
                  "seven", "eight", "nine", "ten")))
```

One run of Listing 6.12 (randomstring.py) produces

```
three
nine
five
nine
three
seven
nine
two
eight
ten
```

The tuple of strings passed to the `choice` function must be enclosed within parentheses so it treated as one parameter (a tuple consisting of 10 strings) rather than 10 string parameters.

6.7 System-specific Functions

The `sys` module provides a number of functions and variables that give programmers access to system-specific information. One useful function is `exit` that terminates an executing program. Listing 6.13 (exitprogram.py) uses the `sys.exit` function to end the program's execution after it prints 10 numbers.

Listing 6.13: `exitprogram.py`

```
import sys

sum = 0
while True:
    x = int(input('Enter a number (999 ends):'))
    if x == 999:
        sys.exit(0)
    sum += x
    print('Sum is', sum)
```

The `sys.exit` function accepts a single integer argument, which it passed back to the operating system when the program completes. The value zero indicates that the program completed successfully; a nonzero value represents the program terminating due to an error of some kind.

6.8 The `eval` and `exec` Functions

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:

```
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?

The __builtins__ module provides an interesting function named eval that attempts to evaluate a string in the same way that the interactive shell would evaluate it. Listing 6.14 (evalfunc.py) illustrates the use of eval.

Listing 6.14: evalfunc.py

```
x1 = eval(input('Entry x1? '))
print('x1 =', x1, ' type:', type(x1))

x2 = eval(input('Entry x2? '))
print('x2 =', x2, ' type:', type(x2))

x3 = eval(input('Entry x3? '))
print('x3 =', x3, ' type:', type(x3))

x4 = eval(input('Entry x4? '))
print('x4 =', x4, ' type:', type(x4))

x5 = eval(input('Entry x5? '))
print('x5 =', x5, ' type:', type(x5))
```

A sample run of Listing 6.14 (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 "evalfunc.py", line 13, in <module>
    x5 = eval(input('Entry x5? '))
  File "<string>", line 1, in <module>
NameError: name 'x6' is not defined
```

Notice that when the user enters the text consisting of a single digit 4, the eval function interprets it as integer 4 and assigns an integer to the variable x1. When the user enters the text 4.0, the assigned 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 unquoted text as a reference to the name x1 established by the first assignment statement. The program bound the name x1 to the integer value 4 when executing the first line of the program. This statement thus binds x4 to the same integer; that is, 4. 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 and 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

could enter multiple entries separated by commas, and the `eval` function would evaluate the text typed by the user as Python tuple. As Listing 6.15 (addintegers4.py) shows, this makes tuple assignment (see Section 2.2) from the `input` function possible.

Listing 6.15: 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 6.16 (enterarith.py) is a simple, one line Python program that behaves like Python's interactive shell, except that it accepts only one expression from the user.

Listing 6.16: enterarith.py

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

A sample run of Listing 6.16 (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 6.16 (enterarith.py); as the program runs the `eval` function compiles the user-supplied text into executable code and executes it to produce 14.

The `exec` function, also from the `__builtins__` module, is similar to the `eval` function. The `exec` function accepts a string parameter that consists of a Python source statement. The `exec` function interprets the statement and executes it. Listing 6.17 (myinterpreter.py) plays the role of a rudimentary Python interpreter.

Listing 6.17: myinterpreter.py

```
while True:
    exec(input(">>>"))
```

```
>>>from sys import exit
>>>x = 45
>>>print(x)
45
>>>x += 1000
>>>print(x)
1045
>>>exit(0)
```

While the `eval` and `exec` functions may seem to open up a number of interesting possibilities, their use actually is very limited. The `eval` and `exec` functions demand much caution on the part of the programmer.

In fact, the examples above that use the `eval` and `exec` functions are not advisable in practice. This is because they enable the user to make the program do things the programmer never intended. Python contains functions that call on the operating system to perform tasks. This functionality includes the possibility of erasing files or formatting entire disk drives. If the user knows the required Python code to accomplish such devious tasks, he or she could hijack the program and cause havoc. As simple, harmless example, consider the following example run of Listing 6.14 (evalfunc.py):

```
Entry x1? 100
x1 = 100  type: <class 'int'>
Entry x2? exec("import sys; sys.exit(0)")
```

During the execution of Listing 6.14 (evalfunc.py) the user entered the text

exec(”import sys; sys.exit(0)”)

The `eval` function then interprets and evaluates the call to `exec`. The evaluation of the `sys.exit` function will, of course, terminate the program. The program is written to interact with the user a while longer, the user terminated it early.

In programs that use `eval` or `exec`, the programmer must preprocess the user input to defend against unwanted and unwelcome program behavior.

6.9 Turtle Graphics

One of the simplest ways to draw pictures is the way we do it by hand with pen and paper. We place the pen on the paper and move the pen, leaving behind a mark on the paper. The length and shape of the mark depends on the movement of the pen. We then can lift the pen from the paper and place it elsewhere on the paper to continue our graphical composition. We may have pens of various colors at our disposal.

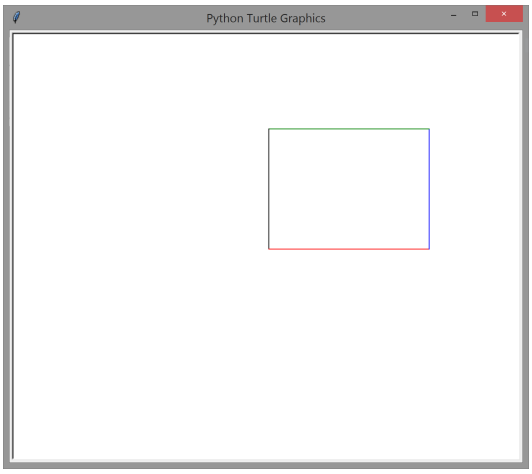
Turtle graphics on a computer display mimics these actions of placing, moving, and turning a pen on a sheet of paper. It is called Turtle graphics because originally the pen was represented as a turtle moving within the display window. Seymour Papert originated the concept of Turtle graphics in his Logo programming language in the late 1960s (see `http://en.wikipedia.org/wiki/Turtle_graphics` for more information about Turtle graphics). Python includes a Turtle graphics library that is relatively easy to use.

In the simplest Turtle graphics program we need only issue commands to a pen (turtle) object. We must import the `turtle` module to have access to Turtle graphics. Listing 6.18 (boxturtle.py) draws a rectangular box.

Listing 6.18: `boxturtle.py`

```
# Draws a rectangular box in the window

import turtle

turtle.pencolor('red')   # Set pen color to red
turtle.forward(200)      # Move pen forward 200 units (create bottom of rectangle)
turtle.left(90)          # Turn pen by 90 degrees
turtle.pencolor('blue')  # Change pen color to blue
turtle.forward(150)      # Move pen forward 150 units (create right wall)
turtle.left(90)          # Turn pen by 90 degrees
```

Figure 6.6 A very simple drawing made with Turtle graphics

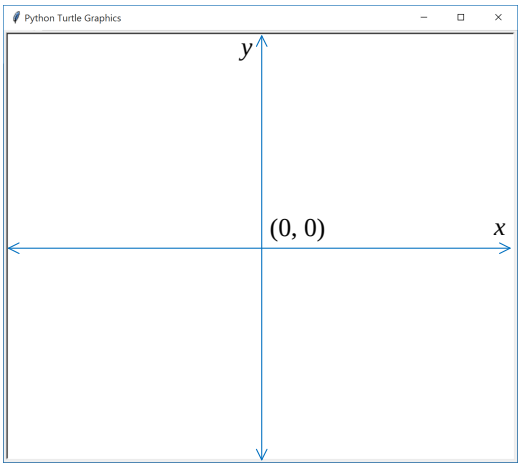



```
turtle.pencolor('green') # Change pen color to green
turtle.forward(200)      # Move pen forward 200 units (create top)
turtle.left(90)          # Turn pen by 90 degrees
turtle.pencolor('black') # Change pen color to black
turtle.forward(150)      # Move pen forward 150 units (create left wall)
turtle.hideturtle()      # Make pen invisible
turtle.exitonclick()     # Wait for user input
```

Figure 6.6 shows the result of running Listing 6.18 (boxturtle.py). By default, the pen starts at the center of the window facing to the right. Figure 6.7 shows the default Turtle graphics coordinate system. Listing 6.18 (boxturtle.py) reveals a few of the functions provided by the `turtle` module:

- `setheading`: sets the pen to face in a particular direction
- `pencolor`: sets pen's current drawing color.
- `forward`: moves the pen forward a given distance at its current heading
- `left`: turns the pen to the left by an angle specified in degrees.
- `right`: turns the pen to the right by an angle specified in degrees.
- `setposition`: moves the pen to a given (x, y) coordinate within the graphics window. Draws a line from the previous position unless the pen is up.
- `title`: sets the text to appear in the window's title bar
- `penup`: disables drawing until the pen is put back down (allows turtle movements without drawing)
- `pendown`: enables the pen to draw when moved
- `hideturtle`: makes the pen object (turtle) itself invisible; this does not affect its ability to draw.
- `tracer`: turns off tracing (drawing animation) when its argument is zero—this speeds up rendering.

Figure 6.7 The default coordinate system for a Turtle graphics picture. The x and y axes do not appear in an actual image. As in the Cartesian coordinate system from algebra, the x values increase from left to right, and the y values increase from bottom to top.



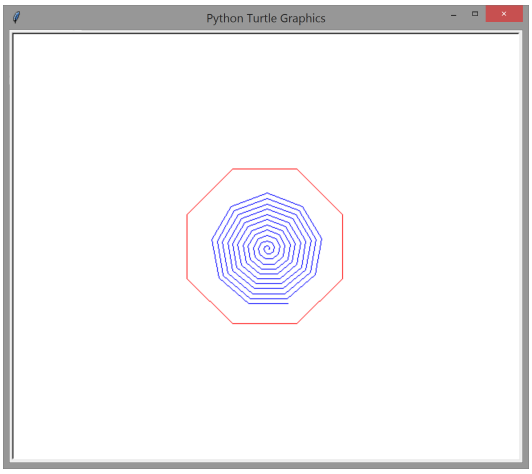
- `update`: renders all pending drawing actions (necessary when tracing is disabled).
- `done`: ends the drawing activity and waits on the user to close the window.
- `exitonclick`: directs the program to terminate when the user clicks the mouse over the window.
- `mainloop`: used in place of `done` to enable the graphics framework to handle events such as user mouse clicks and keystrokes.

Listing 6.19 (`octogon.py`) draws in the display window a blue spiral within a red octogon.

Listing 6.19: `octogon.py`

```
# Draws in the window a spiral surrounded with an octagon

import turtle

# Draw a red octogon centered at (-45, 100)
turtle.pencolor('red')       # Set pen color
turtle.penup()               # Lift pen to move it
turtle.setposition(-45, 100) # Move the pen to coordinates (-45, 100)
turtle.pendown()             # Place pen to begin drawing
for i in range(8):           # Draw the eight sides
    turtle.forward(80)       # Each side is 80 units long
    turtle.right(45)         # Each vertex is 45 degrees


# Draw a blue spiral centered at (0, 0)
distance = 0.2
angle = 40
turtle.pencolor('blue')      # Set pen color
```

Figure 6.8 More Turtle graphics fun: a spiral within an octogon




```
turtle.penup()               # Left pen to move it
turtle.setposition(0, 0)     # Position the pen at coordinates (0, 0)
turtle.pendown()             # Set pen down to begin drawing
for i in range(100):
    turtle.forward(distance)
    turtle.left(angle)
    distance += 0.5

turtle.hideturtle()          # Make pen invisible
turtle.exitonclick()         # Quit program when user clicks mouse button
```

Listing 6.19 (octogon.py) uses the `turtle.penup`, `turtle.setposition`, and `turtle.pendown` functions to move the pen to a particular location without leaving a mark within the display window. The center of the display window is at coordinates (0, 0). Figure 6.8 shows the result of running Listing 6.19 (octogon.py).

After each drawing command such as `forward` or `left` the graphics environment updates the image, thus showing the effect of the command. If you are annoyed that the drawing is too slow, you can speed up the rendering process in several ways. You can add the following statement before moving the pen:

```
turtle.speed(0)    #  Fastest turtle actions
```

The speed function accepts an integer in the range 0...10. The value 1 represents the slowest speed, and the turtle's speed increases as the arguments approach 10. Counterintuitively, 0 represents the fastest turtle speed. The speed function will accept a string argument in place of an integer value; the permissible strings correspond to the following numeric values:

- `"fastest"` is equivalent to 0
- `"fast"` is equivalent to 10
- `"normal"` is equivalent to 6
- `"slow"` is equivalent to 3

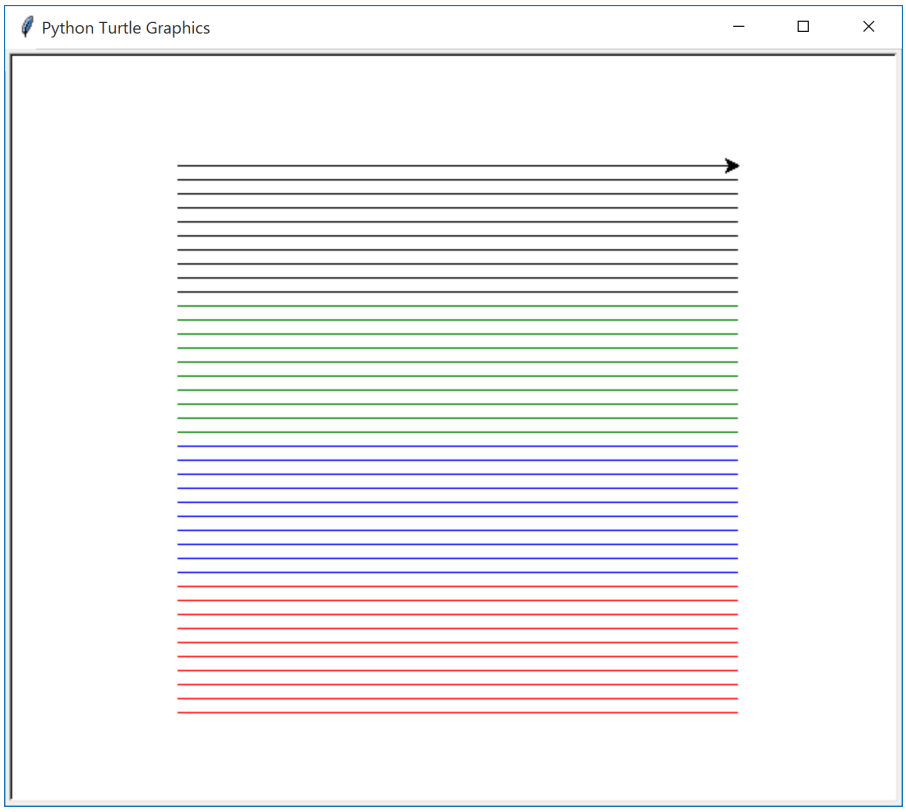
- `"slowest"` is equivalent to 1

The `delay` function provides another way to affect the time it takes to render an image. Rather than controlling the overall speed of the turtle's individual movements and/or turns, the `delay` function specifies the time delay in milliseconds between drawing incremental updates of the image to the screen. Listing 6.20 (speedvsdelay.py) demonstrates the subtle difference in the way `speed` and `delay` functions effect the time to render an image.

Listing 6.20: speedvsdelay.py

```
import turtle

y = -200  # Initial y value

# Default speed and default delay
turtle.color("red")
for x in range(10):
    turtle.penup()
    turtle.setposition(-200, y)
    turtle.pendown()
    turtle.forward(400)
    y += 10

# Slowest speed, but no delay
turtle.speed("slowest")
turtle.delay(0)
turtle.update()
turtle.color("blue")
for x in range(10):
    turtle.penup()
    turtle.setposition(-200, y)
    turtle.pendown()
    turtle.forward(400)
    y += 10


# Fastest speed with a 500 millisecond delay
turtle.speed("fastest")
turtle.delay(500)
turtle.update()
turtle.color("green")
for x in range(10):
    turtle.penup()
    turtle.setposition(-200, y)
    turtle.pendown()
    turtle.forward(400)
    y += 10

# Fastest speed with no delay
turtle.speed("fastest")
turtle.delay(0)
turtle.update()
turtle.color("black")
for x in range(10):
    turtle.penup()
```


Figure 6.9 Listing 6.20 (speedvsdelay.py) demonstrates the different effects of the `turtle.speed` versus `turtle.delay` functions. The program renders the bottom 10 lines using the default speed and delay settings. In the next 10 lines the program draws with the slowest speed but without a delay. The program next draws 10 lines at the fastest speed with a 100 millisecond delay between screen updates. The top 10 lines represent the fastest speed with no delay. Note that the animation is still smooth at a slower speed, but the `delay` function makes the animation choppy at regular time intervals.




```
    turtle.setposition(-200, y)
    turtle.pendown()
    turtle.forward(400)
    y += 10

turtle.done()
```

Figure 6.9 shows the eventual output of Listing 6.20 (speedvsdelay.py). The `speed` function does not sacrifice the smoothness of the animation, while the `delay` function can make the animation choppy at regular time intervals.

You can use the `tracer` function to achieve the fastest rendering possible. Listing 6.21 (noanimation.py) produces the same drawing as Listing 6.20 (speedvsdelay.py), but the picture appears almost instantly.

Listing 6.21: noanimation.py

```
import turtle

y = -200  # Initial y value

# Turn off animation
turtle.tracer(0)
turtle.color("red")
for x in range(10):
    turtle.penup()
    turtle.setposition(-200, y)
    turtle.pendown()
    turtle.forward(400)
    y += 10

turtle.color("blue")
for x in range(10):
    turtle.penup()
    turtle.setposition(-200, y)
    turtle.pendown()
    turtle.forward(400)
    y += 10

turtle.color("green")
for x in range(10):
    turtle.penup()
    turtle.setposition(-200, y)
    turtle.pendown()
    turtle.forward(400)
    y += 10

turtle.color("black")
for x in range(10):
    turtle.penup()
    turtle.setposition(-200, y)
    turtle.pendown()
    turtle.forward(400)
    y += 10

turtle.update()     # Ensure all of image is drawn
turtle.done()
```

When animation is disabled with the `tracer` function, you should call `update` at the point the complete image should appear. When the tracer is active it explicitly draws the penstrokes to the screen as the turtle moves. With the tracer turned off, the programmer must ensure all the image becomes visible by calling the `update` function. Turning the tracer off is the ultimate way to speed up Turtle graphics rendering in Python.

6.10 Other Techniques for Importing Functions and Modules

Section 6.2 introduced the two most common ways programmers use to import functions into Python code. For the sake of completeness, we briefly examine three other importing techniques that you may encounter

in published Python code.

Recall that we can use the `from ... import ...` notation to import some of the functions that a module has to offer. This allows callers to use the base names of the functions without prepending the module name. This technique becomes unwieldy when a programmer wishes to use a large number of functions from a particular module.

The following statement:

```
from math import *
```

makes all the code in the `math` module available to the program. The `*` symbol is the wildcard symbol that represents "everything." Some programmers use an `import` statement like this one for programs that need to use many different functions from a module.

As we will see below, however, best Python programming practice discourages this "import everything" approach.

This "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, within the program, short function names are easier to type than the longer, 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

```
from math import log10
```

may cover all that this program needs from the `math` module. If the programmer instead uses

```
from math import *
```

this statement imports everything from the `math` module, including a function named `degrees` which converts an angle measurement in radians to degrees (from trigonometry, $360^\circ = 2\pi$ 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. Even though the `import` statement brings in the `degrees` function, the programmer is free to redefine `degrees` to be a floating-point variable (recall redefining the `print` function in Section 2.3). If the program does redefine `degrees`, the `math` module's `degrees` function is unavailable if the programmer later discovers its need. 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 used simultaneously for both purposes.

Most significant Python programs must import multiple modules to obtain all the functionality they need. It is possible for two different modules to provide one or more functions with the same name. This is another example of a name collision.

The names of variables and functions available to a program live in that program's *namespace*. The `dir` function prints a program's namespace, and we can experiment with it in Python's interactive interpreter:

```
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__']
>>> x = 2
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'x']
```



```
>>> from math import sqrt
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'sqrt', 'x']
>>> from math import *
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf',
'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp',
'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc', 'x']
```

Observe how assigning the variable `x` adds the name `x` to the interpreter's namespace. Importing just `math.sqrt` adds `sqrt` to the namespace. Finally, importing everything from the `math` module adds many more names. If we attempt to use any of these names in a different way, they will lose their original purpose; for example, the following continues the above interactive sequence:

```
>>> help(exp)
Help on built-in function exp in module math:

exp(...)
    exp(x)

    Return e raised to the power of x.
```

Now let us redefine `exp`:

```
>>> exp = None
>>> help(exp)
Help on NoneType object:

class NoneType(object)
 |  Methods defined here:
 |
 |  __bool__(self, /)
 |      self != 0
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __repr__(self, /)
 |      Return repr(self).
```

If we reassign the `exp` name to refer to `None`, we no longer can use it to compute e^x.

We say that the "import everything" statement *pollutes* the program's namespace. This kind of import adds many names (variables, functions, and other objects) to the collection of names managed by the program. This can cause name collisions as in the example above with the name `degrees`, and it makes it more difficult to work with larger programs. When adding new functionality to such a program we must be careful not to tread on any names that already exist in the program's namespace.

Python best programming practices discourages the use of the "import everything" statement:

```
from math import *
```

since this provides more opportunities for name collisions and renders code less maintainable.

Most Python programmers agree that the best approach imports the whole module, as in

```
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 plain `degrees`. Also, if, for example, modules `mod1` and `mod2` both contain a function named `process`, the module import statement forces programmers to write `mod1.process` and `mod2.process`, thus avoiding the name clash.

We have seen the compromise: import only the functions needed, as in

```
from math import sqrt, log10
```

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. Many programmers find this approach acceptable for smaller programs where the probablility of name clashes is low as the program evolves.

Python provides a way to import a module under a different name. The following statement imports the standard `math` module as `m`:

```
import math as m
```

In this case, the caller would invoke the modules functions in the following way:

```
y = m.sqrt(x)
print(m.log10(100))
```

Note the `m.` prefix attached to the calls of the `sqrt` and `log10` functions. Programmers sometimes use this module renaming import to simplify typing when a module name is long. Listing 6.22 (octogon2.py) is a rewrite of Listing 6.19 (octogon.py) that introduces a new name for the `turtle` module: `t`. We say that `turtle` and `t` are *aliases* for the same module. This in effect shortens the qualified names for each of the function calls. The fact that Listing 6.22 (octogon2.py) runs faster than Listing 6.19 (octogon.py) has nothing to do with the module name aliasing or shorter qualified function names; Listing 6.22 (octogon2.py) runs much faster because it calls the `turtle.delay` function to speed up the drawing.

Listing 6.22: octogon2.py

```
# Draws in the window a spiral surrounded with an octagon

import turtle as t      # Use a shorter name for turtle module

t.delay(0)              # Draw as quickly as possible
t.speed(0)              # Turtle's action as fast as possible
t.hideturtle()          # Do not show the pen
# Draw a red octogon centered at (-45, 100)
t.pencolor('red')       # Set pen color
t.penup()               # Lift pen to move it
t.setposition(-45, 100) # Move the pen to coordinates (-45, 100)
t.pendown()             # Place pen to begin drawing
for i in range(8):      # Draw the eight sides
    t.forward(80)       # Each side is 80 units long
```



```
    t.right(45)         # Each vertex is 45 degrees


# Draw a blue spiral centered at (0, 0)
distance = 0.2
angle = 40
t.pencolor('blue')      # Set pen color
t.penup()               # Left pen to move it
t.setposition(0, 0)     # Position the pen at coordinates (0, 0)
t.pendown()             # Set pen down to begin drawing
for i in range(100):
    t.forward(distance)
    t.left(angle)
    distance += 0.5

t.exitonclick()         # Quit program when user clicks the mouse button
```

When a program imports a module this way the module's original name is not automatically available. This means the name `turtle` is not available to the code in Listing 6.22 (octogon2.py). Code must access the `turtle` module's functiom via `t`.

The practice of using an alias merely to shorten module name is questionable. It essentially hides a recognized standard name and so renders the code less readable. Fortunately, the ability to alias module names is convenient for other purposes besides shortening module names. Suppose a third-party software vendor develops an alternative to Python's standard `math` module. It includes all the functions provided by the `math` module, with exactly the same names. The company markets it as a drop-in replacement for the `math` module. The vendor names this module `fastmath` because the algorithms it uses to implement the mathematical functions are more efficient than those used in the `math` module. As an example, when invoked with the same arguments the `fastmath.sqrt` and `math.sqrt` functions compute identical results, but `fastmath.sqrt` returns its result quicker than `math.sqrt`.

Suppose, too, that we have a large application that performs many mathematical calculations using functions from the `math` module. We would like to try the third-party `fastmath` module to see if it indeed speeds up our application. Among other possible module imports, our application includes the following statement:

```
import math
```

This means our application is sprinkled with statements like

```
y = math.sqrt(max - alpha)
while value < max:
    value += math.log10(inc)
```

calling `math` functions in hundreds of places throughout the code. Note the qualified function names. If we change the import statement to

```
import fastmath
```

we will have to edit the function invocations in hundreds of places within our code. Instead, we can change the import statement to read

```
import fastmath as math
```

Adding just two words to the original import statement enables us to leave everything else in our code alone. Every mention of `math` will refer actually to the `fastmath` module.

In addition to importing an entire module under a new name, Python allows programmers to alias individual functions. Consider the following code:

```
from math import sqrt as sq

print(sq(16))
```

The names of standard functions are well known to experienced Python developers, so such renaming renders a program immediately less less readable. We should not consider renaming standard functions unless we have a very good reason. Some have found this technique useful for resolving name clashes between two modules that define one or more functions with the same name. Returning to our example from above, suppose we wish to compare directly the performance of `math.sqrt` to `fastmath.sqrt`. The process of measuring the relative performance of software is known as *benchmarking*. We need to have both `math.sqrt` and `fastmath.sqrt` available in the same program. The following code performs the benchmark and avoids qualified function names:

```
from time import perf_counter
from math import sqrt as std_sqrt
from fastmath import sqrt as fast_sqrt

start_time = perf_counter()
for n in range(100000):
    std_sqrt(n)
print('Standard:', perf_counter() - start_time)

start_time = perf_counter()
for n in range(100000):
    fast_sqrt(n)
print('Fast:', perf_counter() - start_time)
```

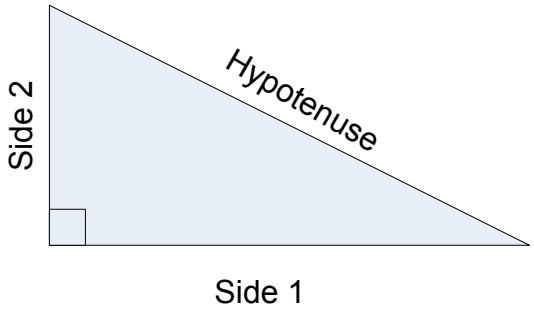
The renamed functions may be confusing. The better approach imports the modules themselves rather than the individual functions:

```
import math, fastmath, time

start_time = time.perf_counter()
for n in range(100000):
    math.sqrt(n)
print(time.perf_counter() - start_time)

start_time = time.perf_counter()
for n in range(100000):
    fastmath.sqrt(n)
print(time.perf_counter() - start_time)
```

This version arguably is better. Without looking elsewhere in the program's source code, the programmer immediately can see exactly which function from which module the program is calling. The qualified names make it perfectly clear which function is which.

While using module and function aliases can be convenient at times, you should use this feature only if necessary. Standard names are well known and recognizable by experienced Python programmers. Giving

Figure 6.10 Right triangle



new names to standard functions and modules unnecessarily can make programs less readable and confuse programmers attenpting to modify or extend an existing application.

When in doubt, follow best practice and use the module import statement. This makes each function call more self-documenting by unambiguously indicating its module of origin.

6.11 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)`

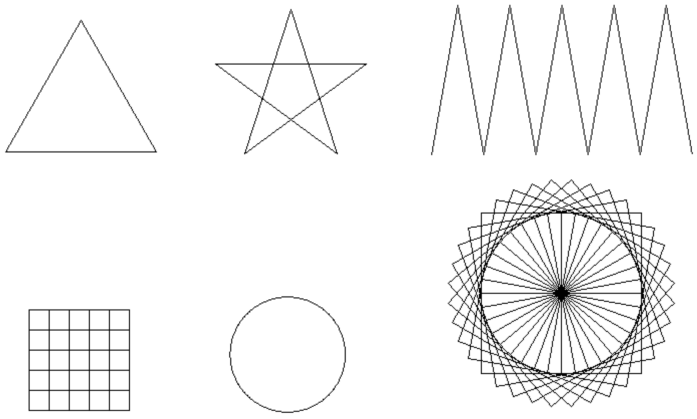
4. From geometry: Write a computer program that, given the lengths of the two sides of a right triangle adjacent to the right angle, computes the length of the hypotenuse of the triangle. (See Figure 6.10.) If you are unsure how to solve the problem mathematically, do a web search for the *Pythagorean theorem*.

5. Write a guessing game program in which the computer chooses at random an integer in the range 1...100. 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.

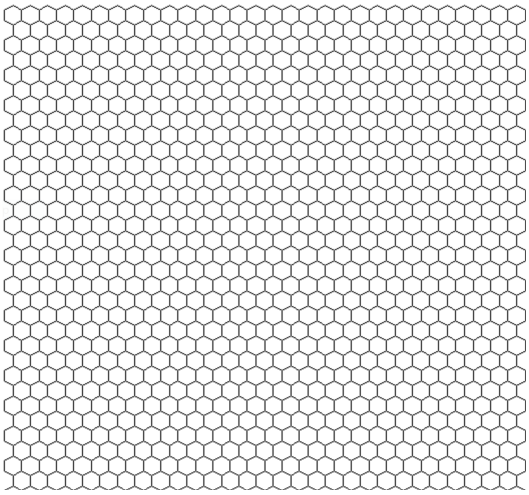
6. Extend Problem 5 by keeping track of the number of guesses the user needed to get the correct answer. Report the number of guesses at the end of the game.

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.

8. For each of the drawings below write a program that draws the shape using functions from Python's Turtle graphics module.



9. Write a program that uses functions from Python's Turtle graphics module to draw a grid of hexagons as shown in the picture below.



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. These pieces assemble to form the problem's 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`, `sqrt`, `randrange`, 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 becomes more difficult to manage. 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**. Complicated monolithic code attempts to do everything that needs to done within the program. The indivisible nature of the code divides the programmer's attention amongst all the tasks the block must perform. In order to write a statement within a block of monolithic code the programmer must be completely familiar with the details of *all* the code in that block. For instance, we must use care when introducing a new variable to ensure that variable's name is not already being used within the block.

- **It is difficult to debug**. If the sequence of code does not work correctly, it may be difficult to find the source of the error. The effects of an erroneous statement that appears earlier in a block of monolithic code may not become apparent until a possibly correct statement later uses the erroneous statement's incorrect result. Programmers naturally focus their attention first to where they observe the program's misbehavior. Unfortunately, when the problem actually lies elsewhere, it takes more time to locate and repair the problem.

- **It is difficult to extend**. Much of the time software developments spend is modifying and extending existing code. As in the case of originally writing the monolithic block of code, a programmer must

understand all the details in the entire sequence of code before attempting to modify it. 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, we 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. This process of is known as *functional decomposition*. 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, often we need a function exhibiting custom behavior unavailable in any standard function. Fortunately, we *can* create our own functions. Once created, we can use (call) these functions in numerous places within a program. If the function's purpose is general enough and we write the function properly, we can 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 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 four parts:

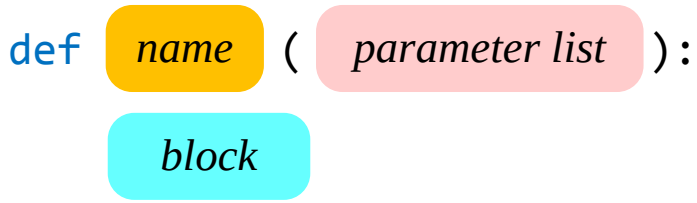
- `def`—The `def` keyword introduces a function definition.
- **Name**—The name is an identifier (see Section 2.3). As with variable names, the name chosen for a function should accurately portray its intended purpose or describe its functionality. (Python provides for specialized anonymous functions called `lambda` expressions, but we defer their introduction until Section 8.7.)
- **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. A colon follows the parameter list.
- **Body**—every function definition has a block of indented statements that constitute the function's body. The body contains the code to execute when callers invoke the function. The code within the body is responsible for producing the result, if any, to return to the caller.

Figure 7.1 shows the general form of a function definition.

A function usually accepts some values from its caller and returns a result back to its caller. Listing 7.1 (doublenumber.py) provides a very simple function definition to illustrate the process.

Listing 7.1: `doublenumber.py`

Figure 7.1 General form of a function definition




```
def double(n):
    return 2 * n  # Return twice the given number

# Call the function with the value 3 and print its result
x = double(3)
print(x)
```

Listing 7.1 (doublenumber.py) is admittedly not very useful, but it illustates several important points:

- The two lines

  ```
  def double(n):
      return 2 * n  # Return twice the given number
  ```

  constitute the definition of the `double` function.

- The `def` keyword marks the beginning of the function's definition.

- The function definition establishes the function's name is `double`.

- The function definition specifies that the function accepts one value from the user. This value will go by the name `n` within the function's body. This variable `n` is the function's only parameter.

- The statement(s) that constitute the working part of the function definition (in this case just one statement) are indented relative to the line containing `def`.

- The `return` keyword indicates the value the function is to communicate by to its caller. In this case, the function simply returns the product of its parameter `n` and 2.

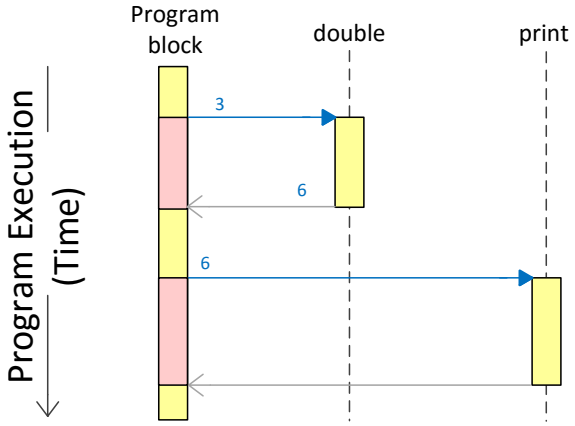
- The code that follows the `double` function's definition:

  ```
  # Call the function with the value 3 and print its result
  x = double(3)
  print(x)
  ```

  calls (or invokes) the function sending it the value 3. The assignment statement binds the function's computed return value to the variable `x`. The program can use `x` as it sees fit; in this case the program simply prints `x`'s value.

The program runs as follows:

Figure 7.2 Calling relationships among functions during the execution of Listing 7.1 (doublenumber.py). Time flows from top to bottom. A vertical bar represents the time in which a block of code is active. Observe that functions are active only during their call. The shaded area within in block represents the time that block is idle, waiting for a function call to complete. Right arrows (→) represent function calls. Function calls show parameters, where applicable. Left arrows (←) represent function returns. Function returns show return values, if applicable.



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. The program thus executes the assignment statement that calls the `double` function with the argument 3. Before the assignment can happen, the program's execution transfers to the body of the `double` function. The code within `double` executes, which simply returns the product of 2 and the parameter passed in (in this case 3).

2. When `double` is finished, control is passed back to the point in the code where it was called, in this case effectively evaluating the expression `double(3)` to be 6. The assignment operator then assigns the value 6 to the variable `x`.

3. The program finally prints the value of `x`, calculated to be 6.

Figure 7.2 contains a diagram illustrating the execution of Listing 7.1 (doublenumber.py) as control passes amongst the various functions. The interaction amongst functions can be quite elaborate, even for relatively simple programs.

It is important to note that a program executes code within a function only when the function is called. If a program contains a function definition, but no code executing within that program calls that function, that function's code will not execute.

As another simple example, consider Listing 7.2 (countto10.py).

Listing 7.2: `countto10.py`

```
#  Counts to ten
for i in range(1, 11):
    print(i, end=' ')
print()
```

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

```
#  Count to ten and print each number
def count_to_10():
    for i in range(1, 11):
        print(i, end=' ')
    print()


print("Going to count to ten . . .")
count_to_10()
print("Going to count to ten again. . .")
count_to_10()
```

Listing 7.3 (countto10func.py) prints

```
Going to count to ten . . .
1 2 3 4 5 6 7 8 9 10
Going to count to ten again. . .
1 2 3 4 5 6 7 8 9 10
```

The empty parentheses in `count_to_10`'s definition indicates that the function does not accept any parameters from its caller. Also, the absence of a `return` statement indicates that this function communicates no information back to its caller. Such functions that get no information in and provide no results can be useful for the effects they achieve (in this case just printing the numbers 1 to 10.

Our `doublenumber` and `count_to_10` functions are a bit underwhelming. The `doublenumber` function could be eliminated, and each call to `doublenumber` could be replaced with a simple variation of the code in its body. The same could be said for the `count_to_10` function, although it is convenient to have the simple one-line statement that hides the complexity of the loop. These examples serve simply to familiarize us with the mechanics of function definitions and invocations. Functions really shine when our problems become more complex.

Our experience using a simple function like `print` shows 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. Our `double` function from above will compute a different result if we pass it a different parameter.

If a function is written to accept information from the caller, the caller 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 `count_to_10` 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 caller needs.

Listing 7.4: countton.py

```
#  Count to n and print each number
def count_to_n(n):
     for i in range(1, n + 1):
         print(i, end=' ')
     print()


print("Going to count to ten . . .")
count_to_n(10)
print("Going to count to five . . .")
count_to_n(5)
```

Listing 7.4 (countton.py) displays

```
Going to count to ten . . .
1 2 3 4 5 6 7 8 9 10
Going to count to five . . .
1 2 3 4 5
```

When the caller 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.

The actual parameter may be a literal value (such as 10 in the expression `count_to_n(10)`), or it may be a variable, as Listing 7.5 (countwithvariable.py) illustrates.

Listing 7.5: countwithvariable.py

```
def count_to_n(n):
     for i in range(1, n + 1):
         print(i, end=' ')
     print()


for i in range(1, 10):
    count_to_n(i)
```

```
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
1 2 3 4 5 6 7 8 9
```

The actual parameter a caller sends to the `count_to_n` function may in fact be any expression that evaluates to an integer.

A caller must pass exactly one integer parameter to `count_to_n` during a call. An attempt to pass no parameters or more than one integer 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 `count_to_n` function passes its parameter on to the `range` expression, and `range` requires all of its arguments to be integers.

```
count_to_n(3.2)   # Run-time error, actual parameter not an integer
```

To recap, in the first line of the function definition:

```
def double(n):
```

we refer to `n` as the formal parameter. A formal parameter is used like a variable within the function's body, and it is local to the function. A formal parameter is the parameter from the perspective of the function definition. During an invocation of `double`, such as `double(2)`, the caller passes actual parameter 2. The actual parameter is the parameter from the caller's point of view. A function invocation, therefore, binds the actual parameters sent by the caller to their corresponding formal parameters.

A caller can pass multiple pieces of information into a function via multiple parameters. Consider Listing 7.6 (regularpolygon.py) that uses Turtle graphics (see Section 6.9) to draw regular polygons with varying numbers of sides. In a regular polygon all sides have the same length, and all angles between the sides are the same (see `https://en.wikipedia.org/wiki/Regular_polygon`).

Listing 7.6: regularpolygon.py

```
import turtle
import random


# Draws a regular polygon with the given number of sides.
# The length of each side is length.
# The pen begins at point(x, y).
# The color of the polygon is color.
def polygon(sides, length, x, y, color):
    turtle.penup()
    turtle.setposition(x, y)
    turtle.pendown()
    turtle.color(color)
    turtle.begin_fill()
    for i in range(sides):
        turtle.forward(length)
```



```
        turtle.left(360//sides)
    turtle.end_fill()


# Disable rendering to speed up drawing
turtle.hideturtle()
turtle.tracer(0)

# Draw 20 random polygons with 3-11 sides, each side ranging
# in length from 10-50, located at random position (x, y).
# Select a color at random from red, green, blue, black, or yellow.
for i in range(20):
    polygon(random.randrange(3, 11), random.randrange(10, 51),
            random.randrange(-250, 251), random.randrange(-250, 251),
            random.choice(("red", "green", "blue", "black", "yellow")))

turtle.update()  # Render image
turtle.exitonclick()  # Wait for user's mouse click
```

The polygon function accepts four parameters:

- `sides` specifies the number of sides,
- `length` specifies the length of each side,
- `x` specifies the x component of the polygon's (x, y) location.
- `y` specifies the y component of the polygon's (x, y) location.
- `color` specifies the polygon's filled color.

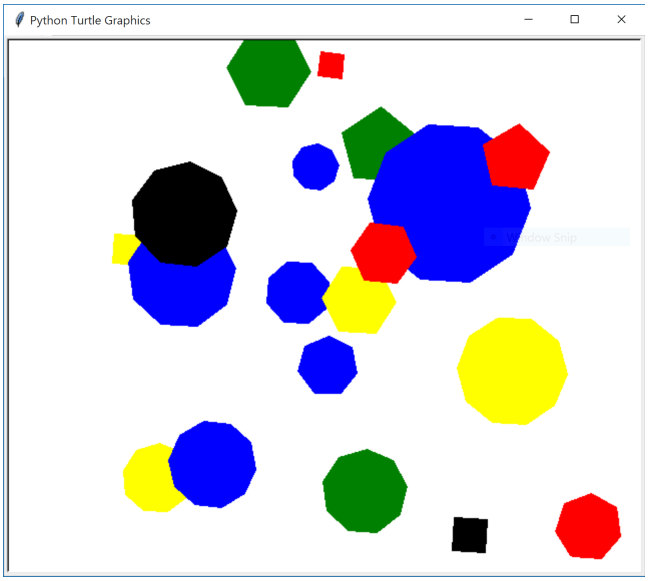
The polygon function does not have a `return` statement, so it does not communicate a result back to its caller. This program introduces the `begin_fill` and `end_fill` functions from the `turtle` module. When placed around code that draws a closed figure, these functions fill the shape with the current drawing color. Figure 7.3 shows a screenshot of a sample run of Listing 7.6 (regularpolygon.py).

While a caller can pass multiple pieces of information into a function via multiple parameters, a function ordinarily passes back to the caller one piece of information via a `return` statement. If necessary, a function may return multiple pieces of information packed up in a tuple or other data structure. Listing 7.7 (midpoint.py) uses a custom function to compute the midpoint between two mathematical points.

Listing 7.7: `midpoint.py`

```
def midpoint(pt1, pt2):
    x1, y1 = pt1   # Extract x and y components from the first point
    x2, y2 = pt2   # Extract x and y components from the second point
    return (x1 + x2)/2, (y1 + y2)/2


#  Get two points from the user
point1 = float(input("Enter first point's x: ")), \
         float(input("Enter first point's y: "))
point2 = float(input("Enter second point's x: ")), \
         float(input("Enter second point's y: "))
#  Compute the midpoint
```

Figure 7.3 A sample run of Listing 7.6 (regularpolygon.py). The number of sides, length of each side, location, and color of each polygon varies based on the arguments passed by the caller.




```
mid = midpoint(point1, point2)
#  Report result to user
print('Midpoint of', point1, 'and', point2, 'is', mid)
```

Listing 7.7 (midpoint.py) accepts two parameters, each of which is a tuple containing two values: the x and y components of a point. Given two mathematical points (x_1, y_1) and (x_2, y_2), the function uses the following formula to compute (x_m, y_m), the midpoint of (x_1, y_1) and (x_2, y_2):

$$(x_m, y_m) = \left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}\right)$$

A sample run of Listing 7.7 (midpoint.py) looks like the following:

```
Enter first point's x: 0
Enter first point's y: 0
Enter second point's x: 1
Enter second point's y: 1
Midpoint of (0.0, 0.0) and (1.0, 1.0) is (0.5, 0.5)
```

The `midpoint` function returns only one result, but that result is a tuple containing two pieces of data. The `mid` variable in Listing 7.7 (midpoint.py) refers to a single tuple object. We will examine tuples in more detail in Chapter 11, but for now it is useful to note that we also can extract the components of the returned tuple into individual numeric variables as follows:

```
mx, my = midpoint(point1, point2)  # Unpack the returned tuple
```

Here, the `midpoint` function still is returning just one value (a tuple), but the assignment statement "unpacks" the individual values stored in the tuple and assigns them to the variables `mx` and `my`. The process of extracting the pieces of a tuple object into separate variables is formally called *tuple unpacking*.

Figure 7.4 Cutting plywood

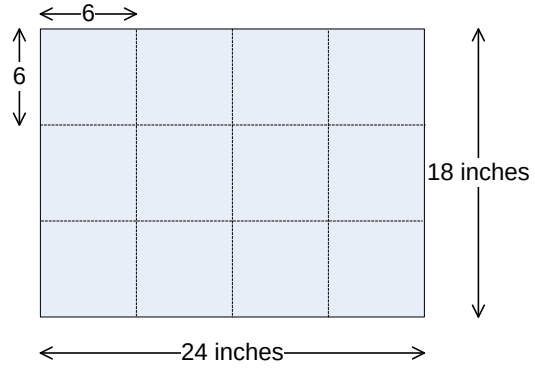
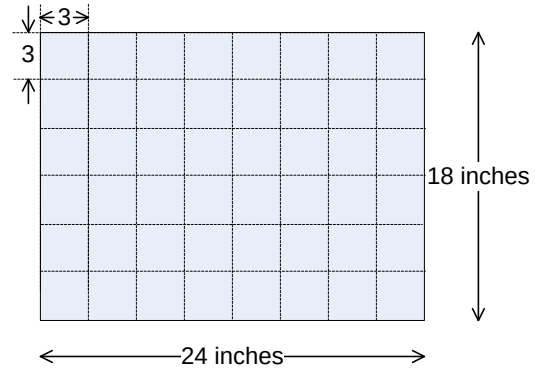


Figure 7.5 Squares too small

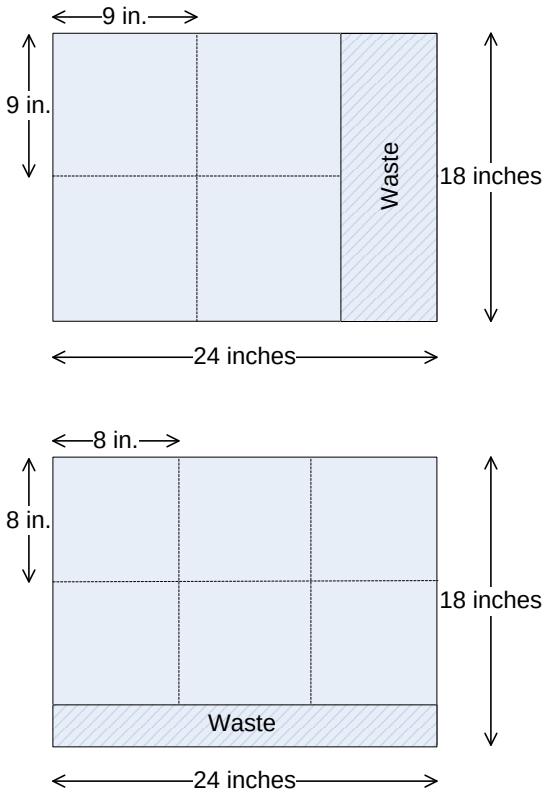


Recall the greatest common divisor (also called greatest common factor) function from elementary mathematics. To determine the GCD of 24 and 18 we list all of their common factors and select the largest one:

24: 1, 2, 3, 4, $\boxed{6}$, 8, 12, 24
18: 1, 2, 3, $\boxed{6}$, 9, 18

The greatest common divisor function is useful for reducing fractions to lowest terms; for example, consider the fraction $\frac{18}{24}$. The greatest common divisor of 18 and 24 is 6, and we can compute the reduced fraction by dividing the numerator and the denominator by 6: $\frac{18 \div 6}{24 \div 6} = \frac{3}{4}$. The GCD function has applications in other areas besides reducing fractions to lowest terms. Consider the problem of dividing a piece of plywood 24 inches long by 18 inches wide into square pieces of maximum size in integer dimensions, without wasting any material. Since the GCF(24, 18) = 6, we can cut the plywood into twelve 6 inch $\times$ 6 inch square pieces as shown in Figure 7.4. If we cut the plywood into squares of any other size without wasting the any of the material, the squares would have to be smaller than 6 inches $\times$ 6 inches; for example, we could make forty-eight 3 inch $\times$ 3 inch squares as shown in pieces as shown in Figure 7.5. If we cut squares larger than 6 inches $\times$ 6 inches, not all the plywood can be used to make the squares. Figure 7.6. shows

Figure 7.6 Squares too large



how some larger squares would fare. In addition to basic arithmetic and geometry, the GCD function plays a vital role in cryptography, enabling secure communication across an insecure network.

We can write a program that computes the GCD of two integers supplied by the user. Listing 7.8 (gcdprog.py) is one such program.

Listing 7.8: gcdprog.py

```
# Compute the greastest common factor of two integers
# provided by the user

# Prompt user for input
num1 = int(input('Please enter an integer: '))
num2 = int(input('Please enter another integer: '))

# Determine the smaller of num1 and num2
min = num1 if num1 < num2 else num2

# 1 definitely is a common factor to all ints
largest_factor = 1
```



```
for i in range(1, min + 1):
    if num1 % i == 0 and num2 % i == 0:
        largest_factor = i   # Found larger factor
# Print the GCD
print(largest_factor)
```

Listing 7.8 (gcdprog.py) implements a straight-forward but naive algorithm that seeks potential factors by considering every integer less than the smaller of the two values provided by the user. This algorithm is not very efficient, especially for larger numbers. Its logic is easy to follow, with no deep mathematical insight required. Soon we will see a better algorithm for computing GCD.

If we need to compute the GCD from several different places within our program, we should package the code in a function rather than copying it to multiple places. The following code fragment defines a Python function that that computes the greatest common divisor of two integers. It determines the largest factor (divisor) common to its parameters:

```
def gcd(num1, num2):
    # Determine the smaller of num1 and num2
    min = num1 if num1 < num2 else num2
    # 1 definitely is a common factor to all ints
    largest_factor = 1
    for i in range(1, min + 1):
        if num1 % i == 0 and num2 % i == 0:
            largest_factor = i   # Found larger factor
    return largest_factor
```

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`, `largest_factor`, and `i`. Local variables have meaning only within their scope. The scope of a local variable is the point within the function's block after its first assignment until the end of that block. This means that when you write a function you can name a local variable without concern 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 also is local to its function, you can reuse the names of formal parameters in different functions without a problem.

Listing 7.9 (localplay.py) illustrates how function definitions provide protection for local variables.

Listing 7.9: `localplay.py`

```
x = 2
print("1. x =", x)  # Print variable x's current value

def fun1():
    x = 10
    print("2. x =", x)  # Print this variable x's current value

print("3. x =", x)  # Print variable x's current value

def fun2():
    x = 20
    print("4. x =", x)  # Print this variable x's current value

print("5. x =", x)  # Print variable x's current value
```

```
fun1()  # Invoke function fun1
fun2()  # Invoke function fun2

print("6. x =", x)  # Print variable x's current value
```

Notice that Listing 7.9 (localplay.py) numbers each print statement in the order of its appearance in the program's source code. When executing the program the interpreter will executes the source code line by line, top to bottom. It executes each statement in turn, but function definitions are special—a function definition packages code into an executable unit to be executed later. The code within a function definition executes only when invoked by a caller. Listing 7.9 (localplay.py), therefore, will not execute the print statements in the order listed in the source code. Listing 7.9 (localplay.py) prints

```
1. x = 2
3. x = 2
5. x = 2
2. x = 10
4. x = 20
6. x = 2
```

The printing statements 1, 2, 5, and 6 all refer to the variable `x` defined outside of functions `fun1` and `fun2`. When `fun1` and `fun2` assign to a variable named `x`, this `x` is local to its respective function. The assignments within `fun1` and `fun2` do not affect the variable involved in printing statements 1, 2, 5, or 6. Note that the printing statements within `fun1` and `fun2` do not execute until the program actually calls the functions. That is why printing statements 2 and 4 appear out of numerical order in the program's execution. In the end, the last printing statement, number 6, prints the value of the original `x` variable that the program assigned in its first line. The code within `fun1` and `fun2`, as they currently are written, cannot disturb the value of this external variable. (In Section 8.1 we shall see how a function can gain access to a variable defined outside the function's definition.)

In the code we have considered in earlier chapters, the name of a variable uniquely identified it and distinguished that variable from another variable. It may seem strange that now we can use the same name in two different functions within the same program to refer to two distinct variables. The block of statements that makes up a function definition constitutes a context for local variables. A simple analogy may help. In the United States, many cities have a street named *Main Street*; for example, there is a thoroughfare named Main Street in San Francisco, California. Dallas, Texas also has a street named Main Street. Each city and town provides its own context for the use of the term *Main Street*. A person in San Francisco asking "How do I get to Main Street?" will receive the directions to San Francisco's Main Street, while someone in Dallas asking the same question will receive Dallas-specific instructions. In a similar manner, assigning a variable within a function block localizes its identity to that function. We can think of a program's execution as a person traveling around the U.S. When in San Francisco, all references to *Main Street* mean San Francisco's Main Street, but when the traveler arrives in Dallas, the term *Main Street* means Dallas' Main Street. A program's thread of execution cannot execute more than one statement at a time, which means it uses its current context to interpret any names it encounters within a statement. Similarly, at the risk of overextending the analogy, a person cannot be physically located in more than one city at a time. Furthermore, Main Street may be a bustling, multi-lane boulevard in one large city, but a street by the same name in a remote, rural township may be a narrow dirt road! Similarly, two like-named variables may mean two completely different things. A variable named `x` is one function may represent an integer, while a different function may use a string variable named `x`.

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 a function invocation is complete and control returns to the caller, the function's variables and parameters go out of scope, and the run-time environment ensures that the memory used by the local variables is freed up for other purposes within the running program. This process of local variable allocation and deallocation happens each time a caller invokes the function.

Once we have written a complete function definition we can use the function within our program. We invoke a programmer-defined function in exactly the same way as a standard library function like `sqrt` (6.4) or `randrange` (6.6). 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:

```
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, we can pass variables, expressions, and literals as actual parameters. The function then computes and returns its result. Here, this result is assigned to the variable `factor`.

How does the function call and parameter mechanism work? It's actually quite simple. The executing program binds the actual parameters, in order, to each of the formal parameters in the function definition and then passes control 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

```
factor = gcd(val, 24)
```

assigns an integer value to `factor`. The expression on the right is a function call, so the executing program invokes the function 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 executes. When the `return` statement in the body of `gcd` executes, program control returns back to where the function was called. The argument of the return statement becomes the value assigned to `factor`.

Note that we can call `gcd` from many different places within the same program, and, since we can pass different parameter values 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 statement prints *12*, 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*. Function composition is nothing new to us, consider the following statement which prints the square root of 16:

  ```
  print(sqrt(16))
  ```

  The actual parameter passed to the `print` function is the result of the `sqrt` function call.

Listing 7.10 (gcdfunc.py) is a complete Python program that uses the gcd function.

Listing 7.10: gcdfunc.py

```
# Compute the greatest common factor of two integers
# provided by the user


def gcd(n1, n2):
    # Determine the smaller of n1 and n2
    min = n1 if n1 < n2 else n2
    # 1 definitely is a common factor to all ints
    largest_factor = 1
    for i in range(1, min + 1):
        if n1 % i == 0 and n2 % i == 0:
            largest_factor = i   # Found larger factor
    return largest_factor


# Exercise the gcd function

# Prompt user for input
num1 = int(input('Please enter an integer: '))
num2 = int(input('Please enter another integer: '))

# Determine the smaller of num1 and num2
min = num1 if num1 < num2 else num2

# Print the GCD
print(gcd(num1, num2))
```

The following shows a sample run of Listing 7.10 (gcdfunc.py):

```
Please enter an integer: 24
Please enter another integer: 18
6
```

Note that the program first defines the gcd function and then uses (calls) it in the program's last line. As usual, we can tell where the function definition ends and the rest of the program begins since the function's block of statements is indented relative to the rest of the program.

Within a program, a function's definition must appear before its use. Consider Listing 7.11 (gcdfuncbad.py), which moves the gcd's definition to the end of the source code.

Listing 7.11: gcdfuncbad.py

```
# NOTE: This program will not run to completion because it
# calls the gcd function before defining it!

# [Attempt to] Compute the greatest common factor of two integers
# provided by the user

# Exercise the gcd function

# Prompt user for input
num1 = int(input('Please enter an integer: '))
num2 = int(input('Please enter another integer: '))

# Print the GCD
print(gcd(num1, num2))


def gcd(n1, n2):
    # Determine the smaller of n1 and n2
    min = n1 if n1 < n2 else n2
    # 1 definitely is a common factor to all ints
    largest_factor = 1
    for i in range(1, min + 1):
        if n1 % i == 0 and n2 % i == 0:
            largest_factor = i   # Found larger factor
    return largest_factor
```

The execution of Listing 7.11 (gcdfuncbad.py) produces an error:

```
Please enter an integer: 2
Please enter another integer: 3
Traceback (most recent call last):
  File "gcdfuncbad.py", line 14, in <module>
    print(gcd(num1, num2))
NameError: name 'gcd' is not defined
```

The Python interpreter executes the code, line by line, until it encounters the call to gcd. If it has not yet seen gcd's definition, it will terminate the program with an error

Functions help us organize our code. It is not uncommon for programmers to write a main controlling function that calls other functions to accomplish the work of the application. Listing 7.12 (gcdwithmain.py) illustrates this organization.

Listing 7.12: gcdwithmain.py

```
#  Computes the greatest common divisor of m and n
def gcd(m, n):
    # Determine the smaller of m and n
    min = m if m < n else n
    # 1 is definitely a common factor to all ints
    largest_factor = 1
```

```
    for i in range(1, min + 1):
        if m % i == 0 and n % i == 0:
            largest_factor = i   # Found larger factor
    return largest_factor


#  Get an integer from the user
def get_int():
    return int(input("Please enter an integer: "))


#  Main code to execute
def main():
    n1 = get_int()
    n2 = get_int()
    print("gcd(", n1, ",",  n2, ") = ", gcd(n1, n2), sep="")


# Run the program
main()
```

The single free statement at the end:

```
main()
```

calls the `main` function which in turn directly calls several other functions (`get_int`, `print`, and `gcd`). The `get_int` function itself directly calls `int` and `input`. In the course of its execution the `gcd` function calls `range`. Figure 7.7 contains a diagram that shows the calling relationships among the function executions during a run of Listing 7.12 (gcdwithmain.py).

The name `main` for the controlling function is arbitrary but traditional; several other popular programming languages (C, C++, Java, C#, Objective-C) require such a function and require it to be named `main`.

7.2 Parameter Passing

When a caller invokes a function that expects a parameter, the caller 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

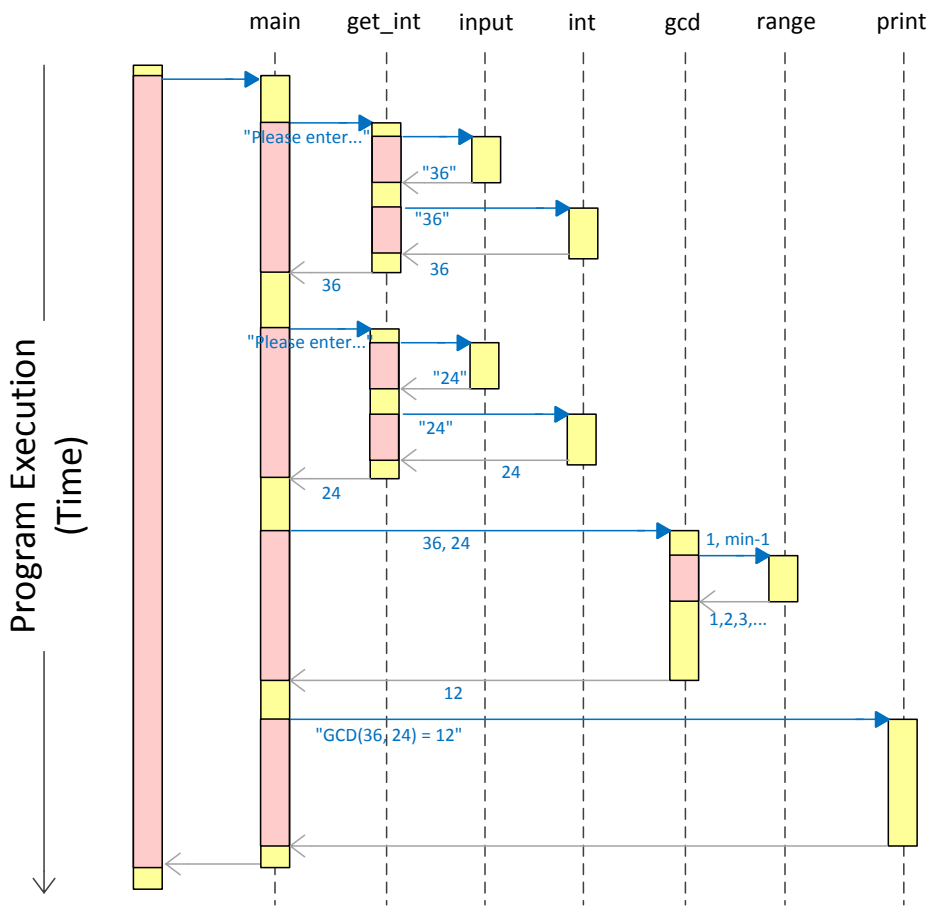
```
x = 4
```

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 caller's actual parameter references an immutable object, the function's activity cannot affect the value of the actual parameter. Listing 7.13 (parampassing.py) illustrates the consequences of passing an immutable type to an function.

Figure 7.7 Calling relationships among functions during the execution of Listing 7.12 (gcdwithmain.py)



Listing 7.13: parampassing.py

```
def increment(x):
   print("Beginning execution of increment, x =", x)
   x += 1   # Increment x
   print("Ending execution of increment, x =", x)


def main():
    x = 5
    print("Before increment, x =", x)
    increment(x)
    print("After increment, x =", x)


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; the variables live in two completely different contexts. Listing 7.13 (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. Inside the `increment` function the statement

```
x += 1
```

is short for

```
x = x + 1
```

The expression `x + 1` refers to 5 + 1 = 6, a different object from 5. The assignment statement re-binds `increment`'s `x` variable to 6. At this point `increment`'s `x` variable and `main`'s `x` variable refer to two different integer objects.

7.3 Documenting Functions

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**. A parameter's name is obvious in the definition, but the expected type and the 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 caller? 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.

Recall Python's multi-line strings, introduced in Section 2.9. When such a string appears as the first line in the block of a function definition, the string is known as a *documentation string*, or *docstring* for short. We can document our gcd function as shown in Listing 7.14 (docgcd.py).

Listing 7.14: docgcd.py

```
def gcd(n1, n2):
    """ Computes the greatest common divisor of integers n1 and n2.  """
    # Determine the smaller of n1 and n2
    min = n1 if n1 < n2 else n2
    # 1 definitely is a common factor to all ints
    largest_factor = 1
    for i in range(1, min + 1):
        if n1 % i == 0 and n2 % i == 0:
            largest_factor = i   # Found larger factor
    return largest_factor
```

Note that Listing 7.14 (docgcd.py) is not executable Python program; it provides only the definition of the gcd function. We can start a Python interactive shell and import the gcd code:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from docgcd import gcd
>>> gcd(18, 24)
6
>>>
```

With the docgcd code loaded into the interactive shell as so, we can type:

```
>>> help(gcd)
Help on function gcd in module docgcd:

gcd(n1, n2)
    Computes the greatest common divisor of integers n1 and n2.

>>>
```

The normal # comments serve as *internal documentation* for developers of the gcd function, while the function's docstring serves as *external documentation* for callers of the function.

Other information often is 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 than appear in a docstring.

The official Python style guide recommends using """ for docstrings rather than '''—see https://www.python.org/dev/peps/pep-0008/. In fact, since the docstring for our gcd function above is only one line of text, the normal ' and " quotation marks are adequate to specify its docstring. We will follow the convention of using """ to delimit our docstrings, even when expressing a single-line documentation.

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

- callers know what the function can do for them (via the docstring),
- callers know how to use the function (via the docstring),
- subsequent programmers that must maintain the function can contact the original author (via the comment) if questions arise about its use or implementation,
- subsequent programmers that must maintain the function can check the Wikipedia reference (via the comment) if questions arise about its implementation, and
- subsequent programmers can evaluate the quality of the algorithm based upon the quality of its source of inspiration (Wikipedia, via the comment).

7.4 Function Examples

This section contains a number of examples of code organization with functions.

7.4.1 Better Organized Prime Generator

Listing 7.15 (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 7.15: primefunc.py

```
from math import sqrt

def 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.
    """
    root = round(sqrt(n)) + 1
    # Try all potential factors from 2 to the square root of n
    for trial_factor in range(2, root):
        if n % trial_factor == 0:  # Is it a factor?
            return False           # Found a factor
    return True                    # No factors found


def 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.
    """
    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 7.15 (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 caller's intentions with the result it returns.

- A thorough comment describing the nature of the function precedes each function. The comment 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

```
if is_prime(value) == True:
```

can be expressed more compactly as

```
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)` all by itself suffices.

Observe that the `return` statement in the `is_prime` function immediately exits the function. In the `for` loop the `return` statement acts like a `break` statement because it immediately exits the loop on the way to immediately exiting the function.

Some purists contend that just as it is better for a loop to have exactly one exit point, it is better for a function to have a single `return` statement. The following code rewrites the `is_prime` function so that uses only one return statement:

```
def is_prime(n):
    result = True # Provisionally, n is prime
    root = round(sqrt(n)) + 1
    # 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
```

This version adds a local variable (`result`) and complicates the logic a little, so we can make a strong case for the original, two-`return` version. The two `return` statements in the original `is_prime` function are close enough textually in the code that the logic is easy to follow.

7.4.2 Command Interpreter

Some functions are useful even if they accept no information from the caller and return no result. Listing 7.16 (calculator.py) uses such a function.

Listing 7.16: `calculator.py`

```
def help_screen():
    """
    Displays information about how the program works.
    Accepts no parameters.
    Returns nothing.
    """
    print("Add:  Adds two numbers")
    print("Subtract:  Subtracts two numbers")
    print("Print:  Displays the result of the latest operation")
    print("Help:  Displays this help screen")
    print("Quit:  Exits the program")
```



```
def menu():
     """
     Displays a menu
     Accepts no parameters
     Returns the string entered by the user.
     """
    return input("=== A)dd S)ubtract P)rint H)elp Q)uit ===")


def main():
    """ Runs a command loop that allows users to perform simple arithmetic.  """
    result = 0.0
    done = False   # Initially not done
    while not done:
        choice = menu()     # Get user's choice

        if choice == "A" or choice == "a":   # Addition
            arg1 = float(input("Enter arg 1: "))
            arg2 = float(input("Enter arg 2: "))
            result = arg1 + arg2
            print(result)
        elif choice == "S" or choice == "s": # Subtraction
            arg1 = float(input("Enter arg 1: "))
            arg2 = float(input("Enter arg 2: "))
            result = arg1 - arg2
            print(result)
        elif choice == "P" or choice == "p": # Print
            print(result)
        elif choice == "H" or choice == "h": # Help
            help_screen()
        elif choice == "Q" or choice == "q": # Quit
            done = True


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.

7.4.3 Restricted Input

Listing 5.39 (betterinputonly.py) forces the user to enter a value within a specified range. We now can easily adapt that concept to a function. Listing 7.17 (betterinputfunc.py) uses a function named get_int_in_range that does not return until the user supplies a proper value.

Listing 7.17: betterinputfunc.py

```
def get_int_in_range(first, last):
    """
    Forces the user to enter an integer within a specified range.
    first is either a minimum or maximum acceptable value.
    last is the corresponding other end of the range, either a maximum or minimum value.
    Returns an acceptable value from the user.
```

```
    """
    # If the larger number is provided first,
    # switch the parameters
    if  first > last:
        first, last = last, first
    # Insist on values in the range first...last
    in_value = int(input("Please enter values in the range " \
                         + str(first) + "..." + str(last) + ": "))
    while in_value < first or in_value > last:
        print(in_value, "is not in the range", first, "...", last)
        in_value = int(input("Please try again: "))
    # in_value at this point is guaranteed to be within range
    return in_value


def main():
    """ Tests the get_int_in_range function """
    print(get_int_in_range(10, 20))
    print(get_int_in_range(20, 10))
    print(get_int_in_range(5, 5))
    print(get_int_in_range(-100, 100))


main()   # Run the program
```

Listing 7.17 (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 7.17 (betterinputfunc.py)

- Parameters delimit the high and low values. 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)
```

7.4.4 Better Die Rolling Simulator

Listing 7.18 (betterdie.py) reorganizes Listing 6.11 (die.py) into functions.

Listing 7.18: betterdie.py

```
from random import randrange

def show_die(spots):
    """
    Draws a picture of a die with number of spots indicated.
    spots is the number of spots on the top face.
    """
    print("+-------+")
    if spots == 1:
        print("|       |")
        print("|   *   |")
        print("|       |")
    elif spots == 2:
        print("| *     |")
        print("|       |")
        print("|     * |")
    elif spots == 3:
        print("|     * |")
        print("|   *   |")
        print("| *     |")
    elif spots == 4:
        print("| *   * |")
        print("|       |")
        print("| *   * |")
    elif spots == 5:
        print("| *   * |")
        print("|   *   |")
        print("| *   * |")
    elif spots == 6:
        print("| * * * |")
        print("|       |")
        print("| * * * |")
    else:
        print(" ***  Error: illegal die value ***")
    print("+-------+")


def roll():
    """ Returns a pseudorandom number in the range 1...6, inclusive """
```

```
    return randrange(1, 7)


def main():
     """ Simulates the roll of a die three times """
     # Roll the die three times
     for i in range(0, 3):
         show_die(roll())


main()   # Run the program
```

In Listing 7.18 (betterdie.py), the `main` function is oblivious to the details of pseudorandom number generation. Also, `main` is not responsible for 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())
```

This is another example of *function composition* function composition. Function composition is not new to us; we have been using with the standard functions `input` and `int` in statements like: statements like

```
x = int(input())
```

7.4.5 Tree Drawing Function

Listing 7.19 (treefunc.py) reorganizes Listing 5.34 (startree.py) into functions.

Listing 7.19: `treefunc.py`

```
def tree(height):
    """
    Draws a tree of a given height.
    """
    row = 0              # First row, from the top, to draw
    while row < height:  # Draw one row for every unit of height
        # Print leading spaces
        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()
```



```
        # Change to the next row
        row += 1


def main():
    """ Allows users to draw trees of various heights """
    height = int(input("Enter height of tree: "))
    tree(height)


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` variables 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 the function in which it is being used.

7.4.6 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 we cannot represent 1/3 as a finite decimal in the base-10 number system, we cannot represent 1/10 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 7.20 (simplefloataddition.py), which adds two double-precision floating-point numbers and checks for a given value.

Listing 7.20: `simplefloataddition.py`

```
def main():
    x = 0.9
    x += 0.1
    if x == 1.0:
        print("OK")
    else:
        print("NOT OK")


main()
```

Listing 7.20 (simplefloataddition.py) reports

```
OK
```

All seems well judging from the behavior of Listing 7.20 (simplefloataddition.py). Next, consider Listing 7.21 (badfloatcheck.py) which attempts to control a loop with a double-precision floating-point number.

Listing 7.21: 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 7.21 (badfloatcheck.py) begins as expected, but it does not end as expected:

```
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
i = 0.9999999999999999
i = 1.0999999999999999
i = 1.2
i = 1.3
i = 1.4000000000000001
i = 1.5000000000000002
i = 1.6000000000000003
i = 1.7000000000000004
i = 1.8000000000000005
i = 1.9000000000000006
i = 2.0000000000000004
```

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 7.20 (simplefloataddition.py), but now there is a problem. Since 0.1 has no exact representation within the constraints of the binary double-precision floating-point number systems, 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, we see that 0.1 added to itself 10 times yields 0.9999999999999999 which is not exactly 1.

Listing 7.21 (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 6.4. Listing 7.22 (floatequalsfunction.py) provides such an `equals` function.

Listing 7.22: floatequalsfunction.py


```
from math import fabs

def 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.
    """
    return a == b or fabs(a - b) < tolerance


def main():
    """ Try out the equals function """
    i = 0.0
    while not equals(i, 1.0, 0.0001):
        print("i =", i)
        i += 0.1


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 4.7 (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.

7.5 Refactoring to Eliminate Code Duplication

Recall Listing 6.21 (noanimation.py) that uses Turtle graphics to draw a collection of lines of various colors. Consider the following code fragment, snipped from Listing 6.21 (noanimation.py):

```
turtle.color("blue")
for x in range(10):
    turtle.penup()
```

```
    turtle.setposition(-200, y)
    turtle.pendown()
    turtle.forward(400)
    y += 10

turtle.color("green")
for x in range(10):
    turtle.penup()
    turtle.setposition(-200, y)
    turtle.pendown()
    turtle.forward(400)
    y += 10
```

What is interesting to note about this code snippet is the similarities between the two sections. These two sections of code are performing exactly the same work, except for the variation of color. Listing 6.21 (noanimation.py) contains two additional sections not shown here that that exhibit the same similarities. This code represents duplication of coding effort, and this is known as *code duplication*.

Code duplication is undesirable for several reasons:

- The extra code to write requires more work on the part of the programmer. Copying and pasting with a good editor can save typing, but the code fragments are not identical in all ways, so a programmer must be sure to make all of the little changes required to produce the desired variance in behavior.
- Code duplication results in code that is more difficult to maintain. Most commercial software is not static; developers constantly work on newer versions. Newer versions typically provide bug fixes and add features. Upon repairing a logic error or enhancing a section of duplicated code, a programmer must track down all the other code sections within the system that duplicate the modified section. Failure to do so can introduce inconsistency into the program's behavior.

Functions are ideal for consolidating duplicate code. Through a process known as code *refactoring*, a programmer can place the common code within a function definition and tune the minor variance in behavior of the duplicated code via parameters. Listing 7.23 (noduplication.py) refactors the duplicated code in Listing 6.21 (noanimation.py) into one function that accepts a color and y value as a parameter.

Listing 7.23: noduplication.py

```
import turtle


def draw_lines(color, y):
    """ Draws 10 horizontal lines of a given color stacked
        on top of each other with the lowest line appearing
        at position y on the y axis. """
    turtle.color(color)
    for x in range(10):
        turtle.penup()
        turtle.setposition(-200, y)
        turtle.pendown()
        turtle.forward(400)
        y += 10
```

```
# Turn off animation
turtle.tracer(0)
draw_lines("red", -200)
draw_lines("blue", -100)
draw_lines("green",0)
draw_lines("black", 100)

turtle.update()     # Ensure all of image is drawn
turtle.done()
```

Now that the originally duplicated code appears in only one place in Listing 7.23 (noduplication.py), any correction or enhancement made to the code within the function automatically will be available to all the callers of that code. This removes the possibility of introducing an inconsistency into the application.

7.6 Custom Functions vs. Standard Functions

Recall the custom square root code we saw in Listing 5.33 (computesquareroot.py). We can package this code in a function. Just like the standard `math.sqrt` function, our custom square root function will accept a single numeric value and return a numeric result. Listing 7.24 (customsquareroot.py) contains the definition of our custom `square_root` function.

Listing 7.24: customsquareroot.py

```
#  File customsquareroot.py

def square_root(val):
    """ Compute an approximation of the square root of x """
    # Compute a provisional square root
    root = 1.0

    # How far off is our provisional root?
    diff = root*root - val

    # Loop until the provisional root
    # is close enough to the actual root
    while diff > 0.00000001 or diff < -0.00000001:
        root = (root + val/root) / 2    # Compute new provisional root
        # How bad is our current approximation?
        diff = root*root - val

    return root


#  Use the standard square root function to compare with our custom function
from math import sqrt

d = 1.0
while d <= 10.0:
    print('{0:6.1f}: {1:16.8f} {2:16.8f}' \
                .format(d, square_root(d), sqrt(d)))
    d += 0.5  # Next d
```

The `main` function in Listing 7.24 (customsquareroot.py) compares the behavior of our custom `square_root` function to the `sqrt` library function. Its output:

```
  1.0:       1.00000000       1.00000000
  1.5:       1.22474487       1.22474487
  2.0:       1.41421356       1.41421356
  2.5:       1.58113883       1.58113883
  3.0:       1.73205081       1.73205081
  3.5:       1.87082869       1.87082869
  4.0:       2.00000000       2.00000000
  4.5:       2.12132034       2.12132034
  5.0:       2.23606798       2.23606798
  5.5:       2.34520788       2.34520788
  6.0:       2.44948974       2.44948974
  6.5:       2.54950976       2.54950976
  7.0:       2.64575131       2.64575131
  7.5:       2.73861279       2.73861279
  8.0:       2.82842713       2.82842712
  8.5:       2.91547595       2.91547595
  9.0:       3.00000000       3.00000000
  9.5:       3.08220700       3.08220700
 10.0:       3.16227766       3.16227766
```

shows only a slight difference for $\sqrt{8}$. The fact that we found one difference in this small collection of test cases justifies using the standard `math.sqrt` function instead of our custom function. 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 time and 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 7.25 (squarerootcomparison.py) tests our custom square root function over a range of 10,000,000 floating point values.

Listing 7.25: squarerootcomparison.py

```
from math import fabs, sqrt

def equals(a, b, tolerance):
```

```
    """
    Consider two floating-point numbers equal when the difference between them is very small.
    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.
    """
    return a == b or fabs(a - b) < tolerance


def square_root(val):
    """
    Computes the approximate square root of val.
    val is a number
    """
    # Compute a provisional square root
    root = 1.0

    # How far off is our provisional root?
    diff = root*root - val

    # Loop until the provisional root
    # is close enough to the actual root
    while diff > 0.00000001 or diff < -0.00000001:
        root = (root + val/root) / 2       # Compute new provisional root
        # How bad is our current approximation?
        diff = root*root - val
    return root


def main():
    d = 0.0
    while d < 100000.0:
        if not equals(square_root(d), sqrt(d), 0.001):
            print('*** Difference detected for', d)
            print(' Expected', sqrt(d))
            print(' Computed', square_root(d))
        d += 0.0001  # Consider next value


main()  # Run the program
```

Listing 7.25 (squarerootcomparison.py) uses our `equals` function from Listing 4.7 (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 7.25 (squarerootcomparison.py) is

```
0.0 : Expected 0.0 but computed 6.103515625e-05
0.0006000000000000001 : Expected 0.024494897427831782 but computed 0.024495072155655266
```

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. One of values that causes the function

to fail may be very important to a particular application, so our function is not trustworthy.

7.7 Exercises

1. Is the following a legal Python program?

```
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?

```
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, y):
    return 2*x + y*y
```

```
def main():
    print(proc(5, 4))


main()
```

5. Is the following a legal Python program?

```
def proc(x, y):
    return 2*x + y*y


def main():
    print(proc(5))


main()
```

6. Is the following a legal Python program?

```
def proc(x):
    return 2*x


def main():
    print(proc(5, 4))


main()
```

7. Is the following a legal Python program?

```
def proc(x):
    print(2*x*x)


def main():
    proc(5)


main()
```

8. 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()
```

9. 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()
```

10. 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()
```

11. 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.

12. What happens if a caller passes too many parameters to a function?

13. What happens if a caller passes too few parameters to a function?

14. What are the rules for naming a function in Python?

15. Consider the following function definitions:

```
from random import randrange

def fun1(n):
    result = 0
    while n:
```

```
        result += n
        n -= 1
    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, 2)
```

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()))
```