
Fundamentals of **Python** Programming

Richard L. Halterman
Southern Adventist University

February 24, 2015

Fundamentals of Python Programming

Copyright © 2014 Richard L. Halterman. All rights reserved.

Formerly *Learning to Program with Python*

Copyright © 2011–2014 Richard L. Halterman. All rights reserved.

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	10
1.7	Summary	11
1.8	Exercises	12
2	Values and Variables	13
2.1	Integer and String Values	13
2.2	Variables and Assignment	17
2.3	Identifiers	23
2.4	Floating-point Numbers	25
2.5	Control Codes within Strings	28
2.6	User Input	29
2.7	The eval Function	31
2.8	Controlling the print Function	32
2.9	String Formatting	34
2.10	Summary	38
2.11	Exercises	39
3	Expressions and Arithmetic	41
3.1	Expressions	41
3.2	Mixed Type Expressions	47

3.3	Operator Precedence and Associativity	47
3.4	Comments	49
3.5	Errors	50
3.5.1	Syntax Errors	50
3.5.2	Run-time Exceptions	51
3.5.3	Logic Errors	53
3.6	Arithmetic Examples	54
3.7	More Arithmetic Operators	56
3.8	Algorithms	58
3.9	Summary	59
3.10	Exercises	61
4	Conditional Execution	65
4.1	Boolean Expressions	65
4.2	Boolean Expressions	66
4.3	The Simple if Statement	67
4.4	The if/else Statement	72
4.5	Compound Boolean Expressions	73
4.6	The pass Statement	77
4.7	Floating-point Equality	78
4.8	Nested Conditionals	79
4.9	Multi-way Decision Statements	89
4.10	Conditional Expressions	95
4.11	Errors in Conditional Statements	97
4.12	Summary	97
4.13	Exercises	98
5	Iteration	103
5.1	The while Statement	103
5.2	Definite Loops vs. Indefinite Loops	109
5.3	The for Statement	110
5.4	Nested Loops	115
5.5	Abnormal Loop Termination	120
5.5.1	The break statement	121
5.5.2	The continue Statement	124

5.6	while/else and for/else	126
5.7	Infinite Loops	128
5.8	Iteration Examples	131
5.8.1	Computing Square Root	131
5.8.2	Drawing a Tree	132
5.8.3	Printing Prime Numbers	135
5.8.4	Insisting on the Proper Input	138
5.9	Summary	139
5.10	Exercises	139
6	Using Functions	147
6.1	Introduction to Using Functions	148
6.2	Standard Mathematical Functions	152
6.3	time Functions	156
6.4	Random Numbers	158
6.5	Importing Issues	161
6.6	Summary	163
6.7	Exercises	164
7	Writing Functions	167
7.1	Function Basics	168
7.2	Main Function	178
7.3	Parameter Passing	179
7.4	Function Examples	181
7.4.1	Better Organized Prime Generator	181
7.4.2	Command Interpreter	183
7.4.3	Restricted Input	184
7.4.4	Better Die Rolling Simulator	185
7.4.5	Tree Drawing Function	187
7.4.6	Floating-point Equality	188
7.5	Custom Functions vs. Standard Functions	190
7.6	Summary	193
7.7	Exercises	193
8	More on Functions	199

8.1	Global Variables	199
8.2	Default Parameters	204
8.3	Introduction to Recursion	205
8.4	Making Functions Reusable	209
8.5	Documenting Functions and Modules	211
8.6	Functions as Data	213
8.7	Lambda Expressions	215
8.8	Generators	218
8.9	Local Function Definitions	224
8.10	Summary	230
8.11	Exercises	231
9	Objects	235
9.1	Using Objects	235
9.2	String Objects	236
9.3	File Objects	240
9.4	Fraction Objects	247
9.5	Turtle Graphics	248
9.6	Graphics with tkinter Objects	250
9.7	Other Standard Python Objects	253
9.8	Object Mutability and Aliasing	253
9.9	Garbage Collection	257
9.10	Summary	258
9.11	Exercises	259
10	Lists	261
10.1	Using Lists	263
10.2	List Traversal	267
10.3	Building Lists	268
10.4	List Membership	273
10.5	List Assignment and Equivalence	274
10.6	List Bounds	278
10.7	Slicing	280
10.8	List Element Removal	282
10.9	Lists and Functions	283

10.10List Methods	285
10.11Prime Generation with a List	286
10.12Command-line Arguments	288
10.13List Comprehensions	290
10.14Summary of List Creation Techniques	294
10.15Lists vs. Generators	294
10.16Summary	295
10.17Exercises	296
11 Sorting and Searching	301
11.1 Sorting	301
11.2 Flexible Sorting	304
11.3 Faster Sorting	306
11.4 Search	308
11.4.1 Linear Search	308
11.4.2 Binary Search	311
11.5 Recursion Revisited	320
11.6 List Permutations	325
11.7 Randomly Permuting a List	334
11.8 Reversing a List	338
11.9 Summary	339
11.10Exercises	339
12 Tuples, Dictionaries, and Sets	343
12.1 Tuples	343
12.2 Arbitrary Argument Lists	347
12.3 Dictionaries	349
12.4 Using Dictionaries	353
12.5 Counting with Dictionaries	355
12.6 Grouping with Dictionaries	358
12.7 Implementing Mathematical Graphs	359
12.8 Keyword Arguments	359
12.9 Sets	362
12.10Set Quantification with all and any	364
12.11Summary	367

12.12Exercises	367
13 Handling Exceptions	369
13.1 Motivation	369
13.2 Common Standard Exceptions	371
13.3 Handling Exceptions	372
13.4 Handling Multiple Exceptions	375
13.5 The Catch-all Handler	377
13.6 Catching Exception Objects	379
13.7 The try Statement's Optional else Block	381
13.8 finally block	382
13.9 Exception Handling Scope	386
13.10Raising Exceptions	392
13.11Using Exceptions	395
13.12Summary	396
13.13Exercises	397
14 Custom Types	399
14.1 Circle Objects	399
14.2 Restricting Access to Members	405
14.3 Bank Account Objects	407
14.4 Stopwatch Objects	410
14.5 Automated Testing	413
14.6 Class Inheritance	415
14.7 Custom Exceptions	417
14.8 Summary	417
14.9 Exercises	418
Index	420

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 must 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. Familiar operators (`=` and `+`) are used instead of some cryptic binary digit sequence that instructs the processor to perform the operation. Since this program is expressed in the Python language, not machine language, 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 referred 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 3.5 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://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 `Untitled-1.py`.

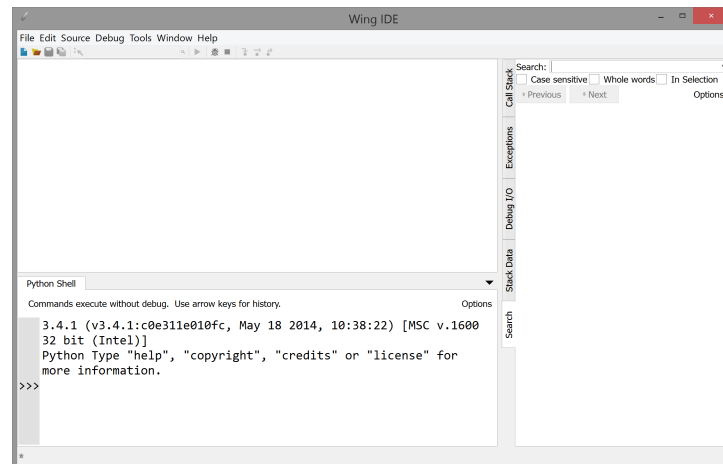
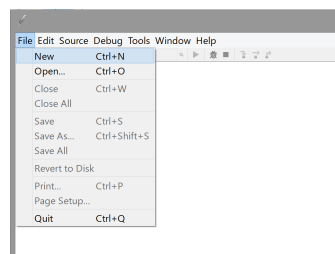
Figure 1.1: *WingIDE 101* running under Microsoft Windows

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

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

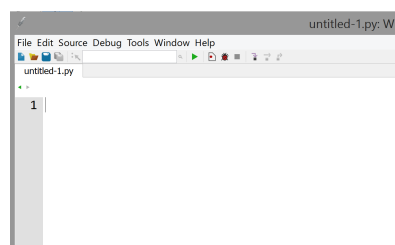


Figure 1.3: The new, untitled editor pane ready for code.

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

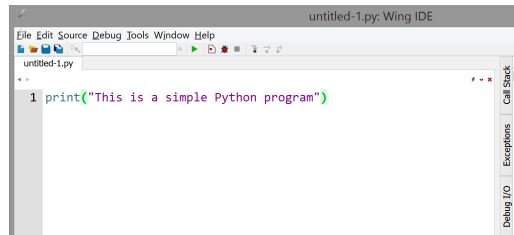


Figure 1.4: The code for the simple program after typed into the editor pane.

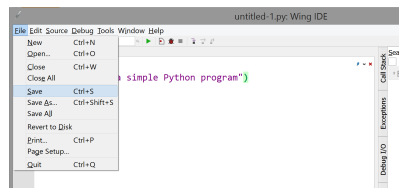


Figure 1.5: Save the Python program

name all Python programs with a .py extension.

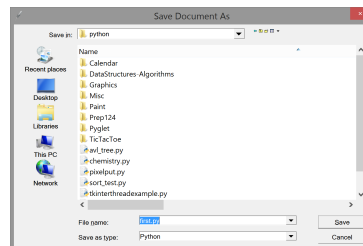


Figure 1.6: The file save dialog allows the user to name the Python file and locate the file in a particular folder.

To execute the program, select the little green triangle under the menu bar, shown in Figure 1.7.

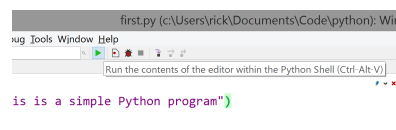
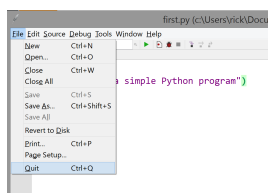


Figure 1.7: Running the program

In the pane labeled *Python Shell* we will see the program's output. Figure 1.8 shows the results of running the program.

Figure 1.8: *WingIDE 101* running under Microsoft Windows

When you are finished programming and wish to quit the IDE, follow the menu sequence *File*→*Quit* as shown in Figure 1.9.

Figure 1.9: *WingIDE 101* running under Microsoft Windows

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 1.10 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 1.10. This means we would show the output of Listing 1.1 (*simple.py*) as

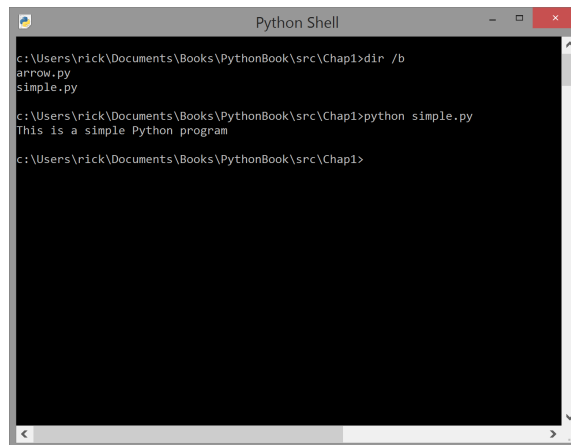


Figure 1.10: Running a Python program from the command line

```
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 will attempt to execute them. Figure 1.11 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



Figure 1.11: The interactive shell allows us to submit Python statements and expressions directly to the interpreter

(>>>). 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 1.12 demonstrates. This means not only can we execute Python programs apart from the *WingIDE 101* de-

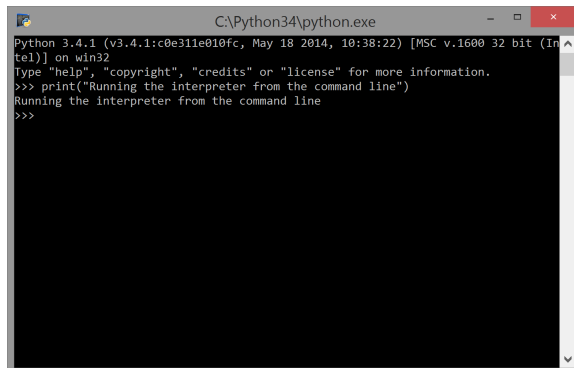


Figure 1.12: Running the Python interpreter from the command line

veloper environment, we also we can access Python's interactive interpreter separately from *WingIDE 101* if we so choose.

Figure 1.11 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 `>>>` 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 IDLE interactive shell, the program's output will be intermingled with the statements you type. In this case the best approach is to type the program into an editor, save the code you type to a file, and then execute the program. Most of the time we use an editor to enter and run our Python programs. The interactive interpreter is most useful for experimenting with small snippets of Python code.

In Listing 1.2 (arrow.py) each `print` statement “draws” a horizontal slice of the arrow. All the horizontal slices stacked on top of each other results in the picture of the arrow. The statements form a *block* of Python code. It is important that no *whitespace* (spaces or tabs) come before the beginning of each statement. In Python the indentation of statements is significant and 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 Summary

- Computers require both hardware and software to operate. Software consists of instructions that control the hardware.
- At the lowest level, the instructions for a computer program can be represented as a sequence of zeros and ones. The pattern of zeros and ones determine the instructions performed by the processor.
- Two different kinds of processors can have different machine languages.
- Application software can be written largely without regard to the underlying hardware. Tools automatically translate the higher-level, abstract language into the machine language required by the hardware.
- A compiler translates a source file into an executable file. The executable file may be run at any time with no further translation needed.
- An interpreter translates a source file into machine language each time a user executes the program.
- Compiled programs generally execute more quickly than interpreted programs. Interpreted languages generally allow for a more interactive development experience.
- Programmers develop software using tools such as editors, compilers, interpreters, debuggers, and profilers.

- Python is a higher-level programming language. It is considered to be a higher-level language than C, C++, Java, and C#.
- An IDE is an integrated development environment—one program that provides all the tools that developers need to write software.
- Messages can be printed in the output window by using Python's `print` function.
- A Python program consists of a code block. A block is made up of statements.

1.8 Exercises

1. What is a compiler?
2. What is an interpreter?
3. How is a compiler similar to an interpreter? How are they different?
4. How is compiled or interpreted code different from source code?
5. What tool does a programmer use to produce Python source code?
6. What is necessary to execute a Python program?
7. List several advantages developing software in a higher-level language has over developing software in machine language.
8. How can an IDE improve a programmer's productivity?
9. What the “official” Python IDE?
10. What is a *statement* in a Python program?

Chapter 2

Values and Variables

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

- numeric values
- 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 non-numeric values. In particular, Python programs can use integer values. The Python statement

```
print(4)
```

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

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

```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:24:06)
[MSC v.1600 32 bit (Intel)] on win32
Type "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 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 an 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 non-numeric 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 quotes when representing a string:

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

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

Consider the following interaction sequence:

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

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

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

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

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

As you might guess, there is little reason for a programmer to perform these kinds of transformations—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 in Section 2.2.

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

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

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

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

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

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

```
>>> '5' + 10
Traceback (most recent call last):
```

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

but the `int` and `str` functions can help:

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

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

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

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

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

2.2 Variables and Assignment

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

Listing 2.1: `variable.py`

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

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

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

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

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

The expression `'x = ' + x` would not be legal; as indicated in Section 2.1, the plus (`+`) operator may not be applied with mixed string and integer operands.

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

Listing 2.3: `multipleassignment2.py`

```
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 they do not run together.

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

Listing 2.4: `tupleassign.py`

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

The Listing 2.4 (`tupleassign.py`) program produces

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

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

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

`x, y, z` is one tuple, and `100, -45, 0` is another tuple. Tuple assignment works as follows: The first variable in the tuple on left side of the assignment operator is assigned the value of the first expression in the tuple on the left side (effectively `x = 100`). Similarly, the second variable in the tuple on left side of the assignment operator is assigned the value of the second expression in the tuple on the left side (in effect `y = -45`). `z` gets the value `0`. A tuple is a kind of Python type, like `int` or `float`, and we explore tuples in more detail in Chapter 12.

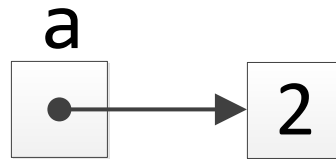


Figure 2.1: Binding a variable to an object

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.

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
```

Figure 2.2 illustrates the variable bindings after the Python interpreter executes the first statement.



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

Figure 2.3 shows how the situation changes after the second statement's execution.

Figure 2.4 shows how the situation changes after the third statement's execution.

Figure 2.5 illustrates the effects of statement four, and finally Figure 2.6 shows the variable bindings after all the statements have executed in the order listed.

Importantly, the statement

```
a = b
```

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

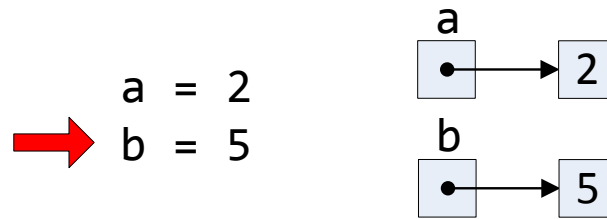


Figure 2.3: How variable bindings change as a program runs: step 2

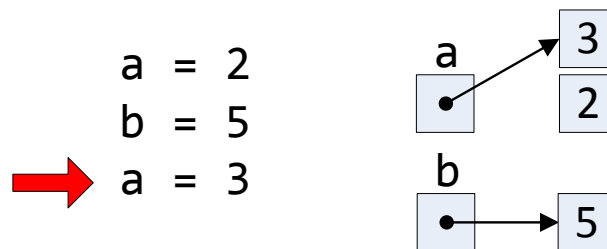


Figure 2.4: How variable bindings change as a program runs: step 3

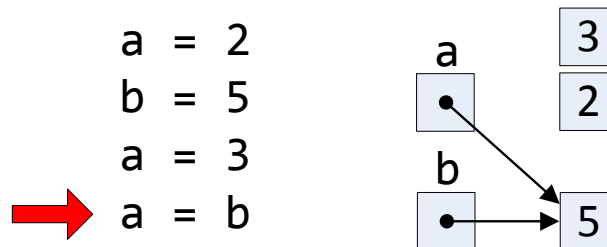


Figure 2.5: How variable bindings change as a program runs: step 4

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

Listing 2.5: changeabletype.py

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

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

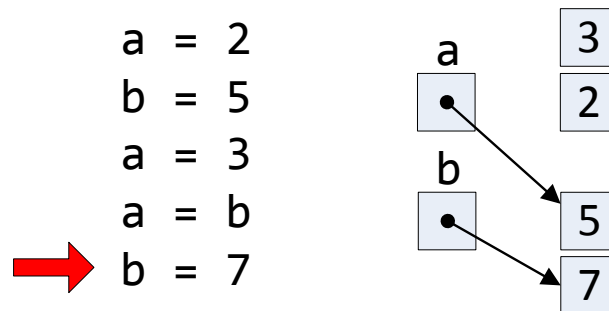


Figure 2.6: How variable bindings change as a program runs: step 5

```
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.

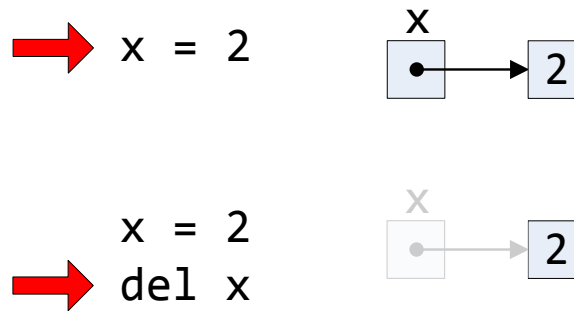


Figure 2.7: Definition and subsequent deletion of variable x

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 identifier must contain at least one character.
- The first character of an identifier 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`

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

Table 2.1: Python keywords

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

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

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

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

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

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

Listing 2.6: pi-print.py

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

The first line in Listing 2.6 (pi-print.py) 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.

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 2.7 (scientificnotation.py) prints some scientific constants using scientific notation.

Listing 2.7: scientificnotation.py

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

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, simply 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 specifies the desired number of decimal places. 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.

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 IDLE interactive shell, but they work properly in a command shell. Listing 2.8 (specialchars.py) prints some strings containing some of these control codes.

Listing 2.8:specialchars.py

```
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 2.8 (specialchars.py) produces

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

On most 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 2.9 (escapequotes.py) shows the various ways in which quotation marks may be embedded within string literals.

Listing 2.9: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 2.9 (escapequotes.py) is

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

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

Listing 2.10: `printpath.py`

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

Listing 2.10 (`printpath.py`) displays

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

2.6 User Input

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

```
x = input()
```

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

Listing 2.11: `usinginput.py`

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

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

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

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

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

Listing 2.12: `addintegers.py`

```
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 2.12 (addintegers.py) shows

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

Lines two and four represent user input, while the program generates the other lines. The program halts after printing the first line and does 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 2.12 (addintegers.py) more compactly using this form of the input function as shown in Listing 2.13 (addintegers2.py).

Listing 2.13: 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 2.14 (addintegers3.py) is even shorter. It combines the input and int functions into one statement.

Listing 2.14: 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 2.14 (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 2.13 (addintegers2.py). We frequently will use functional composition to make our program code simpler.

2.7 The eval Function

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

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

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

Listing 2.15: evalfunc.py

```
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 2.15 (evalfunc.py) produces

```
Entry x1? 4
x1 = 4 type: <class 'int'>
Entry x2? 4.0
x2 = 4.0 type: <class 'float'>
Entry x3? 'x1'
x3 = x1 type: <class 'str'>
Entry x4? x1
x4 = 4 type: <class 'int'>
Entry x5? x6
Traceback (most recent call last):
  File "C:\Users\rick\Documents\Code\Other\python\changeable.py", line 13, 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 4, the variable's type is integer. When the user enters 4.0, the variable is a floating-point variable. For x3, the user supplies the string 'x3' (note the quotes), and the variable's type is string. The more interesting situation is x4. The user enters x1 (no quotes). The eval function evaluates the non-quoted text as a reference to the name x1. The program bound the name x1 to the value 4 when executing the first line of the program. Finally, the user enters x6 (no quotes). Since the quotes are missing, the eval function does not interpret x6 as a literal string; instead eval treats x6 as a name 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 can enter multiple entries separated by commas, and the `eval` function evaluates it as a Python tuple. As Listing 2.16 (`addintegers4.py`) shows, this makes tuple assignment (see Section 2.2) possible.

Listing 2.16: `addintegers4.py`

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

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

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

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

Listing 2.17: `enterarith.py`

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

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

```
4 + 10
14
```

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

2.8 Controlling the print Function

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

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

The expression `end=''` is known as a *keyword argument*. The term keyword here means something 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 2.18 (`printingexample.py`).

Listing 2.18: `printingexample.py`

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

Listing 2.18 (`printingexample.py`) displays

```
ABC  
X  
Y  
Z
```

The statement

```
print()
```

essentially moves the cursor down to next line.

Sometimes it is convenient to divide the output of a single line of printed text over several Python statements. As an example, we may want to compute part of a complicated calculation, print an intermediate result, finish the calculation, and print the final answer with the output all appearing on one line of text. The `end` keyword argument allows us to do so.

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

Listing 2.19: `printsep.py`

```
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 2.19 (`printsep.py`) is

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

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

2.9 String Formatting

Consider Listing 2.20 (`powers10left.py`) which prints the first few powers of 10.

Listing 2.20: `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 2.20 (`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 2.21 (`powers10left2.py`) which again prints the first few powers of 10, albeit in most complicated way.

Listing 2.21: `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 2.21 (`powers10left2.py`) produces output identical to Listing 2.20 (`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 10000000000000
13 100000000000000
14 1000000000000000
15 10000000000000000

```

The third print statement in Listing 2.21 (`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 `107 = 10000000` replaces `{1}`. Figure 2.8 shows how the arguments of `format` substitute for the positional parameters in the formatting string.

Listing 2.21 (`powers10left2.py`) provides no advantage over Listing 2.20 (`powers10left.py`), and it is more complicated. Is the extra effort of string formatting ever useful? Observe that in both programs each 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 2.22 (`powers10right.py`) uses a *string formatter* with enhanced positional parameters to right justify the values it prints.

Listing 2.22: `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))

```

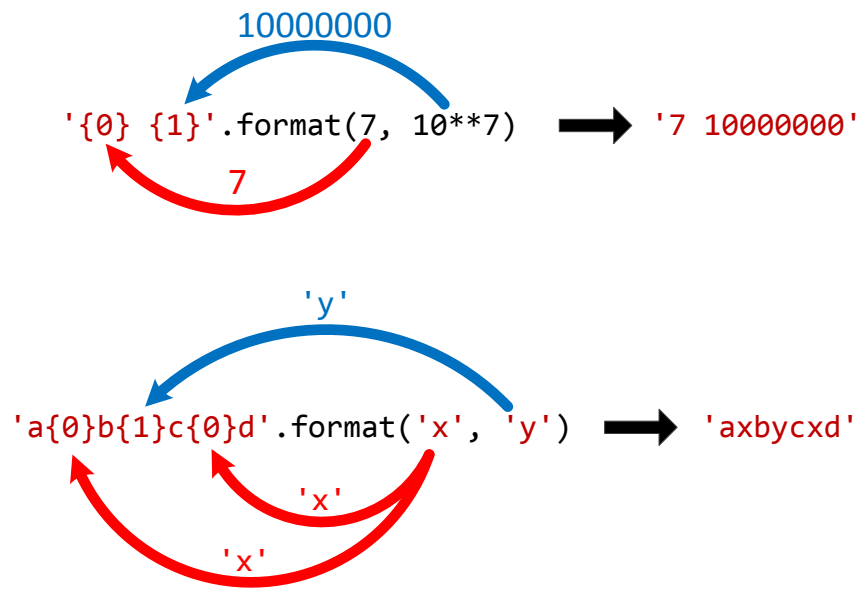


Figure 2.8: Placeholder substitution within a formatting string

```
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 2.22 (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.

2.10 Summary

- Python supports both integer and floating-point kinds of numeric values and variables.
- Python does not permit commas to be used when expressing numeric literals.
- Numbers represented on a computer have limitations based on the finite nature of computer systems.
- Variables are used to store values.
- The `=` operator means *assignment*, not mathematical *equality*.
- A variable can be reassigned at any time.
- A variable must be assigned before it can be used within a program.
- Multiple variables can be assigned in one statement.
- A variable represents a location in memory capable of storing a value.
- The statement `a = b` copies the value stored in variable `b` into variable `a`.
- A variable name is an example of an identifier.
- The name of a variable must follow the identifier naming rules.
- All identifiers must consist of at least one character. The first symbol must be an alphabetic letter or the underscore. Remaining symbols (if any) must be alphabetic letters, the underscore, or digits.
- Reserved words have special meaning within a Python program and cannot be used as identifiers.
- Descriptive variable names are preferred over one-letter names.
- Python is case sensitive; the name `X` is not the same as the name `x`.
- Floating-point numbers approximate mathematical real numbers.
- There are many values that floating-point numbers cannot represent exactly.
- In Python we express scientific notation literals of the form 1.0×10^1 as `1.0e1`.

- Strings are sequences of characters.
- String literals appear within single quote marks (') or double quote marks (").
- Special non-printable control codes like newline and tab are prefixed with the backslash escape character (\).
- The \n character represents a newline.
- The literal backslash character is a string must appear as two successive backslash symbols.
- The input function reads in a string of text entered by the user from the keyboard during the program's execution.
- The input function accepts an optional prompt string.
- Programmers can use the eval function to convert a string representing a numeric expression into its evaluated numeric value.

2.11 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.

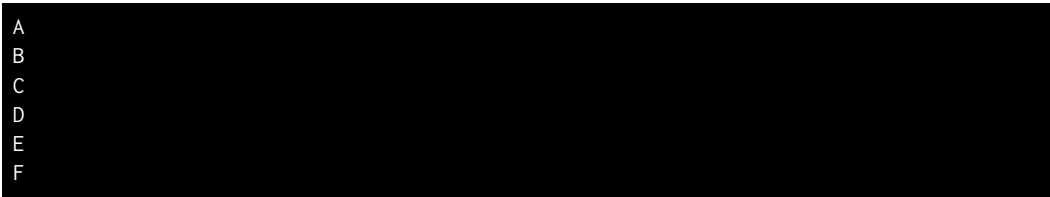
```
x = 7
print(x)
print("x")
```

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 0.0000000000000000000000449 as a much more compact Python literal?
 13. How can you express the literal value 569923412000000000000000000000000000000000000 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:

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

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

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

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

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

This statement is similar to the first statement.

- `sum = value1 + value2;`

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

Table 3.1: Commonly used Python arithmetic binary operators

This is an assignment statement because it 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 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.



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.

$$\begin{array}{r} 6 \\ 4 \overline{) 25} \\ \underline{-24} \\ 1 \end{array}$$

Figure 3.1: 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 non-repeating decimal numbers can be a problem. One-tenth (0.1) is obviously non-repeating, 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 detects 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 use complies or make some other 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

```
(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, -5), or rather is

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

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

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

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

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

```
sum = count = 0
```

which is slightly shorter than tuple assignment:

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

3.4 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.

3.5 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.

3.5.1 Syntax Errors

The interpreter is designed to execute all valid Python programs. The interpreter reads the Python source file and translates it into an 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 running Listing 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.

3.5.2 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
dividend, divisor = eval(input('Please enter two numbers to divide: '))
# 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 integers to divide: 32, 4
32 / 4 = 8.0
```

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

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

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

As another example, consider Listing 3.7 (halve.py).

Listing 3.7: halve.py

```
# Get a number from the user
value = eval(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 = eval(input('Please enter a number to cut in half: '))
  File "<string>", line 1, in <module>
NameError: name 'Bobby' is not defined
```

or

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

Since the programmer cannot predict what the user will provide as input, this program is doomed eventually. Fortunately, in Chapter 13 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.

3.5.3 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 to 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.

3.6 Arithmetic Examples

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

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

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

Listing 3.10 (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.

3.7 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

$$x \text{ op} = \text{exp}$$

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

$$x = x \text{ op } \text{exp}$$

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.

3.8 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. In 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

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

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 = eval(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 of 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.

3.9 Summary

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

- $2 * x + 4$ is an example of a more complex Python numeric expression.
- Expressions can be printed via the `print` function and be assigned to variables.
- A binary operator performs an operation using two operands.
- With regard to binary operators: `+` represents arithmetic addition; `-` represents arithmetic subtraction; `*` represents arithmetic multiplication; `/` represents arithmetic division; `//` represents arithmetic integer division; `%` represents arithmetic modulus, or integer remainder after division.
- A unary operator performs an operation using one operand.
- The `-` unary operator represents the additive inverse of its operand.
- The `+` unary operator has no effect on its operand.
- Arithmetic applied to integer operands yields integer results.
- With a binary operation, floating-point arithmetic is performed if at least one of its operands is a floating-point number.
- Floating-point arithmetic is inexact and subject to rounding errors because floating-point values have finite precision.
- A mixed expression is an expression that contains values and/or variables of differing types.
- In Python, operators have both a precedence and an associativity.
- With regard to the arithmetic operators, Python uses the same precedence rules as standard arithmetic: multiplication and division are applied before addition and subtraction unless parentheses dictate otherwise.
- The arithmetic operators associate left to right; assignment associates right to left.
- Chained assignment can be used to assign the same value to multiple variables within one statement.
- The unary operators `+` and `-` have precedence over the binary arithmetic operators `*`, `/`, `//`, and `%`, which have precedence over the binary arithmetic operators `+` and `-`, which have precedence over the assignment operator.
- Comments are notes within the source code. The interpreter ignores all comments in the source code.
- Comments inform human readers about the code.
- Comments should not state the obvious, but it is better to provide too many comments rather than too few.
- A comment begins with the symbols `#` and continues until the end of the line.
- Source code should be formatted so that it is more easily read and understood by humans.
- Programmers introduce syntax errors when they violate the structure of the Python language.
- The interpreter detects syntax errors during its translation phase before program execution.
- Run-time errors or exceptions are errors that are detected when the program is executing.
- The interpreter detects run-time errors during its execution phase after translation.

- Logic errors elude detection by the interpreter. Improper program behavior indicates a logic error.
- In complicated arithmetic expressions involving many operators and operands, the rules pertaining to mixed arithmetic are applied on an operator-by-operator basis, following the precedence and associativity laws, not globally over the entire expression.
- The `+=` and `-=` operators can be used to increment and decrement variables.
- The family of *op=* operators (`+=`, `-=`, `*=`, `/=`, `//=` and `%=`) allow variables to be changed by a given amount using a particular arithmetic operator.
- Python programs implement algorithms; as such, Python statements do not declare statements of fact or define relationships that hold throughout the program's execution; rather they indicate how the values of variables change as the execution of the program progresses.

3.10 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. Under what circumstances do each of the following run-time errors arise?

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

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

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

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 = eval(input("Please enter the circle's radius: "))
# Print the circumference
print("Circumference is", C)
```

- (a) The program does not produce the intended result. Why?
- (b) How can it be repaired so that it works correctly?

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
```

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

Table 4.1: The Python relational operators

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

Table 4.2: Examples of some Simple Relational Expressions

```

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 3.5 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
```

```
dividend, divisor = eval(input('Please enter two numbers 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: 32, 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 integers to divide: 32, 0
```

the program prints nothing after the user enters the values.

The last non-indented line in Listing 4.2 (`betterdivision.py`) begins with the reserved word `if`. The `if` statement 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:

`if` *condition* :

block

- The reserved word `if` begins a `if` statement.
- The *condition* is a Boolean expression that determines whether or not the body will be executed. A colon (:) must follow the condition.
- The *block* is a block of one or more statements to be executed if the condition is true. The statements within the block must all be indented the same number of spaces from the left. The block within an `if` must be indented more spaces than the line that begins the `if` statement. The block technically is part of the `if` statement. This part of the `if` statement is sometimes called the *body* of the `if`.

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

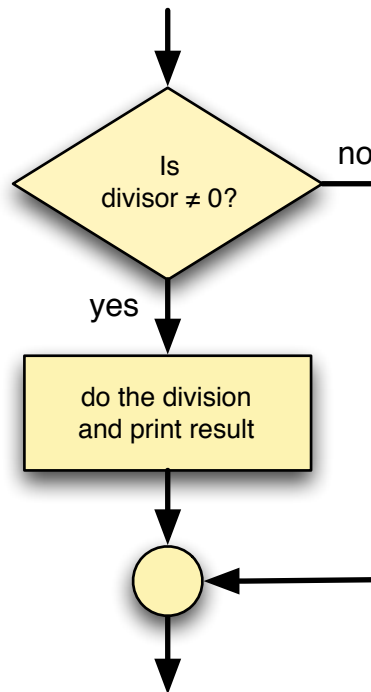


Figure 4.1: if flowchart

```
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.

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

How many spaces should you indent? Python requires at least one, some programmers consistently use two, four is the most popular number, but some prefer a more dramatic display and use eight. A four space indentation for a block is the recommended Python style. This text uses the recommended four spaces to set off each enclosed block. In most programming editors you can set the tab key to insert spaces automatically

so you need not count the spaces as you type. Whichever indent distance you choose, you must use this same distance consistently throughout a Python program.

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

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

Remember when checking for equality, as in

```
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 non-empty string is interpreted as true. Any Python expression can serve as the condition for an `if` statement. In later chapters we will explore additional kinds of expressions and see how they relate to Boolean conditions.

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

Listing 4.4: `leadingzeros.py`

```
# Request input from the user
num = eval(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, divisor = eval(input('Please enter two numbers to divide: '))
# 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 two integers to divide: 32, 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.

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

```
if condition :
    if-block
else:
    else-block
```

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

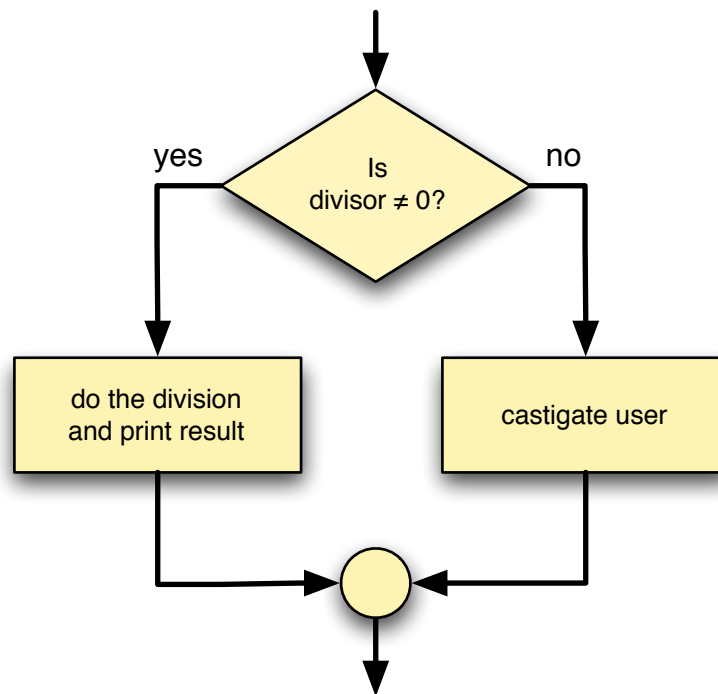


Figure 4.2: if/else flowchart

- 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

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

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

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

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

The Python logical `and` operator works in exactly the same way. If e_1 and e_2 are two Boolean expressions, e_1 `and` e_2 is true only if e_1 and e_2 are both true; if either one is false or both are false, the compound expression is false.

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 conversation we often imply exclusive or in a statement like "Would you like cake `or` ice cream for dessert?" The implication is one or the other, not both. In computer programming the `or` is inclusive; if both subexpressions in an `or` expression are true, the `or` expression is true.

Logical `not` 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 expression `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.

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)
```

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

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

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

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.

The list of our currently know Python operators is shown in Table 4.4.

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

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

Unfortunately, x 's value is irrelevant; the code always prints the word *OK* regardless of the value of x . Since the `==` operator has lower precedence than `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 as well. If x is known to be an integer and not a floating-point number, the expression

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

also would work.

The 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.

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 = eval(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 = eval(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 = eval(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.

...	4	7	3	4	0	6
...	10^5	10^4	10^3	10^2	10^1	10^0
...	100,000	10,000	1,000	100	10	1

$$\begin{aligned}
 473,406 &= 4 \times 10^5 + 7 \times 10^4 + 3 \times 10^3 + 4 \times 10^2 + 0 \times 10^1 + 6 \times 10^0 \\
 &= 400,000 + 70,000 + 3,000 + 400 + 0 + 6 \\
 &= 473,406
 \end{aligned}$$

Figure 4.3: The base 10 place value system

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} .

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 = eval(input("Please enter an integer value in the range 0...1023: "))

```

...	$\boxed{1}$	$\boxed{0}$	$\boxed{0}$	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$
...	2^5	2^4	2^3	2^2	2^1	2^0
...	32	16	8	4	2	1

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

Figure 4.4: The base 2 place value system

```

# 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 $0 \leq \text{value} < 1,024$.
- 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.

- 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 = eval(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)

```

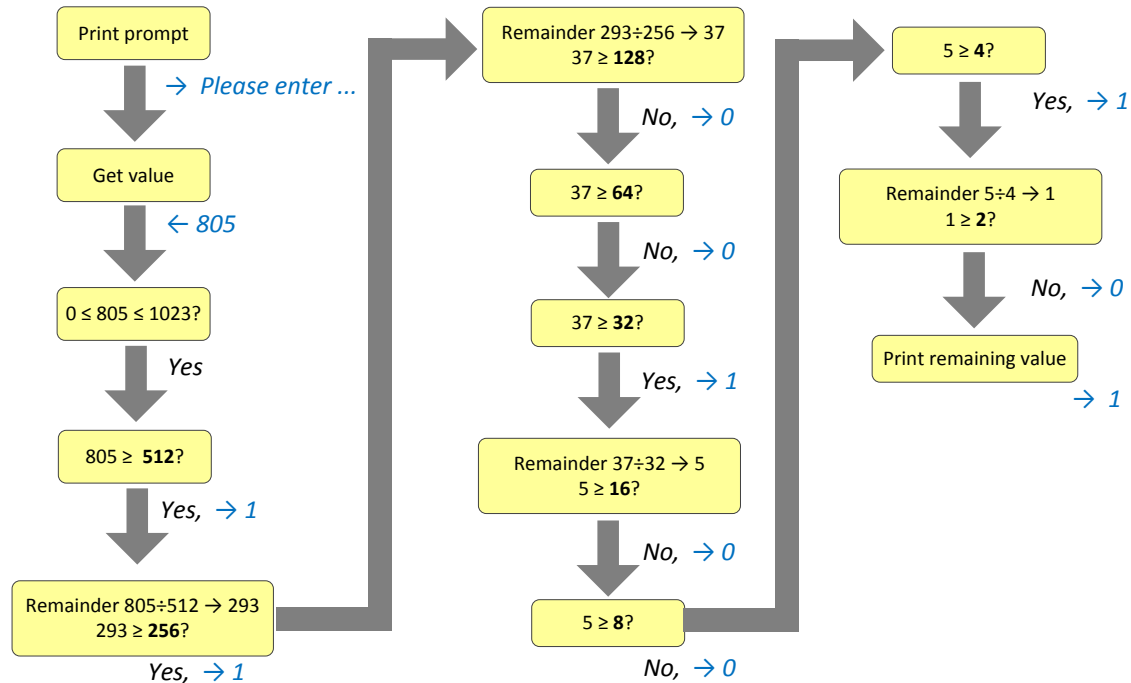


Figure 4.5: The process of the binary number conversion program when the user supplies 805 as the input value.

```

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 ")
                    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.

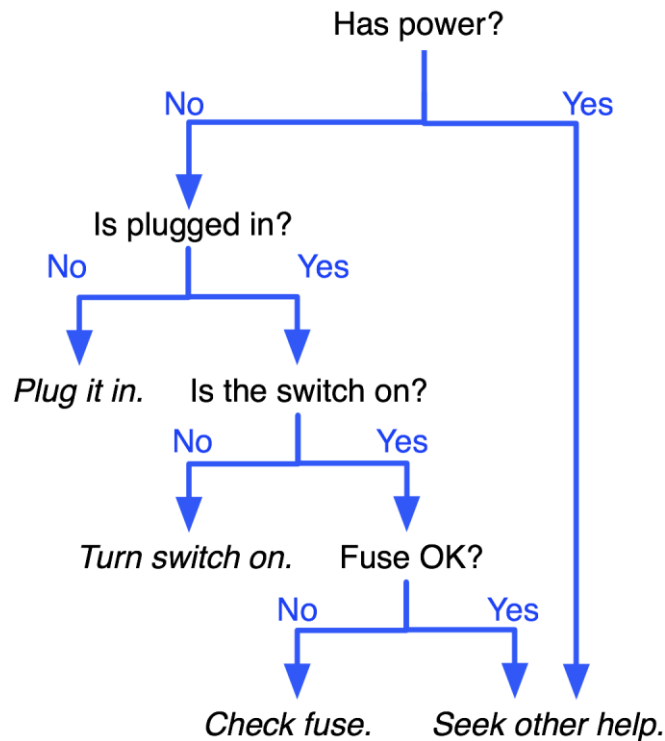


Figure 4.6: Decision tree for troubleshooting a computer system

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 = eval(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 = eval(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 = eval(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 = eval(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

```
if condition-1 :
    block-1
elif condition-2 :
    block-2
elif condition-3 :
    block-3
elif condition-4 :
    block-4
    .
    .
    .
else:
    default-block
```

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

Figure 4.7 compares the structure of the `if/else` statements in a program such as Listing 4.18 (restyleddigittoword.py) to those in a program like Listing 4.12 (binaryconversion.py). In a program like Listing 4.18 (restyleddigittoword.py), the `if/else` statements are nested, while in a program like Listing 4.12 (binaryconversion.py) the `if/else` statements are sequential.

Python provides the tools to construct some very complicated conditional statements. It is important to resist the urge to make things overly complex. Consider the problem of computing the maximum of five integer values provided by the user. The complete solution is left as an exercise in Section 4.13, but here we will outline an appropriate strategy.

Suppose you allow the user to enter all the values at once; for example, for integer variables `n1`, `n2`, `n3`, `n4`, and `n5`:

```
n1, n2, n3, n4, n5 = eval(input('Please enter five integer values: '))
```

Now, allow yourself one extra variable called `max`. All variables have a meaning, and their names should reflect their meaning in some way. We'll let our additional `max` variable mean "maximum I have determined so far." The following is one 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. Follow the same process for `n4` and `n5`.

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 that it is *the* maximum value of all five input numbers. The extra variable `max` is not strictly necessary, but it makes thinking about the problem and its solution easier.

Something to think about: Do you want a series of `if` statements or one large multiway `if/elif/else` construct?

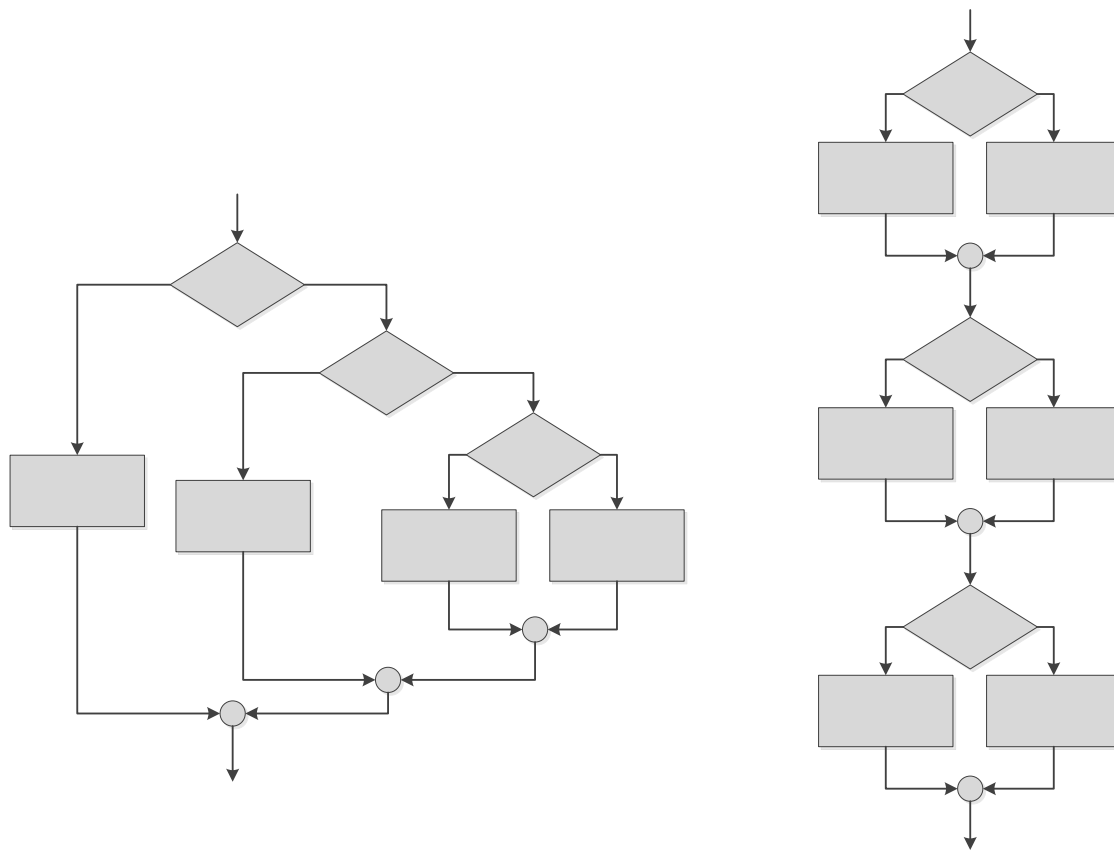


Figure 4.7: The structure of the `if` statements in a program such as Listing 4.18 (`restyleddigittoword.py`) (left) vs. those in a program like Listing 4.12 (`binaryconversion.py`) (right)

Also, you may be tempted to write logic such as

```
if n1 >= n2 and n1 >= n3 and n1 >= n4 and n1 >= n5:
    print('The maximum is', n1)
elif n2 >= n1 and n2 >= n3 and # the rest omitted . . .
```

This will work, but this logic is much more complicated and less efficient (every `>=` and `and` operation requires a few machine cycles to execute). Since it is more complicated, it is more difficult to write correctly, in addition to being more code to type. It is easy to use `>` by mistake instead of `>=`, which will not produce the correct results. Also, if you use this more complicated logic and decide later to add more variables, you will need to change *all* of the `if` conditions in your code and, of course, make sure to modify each one of the conditions correctly. If you implement the simpler strategy outlined before, you need only add one simple `if` statement for each additional variable.

Chapter 5 introduces loops, the ability to execute statements repeatedly. You easily can adapt the first approach to allow the user to type in as many numbers as they like and then have the program report the

maximum number the user entered. The second approach with the more complex logic cannot be adapted in this manner. With the first approach you end up with cleaner, simpler logic, a more efficient program, and code that is easier to extend.

4.10 Conditional Expressions

Consider the following code fragment:

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

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

expression-1 **if** *condition* **else** *expression-2*

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

Listing 4.20: `safedivide.py`

```
# Get the dividend and divisor from the user
dividend, divisor = eval(input('Enter dividend, 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 4.20 (safedivide.py) as Listing 4.21 (safedivideconditional.py).

Listing 4.21: safedivideconditional.py

```
# Get the dividend and divisor from the user
dividend, divisor = eval(input('Enter dividend, 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 4.21 (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 4.22 (absvalueconditional.py) is a small program that provides an example of the conditional expression's use in a statement.

Listing 4.22: absvalueconditional.py

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

Some sample runs of Listing 4.22 (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.

4.11 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`.

4.12 Summary

- Boolean expressions represents the values `True` and `False`.
- The name *Boolean* comes from *Boolean algebra*, the mathematical study of operations on truth values.
- Non-zero numbers and non-empty strings represent true Boolean values. Zero (integer or floating-point) and the empty string (`' '` or `''`) represent false.
- Expressions involving the relational operators (`==`, `!=`, `<`, `>`, `<=`, and `>=`) evaluate to Boolean values.
- Boolean expressions can be combined via `and` (logical *AND*) and `or` (logical *OR*).
- `not` represents logical *NOT*.

- The `if` statement can be used to optionally execute statements.
- The block of statements that are part of the `if` statement are executed only if the `if` statement's condition is true.
- The `if` statement has an optional `else` block to require the selection between two alternate paths of execution.
- The `if/else` statements can be nested to achieve arbitrary complexity.
- The `if/elif/else` statements allow selection of one block of code to execute from many possible options.
- The conditional expression is an expression that evaluates to one of two values depending on a given condition.
- Complex Boolean expressions require special attention, as they are easy to get wrong.

4.13 Exercises

1. What possible values can a Boolean expression have?
2. Where does the term Boolean originate?
3. What is an integer equivalent to `True` in Python?
4. What is the integer equivalent to `False` in Python?
5. Is the value `-16` interpreted as `True` or `False`?
6. Given the following definitions:

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

evaluate the following Boolean expressions:

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

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

evaluate the following Boolean expressions:

- (a) b3
- (b) b4
- (c) not b1
- (d) not b2
- (e) not b3
- (f) not b4
- (g) b1 and b2
- (h) b1 or b2
- (i) b1 and b3
- (j) b1 or b3
- (k) b1 and b4
- (l) b1 or b4
- (m) b2 and b3
- (n) b2 or b3
- (o) b1 and b2 or b3
- (p) b1 or b2 and b3
- (q) b1 and b2 and b3
- (r) b1 or b2 or b3
- (s) not b1 and b2 and b3
- (t) not b1 or b2 or b3
- (u) not (b1 and b2 and b3)
- (v) not (b1 or b2 or b3)
- (w) not b1 and not b2 and not b3
- (x) not b1 or not b2 or not b3
- (y) not (not b1 and not b2 and not b3)
- (z) not (not b1 or not b2 or not b3)

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

- (a) not (x == 2)
- (b) x < 2 or x == 2
- (c) not (x < y)
- (d) not (x <= y)
- (e) x < 10 and x > 20
- (f) x > 10 or x < 20
- (g) x != 0

(h) `x == 0`

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

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

(a) 3

(b) 21

(c) 5

(d) 17

(e) -5

17. 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.

18. 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:

```
while condition :
```

```
    block
```

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

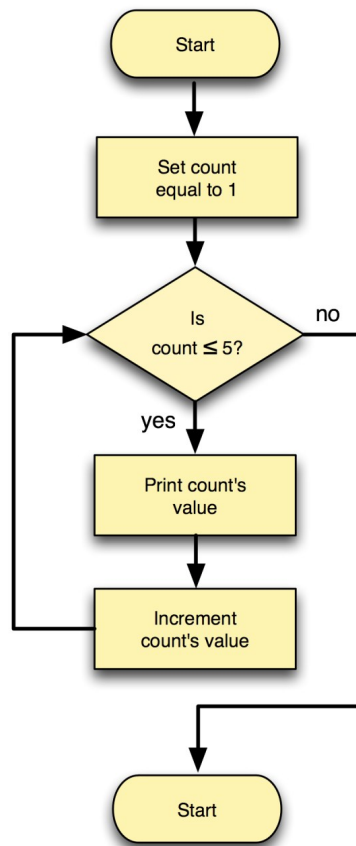


Figure 5.1: while flowchart for Listing 5.2 (iterativecounttofive.py)

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* nor *n*.

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

Listing 5.4: addnonnegatives.py

```

# Allow the user to enter a sequence of non-negative
# numbers. The user ends the list with a negative
# number. At the end the sum of the non-negative
# numbers entered is displayed. The program prints
# zero if the user provides no non-negative 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 = eval(input()) # Get the value
    if entry >= 0:         # Is number non-negative?
        sum += entry      # Only add it if it is non-negative
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 non-negative values entered by the user.

The initialization of entry to zero coupled with the condition `entry >= 0` of the `while` guarantees that the 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.

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
```

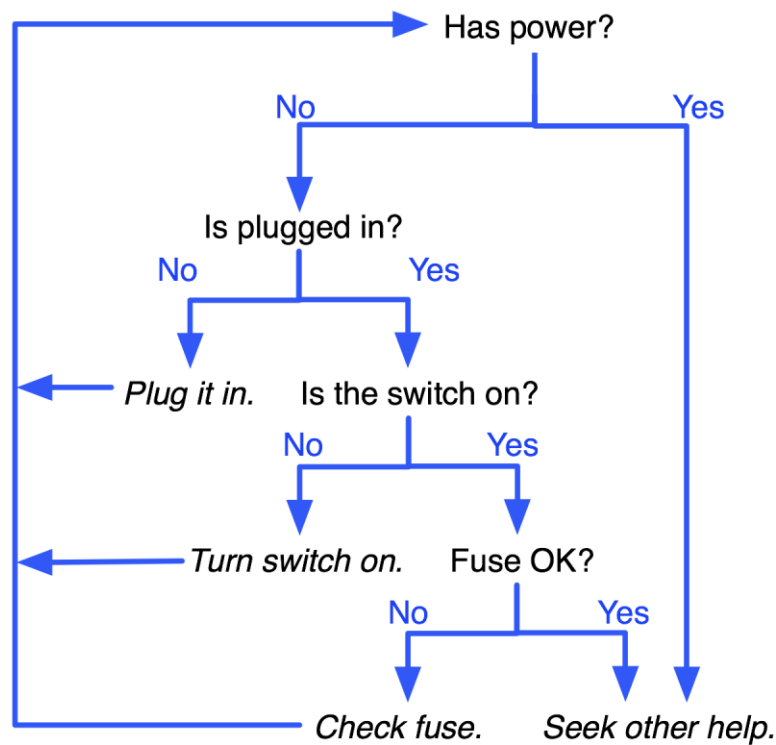


Figure 5.2: Decision tree for troubleshooting a computer system

```

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 non-zero
    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 non-zero
    x -= 1       # Decrement x

```

5.2 Definite Loops vs. Indefinite Loops

In Listing 5.6 (`definite1.py`), code similar to Listing 5.1 (`counttofive.py`), prints the integers from one to 10.

Listing 5.6: `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 5.7 (`definite2.py`).

Listing 5.7: definite2.py

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

Looking at the source code of Listing 5.7 (`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 5.7 (`definite2.py`) is considered to be a definite loop as well.

Compare these programs to Listing 5.8 (`indefinite.py`).

Listing 5.8: indefinite.py

```
done = False                # Enter the loop at least once
while not done:
    entry = eval(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 5.8 (`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 known 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 5.8 (`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 range of values. These values can be a numeric range, or, as we shall, elements of a data structure like a string, list, or tuple. We can rewrite the above `while` loop as

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

The expression `range(1, 11)` creates an object known as an *iterable* that allows the `for` loop to assign to the variable `n` the values 1, 2, ..., 10. 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, ..., 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)` → 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- `range(1, 10)` → 1, 2, 3, 4, 5, 6, 7, 8, 9
- `range(1, 10, 2)` → 1, 3, 5, 7, 9
- `range(10, 0, -1)` → 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
- `range(10, 0, -2)` → 10, 8, 6, 4, 2
- `range(2, 11, 2)` → 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 2.22 (`powers10right.py`) more compactly. Listing 5.9 (`powers10loop.py`) uses a `for` loop to print the first 16 powers of 10.

Listing 5.9: `powers10loop.py`

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

Listing 5.9 (`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

```

The `for` loop can iterate over any iterable object. Like the object created by the range expression, a string is an iterable object. We can use a `for` loop to iterate over the characters that comprise a string. Listing 5.10 (`stringletters.py`) uses a `for` loop to print the individual characters of a string.

Listing 5.10: `stringletters.py`

```

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

```

In the following sample execution of Listing 5.10 (`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 5.10 (`stringletters.py`) assigns to the `letter` variable a string containing a single character.

Listing 5.11 (`stringliteralletters.py`) uses a `for` loop to iterate over a literal string.

Listing 5.11: `stringliteralletters.py`

```

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

```

Listing 5.11 (`stringliteralletters.py`) prints

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

Listing 5.12 (`countvowels.py`) counts the number of vowels in the text provided by the user.

Listing 5.12: `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 5.11 (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)

```

In a `for` loop the range object has complete control over determining the loop variable each time through the loop. To prove this, Listing 5.13 (abusefor.py) attempts to thwart the range's loop variable by changing its value inside the loop.

Listing 5.13: 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 5.13 (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 expression. 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.

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

×	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

Figure 5.3: A 10×10 multiplication table

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 5.14 (`timestable1.py`) shows our first attempt at a multiplication table.

Listing 5.14: `timestable1.py`

```
# Get the number of rows and columns in the table
size = eval(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 5.14 (`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 5.14 (`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 1 to size. Listing 5.15 (`timestable2.py`) contains the needed refinement.

Listing 5.15: `timestable2.py`

```
# Get the number of rows and columns in the table
size = eval(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 5.15 (`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 2.22 (`powers10right.py`) to right justify the numbers within a four-digit area. Listing 5.16 (`timestable3.py`) contains this alignment adjustment.

Listing 5.16: `timestable3.py`

```
# Get the number of rows and columns in the table
size = eval(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 5.16 (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 5.17 (timestable4.py) adds the necessary code.

Listing 5.17: timestable4.py

```
# Get the number of rows and columns in the table
size = eval(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 5.17 (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 5.17 (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 5.18 (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 5.18: 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 5.18 (permuteabc.py) is all six permutations of ABC:

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

Listing 5.19 (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 5.19: 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 5.20 (`whileexitatop.py`) demonstrates this top-exit behavior.

Listing 5.20: `whileexitatop.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 5.20 (`whileexitatop.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 5.21 (`addmiddleexit.py`) is a variation of Listing 5.4 (`addnonnegatives.py`) that illustrates the use of `break`.

Listing 5.21: `addmiddleexit.py`

```
# Allow the user to enter a sequence of non-negative
# numbers. The user ends the list with a negative
# number. At the end the sum of the non-negative
# numbers entered is displayed. The program prints
# zero if the user provides no non-negative 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 = eval(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 5.21 (`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 5.21 (`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 5.22 (`troubleshootloop2.py`) uses `break` statements in place of the Boolean `done` variable.

Listing 5.22: `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 5.21 (`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 does not use a `break`. Figure 5.4 shows how we can transform any `while` loop that uses a `break` statement into a `break`-free version. The no-`break` version introduces a Boolean variable (`looping`), and the

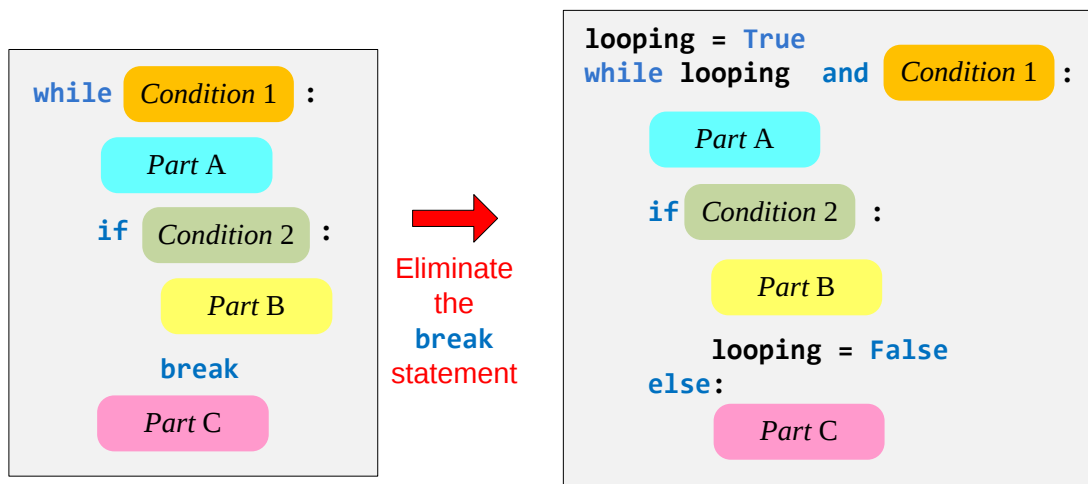


Figure 5.4: The code on the left generically represents any `while` loop that uses a `break` statement. The code on the right shows how we can transform the loop into a functionally equivalent form that does not use `break`.

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 5.23 (`countvowelsnox.py`) shows how we can use a `break` statement to exit a `for` loop prematurely, provided by the user.

Listing 5.23: 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 litle 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 would make the 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 5.24 (continueexample.py) shows the *continue* statement in action.

Listing 5.24: continueexample.py

```
sum = 0
done = False
while not done:
    val = eval(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 5.25 (`nocontinueexample.py`) works exactly like Listing 5.24 (`continueexample.py`), but it avoids the `continue` statement.

Listing 5.25: `nocontinueexample.py`

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

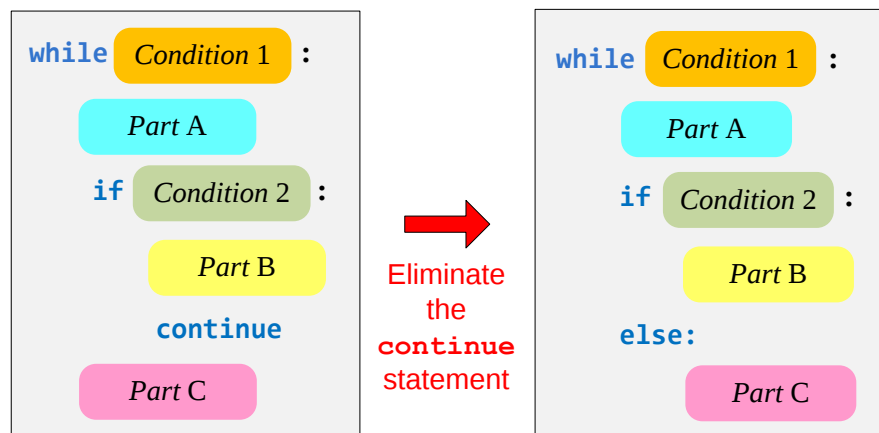


Figure 5.5: The code on the left generically represents any loop that uses a `continue` statement. It is possible to transform the code on the left to eliminate the `continue` statement, as the code on the right shows.

version is no more complex than the `continue` version. Therefore, unlike the `break` statement above, there is no compelling reason to use the `continue` statement. Sometimes a `continue` statement is added at the last minute to an existing loop body to handle an exceptional condition (like ignoring negative numbers in the example above) that initially went unnoticed. If the body of the loop is lengthy, a conditional statement with a `continue` can be added easily near the top of the loop body without touching the logic of the rest of the loop. Therefore, the `continue` statement merely provides a convenient alternative to the programmer.

The `else` version is preferred.

5.6 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 5.26 (`whileelse.py`) shows how the `while/else` statement works.

Listing 5.26: `whileelse.py`

```
# Add five non-negative numbers supplied by the user
count = sum = 0
print('Please provide five non-negative numbers when prompted')
while count < 5:
    # Get value from the user
    val = eval(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 non-negative values to Listing 5.26 (`whileelse.py`), it computes the average of the values provided:

```
Please provide five non-negative 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 non-negative 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 5.27 (`whilenoeelse.py`) uses `if/else` statement to achieve the same effect as Listing 5.26 (`whileelse.py`).

Listing 5.27: `whilenoeelse.py`

```
# Add five non-negative numbers supplied by the user
count = sum = 0
print('Please provide five non-negative numbers when prompted')
while count < 5:
    # Get value from the user
    val = eval(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 5.27 (`whilenoeelse.py`) uses two distinct Python constructs, the `while` statement followed by an `if/else` statement, whereas Listing 5.26 (`whileelse.py`) uses only one, a `while/else` statement. Listing 5.27 (`whilenoeelse.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 5.28 (`countvowelself.py`) shows how the `else` block works with a `for` statement.

Listing 5.28: `countvowelself.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 5.12 (`countvowels.py`), Listing 5.28 (`countvowelself.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 5.29 (`findfactors.py`) that attempts to print all the integers with their associated factors from 1 to 20.

Listing 5.29: `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 5.29 (`findfactors.py`) the debugging task is nontrivial since the program involves nested loops. (Can you find and fix the problem in Listing 5.29 (`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 5.29 (`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 5.29 (`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 5.30 (`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 5.30: `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
```

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

Listing 5.31: `computesquareroot.py`

```
# File computesquareroot.py

# Get value from the user
val = eval(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 2.0069444444444444
1.4142156862745097 squared is 2.000060073048824
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
10.00052895642693 squared is 100.0010579156518
Square root of 100 = 10.00000000139897

```

The real answer, of course, is 10, but our computed result again is well within our programmed tolerance.

While Listing 5.31 (`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 5.32 (startree.py) provides the necessary functionality.

Listing 5.32: startree.py

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

# Draw one row for every unit of height
row = 0
while row < height:
    # Print leading spaces; as row gets bigger, the number of
    # leading spaces gets smaller
    count = 0
    while count < height - row:
        print(end=' ')
        count += 1

    # Print out stars, twice the current row plus one:
    # 1. number of stars on left side of tree
    #    = current row value
    # 2. exactly one star in the center of tree
    # 3. number of stars on right side of tree
    #    = current row value
    count = 0
    while count < 2*row + 1:
        print(end='*')
        count += 1
    # Move cursor down to next line
    print()
    row += 1    # Consider next row
```

The following shows a sample run of Listing 5.32 (startree.py) where the user enters 7:

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

```

*****
*****
*****
*****
*****
*****

```

Listing 5.32 (starttree.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 5.33 (starttreefor.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 5.33: starttreefor.py

```

# Get tree height from user
height = eval(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 5.34 (printprimes.py) provides one solution.

Listing 5.34: printprimes.py

```

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

```

Listing 5.34 (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 5.34 (printprimes.py) is a little more complex than that of Listing 5.32 (starttree.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 5.34 (`printprimes.py`) a bit by using `for` statements instead of `while` statements as shown in Listing 5.35 (`printprimesfor.py`).

Listing 5.35: `printprimesfor.py`

```
max_value = eval(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 simply Listing 5.35 (`printprimesfor.py`) even further by using the `for/else` statement as Listing 5.36 (`printprimesforelse.py`) illustrates.

Listing 5.36: `printprimesforelse.py`

```
max_value = eval(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 5.37 (`betterinputonly.py`) traps the user in a loop until the user provides an acceptable integer value.

Listing 5.37: `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 5.37 (`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 5.37 (`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 Summary

- The `while` statement allows the execution of code sections to be repeated multiple times.
- The condition of the `while` controls the execution of statements within the `while`'s body.
- The statements within the body of a `while` are executed over and over until the condition of the `while` is false.
- If the `while`'s condition is initially false, the body is not executed at all.
- In an infinite loop, the `while`'s condition never becomes false.
- The statements within the `while`'s body must eventually lead to the condition being false; otherwise, the loop will be infinite.
- Do not confuse `while` statements with `if` statements; their structure is very similar (`while` reserved word instead of the `if` word), but they behave differently.
- Infinite loops are rarely intentional and usually are accidental.
- An infinite loop can be diagnosed by putting a printing statement inside its body.
- A loop contained within another loop is called a nested loop.
- Iteration is a powerful mechanism and can be used to solve many interesting problems.
- Complex iteration using nested loops mixed with conditional statements can be difficult to do correctly.
- The `break` statement immediately exits a loop, skipping the rest of the loop's body, without checking to see if the condition is true or false. Execution continues with the statement immediately following the body of the loop.
- In a nested loop, the `break` statement exits only the loop in which the `break` is found.
- The `continue` statement immediately checks the loop's condition, skipping the rest of the loop's body. If the condition is true, the execution continues at the top of the loop as usual; otherwise, the loop is terminated and execution continues with the statement immediately following the loop's body.
- In a nested loop, the `continue` statement affects only the loop in which the `continue` is found.

5.10 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 = eval(input()) # Get the value
```

were moved out of the loop? Is moving the assignment out of the loop a good or bad thing to do? Why?

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

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

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

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

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

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

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

```
a = 0
while a < 100:
    b = 0
    while b < 55:
        print('*', end='')
        b += 1
    print()
    a += 1
```

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

```
a = 0
while a < 100:
    if a % 5 == 0:
        print('*', end='')
    a += 1
print()
```

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

```
a = 0
while a < 100:
    b = 0
    while b < 40:
        if (a + b) % 2 == 0:
```

```
        print('*', end='')
    b += 1
print()
a += 1
```

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

```
a = 0
while a < 100:
    b = 0
    while b < 100:
        c = 0
        while c < 100:
            print('*', end='')
            c += 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. How many asterisks does the following code fragment print?

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

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

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

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

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

29. What is printed by the following code fragment?

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

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

31. What is printed by the following code fragment?

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

32. Rewrite the following code fragment using a `break` statement and eliminating the `done` variable. Your code should behave identically to this code fragment.

```
done = False
n, m = 0, 100
while not done and n != m:
    n = eval(input())
    if n < 0:
        done = True
    print("n =", n)
```

33. Rewrite the following code fragment so it does not use a `break` statement. Your code should behave identically to this code fragment.

```
// Code with break ...
```

34. Rewrite the following code fragment so it eliminates the `continue` statement. Your new code's logic should be simpler than the logic of this fragment.

```
x = 100
while x > 0:
    y = eval(input())
    if y == 25:
        x += 1
```

```

        continue
    x = eval(input())
    print('x =', x)

```

35. What is printed by the following code fragment?

```

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

```

36. Modify Listing 5.17 (timestable4.py) so that 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 user enters 15, a 15×15 table should be printed. Print nothing if the user enters a value larger than 18. Be sure everything lines up correctly, and the table looks attractive.

37. 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 user 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.

38. 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.
39. Write a Python program that allows the user to enter any number of non-negative floating-point values. The user terminates the input list with any negative value. The program then prints the sum, average (arithmetic mean), maximum, and minimum of the values entered. The terminating negative value is **not** used in the computations.

40. Redesign Listing 5.32 (starttree.py) so that it draws a sideways tree pointing right; for example, if the user enters 7, the program would print

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

41. Redesign Listing 5.32 (starttree.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 5.31 (`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 $\{a_1, a_2, a_3, \dots, a_n\}$ is

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

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 5.31 (`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 5.31 (`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 code stored in libraries called *modules*. Programmers can use parts of this library code within their own code to build sophisticated programs.

6.1 Introduction to Using Functions

We have been using functions in Python since the first chapter. These functions include `print`, `input`, `eval`, `int`, `float`, `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 $f(5) = 2 \cdot 5 + 3 = 13$ 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. A program uses a function when specific processing is required. One example of a function is the mathematical square root function. Python has a function in its standard library named `sqrt` (see Section 6.2). The square root function accepts one numeric (integer or floating-point) value and produces a floating-point result; for example, $\sqrt{16} = 4$, so when presented with 16.0, `sqrt` responds with 4.0. Figure 6.1 illustrates the conceptual view of the `sqrt` function. To the user of the square root function, the function is a black box; the user is concerned more



Figure 6.1: Conceptual view of the square root function

about *what* the function does, not *how* it does it.

This `sqrt` function is exactly what we need for our square root program, Listing 5.31 (`computesquareroot.py`). The new version, Listing 6.1 (`standardsquareroot.py`), uses the library function `sqrt` and eliminates the complex logic of the original code.

Listing 6.1: `standardsquareroot.py`

```
from math import sqrt

# Get value from the user
num = eval(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`.

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*. A module is a collection of Python code that can be used in other programs. The statement

```
from math import sqrt
```

makes the `sqrt` function available for use in the program. The `math` module has many other mathematical functions. These include trigonometric, logarithmic, hyperbolic, and other mathematical functions.

When calling a function, 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 cannot change the value of `num` as far as the caller is concerned, it simply uses the variable's value to perform the computation. It is as if we write down the value of `num` on a piece of paper, hand it to `sqrt`, and `sqrt` hands us back a note with the answer. The `sqrt` function does not have access to our original `num` variable; it has only a copy of `num`, as if “written on a piece of paper.” After `sqrt` is finished and gives us its computed answer, it discards its copy of `num` (by analogy, the function “throws away the paper with the copy of `num` we gave it”). Thus, during a function call parameters are temporary, transitory values used only to communicate information to the function.

The `sqrt` function can be called many in other ways, as illustrated in Listing 6.2 (`usingsqrt.py`):

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 is returned to the caller. The caller can use this result in various ways, as shown in Listing 6.2 (`usingsqrt.py`). The statement

```
print(sqrt(16.0))
```

directly prints the result of computing the square root of 16. The statement

```
y = sqrt(x)
```

assigns the result of the function call to the variable `y`. The statement

```
y = sqrt(sqrt(256.0))
```

computes $\sqrt{\sqrt{256}} = \sqrt{16} = 4$. The statement

```
print(sqrt(int('45')))
```

prints the result of computing the square root of the integer equivalent of the string `'45'`.

If the 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, to the caller of the square root function the function is a black box; 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. We are guaranteed that we can influence the function's behavior only via the parameters that we pass, and that nothing else we do can affect what the function does or how it does it. Furthermore, for the types of objects we have considered so far (integers, floating-point numbers, and strings), when a 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.

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` package. 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 value `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 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.

math Module	
sqrt	Computes the square root of a number: $\text{sqrt}(x) = \sqrt{x}$
exp	Computes e raised a power: $\text{exp}(x) = e^x$
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: $\text{pow}(x, y) = x^y$
degrees	Converts a value in radians to degrees: $\text{degrees}(x) = \frac{\pi}{180}x$
radians	Converts a value in degrees to radians: $\text{radians}(x) = \frac{180}{\pi}x$
fabs	Computes the absolute value of a number: $\text{fabs}(x) = x $

Table 6.1: A few of the functions from the math package

The math package also defines the values pi (π) and e (e).

The parameter passed by the 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. The call $\text{pow}(10, 2)$ computes $10^2 = 100$, but the call $\text{pow}(2, 10)$ computes $2^{10} = 1,024$.

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 6.2. 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 the satellite will be from the spacecraft as it orbits the planet. 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

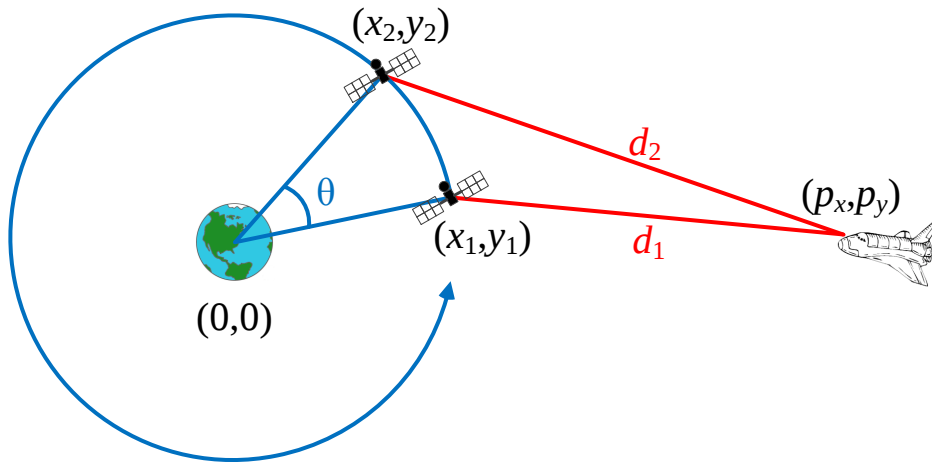


Figure 6.2: Orbital distance problem. In this diagram, the satellite begins at point (x_1, y_1) , a distance of d_1 from the spacecraft. The satellite's orbit takes it to point (x_2, y_2) after an angle of θ rotation. The distance to its new location is d_2 .

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 6.2 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 package
from math import sqrt, sin, cos, pi, radians

# Get coordinates of the stationary spacecraft, (px, py)
px, py = eval(input("Enter coordinates of spacecraft (x,y):"))

# Get starting coordinates of satellite, (x1, y1)
x, y = eval(input("Enter initial satellite coordinates (x,y):"))

# 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
    new_x = x*COS_theta - y*SIN_theta # Compute new x value
    new_y = x*SIN_theta + y*COS_theta # Compute new y value
    x, y = new_x, new_y              # Update (x, y)

```

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 coordinates of spacecraft (x,y):100000, 0
Enter initial satellite coordinates (x,y):20000, 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 tuple 100000, 0 and then the tuple 20000, 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

```

in separate statements rather than using tuple assignment

We can use the square root function to improve the efficiency of Listing 5.34 (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 that need be considered.

Listing 6.4: moreefficientprimes.py

```

from math import sqrt

```

```

max_value = eval(input('Display primes up to what value? '))
value = 2 # Smallest prime number

while value <= max_value:
    # See if value is prime
    is_prime = True # Provisionally, value is prime
    # Try all possible factors from 2 to value - 1
    trial_factor = 2
    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

```

6.3 time Functions

The time package contains a number of functions that relate to time. We will consider two: `clock` and `sleep`.

The `clock` function allows us measure the time of parts of a program's execution. The `clock` returns a floating-point value representing elapsed time in seconds. On Unix-like systems (Linux and Mac OS X), `clock` returns the numbers of seconds elapsed since the program began executing. Under Microsoft Windows, `clock` returns the number of seconds since the first call to `clock`. In either case, with two calls to the `clock` function we can measure *elapsed time*. Listing 6.5 (`timeit.py`) measures how long it takes a user to enter a character from the keyboard.

Listing 6.5: `timeit.py`

```

from time import clock

print("Enter your name: ", end="")
start_time = clock()
name = input()
elapsed = clock() - 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 clock

sum = 0          # Initialize sum accumulator
start = clock()  # Start the stopwatch
for n in range(1, 100000001): # Sum the numbers
    sum += n
elapsed = clock() - 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 5.35 (printprimesfor.py).

Listing 6.7: measureprimespeed.py

```

from time import clock

max_value = 10000
count = 0
start_time = clock() # 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 = clock() - 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 clock

max_value = 10000
count = 0
value = 2      # Smallest prime number
start = clock() # 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 = clock() - start # Stop the stopwatch
    print("Count:", count, " Elapsed time:", elapsed, "sec")
```

An even faster prime generator appears in Listing 10.23 (`fasterprimes.py`); it uses a completely different algorithm to generate prime numbers.

The `sleep` function suspends the program's execution for a specified number of seconds. Listing 6.9 (`countdown.py`) counts down from 10 with one second intervals between numbers.

Listing 6.9: `countdown.py`

```
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 `sleep` function is useful for controlling the speed of graphical animations.

6.4 Random Numbers

Some applications require behavior that appears random. Random numbers are particularly useful in games and simulations. For example, many board games use a die (one of a pair of dice) to determine how many places a player is to advance. (See Figure 6.3.) A die or pair of dice are used in other games of chance. A die is a cube containing spots on each of its six faces. The number of spots range from one to six. A player rolls a die or sometimes a pair of dice, and the side(s) that face up have meaning in the game being played. The value of a face after a roll is determined at random by the complex tumbling of the die. A software adaptation of a game that involves dice would need a way to simulate the random roll of a die.

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

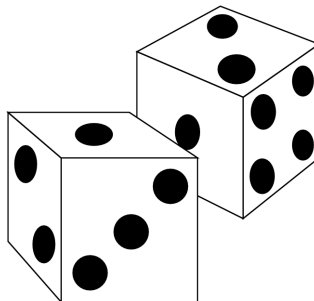


Figure 6.3: A pair of dice

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.

randomfunctions Module	
random	Returns a pseudorandom floating-point number x in the range $0 \leq x < 1$
randrange	Returns a pseudorandom integer value within a specified range.
seed	Sets the random number seed.

Table 6.2: A few of the functions from the random package

The seed function establishes the initial value from which the sequence of pseudorandom numbers is generated. Each call to random or randrange returns the next value in the sequence of pseudorandom values. Listing 6.10 (simplerandom.py) prints 100 pseudorandom integers in the range 1...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 newline
```

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 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 `randrange`. The call `randint(a, b)` is equivalent to `randrange(a, b + 1)`.

6.5 Importing Issues

Python provides three ways to import functions from a module:

- Import one or more specific functions:

```
from math import sqrt, log
```

This statement makes only the `sqrt` and `log` 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.

- Import everything the module has to offer:

```
from math import *
```

The `*` symbol represents “everything.” This statement makes all the code in the `math` module available to the program. If a program needs to use many different functions from the `math` module, some programmers prefer this approach.

- Import the module itself instead of just its components:

```
import math
```

In this case, to use a function the caller must use the following 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 name like this a *qualified name*. The qualified name includes the module name and function name. Many programmers prefer this approach because the complete name unambiguously identifies the function.

Of the three varieties of `import` statements, the “import all” statement is in some ways the easiest to use. The mindset is, “Import everything because we may need some things in the module, but we are not sure exactly what we need starting out.” The source code is shorter: `*` is quicker to type than a list of function names, and short function names are easier to type than qualified function names. While in the short term the “import all” approach may appear to be attractive, in the long term it can lead to problems. As an example, suppose a programmer is writing a program that simulates a chemical reaction in which the rate of the reaction is related logarithmically to the temperature. The statement

```
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, including a function named `degrees` which converts angle measurements 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. The programmer is free to redefine `degrees` to be a floating-point variable (recall redefining the `print` function in Section 2.3), but then the `math` module’s `degrees` function is unavailable if it is needed later. A *name collision* results if the programmer tries to use the same name for both the angle conversion and temperature representation. The same name cannot be used simultaneously for both purposes.

The names of variables and functions available to a program live in that program’s *namespace*. 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 demonstrated with the name `degrees`, and it makes larger programs more difficult to work with and less maintainable.

To summarize, you should avoid the “import everything” statement

```
from math import *
```

since this provides more opportunities for name collisions and makes your code less maintainable. The best approach imports the whole module

```
import math
```

and uses qualified names for the functions the module provides. In the above example, this module import approach solves the name collision problem: `math.degrees` is a different name than `degrees`. A compromise imports only the functions needed:

```
from math import sqrt, log
```

This does not impact the program’s namespace very much, and it allows the program to use short function names. Also, by explicitly naming the functions to import, the programmer is more aware of how the names will impact the program.

You can think of a module as a toolbox. The `math` module is a box containing mathematics tools. The statement

```
from math import *
```

is like bringing the math toolbox into your workroom and dumping everything out on the floor. It may be handy at times, but it makes a mess and can be dangerous (you might trip over one of the tools on the floor).

The statement

```
import math
```

is like bringing the math toolbox into your workroom. When you need a mathematics tool you take it out of the box and use it. When you are finished with it, even if you may need it later, you put it back in the toolbox. If you need it later, you can take it out again because you know right where it is. It is a little more work, but it is more organized.

The statement

```
from math import sqrt, log10
```

is like bringing the math toolbox into your workroom and taking out the two mathematics tools you need for a project. You don't put the tools back until you are finished with them completely. It is not as messy, and you are less likely to trip over a tool on the floor.

6.6 Summary

- The Python standard library provides a collection of functions that you can incorporate into code that you write.
- When faced with the choice of using a standard library function or writing your own code to solve the same problem, choose the library function. The standard function will be tested thoroughly, well documented, and likely more efficient than the code you would write.
- The function is a standard unit of reuse in Python.
- Code that uses a function is known as *caller* code.
- A function has a name, a list of parameters (which may be empty), and a result (which may be `None`). A function performs some computation or action that is useful to callers. Typically a function produces a result based on the parameters passed to it.
- Clients communicate information to a function via its parameters (also known as arguments).
- Standard library functions are organized into modules.
- A module contains a collection of related functions.
- In order to use many standard functions, a caller must use an `import` statement so that the interpreter will use function definitions from the proper module.
- The arguments passed to a function by a caller consist of a comma-separated list enclosed by parentheses.
- Clients calling a function must pass the correct number and types of parameters that the function expects.
- The Python standard module `math` includes a variety of mathematical functions.
- The `clock` function from the `time` module may be used to measure the execution time of parts of programs.

- The `sleep` function suspends the program's execution for a specified number of seconds.
- The `random` module contains a number of functions for working with pseudorandom numbers.
- `randrange(x,y)` returns a pseudorandom integer in the range $x \dots y$. `random()` returns a pseudorandom floating-point number x in the range $0 \leq x < 1$.
- There are three ways to import functions from modules: import certain functions only, import everything, and import the module itself as a unit.
- The complete module import is the best approach, but it requires programmers to use the longer qualified names for functions.
- You should avoid the "import everything" from a module statement. This pollutes the program's namespace and can make programs less maintainable.
- The limited import approach is a compromise between importing everything and importing the module as a unit.

6.7 Exercises

1. Suppose you need to compute the square root of a number in a Python program. Would it be a good idea to write the code to perform the square root calculation? Why or why not?
2. Which of the following values could be produced by the call `random.randrange(0, 100)` function (circle all that apply)?
4.5 34 -1 100 0 99
3. Classify each of the following expressions as *legal* or *illegal*. Each expression represents a call to a standard Python library function.
 - (a) `math.sqrt(4.5)`
 - (b) `math.sqrt(4.5, 3.1)`
 - (c) `random.rand(4)`
 - (d) `random.seed()`
 - (e) `random.seed(-1)`
4. From geometry: Write a computer program that, given the lengths of the two sides of a right triangle adjacent to the right angle, computes the length of the hypotenuse of the triangle. (See Figure 6.4.) 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 $1 \dots 100$. 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.

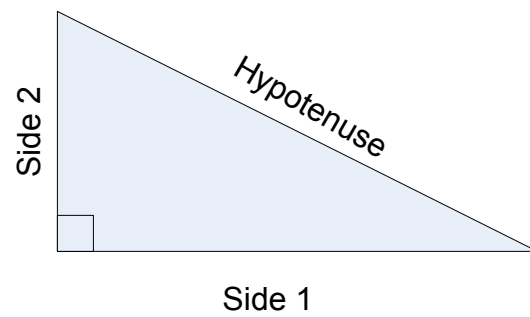


Figure 6.4: Right triangle

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 be 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 will not consider them here.)
- **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.

The simplest function accepts no parameters and returns no value to the caller. Listing 7.1 (simplefunction.py) is a variation of Listing 3.1 (adder.py). In Listing 7.1 (simplefunction.py), the `def` keyword marks the beginning of the prompt function definition.

Listing 7.1: simplefunction.py

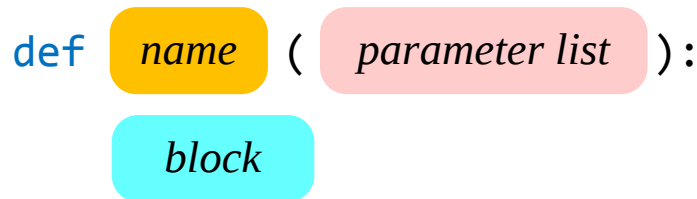


Figure 7.1: General Form of a function definition

```
# Print a message to prompt the user for input
def prompt():
    print("Please enter an integer value: ", end="")

# Start of program
print("This program adds two integers.")
prompt()    # Call the function
value1 = int(input())
prompt()    # Call the function again
value2 = int(input())
sum = value1 + value2
print(value1, "+", value2, "=", sum)
```

The two lines

```
def prompt():
    print("Please enter an integer value: ", end="")
```

constitute the definition of the `prompt` function. The function's name is `prompt`, it has an empty parameter list, and the block that makes up its body consists of just one statement. When invoked, the function simply prints the message *Please enter an integer value:* and leaves the cursor on the same line. The program runs as follows:

1. The program's execution begins with the first line in the "naked" block; that is, the block that is not part of the function definition. The program thus first prints the message *This program adds two integers*.
2. The next statement is a call of the `prompt` function. At this point the program's execution transfers to the body of the `prompt` function. The code within `prompt` is executed until the end of its body. It simply prints the message *Please enter an integer value:*.
3. When `prompt` is finished, control is passed back to the point in the code immediately *after* the call of `prompt`.
4. The executing program next reads the value of `value1` from the keyboard.
5. A second call to `prompt` transfers control back to the code within the `prompt` function. It again prints its message.
6. When the second call to `prompt` finishes, control passes back to the point of the second input statement that assigns `value2` from the keyboard.

7. The program finally executes the remaining two statements in the code, the arithmetic and printing statements.
8. With all of the statements in its block executed, the program terminates.

Figure 7.2 contains a diagram illustrating the execution of Listing 7.1 (simplefunction.py) as control passes amongst the various functions. The interaction amongst functions is quite elaborate, even for such a simple

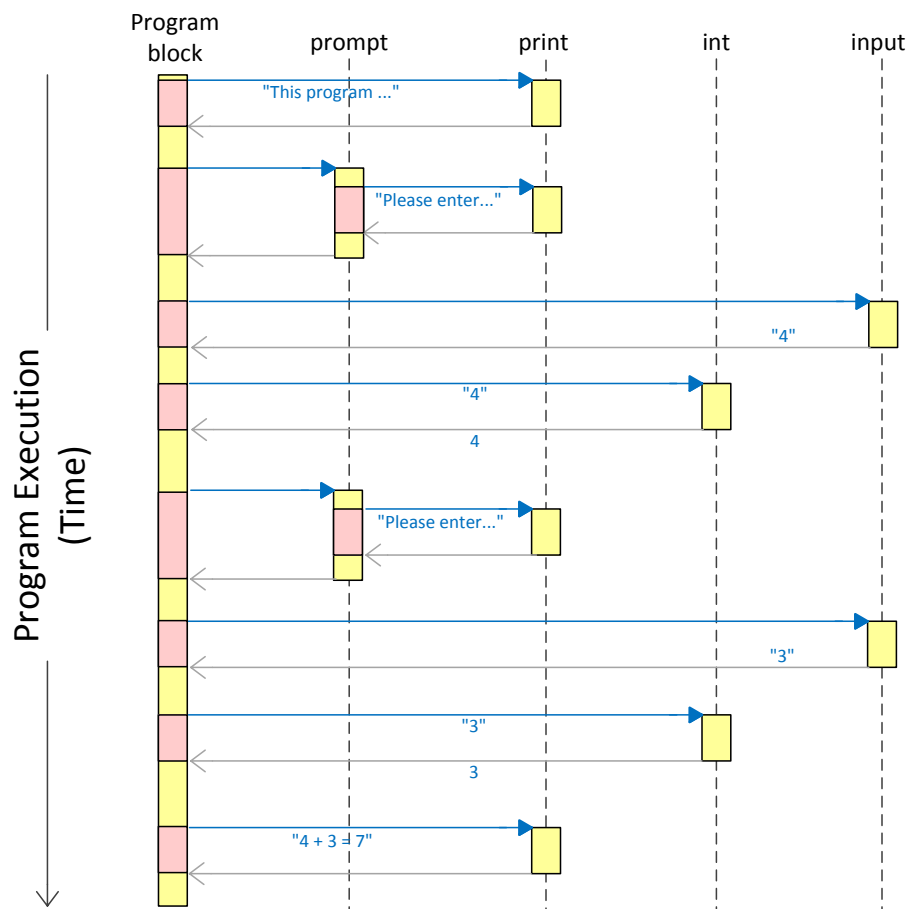


Figure 7.2: Calling relationships among functions during the execution of Listing 7.1 (simplefunction.py). Time flows from top to bottom. A vertical bar represents the time in which a block of code is active. Observe that functions are active only during their call. The shaded area within in block represents the time that block is idle, waiting for a function call to complete. Right arrows ($\rightarrow$) represent function calls. Function calls show parameters, where applicable. Left arrows ($\leftarrow$) represent function returns. Function returns show return values, if applicable.

program.

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 on its own line
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
```

Our prompt and countto10 functions are a bit underwhelming. The prompt function could be eliminated, and each call to prompt could be replaced with the statement in its body. The same could be said for the countto10 function, although it is convenient to have the simple one-line statement that hides the complexity of the loop. Using the prompt function does have one advantage, though. If we remove the prompt function and replace the two calls to prompt with the print statement within prompt, we have to make sure that the two messages printed are identical. If we simply call prompt, we know the two messages printed will be identical.

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. 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 `countto10` function does us little good if we sometimes want to count up to a different number. Listing 7.4 (`countton.py`) generalizes Listing 7.3 (`countto10func.py`) to count as high as the caller needs.

Listing 7.4: `countton.py`

```
# Count to n and print each number on its own line
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 `countton(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
```

We can enhance the prompt function's capabilities as shown in Listing 7.6 (`betterprompt.py`)

Listing 7.6: `betterprompt.py`

```
# Definition of the prompt function
def prompt():
    value = int(input("Please enter an integer value: "))
    return value

print("This program adds together two integers.")
value1 = prompt()    # Call the function
value2 = prompt()    # Call the function again
sum = value1 + value2
print(value1, "+", value2, "=", sum)
```

In this version, `prompt` takes care of the input, so the calling code itself does not have to call the `input` and `int` functions. The assignment statement

```
value1 = prompt()
```

implies `prompt` now produces a result we can assign to a variable or use in some other way. The last statement in the `prompt` function's definition is a `return` statement. A `return` statement specifies the exact result to return to the caller. When a function's execution encounters a `return` statement, control immediately passes back to the caller. The value of the function call is the value specified by the `return` statement, so the statement

```
value1 = prompt()
```

assigns to the variable `value1` the quantity associated with the `return` statement during `prompt`'s execution.

Note that in Listing 7.6 (`betterprompt.py`), we used a variable named `value` inside the `prompt` function. This variable is *local* to the function, meaning we cannot use this particular variable outside of `prompt`. It also means we are free to use that same name outside of the `prompt` function in a different context, and doing so will not interfere with the `value` variable within `prompt`. We say that `value` is a *local variable*.

We can further enhance our `prompt` function. Currently `prompt` always prints the same message. Using parameters, we can customize the message that `prompt` prints. The `prompt` function in Listing 7.7 (`evenbetterprompt.py`) uses parameters to provide a customized message within `prompt`.

Listing 7.7: evenbetterprompt.py

```
# Definition of the prompt function
def prompt(n):
    value = int(input("Please enter integer #", n, ": ", sep=""))
    return value

print("This program adds together two integers.")
value1 = prompt(1)    # Call the function
value2 = prompt(2)    # Call the function again
sum = value1 + value2
print(value1, "+", value2, "=", sum)
```

In Listing 7.7 (`evenbetterprompt.py`), the parameter influences the message that the prompt function prints. Now the function prompts the user to enter value #1 or value #2. The call

```
value1 = prompt(1)
```

passes the integer 1 to the prompt function. This process binds the actual parameter 1 to the function's formal parameter `n`. The process works as if the prompt function contained the assignment statement

```
n = 1
```

as its first statement.

To recap, in the first line of the function definition:

```
def prompt(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 `prompt`, such as `prompt(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. A function ordinarily passes back to the caller one piece of information via a `return` statement, but a function may return multiple pieces of information packed up in a tuple or other data structure. Listing 7.8 (`midpoint.py`) uses a custom function to compute the midpoint between two mathematical points.

Listing 7.8: 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 = eval(input("Enter first point (x, y): "))
point2 = eval(input("Enter second point (x, y): "))
# Compute the midpoint
mid = midpoint(point1, point2)
# Report result to user
print('Midpoint of', point1, 'and', point2, 'is', mid)
```

Listing 7.8 (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 7.8 (midpoint.py) looks like the following:

```
Enter first point (x, y): (0, 0)
Enter second point (x, y): (1, 1)
Midpoint of (0, 0) and (1, 1) 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 7.8 (midpoint.py) refers to a single tuple object. We will examine tuples in more detail in Chapter 12, 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*.

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, 6, 8, 12, 24
 18: 1, 2, 3, 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 so we divide the numerator and the denominator of the fraction 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 $\text{GCF}(24, 18) = 6$, we can cut the plywood into twelve 6 inch $\times$ 6 inch square pieces as shown in Figure 7.3. If we cut the plywood into squares of any other size without wasting 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 7.4. If we cut squares larger than 6 inches $\times$ 6 inches, not all the plywood can be used to make the squares. Figure 7.5. shows 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.

The following code defines a function that computes the greatest common divisor of two integers. It determines 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 is definitely a common factor to all ints
    largestFactor = 1
    for i in range(1, min + 1):
```

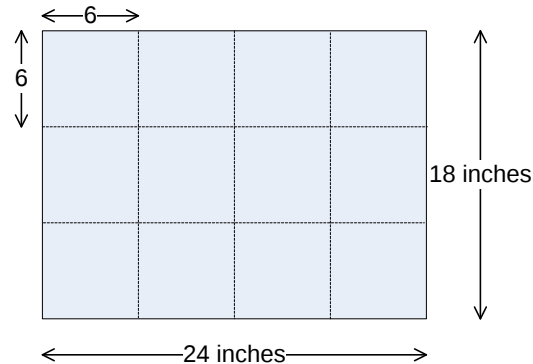


Figure 7.3: Cutting plywood

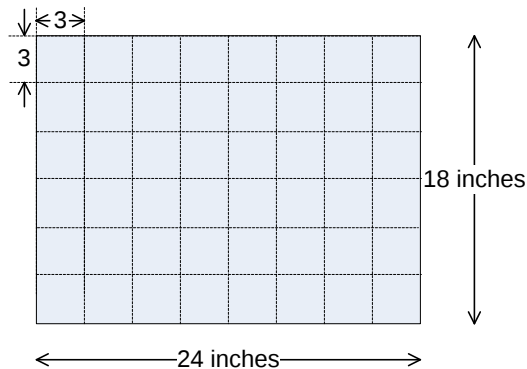


Figure 7.4: Squares too small

```

if num1 % i == 0 and num2 % i == 0:
    largestFactor = i    # Found larger factor
return largestFactor

```

This function is named `gcd` and expects two integer arguments. Its formal parameters are named `num1` and `num2`. It returns an integer result. The function uses three local variables: `min`, `largestFactor`, and `i`. Local variables have meaning only within their scope. The scope of a local variable is the point within the function's block after its 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 is a local variable, you can reuse the names of formal parameters in different functions without a problem.

Another advantage of local variables is that they occupy space in the computer's memory only when the function is executing. The run-time environment allocates space in the computer's memory for local

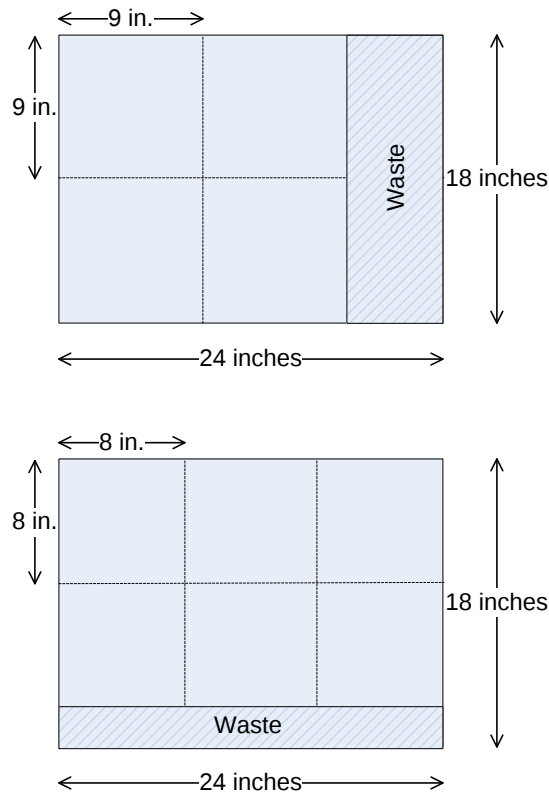


Figure 7.5: Squares too large

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 calls the function.

Once a function has been defined, callers can use it. A programmer-defined function is invoked in exactly the same way as a standard library function like `sqrt` (6.2) or `randrange` (6.4). If the function returns a value, then its invocation can be used anywhere an expression of that type can be used. The function `gcd` can be called as part of an assignment statement:

```
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 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*.

7.2 Main Function

Functions help us organize our code. It is common for Python programmers to use a function named `main` to hold the statements that to this point we have not placed within a function. Listing 7.9 (`gcdwithmain.py`) illustrates the typical Python code organization.

Listing 7.9: `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
    largestFactor = 1
    for i in range(1, min + 1):
        if m % i == 0 and n % i == 0:
            largestFactor = i # Found larger factor
    return largestFactor

# 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 7.6 contains a diagram that shows the calling relationships among the function executions during a run of Listing 7.9 (`gcdwithmain.py`).

7.3 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 7.10 (`parampassing.py`) illustrates the consequences of passing an immutable type to a function.

Listing 7.10: `parampassing.py`

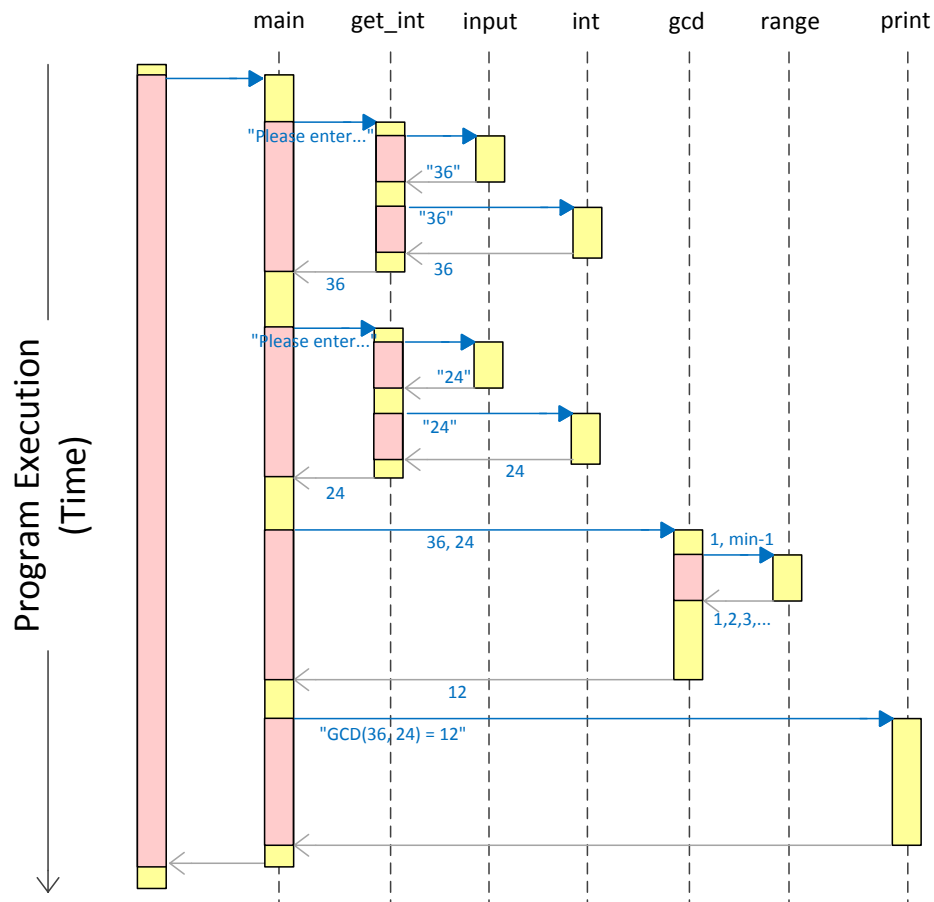


Figure 7.6: Calling relationships among functions during the execution of Listing 7.9 (gcdwithmain.py)

```
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 7.10 (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.

7.4 Function Examples

This section contains a number of examples of code organization with functions.

7.4.1 Better Organized Prime Generator

Listing 7.11 (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 7.11: primefunc.py

```
from math import sqrt

# is_prime(n)
#   Determines the primality of a given value
#   n an integer to test for primality
#   Returns true if n is prime; otherwise, returns false
def is_prime(n):
    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

# main
#   Tests for primality each integer from 2
#   up to a value provided by the user.
#   If an integer is prime, it prints it;
#   otherwise, the number is not printed.
```

```
def main():
    max_value = int(input("Display primes up to what value? "))
    for value in range(2, max_value + 1):
        if is_prime(value):          # See if value is prime
            print(value, end=" ")    # Display the prime number
    print()                          # Move cursor down to next line

main()    # Run the program
```

Listing 7.11 (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 7.12 (`calculator.py`) uses such a function.

Listing 7.12: `calculator.py`

```

# help_screen
#     Displays information about how the program works
#     Accepts no parameters
#     Returns nothing
def help_screen():
    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")

# menu
#     Display a menu
#     Accepts no parameters
#     Returns the string entered by the user.
def menu():
    # Display a menu
    return input("=== A)dd S)ubtract P)rint H)elp Q)uit ===")

# main
#     Runs a command loop that allows users to
#     perform simple arithmetic.
def main():
    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 5.37 (`betterinputonly.py`) forces the user to enter a value within a specified range. We now can easily adapt that concept to a function. Listing 7.13 (`betterinputfunc.py`) uses a function named `get_int_in_range` that does not return until the user supplies a proper value.

Listing 7.13: `betterinputfunc.py`

```

# 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
def get_int_in_range(first, last):
    # 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

# main
#   Tests the get_int_in_range function
def main():
    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 7.13 (`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 7.13 (`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 7.14 (`betterdie.py`) reorganizes Listing 6.11 (`die.py`) into functions.

Listing 7.14: `betterdie.py`

```
from random import randrange

# show_die(spots)
#     Draws a picture of a die with number of spots
#     indicated spots is the number of spots on the top face
def show_die(spots):
    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("+-----+")

# roll
# Returns a pseudorandom number in the range 1...6, inclusive
def roll():
    return randrange(1, 7)

# main
# Simulates the roll of a die three times
def main():
    # Roll the die three times
    for i in range(0, 3):
        show_die(roll())

main() # Run the program

```

In Listing 7.14 (`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 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 7.15 (treefunc.py) reorganizes Listing 5.32 (startree.py) into functions.

Listing 7.15: treefunc.py

```
# tree(height)
#   Draws a tree of a given height
#   height is the height of the displayed tree
def tree(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

# main
#   Allows users to draw trees of various heights
def main():
    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 7.16 (`simplefloataddition.py`), which adds two double-precision floating-point numbers and checks for a given value.

Listing 7.16: `simplefloataddition.py`

```
def main():
    x = 0.9
    x += 0.1
    if x == 1.0:
        print("OK")
    else:
        print("NOT OK")

main()
```

Listing 7.16 (`simplefloataddition.py`) reports

```
OK
```

All seems well judging from the behavior of Listing 7.16 (`simplefloataddition.py`). Next, consider Listing 7.17 (`badfloatcheck.py`) which attempts to control a loop with a double-precision floating-point number.

Listing 7.17: `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 7.17 (`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 7.16 (`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 7.17 (`badfloatcheck.py`) demonstrates that the `==` and `!=` operators are of questionable worth when comparing floating-point values. The better approach is to check to see if two floating-point values are *close enough*, which means they differ by only a very small amount. When comparing two floating-point numbers x and y , we essentially must determine if the absolute value of their difference is small; for example, $|x - y| < 0.00001$. We can construct an `equals` function and incorporate the `fabs` function introduced in 6.2. Listing 7.18 (`floatequalsfunction.py`) provides such an `equals` function.

Listing 7.18: `floatequalsfunction.py`

```

from math import fabs

# equals(a, b, tolerance)
#   Returns true if a = b or |a - b| < tolerance.
#   If a and b differ by only a small amount
#   (specified by tolerance), a and b are considered
#   "equal." Useful to account for floating-point
#   round-off error.
#   The == operator is checked first since some special
#   floating-point values such as floating-point infinity
#   require an exact equality check.
def equals(a, b, tolerance):
    return a == b or fabs(a - b) < tolerance

# Try out the equals function
def main():
    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 Custom Functions vs. Standard Functions

Recall the custom square root code we saw in Listing 5.31 (`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 7.19 (`customsquareroot.py`) contains the definition of our custom `square_root` function.

Listing 7.19: `customsquareroot.py`

```
# File customsquareroot.py

# Compute an approximation of the square root of x
def square_root(val):
    # 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 7.19 (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 7.20 (squarerootcomparison.py) tests our custom square root function over a range of 10,000,000 floating point values.

```

from math import fabs, sqrt

# Consider two floating-point numbers equal when
# the difference between them is very small.
# equals(a, b, tolerance)
#     Returns true if  $a = b$  or  $|a - b| < \text{tolerance}$ .
#     If a and b differ by only a small amount
#     (specified by tolerance), a and b are considered
#     "equal." Useful to account for floating-point
#     round-off error.
#     The == operator is checked first since some special
#     floating-point values such as floating-point infinity
#     require an exact equality check.
def equals(a, b, tolerance):
    return a == b or fabs(a - b) < tolerance

# Computes the approximate square root of val
# val is an number
def square_root(val):
    # 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

```

```
0.0 : Expected 0.0 but computed 6.103515625e-05
0.00060000000000000001 : 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.

7.6 Summary

- The development of larger, more complex programs is more manageable when the program consists of multiple programmer-defined functions.
- Every function has one definition but can have many invocations.
- A function definition includes the function's name, parameters, and body.
- A function name, like a variable name, is an identifier.
- Formal parameters are the parameters as they appear in a function's definition; actual parameters are the arguments supplied by the caller.
- Formal parameters essentially are variables local to the function; actual parameters passed by the caller may be variables, expressions, or literal values.
- A function invocation binds the actual parameters to the formal parameters.
- Clients must pass to functions the number of parameters specified in the function definition. The types of the actual parameters must be compatible with the ways the formal parameters are used within the function definition.
- If the formal parameter is bound to an immutable type like a number or string, the function cannot affect the caller's actual parameter.
- Variables defined within a function are local to that function definition. Local variables cannot be seen by code outside the function definition.
- During a program's execution, local variables live only when the function is executing. When a particular function call is finished, the space allocated for its local variables is freed up.

7.7 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):  
    print(x + 2)  
  
def main():  
    proc(5)  
  
main()
```

5. Is the following a legal Python program?

```
def proc(x, y):  
    return 2*x + y*y  
  
def main():  
    print(proc(5, 4))  
  
main()
```

6. Is the following a legal Python program?

```
def proc(x, y):  
    return 2*x + y*y  
  
def main():  
    print(proc(5))  
  
main()
```

7. Is the following a legal Python program?

```
def proc(x):  
    return 2*x  
  
def main():  
    print(proc(5, 4))  
  
main()
```

8. Is the following a legal Python program?

```
def proc(x):  
    print(2*x*x)  
  
def main():  
    proc(5)  
  
main()
```

9. The programmer was expecting the following program to print 200. What does it print instead? Why does it print what it does?

```
def proc(x):  
    x = 2*x*x  
  
def main():  
    num = 10  
    proc(num)  
    print(num)  
  
main()
```

10. Is the following program legal since the variable `x` is used in two different places (`proc` and `main`)? Why or why not?

```
def proc(x):  
    return 2*x*x  
  
def main():  
    x = 10  
    print(proc(x))  
  
main()
```

11. Is the following program legal since the actual parameter has a different name from the formal parameter (`y` vs. `x`)? Why or why not?

```
def proc(x):  
    return 2*x*x  
  
def main():  
    y = 10  
    print(proc(y))
```

```
main()
```

12. Complete the following distance function that computes the distance between two geometric points (x_1, y_1) and (x_2, y_2) :

```
def distance(x1, y1, x2, y2):  
    ...
```

Test it with several points to convince yourself that is correct.

13. What happens if a caller passes too many parameters to a function?
14. What happens if a caller passes too few parameters to a function?
15. What are the rules for naming a function in Python?
16. Consider the following function definitions:

```
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()))
```


Chapter 8

More on Functions

This chapter covers some additional aspects of functions in Python. It introduces recursion, a key concept in computer science.

8.1 Global Variables

Variables defined within functions are local variables. Local variables have some very desirable properties:

- The memory required to store a local variable is used only when the variable is in scope; that is, the variable exists only during the function's execution. When the program's execution leaves the scope of a local variable, the memory for that variable is freed up. This memory then is reused for the local variables of other functions as needed.
- The same variable name can be used in different functions without any conflict. The interpreter derives all of its information about a local variable from that variable's definition within the function. If the interpreter attempts to execute a statement that uses a variable that has not been defined, the interpreter issues a run-time error. When executing code in one function the interpreter will not look for a variable definition in another function. Thus, there is no way a local variable in one function can interfere with a local variable defined in another function.

A local variable is transitory, so it disappears in between function invocations. Sometimes it is desirable to have a variable that exists independent of any function executions. In contrast to a local variable, a *global variable* lives outside of all functions and is not local to any particular function. This means any function is capable of accessing and/or modifying a global variable.

A variable is defined when it is assigned to an object. Any variable assigned within a function is local to that function, unless the variable is declared to be a global variable using the `global` reserved word. Listing 8.1 (`globalcalculator.py`) is a modification of Listing 7.12 (`calculator.py`) that uses a global variables named `result`, `arg1`, and `arg2` that are shared by several functions in the program.

Listing 8.1: `globalcalculator.py`

```
# help_screen
# Displays information about how the program works
# Accepts no parameters
```

```
# Returns nothing
def help_screen():
    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")

# menu
# Display a menu
# Accepts no parameters
# Returns the string entered by the user.
def menu():
    # Display a menu
    return input("=== A)dd S)ubtract P)rint H)elp Q)uit ===")

# Global variables used by several functions
result = 0.0
arg1 = 0.0
arg2 = 0.0

# get_input
# Assigns the globals arg1 and arg2 from user keyboard
# input
def get_input():
    global arg1, arg2 # arg1 and arg2 are globals
    arg1 = float(input("Enter argument #1: "))
    arg2 = float(input("Enter argument #2: "))

# report
# Reports the value of the global result
def report():
    # Not assigning to result, global keyword not needed
    print(result)

# add
# Assigns the sum of the globals arg1 and arg2
# to the global variable result
def add():
    global result # Assigning to result, global keyword needed
    result = arg1 + arg2

# subtract
# Assigns the difference of the globals arg1 and arg2
# to the global variable result
def subtract():
    global result # Assigning to result, global keyword needed
    result = arg1 - arg2

# main
# Runs a command loop that allows users to
# perform simple arithmetic.
def main():
    done = False # Initially not done
```

```

while not done:
    choice = menu()      # Get user's choice

    if choice == "A" or choice == "a":  # Addition
        get_input()
        add()
        report()
    elif choice == "S" or choice == "s": # Subtraction
        get_input()
        subtract()
        report()
    elif choice == "P" or choice == "p": # Print
        report()
    elif choice == "H" or choice == "h": # Help
        help_screen()
    elif choice == "Q" or choice == "q": # Quit
        done = True

main()

```

Listing 8.1 (globalcalculator.py) uses global variables `result`, `arg1`, and `arg2`. These names no longer appear in the main function. The program accesses and/or modifies these global variables in four different functions: `get_input`, `report`, `add`, and `subtract`. The `global` keyword within a function's block of code identifies the variables which are global variables. Notice that if a function uses a global variable without assigning its value, the `global` declaration is not necessary. This is because variable assignment is variable definition, and a local variable must be defined within a function.



A function may use a global variable without declaring it with the `global` keyword if the function does not assign a variable of that name anywhere in its body. A function that assigns a global variable must declare that variable as global with the `global` keyword.

If a function defines a local variable with the same name as a global variable, the global variable becomes inaccessible to code within the function. We say the local variable *hides* the like-named global variable from code in the function's body.

When is it acceptable to use global variables, and when is it better to use local variables? In general, local variables are preferable to global variables for several reasons:

- When a function uses local variables exclusively and performs no other input operations (like calling the input function), only parameters passed in by the caller can influence the function's behavior. If a global variable appears in a function, the function's behavior potentially is affected by every other function that can modify that global variable. As a simple example, consider the following trivial function that appears in a program:

```

def increment(n):
    return n + 1

```

Can you predict what the following statement within that program will print?

```
print(increment(12))
```

If your guess is 13, you are correct. The increment function simply returns the result of adding one to its argument. The increment function behaves the same way each time it is called with the same argument.

Next, consider the following three functions that appear in some program:

```
def process(n):
    return n + m    # m is a global integer variable

def assign_m():
    global m
    m = 5

def inc_m():
    global m
    m += 1
```

Can you predict what the following statement within the program will print?

```
print(process(12))
```

We cannot predict what this statement in isolation will print. The following scenarios all produce different results:

```
assign_m()
print(process(12))

prints 17,

m = 10    # m is the global
print(process(12))
```

```
prints 22,

m = 0    # m is the global
inc_m()
inc_m()
print(process(12))
```

```
prints 14, and

assign_m()
inc_m()
inc_m()
print(process(12))
```

prints 19. The identical printing statements print different values depending on the cumulative effects of the program's execution up to that point.

It may be difficult to locate an error if a function that uses a global variable fails because it may be the fault of *another* function that assigned an incorrect value to the global variable. The situation may be more complicated than the one portrayed in the simple examples above; consider:

```
assign_m()
.
.    # 30 statements in between, some of which may change a,
.    # b, and m
```

```

.
if a < 2 and b <= 10:
    m = a + b - 100
.
.   # 20 statements in between, some of which may change m
.
print(process(12))

```

- A nontrivial program that uses global variables will be more difficult for a human reader to understand than one that does not. When examining the contents of a function, a global variable requires the reader to look elsewhere (outside the function) for its meaning:

```

# Linear function
def f(x):
    return m*x + b

```

What are *m* and *b*? How, where, and when are they assigned or re-assigned?

- A function that uses only local variables can be tested for correctness in isolation from other functions, since other functions do not affect the behavior of this function. This function's behavior is only influenced only by its parameters, if it has any.

The exclusion of global variables from a function leads to *functional independence*. A function that depends on information outside of its scope to correctly perform its task is a *dependent function*. When a function operates on a global variable it depends on that global variable being in the correct state for the function to complete its task correctly. Nontrivial programs that contain many dependent functions are more difficult debug and extend. A truly independent function that use no global variables and uses no programmer-defined functions to help it out can be tested for correctness in isolation. Additionally, an independent function can be copied from one program, pasted into another program, and work without modification. Functional independence is a desirable quality.

The exclusion of global variables from a function's definition does not guarantee that the function always will produce the same results given the same parameter values; consider

```

def compute(n):
    favorite = eval(input("Please enter your favorite number: "))
    return n + favorite

```

The `compute` function avoids global variables, yet we cannot predict the value of the expression `compute(12)`. Recall the `increment` function from above:

```

def increment(n):
    return n + 1

```

Its behavior is totally predictable. Furthermore, `increment` does not modify any global variables, meaning its code all by itself cannot in any way influence the overall program's behavior. We say that `increment` is a *pure function*. A pure function cannot perform any input or output (for example, use the `print` or `input` statements), nor may it use global variables. While `increment` is pure, the `compute` function is impure. The following function is impure also, since it performs output:

```

def increment_and_report(n):
    print("Incrementing", n)
    return n + 1

```

A pure function simply computes its return value and has no other observable side effects.

A function that calls only other pure functions and otherwise would be considered pure is itself a pure function; for example:

```
def double_increment(n):  
    return increment(n) + 1
```

`double_increment` is a pure function since `increment` is pure; however, `double_increment_with_report`:

```
def double_increment_with_report(n):  
    return increment_and_report(n) + 1
```

is not a pure function since it calls `increment_and_report` which is impure.

8.2 Default Parameters

We have seen how callers may invoke some Python functions with differing numbers of parameters. Compare

```
a = input()
```

to

```
a = input("Enter your name: ")
```

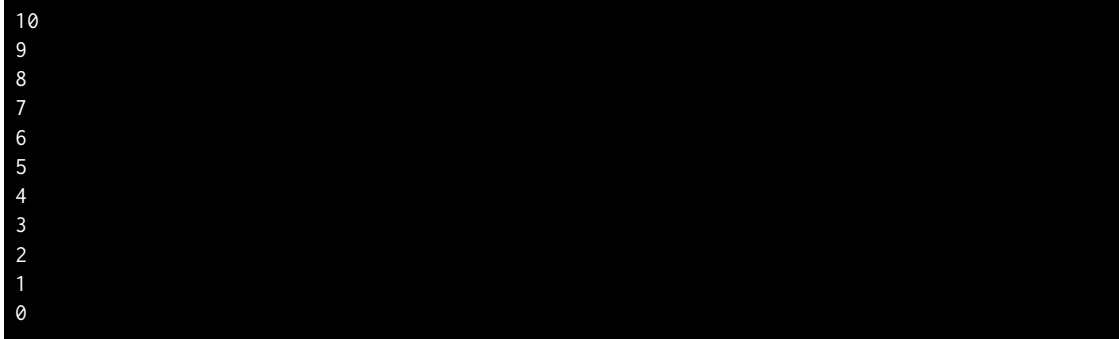
We can define our own functions that accept a varying number of parameters by using a technique known as *default parameters*. Consider the following function that counts down:

```
def countdown(n=10):  
    for count in range(n, -1, -1): # Count down from n to zero  
        print(count)
```

The formal parameter expressed as `n=10` represents a default parameter or default argument. If the caller does not supply an actual parameter, the formal parameter `n` is assigned 10. The following call

```
countdown()
```

prints



```
10  
9  
8  
7  
6  
5  
4  
3  
2  
1  
0
```

but the invocation

```
countdown(5)
```

displays

```
5
4
3
2
1
0
```

As we can see, when the caller does not supply a parameter specified by a function, and that parameter has a default value, the default value is used during the caller's call.

We may mix non-default and default parameters in the parameter lists of a function declaration, but all default parameters within the parameter list must appear after all the non-default parameters. This means the following definitions are acceptable:

```
def sum_range(n, m=100):    # OK, default follows non-default
    sum = 0
    for val in range(n, m + 1):
        sum += val
```

and

```
def sum_range(n=0, m=100):  # OK, both default
    sum = 0
    for val in range(n, m + 1):
        sum += val
```

but the following definition is illegal, since a default parameter precedes a non-default parameter in the function's parameter list:

```
def sum_range(n=0, m):      # Illegal, non-default follows default
    sum = 0
    for val in range(n, m + 1):
        sum += val
```

8.3 Introduction to Recursion

The *factorial* function is widely used in combinatorial analysis (counting theory in mathematics), probability theory, and statistics. The factorial of n usually is expressed as $n!$. Factorial is defined for non-negative integers as

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdots 3 \cdot 2 \cdot 1$$

and $0!$ is defined to be 1. Thus $6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720$. Mathematicians precisely define factorial in this way:

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n \cdot (n-1)!, & \text{otherwise.} \end{cases}$$

This definition is *recursive* since the $!$ function is being defined, but $!$ is used also in the definition. A Python function can be defined recursively as well. Listing 8.2 (factorialtest.py) includes a factorial function that exactly models the mathematical definition.

Listing 8.2: factorialtest.py

```

# factorial(n)
#   Computes n!
#   Returns the factorial of n.
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

def main():
    # Try out the factorial function
    print(" 0! = ", factorial(0))
    print(" 1! = ", factorial(1))
    print(" 6! = ", factorial(6))
    print("10! = ", factorial(10))

main()

```

Listing 8.2 (factorialtest.py) produces

```

0! = 1
1! = 1
6! = 720
10! = 3628800

```

Observe that the factorial function in Listing 8.2 (factorialtest.py) uses no loop to compute its result. The factorial function simply calls itself. The call `factorial(6)` is computed as follows:

```

factorial(6) = 6 * factorial(5)
              = 6 * 5 * factorial(4)
              = 6 * 5 * 4 * factorial(3)
              = 6 * 5 * 4 * 3 * factorial(2)
              = 6 * 5 * 4 * 3 * 2 * factorial(1)
              = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
              = 6 * 5 * 4 * 3 * 2 * 1 * 1
              = 6 * 5 * 4 * 3 * 2 * 1
              = 6 * 5 * 4 * 3 * 2
              = 6 * 5 * 4 * 3
              = 6 * 5 * 4 * 6
              = 6 * 5 * 24
              = 6 * 120
              = 720

```

Note that we can optimize the factorial function slightly by changing the `if`'s condition from `n == 0` to `n < 2`. This change results in a function execution trace that eliminates one function call at the end:

```

factorial(6) = 6 * factorial(5)
              = 6 * 5 * factorial(4)
              = 6 * 5 * 4 * factorial(3)
              = 6 * 5 * 4 * 3 * factorial(2)
              = 6 * 5 * 4 * 3 * 2 * factorial(1)
              = 6 * 5 * 4 * 3 * 2 * 1

```

$$\begin{aligned}
 &= 6 * 5 * 4 * 3 * 2 \\
 &= 6 * 5 * 4 * 6 \\
 &= 6 * 5 * 24 \\
 &= 6 * 120 \\
 &= 720
 \end{aligned}$$

Figure 8.1 illustrates the chain of recursive factorial invocations when executing the statement `print(factorial(6))`.

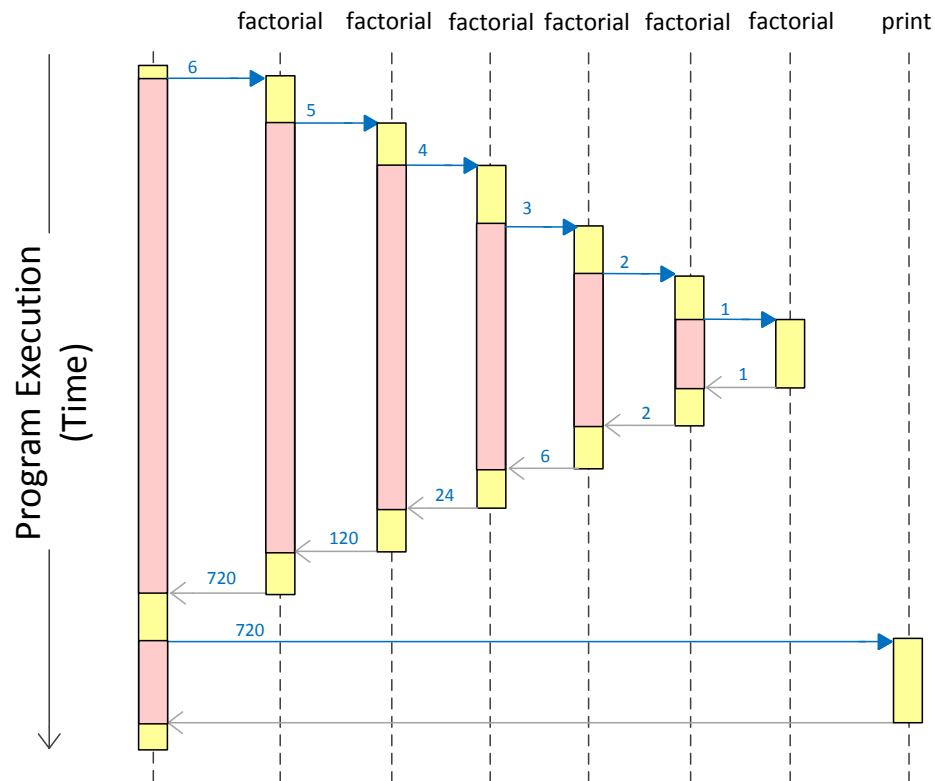


Figure 8.1: A graphical representation of the chain of recursive factorial invocations when executing the statement `print(factorial(6))`, where the `factorial` function is from Listing 8.2 (`factorialtest.py`) with the condition optimized to `n < 2`. The vertical bars represent the time a function invocation is active. The shaded area within each bar represents the time that the function, while still active, is idle, waiting for a function it calls to complete. Note that during the process of recursion all earlier function invocations in the call chain remain active (but idle) until all the functions further down the call chain return.

A correct simple recursive function definition is based on four key concepts:

1. The function optionally must call itself within its definition; this is the *recursive case*.

2. The function optionally must *not* call itself within its definition; this is the *base case*.
3. Some sort of conditional execution (such as an `if/else` statement) selects between the recursive case and the base case based on one or more parameters passed to the function.
4. Each invocation that does correspond to the base case must call itself with parameter(s) that move the execution closer to the base case. The function's recursive execution must converge to the base case.

Each recursive invocation must bring the function's execution closer to its base case. The factorial function calls itself in the `else` block of the `if/else` statement. Its base case is executed if the condition of the `if` statement is true. Since the factorial is defined only for non-negative integers, the initial invocation of factorial must be passed a value of zero or greater. A zero parameter (the base case) results in no recursive call. Any other positive parameter results in a recursive call with a parameter that is closer to zero than the one before. The nature of the recursive process progresses towards the base case, upon which the recursion terminates.

Recursion is not our only option when computing a factorial. Listing 8.3 (`nonrecursfact.py`) provides a non-recursive factorial function.

Listing 8.3: `nonrecursfact.py`

```
# factorial(n)
#   Computes n!
#   Returns the factorial of n.
def factorial(n):
    product = 1
    while n:
        product *= n
        n -= 1
    return product

def main():
    # Try out the factorial function
    print(" 0! = ", factorial(0))
    print(" 1! = ", factorial(1))
    print(" 6! = ", factorial(6))
    print("10! = ", factorial(10))

main()
```

Which factorial function is better, the recursive or non-recursive version? Generally, if both the recursive and non-recursive functions implement the same basic algorithm, the non-recursive function will be more efficient. A function call is a relatively expensive operation compared to a variable assignment or comparison. The body of the non-recursive factorial function invokes no functions, but the recursive version calls a function—it calls itself—during all but the last recursive invocation. The iterative version of factorial is therefore more efficient than the recursive version.

Even though the iterative version of the factorial function is technically more efficient than the recursive version, on most systems you could not tell the difference. The reason is the factorial function “grows” fast, meaning it returns fairly large results for relatively small arguments.

Recall the `gcd` function from Section 7.4. It computed the greatest common divisor (also known as greatest common factor) of two integer values. It works, but it is not very efficient. Listing 8.4 (`gcd.py`) uses a better algorithm. It is based on one of the oldest algorithms known, attributed to Euclid around 300 B.C.

Listing 8.4: gcd.py

```
# gcd(m, n)
#     Uses Euclid's method to compute
#     the greatest common divisor
#     (also called greatest common
#     factor) of m and n.
#     Returns the GCD of m and n.
def gcd(m, n):
    if n == 0:
        return m
    else:
        return gcd(n, m % n)

def iterative_gcd(num1, num2):
    # Determine the smaller of num1 and num2
    min = num1 if num1 < num2 else num2
    # 1 is definitely a common factor to all integers
    largestFactor = 1;
    for i in range(1, min + 1):
        if num1 % i == 0 and num2 % i == 0:
            largestFactor = i # Found larger factor
    return largestFactor

def main():
    # Try out the gcd function
    for num1 in range(1, 101):
        for num2 in range(1, 101):
            print("gcd of", num1, "and", num2, "is", gcd(num1, num2))

main()
```

Note that this gcd function is recursive. The algorithm it uses is much different from our original iterative version. Because of the difference in the algorithms, this recursive version is actually much more efficient than our original iterative version. A recursive function, therefore, cannot be dismissed as inefficient just because it is recursive. We will revisit recursion in Section 11.5.

8.4 Making Functions Reusable

In a function definition we can package functionality that can be used in many different places within a program. We have yet to see how we can easily reuse our function definitions in other *programs*. For example, our `is_prime` function in Listing 7.11 (`primefunc.py`) works well within Listing 7.11 (`primefunc.py`), and we could put it to good use in other programs that need to test primality (encryption software, for example, makes heavy use of prime numbers). We could use the copy-and-paste feature of our favorite text editor to copy the `is_prime` function definition from Listing 7.11 (`primefunc.py`) into the new encryption program we are developing. It is possible to reuse a function in this way only if the function definition does not use any programmer-defined global variables nor any other programmer-defined functions. If a function does use any of these programmer-defined external entities, we must include these dependencies as well in the new code for the function to be viable. Said another way, the code in the function definition ideally will use only local variables and parameters. Such a function truly is independent and easily transportable to other programs.

The notion of copying source code from one program to another is not ideal, however. It is too easy for the copy to be incomplete or otherwise incorrect. Furthermore, such code duplication is wasteful. If 100 programs on a particular system all need to use the `is_prime` function, under this scheme they must all include the `is_prime` code. This redundancy wastes space. Finally, in perhaps the most compelling demonstration of the weakness of this copy-and-paste approach, what if a bug is discovered in the `is_prime` function that all 100 programs are built around? When the error is discovered and fixed in one program, the other 99 programs will still contain the bug. Their source code must be updated, and it may be difficult to determine which files need to be fixed. The problem becomes much worse if the code has been released to the general public. It may be impossible to track down and correct all the copies of the faulty function. The situation would be the same if a correct `is_prime` function were updated to be made more efficient. The problem is this: all the programs using `is_prime` define their *own* `is_prime` function; while the function definitions are meant to be identical, there is no mechanism tying all these common definitions together. We really would like to reuse the function as is without copying it.

Fortunately, Python makes it easy for developers to package their functions into *modules*. Programmers can build modules independently from the programs that use them. Software engineers did exactly that when developing Python's standard modules. They did so without the foreknowledge of exactly how we would use the functions they provide.

A Python source file constitutes a module. Consider the module Listing 8.5 (`primecode.py`).

Listing 8.5: `primecode.py`

```
# Contains the definition of the is_prime function
from math import sqrt

# Returns True if non-negative integer n is prime;
# otherwise, returns false
def is_prime(n):
    trial_factor = 2
    root = sqrt(n)

    while trial_factor <= root:
        if n % trial_factor == 0:      # Is trial factor a factor?
            return False             # Yes, return right away
        trial_factor += 1             # Consider next potential factor

    return True                       # Tried them all, must be prime
```

Other Python programs can use the code within the Listing 8.5 (`primecode.py`) file. In the simplest case, this module (file) appears in the same directory (folder) as the calling code file that uses it. Listing 8.6 (`usingprimecode.py`) contains a sample program that uses our packaged `is_prime` function.

Listing 8.6: `usingprimecode.py`

```
from primecode import is_prime

def main():
    num = int(input("Enter an integer: "))
    if is_prime(num):
        print(num, "is prime")
    else:
        print(num, "is NOT prime")
```

The statement

```
from primecode import is_prime
```

directs the interpreter to import the `is_prime` function from the file `primecode.py`, which is the `primecode` module.

If we want our Listing 8.5 (`primecode.py`) module to be more widely available, we can place the file in a special Python library folder. This makes it available to all users on the system.

8.5 Documenting Functions and Modules

It is good practice to document a function's definition with information that aids programmers who may need to use or extend the function. The essential information includes:

- **The purpose of the function.** The function's purpose is not always evident merely from its name. This is especially true for functions that perform complex tasks. A few sentences explaining what the function does can be helpful.
- **The role of each parameter.** 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.

Before we consider the standard Python way of documenting functions and modules, we must introduce a new kind of string. Python supports multi-line strings. Triple quotes (`'''` or `"""`) delimit strings that can span multiple lines in the source code. Consider Listing 8.7 (`multilinestring.py`) that uses a multi-line string.

Listing 8.7: `multilinestring.py`

```
x = '''
This is a multi-line
    string that goes on
for three lines!
'''
print(x)
```

Listing 8.7 (`multilinestring.py`) displays

```
This is a multi-line
    string that goes on
for three lines!
```

Observe that the multi-line string obeys indentation and line breaks—essentially reproducing the same formatting as in the source code.

When such a string is the first line in the block of a function definition or the first line in a module, the string is known as a *documentation string*, or *docstring* for short. We can document our `is_prime` function as shown in Listing 8.8 (`docprime.py`).

Listing 8.8: `docprime.py`

```
'''
Contains the definition of the is_prime function
'''
from math import sqrt

def is_prime(n):
    '''
    Returns True if non-negative integer n is prime;
    otherwise, returns false
    '''
    trial_factor = 2
    root = sqrt(n)

    while trial_factor <= root:
        if n % trial_factor == 0:      # Is trial_factor a factor?
            return False              # Yes, return right away
        trial_factor += 1              # Consider next potential factor

    return True                       # Tried them all, must be prime
```

With the `docprime` module loaded into the interactive shell we can type:

```
>>> help(is_prime)
Help on function is_prime in module docprime:

is_prime(n)
    Returns True if non-negative integer n is prime;
    otherwise, returns false
>>>
```

The normal comments serve as *internal documentation* for developers of the `is_prime` function, while the function docstring serves as *external documentation* for caller 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 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).

8.6 Functions as Data

In Python, a function is special kind of object, just as integers, and strings are objects. Consider the following sequence in the interactive shell:

```

>>> type(2)
<class 'int'>
>>> type('Rick')
<class 'str'>
>>> from math import sqrt
>>> type(sqrt)
<class 'builtin_function_or_method'>

```

The `sqrt` function has the Python type `builtin_function_or_method`. The word *class* indicates that `builtin_function_or_method` is an object type, just as `int` and `str` are object types. As such, we can treat a function as if it were a data value; for example, we can assign a variable to a function, as shown here:

```

from math import sqrt

x = sqrt      # Assign x to sqrt function object
print(x(16))  # Prints 4.0

```

We also can pass a function as a parameter to another function. Listing 8.9 (arithmeticeval.py) includes a function that accepts a function as a parameter.

Listing 8.9: arithmeticeval.py

```
def add(x, y):
    """
    Adds the parameters x and y and returns the result
    """
    return x + y

def multiply(x, y):
    """
    Multiplies the parameters x and y and returns the result
    """
    return x * y

def evaluate(f, x, y):
    """
    Calls the function f with parameters x and y:
    f(x, y)
    """
    return f(x, y)

def main():
    """
    Tests the add, multiply, and evaluate functions
    """
    print(add(2, 3))
    print(multiply(2, 3))
    print(evaluate(add, 2, 3))
    print(evaluate(multiply, 2, 3))

main() # Call main
```

Listing 8.9 (arithmeticeval.py) prints

```
5
6
5
6
```

The first parameter of `evaluate`, `f`, is a function. Examining the code in the body of `evaluate`, we can see that `f` must be a function that can be called with two arguments. We also see that `evaluate` calls `f` passing it the second and third parameter it receives from its caller. The expression

`evaluate(add, 2, 3)`

passes the `add` function and the literal values 2 and 3 to `evaluate`. The `evaluate` function then invokes the `add` function with arguments 2 and 3.

A closer examination of the `add` function reveals that it applies the `+` operator to its two parameters. If its parameters are numbers, it computes the sum of its parameters. Notice, however, that the call

`print(evaluate(add, '2', '3'))`

would print

```
23
```

since applying the `+` operator to strings represents string concatenation instead of arithmetic addition. The `*` operator is not so flexible, as the following statement produces an error:

```
print(evaluate(multiply, '2', '3')) # Produces an exception
```

The `*` operator is not defined to operate on two string operands.

We will see in Section 11.2 that the ability to pass function objects around enables us to develop flexible algorithms that can be adapted at run time.

8.7 Lambda Expressions

CAUTION! SECTION UNDER CONSTRUCTION

H E R E

One of the primary benefits of functions is that we can write a function's code once and invoke it from many different places within the program (and even invoke it from other programs). Ordinarily, in order to call a function, we must know its name. Almost all the examples we have seen have invoked a function via its name. Listing 8.9 (`arithmeticval.py`) in Section 8.6 provided examples of invoking functions without using their names directly. Here we saw the `evaluate` function that accepts a function as a parameter:

```
def evaluate(f, x, y):
    return f(x, y)
```

`evaluate` calls `f`. The question is, what function does `evaluate` call here? `f` is the name of its parameter; there is no function named `f` available to the program. The answer, of course, `evaluate` invokes the function passed in from the caller. The function named `main` in Listing 8.9 (`arithmeticval.py`) calls `evaluate` passing the `add` function on one occasion and the `multiply` function on another.

The code in the `evaluate` function demands that callers send a function as the first parameter. Does this mean we have to write a separate function using `def` in order to call `evaluate`? What if we want to ensure that our function will execute exactly one time and only when invoked by `evaluate`?

Python supports the definition of simple, anonymous functions via `lambda` expressions. The general form of a `lambda` expression is

lambda *parameterlist* : *expression*

where:

- `lambda` is a reserved word that introduces a `lambda` expression.
- *parameterlist* is a comma-separated list of parameters as you would find in the function definition (but notice the lack of parentheses).

- *expression* is a single Python expression. The *expression* cannot be a complete statement.

The term *lambda* comes from *lambda calculus* (see http://en.wikipedia.org/wiki/Lambda_calculus), a function-based mathematical system developed by Alonzo Church in the 1930s. Concepts from lambda calculus led to the development of the modern computer.

We can use a `lambda` expression in a call to the `evaluate` function:

```
evaluate(lambda x, y: x * y, 2, 3)
```

The `lambda` keyword signifies the definition of an unnamed function; thus, the first argument being passed to `evaluate` is indeed a function. Notice that result of passing the `lambda` expression here is the same as passing the `multiply` function—both compute the product of the two parameters.

The following interactive session shows some additional examples of `lambda` expressions:

```
>>> def evaluate(f, x, y):
...     return f(x, y)
...
>>> evaluate(lambda x, y: 3*x + y, 10, 2)
32
>>> evaluate(lambda x, y: print(x, y), 10, 2)
10 2
>>> evaluate(lambda x, y: 10 if x == y else 2, 5, 5)
10
>>> evaluate(lambda x, y: 10 if x == y else 2, 5, 3)
2
```

The expression following the colon (:) in a `lambda` expression cannot be a Python statement. The conditional expression, for example, is acceptable, but an `if/else` statement is illegal. The expression's value is what the anonymous function *returns*, but the keyword `return` itself may not appear with the expression. Assignments are not possible within `lambda` expressions, and loops are not allowed.

One interesting aspect of `lambda` functions is that they form what is known as a *closure*. A closure is a function that can capture the context of its surrounding environment. Listing 8.10 (`closurein.py`) ...

Listing 8.10: `closurein.py`

```
def evaluate(f, x, y):
    return f(x, y)

def main():
    a = int(input('Enter an integer:'))
    print(evaluate(lambda x, y: False if x == a else True, 2, 3))

main()
```

Note that `main` creates a function (the `lambda` expression) that it passes to `evaluate`. It is important to note that the statement

```
print(evaluate(lambda x, y: False if x == a else True, 2, 3))
```

passes just three parameters to `evaluate`: a function and two integer values. The `main` function's local variable `a` is not passed as a parameter; instead, it is part of the code of the first parameter. The variable `a` is encoded into the `lambda` expression. We say the function definition (`lambda` expression) *captures*

the variable `a`. When the `evaluate` invokes the function sent by the caller, `evaluate` has no access to a variable named `a`. The `a` involved in the conditional expression is captured from `main`. This is an example of a closure transporting a captured variable into a function call.

For an example of a closure transporting a captured local variable out of a function, consider Listing 8.11 (`makeadder.py`) which includes a function that returns a function (`lambda` expression) to its caller.

Listing 8.11: `makeadder.py`

```
def make_adder():
    loc_val = 2    # Local variable definition
    return lambda x: x + loc_val    # Returns a function

def main():
    f = make_adder()
    print(f(10))
    print(f(2))

main()
```

Ordinarily when a function returns all of its local variables disappear. This means that after the following statement in the `main` function executes:

```
f = make_adder()
```

`make_adder`'s `loc_val` local variable should no longer exist. The function that `make_adder` returns, however, uses `loc_val` in its computation. This means the function that `make_adder` returns is a closure that captures the `make_adder`'s local variable `loc_val`. In the output of Listing 8.11 (`makeadder.py`) we can see that function `f` still has knowledge of `loc_val`'s value:

```
12
4
```

We can assign a variable to a `lambda` expression:

```
>>> f = lambda x: 2*x
>>> f(10)
20
```

This gives the function a name, so the statement

```
f = lambda x: 2*x
```

is equivalent to the function definition

```
def f(x):
    return 2*x
```

The function definition with `def` is much more powerful, however, because it can contain any number of arbitrary Python statements.

While not a particularly useful application of `lambda`, to demonstrate the regularity of the Python language, we can define an anonymous function and invoke it immediately; for example, the statement

```
print((lambda x, y: x * y)(2, 3))
```

prints 6. In this case the statement `print(2*3)` or, even better, `print(6)` produces the same result much more simply.

8.8 Generators

Ordinarily when a function returns to its caller the function relinquishes all of the memory for its local variables and parameters. The executing program then reuses this memory during calls to other functions, including re-calls to the same function. As a consequence, during every invocation a function begins fresh, with no traces of its past execution. This means a function normally cannot remember anything about past invocations.

We could use global variables to allow a function to remember some information. The function `remember` in Listing 8.12 (`funcmemory.py`) uses a global variable to keep track of the number of times it has been invoked.

Listing 8.12: `funcmemory.py`

```
count = 0      # A global count variable

def remember():
    global count
    count += 1  # Count this invocation
    print('Calling remember (#' + str(count) + ')')

print('Beginning program')
remember()
remember()
remember()
remember()
remember()
print('Ending program')
```

Listing 8.12 (`funcmemory.py`) prints

```
Beginning program
Calling remember (#1)
Calling remember (#2)
Calling remember (#3)
Calling remember (#4)
Calling remember (#5)
Ending program
```

Functions that access no global variables have precisely predictable behavior, which is a very desirable quality. In isolation we have no way to predict what the following statement will print:

```
remember()
```

We need to know the complete context. Certainly we need to know how many times the executing program previously called `remember`. Even that knowledge is insufficient in the context of a larger program that involves other functions. Other functions conceivably could manipulate the global count variable in between invocations to `remember`.

In order to write functions with persistence we need to use programming *objects*. We consider objects in depth in Chapters 9 and beyond, but for now we will consider a Python programming feature that invisibly uses an object behind the scenes.

A *generator* is a programming object that produces (that is, *generates*) a sequence of values. Code that uses a generator may obtain one value in the sequence at a time without the possibility of revisiting earlier values. We say the code that uses the generator *consumes* the generator's product.

Given only our current knowledge of functions, we can easily make and use generator objects. We create a generator within a function and “return” it. We do not use the `return` keyword; instead, we use the `yield` keyword. The code within the function definition specifies the behavior of the generator. A few simple examples illustrate how this works.

First, consider the module defined in Listing 8.13 (yieldsequence.py).

Listing 8.13: yieldsequence.py

```
def gen():
    yield 3
    yield 'wow'
    yield -1
    yield 1.2
```

The following interactive sequence reveals some information about the `gen` function within Listing 8.13 (yieldsequence.py):

```
>>> from yieldsequence import gen
>>> gen
<function gen at 0x00FA14B0>
>>> type(gen)
<class 'function'>
>>> gen()
<generator object gen at 0x00FAA300>
>>> type(gen())
<class 'generator'>
>>> x = gen()
>>> x
<generator object gen at 0x00FAA300>
>>> type(x)
<class 'generator'>
```

We see that `gen` is just a function, and, when invoked, `gen` returns a generator object. What can we do with a generator object?

Python has a built-in function named `next` that accepts a generator object and returns the next value in the generator's sequence. Consider the following interactive sequence:

```
>>> from yieldsequence import gen
>>> x = gen()
>>> next(x)
3
>>> next(x)
'wow'
>>> next(x)
-1
>>> next(x)
```

```
1.2
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

The statement

```
x = gen()
```

binds variable `x` to the generator object that `gen` returns. Once we have a generator object we can use the `next` function to extract the values in its sequence. Observe that we get an error if we ask the generator to provide a value after the final value in its sequence.

Programmers generally do not use the `next` function directly. Instead, they leave it to the `for` statement to call `next` behind the scenes. A generator object is one example of an iterable object. We learned in Section 5.3 that Python's `for` statement is built to work with an iterable object. The `for` statement, therefore, can iterate over the sequence of values produced by a generator object. Listing 8.14 (`forgenerator.py`) shows that the `for` statement works naturally with the generator our `gen` function produces.

Listing 8.14: `forgenerator.py`

```
def gen():
    yield 3
    yield 'wow'
    yield -1
    yield 1.2

for i in gen():
    print(i)
```

Listing 8.14 (`forgenerator.py`) prints

```
3
wow
-1
1.2
```

The `for` statement basically receives the next value from the generator each time through the loop. The loop stops automatically when the generator has no more values left.

The `yield` statement within the function generates the values of the sequence. It is uncommon to provide separate `yield` statements for each value the generator is to produce. More likely, one `yield` statement executes multiple times within a loop. Listing 8.15 (`regulargenerator.py`) shows a more common scenario.

Listing 8.15: `regulargenerator.py`

```
def generate_multiples(m, n):
    count = 0
    while count < n:
        yield m * count
        count += 1

def main():
```

```
for mult in generate_multiples(3, 6):
    print(mult, end=' ')
print()

if __name__ == '__main__':
    main()
```

Listing 8.15 (regulargenerator.py) prints the first six multiples of three:

```
0 3 6 9 12 15
```

The `generate_multiples` function in Listing 8.15 (regulargenerator.py) contains only one `yield` statement, but the loop executes the `yield` statement *n* times.

Listing 8.16 (myrange.py) shows how we can use a generator to simulate the behavior of the built-in `range` expression.

Listing 8.16: myrange.py

```
def myrange(arg1, arg2=None, step=1):
    if arg2 != None: # Do we have at least two arguments?
        begin = arg1
        end = arg2
    else: # We must have just one argument
        begin = 0 # Begin value is zero by default
        end = arg1
    i = begin
    while i != end:
        yield i
        i += step

print('0 to 9:', end=' ')
for i in myrange(10):
    print(i, end=' ')
print()

print('1 to 10:', end=' ')
for i in myrange(1, 11):
    print(i, end=' ')
print()

print('2 to 18 by twos:', end=' ')
for i in myrange(2, 20, 2):
    print(i, end=' ')
print()

print('20 down to 2 by twos:', end=' ')
for i in myrange(20, 0, -2):
    print(i, end=' ')
print()
```

Listing 8.16 (myrange.py) prints

```
0 to 9: 0 1 2 3 4 5 6 7 8 9
1 to 10: 1 2 3 4 5 6 7 8 9 10
2 to 18 by twos: 2 4 6 8 10 12 14 16 18
20 down to 2 by twos: 20 18 16 14 12 10 8 6 4 2
```

While our `myrange` function works like Python's built-in `range` expression, in fact, `range` is different. A simple exercise with the interactive shell reveals:

```
>>> range
<class 'range'>
>>> range(10)
range(0, 10)
>>> type(range)
<class 'type'>
>>> type(range(10))
<class 'range'>
```

The expression `range(0, 10)` does not return a generator object but instead creates and returns a `range` object. Furthermore, the interactive sequence shows that `range` is not a function at all; it is a *class*. In reality, the expression `range(0, 10)` calls the `range` class *constructor*. We will not be concerned with such details about objects until Chapter 14. For now we will be content with the understanding that the `for` statement is designed to work with iterable objects, and generators and `range` objects are both instances of iterable objects.

Our `myrange` function may be interesting, but it offers no advantage over the built-in `range` expression. It is time to use a generator in a more interesting way. Recall Listing 7.11 (`primefunc.py`) that uses a function named `is_prime` in the course of printing the prime numbers within a range specified by the user. What if we wish to print only the prime numbers within a range that end with the digit 3? What if wish to add up all the prime numbers within a given range? A generator is ideal for implementing the more modular and flexible code required to generate prime numbers independent of how a program uses them. Listing 8.17 (`generatedprimes.py`) uses a generator function to produce the sequence of prime numbers. The caller (`main`) then can select which values to print and sum the numbers in a sequence. In Listing 8.17 (`generatedprimes.py`) we further tune the `is_prime` function from the observations that two is the only even prime number and that no prime number except two may have a factor that is even. Applying these facts allows us to cut by one-half the potential factors to consider within the loop.

Listing 8.17: `generatedprimes.py`

```
# Contains the definition of the is_prime function
from math import sqrt

def is_prime(n):
    ''' Returns True if non-negative integer n is prime;
        otherwise, returns false '''
    if n == 2:                # 2 is the only even prime number
        return True
    if n < 2 or n % 2 == 0:    # Handle simple cases immediately
        return False         # No evens and nothing less than 2
    trial_factor = 3
    root = sqrt(n)
    while trial_factor <= root:
        if n % trial_factor == 0:    # Is trial factor a factor?
            return False           # Yes, return right away
        trial_factor += 2           # Next potential factor, skip evens
```

```

    return True                                # Tried them all, must be prime

def prime_sequence(begin, end):
    ''' Generates the sequence of prime numbers between begin and end. '''
    for value in range(begin, end + 1):
        if is_prime(value):                    # See if value is prime
            yield value                        # Produce the prime number

def main():
    ''' Experiments with the prime number generator '''
    min_value = int(input("Enter start of range: "))
    max_value = int(input("Enter last of range: "))

    print('Print all the primes from', min_value, 'to', max_value)
    for value in prime_sequence(min_value, max_value):
        print(value, end=' ') # Display the prime number
    print() # Move cursor down to next line

    print('Print all the primes in that range that end with digit 3')
    for value in prime_sequence(min_value, max_value):
        if value % 10 == 3:                    # See if value's ones digit is 3
            print(value, end=' ') # Display the number
    print() # Move cursor down to next line

    # Add up all the primes in the range
    sum = 0
    for value in prime_sequence(min_value, max_value):
        sum += value
    print('The sum of the primes in that range is', sum)

    # Decorate the output
    print('Fancier display')
    for value in prime_sequence(min_value, max_value):
        print('<' + str(value) + '>', end='')

if __name__ == '__main__':
    main() # Run the program

```

Listing 8.17 (generatedprimes.py) prints

```

Enter start of range: 20
Enter last of range: 50
Print all the primes from 20 to 50
23 29 31 37 41 43 47
Print all the primes in that range that end with digit 3
23 43
The sum of the primes in that range is 251
Fancier display
<23><29><31><37><41><43><47>

```

Each call of the expression

```
prime_sequence(min_value, max_value)
```

creates a new generator object ready to use.

As Listing 8.17 (`generatedprimes.py`) demonstrates, a generator is useful when you need to centralize and reuse code that produces a sequence of values and you cannot predict how consumers of the sequence will use those values.

8.9 Local Function Definitions

CAUTION! SECTION UNDER CONSTRUCTION

In Section 7 we introduced functional decomposition. Programmers can use functions to break up a large, complex, monolithic program into smaller, more manageable pieces. The code within a function is somewhat insulated from the rest of the program, in that a caller can influence the behavior of a function only via the function's parameters and any global variables the function may use. Any local variables the function uses are invisible to code outside of the function.

Suppose we develop a function that itself becomes large and unwieldy. We further can break down the large function into smaller pieces as Figure 8.2 shows. The perhaps unintended consequence of this

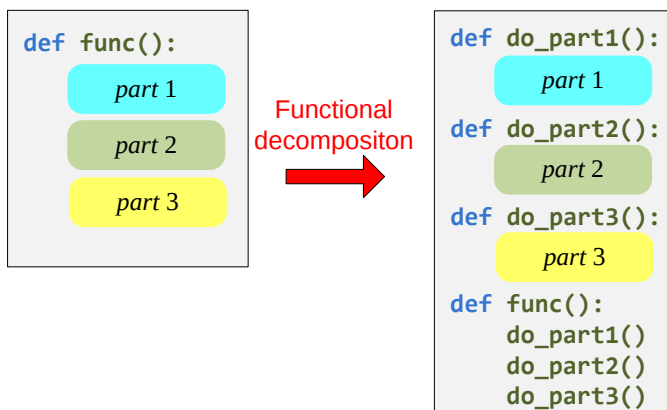


Figure 8.2: Decomposing a larger function into a collection of smaller functions. Callers now have access to the individual functions `do_part1`, `do_part2`, and `do_part3`, as well as to `func`.

functional decomposition is that callers now can bypass the original function and access its pieces directly and individually. Sometimes this more fine-grained access is desirable, but at other times programmers do not want to expose that level of detail to callers.

Generalizing the concept of local variables, Python permits programmers to define functions within other function definitions. These local functions are available to the code *within* their enclosing function but are inaccessible *outside* their enclosing function. Figure 8.3 shows how to restructure the `func` function definition using local functions to enable functional decomposition without violating the code encapsulation of the original monolithic function. Listing 8.18 (`boxmeasure.py`) includes a function named `surface_area` that computes the surface area of a rectangular box. The function expects the eight points that represent the corners of the box. Note that the `surface_area` function uses a local function to compute the area of each of its sides. Listing 8.18 (`boxmeasure.py`) also includes a function named `volume` also computes the box's

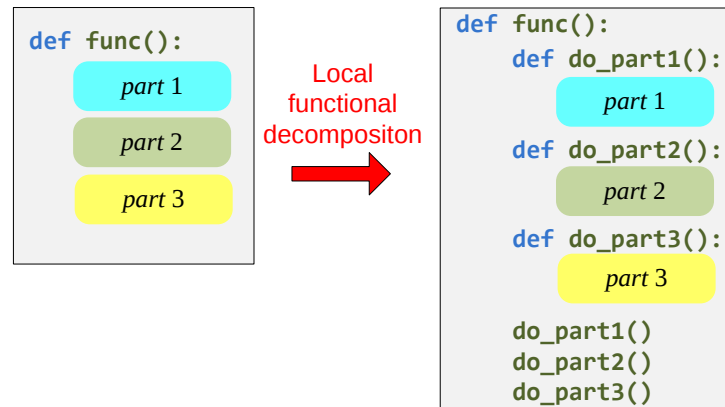


Figure 8.3: The local functions `do_part1`, `do_part2`, and `do_part3` are available to function `func` but are inaccessible outside the `func` function

volume.

Listing 8.18: `boxmeasure.py`

```
from math import fabs

def surface_area(x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4,
                x5, y5, z5, x6, y6, z6, x7, y7, z7, x8, y8, z8):
    ''' Computes the surface area of a rectangular box
        (cuboid) defined by the 3D points (x,y,z) of
        its eight corners:

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

        ...

    # Local helper function to compute area
    def area(length, width):
        ''' Computes the area of a length x width rectangle '''
        return length * width

    # Main code for surface_area function
    # Compute area of front face
    length = fabs(x2 - x1)
    height = fabs(y3 - y1)
    front_area = area(length, height)
    # Compute area of side face
    width = fabs(z5 - z1)
    side_area = area(width, height)
```

```

# Compute area of top face
top_area = area(length, width)
# Compute and return surface area: front/back,
# left side/right side, and top/bottom faces
return 2*front_area + 2*side_area + 2*top_area

def volume(length, width, height):
    ''' Computes the volume of a rectangular box
        (cuboid) defined by its length, width, and height '''
    return length * width * height

# Get the coordinates of the box's corners from the user
print('Enter the coordinates of each of the box\'s corners')
print('''
      7-----8
      /|      /|
      3-----4 |
      | |      | |
      | 5----|-6
      | /      | /
      1-----2
      ''')
x1, y1, z1 = eval(input('Corner 1 (x,y,z):'))
x2, y2, z2 = eval(input('Corner 2 (x,y,z):'))
x3, y3, z3 = eval(input('Corner 3 (x,y,z):'))
x4, y4, z4 = eval(input('Corner 4 (x,y,z):'))
x5, y5, z5 = eval(input('Corner 5 (x,y,z):'))
x6, y6, z6 = eval(input('Corner 6 (x,y,z):'))
x7, y7, z7 = eval(input('Corner 7 (x,y,z):'))
x8, y8, z8 = eval(input('Corner 8 (x,y,z):'))

# Compute the surface area of the box
print('Surface area:', surface_area(x1, y1, z1, x2, y2, z2,
                                   x3, y3, z3, x4, y4, z4,
                                   x5, y5, z5, x6, y6, z6,
                                   x7, y7, z7, x8, y8, z8))

# Compute the volume of the box
ln = fabs(x2 - x1) # Compute length
wd = fabs(z5 - z1) # Compute width
ht = fabs(y3 - y1) # Compute height
print('Volume:', volume(ln, wd, ht))

```

The following shows a sample run of Listing 8.18 (boxmeasure.py):

```

Enter the coordinates of each of the box's corners

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

```

```

Corner 1 (x,y,z):(0,0,0)
Corner 2 (x,y,z):(2,0,0)
Corner 3 (x,y,z):(0,2,0)
Corner 4 (x,y,z):(2,2,0)
Corner 5 (x,y,z):(0,0,2)
Corner 6 (x,y,z):(2,0,2)
Corner 7 (x,y,z):(0,2,2)
Corner 8 (x,y,z):(2,2,2)
Surface area: 24.0
Volume: 8.0

```

Only code within the `surface_area` function can use the `area` function. We could define a function named `area` local to the `volume` function, and, if we did, the two like-named functions would be completely separate functions.

Local functions can access the local variables and other local functions defined by enclosing function. As with the global functions we have seen before this section, any variable defined within a local function is a local variable of that function. If we need a local function to modify a variable defined in its outer scope (its enclosing function), we must declare the variable as `nonlocal`. The `global` keyword declares a variable as truly global, so we cannot use the `global` keyword in place of `nonlocal` in this situation.

Listing 8.19 (`localcounter.py`) uses a local function to mimic a generator.

Listing 8.19: `localcounter.py`

```

def create_counter(n):
    ''' Creates a counting function that counts up to n '''
    count = 0

    def counter():      # Local function
        ''' Increments the outer variable unless it
            has reached its limit '''
        nonlocal count
        if count < n:
            count += 1
        return count

    return counter      # Returns a function

ctr = create_counter(4)
print(ctr())
print(ctr())
print(ctr())
print(ctr())
print(ctr())
print(ctr())

```

Listing 8.19 (`localcounter.py`) prints

```

1
2
3
4
4

```

4

Note that the `create_counter` function returns a function; it returns its own local function. This returned local function “remembers” the value of its enclosing function’s local variable `count`; thus, it represents a closure (see Section 8.7). Since the `count` variable in the enclosing function is not global, no outside code can modify the `count` variable except by calling the function that `create_counter` returns.

It may appear that `create_counter` is similar to a generator function (see Section 8.8). Unfortunately the `create_counter` function in Listing 8.19 (`localcounter.py`) does not create a generator object, as it has no `yield` statement. This means we cannot use it in a `for` statement, and it does not work with the global `next` function.

As another example of a function returning a local closure, consider the calculation of a *derivative*. Those familiar with basic calculus will recall the derivative of a function f at a is defined to be

$$f'(a) = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h}$$

If you are unfamiliar with calculus, all you need to know is that the derivative of a function is itself a function; the above formula shows how to transform a function into its derivative. The process of computing a derivative is known as *differentiation*. The $\lim_{h \rightarrow 0}$ notation indicates that the answer becomes more precise as the value h gets closer to zero. Letting h be exactly zero would result in division by zero, which is undefined.

Based on the mathematical definition we can define a Python function that computes the derivative of another function, as shown here:

```
def derivative(f, h):
    return lambda x: (f(x + h) - f(x)) / h
```

Note that the derivative function returns a function—a `lambda` expression is a simple function definition (see Section 8.7). The function that `derivative` returns is a closure because it captures the function parameters `f` and `h`.

Our derivative function allows us to compute the derivative of a function at a given value. This is known as *numerical differentiation*. Another approach (the one emphasized in calculus courses) uses *symbolic differentiation*. Symbolic differentiation transforms the formula for a function into a different formula. The details of symbolic differentiation are beyond the scope of this text, but we will use one of its results for a particular function to check our computed numerical results.

A particular function f is defined as follows:

$$f(x) = 3x^2 + 5$$

f ’s derivative is f' :

$$f'(x) = 6x$$

Listing 8.20 (`derivative.py`) uses the derivative function on $f(x) = 3x^2 + 5$ and compares its results with the known solution, $f'(x) = 6x$.

Listing 8.20: `derivative.py`

```
def derivative(f, h):
    ''' Approximates the derivative of function f
        given an h value. The closer h is to zero,
        the better the estimate. '''
```

```

    return lambda x: (f(x + h) - f(x)) / h

def fun(x):    # The function we wish to differentiate
    return 3*x**2 + 5

def ans(x):    # The known derivative to function fun
    return 6*x

# Difference: Approximation better as h -> 0
h = 0.0000001

# Compute the function representing an approximation
# of the derivative of function fun
der = derivative(fun, h)

# Compare the computed derivative to the exact derivative
# derived symbolically
x = 5.0
print('-----')
print('
Approx.    Actual')
print('  x      f(x)      h      f\'(x)      f\'(x)')
print('-----')
while x < 5.1:
    print('{:.5f}   {:.5f}   {:.8f}   {:.5f}   {:.5f}'.format(x, fun(x), h, der(x), ans(x)))
    x += 0.01

```

With $h = 0.0000001$, Listing 8.20 (derivative.py) produces good results to the fifth decimal place:

x	f(x)	h	Approx. f'(x)	Actual f'(x)
5.00000	80.00000	0.00000010	30.00000	30.00000
5.01000	80.30030	0.00000010	30.06000	30.06000
5.02000	80.60120	0.00000010	30.12000	30.12000
5.03000	80.90270	0.00000010	30.18000	30.18000
5.04000	81.20480	0.00000010	30.24000	30.24000
5.05000	81.50750	0.00000010	30.30000	30.30000
5.06000	81.81080	0.00000010	30.36000	30.36000
5.07000	82.11470	0.00000010	30.42000	30.42000
5.08000	82.41920	0.00000010	30.48000	30.48000
5.09000	82.72430	0.00000010	30.54000	30.54000
5.10000	83.03000	0.00000010	30.60000	30.60000

Even with h as large as 0.01, the results are not too bad:

x	f(x)	h	Approx. f'(x)	Actual f'(x)
5.00000	80.00000	0.01000000	30.03000	30.00000
5.01000	80.30030	0.01000000	30.09000	30.06000
5.02000	80.60120	0.01000000	30.15000	30.12000
5.03000	80.90270	0.01000000	30.21000	30.18000
5.04000	81.20480	0.01000000	30.27000	30.24000

5.05000	81.50750	0.01000000	30.33000	30.30000
5.06000	81.81080	0.01000000	30.39000	30.36000
5.07000	82.11470	0.01000000	30.45000	30.42000
5.08000	82.41920	0.01000000	30.51000	30.48000
5.09000	82.72430	0.01000000	30.57000	30.54000
5.10000	83.03000	0.01000000	30.63000	30.60000

In Listing 8.20 (derivative.py), the statement

```
der = derivative(fun, h)
```

assigns `der` to the function returned by `derivative`; thus, `der(x)` returns the value of the derivative of `fun` at `x`. In order for `der` to compute its answer, it must have access to function `fun`. It has this access because `der` is a closure that captured `fun` during the call to `derivative`.

8.10 Summary

- A global variable is defined outside of all functions and it available to all functions within its scope.
- A global variable exists for the life of the program, but local variables are created during a function call and are discarded when the function's execution has completed.
- Modifying a global variable can directly affect the behavior of any function that uses that global variable. A function that uses a global variable cannot be tested in isolation since its behavior can vary depending on how code outside the function modifies the global variable it uses.
- The behavior of an independent function is determined strictly by the parameters passed into it. An independent function will not use global variables.
- Local variables are preferred to global variables, since the indiscriminate use of global variables leads to functions that are less flexible, less reusable, and more difficult to understand.
- Programmers can define default values for functions parameters; these default parameters are substituted for parameters not supplied by callers.
- In functions that use default parameters, the default parameters must appear after all the non-default parameters in the function's parameter list.
- A recursive function optionally must call itself or not as determined by a conditional statement. The call of itself is the recursive case, and the base case does not make the recursive all. Each recursive call should move the computation closer to the base case.
- One or more functions in a file make up a module. Client programs can import these functions with an `import` statement.
- Multi-line strings are enclosed with triple quote marks (`'''` or `"""`). Such strings retain the same formatting as they appear in the source code.
- Document strings within functions and modules allow client programmers to obtain useful information about the functions and modules.
- Programmers should document each function indicating the function's purpose and the role(s) of its parameter(s) and return value. Additional information about the function's author, date of last modification, and other information may be required in some situations.

- A function can be passed as a parameter to another function. This ability enables the creation of more flexible algorithms.
- A generator object produces a sequence of values.
- A generator is an example of an iterable object. The `for` statement can iterate over sequence of values produced by a generator.
- The `yield` statement used in place of a `return` statement within a function definition produces a generator object.

8.11 Exercises

1. Consider the following Python code:

```
def sum1(n):
    s = 0
    while n > 0:
        s += 1
        n -= 1
    return s

val = 0

def sum2():
    s = 0
    while val > 0:
        s += 1
        val -= 1
    return s

def sum3():
    s = 0
    for i in range(val, 0, -1):
        s += 1
    return s

def main():
    # See each question below for details

main() # Call main function
```

- (a) What is printed if `main` is written as follows?

```
def main():
    global val
    val = 5
    print(sum1(input))
    print(sum2())
    print(sum3())
```

- (b) What is printed if main is written as follows?

```
def main():
    global val
    val = 5
    print(sum1(input))
    print(sum3())
    print(sum2())
```

- (c) What is printed if main is written as follows?

```
def main():
    global val
    val = 5
    print(sum2())
    print(sum1(input))
    print(sum3())
```

- (d) Which of the functions sum1, sum2, and sum3 produce a side effect? What is the side effect?
- (e) Which function may not use the val variable?
- (f) What is the scope of the variable val? What is its lifetime?
- (g) What is the scope of the variable i? What is its lifetime?
- (h) Which of the functions sum1, sum2, and sum3 demonstrate good functional independence? Why?

2. Consider the following Python code:

```
def next_int1():
    cnt = 0
    cnt += 1
    return cnt

global_count = 0

def next_int2():
    global_count += 1
    return global_count

def main():
    for i = range(0, 5):
        print(next_int1(), next_int2())

main()
```

- (a) What does the program print?
- (b) Which of the functions next_int1 and next_int2 is the best function for the intended purpose? Why?
- (c) What is a better name for the function named next_int1?
- (d) The next_int2 function works in this context, but why is it not a good implementation of function that always returns the next largest integer?

3. What does the following Python program print?

```
def sum(m=0, n=0, r=0):  
    return m + n + r  
  
def main():  
    print(sum())  
    print(sum(4))  
    print(sum(4, 5))  
    print(sum(5, 4))  
    print(sum(1, 2, 3))  
    print(sum(2.6, 1.0, 3))  
  
main()
```

4. Consider the following function:

```
def proc(n):  
    if n < 1:  
        return 1  
    else:  
        return proc(n/2) + proc(n - 1)
```

Evaluate each of the following expressions:

- (a) `proc(0)`
 - (b) `proc(1)`
 - (c) `proc(2)`
 - (d) `proc(3)`
 - (e) `proc(5)`
 - (f) `proc(10)`
5. Rewrite the `gcd` function so that it implements Euclid's method but uses iteration instead of recursion.
6. Classify the following functions as pure or impure. `x` is a global variable.

- (a)

```
def f1(m, n):  
    return 2*m + 3*n
```
- (b)

```
def f2(n)  
    return n - 2
```
- (c)

```
def f3(n):  
    return n - x
```
- (d)

```
def f4(n):  
    print(2*n)
```
- (e)

```
def f5(n):  
    m = eval(input())  
    return m * n
```

```
(f) def f6(n):  
    m = 2*n  
    p = 2*m - 5  
    return p - n
```

7. Modify the main function in Listing 8.17 (`generatedprimes.py`) that so that it prints all the prime numbers less than 1,000 that contain the digit 2 or digit 3 (or both).
8. Write a generator function named `evens` that enables the following code:

```
for n in evens_less_than(12):  
    print(n, end=' ')  
print()
```

to print

```
2 4 6 8 10
```

that is, all positive even numbers less than 12.

9. Write a generator function named `oscillate` that enables the following code:

```
for n in oscillate(-3, 5):  
    print(n, end=' ')  
print()
```

to print

```
-3 3 -2 2 -1 1 0 0 1 -1 2 -2 3 -3 4 -4
```

Chapter 9

Objects

In the hardware arena, a personal computer is built by assembling a motherboard (a circuit board containing sockets for a microprocessor and assorted support chips), a processor, memory, a video card, a disk controller, a disk drive, a case, a keyboard, a mouse, and a monitor. The video card by itself is a sophisticated piece of hardware containing a video processor chip, memory, and other electronic components. A technician does not need to assemble the card; the card is used as is off the shelf. The video card provides a substantial amount of functionality in a standard package. One video card can be replaced with another card from a different vendor or with another card with different capabilities. The overall computer will work with either card (subject to availability of drivers for the operating system) because standard interfaces allow the components to work together.

Software development today is increasingly *component based*. Software components are used like hardware components. A software system can be built largely by assembling pre-existing software building blocks. Python supports various kinds of software building blocks. The simplest of these is the *function* that we investigated in Chapter 6 and Chapter 7. A more powerful technique uses software *objects*.

Python is *object oriented*. Most modern programming languages support object-oriented (OO) development to one degree or another. An OO programming language allows the programmer to define, create, and manipulate objects. Objects bundle together data and functions. Like other variables, each Python object has a type, or *class*. The terms *class* and *type* are synonymous.

In this chapter we explore some of the classes available in the Python standard library.

9.1 Using Objects

An object is an instance of a class. We have been using objects since the beginning, but we have not taken advantage of all the capabilities that objects provide. Integers, floating-point numbers, strings, and functions are all objects in Python. With the exception of function objects, we have treated these objects as passive data. We can assign an integer value to a variable and then use that variable's value. We can add two floating-point numbers and concatenate two strings with the + operator. We can pass objects to functions and functions can return objects.

In object-oriented programming, rather than treating data as passive values and functions as active agents that manipulate data, we fuse data and functions together into software units called *objects*. A typical object consists of two parts: *data* and *methods*. An object's data is sometimes called its *attributes*.

or *fields*. Methods are like functions, and they are known also as *operations*. The data and methods of an object constitutes the object's *members*. The code that uses an object is called the object's *client*. We say that an object provides a service to its clients. The services provided by an object can be more elaborate than those provided by simple functions because objects make it easy to store persistent data.

9.2 String Objects

We have been using string objects—instances of class `str`—for some time. Objects bundle data and functions together, and the data that comprise a string consist of the sequence of characters that make up the string. We now turn our attention to string methods. Listing 9.1 (`stringupper.py`) shows how a programmer can use the `upper` method available to string objects.

Listing 9.1: `stringupper.py`

```
name = input("Please enter your name: ")
print("Hello " + name.upper() + ", how are you?")
```

Listing 9.1 (`stringupper.py`) capitalizes (converts to uppercase) all the letters in the string the user enters:

```
Please enter your name: Rick
Hello RICK, how are you?
```

The expression

`name.upper()`

within the `print` statement represents a *method call*. The general form of a method call is

object . *method name* (*parameter list*)

- *object* is an expression that represents object. In the example in Listing 9.1 (`stringupper.py`), `name` is a reference to a string object.
- The period, pronounced *dot*, associates an object expression with the method to be called.
- *methodname* is the name of the method to execute.
- The *parameterlist* is comma-separated list of parameters to the method. For some methods the parameter list may be empty, but the parentheses always are required. The parameter list for a method call works exactly like the parameter list for a function call.

Except for the object prefix, a method call works like a function call. It can return a value to its caller.

The `upper` method returns a new string. The `upper` method does not affect the object upon which it is called; this means `name.upper()` does not modify the `name` object.

A method may accept parameters. Listing 9.2 (`rjustprog.py`), uses the `rjust` string method to right justify a string padded with a specified character.

Listing 9.2: rjustprog.py

```
word = "ABCD"
print(word.rjust(10, "*"))
print(word.rjust(3, "*"))
print(word.rjust(15, ">"))
print(word.rjust(10))
```

The output of Listing 9.2 (rjustprog.py):

```
*****ABCD
ABCD
>>>>>>>>>>ABCD
      ABCD
```

shows

- `word.rjust(10, "★")` right justifies the string "ABCD" within a 10-character horizontal area padded with ★ characters.
- `word.rjust(3, "★")` does not return a different string from the original "ABCD" since the specified width (3) is less than or equal to the length of the original string (4).
- `word.rjust(10)` shows that the default padding character is a space.

The following interactive sequence shows that we can call a method from a string literal:

```
>>> 'aBcDeFgHiJ'.upper()
'ABCDEFGHij'
>>> 'This is a sentence.'.rjust(25, '-')
'-----This is a sentence.'
```

This syntax may look somewhat familiar; we introduced the string format method in Section 2.9 and invoked it with a string literal. We also can use a string variable, as shown here:

```
>>> '{0} {1}'.format(23, 9)
'23 9'
>>> s = '{0} {1}'
>>> s.format(23, 9)
'23 9'
```

Table 9.1 lists some of the methods available to `str` objects.

str Methods	
upper	Returns a copy of the original string with all the characters converted to uppercase
lower	Returns a copy of the original string with all the characters converted to lower case
rjust	Returns a string right justified within an area padded with a specified character which defaults to a space
ljust	Returns a string left justified within an area padded with a specified character which defaults to a space
center	Returns a copy of the string centered within an area of a given width and optional fill characters; fill characters default to spaces
strip	Returns a copy of the given string with the leading and trailing whitespace removed; if provided an optional string, the strip function strips leading and trailing characters found in the parameter string
startswith	Determines if the string parameter is a prefix of the invoking string
endswith	Determines if the string parameter is a suffix of the invoking string
count	Determines the number times the string parameter is found as a substring within the invoking string; the count includes only non-overlapping occurrences
find	Returns the lowest index where the string parameter is found as a substring of the invoking string; returns <code>-1</code> if the parameter is not a substring of the invoking string
format	Embeds formatted values in a string using <code>{0}</code> , <code>{1}</code> , etc. position parameters

Table 9.1: A few of the methods available to str objects

Listing 9.3 (stripandcount.py) demonstrates two of the string methods.

Listing 9.3: stripandcount.py

```
# Strip leading and trailing whitespace and count substrings
s = "   ABCDEFGHBCDIJKLMNOPQRSBCDTUVWXYZ   "
print("[", s, "]", sep="")
s = s.strip()
print("[", s, "]", sep="")

# Count occurrences of the substring "BCD"
print(s.count("BCD"))
```

Listing 9.3 (stripandcount.py) displays:

```
[   ABCDEFGHBCDIJKLMNOPQRSBCDTUVWXYZ   ]
[ABCDEFGHBCDIJKLMNOPQRSBCDTUVWXYZ]
```

3

The `str` class provides a `__getitem__` method that returns the character at a given position within the string. Since the method's name begins with two underscores (`__`), the method is meant for internal class use and not for clients. The `__getitem__` method is special, as clients can access it via a special syntax:

```
>>> s = 'ABCEFGHI'
>>> s
'ABCEFGHI'
>>> s.__getitem__(0)
'A'
>>> s.__getitem__(1)
'B'
>>> s.__getitem__(2)
'C'
>>> s[0]
'A'
>>> s[1]
'B'
>>> s[2]
'C'
```

The square brackets when used with an object in the manner shown above invoke that object's `__getitem__` method. In the case of string objects the integer within the square brackets, known as an *index*, represents the distance from the beginning of the string from which to obtain a character. For string `s`, `s[0]` is the first character in the string, `s[1]` is the second character, as so forth.

Strings also provide a `__len__` method that returns the number of characters that make up the string. Again, since the name `__len__` begins with two underscores, clients are supposed to invoke it in a different way. The following shows the preferred way of determining a string's length:

```
>>> s
'ABCEFGHI'
>>> s = 'ABCEFGHI'
>>> s
'ABCEFGHI'
>>> len(s)
8
>>> s.__len__()
8
```

The expressions `len(s)` and `s.__len__()` are functional equivalent. Instead of calling the `__len__` method directly, clients should use the global `len` function. Listing 9.4 (`printcharacters.py`) uses the `len` function and `[]` index operator to print the individual characters that make up a string.

Listing 9.4: `printcharacters.py`

```
s = "ABCDEFGHIJK"
print(s)
for i in range(len(s)):
    print("[", s[i], "]", sep="", end="")
print() # Print newline

for ch in s:
```

```
print("<", ch, ">", sep="", end="")
print() # Print newline
```

Strings are immutable objects. This means we cannot modify the contents of a string object:

```
s = 'ABCDEFGHIJKLMN'
s[3] = 'S' # Illegal, strings are immutable
```

String immutability means a method such as `strip` may not change a given string:

```
s = "   ABC   "
s.strip() # s is unchanged
print("<" + s + ">") # Prints <   ABC   >, not <ABC>
```

In order to strip the leading and trailing whitespace as far as the string bound to the variable `s` is concerned, we must reassign `s`:

```
s = "   ABC   "
s = s.strip() # Note the reassignment
print("<" + s + ">") # Prints <ABC>
```

The `strip` method returns a new string; the string on whose behalf `strip` is called is not modified. To effectively strip the whitespace from a string, a client must, as in this example, reassign its variable to the string passed back by the `strip` method.

When treated as a Boolean expression, the empty string (`' '`) is interpreted as `False`, and all other strings are considered `True`.

9.3 File Objects

So far all the programs we have seen lose their data at the end of their execution. Most useful applications, however, require greater data persistence. Imagine using a word processor that does *not* allow you to save your document and retrieve it later for further editing. Most modern operating systems store persistent data in *files*. A word processor, for example, could store one of its documents in a file named `thesis.doc`.

Fortunately, Python's standard library has a file class that makes it easy for programmers to make objects that can store data to, and retrieve data from, disk. The formal name of the class of file objects we will be using is `TextIOWrapper`, and it is found in the `io` module. Since file processing is such a common activity, the functions and classes defined in the `io` module are available to any program, and no `import` statement is required.

The statement

```
f = open('myfile.txt', 'r')
```

creates and returns a file object (literally a `TextIOWrapper` object) named `f`. The first argument to `open` is the name of the file, and the second argument is a mode. The `open` function supports the following modes:

- `'r'` opens the file for reading
- `'w'` opens the file for writing; creates a new file
- `'a'` opens the file to append data to it

The statement

```
f = open('myfile.txt', 'r')
```

creates a file object named `f` capable of reading the contents of the text file named `myfile.txt`. If the file does not exist or the user of the program does not have adequate permissions to open the file, the `open` function will raise an exception.

The statement

```
f = open('myfile.txt', 'w')
```

creates and returns a file object named `f` capable of writing data to the text file named `myfile.txt`. If the file does not exist, the function creates the file on disk. If a file by that name currently exists, new data will replace the current data stored in the file. This means any pre-existing data in the file will be lost.

The statement

```
f = open('myfile.txt', 'a')
```

creates and returns a file object named `f` capable of writing data to the text file named `myfile.txt`. If the file does not exist, the function creates the file on disk. If a file by that name currently exists, new data will be appended after the pre-existing data in that file. This means that the original data in the file is not lost.

If the second argument to the `open` function is missing, it defaults to `'r'`, so the statement

```
f = open(fname)
```

is equivalent to

```
f = open(fname, 'r')
```

Once you have a file object capable of writing (opened with `'w'` or `'a'`) you can save data to the file associated with that file object using the `write` method. For a file object named `f`, the statement

```
f.write('data')
```

stores the string `'data'` to the file. The three statements

```
f.write('data')
f.write('compute')
f.write('process')
```

writes the text `'datacomputeprocess'` to the file. If our intention is to retrieve the three separate original strings, we must add delimiters to separate the pieces of data. Newline characters serve as good delimiters:

```
f.write('data\n')
f.write('compute\n')
f.write('process\n')
```

This places each word on its own line in the text file. The advantage of storing each piece of data on its own line of text is that it makes it easier to read the data from the file with a `for` statement. If `f` is a file object created for reading, the following code:

```
for line in f:
    print(line.strip())
```

reads in each line of text from the file and prints it out. The variable `line` is a string, and we use the `strip` method to remove the trailing newline (`'\n'`) character.

We also can read the contents of the entire file into a single string using the file object's `read` method:

```
contents = f.read()
```

Given the text file from above, the code

```
in = open('compterm.txt', 'r')
s = in.read()
```

assigns to `s` the string `'data\ncompute\nprocess\n'`.

The `open` method opens a file for reading or writing, and the `read`, `write`, and other such methods enable the program to interact with the file. When the executing program is finished with its file processing it must call the `close` method to close the file properly. Failure to close a file can have serious consequences when writing to a file, as data meant to be saved could be lost. Every call to the `open` function should have a corresponding call to the file object's `close` method.

Listing 9.5 (`simplefileread.py`) opens a file named `data.dat` for reading and reads in and prints out each line of text:

Listing 9.5: `simplefileread.py`

```
f = open('data.dat')      # f is a file object
for line in f:             # Read each line as text
    print(line.strip())    # Remove trailing newline character
f.close()                 # Close the file
```

If the file `data.dat` does not exist or there are issues such as the user does not have sufficient permissions to read the file, the executing program will raise an exception.

Since it is important to always close a file after opening it, Python offers a simpler way to express Listing 9.5 (`simplefileread.py`) that automatically closes the file when finished. Listing 9.6 (`simplerread.py`) uses the `with/as` statement to create what is known as a *context manager* that ensures the file is closed.

Listing 9.6: `simplerread.py`

```
with open('data.dat') as f:  # f is a file object
    for line in f:           # Read each line as text
        print(line.strip())  # Remove trailing newline character
    # No need to close the file
```

The general form of the `with/as` statement is

with object-creation **as** object :

block

- The reserved word `with` begins the `with/as` statement.
- The expression *object-creation* attempts to create and return an object. If the *object-creation* expression is unsuccessful, the statement's execution does not continue.
- The reserved word `as` binds the object created by the *object-creation* expression to a variable.
- *object* is bound to the object created by the *object-creation* expression.
- *block* contains the code that uses the object bound to *object*.

The `with/as` statement can work with classes like `TextIOWrapper` that provide a particular protocol for initialization and finalization. In the case of our file object, if the call to open proceeds without an error, the program will execute the code in the `with/as` block. When the code in the `with/as` has finished executing, the statement executes any finalization actions the class requires. In this case, the finalization code of the `TextIOWrapper` class closes the file associated with file object `f`. Only certain classes support the initialization/finalization protocol in a way that is compatible with the `with/as` statement. Such classes provide a method named `__enter__` that performs the initialization and a method named `__exit__` that performs the finalization.

Listing 9.7 (`numbersaver.py`) allows the user to enter numbers from the keyboard and save them to a file. It also allows the user retrieve the values previously saved to a file. The user can specify the name of the file and, thus, work with multiple files.

Listing 9.7: `numbersaver.py`

```
'''
Uses Python's file class to store data to and retrieve data from
a text file.
'''
def load_data(filename):
    ''' Print the elements stored in the text file named filename. '''
    # Open file to read
    with open(filename) as f:    # f is a file object
        for line in f:          # Read each line as text
            print(int(line))    # Convert to integer and append to the list

def store_data(filename):
    ''' Allows the user to store data to the text file named filename. '''
    with open(filename, 'w') as f:    # f is a file object
        number = 0
        while number != 999:          # Loop until user provides magic number
            number = int(input('Please enter number (999 quits):'))
            if number != 999:
                f.write(str(number) + '\n') # Convert integer to string to save
            else:
                break                    # Exit loop

def main():
    ''' Interactive function that allows the user to
    create or consume files of numbers. '''
    done = False

    while not done:
```

```
cmd = input('Save Load Quit: ')
if cmd == 'S' or cmd == 's':
    store_data(input('Enter file name:'))
elif cmd == 'L' or cmd == 'l':
    load_data(input('Enter filename:'))
elif cmd == 'Q' or cmd == 'q':
    done = True

if __name__ == '__main__':
    main()
```

The following shows one particular run of Listing 9.7 (numbersaver.py):

```
Save Load Quit: s
Enter file name: numbers.txt
Please enter number (999 quits): 10
Please enter number (999 quits): 20
Please enter number (999 quits): 30
Please enter number (999 quits): 40
Please enter number (999 quits): 50
Please enter number (999 quits): 999
Save Load Quit: q
```

This run creates a file named numbers.txt. We can run the program again to retrieve the previously entered values:

```
Save Load Quit: l
Enter filename: numbers.txt
10
20
30
40
50
Save Load Quit: q
```

The literal name of Python's file class is `TextIOWrapper` from the `io` module. The kind of files processed by this file class are known as *text files*. Text files store character data, and we can use a simple editor to create and modify text files. Many applications prevent the easy modification of data files outside of the application by encoding the data in a special way. Depending on the data, the application also may encode the files to save space.

We can combine Python's string objects and file objects to create some powerful file processing programs. In particular we can open one file, read its contents, and write a modified form of its contents to a second file. Listing 9.8 (convertupper.py) is a module providing a simple utility function, `capitalize`, that capitalizes the text within a file.

Listing 9.8: convertupper.py

```
'''
convertupper.py
'''
def capitalize(filename):
    ''' Creates a new file with the prefix 'upper_'
```

```
        added to the name of the original file.
        All the alphabetic characters in the new
        are capitalized. This function does not
        disturb the contents of the original file. '''
with open(filename, 'r') as infile:
    with open('upper_' + filename, 'w') as outfile:
        for line in infile:
            line = line.strip().upper()
            print(line, file=outfile)
```

The capitalize function creates a new file, named with the upper_ prefix added to the name of the original file.

Suppose we have a text file named declaration.txt containing the introduction and preamble to the United States Declaration of Independence, as shown here:

When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume among the powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

We hold these truths to be self-evident, that all men are created equal, that they are endowed by their Creator with certain unalienable Rights, that among these are Life, Liberty and the pursuit of Happiness.--That to secure these rights, Governments are instituted among Men, deriving their just powers from the consent of the governed, --That whenever any Form of Government becomes destructive of these ends, it is the Right of the People to alter or to abolish it, and to institute new Government, laying its foundation on such principles and organizing its powers in such form, as to them shall seem most likely to effect their Safety and Happiness. Prudence, indeed, will dictate that Governments long established should not be changed for light and transient causes; and accordingly all experience hath shewn, that mankind are more disposed to suffer, while evils are sufferable, than to right themselves by abolishing the forms to which they are accustomed. But when a long train of abuses and usurpations, pursuing invariably the same Object evinces a design to reduce them under absolute Despotism, it is their right, it is their duty, to throw off such Government, and to provide new Guards for their future security.--Such has been the patient sufferance of these Colonies; and such is now the necessity which constrains them to alter their former Systems of Government. The history of the present King of Great Britain is a history of repeated injuries and usurpations, all having in direct object the establishment of an absolute Tyranny over these States. To prove this, let Facts be submitted to a candid world.

Listing 9.9 (runconvert.py) produces a new file named upper_declaration.txt.

Listing 9.9: runconvert.py

```
from convertupper import capitalize

capitalize('declaration.txt')
```

The file upper_declaration.txt will contain the following text:

WHEN IN THE COURSE OF HUMAN EVENTS, IT BECOMES NECESSARY FOR ONE PEOPLE TO DISSOLVE THE POLITICAL BANDS WHICH HAVE CONNECTED THEM WITH ANOTHER, AND TO ASSUME AMONG THE POWERS OF THE EARTH, THE SEPARATE AND EQUAL STATION TO WHICH THE LAWS OF NATURE AND

OF NATURE'S GOD ENTITLE THEM, A DECENT RESPECT TO THE OPINIONS OF MANKIND REQUIRES THAT THEY SHOULD DECLARE THE CAUSES WHICH IMPEL THEM TO THE SEPARATION.

WE HOLD THESE TRUTHS TO BE SELF-EVIDENT, THAT ALL MEN ARE CREATED EQUAL, THAT THEY ARE ENDOWED BY THEIR CREATOR WITH CERTAIN UNALIENABLE RIGHTS, THAT AMONG THESE ARE LIFE, LIBERTY AND THE PURSUIT OF HAPPINESS.--THAT TO SECURE THESE RIGHTS, GOVERNMENTS ARE INSTITUTED AMONG MEN, DERIVING THEIR JUST POWERS FROM THE CONSENT OF THE GOVERNED, --THAT WHENEVER ANY FORM OF GOVERNMENT BECOMES DESTRUCTIVE OF THESE ENDS, IT IS THE RIGHT OF THE PEOPLE TO ALTER OR TO ABOLISH IT, AND TO INSTITUTE NEW GOVERNMENT, LAYING ITS FOUNDATION ON SUCH PRINCIPLES AND ORGANIZING ITS POWERS IN SUCH FORM, AS TO THEM SHALL SEEM MOST LIKELY TO EFFECT THEIR SAFETY AND HAPPINESS. PRUDENCE, INDEED, WILL DICTATE THAT GOVERNMENTS LONG ESTABLISHED SHOULD NOT BE CHANGED FOR LIGHT AND TRANSIENT CAUSES; AND ACCORDINGLY ALL EXPERIENCE HATH SHEWN, THAT MANKIND ARE MORE DISPOSED TO SUFFER, WHILE EVILS ARE SUFFERABLE, THAN TO RIGHT THEMSELVES BY ABOLISHING THE FORMS TO WHICH THEY ARE ACCUSTOMED. BUT WHEN A LONG TRAIN OF ABUSES AND USURPATIONS, PURSUING INVARIABLY THE SAME OBJECT EVINCES A DESIGN TO REDUCE THEM UNDER ABSOLUTE DESPOTISM, IT IS THEIR RIGHT, IT IS THEIR DUTY, TO THROW OFF SUCH GOVERNMENT, AND TO PROVIDE NEW GUARDS FOR THEIR FUTURE SECURITY.--SUCH HAS BEEN THE PATIENT SUFFERANCE OF THESE COLONIES; AND SUCH IS NOW THE NECESSITY WHICH CONSTRAINS THEM TO ALTER THEIR FORMER SYSTEMS OF GOVERNMENT. THE HISTORY OF THE PRESENT KING OF GREAT BRITAIN IS A HISTORY OF REPEATED INJURIES AND USURPATIONS, ALL HAVING IN DIRECT OBJECT THE ESTABLISHMENT OF AN ABSOLUTE TYRANNY OVER THESE STATES. TO PROVE THIS, LET FACTS BE SUBMITTED TO A CANDID WORLD.

Table 9.2 summarizes some of the functions and methods available to file objects.

TextIOWrapper Methods	
open	A function that returns a file object (instance of <code>io.TextIOWrapper</code>).
read	A method that reads the contents of a text file into a single string.
write	A method that writes a string to a text file.
close	A method that closes the file from further processing. When writing to a file, the close method ensures that all data sent to the file is saved to the file.

Table 9.2: A few of the functions and methods available to file objects

Objects usually contain data in addition to methods. `TextIOWrapper` objects store integer, string, and Boolean information. The following interactive session reveals some of the data stored in file objects:

```
>>> f = open('temp.dat', 'w')
>>> f.name
'temp.dat'
>>> f._CHUNK_SIZE
8192
>>> f.mode
'w'
>>> f.encoding
'cp1252'
>>> f.line_buffering
False
```

We say that `name`, `_CHUNK_SIZE`, `encoding`, and `line_buffering` are all fields or variables of the object `f`. These are just like the variables we have been using, except that we must prefix their name with their associated object and a dot (`.`). Since these names refer to data, not methods, no parentheses appear at the end. If we have two different file objects, `f` and `g`, `f.name` may be different from `g.name`. The statement

```
x = 2
```

binds the variable `x` to the value 2, whereas the statement

```
obj.x = 2
```

binds the `x` field of object `obj` to the value 2.

9.4 Fraction Objects

The `fractions` module provides the `Fraction` class. Fraction objects model mathematical rational numbers; that is, the ratio of two integers. Rational numbers contain a *numerator* and *denominator*. Listing 9.10 (`fractionplay.py`) makes and uses some Fraction objects.

Listing 9.10: `fractionplay.py`

```
from fractions import Fraction

f1 = Fraction(3, 4)    # Make the fraction 3/4
print(f1)             # Print it
print(f1.numerator)   # Print numerator
print(f1.denominator) # Print denominator
print(float(f1))      # Floating-point equivalent
f2 = Fraction(1, 8)    # Make another fraction, 1/8
print(f2)             # Print the second fraction
f3 = f1 + f2           # Add the two fractions
print(f3)             # 3/4 + 1/8 = 6/8 + 1/8 = 7/8
```

Listing 9.10 (`fractionplay.py`) prints

```
3/4
3
4
0.75
1/8
7/8
```

The statement

```
f1 = Fraction(3, 4)
```

creates a `Fraction` object and assigns the variable `f1` to the object. The expression `Fraction(3, 4)` calls a *class constructor*. Class constructors allow clients to supply data used in the formation of a new object. In this case, the first parameter represents the numerator of the new fraction object, and the second parameter represents the denominator of the object. The `Fraction(3, 4)` expression returns a reference to the newly created fraction object, and the statement

```
f1 = Fraction(3, 4)
```

binds the variable `f1` to this object.

We see from Listing 9.10 (`fractionplay.py`) that `Fraction` objects contain `numerator` and `denominator` fields. The expression `f1.numerator` represents the `numerator` field of object `f1`.

The program appears to be devoid of any method calls, but the addition statement involves a method call behind the scenes. Python reserves special names for some methods. The `Fraction` class provides a method named `__add__`. This allows clients to add together two fraction objects as shown in Listing 9.10 (`fractionplay.py`):

```
f3 = f1 + f2
```

This is a nicer way of expressing

```
f3 = f1.__add__(f2)
```

We say the former statement using the `+` operator is *syntactic sugar* for latter statement that uses the explicit `__add__` method. Most human readers prefer the version with `+`, but, in reality, both statements are identical to the Python interpreter. The `str` class of string objects also provides a `__add__` method. If `s` and `t` are string objects, the string concatenation expression `s + t` is syntactic sugar for `s.__add__(t)`.

The `Fraction` class includes a number of these special methods that exploit syntactic sugar; examples include the following (`f` and `g` reference `Fraction` objects):

- `__mult__`, multiplication: `f.__mul__(g)` is equivalent to `f * g`
- `__eq__`, relational quality: `f.__eq__(g)` is equivalent to `f == g`
- `__gt__`, greater than: `f.__gt__(g)` is equivalent to `f > g`
- `__sub__`, subtraction: `f.__sub__(g)` is equivalent to `f - g`
- `__neg__`, unary minus: `f.__neg__()` is equivalent to `-f`

9.5 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 create a `Turtle` object and issue it commands. We must import the `turtle` module to have access to `Turtle` objects. Listing 9.11 (`boxturtle.py`) draws a rectangular box with a `Turtle` object.

Listing 9.11: `boxturtle.py`

```
''' Draws a rectangular box in the window '''

from turtle import Turtle, mainloop

t = Turtle()          # Create a turtle object named t
t.pencolor('red')     # t's pen color is red
t.forward(200)        # Move turtle t forward 200 units (create bottom of rectangle)
t.left(90)            # Turn turtle left by 90 degrees
t.pencolor('blue')    # Change t's pen color to blue
t.forward(150)        # Move turtle t forward 150 units (create right wall)
t.left(90)            # Turn turtle left by 90 degrees
t.pencolor('green')   # Change t's pen color to green
t.forward(200)        # Move turtle t forward 200 units (create top)
t.left(90)            # Turn turtle left by 90 degrees
t.pencolor('black')   # Change t's pen color to black
t.forward(150)        # Move turtle t forward 150 units (create left wall)
t.hideturtle()        # Make turtle t invisible
mainloop()            # Await user input
```

Figure 9.1 shows the result of running Listing 9.11 (boxturtle.py). By default, the turtle starts at the center

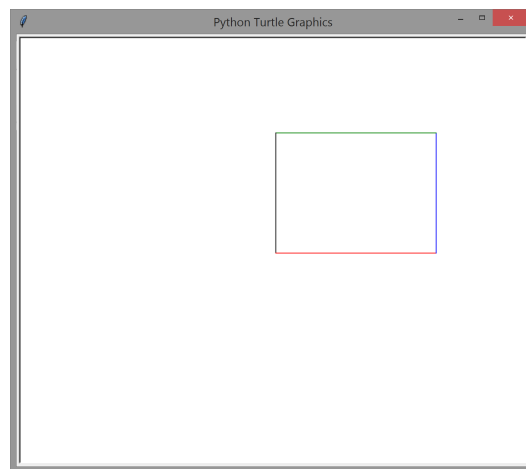


Figure 9.1: A very simple drawing made with Turtle graphics

of the window facing to the right. Listing 9.11 (boxturtle.py) reveals a few of the methods provided by the Turtle class:

- `pencolor`: sets turtle's current drawing color.
- `forward`: moves the turtle forward the specified number of units
- `left`: turns the turtle to the left by an angle specified in degrees.
- `hideturtle`: makes the turtle (not its drawing) invisible.

Listing 9.12 (octagon.py) draws in the display window a blue spiral within a red octagon.

Listing 9.12: octogon.py

```
''' Draws in the window a spiral surrounded with an octogon '''

from turtle import *

def octogon(t, x, y, color):
    ''' Draws with turtle t an octogon centered at (x, y)
        with the specified color '''
    t.pencolor(color) # Set pen color
    t.penup()         # Lift pen to move it
    t.setposition(x, y) # Move the pen to coordinates (x, y)
    t.pendown()        # Place pen to begin drawing
    for i in range(8): # Draw the eight sides
        t.forward(80)
        t.right(45)

def spiral(t, x, y, color):
    ''' Draws with turtle t a spiral centered at (x, y)
        with the specified color '''
    distance = 0.2
    angle = 40
    t.pencolor(color) # Set pen color
    t.penup()         # Left pen to move it
    t.setposition(x, y) # Position the pen at coordinates (x, y)
    t.pendown()        # Set pen down to begin drawing
    for i in range(100):
        t.forward(distance)
        t.left(angle)
        distance += 0.5

t = Turtle() # Create a turtle object named t
octogon(t, -45, 100, 'red')
spiral(t, 0, 0, 'blue')
t.hideturtle() # Make turtle t invisible
done()
```

Listing 9.12 (octogon.py) uses the `penup`, `setposition`, and `pendown` methods of the `Turtle` class 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 9.2 shows the result of running Listing 9.12 (octogon.py).

9.6 Graphics with tkinter Objects

The `tkinter` module provides classes for building graphical user interfaces via the cross-platform Tk toolkit. The `tkinter` module is much larger and more complex than the `turtle` module. We have several more Python concepts to explore before we can exploit all of the GUI building features that the `tkinter` module provides. In the meantime, Listing 9.13 (tkinterlight.py) provides a fully functioning interactive program that models a traffic light. The light changes from red to green to yellow to red . . . , when the user presses the graphical button labeled *Change*.

Listing 9.13: tkinterlight.py

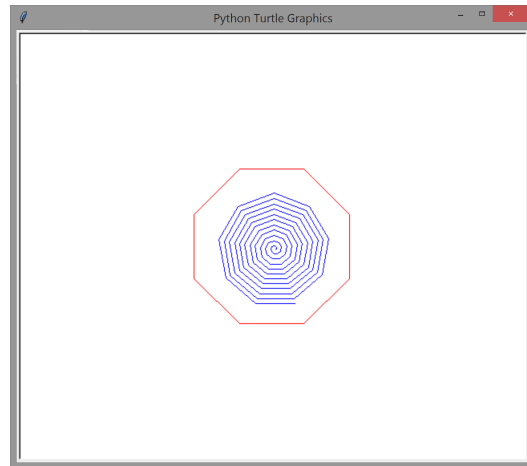


Figure 9.2: More Turtle graphics fun: a spiral within an octagon

```

from tkinter import Tk, Canvas, Button

# Global variables
color = 'red'    # The light's current color
root = Tk()      # Create the main window
# Create a drawing surface within the window
canvas = Canvas(root, width=400, height=300)
canvas.pack()

# Traffic light frame
canvas.create_rectangle(150, 20, 250, 280, fill='gray')
# Red lamp
red_lamp = canvas.create_oval(170, 40, 230, 100, fill='red')
# Yellow lamp
yellow_lamp = canvas.create_oval(170, 120, 230, 180, fill='black')
# Green lamp
green_lamp = canvas.create_oval(170, 200, 230, 260, fill='black')

def do_button_press():
    ''' The window manager calls this function when the user
        presses the graphical button. '''
    global color
    if color == 'red':
        color = 'green'
        canvas.itemconfigure(red_lamp, fill='black')    # Turn red off
        canvas.itemconfigure(green_lamp, fill='green')  # Turn green on
    elif color == 'green':
        color = 'yellow'
        canvas.itemconfigure(green_lamp, fill='black')  # Turn green off
        canvas.itemconfigure(yellow_lamp, fill='yellow') # Turn yellow on
    elif color == 'yellow':
        color = 'red'

```

```
        canvas.itemconfigure(yellow_lamp, fill='black') # Turn yellow off
        canvas.itemconfigure(red_lamp, fill='red')      # Turn red on

# Create a graphical button and ensure it calls the do_button_press
# function when the user presses it
button = Button(canvas, text='Change', command=do_button_press)
# Position button
button.place(x=10, y=100)

# Start the GUI event loop
root.mainloop()
```

Figure 9.3 shows the result of running Listing 9.13 (tkinterlight.py).

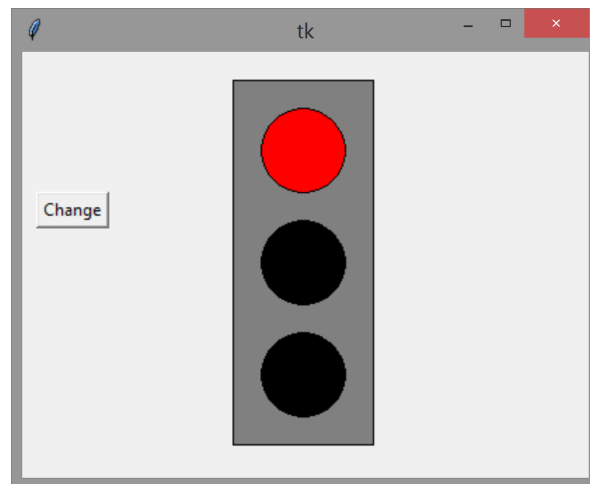


Figure 9.3: An interactive graphical traffic light. The user presses the *Change* button to cycle the traffic signals.

Listing 9.13 (tkinterlight.py) uses three classes from tkinter:

- Tk: This class represents a graphical window. The statement

```
root = Tk()
```

creates a Tk object named `root`. The `root` object corresponds to the application's main graphical window. The statement

```
root.mainloop()
```

calls the `mainloop` method on behalf of the `root` window object to start the graphical program. This method starts the process in motion that enables the user to provide input to the application and allows the application to provide visual feedback to the user.

- Canvas: This class represents a drawing area within a graphical window. The statement

```
canvas = Canvas(root, width=400, height=300)
```

creates a Canvas object named `canvas` that is associated with the root window. The canvas objects dimensions are 400 pixels wide and 300 pixels tall. The statement

```
canvas.pack()
```

makes the canvas object fill the graphical window. The statement

```
canvas.create_rectangle(150, 20, 250, 280, fill='gray')
```

invokes the `create_rectangle` method to add a gray rectangle of the given size to the canvas. This rectangle represents the traffic light's frame that holds the lamps. The `create_oval` methods work similarly for creating the circles representing the lamps of the traffic light. The statement

```
canvas.itemconfigure(red_lamp, fill='black')
```

calls the `itemconfigure` method to change the color of the circle named `red_lamp`.

- **Button:** This class represents a graphical button the user can press. The statement

```
button = Button(canvas, text='Change', command=do_button_press)
```

creates a Button object named `button` attached to the canvas object. The button is labeled *Change*, and it reacts to a user's press by calling the `do_button_press` function. The statement

```
button.place(x=10, y=100)
```

positions the button at the specified location within the canvas.

Note that in the `do_button_press` function of Listing 9.13 (`tkinterlight.py`) we must declare that `color` is a global variable because the function reassigns it. Without the `global` declaration `do_button_press` would treat `color` as a local variable. Since the code within `do_button_press` does not assign the canvas variable, `canvas` is implicitly global within the function.

9.7 Other Standard Python Objects

Python provides many other standard classes. Among these, lists, tuples, dictionaries, and sets are particularly useful. These classes are so general, useful, and widely-used that we devote the next few chapters to exploring them in detail.

9.8 Object Mutability and Aliasing

Recall that a variable is a name that labels an object. We informally have treated a variable as the object it represents; for example, given the following statement

```
frac1 = Fraction(1, 2)
```

we often refer to the object `frac1`. In fact, the object is the `Fraction` object representing the rational number $\frac{1}{2}$, and `frac1` is merely a name by which we can access the `Fraction` object. This informality has not been a problem so far, but as we explore objects more deeply we need to be more careful in how we talk about and use objects.

Consider Listing 9.14 (`fractionassign.py`) which uses `Fraction` variables.

Listing 9.14: fractionassign.py

```
from fractions import Fraction

# Assign some Fraction variables
f1 = Fraction(1, 2)
f2 = Fraction(1, 2)
f3 = f1

# Examine the objects involved
print('f1 =', f1)
print('f2 =', f2)
print('f3 =', f3)

# Examine the numerators and denominators separately
print('f1 numerator, denominator:', f1.numerator, f1.denominator)
print('f2 numerator, denominator:', f2.numerator, f2.denominator)
print('f3 numerator, denominator:', f3.numerator, f3.denominator)

# Compare the fractions
print('f1 == f2?', f1 == f2)
print('f1 == f3?', f1 == f3)
print('f1 is f2?', f1 is f2)
print('f1 is f3?', f1 is f3)
```

Listing 9.14 (fractionassign.py) prints

```
f1 = 1/2
f2 = 1/2
f3 = 1/2
f1 numerator, denominator: 1 2
f2 numerator, denominator: 1 2
f3 numerator, denominator: 1 2
f1 == f2? True
f1 == f3? True
f1 is f2? False
f1 is f3? True
```

To better understand the behavior of Listing 9.14 (fractionassign.py) we will examine each of its parts in detail. The statement

```
f1 = Fraction(1, 2)
```

calls the `Fraction` class constructor which creates a new `Fraction` object with its `numerator` field set to 1 and its `denominator` field set to 2. It assigns the variable `f1` to this new fraction object. The statement

```
f2 = Fraction(1, 2)
```

creates a new `Fraction` object with its `numerator` field set to 1 and its `denominator` field set to 2. It assigns the variable `f2` to this fraction object. The statement

```
f3 = f1
```

assigns the variable `f3` to the same fraction object to which `f1` is assigned. Note that since the statement does not involve the `Fraction` class constructor, it does not create a new fraction object. At this point we

have two Fraction objects and three variables bound to Fraction objects. Figure 9.4 illustrates the relationships amongst the Fraction objects and variables that result from these three assignment statements. The variables `f1` and `f3` refer to the same object. We say that `f1` *aliases* `f3`. Said another way, the variables

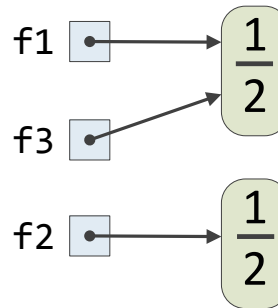


Figure 9.4: An illustration of the relationships amongst the fraction objects and variables resulting from the assignment statements in Listing 9.14 (`fractionassign.py`).

`f1` and `f3` are aliases. The Fraction class implements a method named `__eq__` which enables clients to compare two fraction objects for logical equality using the familiar `==` operator. The statements

```
print('f1 == f2?', f1 == f2)
print('f1 == f3?', f1 == f3)
```

reveal that all three variables refer to objects that are logically equal to each other. The Fraction.`__eq__` method (`==`) compares the numerator and denominator fields of two fraction objects to determine equality.

Sometimes logical equality is not sufficient, and we may need to know if two variables refer to the same object. Python's `is` operator tests to see if two variables refer to the same object. The statements

```
print('f1 is f2?', f1 is f2)
print('f1 is f3?', f1 is f3)
```

show that the variables `f1` and `f2` reference two different objects, but `f1` and `f3` refer to the same object. This proves that `f1` and `f3` are aliases. Python also has an `id` function that returns an integer that is unique to a particular object. (For most Python implementations this number is the starting address in memory where the executing program has placed the object.) If `a` and `b` are objects, `a is b` is true exactly when `id(a) == id(b)`.

Prior to this chapter we have restricted our attention to the classes `int`, `float`, `str`, and `bool`. Object aliasing has no practical consequences for programmers restricted to these data types. Instances of these classes are all immutable objects, which means an object of any these of these types cannot change its state after its creation. The integer 3 always is 3, for example, and the string object `'Fred'` cannot change to `'Free'`. Instances of the Fraction class are immutable also.

Aliasing can be an issue for mutable objects. In Python's Turtle graphics library (Section 9.5), Turtle objects are mutable. Programmers can move a turtle object, change its orientation, and change its pen color. Each of these actions changes the state of the turtle and affects the way the turtle draws within the graphics window.

Listing 9.15 (`multiturtle.py`) uses two Turtle objects.

Listing 9.15: multiturtle.py

```
''' Uses two Turtle objects to draw on the screen '''

from turtle import *

# Part 1
t1 = Turtle()      # Create a turtle object named t1
t2 = Turtle()      # Create a second turtle object named t2
t1.pencolor('red')  # t1's pen color is red
t2.pencolor('blue') # t2's pen color is blue
t2.left(90)        # Point t2 up (t1 autoatically points to the right)
t1.forward(100)    # Move turtle t1 forward 100 units
t2.forward(50)     # Move turtle t2 forward 50 units

# Part 2
t2 = t1            # Make the second turtle just like the first?
t2.right(45)       # Turn turtle 2 (but not turtle 1?)
t2.forward(50)     # Move turtle 2 (why does turtle1 move instead?)

done()
```

The first part of Listing 9.15 (multiturtle.py) draws the picture shown in Figure 9.5. The beginning of Part 2



Figure 9.5: The drawing that results from executing the first part of Listing 9.15 (multiturtle.py).

of Listing 9.15 (multiturtle.py) reassigns `t2` to `t1`. At this point turtles `t1` and `t2` are aliases, and any further actions taken either through `t1` or `t2` affect only one of the turtles. Figure 9.6 shows the final results of Listing 9.15 (multiturtle.py). The turning and moving of `t2` does not affect the original turtle 2. Instead,

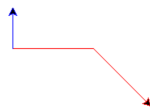


Figure 9.6: The final drawing rendered by Listing 9.15 (multiturtle.py).

changing `t2` affects the `t1`'s turtle, as `t2` and `t1` refer to the same `Turtle` object.

Aliasing of mutable types can be a problem for beginning programmers because it is easy to believe the statement

```
t2 = t1
```

makes `t2` a copy of `t1` and that they remain distinct objects. If the situation arises where your program is managing what you believe to be similar but separate objects and changing the characteristics of one object unexpectedly changes one or more of the other objects in exactly the same way, you likely have an unintended aliasing problem.

9.9 Garbage Collection

The following statement:

```
f = Fraction(2, 3)
```

creates a `Fraction` object representing $\frac{2}{3}$ and assigns the variable `f` to the object. The Python interpreter acts on this statement by reserving enough space in the computer's memory to hold the object. It also performs any initialization that the object requires, in this case it sets `f.numerator` to 2 and `f.denominator` to 3.

What happens to the object associated with `f` when we reassign the variable `f`? The following statement:

```
# f is the Fraction object created above
f = None
```

redirects `f` to refer to the special object `None`, which represents no object. At this point *no* variable references the $\frac{2}{3}$ fraction object created earlier. This means the object effectively is cut off from the remainder of the program's execution or the remainder of the interactive session. This abandoned object is classified as *garbage*. The term *garbage* is a technical term used in computer science that refers to memory allocated by an executing program that the program no longer can access. The Python interpreter cleans up garbage through a process called *garbage collection*. Python uses a reference counting garbage collector that automatically reclaims the space occupied by abandoned objects.

Reference counting garbage collection works as follows. All objects have an associated reference count. When the executing program creates a new object and assigns a variable to it, it sets the object's reference count to 1. The following statement:

```
f = Fraction(2, 3) # The 2/3 object initially has a reference count of 1
```

A reference count of one means that exactly one variable is assigned to the object. Making an alias, as in

```
g = f # The 2/3 object now has a reference count of 1 + 1 = 2
```

increments the $\frac{2}{3}$ object's reference count by one. If we make another alias, as in

```
h = f # The 2/3 object now has a reference count of 2 + 1 = 3
```

the assignment statement increases the object's reference count to 3. If reassign `f`, `g`, or `h`, as in

```
g = Fraction(9/10) # Reassign a new object to g
```

the reference count of the $\frac{2}{3}$ object decreases by one (and sets the $\frac{9}{10}$ object's reference count to 1). If we reassign `f`:

```
f = None
```

this leaves only variable `h` referencing the $\frac{2}{3}$, so the object's reference count is 1. If we finally reassign `h`:

```
h = 15
```

the $\frac{2}{3}$ object's reference count drops to zero. An object with a reference count of zero is garbage, and the garbage collector will automatically reclaim the space held by the object so it can be recycled and used elsewhere.

9.10 Summary

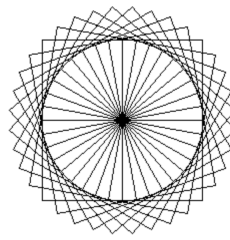
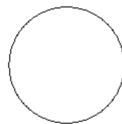
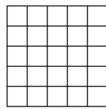
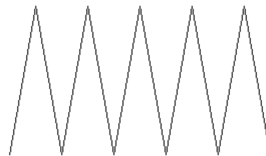
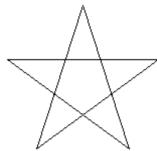
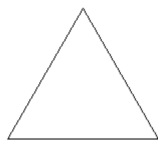
- An object is an instance of a class.
- The terms *class* and *type* are synonymous.
- Integers, floating-point numbers, strings, lists, and functions are examples of objects we have seen in earlier chapters.
- Typically objects are a combination of data (attributes or fields) and methods (operations)
- An object's data and methods constitute the object's members.
- The code that uses the services provided by an object is known as the client of the object.
- Methods are like functions associated with a class of objects.
- Members that begin and end with two underscores `__` are meant for internal use by objects; clients usually do not use these members directly.
- Methods are called (or invoked) on behalf of objects or classes.
- The dot (`.`) operator associates an object or class with a member.
- Clients may not call a method without its associated object or class.
- The `str` class represents string objects.
- String objects are immutable. You may reassign a variable to another string object, but you may not modify the contents of an existing string object. This means no `str` method may alter an existing string object. When client code wishes to achieve the effect of modifying a string via one of the string's methods, the client code must reassign its variable with the result passed back by the method.
- The `str` class contains a number methods useful for manipulating strings.
- The `open` function returns a file object that enables programs to store data to, and retrieve data from, text files on disk.
- The standard `Fraction` class in the `fractions` package model mathematical rational numbers.
- The `Fraction` class provides a number of special methods like `__add__`, `__mul__`, and `__sub__` that allow programmers to use the `+`, `*`, and `-` operators with `Fraction` objects.
- The `with/as` statement creates an execution context for an object that manages the object's finalization.
- The `turtle` package provides classes such as `Turtle` for drawing pictures with simple commands.
- The `tkinter` package provides classes such as `Canvas` and `Button` with which programmers can build applications with graphical user interfaces.

- The classes `int`, `float`, `bool`, `str`, and `Fraction` represent immutable types.
- Class constructors create new objects; simple assignment of one variable to another creates an alias.
- To the unwary programmer, aliases can produce problems for mutable types.
- Objects that become inaccessible from an executing program or interactive session are known as garbage.
- The interpreter keeps track of the reference count for every object. The reference count represents the number of variables assigned to the object.
- Most implementations of Python use a reference counting garbage collector.
- The interpreter performs garbage collection automatically, and the process is invisible to the programmer and program user.

9.11 Exercises

CAUTION! SECTION UNDER CONSTRUCTION

1. For each of the drawings below write a program that draws the shape using Python's Turtle graphics module



Chapter 10

Lists

The variables we have used to this point can bind to only one object at a time. As we have seen, we can use individual variables to create some interesting and useful programs; however, variables that can represent only one value at a time do have their limitations. Consider Listing 10.1 (averagenumbers.py) which averages five numbers entered by the user.

Listing 10.1: averagenumbers.py

```
def main():
    print("Please enter five numbers: ")
    # Allow the user to enter in the five values.
    n1 = eval(input("Please enter number 1: "))
    n2 = eval(input("Please enter number 2: "))
    n3 = eval(input("Please enter number 3: "))
    n4 = eval(input("Please enter number 4: "))
    n5 = eval(input("Please enter number 5: "))
    print("Numbers entered:", n1, n2, n3, n4, n5)
    print("Average:", (n1 + n2 + n3 + n4 + n5)/5)

main()
```

A sample run of Listing 10.1 (averagenumbers.py) looks like:

```
Please enter five numbers:
Please enter number 1: 34.2
Please enter number 2: 10.4
Please enter number 3: 18.0
Please enter number 4: 29.3
Please enter number 5: 15.1
Numbers entered: 34.2 10.4 18.0 29.3 15.1
Average: 21.4
```

The program conveniently displays the values the user entered and then computes and displays their average.

Suppose the number of values to average must increase from five to 25. If we use Listing 10.1 (averagenumbers.py) as a guide, we would need to introduce twenty additional variables, and the over-

all length of the program necessarily would grow. Averaging 1,000 numbers using this approach would be impractical.

Listing 10.2 (averagenumbers2.py) provides an alternative approach for averaging numbers that uses a loop.

Listing 10.2: averagenumbers2.py

```
def main():
    sum = 0.0
    NUMBER_OF_ENTRIES = 5
    print("Please enter", NUMBER_OF_ENTRIES, " numbers: ")
    for i in range(0, NUMBER_OF_ENTRIES):
        num = eval(input("Enter number " + str(i) + ": "))
        sum += num
    print("Average:", sum/NUMBER_OF_ENTRIES)

main()
```

Listing 10.2 (averagenumbers2.py) behaves slightly differently from Listing 10.1 (averagenumbers.py), as the following sample run using the same data shows:

```
Please enter 5 numbers:
Enter number 0: 34.2
Enter number 1: 10.4
Enter number 2: 18.0
Enter number 3: 29.3
Enter number 4: 15.1
Average: 21.4
```

We can modify Listing 10.2 (averagenumbers2.py) to average 25 values much more easily than Listing 10.1 (averagenumbers.py) that must use 25 separate variables—just change the value of `NUMBER_OF_ENTRIES`. In fact, the coding change to average 1,000 numbers is no more difficult. However, unlike the original average program, this new version does not at the end display all the numbers entered. This is a significant difference; it may be necessary to retain all the values entered for various reasons:

- All the values can be redisplayed after entry so the user can visually verify their correctness.
- The values may need to be displayed in some creative way; for example, they may be placed in a graphical user interface component, like a visual grid (spreadsheet).
- The values entered may need to be processed in a different way after they are all entered; for example, we may wish to display just the values entered above a certain value (like greater than zero), but the limit is not determined until after all the numbers are entered.

In all of these situations we must retain all the values for future recall.

We need to combine the advantages of both of the above programs; specifically we want

- the ability to retain individual values, and
- the ability to dispense with creating individual variables to store all the individual values

These may seem like contradictory requirements, but Python provides a standard data structure that simultaneously provides both of these advantages—the list.

10.1 Using Lists

A list is an object that holds a collection of objects; it represents a sequence of data. In that sense, a list is similar to a string, except a string can hold only characters. A list can hold any Python object. A list need not be homogeneous; that is, the elements of a list do not all have to be of the same type.

Like any other variable, a list variable can be local or global, and it must be defined (assigned) before its use. The following code fragment defines a list named `lst` that holds the integer values 2, −3, 0, 4, −1:

```
lst = [2, -3, 0, 4, -1]
```

The right-hand side of the assignment statement is a literal list. The elements of the list appear within square brackets (`[]`), and commas separate the elements. The following statement assigns the empty list to a variable named `a`:

```
a = []
```

We can print list literals and lists referenced through variables:

```
lst = [2, -3, 0, 4, -1]  # Assign the list
print([2, -3, 0, 4, -1]) # Print a literal list
print(lst)               # Print a list via a variable
```

The above code prints

```
[2, -3, 0, 4, -1]
[2, -3, 0, 4, -1]
```

We may access the elements contained in a list via their position within the list. We access individual elements of a list using square brackets:

```
lst = [2, -3, 0, 4, -1]  # Assign the list
lst[0] = 5               # Make the first element 5
print(lst[1])            # Print the second element
lst[4] = 12              # Make the last element 12
print(lst)               # Print a list variable
print([10, 20, 30][1])   # Print second element of literal list
```

This code prints

```
-3
[5, -3, 0, 4, 12]
20
```

The number within the square brackets is called the *index*. A non-negative index indicates the distance from the beginning of the list. The expression `lst[0]` therefore indicates the element at the very beginning (a distance of zero from the beginning) of `lst`, and `lst[1]` is the second element (a distance of one away from the beginning). We can read aloud the expression `a[3]` as “a sub three,” where the index 3 represents a *subscript*. The subscript terminology is borrowed from mathematicians who use subscripts to reference elements in a mathematical vector or matrix; for example, V_2 represents the second element in vector V . Unlike the convention often used in mathematics, however, the first element in a list is at position *zero*, not one. As mentioned above, the index indicates the distance from the beginning; thus, the very first element is at a distance of zero from the beginning of the list. The first element of list `a` is `a[0]`. As a consequence of a zero beginning index, if list `a` holds n elements, the last element in `a` is `a[n − 1]`, not `a[n]`.

If a is a list with n elements, and i is an integer such that $0 \leq i < n$, then $a[i]$ is an element in the list.

A negative list index represents a negative offset from an imaginary element one past the end of the list. For list a , the expression $a[-1]$ represents the last element in a . The expression $a[-2]$ represents the next to the last element, and so forth. If a contains n elements, the expression $a[0]$ corresponds to $lst[-n]$. Listing 10.3 (`negindex.py`) illustrates the use of negative indices to print a list in reverse.

Listing 10.3: `negindex.py`

```
def main():
    data = [10, 20, 30, 40, 50, 60]

    # Print the individual elements with negative indices
    print(data[-1])
    print(data[-2])
    print(data[-3])
    print(data[-4])
    print(data[-5])
    print(data[-6])

main() # Execute main
```

Listing 10.3 (`negindex.py`) prints

```
60
50
40
30
20
10
```

Figure 10.1 visualizes the list assigned as

```
lst = [5, -3, 12]
```

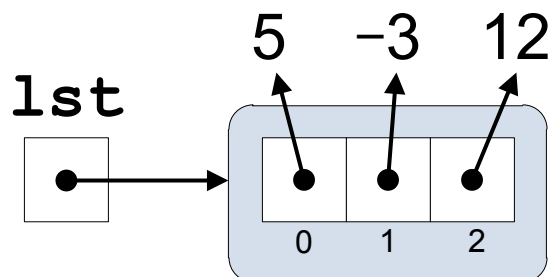


Figure 10.1: A simple list with three elements. The small number below a list element represents the index of that element.

Listing 10.4 (heterolist.py) demonstrates that lists may be heterogeneous; that is, a list can hold elements of varying types.

Listing 10.4: heterolist.py

```
collection = [24.2, 4, 'word', eval, 19, -0.03, 'end']
print(collection[0])
print(collection[1])
print(collection[2])
print(collection[3])
print(collection[4])
print(collection[5])
print(collection[6])
print(collection)
```

Listing 10.4 (heterolist.py) prints

```
24.2
4
word
<built-in function eval>
19
-0.03
end
[24.2, 4, 'word', <built-in function eval>, 19, -0.03, 'end']
```

We clearly see that a single list can hold integers, floating-point numbers, strings, and even functions. A list can hold other lists; the following code

```
col = [23, [9.3, 11.2, 99.0], [23], [], 4, [0, 0]]
print(col)
```

prints

```
[23, [9.3, 11.2, 99.0], [23], [], 4, [0, 0]]
```

Four of the elements of the list `col` are themselves lists.

The following interactive sequence shows how placing variables in a list actually copies the variable's values into the list:

```
>>> x = 5
>>> y = 'ABC'
>>> z = [x, y]
>>> seq = [x, y, z]
>>> seq
[5, 'ABC', [5, 'ABC']]
>>> x = 0
>>> y = 10
>>> seq
[5, 'ABC', [5, 'ABC']]
```

As this sequence demonstrates, changing the variable `x` does not affect the list built from `x`'s original value.

We can treat the elements of a list we access via `[]` as any other variable; for example,

```

nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
# Print the fourth element
print(nums[3])
# Make the third element the average of two other elements
nums[2] = (nums[0] + nums[9])/2;
# Assign elements at indices 1 and 4 from user input
# using tuple assignment
nums[1], nums[4] = eval(input("Enter a, b: "))

```

The fact that we can write an assignment statement such as

```
lst[4] = 12
```

indicates that lists are *mutable* objects. In contrast, strings are immutable objects, so it is not possible to modify the contents of a string object. We can, however, use the `[]` subscripting operator to access the individual characters (actually strings of length one) that comprise a string; for example,

```

>>> s = 'ABCEFGHI'
>>> s[0]
'A'
>>> s[1]
'B'

```

We know that string objects are immutable, so the following code generates an exception:

```

>>> s = 'ABCEFGHI'
>>> s[0] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

```

When an expression involving the subscripting operator appears on the side of the assignment operator, as in

```
lst[0] = 4
```

the interpreter calls the object's `__setitem__` method. An equivalent statement is

```
lst.__setitem__(0, 4)
```

The `str` class has no `__setitem__` method. All other situations that use the `[]` operator within an expression that do not attempt to modify the associated object correspond to the object's `__getitem__` method. For example, the following two statements are equivalent:

```

# The following two statements are equivalent:
x = lst[3]
x = lst.__getitem__(3)

```

String and list objects both have a `__getitem__` method. Programmers, of course, ordinarily should use the syntactically sugared `[]` expressions and not invoke directly an object's `__setitem__` and `__getitem__` methods.

The expression within `[]` must evaluate to an integer; some examples include

- an integer literal: `a[34]`

- an integer variable: `a[x]` (`x` must be an integer)
- an integer arithmetic expression: `a[x + 3]` (`x` must be an integer)
- an integer result of a function call that returns an integer: `a[max(x, y)]` (`max` must return an integer when called with `x` and `y`)
- an element of another or the same list: `a[b[3]]` (element `b[3]` must be an integer)

10.2 List Traversal

The action of moving through a list visiting each element is known as *traversal*. A list is a kind of iterable object, so we can use a `for` loop to visit each element in order within a list. Listing 10.5 (`heterolistfor.py`) uses a `for` loop and behaves identically to Listing 10.4 (`heterolist.py`).

Listing 10.5: `heterolistfor.py`

```
collection = [24.2, 4, 'word', eval, 19, -0.03, 'end']
for item in collection:
    print(item)      # Print each element
```

The built-in function `len` returns the number of elements in a list: The code segment

```
print(len([2, 4, 6, 8]))
a = [10, 20, 30]
print(len(a))
```

prints

```
4
3
```

The name `len` stands for *length*. The index of the last element in list `lst` is `lst[len(lst) - 1]`.

If you have some experience in other programming languages, you may be tempted for use `len` and an explicit list index with a `for` loop as shown in Listing 10.6 (`heterolistforindex.py`). to Listing 10.5 (`heterolistfor.py`).

Listing 10.6: `heterolistforindex.py`

```
collection = [24.2, 4, 'word', eval, 19, -0.03, 'end']
for i in range(len(collection)):    # Not the preferred way to traverse a list
    print(collection[i])           # Print each element
```

Listing 10.6 (`heterolistforindex.py`) works identically to Listing 10.5 (`heterolistfor.py`), but Listing 10.5 (`heterolistfor.py`) demonstrates the preferred way to iterate over the elements of a list. Listing 10.5 (`heterolistfor.py`) exploits a list object's iterable property and it more efficient and elegant than the technique used in Listing 10.6 (`heterolistforindex.py`).

How could we print the elements in a list in reverse order? We could use our familiar `for i in range...` construct and use an explicit index to count backwards:

```
nums = [2, 4, 6, 8]
# Print last element to first (zero index) element
for i in range(len(nums) - 1, -1, -1):
    print(nums[i])
```

This fragment prints

```
8
6
4
2
```

An even better way to iterate over list elements from back to front uses Python's built-in function named `reversed`:

```
nums = [2, 4, 6, 8]
# Print last element to first (zero index) element
for i in reversed(nums):
    print(nums[i])
```

Not only is this version shorter, it actually is more efficient than the version that uses `range` and `len`. The `reversed` expression creates an iterable object that enables the `for` statement to traverse the elements of the list in reverse. The expression `reversed(nums)` does not affect the contents of the list `nums`; it simply enables a backwards traversal of the elements.

10.3 Building Lists

Python supports several other ways of building a list besides enumerating all the list's elements in a list literal. We can construct a new list from two existing lists using concatenation. The plus (+) operator concatenates lists in the same way it concatenates strings. The following shows some experiments in the interactive shell with list concatenation:

```
>>> a = [2, 4, 6, 8]
>>> a
[2, 4, 6, 8]
>>> a + [1, 3, 5]
[2, 4, 6, 8, 1, 3, 5]
>>> a
[2, 4, 6, 8]
>>> a = a + [1, 3, 5]
>>> a
[2, 4, 6, 8, 1, 3, 5]
>>> a += [10]
>>> a
[2, 4, 6, 8, 1, 3, 5, 10]
>>> a += 20
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    a += 20
TypeError: 'int' object is not iterable
```

The statement

```
a = [2, 4, 6, 8]
```

assigns the given list literal to the variable `a`. The expression

```
a + [1, 3, 5]
```

evaluates to the list `[2, 4, 6, 8, 1, 3, 5]`, but the statement does not change the list to which `a` refers. The statement

```
a = a + [1, 3, 5]
```

actually reassigns `a` to the new list `[2, 4, 6, 8, 1, 3, 5]`. The statement

```
a += [10]
```

updates `a` to be the new list `[2, 4, 6, 8, 1, 3, 5, 10]`. Observe that the `+` will concatenate two lists, but it cannot join a list and a non-list. The following statement

```
a += 20
```

is illegal since `a` refers to a list, and `20` is an integer, not a list. If used within a program under these conditions, this statement will produce a run-time exception.

If we wish to append a variable's value to a list, we similarly must first enclose it within square brackets:

```
>>> x = 2
>>> a = [0, 1]
>>> a += [x]
>>> a
[0, 1, 2]
```

Listing 10.7 (`builduserlist.py`) shows how to build lists as the program executes.

Listing 10.7: `builduserlist.py`

```
# Build a custom list of non-negative integers specified by the user

def make_list():
    """
    Builds a list from input provided by the user.
    """
    result = []      # List to return is initially empty
    in_val = 0       # Ensure loop is entered at least once
    while in_val >= 0:
        in_val = int(input("Enter integer (-1 quits): "))
        if in_val >= 0:
            result += [in_val]  # Add item to list
    return result

def main():
    col = make_list()
    print(col)

main()
```

A sample run of Listing 10.7 (`builduserlist.py`) produces

```

Enter integer (-1 quits): 23
Enter integer (-1 quits): 100
Enter integer (-1 quits): 44
Enter integer (-1 quits): 19
Enter integer (-1 quits): 19
Enter integer (-1 quits): 101
Enter integer (-1 quits): 98
Enter integer (-1 quits): -1
[23, 100, 44, 19, 19, 101, 98]

```

If the user enters a negative number initially, we get:

```

Enter integer (-1 quits): -1
[]

```

There are several ways to build a list without explicitly listing every element in the list. We can use range to produce a regular sequence of integers. The range object returned by range is not itself a list, but we can make a list from a range using the list function, as Listing 10.8 (makeintegerlists.py) demonstrates.

Listing 10.8: makeintegerlists.py

```

def main():
    a = list(range(0, 10))
    print(a)
    a = list(range(10, -1, -1))
    print(a)
    a = list(range(0, 100, 10))
    print(a)
    a = list(range(-5, 6))
    print(a)

main()

```

Listing 10.8 (makeintegerlists.py) prints

```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]

```

We can use the list conversion function to make a list out of any generator (see Section 8.8 for information about generators). Listing 10.9 (generator2list.py) uses the prime_sequence generator we saw in Listing 8.17 (generatedprimes.py) to make a list containing all the prime number in the range 20 to 50.

Listing 10.9: generator2list.py

```

from math import sqrt

def is_prime(n):
    ''' Returns True if non-negative integer n is prime;
        otherwise, returns false '''
    if n == 2:
        # 2 is the only even prime number
        return True

```

```

    if n < 2 or n % 2 == 0:          # Handle simple cases immediately
        return False                # No evens and nothing less than 2
    trial_factor = 3
    root = sqrt(n)
    while trial_factor <= root:
        if n % trial_factor == 0:   # Is trial factor a factor?
            return False            # Yes, return right away
        trial_factor += 2           # Next potential factor, skip evens
    return True                      # Tried them all, must be prime

def prime_sequence(begin, end):
    ''' Generates the sequence of prime numbers between begin and end. '''
    for value in range(begin, end + 1):
        if is_prime(value):         # See if value is prime
            yield value              # Produce the prime number

def main():
    ''' Make a list from a generator '''
    # Build the list of prime numbers in the range 20 to 50
    primes = list(prime_sequence(20, 50))
    print(primes)

if __name__ == '__main__':
    main()    # Run the program

```

Listing 10.9 (generator2list.py) displays the list built from our custom prime number generator:

```
[23, 29, 31, 37, 41, 43, 47]
```

It is easy to make a list in which all the elements are the same or a pattern of elements repeat. The `*` operator, when applied to a list and an integer, “multiplies” the elements of a list. The expression

```
[0] * 5
```

produces the list `[0, 0, 0, 0, 0]`. The integer multiplier may be any valid integer expression. Listing 10.10 (makeuniformlists.py) builds several lists using the `*` list multiplication operator.

Listing 10.10: makeuniformlists.py

```

def main():
    a = [0] * 10
    print(a)

    a = [3.4] * 5
    print(a)

    a = 3 * ['ABC']
    print(a)

    a = 4 * [10, 20, 30]
    print(a)

    n = 3    # Use a variable multiplier

```

```

a = n * ['abc', 22, 8.7]
print(a)

main()

```

The output of Listing 10.10 (makeuniformlists.py) is

```

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[3.4, 3.4, 3.4, 3.4, 3.4]
['ABC', 'ABC', 'ABC']
[10, 20, 30, 10, 20, 30, 10, 20, 30, 10, 20, 30]
['abc', 22, 8.7, 'abc', 22, 8.7, 'abc', 22, 8.7]

```

Observe that the integer multiplier may appear either to left or the right of the `*` operator, and the effects are the same. This means the list multiplication `*` operator is *commutative*.

The `*` multiplier operator works similarly with strings. Consider the following interactive sequence:

```

>>> 'abc' * 3
'abccabccabcc'

```

We now have all the tools we need to build a program that flexibly averages numbers while retaining all the values the user enters. Listing 10.11 (listaverage.py) uses an list and a loop to achieve the generality of Listing 10.2 (averagenumbers2.py) with the ability to retain all input for later redisplay.

Listing 10.11: listaverage.py

```

def main():
    # Set up variables
    sum = 0.0
    NUMBER_OF_ENTRIES = 5
    numbers = []

    # Get input from user
    print("Please enter", NUMBER_OF_ENTRIES, "numbers: ")
    for i in range(0, NUMBER_OF_ENTRIES):
        num = eval(input("Enter number " + str(i) + ": "))
        numbers += [num]
        sum += num

    # Print the numbers entered
    print(end="Numbers entered: ")
    for num in numbers:
        print(num, end=" ")
    print()    # Print newline

    # Print average
    print("Average:", sum/NUMBER_OF_ENTRIES)

main() # Execute main

```

The output of Listing 10.11 (listaverage.py) is similar to the original Listing 10.1 (averagenumbers.py) program:

```
Please enter 5 numbers:
Enter number 0: 9.0
Enter number 1: 3.5
Enter number 2: 0.2
Enter number 3: 100.0
Enter number 4: 15.3
Numbers entered: 9.0 3.5 0.2 100.0 15.3
Average: 25.6
```

Unlike the original program, however, we now conveniently can extend this program to handle as many values as we wish. We need only change the definition of the `NUMBER_OF_ENTRIES` variable to allow the program to handle any number of values. This centralization of the definition of the list's size eliminates duplicating a literal numeric value and leads to a program that is more maintainable. Suppose every occurrence of `NUMBER_OF_ENTRIES` were replaced with the literal value 5. The program would work exactly the same way, but changing the size would require touching many places within the program. When duplicate information is scattered throughout a program, it is a common mistake to update some but not all of the information when a change is to be made. If all of the duplicate information is not updated to agree, the inconsistencies result in logic errors within the program. By faithfully using the `NUMBER_OF_ENTRIES` variable throughout the program instead of the literal numeric value, we can avoid the problems of these potential inconsistencies.

The first loop in Listing 10.11 (`listaverage.py`) collects all five input values from the user. The second loop prints all the numbers the user entered.

When treated as a Boolean expression, the empty list (`[]`) is interpreted as `False`, and all other lists are considered `True`. This means `bool([])` evaluates to `True`, since `[]` is not empty; it contains one element—the empty list.

10.4 List Membership

We can use the Python `in` operator to determine if an object is an element in a list. If `lst` is a list, the expression `x in lst` evaluates to `True` if `x` is an element in `lst`; otherwise, the expression is `False`. Similarly, the expression `x not in lst` evaluates to `True` if `x` is not an element in `lst`; otherwise, the expression is `False`. The expression `x not in lst` is equivalent to `not(x in lst)`. Listing 10.12 (`listmembership.py`) exercises Python's `in` operator.

Listing 10.12: `listmembership.py`

```
lst = list(range(0, 21, 2))
for i in range(-2, 23):
    if i in lst:
        print(i, 'is a member of', lst)
    if i not in lst:
        print(i, 'is NOT a member of', lst)
```

Listing 10.12 (`listmembership.py`) prints

```
-2 is NOT a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
-1 is NOT a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
0 is a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
1 is NOT a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

```
2 is a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
3 is NOT a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
4 is a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
5 is NOT a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
6 is a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
7 is NOT a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
8 is a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
9 is NOT a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
10 is a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
11 is NOT a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
12 is a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
13 is NOT a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
14 is a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
15 is NOT a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
16 is a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
17 is NOT a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
18 is a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
19 is NOT a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
20 is a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
21 is NOT a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
22 is NOT a member of [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Note that the `in` operator produces a Boolean result; it can reveal whether or not an object is a member of a list. It cannot reveal the location (index) of the object it finds. We consider options for locating elements in Section 11.4.

10.5 List Assignment and Equivalence

Given the assignment

```
lst = [2, 4, 6, 8]
```

the expression `lst` is very different from the expression `lst[2]`. The expression `lst` is a reference to the list, while `lst[2]` is a reference to a particular element in the list, in this case the integer 6. The integer 6 is immutable (see Section 7.3); a literal integer cannot change to be another value. Six is always six. A variable, of course, can change its value and its type through assignment. Variable assignment changes the object to which the variable is bound. Recall Figures 2.2–2.6, and consider the Listing 10.13 (`listassignment.py`).

Listing 10.13: `listassignment.py`

```
a = [10, 20, 30, 40]
b = [10, 20, 30, 40]
print('a =', a)
print('b =', b)
b[2] = 35
print('a =', a)
print('b =', b)
```

Figure 10.2 shows the consequences of each of the assignment statements in Listing 10.13 (`listassignment.py`),

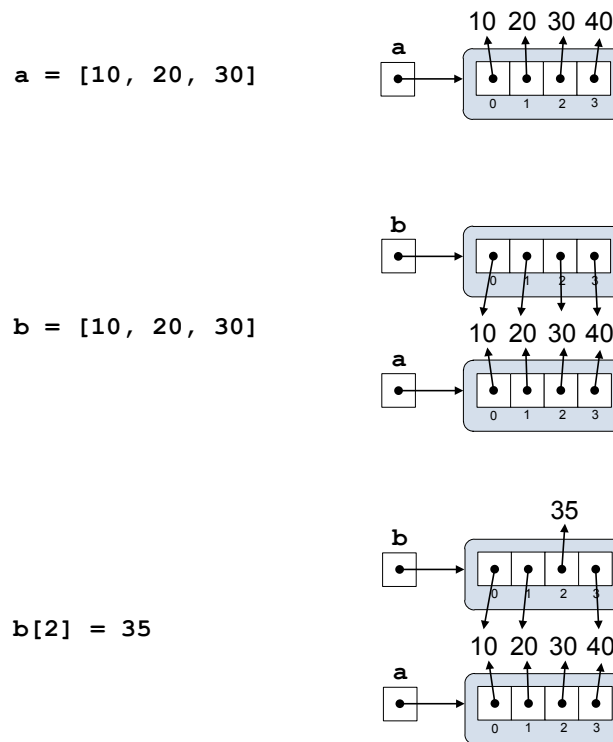


Figure 10.2: State of Listing 10.13 (listassignment.py) as the assignment statements execute

As Figure 10.2 illustrates, variables `a` and `b` refer to two different list objects; however, the elements of both lists bind to the same (immutable) values. Reassigning an element of list `b` does not affect list `a`. The output of Listing 10.13 (listassignment.py) verifies this analysis:

```
a = [10, 20, 30, 40]
b = [10, 20, 30, 40]
a = [10, 20, 30, 40]
b = [10, 20, 35, 40]
```

Now consider Listing 10.14 (listalias.py), a subtle variation of Listing 10.13 (listassignment.py). At first glance, the code in Listing 10.14 (listalias.py) looks like it may behave exactly like Listing 10.13 (listassignment.py).

Listing 10.14: listalias.py

```
a = [10, 20, 30, 40]
b = a
print('a =', a)
print('b =', b)
b[2] = 35
print('a =', a)
print('b =', b)
```

As Figure 10.3 illustrates, the second assignment statement causes variables *a* and *b* to refer to the *same* list object. We say that *a* and *b* are *aliases*. Reassigning *b*[2] changes *a*[2] as well, as Listing 10.14

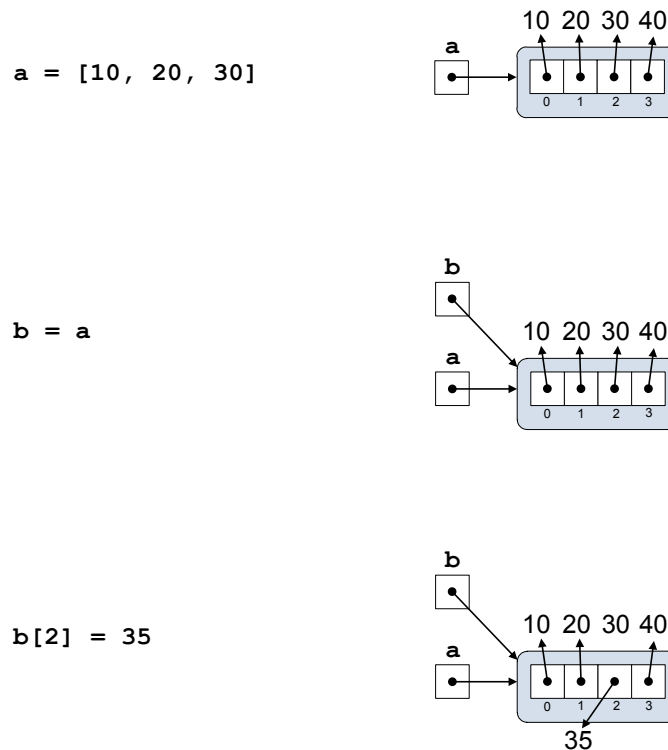


Figure 10.3: State of Listing 10.14 (`listalias.py`) as the assignment statements execute

(`listalias.py`)’s output shows:

```
a = [10, 20, 30, 40]
b = [10, 20, 30, 40]
a = [10, 20, 35, 40]
b = [10, 20, 35, 40]
```

If *a* refers to a list, the statement

```
b = a
```

does not make a copy of *a*’s list. Instead it makes *a* and *b* aliases to the same list. Lists are *mutable* data structures. We may reassign individual list elements via `[]`. If more than one variable is bound to the same list, any element modification through one of the variables will affect the list from the point of view of all the aliased variables.

The familiar `==` equality operator determines if two lists contain the same elements. The `is` operator determines if two variables alias the same list. Listing 10.15 (`listequivalence.py`) demonstrates the difference between the two operators.

Listing 10.15: listequivalence.py

```
# a and b are distinct lists that contain the same elements
a = [10, 20, 30, 40]
b = [10, 20, 30, 40]
print('Is ', a, ' equal to ', b, '?', sep='', end=' ')
print(a == b)

print('Are ', a, ' and ', b, ' aliases?', sep='', end=' ')
print(a is b)

# c and d alias are distinct lists that contain the same elements
c = [100, 200, 300, 400]
d = c      # Makes d an alias of c
print('Is ', c, ' equal to ', d, '?', sep='', end=' ')
print(c == d)

print('Are ', c, ' and ', d, ' aliases?', sep='', end=' ')
print(c is d)
```

Listing 10.15 (listequivalence.py) prints

```
Is [10, 20, 30, 40] equal to [10, 20, 30, 40]? True
Are [10, 20, 30, 40] and [10, 20, 30, 40] aliases? False
Is [100, 200, 300, 400] equal to [100, 200, 300, 400]? True
Are [100, 200, 300, 400] and [100, 200, 300, 400] aliases? True
```

When comparing lists `lst1` and `lst2`, if the expression `lst1 is lst2` evaluates to `True`, the expression `lst1 == lst2` is guaranteed to be `True`.

What if we wish to make a copy of an existing list? Listing 10.16 (listcopy.py) shows one way to accomplish this.

Listing 10.16: listcopy.py

```
def list_copy(lst):
    result = []
    for item in lst:
        result += [item]
    return result

def main():
    # a and b are distinct lists that contain the same elements
    a = [10, 20, 30, 40]
    b = list_copy(a)      # Make a copy of a
    print('a =', a, ' b =', b)

    print('Is ', a, ' equal to ', b, '?', sep='', end=' ')
    print(a == b)

    print('Are ', a, ' and ', b, ' aliases?', sep='', end=' ')
    print(a is b)

    b[2] = 35            # Change an element of b
    print('a =', a, ' b =', b)
```

```
main()
```

The output of Listing 10.16 (listcopy.py) reveals:

```
a = [10, 20, 30, 40]    b = [10, 20, 30, 40]
Is [10, 20, 30, 40] equal to [10, 20, 30, 40]? True
Are [10, 20, 30, 40] and [10, 20, 30, 40] aliases? False
a = [10, 20, 30, 40]    b = [10, 20, 35, 40]
```

The `list_copy` function in Listing 10.16 (listcopy.py) makes an actual copy of `a`. Changing an element of `b` does not affect list `a`.

In Section 10.7 we will see a more effective way to copy a list.

We can use `range` to create a range of values that the `for` statement can consume, but this `range` object is not a list. The following interactive sequence shows how we can use the `list` function to make a list out of a `range` object:

```
>>> r = range(10)
>>> r
range(0, 10)
>>> type(r)
<class 'range'>
>>> list(r)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> lst = list(r)
>>> lst
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> type(lst)
<class 'list'>
```

The expression `list(range(5))`, therefore, creates the list `[0, 1, 2, 3, 4]`.

The flexibility of the `range` expression makes it easy to create a variety of different kinds of lists with regular structure. Listing 10.17 (rangetolist.py) explores several possibilities.

Listing 10.17: rangetolist.py

```
print(list(range(11)))
print(list(range(10, 101, 10)))
print(list(range(10, -1, -1)))
```

Listing 10.17 (rangetolist.py) produces

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

10.6 List Bounds

In the following code fragment:

```
a = [10, 20, 30, 40]
```

All of the following expressions are valid: `a[0]`, `a[1]`, `a[2]` and `a[3]`. The expression `a[4]` does not represent a valid element in the list. An attempt to use this expression, as in

```
a = [10, 20, 30, 40]
print(a[4])    # Out-of-bounds access
```

results in a run-time exception. The interpreter will insist that the programmer use an integral value for an index, but in order to prevent a run-time exception the programmer must ensure that the index used is within the bounds of the list. Consider the following code:

```
# Make a list containing 100 zeros
v = [0] * 100
# User enters x at run time
x = int(input("Enter an integer: "))
v[x] = 1    # Is this OK? What is x?
```

Since a list index may consist of an arbitrary integer expression, the interpreter checks every attempt to access a list. If the interpreter detects an out-of-bounds index, the interpreter raises an `IndexError` (list index out-of-bounds) exception. The programmer must ensure the provided index is in bounds to prevent such a run-time error.

The above unreliable code can be helped with conditional access:

```
# Make a list containing 100 zeros
v = [0] * 100
# User enters x at run time
x = int(input("Enter an integer: "))
# Ensure index is within list bounds
if 0 <= x < len(v):
    v[x] = 1    # This should be fine
else:
    print("Value provided is out of range")
```

Listing 10.18 (`badreverse.py`) attempts to print the list's elements in reverse order, but it fails to stay inside the bounds of the list.

Listing 10.18: `badreverse.py`

```
def make_list():
    """
    Builds a list from input provided by the user.
    """
    result = []    # List to return is initially empty
    in_val = 0     # Ensure loop is entered at least once
    while in_val >= 0:
        in_val = int(input("Enter integer (-1 quits): "))
        if in_val >= 0:
            result = result + [in_val]    # Add item to list
    return result

def main():
    col = make_list()
    # Print the list in reverse
```

```

    for i in range(len(col), 0, -1):
        print(col[i], end=" ")
        print()
main()

```

The `for` statement

```

for i in range(len(col), 0, -1):
    print(col[i], end=" ")

```

considers first the element at `col[len(col)]`, which is one index past the end of the list. The corrected `for` statement is

```

for i in range(len(col) - 1, -1, -1):
    print(col[i], end=" ")

```

10.7 Slicing

We can make a new list from a portion of an existing list using a technique known as *slicing*. A list slice is an expression of the form

list [*begin* : *end* : *step*]

where

- *list* is a list—a variable referring to a list object, a literal list, or some other expression that evaluates to a list,
- *begin* is an integer representing the starting index of a subsequence of the list, and
- *end* is an integer that is one larger than the index of the last element in a subsequence of the list.
- *step* is an integer that specifies the stride size through the list. A step size of three, for example, would include every third element in the list within the specified range. Negative step values reverse the direction of the slice.

If missing, the *begin* value defaults to 0. A *begin* value less than zero is treated as zero. If the *end* value is missing, it defaults to the length of the list. An *end* value greater than the length of the list is treated as the length of the list. The default step value is 1. The examples provided in Listing 10.19 (`listslice.py`) best illustrate how list slicing works.

Listing 10.19: `listslice.py`

```

lst = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120]
print(lst)           # [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120]
print(lst[0:3])      # [10, 20, 30]
print(lst[4:8])      # [50, 60, 70, 80]
print(lst[2:5])      # [30, 40, 50]
print(lst[-5:-3])    # [80, 90]

```

```

print(lst[:3])      # [10, 20, 30]
print(lst[4:])      # [50, 60, 70, 80, 90, 100, 110, 120]
print(lst[:])       # [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120]
print(lst[-100:3])  # [10, 20, 30]
print(lst[4:100])   # [50, 60, 70, 80, 90, 100, 110, 120]
print(lst[2:-2:2])  # [30, 50, 70, 90]
print(lst[::-2])    # [10, 30, 50, 70, 90, 110]
print(lst[::-1])    # [120, 110, 100, 90, 80, 70, 60, 50, 40, 30, 20, 10]

```

Observe that when a slice involves a negative step value the first argument in the slice represents the end of the reverse slice, and the second argument is the beginning of the slice.

Slicing is the easiest way to make a copy of a list. The expression `lst[:]` evaluates to a copy of list `lst`. The `list_copy` function we saw in Listing 10.16 (`listcopy.py`) made for an interesting exercise, but list slicing is shorter, simpler way to achieve the same result.

Listing 10.20 (`prefixsuffix.py`) prints all the prefixes and suffixes of the list `[1, 2, 3, 4, 5, 6, 7, 8]`.

Listing 10.20: `prefixsuffix.py`

```

a = [1, 2, 3, 4, 5, 6, 7, 8]
print('Prefixes of', a)
for i in range(0, len(a) + 1):
    print('<', a[0:i], '>', sep='')
print('-----')
print('Suffixes of', a)
for i in range(0, len(a) + 1):
    print('<', a[i:len(a) + 1], '>', sep='')

```

Listing 10.20 (`prefixsuffix.py`) prints

```

Prefixes of [1, 2, 3, 4, 5, 6, 7, 8]
<[]>
<[1]>
<[1, 2]>
<[1, 2, 3]>
<[1, 2, 3, 4]>
<[1, 2, 3, 4, 5]>
<[1, 2, 3, 4, 5, 6]>
<[1, 2, 3, 4, 5, 6, 7]>
<[1, 2, 3, 4, 5, 6, 7, 8]>
-----
Suffixes of [1, 2, 3, 4, 5, 6, 7, 8]
<[1, 2, 3, 4, 5, 6, 7, 8]>
<[2, 3, 4, 5, 6, 7, 8]>
<[3, 4, 5, 6, 7, 8]>
<[4, 5, 6, 7, 8]>
<[5, 6, 7, 8]>
<[6, 7, 8]>
<[7, 8]>
<[8]>
<[]>

```

When the slicing expression appears on the left side of the assignment operator it can modify the contents of the list. This is known as *slice assignment*. A slice assignment can modify a list by removing or

adding a subrange of elements in an existing list. Listing 10.21 (listslicemod.py) demonstrates how to use slice assignment to modify a list.

Listing 10.21: listslicemod.py

```
lst = [10, 20, 30, 40, 50, 60, 70, 80]
print(lst)           # Print the list
lst[2:5] = ['a', 'b', 'c'] # Replace [30, 40, 50] segment with ['a', 'b', 'c']
print(lst)
print('=====')
lst = [10, 20, 30, 40, 50, 60, 70, 80]
print(lst)           # Print the list
lst[2:6] = ['a', 'b'] # Replace [30, 40, 50, 60] segment with ['a', 'b']
print(lst)
print('=====')
lst = [10, 20, 30, 40, 50, 60, 70, 80]
print(lst)           # Print the list
lst[2:2] = ['a', 'b', 'c'] # Insert ['a', 'b', 'c'] segment at index 2
print(lst)
print('=====')
lst = [10, 20, 30, 40, 50, 60, 70, 80]
print(lst)           # Print the list
lst[2:5] = [] # Replace [30, 40, 50] segment with [] (delete the segment)
print(lst)
```

Listing 10.21 (listslicemod.py) displays:

```
[10, 20, 30, 40, 50, 60, 70, 80]
[10, 20, 'a', 'b', 'c', 60, 70, 80]
=====
[10, 20, 30, 40, 50, 60, 70, 80]
[10, 20, 'a', 'b', 70, 80]
=====
[10, 20, 30, 40, 50, 60, 70, 80]
[10, 20, 'a', 'b', 'c', 30, 40, 50, 60, 70, 80]
=====
[10, 20, 30, 40, 50, 60, 70, 80]
[10, 20, 60, 70, 80]
```

10.8 List Element Removal

We have seen how to append elements to a list using the list concatenation operator (+). We can use `del` to remove a specific element from a list via its index. The following sequence uses `range` to build a list and `del` to remove one of the list's elements:

```
>>> a = list(range(10, 51, 10))
>>> a
[10, 20, 30, 40, 50]
>>> del a[2]
>>> a
[10, 20, 40, 50]
```

We can remove a contiguous range of elements of a list using `del` with a slice, as shown here:

```
>>> b = list(range(20))
>>> b
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> del b[5:15]
>>> b
[0, 1, 2, 3, 4, 15, 16, 17, 18, 19]
```

As with scalar variables, you can `del` multiple list elements with one `del` statement, but it requires much more care. Consider the following:

```
>>> c = list(range(20))
>>> c
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> del c[1], c[18]
>>> c
[0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
```

You might expect the `del` statement to remove element 1 and element 18. Instead, it removed 1 and 19. The deletion progresses from left to right, so the statement first removes 1, leaving the following list:

```
[0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

The subsequent removal of `c[18]` occurs on this new list, not the original list. Element 19 now is at index 18, so the statement deletes element 19 instead of element 18. Consider the following list:

```
d = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

suppose we wish to delete elements 2, 3, 4, 5, 6, and 18. We might try the following `del` statement:

```
del c[2:7], c[18]
```

This statement, however, produces a run-time exception. The error arises because removing the slice of elements shortens the list so index 18 is out of range. Reordering the statement produces the desired results:

```
del c[18], c[2:7]
```

since it removes elements from the back to the front.

Because faulty attempts at multiple list element deletions can lead to surprising results it is best to restrict a list `del` statement to a single element or a single slice.

10.9 Lists and Functions

We can pass a list as an argument to a function, as shown in Listing 10.22 (`listfunc.py`)

Listing 10.22: `listfunc.py`

```
def sum(lst):
    """
    Adds up the contents of a list of numeric
    values
```

```
    lst is the list to sum
    Returns the sum of all the elements
    or zero if the list is empty.
    '''
    result = 0
    for item in lst:
        result += item
    return result

def make_zero(lst):
    '''
    Makes every element in list lst zero
    '''
    for i in range(len(lst)):
        lst[i] = 0

def random_list(n):
    '''
    Builds a list of n integers, where each integer
    is a pseudorandom number in the range 0...99.
    Returns the random list.
    '''
    import random
    result = []
    for i in range(n):
        rand = random.randrange(100)
        result += [rand]
    return result

def main():
    a = [2, 4, 6, 8]
    # Print the contents of the list
    print(a)
    # Compute and display sum
    print(sum(a))
    # Zero out all the elements of list
    make_zero(a)
    # Reprint the contents of the list
    print(a)
    # Compute and display sum
    print(sum(a))
    # Test empty list
    a = []
    print(a)
    print(sum(a))
    # Test pseudorandom list with 10 elements
    a = random_list(10)
    print(a)
    print(sum(a))

main()
```

In Listing 10.22 (`listfunc.py`) the functions `sum` and `make_zero` accept a parameter of type `list`. Section 7.3 addressed the consequences of passing immutable types like integers and strings to functions. Since `list` objects are mutable, passing to a function a reference to a `list` object binds the formal parameter to the `list` object. This means the formal parameter becomes an alias of the actual parameter. The `sum` method does not attempt to modify its parameter, but the `make_zero` method changes every element in the `list` to zero. This means the `make_zero` function will modify the a `list` object in `main`.

10.10 List Methods

All Python `list`s are instances of the `list` class. Table 10.1 lists some of the methods available to `list` objects.

list Methods	
<code>count</code>	Returns the number of times a given element appears in the list. Does not modify the list.
<code>insert</code>	Inserts a new element before the element at a given index. Increases the length of the list by one. Modifies the list.
<code>append</code>	Adds a new element to the end of the list. Modifies the list.
<code>index</code>	Returns the lowest index of a given element within the list. Produces an error if the element does not appear in the list. Does not modify the list.
<code>remove</code>	Removes the first occurrence (lowest index) of a given element from the list. Produces an error if the element is not found. Modifies the list if the item to remove is in the list.
<code>reverse</code>	Physically reverses the elements in the list. The list is modified.
<code>sort</code>	Sorts the elements of the list in ascending order. The list is modified.

Table 10.1: A few of the methods available to `list` objects

Since `list`s are mutable data structures, the `list` class has both `__getitem__` and `__setitem__` methods. The statement

```
x = lst[2]
```

behind the scenes becomes the method call

```
x = list.__getitem__(lst, 2)
```

and the statement

```
lst[2] = x
```

maps to

```
list.__setitem__(lst, 2, x)
```

The `str` class does not have a `__setitem__` method, since strings are immutable.

The code

```
lst = ["one", "two", "three"]
lst += ["four"]
```

is equivalent to

```
lst = ["one", "two", "three"]
lst.append("four")
```

but the version using `append` is more efficient.

10.11 Prime Generation with a List

Listing 10.23 (`fasterprimes.py`) uses an algorithm developed by the Greek mathematician Eratosthenes who lived from 274 B.C. to 195 B.C. Called the Sieve of Eratosthenes, the principle behind the algorithm is simple: Make a list of all the integers two and larger. Two is a prime number, but any multiple of two cannot be a prime number (since a multiple of two has two as a factor). Go through the rest of the list and mark out all multiples of two (4, 6, 8, ...). Move to the next number in the list (in this case, three). If it is not marked out, it must be prime, so go through the rest of the list and mark out all multiples of that number (6, 9, 12, ...). Continue this process until you have listed all the primes you want.

Listing 10.23 (`fasterprimes.py`) implements the Sieve of Eratosthenes in a Python function.

Listing 10.23: `fasterprimes.py`

```
# Display the prime numbers between 2 and 500

# Largest potential prime considered
MAX = 500

def main():
    # Each position in the Boolean list indicates
    # if the number of that position is not prime:
    # false means "prime," and true means "composite."
    # Initially all numbers are prime until proven otherwise
    nonprimes = MAX * [False] # Initialize to all False

    # First prime number is 2; 0 and 1 are not prime
    nonprimes[0] = nonprimes[1] = True

    # Start at the first prime number, 2.
    for i in range(2, MAX + 1):
        # See if i is prime
        if not nonprimes[i]:
            print(i, end=" ")
            # It is prime, so eliminate all of its
            # multiples that cannot be prime
            for j in range(2*i, MAX + 1, i):
                nonprimes[j] = True
    print() # Move cursor down to next line
```

How much better is the algorithm in Listing 10.23 (`fasterprimes.py`) than the square-root-optimized version we saw in Listing 6.8 (`timemoreefficientprimes.py`)? Listing 10.24 (`timeprimes.py`) compares the execution speed of the two algorithms.

Listing 10.24: `timeprimes.py`

```
# Count the number of prime numbers less than
# 2 million and time how long it takes
# Compares the performance of two different
# algorithms.

from time import clock
from math import sqrt

def count_primes(n):
    """
    Generates all the prime numbers from 2 to n - 1.
    n - 1 is the largest potential prime considered.
    """
    start = clock()    # Record start time

    count = 0
    for val in range(2, n):
        root = round(sqrt(val)) + 1
        # Try all potential factors from 2 to the square root of n
        for trial_factor in range(2, root):
            if val % trial_factor == 0: # Is it a factor?
                break                # Found a factor
        else:
            count += 1                # No factors found

    stop = clock()    # Stop the clock
    print("Count =", count, "Elapsed time:", stop - start, "seconds")

def seive(n):
    """
    Generates all the prime numbers from 2 to n - 1.
    n - 1 is the largest potential prime considered.
    Algorithm originally developed by Eratosthenes.
    """
    start = clock()    # Record start time

    # Each position in the Boolean list indicates
    # if the number of that position is not prime:
    # false means "prime," and true means "composite."
    # Initially all numbers are prime until proven otherwise
    nonprimes = n * [False] # Global list initialized to all False

    count = 0

    # First prime number is 2; 0 and 1 are not prime
```

```

nonprimes[0] = nonprimes[1] = True

# Start at the first prime number, 2.
for i in range(2, n):
    # See if i is prime
    if not nonprimes[i]:
        count += 1
        # It is prime, so eliminate all of its
        # multiples that cannot be prime
        for j in range(2*i, n, i):
            nonprimes[j] = True
    # Print the elapsed time
    stop = clock()
    print("Count =", count, "Elapsed time:", stop - start, "seconds")

def main():
    count_primes(2000000)
    seive(2000000)

main()

```

Since printing to the screen takes up the majority of the time, Listing 10.24 (timeprimes.py) counts the number of primes rather than printing each one. This allows us to better compare the behavior of the two approaches. The square root version has been optimized slightly more: the floating-point root variable is not an integer. The less than comparison between two integers is faster than the floating-point equivalent.

The output of Listing 10.24 (timeprimes.py) on one system reveals

```

Count = 148933 Elapsed time: 37.57788172418102 seconds
Count = 148933 Elapsed time: 1.028922514194747 seconds

```

Our previous “optimized” version requires almost 38 seconds to count the number of primes less than two million, while the version based on the Sieve of Eratosthenes takes only about one second.

10.12 Command-line Arguments

The `sys` module provides a global variable named `argv` that is a list of extra text that the user can supply when launching an application from the operating system shell (normally called the *command prompt* in Windows and *terminal* in OS X and Linux). To run a program stored in the file `myprog.py`, the user would type the command

```
python myprog.py
```

Some programs expect or allow the user to provide extra information. Listing 10.25 (cmdlineargs.py) is a program meant to be executed from the command line with extra arguments. It simply reports the extra information the user supplied.

Listing 10.25: cmdlineargs.py

```
import sys
```

```
for arg in sys.argv:
    print('[' + arg + '']')
```

The following shows one run of Listing 10.25 (cmdlineargs.py):

```
C:\Code>python cmdlineargs.py -h 45 extra
[cmdlineargs.py]
[-h]
[45]
[extra]
```

The shell printed C:

Code_i, and the user typed the remainder of the first line. In response, the program printed four lines that follow. Notice that `argv[0]` is simply the name of the file containing the program. `argv[1]` is the string `'-h'`, `argv[2]` is the string `'45'`, and `argv[3]` is the string `'extra'`.

In Listing 10.26 (sqrtcmdline.py), the user supplies a range of integers on the command line. The program then prints all the integers in that range along with their square roots.

Listing 10.26: sqrtcmdline.py

```
from sys import argv
from math import sqrt

if len(argv) < 3:
    print('Supply range of values')
else:
    for n in range(int(argv[1]), int(argv[2]) + 1):
        print(n, sqrt(n))
    print()
```

The following shows some sample runs of Listing 10.26 (sqrtcmdline.py):

```
C:\Code>python sqrtcmdline.py
Supply range of values

C:\Code>python sqrtcmdline.py 2
Supply range of values

C:\Code>python sqrtcmdline.py 2 10
2 1.4142135623730951
3 1.7320508075688772
4 2.0
5 2.23606797749979
6 2.449489742783178
7 2.6457513110645907
8 2.8284271247461903
9 3.0
10 3.1622776601683795
```

10.13 List Comprehensions

Mathematicians often represent sets in two different ways:

1. set roster notation, which enumerates the elements in the set, and
2. set builder notation, which describes the contents of the set using a rule for constructing the set's elements

We can express P , the set of perfect squares less than 50, using both of these methods:

1. set roster notation: $P = \{0, 1, 4, 9, 16, 25, 36, 49\}$
2. set builder notation: $P = \{x^2 \mid x \in \{0, 1, 2, 3, 4, 5, 6, 7\}\}$

The set roster notation example is obvious—it just lists the elements of the set. We read the set builder notation example as “ P is the set of all squares of x , such that x is taken from the set $\{0, 1, 2, 3, 4, 5, 6, 7\}$. Set builder notation in mathematics is essential for representing very large sets and infinite sets; for example, consider $S = \{x^2 \mid x \in \mathbb{Z}\}$, the set of *all* perfect squares ($\mathbb{Z}$ is the infinite set of integers).

We have seen how to express lists in Python in a manner similar to set roster notation in mathematics:

```
P = [1, 4, 9, 16, 25, 36, 49]
```

Python also supports a technique similar to set builder notation, called *list comprehension*. In Python, we can express list P above as

```
P = [x**2 for x in [0, 1, 2, 3, 4, 5, 6, 7]]
```

or, using a range expression as

```
P = [x**2 for x in range(8)]
```

Note the use of the keywords `for` and `in` within the square brackets. The `for` keyword takes the place of the mathematical $\mid$ symbol (usually pronounced “such that” or “where”), and `in` is the $\in$ membership relation. All list comprehensions use `for` within the square brackets.

We have seen that the range expression is useful for building lists with a regular, predictable ordering. As another example, consider the following code that builds a list of the multiples of four from 4 to 40:

```
>>> lst = list(range(4, 41, 4))
>>> lst
[4, 8, 12, 16, 20, 24, 28, 32, 36, 40]
```

One limitation of range is that its arguments all must be integers. Suppose we wish to create succinctly a list of floating-point numbers in a regular sequence. We cannot use the range expression by itself to succinctly express such a list, but a list comprehension is ideal for the task. The following interactive sequence creates a list containing the first ten multiples of one-half:

```
>>> halves = [x/2 for x in range(10)]
>>> halves
[0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]
```

The single slash (/) performs floating-point division, populating the list with floating-point values.

We can use a list comprehension to filter certain elements out of a list. In the following example we take a list containing integers and produce a new list with all the negative values omitted:

```
>>> lst = [9, -44, 0, 30, 2, -6, -8, 9, -8, 23]
>>> lst
[9, -44, 0, 30, 2, -6, -8, 9, -8, 23]
>>> [x for x in lst if x >= 0]
[9, 0, 30, 2, 9, 23]
```

The expression `[x for x in lst if x >= 0]` selects all the elements from list `lst` except for those less than zero. If we instead want a list of the negative values, invert the logic of the condition:

```
>>> lst
[9, -44, 0, 30, 2, -6, -8, 9, -8, 23]
>>> [x for x in lst if x < 0]
[-44, -6, -8, -8]
```

The next example begins with a mixed list of numeric values and strings. The following interactive sequence shows how we can use list comprehensions to selectively filter out of the list all the strings or all the numbers:

```
>>> lst = ['ABC', 23.4, 7, 'Wow', 16, 'xyz', 10]
>>> lst
['ABC', 23.4, 7, 'Wow', 16, 'xyz', 10]
>>> [x for x in lst if type(x) != str]
[23.4, 7, 16, 10]
>>> lst
['ABC', 23.4, 7, 'Wow', 16, 'xyz', 10]
>>> [x for x in lst if type(x) == str]
['ABC', 'Wow', 'xyz']
```

Neither list comprehension expression affects the original list `lst`.

The lists we build with list comprehension are not limited to elements with simple, single values. The following interactive sequence builds a list containing tuples of the first 10 positive integers paired with their squares:

```
>>> squares = [(x, x**2) for x in range(1, 11)]
>>> squares
[(1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64), (9, 81), (10, 100)]
```

We can use multiple `for` expressions within a single list comprehension. This is useful for combining elements from different lists. In the following example we make tuples out of the elements of two different lists:

```
>>> pairs = [(x, y) for x in [1, 2, 3] for y in ['a', 'b']]
>>> pairs
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')]
```

We can use the `if` keyword to add conditions for list membership. To start with, suppose we wish to express the list containing the ordered pairs of elements taken from the list `[1, 2, 3]`. That is easy enough:

```
>>> points = [(x, y) for x in [1, 2, 3] for y in [1, 2, 3]]
>>> points
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
```

We can use variables within our list comprehension to shorten it a bit:

```
>>> V = [1, 2, 3]
>>> points = [(x, y) for x in V for y in V]
>>> points
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
```

Now suppose we wish to exclude pairs that contain the same elements. We can add an **if** expression to the end of the list comprehension to impose additional conditions on the list members:

```
>>> V = [1, 2, 3]
>>> points = [(x, y) for x in V for y in V if x != y]
>>> points
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

Here we see no number is paired with itself. The following list comprehension excludes pairs with components that sum to an even number:

```
>>> points = [(x, y) for x in [1, 2, 3] for y in [1, 2, 3] if (x + y) % 2 != 0]
>>> points
[(1, 2), (2, 1), (2, 3), (3, 2)]
```

We see (1, 3) is missing because $1 + 3 = 4$, and 4 is even.

In the following example we build an almost regular list that omits a few elements:

```
>>> L = [x for x in range(20) if x not in [12, 8, 3, 17]]
>>> L
[0, 1, 2, 4, 5, 6, 7, 9, 10, 11, 13, 14, 15, 16, 18, 19]
```

This list contains all the elements in `range(20)` in order, but it specifically excludes the values 3, 8, 12, and 17.

We can use a Boolean expression to build a list of Boolean values:

```
>>> [x > 2 for x in [5, 1, 0, 7, 2, 7, 3]]
[True, False, False, True, False, True, True]
```

We will see in Section 12.10 how to apply such Boolean conditions in a slightly different way to determine quickly and easily if *all* the elements in a list possess a given property or if *any* elements in a list possess a given property.

List comprehensions not essential to any program. Consider the list comprehension above:

```
L = [x for x in range(20) if x not in [12, 8, 3, 17]]
```

We can rewrite this as

```
L = []
for x in range(20):
    if x not in [12, 8, 3, 17]:
        L += [x]
```

While our expanded code is functionally equivalent to the list comprehension, in general a list comprehension will be more efficient than its equivalent expanded form.

Python list comprehensions are powerful and can be quite complex. When programming it sometimes is easier to build the list without list comprehension and later discover a way to transform the code to use list comprehension. As an example, consider the task of building a list that contains all the prime numbers less than 100. For all $n \geq 2$, the following list comprehension creates a list of all the factors of n , not including 1 and n itself:

```
[x for x in range(2, n) if n % x == 0]
```

The following expression evaluates to true if n is a prime number; otherwise, it evaluates to false:

```
[x for x in range(2, n) if n % x == 0] == []
```

(Note that this expression does not take advantage of the square root optimization we saw in Listing 6.4 (`moreefficientprimes.py`).) In Python, the empty list (`[]`) is considered false, and any non-empty list is considered true, so the following expression reveals the primality of n as well:

```
not [x for x in range(2, n) if n % x == 0]
```

To make a list of all the prime numbers less than 80, we can use a list comprehension expression that begins like the following:

```
[p for p in range(2, 80) if ???]
```

We then can replace the `???` with the list comprehension from above that determines if p is a prime number:

```
[p for p in range(2, 80) if not [x for x in range(2, p) if p % x == 0]]
```

In the Python shell we can see the list this expression produces:

```
>>> [p for p in range(2, 80) if not [x for x in range(2, p) if p % x == 0]]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79]
```

List comprehensions are a valuable, powerful tool for creating lists. While the prime numbers example demonstrates the power of Python's list comprehensions, many programmers would consider the resulting list comprehension expression above to be a bit too complex and tricky. Additionally, it is not very efficient. It creates an internal, temporary list of factors for each number it must consider. Simpler list comprehension expressions that avoid nested lists generally are efficient and readable. As a rule, you should use list comprehensions when they make your code simpler and more readable, but do not go out of your way to construct arcane list comprehension expressions just because you can.

If we replace the outer square brackets of a list comprehension with parentheses, the expression becomes a *generator expression* (sometimes called a *generator comprehension*). A generator expression creates a generator (Section 8.8) instead of a list. We can iterate over a generator with a `for` loop, as Listing 10.27 (`generatorexpression.py`) illustrates.

Listing 10.27: `generatorexpression.py`

```
for val in (2**x for x in range(10)):
    print(val, end=' ')
print()
```

Listing 10.27 (`generatorexpression.py`) prints

```
1 2 4 8 16 32 64 128 256 512
```

If you need to keep around the values of a sequence for additional processing, store them in a list and possibly use a list comprehension to make the list. If you simply need to visit the elements in a sequence once, building a list is overkill; use a generator to produce the sequence's elements as needed. The list has to store all of its elements in memory for the life of the list, but a generator produces only one element at a time. This means the list `[x for x in range(n)]` will consume a large amount of the computer's memory if `n` is large. The generator `(x for x in range(n))` uses a relatively small, constant amount of memory regardless of `n`'s value.

Section 10.15 compares lists and generators in more detail.

10.14 Summary of List Creation Techniques

So far we have seen a variety of ways to create lists in Python. To recap, we will look at several different ways to create the following list:

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

- **Literal enumeration:**

```
L = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

- **Piecemeal assembly:**

```
L = []
for i in range(2, 21, 2):
    L += [i]
L = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

- **Creation from a generator or range expression:**

```
L = list(range(2, 21, 2))
```

- **List comprehension:**

```
L = [x for x in range(1, 21) if x % 2 == 0]
```

- **Combination of methods with list concatenation:**

```
L = list(range(2, 9, 2)) + [10, 12, 14] + [x for x in range(16, 21, 2)]
```

Chapter 12 introduces several new ways to create lists from other Python data structures.

10.15 Lists vs. Generators

We now have seen two ways of expressing a sequence of values: generators and lists. How are they similar? Generators and lists share the following characteristics:

- Both generators and lists represent a sequence of values. This means the order of the values matters. There is a first element and a last element in a sequence. Except for the first element in the sequence, every element has a predecessor. Except for the last element, every element in the sequence has a successor.
- We can iterate over both generators and lists using the `for` statement.

How are generators and lists different? Generators and lists differ in the following ways:

- The elements in a list persist for the life of the list, but the elements produced by a generator become available in turn as the iteration with the generator progresses. Once the iteration moves on from the current element, previous elements are unavailable from that particular generator object.
- The memory required for a list generally will be greater than that for a generator that can produce the same sequence of values. This is because the generator manages only one element at a time, while a list must store simultaneously all the elements in the sequence.
- Lists provide *random access*. This means any value at any position in the list is available at any time. A generator serves up values one at a time, in order from the first to the last. We cannot obtain the i^{th} element from a generator without first requesting all the elements that come before it.
- Lists support forward and backward traversal. Generators provide only forward traversal (reversed, for example, does not work with a generator object).

If you need to create a sequence of values and random access is unnecessary, a generator is the better choice. The generator's sequence behaves like a *lazy list*; that is, the sequence element exists only at the time it is needed. If, on the other hand, your program needs to have all the values of a sequence available at any time during the program's execution, a list is the better choice. Unlike generators, a list usually must be fully populated with all of its elements before it truly is useful to the program.

10.16 Summary

CAUTION! SECTION UNDER CONSTRUCTION

- A list represents an ordered sequence of objects
- An element in a list may be accessed via its index using `[]`. The first element is at index 0. If the list contains n elements, the index of the last element is $n - 1$.
- A positive list index is an offset from the beginning of the list. A negative list index is an offset back from an imaginary element one past the end of the list.
- A list may elements of different types.
- List literals list their elements in a comma-separated list enclosed within square brackets `[]`.
- The `len` function returns the length of the list
- A list index is sometimes called a subscript.
- A list subscript must evaluate to an integer. Integer literals, variables, and expressions can be used as list indices.

- A `for` loop is a convenient way to traverse the contents of a list.
- Like other variables, a list variable can be local, global, or a function parameter.
- Direct list assignment produces an alias. A slice of a whole list makes an actual copy of the list.
- The `==` tests for equal contents within lists; the `is` operator tests for list aliases.
- A list may be passed to a function. The formal parameter within the function becomes an alias of the actual parameter passed by the caller. This means functions may modify the contents of a list, and the modification will affect the caller's copy of the list.
- It is the programmer's responsibility to stay within the bounds of a list. Venturing outside the bounds of a list results in a run-time error.
- Lists are mutable objects. Integers, floating-point, and string values are immutable.
- Parts of lists can be expressed with slices. A slice is a copy of a subrange of elements in a list.
- List slices on the right side the assignment operator can modify lists by removing or adding a subrange of elements in an existing list.

10.17 Exercises

CAUTION! SECTION UNDER CONSTRUCTION

1. Can a Python list hold a mixture of integers and strings?
2. What happens if you attempt to access an element of a list using a negative index?
3. What Python statement produces a list containing contains the values 45, -3, 16 and 8, in that order?
4. Given the statement

```
lst = [10, -4, 11, 29]
```

- (a) What expression represents the very first element of `lst`?
 - (b) What expression represents the very last element of `lst`?
 - (c) What is `lst[0]`?
 - (d) What is `lst[3]`?
 - (e) What is `lst[1]`?
 - (f) What is `lst[-1]`?
 - (g) What is `lst[-4]`?
 - (h) Is the expression `lst[3.0]` legal or illegal?
5. Given the statements

```
lst = [3, 0, 1, 5, 2]
x = 2
```

evaluate the following expressions:

- (a) `lst[0]`?
- (b) `lst[3]`?
- (c) `lst[x]`?
- (d) `lst[-x]`?
- (e) `lst[x + 1]`?
- (f) `lst[x] + 1`?
- (g) `lst[lst[x]]`?
- (h) `lst[lst[lst[x]]]`?

6. What function returns the number of elements in a list?

7. What expression represents the empty list?

8. Given the list

```
lst = [20, 1, -34, 40, -8, 60, 1, 3]
```

evaluate the following expressions:

- (a) `lst`
- (b) `lst[0:3]`
- (c) `lst[4:8]`
- (d) `lst[4:33]`
- (e) `lst[-5:-3]`
- (f) `lst[-22:3]`
- (g) `lst[4:]`
- (h) `lst[:]`
- (i) `lst[:4]`
- (j) `lst[1:5]`
- (k) `-34 in lst`
- (l) `-34 not in lst`
- (m) `len(lst)`

9. An assignment statement containing the expression `a[m:n]` on the left side and a list on the right side can modify list `a`. Complete the following table by supplying the m and n values in the slice assignment statement needed to produce the indicated list from the given original list.

Original List	Target List	Slice indices	
		m	n
[2, 4, 6, 8, 10]	[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]		
[2, 4, 6, 8, 10]	[-10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10]		
[2, 4, 6, 8, 10]	[2, 3, 4, 5, 6, 7, 8, 10]		
[2, 4, 6, 8, 10]	[2, 4, 6, 'a', 'b', 'c', 8, 10]		
[2, 4, 6, 8, 10]	[2, 4, 6, 8, 10]		
[2, 4, 6, 8, 10]	[]		
[2, 4, 6, 8, 10]	[10, 8, 6, 4, 2]		
[2, 4, 6, 8, 10]	[2, 4, 6]		
[2, 4, 6, 8, 10]	[6, 8, 10]		
[2, 4, 6, 8, 10]	[2, 10]		
[2, 4, 6, 8, 10]	[4, 6, 8]		

10. Write the list represented by each of the following expressions.

- (a) `[8] * 4`
- (b) `6 * [2, 7]`
- (c) `[1, 2, 3] + ['a', 'b', 'c', 'd']`
- (d) `3 * [1, 2] + [4, 2]`
- (e) `3 * ([1, 2] + [4, 2])`

11. Write the list represented by each of the following list comprehension expressions.

- (a) `[x + 1 for x in [2, 4, 6, 8]]`
- (b) `[10*x for x in range(5, 10)]`
- (c) `[x for x in range(10, 21) if x % 3 == 0]`
- (d) `[(x, y) for x in range(3) for y in range(4)]`
- (e) `[(x, y) for x in range(3) for y in range(4) if (x + y) % 2 == 0]`

12. Provide a list comprehension expression for each of the following lists.

- (a) `[1, 4, 9, 16, 25]`
- (b) `[0.25, 0.5, 0.75, 1.0, 1.25, 1.5]`
- (c) `[('a', 0), ('a', 1), ('a', 2), ('b', 0), ('b', 1), ('b', 2)]`

13. If `lst` is a list, what expression indicates whether or not `x` is a member of `lst`?

14. What does `reversed` do?

15. Suppose you need to print out tuples that consist of a number and its square, as shown here with the first nine tuples:

```
(0, 0) (1, 1) (2, 4) (3, 9) (4, 16) (5, 25) (6, 36) (7, 49) (8, 64)
```

Which one of the following functions, `fun1` or `fun2`, is more appropriate for the problem at hand, and why?

```
def fun1(n):
    for t in [(x, x**2) for x in range(n)]:
        print(t, end=' ')
    print()
```

```
def fun2(n):
    for t in ((x, x**2) for x in range(n)):
        print(t, end=' ')
    print()
```

16. Complete the following function that adds up all the *positive* values in a list of integers. For example, if list `a` contains the elements `3, -3, 5, 2, -1, and 2`, the call `sum_positive(a)` would evaluate to `12`, since $3 + 5 + 2 + 2 = 12$. The function returns zero if the list is empty.

```
def sum_positive(a):
    # Add your code...
```

17. Complete the following function that counts the even numbers in a list of integers. For example, if list `a` contains the elements 3, 5, 2, -1, and 2, the call `count_evens(a)` would evaluate to 4, since $2 + 2 = 4$. The function returns zero if the list is empty. The function does not affect the contents of the list.

```
def count_evens(a):  
    # Add your code...
```

18. Write a function named `print_big_enough` that accepts two parameters, a list of numbers and a number. The function should print, in order, all the elements in the list that are at least as large as the second parameter.
19. Write a function named `reverse` that reorders the contents of a list so they are reversed from their original order. `a` is a list. Note that your function must physically rearrange the elements within the list, not just print the elements in reverse order.

Chapter 11

Sorting and Searching

Lists, introduced in Chapter 10, are convenient structures for storing large amounts of data. In this chapter we examine several algorithms that allow us to rearrange the elements of a list in a regular way and efficiently search for elements within a list.

11.1 Sorting

Sorting—arranging the elements within a list into a particular order—is a common activity. For example, a list of integers may be arranged in ascending order (that is, from smallest to largest). A list of strings may be arranged in lexicographical (commonly called alphabetical) order. Many sorting algorithms exist, and some perform much better than others. We will consider one sorting algorithm that is relatively easy to describe and implement.

The *selection sort* algorithm is relatively easy to implement and easy to understand how it works. Its performance is acceptable for smaller lists. If A is a list, and i represents a list index, selection sort works as follows:

1. Set n = length of list A .
2. Set $i = 0$.
3. Examine all the elements $A[j]$, where $i < j < n$. (This simply means to consider all the elements in the list from index i to the end.) If any of these elements is less than $A[i]$, then exchange $A[i]$ with the smallest of these elements. (This ensures that all elements after position i are greater than or equal to $A[i]$.)
4. If i is less than $n - 1$, increase i by 1 and go to Step 2.
5. Done; list A is sorted.

The command to “go to Step 2” in Step 4 represents a loop. When the value of i in Step 4 equals n , the algorithm goes to Step 5 and terminates with a sorted list.

We can begin to translate the above description into Python as follows:

```

n = len(A)
for i in range(n - 1):
    # Examine all the elements A[j], where i < j < n.
    # If any of these A[j] is less than A[i],
    # then exchange A[i] with the smallest of these elements.

```

The directive at Step 3 beginning with “Examine all the elements $A[j]$, where $i < j < n$ ” also must be implemented as a loop. We continue refining our implementation with:

```

n = len(A)
for i in range(n - 1):
    # Examine all the elements A[j], where i < j < n.
    for j in range(i + 1, n):
        # Find an element smaller than A[i], if possible
        # If any A[j] is less than A[i],
        # then exchange A[i] with the smallest of these elements.

```

In order to determine if any of the elements is less than $A[i]$, we introduce a new variable named `small`. The purpose of `small` is to keep track of the position of the smallest element found so far. We will set `small` equal to `i` initially because we wish to locate any element less than the element located at position `i`.

```

n = len(A)
for i in range(n - 1):
    # small is the position of the smallest value we've seen
    # so far; we use it to find the smallest value less than A[i]
    small = i
    for j in range(i + 1, n):
        if A[j] < A[small]:
            small = j # Found a smaller element, update small
    # If small changed, we found an element smaller than A[i]
    if small != i:
        # exchange A[small] and A[i]

```

Listing 11.1 (`sortintegers.py`) provides the complete Python implementation of the `selection_sort` function within a program that tests it out.

Listing 11.1: `sortintegers.py`

```

from random import randint

def random_list():
    """
    Produce a list of pseudorandom integers.
    The list's length is chosen pseudorandomly in the
    range 3-20.
    The integers in the list range from -50 to 50.
    """
    result = []
    count = randint(3, 20)
    for i in range(count):
        result += [randint(-50, 50)]
    return result

```

```

def selection_sort(lst):
    """
    Arranges the elements of list lst in ascending order.
    Physically rearranges the elements of lst.
    """
    n = len(lst)
    for i in range(n - 1):
        # Note: i, small, and j represent positions within lst
        # lst[i], lst[small], and lst[j] represent the elements at
        # those positions.
        # small is the position of the smallest value we've seen
        # so far; we use it to find the smallest value less
        # than lst[i]
        small = i
        # See if a smaller value can be found later in the list
        # Consider all the elements at position j, where i < j < n
        for j in range(i + 1, n):
            if lst[j] < lst[small]:
                small = j          # Found a smaller value
        # Swap lst[i] and lst[small], if a smaller value was found
        if i != small:
            lst[i], lst[small] = lst[small], lst[i]

def main():
    """
    Tests the selection_sort function
    """
    for n in range(10):
        col = random_list()
        print(col)
        selection_sort(col)
        print(col)
        print('=====')

main()

```

One run of Listing 11.1 (sortintegers.py) produces:

```

[-23, 47, -3, 4, 5, -46, 26, -27]
[-46, -27, -23, -3, 4, 5, 26, 47]
=====
[32, -10, -4, 41, 10, -1, -31, 3, 28, -31, -33, 46, -45, -6, 37]
[-45, -33, -31, -31, -10, -6, -4, -1, 3, 10, 28, 32, 37, 41, 46]
=====
[11, -19, 20, 43, -19, 20, -18, -17]
[-19, -19, -18, -17, 11, 20, 20, 43]
=====
[9, -22, -41, 35, 10, 48, 9, 14, -20]
[-41, -22, -20, 9, 9, 10, 14, 35, 48]
=====
[-38, -3, -7, 41, -8, -11, -23, 9, -47, 38]
[-47, -38, -23, -11, -8, -7, -3, 9, 38, 41]
=====
[-47, 1, -37, 16, -40, -14, 2, 38, 43, 19, 45]
[-47, -40, -37, -14, 1, 2, 16, 19, 38, 43, 45]

```

```

=====
[8, 39, 35, -42]
[-42, 8, 35, 39]
=====
[-8, -22, -13, 47, -28, -46, -21, -42, 27, 14, 47, -21, 2, -47]
[-47, -46, -42, -28, -22, -21, -21, -13, -8, 2, 14, 27, 47, 47]
=====
[37, -21, -32, -7]
[-32, -21, -7, 37]
=====
[33, -42, -26, 35, 37, 36, -1, 47, 24, 5, 41, -6, 48, 6, 43]
[-42, -26, -6, -1, 5, 6, 24, 33, 35, 36, 37, 41, 43, 47, 48]
=====

```

Notice that in each case the `selection_sort` function rearranges the elements in the pseudorandomly generated list into correct ascending order. To check the correctness of our sort we need to be sure that:

- the sorted list contains the same number of elements as the original, unsorted list,
- no elements in the original list are missing,
- no elements in the sorted list appear more frequently than they did in the original, unsorted list, and
- the elements appear in ascending order.

The output of Listing 11.1 (`sortintegers.py`) provides evidence that our `selection_sort` function is working correctly.

11.2 Flexible Sorting

What if we wish to change the behavior of the sorting function in Listing 11.1 (`sortintegers.py`) so that it arranges the elements in descending order instead of ascending order? It is actually an easy modification; simply change the line

```
if lst[j] < lst[small]:
```

to be

```
if lst[j] > lst[small]:
```

What if instead we want to change the sort so that it sorts the elements in ascending order except that all the even numbers in the list appear before all the odd numbers? This modification would be a little more complicated, but, with some effort, we could modify our `selection_sort` function to achieve this effect.

The next question is more intriguing: How can we rewrite the `selection_sort` function so that, by passing an additional parameter, it can sort the list in any way we want?

We can make our sort function more flexible by passing an ordering function as a parameter (see Section 8.6 for examples of functions as parameters to other functions). Listing 11.2 (`flexiblesort.py`) arranges the elements in a list two different ways using the same `selection_sort` function.

Listing 11.2: `flexiblesort.py`

```

def random_list():
    '''
    Produce a list of pseudorandom integers.
    The list's length is chosen pseudorandomly in the
    range 3-20.
    The integers in the list range from -50 to 50.
    '''
    from random import randrange
    result = []
    count = randrange(3, 20)
    for i in range(count):
        result += [randrange(-50, 50)]
    return result

def less_than(m, n):
    ''' Returns true if m is less than n; otherwise, returns false '''
    return m < n

def greater_than(m, n):
    ''' Returns true if m is greater than n; otherwise, returns false '''
    return m > n

def selection_sort(lst, cmp):
    '''
    Arranges the elements of list lst in ascending order.
    The comparer function cmp is used to order the elements.
    The contents of lst are physically rearranged.
    '''
    n = len(lst)
    for i in range(n - 1):
        # Note: i, small, and j represent positions within lst
        # lst[i], lst[small], and lst[j] represent the elements at
        # those positions.
        # small is the position of the smallest value we've seen
        # so far; we use it to find the smallest value less
        # than lst[i]
        small = i
        # See if a smaller value can be found later in the list
        # Consider all the elements at position j, where i < j < n.
        for j in range(i + 1, n):
            if cmp(lst[j], lst[small]):
                small = j          # Found a smaller value
        # Swap lst[i] and lst[small], if a smaller value was found
        if i != small:
            lst[i], lst[small] = lst[small], lst[i]

def main():
    '''
    Tests the selection_sort function
    '''
    original = random_list()      # Make a random list
    working = original[:]         # Make a working copy of the list
    print('Original: ', working)

```

```

selection_sort(working, less_than) # Sort ascending
print('Ascending: ', working)
working = original[:] # Make a working copy of the list
print('Original: ', working)
selection_sort(working, greater_than) # Sort descending
print('Descending:', working)

main()

```

The output of Listing 11.2 (flexiblesort.py) is

```

Original:  [-8, 24, -46, -7, -26, -29, -44]
Ascending: [-46, -44, -29, -26, -8, -7, 24]
Original:  [-8, 24, -46, -7, -26, -29, -44]
Descending: [24, -7, -8, -26, -29, -44, -46]

```

The comparison function passed to the sort routine customizes the sort's behavior. The basic structure of the sorting algorithm does not change, but its notion of ordering is adjustable. If the second parameter to `selection_sort` is `less_than`, the function arranges the elements ascending order. If the second parameter instead is `greater_than`, the function sorts the list in descending order. More creative orderings are possible with more elaborate comparison functions.

Selection sort is a relatively efficient simple sort, but more advanced sorts are, on average, much faster than selection sort, especially for large data sets. One such general purpose sort is *Quicksort*, devised by C. A. R. Hoare in 1962. Quicksort is the fastest known general purpose sort.

11.3 Faster Sorting

CAUTION! SECTION UNDER CONSTRUCTION

Listing 11.3 (sortcompare.py) implements the quicksort algorithm.

Listing 11.3: sortcompare.py

```

''' Compares the execution time of selection sort to that of
    Quicksort '''

from stopwatch import Stopwatch

def selection_sort(lst):
    ''' Arranges the elements of list lst in ascending order.
        Physically rearranges the elements of lst.
        Uses the selection sort algorithm. '''
    n = len(lst)
    for i in range(n - 1):
        # Note: i, small, and j represent positions within lst
        # lst[i], lst[small], and lst[j] represent the elements at
        # those positions.
        # small is the position of the smallest value we've seen
        # so far; we use it to find the smallest value less
        # than lst[i]
        small = i

```

```

    # See if a smaller value can be found later in the list
    # Consider all the elements at position j, where i < j < n
    for j in range(i + 1, n):
        if lst[j] < lst[small]:
            small = j        # Found a smaller value
    # Swap lst[i] and lst[small], if a smaller value was found
    if i != small:
        lst[i], lst[small] = lst[small], lst[i]

def qsort(lst, left, right):
    ''' Arranges the elements within the segment from index left to index
        right of list lst into ascending order.
        Physically rearranges the elements of lst.
        This is the recursive helper function for the quicksort function.
        Uses the quicksort algorithm. '''
    i = left                # i starts at the front
    j = right               # j starts at the back
    x = lst[(left + right)//2] # x is the element in the middle
    while i <= j:
        while lst[i] < x:    # Position of lst[i] OK relative to x
            i += 1          # i moves towards the back
        while x < lst[j]:    # Position of lst[j] OK relative to x
            j -= 1          # j moves towards the front
        if i <= j:          # i and j have not passed each other
            lst[i], lst[j] = lst[j], lst[i]
            i += 1
            j -= 1
    if left < j:            # Has j reached the front?
        qsort(lst, left, j) # Recursively sort the front part
    if i < right:           # Has i reached the back?
        qsort(lst, i, right) # Recursively sort the front part

def quicksort(lst):
    ''' Arranges the elements of list lst in ascending order.
        Physically rearranges the elements of lst.
        Uses the quicksort algorithm via the recursive qsort function. '''
    if len(lst) > 1:        # No need to sort a list containing fewer than two elements
        qsort(lst, 0, len(lst) - 1)

def permute(lst):
    ''' Randomly permutes the contents of list lst '''
    from random import randrange
    n = len(lst)
    for i in range(n - 1):
        pos = randrange(i, n) # i <= pos < n
        lst[i], lst[pos] = lst[pos], lst[i]

def is_ascending(lst):
    ''' Returns true if all the elements of list lst appear in
        ascending order; otherwise, returns false. '''
    for i in range(1, len(lst)):
        if (lst[i - 1] > lst[i]):
            return False    # Found out-of-order elements
    return True             # All elements in proper order

```

```

def compare_sorts(nums):
    ''' Compares the execution speed of selection sort and
        quicksort on list nums. '''
    timer = Stopwatch()
    permute(nums)
    copy1 = nums[:]
    copy2 = nums[:]
    print('Selection sort:', is_ascending(copy1), end=' ')
    timer.start()
    selection_sort(copy1)
    timer.stop()
    print(is_ascending(copy1), end=' ')
    print('{0:.2f} seconds elapsed'.format(timer.elapsed()))
    timer.reset()
    print('Quicksort', is_ascending(copy2), end=' ')
    timer.start()
    quicksort(copy2)
    timer.stop()
    print(is_ascending(copy2), end=' ')
    print('{0:.2f} seconds elapsed'.format(timer.elapsed()))

def main():
    ''' Compares the execution speed of selection sort and
        quicksort on lists of various sizes. '''
    for i in range(0, 10001, 1000):
        print('---- List size:', i, '-----')
        lst = list(range(i))
        compare_sorts(lst)

if __name__ == '__main__':
    main()

```

11.4 Search

Searching a list for a particular element is a common activity. We examine two basic strategies: linear search and binary search.

11.4.1 Linear Search

Listing 11.4 (linearsearch.py) uses a function named `locate` that returns the position of the first occurrence of a given element in a list; if the element is not present, the function returns `None`.

Listing 11.4: linearsearch.py

```

def locate(lst, seek):
    '''
    Returns the index of element seek in list lst,
    if seek is present in lst.
    Returns None if seek is not an element of lst.

```

```

    lst is the list in which to search.
    seek is the element to find.
    """
    for i in range(len(lst)):
        if lst[i] == seek:
            return i      # Return position immediately
    return None           # Element not found

def format(i):
    """
    Prints integer i right justified in a 4-space
    horizontal area. Prints "****" if i > 9,999.
    """
    if i > 9999:
        print("****")      # Too big!
    else:
        print("{0:4d}".format(i))

def show(lst):
    """
    Prints the contents of list lst
    """
    for item in lst:
        print("{0:4d}".format(item), end='') # Print element right justifies in 4 spaces
    print()                                # Print newline

def draw_arrow(value, n):
    """
    Print an arrow to value which is an element in a list.
    n specifies the horizontal offset of the arrow.
    """
    print((' {0:>}' + str(n) + ' ').format("  ^  "))
    print((' {0:>}' + str(n) + ' ').format("  |  "))
    print((' {0:>}' + str(n) + ' ').format("  +-- ", value))

def display(lst, value):
    """
    Draws an ASCII art arrow showing where
    the given value is within the list.
    lst is the list.
    value is the element to locate.
    """
    show(lst)                # Print contents of the list
    position = locate(lst, value)
    if position != None:
        position = 4*position + 7 # Compute spacing for arrow
        draw_arrow(value, position)
    else:
        print("(" + value + " not in list)", sep='')
    print()

def main():
    a = [100, 44, 2, 80, 5, 13, 11, 2, 110]

```

```

    display(a, 13)
    display(a, 2)
    display(a, 7)
    display(a, 100)
    display(a, 110)

main()

```

Note, for example, that if n is 20, the expression

```
'{0:>}' + str(n) + '}'.format("  ^  ")
```

right justifies the string ' ^ ' within 20 spaces; that is,

```
'      ^      '
```

The output of Listing 11.4 (linearssearch.py) is

```

100 44  2 80  5 13 11  2 110
                ^
                |
                +-- 13

100 44  2 80  5 13 11  2 110
          ^
          |
          +-- 2

100 44  2 80  5 13 11  2 110
(7 not in list)

100 44  2 80  5 13 11  2 110
  ^
  |
  +-- 100

100 44  2 80  5 13 11  2 110
                        ^
                        |
                        +-- 110

```

The key function in Listing 11.4 (linearssearch.py) is `locate`; all the other functions simply lead to a more interesting display of `locate`'s results. If `locate` finds a match, the function immediately returns the position of the matching element; otherwise, if after examining all the elements of the list `locate` cannot find the element sought, the function returns `None`. Here `None` indicates the function could not return a valid answer. The calling code, in this example the `display` function, must ensure that `locate`'s result is not `None` before attempting to use the result as an index into a list.

The kind of search performed by `locate` is known as *linear search*, since the algorithm takes a straight line path from the beginning of the list to the end of the list considering each element in order. Figure 11.1 illustrates linear search.

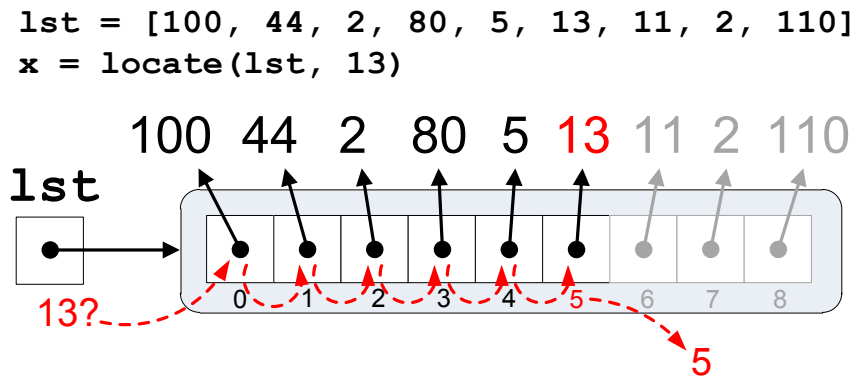


Figure 11.1: Linear search first considers the element at index 0, then index 1, then index 2, etc. until it finds the element it seeks or reaches the back of the list. The algorithm progresses through the list in a straight line without jumping around.

11.4.2 Binary Search

Linear search is acceptable for relatively small lists, but the process of examining each element in a large list is time consuming. An alternative to linear search is *binary search*. In order to perform binary search, a list must be in sorted order. Binary search exploits the sorted structure of the list using a clever but simple strategy that quickly zeros in on the element to find:

1. If the list is empty, return `None`.
2. Check the element in the middle of the list. If that element is what you are seeking, return its position. If the middle element is larger than the element you are seeking, perform a binary search on the first half of the list. If the middle element is smaller than the element you are seeking, perform a binary search on the second half of the list.

This approach is analogous to looking for a telephone number in the phone book in this manner:

1. Open the book at its center. If the name of the person is on one of the two visible pages, look at the phone number.
2. If not, and the person's last name is alphabetically less the names on the visible pages, apply the search to the left half of the open book; otherwise, apply the search to the right half of the open book.
3. Discontinue the search with failure if the person's name should be on one of the two visible pages but is not present.

We can implement the binary search algorithm as a Python function as shown in Listing 11.5 (`binarysearch.py`).

Listing 11.5: `binarysearch.py`

```
def binary_search(lst, seek):
```

```

'''
Returns the index of element seek in list lst,
if seek is present in lst.
Returns None if seek is not an element of lst.
lst is the list in which to search.
seek is the element to find.
'''
first = 0          # Initialize the first position in list
last = len(lst) - 1 # Initialize the last position in list
while first <= last:
    # mid is middle position in the list
    mid = first + (last - first + 1)//2 # Note: Integer division
    if lst[mid] == seek:
        return mid # Found it
    elif lst[mid] > seek:
        last = mid - 1 # continue with 1st half
    else: # v[mid] < seek
        first = mid + 1 # continue with 2nd half
return None # Not there

def format(i):
    '''
    Prints integer i right justified in a 4-space
    horizontal area. Prints "****" if i > 9,999.
    '''
    if i > 9999:
        print("****") # Too big!
    else:
        print("{0:4d}".format(i))

def show(lst):
    '''
    Prints the contents of list lst
    '''
    for item in lst:
        print("{0:4d}".format(item), end='') # Print element right justifies in 4 spaces
    print() # Print newline

def draw_arrow(value, n):
    '''
    Print an arrow to value which is an element in a list.
    n specifies the horizontal offset of the arrow.
    '''
    print((' {0:>}' + str(n) + ' ').format(" ^ "))
    print((' {0:>}' + str(n) + ' ').format(" | "))
    print((' {0:>}' + str(n) + ' {1}').format(" +-- ", value))

def display(lst, value):
    '''
    Draws an ASCII art arrow showing where
    the given value is within the list.
    lst is the list.

```

```

    value is the element to locate.
    ...

    show(lst)                # Print contents of the list
    position = binary_search(lst, value)
    if position != None:
        position = 4*position + 7    # Compute spacing for arrow
        draw_arrow(value, position)
    else:
        print("(", value, " not in list)", sep='')
    print()

def main():
    a = [2, 5, 11, 13, 44, 80, 100, 110]
    display(a, 13)
    display(a, 2)
    display(a, 7)
    display(a, 100)
    display(a, 110)

main()

```

In the `binary_search` function:

- The initializations of `first` and `last`:

```

first = 0          # Initialize the first position in list
last = len(lst) - 1 # Initialize the last position in list

```

ensure that `first` is less than or equal to `last` for a nonempty list. If the list is empty, `first` is zero, and `last` is equal to `len(lst) - 1 = 0 - 1 = -1`. So in the case of an empty list the function will skip the loop and return `None`. This is correct behavior because an empty list cannot possibly contain any item we seek.

- The calculation of `mid` ensures that $\text{first} \leq \text{mid} \leq \text{last}$.
- If `mid` is the location of the sought element (checked in the first `if` statement), the loop terminates, and returns the correct position.
- The `elif` and `else` blocks ensure that either `last` decreases or `first` increases each time through the loop. Thus, if the loop does not terminate for other reasons, eventually `first` will be larger than `last`, and the loop will terminate. If the loop terminates for this reason, the function returns `None`. This is the correct behavior.
- The modification to either `first` or `last` in the `elif` and `else` blocks exclude irrelevant elements from further search. The number of elements to consider is cut in half each time through the loop.

Figure 11.2 illustrates how binary search works.

The implementation of the binary search algorithm is more complicated than the simpler linear search algorithm. Ordinarily simpler is better, but for algorithms that process data structures that potentially hold large amounts of data, more complex algorithms employing clever tricks that exploit the structure of the data (as binary search does) often dramatically outperform simpler, easier-to-code algorithms.

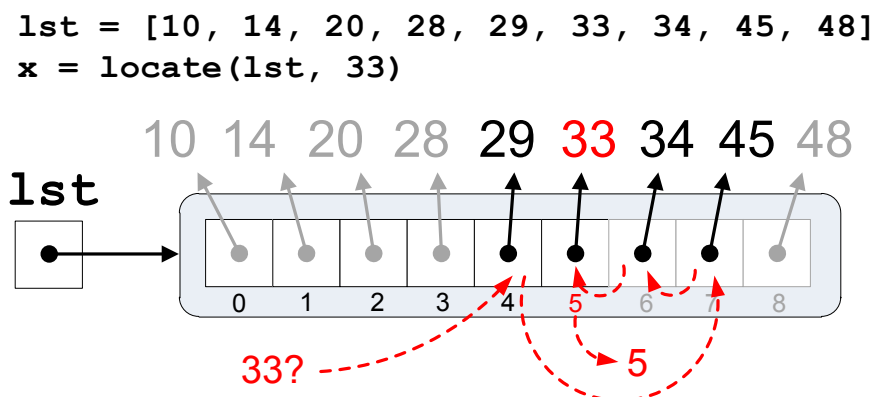


Figure 11.2: Binary search begins in the middle of an ordered list. The algorithm jumps forward or backward as needed in decreasing stride amounts. Each successive probe reduces the number of elements in its search space by one-half.

For a fair comparison of linear vs. binary search, suppose we want to locate an element in a sorted list. An ordered list is essential for binary search, but it can be helpful for linear search as well. The revised linear search algorithm for ordered lists is

```
# This version requires list lst to be sorted in
# ascending order.
def linear_search(lst, seek):
    i = 0                # Start at beginning
    n = len(lst)         # Length of list
    while i < n and lst[i] <= seek:
        if lst[i] == seek:
            return i      # Return position immediately
    return None          # Element not found
```

Notice that, as in the original version of linear search, the loop will terminate when it has examined all the elements, but this version will terminate early when it encounters an element larger than the sought element. Since the list is sorted, there is no need to continue the search once the search has found an element larger than the value sought; seek cannot appear after a larger element in a sorted list.

Suppose a list to search contains n elements. In the worst case—looking for an element larger than any currently in the list—the loop in linear search takes n iterations. In the best case—looking for an element smaller than any currently in the list—the function immediately returns without considering any other elements. The number of loop iterations thus ranges from 1 to n , and so on average linear search requires $\frac{n}{2}$ comparisons before the loop finishes and the function returns.

Now consider binary search. After each comparison the size of the list remaining to consider is one-half the original size. If the binary search algorithm does not locate the element on its first probe, the number of remaining elements to search is $\frac{n}{2}$. The next time through the loop, the number of elements left to consider drops to $\frac{n}{4}$, then $\frac{n}{8}$, and so forth. The problem of determining how many times a set of things can be divided in half until only one element remains can be solved with a base-2 logarithm. For binary search, the worst

case scenario of not finding the sought element requires the loop to make $\log_2 n$ iterations.

How does this analysis help us determine which search is better? The quality of an algorithm is judged by two key characteristics:

- How much time (processor cycles) does it take to run?
- How much space (memory) does it take to run?

In our situation, both search algorithms process the list with only a few extra local variables, so for large lists they both require essentially the same space. The big difference here is speed. Binary search performs more elaborate computations each time through the loop, and each operation takes time, so perhaps binary search is slower. Linear search is simpler (fewer operations through the loop), but perhaps its loop executes many more times than the loop in binary search, so overall it is slower.

We can deduce the faster algorithm in two ways: empirically and analytically. An empirical test is an experiment; we carefully implement both algorithms and then measure their execution times. The analytical approach analyzes the source code to determine how many operations the computer's processor must perform to run the program on a problem of a particular size.

Listing 11.6 (searchcompare.py) gives us some empirical results.

Listing 11.6: searchcompare.py

```
'''
Compares the running times of linear search and
binary search on lists of various sizes.
'''

def binary_search(lst, seek):
    '''
    Returns the index of element seek in list lst,
    if seek is present in lst.
    lst must be in sorted order.
    Returns None if seek is not an element of lst.
    lst is the list in which to search.
    seek is the element to find.
    '''
    first = 0          # Initially the first element in list
    last = len(lst) - 1 # Initially the last element in list
    while first <= last:
        # mid is middle of the list
        mid = first + (last - first + 1)//2 # Note: Integer division
        if lst[mid] == seek:
            return mid      # Found it
        elif lst[mid] > seek:
            last = mid - 1   # continue with 1st half
        else: # v[mid] < seek
            first = mid + 1  # continue with 2nd half
    return None            # Not there

def ordered_linear_search(lst, seek):
    '''
    Returns the index of element seek in list lst,
    if seek is present in lst.
```

```

    lst must be in sorted order.
    Returns None if seek is not an element of lst.
    lst is the list in which to search.
    seek is the element to find.
    '''
    i = 0
    n = len(lst)
    while i < n and lst[i] <= seek:
        if lst[i] == seek:
            return i        # Return position immediately
        i += 1
    return None            # Element not found

def run_search(lst, seeks, search, trials):
    '''
    Searches for all the elements in an ordered list (lst)
    using search function search. Averages the running time over
    trials runs. Returns the average time.
    '''
    from time import clock
    n = len(lst)
    elapsed = 0
    start = clock()        # Start the clock
    for i in range(trials):
        for elem in seeks:
            i = search(lst, elem)
            if i != lst[i]:
                print("error")
    stop = clock()         # Stop the clock
    elapsed += stop - start
    return elapsed/trials   # Average time for search

def test_searches(lst, seeks, trials):
    '''
    Measures the running times of ordered linear search and binary
    search on a given list. Averages the times over n runs.
    '''
    # Find each element using ordered linear search
    lin = run_search(lst, seeks, ordered_linear_search, trials)
    # Find each element using binary search
    bin = run_search(lst, seeks, binary_search, trials)
    # Print the results
    print('{0:6} {1:10.5f} {2:10.5f} {3:8.1f}'\
          .format(len(lst), lin, bin, lin/bin))

def make_search_set(n):
    '''
    Make a list of elements to seek
    '''
    from random import randrange
    result = []
    for i in range(n):
        result += [randrange(n)]
    return result

```

```

def main():
    """
    Makes a table comparing the running times of ordered linear
    search vs. binary search on lists of various sizes.
    """

    # Number of trials over which to average the results
    trials = 10

    # Print table header
    print('  Size   Linear   Binary   Speedup')
    print('-----')
    # Small lists: 10 to 100, in steps of 10
    for size in range(10, 100, 10):
        test_list = list(range(size))
        seek_list = make_search_set(size)
        test_searches(test_list, seek_list, trials)
    # Medium lists: 100 to 1,000, in steps of 100
    for size in range(100, 1000, 100):
        test_list = list(range(size))
        seek_list = make_search_set(size)
        test_searches(test_list, seek_list, trials)
    # Large lists: 1,000 to 5,000, in steps of 500
    for size in range(1000, 5001, 500):
        test_list = list(range(size))
        seek_list = make_search_set(size)
        test_searches(test_list, seek_list, trials)

if __name__ == '__main__':
    main()

```

The main function in Listing 11.6 (searchcompare.py) builds lists of sequential integer values, [1, 2, 3, ...] of various sizes.

The program assigns each of these lists in turn to the `test_list` variable. The program also builds lists of random integer values (referenced via `seek_list`) to be used as search candidates for each `test_list`. It then passes these lists off to the `test_searches` function to measure the running times of the two search functions. The `test_searches` function, in turn, calls the `run_search` function to test a particular search function. The `run_search` function uses the elements from main's `seek_list` as search candidates. The `run_search` function searches for all the elements in `seek_list` a specified number of times and averages the running times. The main function directs `test_searches` to average 10 runs for each of the list sizes.

On one system, Listing 11.6 (searchcompare.py) produces the following table:

Size	Linear	Binary	Speedup
10	0.00003	0.00003	1.2
20	0.00012	0.00006	2.0
30	0.00024	0.00012	1.9
40	0.00036	0.00016	2.3
50	0.00049	0.00021	2.3
60	0.00082	0.00026	3.1
70	0.00104	0.00032	3.2
80	0.00138	0.00039	3.6
90	0.00202	0.00043	4.7

100	0.00245	0.00049	5.0
200	0.00868	0.00115	7.5
300	0.01962	0.00185	10.6
400	0.03215	0.00262	12.3
500	0.05158	0.00342	15.1
600	0.07590	0.00437	17.4
700	0.10805	0.00522	20.7
800	0.13329	0.00600	22.2
900	0.17307	0.00687	25.2
1000	0.20985	0.00780	26.9
1500	0.49346	0.01256	39.3
2000	0.85834	0.01739	49.4
2500	1.39785	0.02284	61.2
3000	1.95955	0.02802	69.9
3500	2.71689	0.03321	81.8
4000	3.56608	0.03960	90.1
4500	4.45774	0.04446	100.3
5000	7.73395	0.04983	155.2

The rightmost column of the table shows the speedup factor of binary search over ordered linear search. Notice that the speedup increases as the list length grows. For lists that contain more than 4,500 elements binary search is more than 100 times faster than linear search.

Empirically, binary search performs dramatically better than linear search. The left side of Figure 11.3 plots the results produced by Listing 11.6 (`searchcompare.py`) for lists containing up to 1,000 elements.

In addition to empirical observations, we can judge which algorithm is better by analyzing the source code for each function. Each arithmetic operation, assignment, logical comparison, function call, and list access requires time to execute. We will assume each of these activities requires one unit of processor “time.” This assumption is not strictly true, but it will give good results for relative comparisons. Since we will follow the same rules when analyzing both search algorithms, the relative results for comparison purposes will be fairly accurate.

We first consider linear search. We determined that, on average, the loop makes $\frac{n}{2}$ iterations for a list of size n . The initialization of `i` happens only one time during each call to `linear_search`. All other activity involved with the loop except the `return` statements happens $\frac{n}{2}$ times. The function returns either `i` or `None`, and it may execute at most one `return` statement during each call. Table 11.1 shows the breakdown for linear search. The results in Table 11.1 indicate the running time of the `linear_search` function can be expressed as a simple mathematical linear function: $f(n) = 3n + 4$.

Next, we consider binary search. We determined that in the worst case the loop in `binary_search` iterates $\log_2 n$ times if the list contains n elements. The `binary_search` function performs the two initializations before the loop just once per call. Most of the actions within the loop occur $\log_2 n$ times, except that only one `return` statement can be executed per call, and in the `if/elif/else` statement only one path can be chosen per loop iteration. Table 11.2 shows the complete analysis of binary search. We see that the execution time for binary search can be expressed as the logarithmic function $12\log_2 n + 6$.

Figure 11.3 compares the empirical results with the analytical results for lists containing 100 to 1,000 elements. The left side of Figure 11.3 plots the values produced by Listing 11.6 (`searchcompare.py`), and the right side of Figure 11.3 plots the two functions $3n + 4$ and $12\log_2 n + 6$. In these two graphs we can compare the growth rates of the two search techniques by examining the shapes of the curves. Notice how closely the two graphs compare to each other. In both graphs the gap between the linear search curve and binary search curve increasingly widens at the same rate as the list size increases. The binary search curve appears to be effectively flat, although it really is growing very slowly, much more slowly than the linear

Action	Operation(s)	Operation Count	Times Executed	Total Cost
<code>i = 0</code>	<code>=</code>	1	1	1
<code>n = len(lst)</code>	<code>=, len</code>	2	1	2
<code>while i < n and lst[i] <= seek:</code>	<code><, and, <=, []</code>	4	$\frac{n}{2}$	$2n$
<code>if lst[i] == seek:</code>	<code>[], ==</code>	2	$\frac{n}{2}$	n
<code>return i</code>	<code>return</code>	1	$\frac{1}{2}$	$\frac{1}{2}$
<code>return None</code>	<code>return</code>	1	$\frac{1}{2}$	$\frac{1}{2}$
Total time units				$3n + 4$

Table 11.1: Analysis of Linear Search Algorithm. The $\frac{n}{2}$ loop iterations is based on the average time to locate an element. The function will execute exactly one of the two return statements during a given call, so each is given a cost of $\frac{1}{2}$.

Action	Operation(s)	Operation Count	Times Executed	Total Cost
<code>first = 0</code>	<code>=</code>	1	1	1
<code>last = len(lst) - 1</code>	<code>=, len, -</code>	3	1	3
<code>while first <= last:</code>	<code><=</code>	1	$\log_2 n$	$\log_2 n$
<code>mid=first+(last-first+1)//2</code>	<code>=, +, -, +, //</code>	5	$\log_2 n$	$5\log_2 n$
<code>if lst[mid] == seek:</code>	<code>[], ==</code>	2	$\log_2 n$	$2\log_2 n$
<code>return mid</code>	<code>return</code>	1	1	1
<code>elif lst[mid] > seek:</code>	<code>[], ></code>	2	$\log_2 n$	$2\log_2 n$
<code>last = mid - 1</code>	<code>=, -</code>	2	$\frac{1}{2} \log_2 n$	$\log_2 n$
<code>else:</code>		0		0
<code>first = mid + 1</code>	<code>=, +</code>	2	$\frac{1}{2} \log_2 n$	$\log_2 n$
<code>return None</code>	<code>return</code>	1	1	1
Total time units				$12\log_2 n + 6$

Table 11.2: Analysis of Binary Search Algorithm. Each time through the loop the function executes either the elif or else statement, so each one is charged is charged $\frac{1}{2}$ its actual cost.

search curve. The bottom line is that binary search is fast even for large lists.

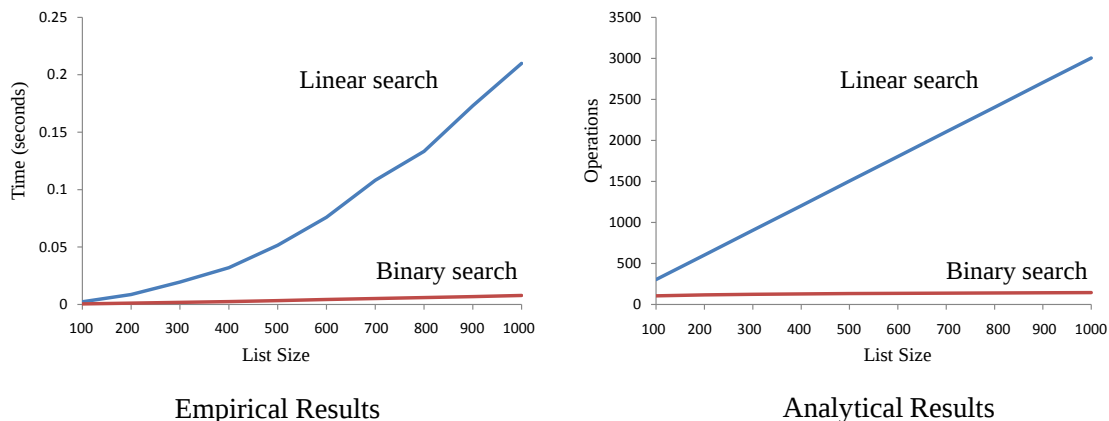


Figure 11.3: Linear search vs. binary search. The two graphs plot the execution speeds of ordered linear search and binary search on lists with 100 to 1,000 elements. The graph on the left plots data from timing the program’s execution. The graph on the right plots the functions derived from analyzing the Python source code. Notice how closely the two graphs correspond.

11.5 Recursion Revisited

In Section 8.3 we saw recursive functions for factorial and greatest common divisor. Suppose we have a recursive function named `f` that accepts a single parameter. Recall that recursion works in the following manner:

1. If function `f`’s argument selects the base case of the recursion, return the default answer;
2. otherwise, do something with the argument and invoke `f` with an argument that is closer to the base case.

We can restate this in a more informal way:

1. If `f`’s argument represents a trivial problem, return the default, easy answer.
2. If `f`’s argument represents a non-trivial problem, return the result of computing part of the problem and combining it with the solution to a smaller or simpler problem.

The phrase “the solution to a smaller or simpler problem” is the recursive call. As the recursion progresses, the function works on smaller and/or simpler problems until it reaches a trivial problem, at which point the recursive process is over.

Many data structures lend themselves to recursive algorithms. Consider the task of counting the occurrences of a particular element within a list. We will name the function `count`, and it will accept a list and an

additional parameter that represents the element to count. Given a correctly implemented count function, the following code fragment,

```
lst1 = [21, 19, 31, 22, 14, 31, 22, 6, 31]
print(count(lst1, 31))
lst2 = ['FRED', [2, 3], 44, 'WILMA', 'FRED', 8, 'BARNEY']
print(count(lst2, 'FRED'))
print(count(lst2, 'BETTY'))
print(count([], 16))
```

should print

```
3
2
0
0
```

We will think about the problem *recursively*. What should the function do if the list is empty? Is this a trivial problem? What should the function do if the list is not empty?

If the list is empty, the problem is trivial. No matter what we are trying to count, it will not appear in an empty list. In this case the count function simply can return zero.

What if the list is not empty? We can look at the first element. If the first element is equal to the value we wish to count, we know we have one occurrence, so the answer is one plus the number of times the value appears in the rest of the list. If the first element is not equal to the value we wish to count, the answer is just the number of times the value appears in the rest of the list.

Do you see that “the number of times the value appears in the rest of the list” is a recursive call on a shorter list? Each recursive call processes a shorter and shorter list until the list is empty (the trivial case). All along the chain of recursive function calls the function is either adding one or not adding one to count’s ultimate answer. Listing 11.7 (recursivecount.py) implements our recursive count function.

Listing 11.7: recursivecount.py

```
def count(lst, item):
    ''' Counts the number of occurrences of item within the list lst '''
    if len(lst) == 0:      # Is the list empty?
        return 0          # Nothing can appear in an empty list
    else:
        # Count the occurrences in the rest of the list
        # (all but the first element)
        count_rest = count(lst[1:], item)
        if lst[0] == item:
            return 1 + count_rest
        else:
            return count_rest

def main():
    lst1 = [21, 19, 31, 22, 14, 31, 22, 6, 31]
    print(count(lst1, 31))
    lst2 = ['FRED', [2, 3], 44, 'WILMA', 'FRED', 8, 'BARNEY']
    print(count(lst2, 'FRED'))
    print(count(lst2, 'BETTY'))
    print(count([], 16))
```

```
if __name__ == '__main__':
    main()
```

The expression `count([21, 19, 31, 14, 31, 6, 31], 31)` would evaluate as

```
count([21, 19, 31, 14, 31, 6, 31], 31) = count([19, 31, 14, 31, 6, 31], 31)
                                         = count([31, 14, 31, 6, 31], 31)
                                         = 1 + count([14, 31, 6, 31], 31)
                                         = 1 + count([31, 6, 31], 31)
                                         = 1 + 1 + count([6, 31], 31)
                                         = 1 + 1 + count([31], 31)
                                         = 1 + 1 + 1 + count([], 31)
                                         = 1 + 1 + 1 + 0
                                         = 1 + 1 + 1
                                         = 1 + 2
                                         = 3
```

While the `count` function in Listing 11.7 (`recursivecount.py`) works properly, it has one, potentially big disadvantage. Each time the function's execution selects the recursive route it slices the list. Slicing a list creates a copy of the list (see Section 10.7). This means every call to `count` in the recursive call chain makes a complete copy of the list, except for the first element of each successive list. If the list is long, this can unnecessarily consume a large amount of the computer's memory. How big can it get? Consider an initial call to `count` that passes a list of 1,000 elements. The first recursive call passes a new list of 999 elements. The second recursive call passes a new list of 998 elements, and so forth. By the time it completes, the `count` will have created 999 extra lists, holding a combined total of 499,500 elements. The space required to process the list is about 500 times the size of original list. Not only does the recursion use more memory, copying the list takes time. This excessive list copying slows down the program's execution.

It would be better to implement the `count` function so that it does not make any copies. Listing 11.8 (`inplacecount.py`) implements a recursive count function that makes no copies of the list.

Listing 11.8: `inplacecount.py`

```
def count_helper(lst, pos, item):
    ''' Counts the number of occurrences of item within the list
        lst. pos represents the current position under
        examination within the list. '''
    if pos == len(lst): # Are we past the end of the list?
        return 0        # Nothing can appear past the end
    else:
        # Count the occurrences in the rest of the list
        # (all but the first element)
        count_rest = count_helper(lst, pos + 1, item)
        if lst[pos] == item:
            return 1 + count_rest
        else:
            return count_rest

def count(lst, item):
    ''' Counts the number of occurrences of item within the list
        lst. Delegates the work to the recursive count_helper
        function, passing zero as the initial position (which is
        the index of the first element in the list). '''
```

```

    return count_helper(lst, 0, item)

def main():
    lst1 = [21, 19, 31, 22, 14, 31, 22, 6, 31]
    print(count(lst1, 31))
    lst2 = ['FRED', [2, 3], 44, 'WILMA', 'FRED', 8, 'BARNEY']
    print(count(lst2, 'FRED'))
    print(count(lst2, 'BETTY'))
    print(count([], 16))

if __name__ == '__main__':
    main()

```

Listing 11.8 (inplacecount.py) uses two functions to do the counting. Its count function merely calls count_helper with the proper initial parameters. The count_helper function does all the interesting work. Instead of creating copies of the list, count_helper accepts an additional parameter, an index, for the recursion to keep track of its position within the list. The list parameter is an alias of the original list, not a copy. When a function can process a list without making a copy, we say the function processes the list *in place*.

You may be thinking that it would be simpler to implement our counting function with a loop in a manner similar to linear search:

```

def count(lst, item):
    cnt = 0    # Initialize item count
    for elem in lst:
        if elem == item:
            cnt += 1    # Found an item, count it
    return cnt

```

This processes the list in place and does not use recursion. This version of count actually is superior to both recursive versions. As we saw in Section 8.3, every function call requires a little extra time and memory. If two functions—one iterative and one recursive—faithfully implement the same algorithm, the iterative version will be more efficient.

A recursive function does have one distinct advantage over a non-recursive function, though. A recursive function does not just call itself; the self-call eventually returns back to the site of its invocation. Each recursive invocation has its own parameters and creates its own local variables. When the function returns to itself, it “remembers” its original parameters and local variables the way they were before the recursive invocation. We say the function *unwinds* back to its previous state.

Listing 11.9 (recursivememory.py) demonstrates the unwinding power of recursion.

Listing 11.9: recursivememory.py

```

from random import randint

def rec(n, depth):
    ''' Prints the value of the parameter n and local variable
        rand before and after the recursive call.
        n is the parameter of interest.
        depth represents the depth of the recursion. '''
    rand = randint(0, 1000)    # Make a random number
    print(' ' * depth, 'Entering: n =', n, ' rand =', rand)
    if n == 0:

```

```

        print(' ' * depth, ' *** Recursion over ***')
    else:
        rec(n - 1, depth + 1)
    print(' ' * depth, 'Exiting: n =', n, ' rand =', rand)

rec(10, 0)

```

The `rec` function in Listing 11.9 (`recursivememory.py`) is recursive. The first parameter, `n`, is the parameter of interest. The second parameter, `depth`, reflects the depth of the recursion. The `depth` variable controls the indentation level of each printed line.

Listing 11.9 (`recursivememory.py`) prints

```

Entering: n = 10  rand = 716
  Entering: n = 9  rand = 970
    Entering: n = 8  rand = 21
      Entering: n = 7  rand = 835
        Entering: n = 6  rand = 11
          Entering: n = 5  rand = 759
            Entering: n = 4  rand = 168
              Entering: n = 3  rand = 65
                Entering: n = 2  rand = 86
                  Entering: n = 1  rand = 238
                    Entering: n = 0  rand = 991
                      *** Recursion over ***
                        Exiting: n = 0  rand = 991
                          Exiting: n = 1  rand = 238
                            Exiting: n = 2  rand = 86
                              Exiting: n = 3  rand = 65
                                Exiting: n = 4  rand = 168
                                  Exiting: n = 5  rand = 759
                                    Exiting: n = 6  rand = 11
                                      Exiting: n = 7  rand = 835
                                        Exiting: n = 8  rand = 21
                                          Exiting: n = 9  rand = 970
                                            Exiting: n = 10  rand = 716

```

Observe that the unwinding of the recursive calls restores the original values of the parameter `n` and local variable `rand`. Since `rand` is assigned pseudo-probabilistically, it is not possible to restore its original value without first storing it somewhere for later retrieval. The function-call-and-return process automatically takes care of saving and restoring the previous values of local variables.

It is more difficult to write a non-recursive function that has this ability to unwind itself to a previous program state. Such non-recursive implementations usually offer no efficiency advantages. Listing 11.10 (`nonrecursivememory.py`) provides a non-recursive version of the `rec` function.

Listing 11.10: `nonrecursivememory.py`

```

from random import randint

def nonrec(n, depth):
    ''' Prints the value of the parameter n and local variable
        rand before and after the recursive call.
        n is the parameter of interest.
        depth represents the depth of the recursion. '''

```

```

history = []
while n != 0:
    rand = randint(0, 1000) # Make a random number
    history += [(n, depth, rand)] # Remember original values of n and depth
    print(' ' * depth, 'Entering: n =', n, ' rand =', rand)
    n -= 1
    depth += 1
    print(' ' * depth, ' *** Recursion over ***')
while len(history) > 0:
    n, depth, rand = history[-1]
    del history[-1]
    print(' ' * depth, 'Exiting: n =', n, ' rand =', rand)

nonrec(10, 0)

```

The output of Listing 11.10 (`nonrecursivememory.py`) is identical to the output of Listing 11.9 (`recursivememory.py`). Listing 11.10 (`nonrecursivememory.py`) uses two separate, sequential loops and an accessory list to simulate the recursive behavior of Listing 11.9 (`recursivememory.py`). The purpose of the local history list is to remember the current state of the variables `n`, `depth`, and `rand` so the executing program can restore their values after the simulated recursion “returns.” The extra space for the history list and extra time spent managing the history list is comparable to the space and time the recursive function requires. It is much easier to allow the magic of recursion to automatically take care of saving and restoring the function’s local variables and parameters.

The ability of a function to remember its current state, call itself on a subproblem, and return to its previous state is essential to some algorithms. Section 11.6 explores one such algorithm.

11.6 List Permutations

Sometimes it is useful to consider all the possible arrangements of the elements within a list. A sorting algorithm, for example, must work correctly on any initial arrangement of elements in a list. To test a sort function, a programmer could check to see if it produces the correct result for all arrangements of a relatively small list. We saw in Section 5.4 that an arrangement of a sequence of ordered items is called a *permutation*. Listing 5.18 (`permuteabc.py`) *prints* all the permutations of the sequence *ABC*. We need something more flexible: a function that *generates* all the possible permutations of any list. The function will accept a list as a parameter and return a list containing all the permutations of the parameter. (Note that the return value is a list of lists.) Listing 11.11 (`listpermutations.py`) contains functions that build a new list containing all the permutations of a given list.

Listing 11.11: `listpermutations.py`

```

def perm(lst, begin, result):
    ''' Creates a list (result) containing all the permutations of the
        elements of a given list (lst), beginning with a
        specified index (begin).
        This is a helper function for the permutations function. '''
    end = len(lst) - 1 # Index of the last element
    if begin == end:
        result += [lst[:]] # Copy lst into result
    else:
        for i in range(begin, end + 1): # Consider all indices

```

```

        # Interchange the element at the first position
        # with the element at position i
        lst[begin], lst[i] = lst[i], lst[begin]
        # Recursively permute the rest of the list
        perm(lst, begin + 1, result)
        # Undo the earlier interchange
        lst[begin], lst[i] = lst[i], lst[begin]

def permutations(lst):
    ''' Returns a list containing all the permutations of the
        orderings of the elements of a given list (lst).
        Delegates the hard work to the perm function. '''
    result = []
    perm(lst, 0, result)
    return result

def main():
    ''' Tests the permutations function. '''
    a = list(range(3)) # Make list [0, 1, 2]
    print('List:', a) # Print the list
    # Generate and print all permutations of the list
    print('Permutations:', permutations(a))

if __name__ == '__main__':
    main()

```

Listing 11.11 (listpermutations.py) prints

```

List: [0, 1, 2]
Permutations: [[0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 1, 0], [2, 0, 1]]

```

Examining program's output closely, we see it is a list that contains all the permutations of the list `[0, 1, 2]`.

The `perm` function in Listing 11.11 (`listpermutations.py`) is a recursive function, as it calls itself inside of its definition. We have seen how recursion can be an alternative to iteration; however, the `perm` function here uses *both* iteration *and* recursion together to generate all the arrangements of a list. At first glance, the combination of these two algorithm design techniques as used here may be difficult to follow, but we actually can understand the process better if we *ignore* some of the details of the code.

First, notice that in the recursive call the argument `begin` is one larger. This means as the recursion progresses the beginning index keeps increasing until it reaches the index of the last element in the list. The recursion terminates when `begin` becomes equal to the last index.

In its simplest form the function looks like this:

```

def perm(lst, begin, result):
    end = len(lst) - 1 # Index of the last element
    if begin == end:
        # Add the current list to the list of permutations
    else:
        # Do the interesting part of the algorithm

```

Let us zoom in on the interesting part of the algorithm (less the comments):

```

for i in range(begin, end + 1):

```

```

    lst[begin], lst[i] = lst[i], lst[begin]
    perm(lst, begin + 1, result)
    lst[begin], lst[i] = lst[i], lst[begin]

```

If the mixture of iteration and recursion is confusing, eliminate iteration! If a loop iterates a fixed number of times, you may replace the loop with the statements in its body duplicated that number times; for example, we can rewrite the code

```

for i in range(5):
    print(i)

```

as

```

print(0)
print(1)
print(2)
print(3)
print(4)

```

Notice that the loop is gone. This process of transforming a loop into the series of statements that the loop would perform is known as *loop unrolling*. Compilers and interpreters can unroll loops behind the scenes to make the code's execution faster. After unrolling the loop, the loop control variable (in this case *i*) is gone, so there is no need to initialize *i* (done once) and, more importantly, no need to check and update *i* during each iteration of the loop.

Our purpose for unrolling the loop in `perm` is not to optimize it. Instead we are trying to understand better how the algorithm works. In order to unroll `perm`'s loop, we will consider the case for lists containing exactly three elements. In this case we would hardcode the `for` statement in the `perm` function as

```

for i in range(begin, 3):
    lst[begin], lst[i] = lst[i], lst[begin]
    perm(lst, begin + 1, result)
    lst[begin], lst[i] = lst[i], lst[begin]

```

and we can unroll this code into

```

lst[begin], lst[0] = lst[0], lst[begin]    # Swap
perm(lst, begin + 1, result)
lst[begin], lst[0] = lst[0], lst[begin]    # Swap back

lst[begin], lst[1] = lst[1], lst[begin]    # Swap
perm(lst, begin + 1, result)
lst[begin], lst[1] = lst[1], lst[begin]    # Swap back

lst[begin], lst[2] = lst[2], lst[begin]    # Swap
perm(lst, begin + 1, result)
lst[begin], lst[2] = lst[2], lst[begin]    # Swap back

```

Once the loop is gone, we see we have simply a series of recursive calls of `perm` sandwiched by element swaps. The first swap interchanges an element in the list with the first element. The second swap reverses the effects of the first swap. This series of swap-permute-swap operations allows each element in the list to have its turn being the first element in the permuted list. The `perm` recursive call generates all the permutations of the rest of the list. Figure 11.4 traces the recursive process of generating all the permutations

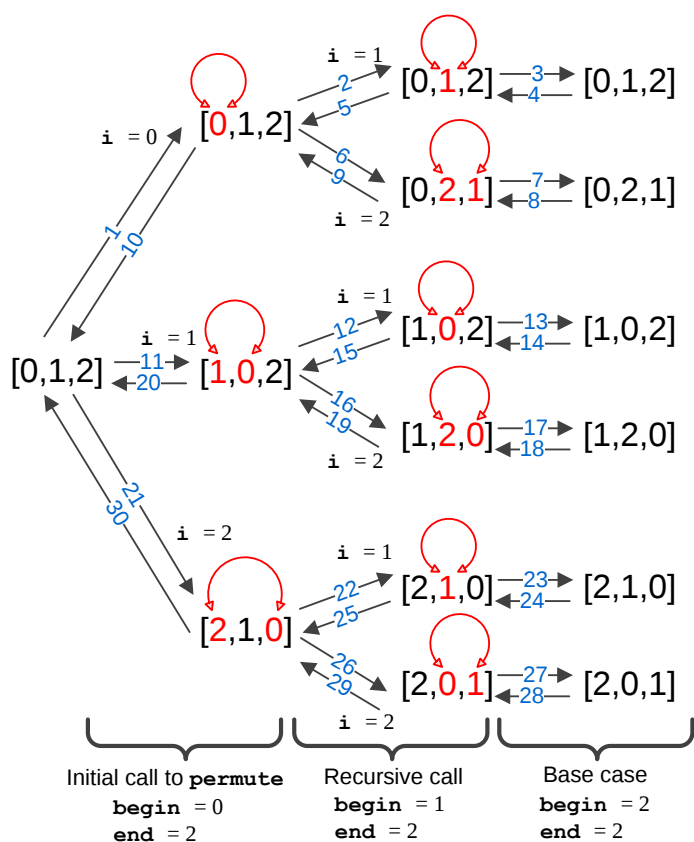


Figure 11.4: A tree mapping out the recursive process of the `perm` function operating on the list `[0, 1, 2]`. The second column from the left shows the original contents of the list after the first swap but before the first recursive call to `perm`. The swapped elements appear in red. The third column shows the contents of the list at the second level of recursion. In the third column the elements at index zero are fixed, as this recursion level is using `begin` with a value of one instead of zero. The `for` loop within this recursive call swaps the elements highlighted in red. The rightmost column is the point where `begin` equals the index of the last element, and so the `perm` function does not call itself, effectively terminating the recursion.

of the list `[0, 1, 2]`. The leftmost third of Figure 11.4 shows the original contents of the list and the initial call of `perm`. The three branches represent the three iterations of the `for` loop: `i` varying from `begin` (0) to the last index (2). The lists indicate the state of the list after the first swap but before the recursive call to `perm`.

The middle third of Figure 11.4 shows the state of the list during the first recursive call to `perm`. The two branches represent the two iterations of the `for` loop: `i` varying from `begin` (1) to the last index (2). The lists indicate the state of the list after the first swap but before the next recursive call to `perm`. At this level of recursion the element at index zero is fixed, and the remainder of the processing during this chain of recursion is restricted to indices greater than zero.

The rightmost third of Figure 11.4 shows the state of the list during the second recursive call to `perm`. At this level of recursion the elements at indices zero and one are fixed, and the remainder of the processing during this chain of recursion is restricted to indices greater than one. This leaves the element at index two, but this represents the base case of the recursion because `begin (2)` equals the index of the last element (2). In this case the function makes no more recursive calls to itself. The function merely adds a copy of the current list to the list of permutations.

The arrows in Figure 11.4 represent a call to, or a return from, `perm`. They illustrate the recursive call chain. The arrows pointing left to right represent a call, and the arrows pointing from right to left represent a return from the function. The numbers associated with arrow indicate the order in which the calls and returns occur during the execution of `perm`.

We can augment the `perm` function to better illustrate the iterative and recursive processes. With a technique known as *code instrumentation*, we will add statements that provide insight into the algorithm's progression. The term *instrumentation* mirrors its meaning outside the realm of programming. A motor vehicle, for example, has an instrument panel containing several different instruments. The speedometer indicates the vehicle's current speed, and the tachometer provides the vehicle's engine's RPMs. Neither of these devices is absolutely essential for driving the vehicle, but they do give the driver more precise information about the state of the driving experience.

Listing 11.12 (`perminstrumented.py`) instruments the `perm` function by adding print statements that indicate state of its list as it loops and calls itself recursively.

Listing 11.12: `perminstrumented.py`

```
def perm(lst, begin, result, depth):
    ''' Creates a list (result) containing all the permutations of the
        elements of a given list (lst), beginning with a
        specified index (begin).
        Printing statements report the progression of the
        function's recursion. The depth parameter indicates the
        depth of the recursion.
        This is a helper function for the permutations function. '''
    print(' ' * depth, 'begin =', begin)
    end = len(lst) - 1 # Index of the last element
    if begin == end:
        result += [lst[:]] # Copy lst into result
        print(' ' * depth, ' *')
    else:
        for i in range(begin, end + 1):
            # Interchange the element at the first position
            # with the element at position i
            lst[begin], lst[i] = lst[i], lst[begin]
            print(' ' * depth, ' ', lst[begin], '<-->', lst[i], ' ', lst)
            # Recursively permute the rest of the list
            perm(lst, begin + 1, result, depth + 1)
            # Undo the earlier interchange
            lst[begin], lst[i] = lst[i], lst[begin]

def permutations(lst):
    ''' Returns a list containing all the permutations of the
        orderings of the elements of a given list (lst).
        Delegates the hard work to the perm function. '''
    result = []
    perm(lst, 0, result, 0) # Initial call with depth = 0
```

```

    return result

def main():
    a = list(range(3))
    print(permutations(a))

if __name__ == '__main__':
    main()

```

Figure 11.5 shows the output of Listing 11.12 (perminstrumented.py).

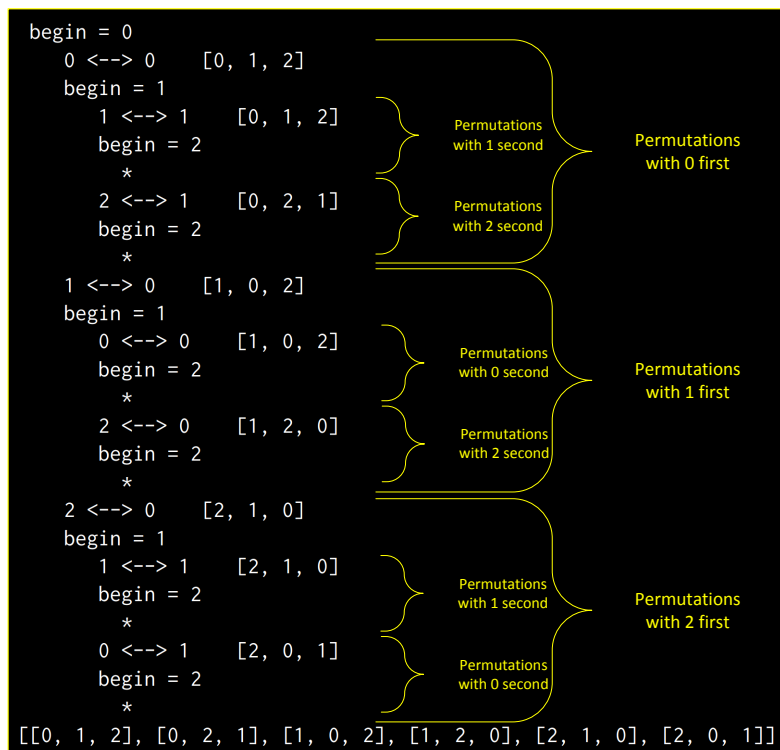


Figure 11.5: Output of Listing 11.12 (perminstrumented.py). The nested, inner sections show the recursive executions that place the element at index one. The outer sections represent the initial recursive calls the establish the element at index zero. The asterisk (*) indicates the end of the recursion. Note that the recursion ends exactly when begin is 2. Since 2 is the last index, there is no need to continue.

Now that we have a function that produces a list containing all the permutations of a given list in a regular fashion, we must admit that the function is practical only for relatively small lists. Consider a list that contains just 25 elements. How many distinct permutations are there of this list? The factorial function counts the number of ways of arranging the elements in a sequence:

$$25! = 15,511,210,043,330,985,984,000,000$$

Thus, our lowly list containing just 25 elements has 15,511,210,043,330,985,984,000,000 distinct permutations. Computers are fast and easily deal with large numbers, so this should not be a problem, should it? If each element of the list occupied just one byte of storage (the actual size is more than one byte), one permutation of the list containing 25 elements would require 25 bytes or memory. The list containing all the permutations would require

$$\begin{aligned} 25 \text{ bytes} \cdot 25! &= 387,780,251,083,274,649,600,000,000 \text{ bytes} \\ &\approx 387,780 \text{ zettabytes} \end{aligned}$$

(1 zettabyte = 1 billion terabytes.) 387,780 zettabytes is about 140,000 times greater than 2.7 zettabytes, the estimated total data storage space in all media (hard drives, solid state drives, CDs, DVDs, tape, etc.) found on the planet (see <http://en.wikipedia.org/wiki/Zettabyte>). It is safe to assume that your laptop or desktop would not have enough RAM to hold the list of all permutations. Besides, if your program could generate one permutation each nanosecond (an unreasonably fast rate even with today's fastest processors), the program would require

$$\begin{aligned} 25! \text{ nanoseconds} &= 15,511,210,043,330,985,984,000,000 \text{ nanoseconds} \\ &\approx 15,511,210,043,330,986 \text{ seconds} \\ &\approx 4,308,669,456,481 \text{ hours} \\ &\approx 179,527,894,020 \text{ days} \\ &\approx 491,857,244 \text{ years} \end{aligned}$$

Most users would be unwilling to wait almost *five million centuries* for the list of permutations that, by the way, is too large to store on any computer system on the planet!

Listing 11.11 (`listpermutations.py`) is impractical for all but relatively small lists because the `perm` function does not return until after building the list containing all the permutations. The basic algorithm is sound, however, and fortunately we can salvage it nicely using generators. (We first explored generators in Section 8.8.) Instead of producing the entire list of permutations, our function will yield each permutation one at a time.

We know that a function that produces a generator must use a `yield` statement rather than `return`. What we have yet to see is how this works with recursion.

The `perm` function in Listing 11.11 (`listpermutations.py`) adds a new list permutation to its result list with the following statement in its base case:

```
result += [lst[:]] # Copy lst into result
```

The expression `lst[:]` makes a copy of the `lst`. We want to `yield` a copy of the list instead of adding a copy of it to another list. This is an easy change, as we rewrite the statement to be

```
yield lst[:] # Yield a copy of lst
```

We covered the base case perfectly, what happens in the recursive case? Recursive generators are a little different from the iterative generators we saw earlier. Since the `if` block contains a `yield` statement, the `else` block needs one as well. What we want to yield is what the recursive call to `perm` eventually yields when it reaches its base case. When we need to yield a value from a recursive call we must use the `yield from` statement. The `yield from` statement indicates the generator should yield the result that the chain of recursive calls ultimately yields when it reaches its terminal base case. Listing 11.13 (`generatepermutations.py`) shows how to use `yield` and `yield from` in a recursive generator.

Listing 11.13: generatepermutations.py

```

def perm(lst, begin):
    ''' Generates the sequence of all the permutations of the
        elements of a given list (lst), beginning with a
        specified index (begin).
        This is a helper function for the permutations function. '''
    end = len(lst) - 1 # Index of the last element
    if begin == end:
        yield lst[:] # Yield a copy of lst
    else:
        for i in range(begin, end + 1): # Consider all indices
            # Interchange the element at the first position
            # with the element at position i
            lst[begin], lst[i] = lst[i], lst[begin]
            # Recursively permute the rest of the list
            yield from perm(lst, begin + 1)
            # Undo the earlier interchange
            lst[begin], lst[i] = lst[i], lst[begin]

def permutations(lst):
    ''' Generates the sequence of all the permutations of the
        elements of list lst.
        Delegates the hard work to the perm function. '''
    yield from perm(lst, 0)

def main():
    ''' Tests the permutations function. '''
    a = list(range(3)) # Make list [0, 1, 2]
    print('List:', a) # Print the list
    # Generate and print all permutations of the list
    for p in permutations(a):
        print(p, end=' ')
    print()

if __name__ == '__main__':
    main()

```

Listing 11.13 (generatepermutations.py) prints

```

List: [0, 1, 2]
[0, 1, 2] [0, 2, 1] [1, 0, 2] [1, 2, 0] [2, 1, 0] [2, 0, 1]

```

The permutations function must yield the result of the call to perm, so the `yield from` statement appears there as well. This is because the permutations function itself does not create the value; instead, permutations relies on the perm function to create the value. A function that relies on another function to create the yielded value must use `yield from`. Note that this is consistent with the way `yield from` works with recursion.

Note that since we are no longer building a physical list, we do not need the extra `result` parameter in the perm function.

The perm function in our generator code creates and yields just one permutation at a time. If the caller does not store each generated list but merely prints it out or uses it in some other way and discards it before obtaining the next list, the program will not run into the memory limitations of the original version.

This does not help the time it takes to produce all the permutations; however, the generator `perm` function “returns” a permutation immediately, thus avoiding the problems with the original version that tried to make all the permutations before returning. This means the caller can get into and out of the function quickly. While the program still would centuries to complete if asked to print all the permutations of a list with 25 elements, it could print the first 100 permutations very quickly:

```
lst = list(range(25))
count = 0
for p in permutations(lst):    # Too many to see them all!
    print(p, end=' ')
    count += 1
    if count == 100:
        break                 # Just print the first 100 permutations
print()
```

Note that we can always build a list of permutations with our generator version of the permutations function by using the list conversion function: The statement

```
lst = list(permutations([0, 1, 2]))
```

builds such a list. Be aware, however, that just as in the non-generator version, memory and time constraints limit the size of the list used in a statement like this one.

While Listing 11.13 (`generatepermutations.py`) is a good exercise in recursive list processing, the Python standard library provides a generator-like object named `permutations` in the `itertools` module that works almost like our `permutations` function. Surprisingly, the standard `permutations` generator produces tuples instead of lists, as Listing 11.14 (`stdpermutations.py`) demonstrates.

Listing 11.14: `stdpermutations.py`

```
# Use the standard permutations function to list
# the possible arrangements of elements in a list.

from itertools import permutations

def main():
    a = [0, 1, 2]
    for p in permutations(a):
        print(p, end=' ')
    print()

main()
```

Listing 11.14 (`stdpermutations.py`) produces

```
(0, 1, 2) (0, 2, 1) (1, 0, 2) (1, 2, 0) (2, 0, 1) (2, 1, 0)
```

In order to more match the behavior of Listing 11.13 (`generatepermutations.py`) we must convert each tuple to a list. Listing 11.15 (`stdpermutations2list.py`) uses the `list` function to perform the conversion.

Listing 11.15: `stdpermutations2list.py`

```
# Use the standard permutations function to list
# the possible arrangements of elements in a list.
```

```

from itertools import permutations

def main():
    a = [0, 1, 2]
    for p in permutations(a):
        print(list(p), end=' ')
    print()

main()

```

Listing 11.15 (stdpermutations2list.py) produces

```
[0, 1, 2] [0, 2, 1] [1, 0, 2] [1, 2, 0] [2, 0, 1] [2, 1, 0]
```

It is evident that the standard permutations object uses a slightly different algorithm because it orders the results slightly differently from Listing 11.13 (generatepermutations.py).

11.7 Randomly Permuting a List

We have seen that generating all the permutations of a large list is computationally intractable. Often, however, we merely need to produce one permutation chosen at random. For example, we may need to randomly rearrange the contents of an ordered list so that we can test a sort function to see if it will produce the original list. We could generate all the permutations, put each one in a list, and select a permutation at random from that list. This approach is inefficient, especially as the length of the list to permute grows larger. Fortunately, we can randomly permute the contents of a list easily and quickly. Listing 11.16 (randompermute.py) contains a function named `permute` that randomly permutes the elements of a list.

Listing 11.16: randompermute.py

```

from random import randrange

def permute(lst):
    """
    Randomly permutes the contents of list lst
    """
    n = len(lst)
    for i in range(n - 1):
        pos = randrange(i, n) # i <= pos < n
        lst[i], lst[pos] = lst[pos], lst[i]

def main():
    """
    Tests the permute function that randomly permutes the
    contents of a list
    """
    a = [1, 2, 3, 4, 5, 6, 7, 8]
    print('Before:', a)
    permute(a)
    print('After :', a)

main()

```

Notice that the `permute` function in Listing 11.16 (`randompermute.py`) uses a simple un-nested loop and no recursion. The `permute` function varies the `i` index variable from 0 to the index of the next to last element in the list. An index greater than `i` is chosen pseudorandomly using `randrange` (see Section 6.4), and the elements at position `i` and the random position are exchanged. At this point all the elements at position `i` and smaller are fixed and will not change as the function's execution continues. The index `i` is incremented, and the process continues until all the `i` values have been considered.

To be correct, our `permute` function must be able to generate any valid permutation of the list. It is important that our `permute` function is able produce all possible permutations with equal probability; said another way, we do not want our `permute` function to generate some permutations more often than others. The `permute` function in Listing 11.16 (`randompermute.py`) is fine, but consider a slight variation of the algorithm:

```
def faulty_permute(lst):
    """
    An attempt to randomly permute the contents of list lst
    """
    n = len(lst)
    for i in range(n - 1):
        pos = randrange(0, n)    # 0 <= pos < n
        lst[i], lst[pos] = lst[pos], lst[i]
```

Do you see the difference between `faulty_permute` and `permute`? In `faulty_permute`, the random index is chosen from all valid list indices, whereas `permute` restricts the random index to valid indices greater than or equal to `i`. This means that any element within `lst` can be exchanged with the element at position `i` during any loop iteration. While this approach may superficially appear to be just as good as `permute`, it in fact produces an uneven distribution of permutations. Listing 11.17 (`comparepermutations.py`) exercises each permutation function 1,000,000 times on the list `[1, 2, 3]` and tallies each permutation. There are exactly six possible permutations of this three-element list.

Listing 11.17: `comparepermutations.py`

```
from random import randrange

# Randomly permute a list
def permute(lst):
    """
    Randomly permutes the contents of list lst
    """
    n = len(lst)
    for i in range(n - 1):
        pos = randrange(i, n)    # i <= pos < n
        lst[i], lst[pos] = lst[pos], lst[i]

# Randomly permute a list?
def faulty_permute(lst):
    """
    An attempt to randomly permute the contents of list lst
    """
    n = len(lst)
    for i in range(n):
        pos = randrange(0, n)    # 0 <= pos < n
        lst[i], lst[pos] = lst[pos], lst[i]
```

```

def classify(a):
    """
    Classify a list as one of the six permutations
    """
    sum = 100*a[0] + 10*a[1] + a[2]
    if sum == 123: return 0
    elif sum == 132: return 1
    elif sum == 213: return 2
    elif sum == 231: return 3
    elif sum == 312: return 4
    elif sum == 321: return 5
    else: return -1

def report(a):
    """
    Report the accumulated statistics
    """
    print("1,2,3: ", a[0])
    print("1,3,2: ", a[1])
    print("2,1,3: ", a[2])
    print("2,3,1: ", a[3])
    print("3,1,2: ", a[4])
    print("3,2,1: ", a[5])

def run_test(perm, runs):
    """
    Uses a permutation function to generate the permutations
    of the list [1,2,3]
    perm: the permutation function to test
    runs: the number permutations to perform
    """
    # The list to permute
    original = [1, 2, 3]

    # permutation_tally list keeps track of each permutation pattern
    # permutation_tally[0] counts {1,2,3}
    # permutation_tally[1] counts {1,3,2}
    # permutation_tally[2] counts {2,1,3}
    # permutation_tally[3] counts {2,3,1}
    # permutation_tally[4] counts {3,1,2}
    # permutation_tally[5] counts {3,2,1}
    permutation_tally = 6 * [0] # Clear all the counters
    for i in range(runs): # Run runs times
        # working holds a copy of original is gets permuted and tallied
        working = original[:]
        # Permute the list with the permutation algorithm
        perm(working)
        # Count this permutation
        permutation_tally[classify(working)] += 1
    report(permutation_tally) # Report results

def main():

```

```

# Each test performs one million permutations
runs = 1000000

print("--- Random permute #1 -----")
run_test(permute, runs)

print("--- Random permute #2 -----")
run_test(faulty_permute, runs)

main()

```

In Listing 11.17 (`comparepermutations.py`)’s output, permute #1 corresponds to our original permute function, and permute #2 is the `faulty_permute` function. The output of Listing 11.17 (`comparepermutations.py`) reveals that the faulty permutation function favors some permutations over others:

```

--- Random permute #1 -----
1,2,3: 166176
1,3,2: 166957
2,1,3: 167012
2,3,1: 166668
3,1,2: 166182
3,2,1: 167005
--- Random permute #2 -----
1,2,3: 148811
1,3,2: 184870
2,1,3: 185251
2,3,1: 184763
3,1,2: 148200
3,2,1: 148105

```

In one million runs, the permute function provides an even distribution of the six possible permutations of `[1, 2, 3]`. The `faulty_permute` function generates the permutations `[1, 3, 2]`, `[2, 1, 3]`, and `[2, 3, 1]` more often than the permutations `[1, 2, 3]`, `[3, 1, 2]`, and `[3, 2, 1]`.

To see why `faulty_permute` misbehaves, we need to examine all the permutations it can produce during one call. Figure 11.6 shows a hierarchical structure that maps out how `faulty_permute` transforms its list parameter each time through the `for` loop. The top of the tree shows the original list, `[1, 2, 3]`. The

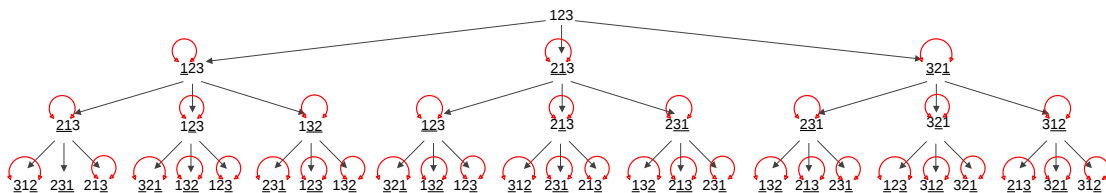


Figure 11.6: A tree mapping out the ways in which `faulty_permute` can transform the list `[1, 2, 3]` at each iteration of its `for` loop

second row shows the three possible resulting lists after the first iteration of the `for` loop. The leftmost list represents the element at index zero swapped with the element at index zero (effectively no change). The

second list on the second row represents the interchange of the elements at index 0 and index 1. The third list on the second row results from the interchange of the elements at positions 0 and 2. The underlined elements represent the elements most recently swapped. If only one item in the list is underlined, the function merely swapped the item with itself. The bottom row contains all the possible outcomes of the `faulty_permute` function given the list `[1, 2, 3]`.

As Figure 11.6 shows, the lists `[1, 3, 2]`, `[2, 1, 3]`, and `[2, 3, 1]` each appear five times in the last row, while `[1, 2, 3]`, `[3, 1, 2]`, and `[3, 2, 1]` each appear only four times. There are a total of 27 possible outcomes, so some permutations appear $\frac{4}{27} = 14.815\%$ of the time, while the others appear $\frac{5}{27} = 18.519\%$ of the time. Notice that these percentages agree with our experimental results from Listing 11.17 (`comparepermutations.py`).

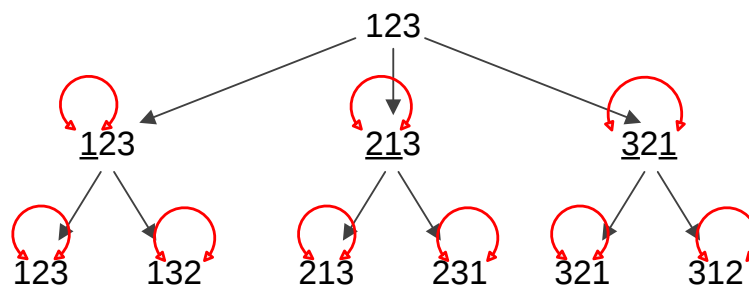


Figure 11.7: A tree mapping out the ways in which `permute` can transform the list `[1, 2, 3]` at each iteration of its for loop

Compare Figure 11.6 to Figure 11.7. The second row of the tree for `permute` is identical to the second row of the tree for `faulty_permute`, but the third rows are different. The second time through its loop the `permute` function does not attempt to exchange the element at index zero with any other elements. We see that none of the first elements in the lists in row three are underlined. The third row contains exactly one instance of each of the possible permutations of `[1, 2, 3]`. This means that the correct `permute` function is not biased towards any of the individual permutations, and so the function can generate all the permutations with equal probability. The `permute` function has a $\frac{1}{6} = 16.667\%$ probability of generating a particular permutation; this number agrees with our the experimental results of Listing 11.17 (`comparepermutations.py`).

11.8 Reversing a List

Listing 11.18 (`listreverse.py`) contains a recursive function named `rev` that accepts a list as a parameter and returns a new list with all the elements of the original list in reverse order.

Listing 11.18: `listreverse.py`

```
def rev(lst):
    return [] if len(lst) == 0 else rev(lst[1:]) + lst[0:1]
```

```
print(rev([1, 2, 3, 4, 5, 6, 7]))
```

Python has a standard function, `reversed`, that accepts a list parameter. The `reversed` function does not return a list but instead returns an iterable object that can be used like a generator or range within a `for` loop (see Section 5.3). Listing 11.19 (`reversed.py`) shows how `reversed` can be used to print the contents of a list backwards.

Listing 11.19: `reversed.py`

```
for item in reversed([1, 2, 3, 4, 5, 6, 7]):  
    print(item)
```

We can use the list conversion function to make a new list object out of the iterator object that `reversed` creates:

```
rev = list(reversed([1, 2, 3, 4, 5, 6, 7]))
```

In Section 10.10 we will see how to reverse in place the elements in a list using a special function-like object called a *method*.

11.9 Summary

- Various algorithms exist for sorting lists. Selection sort is a simple algorithm for sorting a list.
- A list formal parameter aliases the actual parameter passed by the caller. This means any modifications a function makes to the contents of the list will affect the caller's own list. This concept allows a sort or permutation routine to physically rearrange the elements in a list for the caller's benefit.
- Linear search is useful for finding elements in an unordered list. Binary search can be used on ordered lists, and due to the nature of its algorithm, binary search is very fast, even on large lists.
- A permutation of a list is a reordering of its elements.
- Care must be taken when producing a random permutation of a list to ensure all the possible outcomes are equally likely.

11.10 Exercises

1. Complete the following function that reorders the contents of a list so they are reversed from their original order. For example, a list containing the elements 2, 6, 2, 5, 0, 1, 2, 3 would be transformed into 3, 2, 1, 0, 5, 2, 6, 2. Note that your function must physically rearrange the elements within the list, not just print the elements in reverse order.

```
def reverse(lst):  
    # Add your code...
```

2. Complete the following function that reorders the contents of a list of integers so that all the even numbers appear before any odd number. The even values are sorted in ascending order with respect to themselves, and the odd numbers that follow are also sorted in ascending order with respect to

themselves. For example, a list containing the elements 2, 1, 10, 4, 3, 6, 7, 9, 8, 5 would be transformed into 2, 4, 6, 8, 10, 1, 3, 5, 7, 9. Note that your function must physically rearrange the elements within the list, not just print the elements in the desired order.

```
def special_sort(lst):  
    # Add your code...
```

3. Create a special comparison function to be passed to our flexible selection sort function. The special comparison function should enable the sort function to arrange the elements of a list in the order specified in Exercise 2.
4. Complete the following function that filters negative elements out of a list. The function returns the filtered list and the original list is unchanged. For example, if a list containing the elements 2, -16, 2, -5, 0, 1, -2, -3 is passed to the function, the function would return the list containing 2, 2, 0, 1. Note the original ordering of the non-negative values is unchanged in the result.

```
def filter(a):  
    # Add your code...
```

5. Complete the following function that shifts all the elements of a list backward one place. The last element that gets shifted off the back end of the list is copied into the first (0th) position. For example, if a list containing the elements 2, 1, 10, 4, 3, 6, 7, 9, 8, 5 is passed to the function, it would be transformed into 5, 2, 1, 10, 4, 3, 6, 7, 9, 8. Note that your function must physically rearrange the elements within the list, not just print the elements in the shifted order.

```
def rotate(lst):  
    # Add your code...
```

6. Complete the following function that determines if the number of even and odd values in an integer list is the same. The function would return true if the list contains 5, 1, 0, 2 (two evens and two odds), but it would return false for the list containing 5, 1, 0, 2, 11 (too many odds). The function should return true if the list is empty, since an empty list contains the same number of evens and odds (0 for both). The function does not affect the contents of the list.

```
def balanced(a):  
    # Add your code...
```

7. Complete the following function that returns true if a list `lst` contains duplicate elements; it returns false if all the elements in `lst` are unique. For example, the list `[2, 3, 2, 1, 9]` contains duplicates (2 appears more than once), but the list `[2, 1, 0, 3, 8, 4]` does not (none of the elements appear more than once).

An empty list has no duplicates. The function does not affect the contents of the list.

```
def has_duplicates(lst):  
    # Add your code...
```

8. Can linear search be used on an unsorted list? Why or why not?
9. Can binary search be used on an unsorted list? Why or why not?
10. How many different orderings are there for the list `[4, 3, 8, 1, 10]`?

11. Complete the following function that determines if two lists contain the same elements, but not necessarily in the same order. The function would return true if the first list contains 5, 1, 0, 2 and the second list contains 0, 5, 2, 1. The function would return false if one list contains elements the other does not or if the number of elements differ. This function could be used to determine if one list is a permutation of another list. The function does not affect the contents of either list.

```
def is_permutation(a, b):  
    # Add your code...
```

12. Listing 11.11 (listpermutations.py) contains the following statement:

```
result += [lst[:]]
```

- (a) What happens if you change the statement to the following?

```
result += [lst]
```

- (b) Explain why this modified statement produces the result that it does.

Chapter 12

Tuples, Dictionaries, and Sets

Lists, introduced in Chapter 10, are convenient data structures for representing sequences of data. In this chapter we examine several other ways that Python provides for storing aggregate data: tuples, dictionaries, and sets.

12.1 Tuples

Tuples are similar to lists, except tuples are immutable. Listing 12.1 (tupletest.py) compares the usage of lists versus tuples.

Listing 12.1: tupletest.py

```
my_list = [1, 2, 3, 4, 5, 6, 7]      # Make a list
my_tuple = (1, 2, 3, 4, 5, 6, 7)    # Make a tuple
print('The list:', my_list)         # Print the list
print('The tuple:', my_tuple)       # Print the tuple
print('The first element in the list:', my_list[0]) # Access an element
print('The first element in the tuple:', my_tuple[0]) # Access an element
print('All the elements in the list:', end=' ')
for elem in my_list:                # Iterate over the elements of a list
    print(elem, end=' ')
print()
print('All the elements in the tuple:', end=' ')
for elem in my_tuple:               # Iterate over the elements of a tuple
    print(elem, end=' ')
print()
print('List slice:', my_list[2:5])   # Slice a list
print('Tuple slice:', my_tuple[2:5]) # Slice a tuple
print('Try to modify the first element in the list . . .')
my_list[0] = 9                      # Modify the list
print('The list:', my_list)
print('Try to modify the first element in the list . . .')
my_tuple[0] = 9                     # Is tuple modification possible?
print('The tuple:', my_tuple)
```

Feature	List	Tuple
Mutability	mutable	immutable
Creation	<code>lst = [i, j]</code>	<code>tpl = (i, j)</code>
Element access	<code>a = lst[i]</code>	<code>a = tpl[i]</code>
Element modification	<code>lst[i] = a</code>	<i>Not possible</i>
Element addition	<code>lst += [a]</code>	<i>Not possible</i>
Element removal	<code>del lst[i]</code>	<i>Not possible</i>
Slicing	<code>lst[i:j:k]</code>	<code>tpl[i:j:k]</code>
Slice assignment	<code>lst[i:j] = []</code>	<i>Not possible</i>
Iteration	<code>for elem in lst:...</code>	<code>for elem in tpl:...</code>

Table 12.1: Python lists versus tuples

Listing 12.1 (tupletest.py) prints

```

The list: [1, 2, 3, 4, 5, 6, 7]
The tuple: (1, 2, 3, 4, 5, 6, 7)
The first element in the list: 1
The first element in the tuple: 1
All the elements in the list: 1 2 3 4 5 6 7
All the elements in the tuple: 1 2 3 4 5 6 7
List slice: [3, 4, 5]
Tuple slice: (3, 4, 5)
Try to modify the first element in the list . . .
The list: [9, 2, 3, 4, 5, 6, 7]
Try to modify the first element in the list . . .
Traceback (most recent call last):
  File "tupletest.py", line 26, in <module>
    main()
  File "tupletest.py", line 22, in main
    my_tuple[0] = 9
TypeError: 'tuple' object does not support item assignment

```

We see that Listing 12.1 (tupletest.py) does not run to completion. The next to the last statement in the program:

```
my_tuple[0] = 9
```

generates a run-time exception because tuples are immutable. Once we create tuple object, we cannot change that object's contents.

Table 12.1 compares lists to tuples.

Unlike with lists, we cannot modify an element within a tuple, we cannot add elements to a tuple, and we cannot remove elements from a tuple. If we have a variable assigned to a tuple, we always can reassign that variable to a different tuple. Such an assignment simply binds the variable to a different tuple object—it does not modify the tuple to which the variable originally was bound.

The parentheses are optional in the following statement:

```
my_tuple = (1, 2, 3)
```

The following statement is equivalent:

```
my_tuple = 1, 2, 3
```

Lists can hold heterogeneous data types, and so too can tuples:

```
>>> t = (2, 'Fred', 41.2, [30, 20, 10])
>>> t
(2, 'Fred', 41.2, [30, 20, 10])
```

In general practice, however, many Python programmers favor storing only homogeneous types in lists and prefer tuples for holding heterogeneous types.

We can convert a tuple to a list using the `list` function, and the `tuple` function performs the reverse conversion. The following interactive sequence demonstrates the use of the conversion functions:

```
>>> tpl = 1, 2, 3, 4, 5, 6, 7, 8
>>> tpl
(1, 2, 3, 4, 5, 6, 7, 8)
>>> list(tpl)
[1, 2, 3, 4, 5, 6, 7, 8]
>>> lst = ['a', 'b', 'c', 'd']
>>> lst
['a', 'b', 'c', 'd']
>>> tuple(lst)
('a', 'b', 'c', 'd')
```

Neither the `list` nor `tuple` function actually modifies its argument; that is, `tuple(lst)` does not modify `lst`, and `list(tpl)` does not modify `tpl` (since tuples are immutable, any modification would be impossible anyway). The `list` function makes a new list out of the contents of a tuple, and the `tuple` function makes a new tuple out of the elements in a list.

We can use the built-in `zip` function to generate a sequence of tuples from two lists. Consider the following interactive sequence:

```
>>> lst1 = [1, 2, 3, 4, 5, 6, 7, 8]
>>> lst2 = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> for t in zip(lst1, lst2):
...     print(t)
...
(1, 'a')
(2, 'b')
(3, 'c')
(4, 'd')
(5, 'e')
(6, 'f')
(7, 'g')
(8, 'h')
```

You can think of the `zip` function working like a physical zipper. A physical zipper pairs up two sets of interlocking (usually metal) teeth, closing an opening in a garment or bag. The Python `zip` function pairs up elements from two different sequences. The paired-up elements are tuples, and the sequences can be lists or sequences constructed from generators (see Section 8.8). If one of the sequences is shorter than the other, the `zip` function stops at the shorter sequence.

Listing 12.2 (`zipseq.py`) constructs a sequence of tuples with their first elements derived from a list and their second elements obtained from a generator. Note that the generator's sequence is shorter than the list.

Listing 12.2: zipseq.py

```
def gen(n):
    for i in range(n):
        yield i**2

for p in zip([10, 20, 30, 40, 50, 60], gen(4)):
    print(p, end=' ')
print()
```

Listing 12.2 (zipseq.py) prints

```
(10, 0) (20, 1) (30, 4) (40, 9)
```

The `zip` function does not return a list; like `range`, it returns an object over which we can iterate. We can make a list from a `zip` object using the `list` conversion function:

```
>>> list(zip(range(5), range(10, 0, -1)))
[(0, 10), (1, 9), (2, 8), (3, 7), (4, 6)]
```

We can use the `zip` function and list comprehension to build elaborate lists. Suppose we wish to make a new list from two existing lists. The first element in our new list will be the sum of the first elements from the two original lists. Similarly, the second element in our new list will be the sum of the second elements in the two original lists, and so forth. We can use `zip` to pair up the elements, as the following interactive sequence illustrates:

```
>>> for p in zip([1, 2, 3, 4, 5], [10, 11, 12, 13, 14]):
...     print(p)
...
(1, 10)
(2, 11)
(3, 12)
(4, 13)
(5, 14)
```

We want to add together the components of each tuple. To print each sum we could write

```
>>> for (x, y) in zip([1, 2, 3, 4, 5], [10, 11, 12, 13, 14]):
...     print(x + y)
...
11
13
15
17
19
```

We can reassemble these pieces into a list comprehension to build our list of sums:

```
>>> [x + y for (x, y) in zip([1, 2, 3, 4, 5], [10, 11, 12, 13, 14])]
[11, 13, 15, 17, 19]
```

When treated as a Boolean expression, the empty tuple `()` is interpreted as `False`, and any other tuple is considered `True`.

Since they are so similar, why does Python have both lists and tuples? Under some circumstances an executing program can perform optimizations on immutable objects that would be impossible with mutable objects. These optimizations can increase the program's performance. Also, it is easier in general to reason about the behavior of programs that use immutable objects. The fact that some objects cannot change makes it easier to understand how a section of code works, or, during debugging, why the section of code does not work.

12.2 Arbitrary Argument Lists

We can easily write a function that adds two numbers; consider the following `sum` function:

```
def sum(a, b):  
    return a + b
```

What if we need `sum` to be flexible enough to add two *or* three numbers? We can implement such a function with default arguments (Section 8.2). Listing 12.3 (`add2or3.py`) illustrates such a function

Listing 12.3: `add2or3.py`

```
def sum(a, b, c=0):  
    return a + b + c    # Adding zero will not affect a + b  
  
print(sum(3, 4))  
print(sum(3, 4, 5))
```

The `sum` function in Listing 12.3 (`add2or3.py`) handles two and three arguments equally well:

```
7  
12
```

A function that is capable of adding up to five numbers is equally easy:

```
def sum(a, b=0, c=0, d=0, e=0):  
    return a + b + c + d + e
```

Suppose we wish to write a `sum` function that can add as many numbers as the caller can supply? The default parameter approach is not practical in this situation. Our function must be able to handle 1,000 numbers or more, and these numbers must be separate arguments—not a single list containing 1,000 numbers.

When we define a function we specify the individual parameters it accepts, providing default values as needed. In the function definitions we have seen to this point the number of parameters is fixed. We need a way to define a function in such a way so that it can accept an arbitrary number of parameters. Fortunately Python has a mechanism for specifying that a function can accept an arbitrary number of parameters. Listing 12.4 (`addmany.py`) illustrates how to write such a function.

Listing 12.4: `addmany.py`

```
def sum(*nums):  
    s = 0                # Initialize sum to zero  
    for num in nums:     # Consider each argument passed to the function  
        s += num         # Accumulate their values  
    return s             # Return the sum
```

```
print(sum(3, 4))
print(sum(3, 4, 5))
print(sum(3, 3, 3, 3, 4, 1, 9, 44, -2, 8, 8))
```

The `sum` function in Listing 12.4 (`addmany.py`) handles as many actual parameters as the client can provide; the program prints

```
7
12
84
```

The single asterisk (*) before the formal parameter `nums` indicates the parameter is not necessarily a single value but potentially a collection of values. Listing 12.5 (`addmanyaugmented.py`) reveals what really is going on behind the scenes.

Listing 12.5: `addmanyaugmented.py`

```
def sum(*nums):
    print(nums)          # See what nums really is
    s = 0                # Initialize sum to zero
    for num in nums:     # Consider each argument passed to the function
        s += num         # Accumulate their values
    return s             # Return the sum

print(sum(3, 4))
print(sum(3, 4, 5))
print(sum(3, 3, 3, 3, 4, 1, 9, 44, -2, 8, 8))
```

The `sum` function in Listing 12.5 (`addmanyaugmented.py`) prints the following:

```
(3, 4)
7
(3, 4, 5)
12
(3, 3, 3, 3, 4, 1, 9, 44, -2, 8, 8)
84
```

Listing 12.5 (`addmanyaugmented.py`) exposes the fact that the formal parameter `nums` is a tuple wrapping all the actual parameters sent by the caller. Since `nums` is simply a tuple, we can iterate over it with the `for` statement to extract all the actual parameters provided by the caller.

A function definition may contain at most one of these arbitrary arguments parameters, and, if present, this parameter must appear after all the named, single formal parameters, if any. In the following `sum` function callers must provide at least two parameters but may pass more:

```
def sum(num1, num2, *extranums):
    s = num1 + num2
    for n in extranums:
        s += n
    return s
```

Note that the formal parameters `num1` and `num2` must appear before `*extranums` in `sum`'s formal parameter list.

As we have seen, a formal parameter declared with an asterisk is really a tuple from the function's perspective. The caller can pass individual arguments not packed within a tuple. The Python interpreter takes care of the packing the caller's arguments into a tuple during the call.

This process works in reverse also. Consider the following function that accepts four parameters:

```
def f(a, b, c, d):  
    print('a =', a, ' b = ', b, ' c = ', c, ' d = ', d)
```

We can pass a single argument to function `f` if we expression it in the proper way:

```
args = (10, 20, 30, 40)  
f(*args)
```

The variable `args` is a tuple, but `f` does not accept a single tuple; it accepts four parameters. Expressing the actual parameter as `*args` enables the interpreter to unpack the tuple into the four parameters the function expects. Note that the tuple must contain exactly the number of parameters that the function expects.

Given the definition of function `f` above, the following call is legal:

```
f(*(10, 20, 30, 40))    # Legal, unpacks the tuple into separate args
```

but the following is illegal:

```
f((10, 20, 30, 40))    # Illegal, function does not accept a tuple
```

12.3 Dictionaries

Lists and tuples are convenient for storing collections of data, but they have some limitations. For one, we locate an element within a list or tuple based on its position (index). While this approach is fine for many applications, in other situations this access-by-index approach is awkward or inefficient.

A Python *dictionary* is an *associative container* which permits access based on a *key*, rather than an index. Unlike an index, a key is not restricted to an integer expression. The following interactive sequence builds a simple dictionary that uses string keys:

```
>>> d = {} # Make an empty dictionary  
>>> d  
{}  
>>> # Add an element  
... d['Fred'] = 44  
>>> d  
{'Fred': 44}  
>>> # Add another element  
... d['Wilma'] = 31  
>>> d  
{'Fred': 44, 'Wilma': 31}  
>>> print(d)  
{'Fred': 44, 'Wilma': 31}  
>>> d['Fred']  
44  
>>> d['Wilma']  
31  
>>> d['Dino']
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Dino'
>>> d[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
```

Notice that, unlike a list which uses square brackets (`[]`), the contents of a dictionary appear within curly braces (`{}`). To access an element within a dictionary, however, we use square brackets exactly as we would with a list. In a dictionary every *key* has an associated *value*. The dictionary `d` from the interactive sequence above pairs the key `'Fred'` with the value 44. It also pairs the key `'Wilma'` with the value 31.

When associating a value with a key, as in

```
d['Fred'] = 44    # Associate value 44 with key 'Fred'
```

there is no restriction on the key within the square brackets; if the key within the square brackets does not exist in the dictionary, the statement adds the key and pairs it with the value on the right of the assignment operator. If the key already exists in the dictionary, the statement replaces the value previously associated with the key with the new value on the right of the assignment operator.

Accessing a value with a given key is a different story. In the statement

```
x = d['Fred']
```

`'Fred'` must be a valid key in dictionary `d`, or the program will raise an exception. A valid key is a key that is present in the dictionary. At the end of the interaction sequence above `'Fred'` is a valid key but `'Dino'` is not. We see the interpreter's reaction when we attempt to use an invalid key: the interpreter generates a `KeyError` exception.

We can check to see if a key is present in a dictionary with the `in` operator:

```
if 'Fred' in d:      # Check to see if 'Fred' is a valid key
    print(d['Fred']) # Print the value associated with key 'Fred'
else:
    print('\nFred\n is not a key in d') # Warn user of missing key
```

A dictionary key may be of any immutable type. This means all of the following can serve as keys within a dictionary: integers, floating-point numbers, strings, Booleans, and tuples. Since lists are mutable objects, a list may not be a key. A dictionary is a mutable object, so a dictionary cannot use itself or another dictionary object as a key. A value within a dictionary may any valid Python type, immutable or mutable.

The keys within a given dictionary may be of mixed types; consider the following interactive sequence:

```
>>> s = {}
>>> s[8] = 44
>>> s[8]
44
>>> s['Alpha'] = 'up'
>>> s['Alpha']
'up'
>>> s[True] = 'right'
>>> s[True]
'right'
>>> s[10 < 20]
```

```
'right'
>>> s['Beta'] = 100
>>> s
{8: 44, True: 'right', 'Beta': 100, 'Alpha': 'up'}
>>> s[3.4] = True
>>> s
{8: 44, True: 'right', 'Beta': 100, 3.4: True, 'Alpha': 'up'}
>>> s[2 == -2] = 'wrong'
>>> s[False]
'wrong'
>>> s
{False: 'wrong', True: 'right', 'Beta': 100, 'Alpha': 'up', 3.4: True,
8: 44}
>>> x = 8
>>> s[x]
44
>>> y = 15
>>> s[y] = 'down'
>>> lst = [1, 2, 3]
>>> s[17] = lst
>>> s
{False: 'wrong', True: 'right', 'Beta': 100, 'Alpha': 'up', 3.4: True,
17: [1, 2, 3], 8: 44, 15: 'down'}
```

This interactive sequence reveals several dictionary characteristics:

- The keys in a dictionary may have different types
- The values in a dictionary may have different types
- The values in a dictionary may be mutable objects
- The order of *key:value* pairs in a dictionary are independent of the order of their insertion into the dictionary

We can initialize a dictionary using the same syntax as the output that the `print` function displays. The following statement populates the dictionary `d` with four *key:value* entries:

```
d = {'Fred': 44, 'Wilma': 39, 'Barney': 40, 'Betty': 41}
print(d)
```

Despite the order supplied during `d`'s initialization, on one system the code above prints

```
{'Wilma': 39, 'Fred': 44, 'Barney': 40, 'Betty': 41}
```

Observe that the `print` function neither lists the keys in lexicographical order nor lists the values in numerical order. While an executing program must store a dictionary's contents in memory in some particular order, the exact internal ordering of the elements within a dictionary can vary from one program execution to the next. This example further demonstrates that programmers cannot depend on a specific ordering of the elements within a dictionary. Unlike in a list or other sequence type, the notions of order and position have no meaning within a dictionary.

The `keys` method of the dictionary class returns a sequence of all the keys in `d`. The following code demonstrates:

```
d = {'Fred': 44, 'Wilma': 39, 'Barney': 40, 'Betty': 41}
for k in d.keys():
    print(k, end=' ')
print()
```

Upon one execution of this code it prints

```
Barney Betty Fred Wilma
```

The exact order of the printed keys will vary, even between executions of the same program. The following code behaves similarly:

```
d = {'Fred': 44, 'Wilma': 39, 'Barney': 40, 'Betty': 41}
for k in d:
    print(k, end=' ')
print()
```

This code produces the same output as iterating over `d.keys()`:

```
Barney Betty Fred Wilma
```

with the above caveat that the exact order of the printed keys will vary.

The `values` method of the dictionary class returns a sequence of all the values in `d`. The following code illustrates:

```
d = {'Fred': 44, 'Wilma': 39, 'Barney': 40, 'Betty': 41}
for v in d.values():
    print(v, end=' ')
print()
```

This code prints

```
40 41 44 39
```

We can obtain a sequence of tuples of *key:value* pairs of a dictionary with the `items` method:

```
d = {'Fred': 44, 'Wilma': 39, 'Barney': 40, 'Betty': 41}
for k, v in d.items():
    print(k, v)
```

This code fragment prints

```
Barney 40
Betty 41
Fred 44
Wilma 39
```

Suppose we have the list `['Fred', 'Wilma', 'Barney', 'Betty']` and the list `[4174, 2287, 5003, 2012]`. We know we can zip them together into a list of tuples using `zip` (Section 12.1). We can use the `dict` function to create a dictionary of *key:value* pairs formed from the tuples, as the following interactive sequence shows:

```
>>> names = ['Fred', 'Wilma', 'Barney', 'Betty']
>>> numbers = [4174, 2287, 5003, 2012]
>>> names
['Fred', 'Wilma', 'Barney', 'Betty']
>>> numbers
[4174, 2287, 5003, 2012]
>>> d = dict(zip(names, numbers))
>>> d
{'Barney': 5003, 'Wilma': 2287, 'Betty': 2012, 'Fred': 4174}
```

The elements of the list specified as the first actual parameter to `zip` become the dictionary keys, and the elements of the list specified as the second argument to `zip` form the values in the dictionary. As we noted earlier, the ordering of the *key:value* pairs is different from their order in the original lists, but the first element from the `names` list is paired with the first element of the `numbers` list, the second element from `names` is paired with the second element of `numbers`, and so forth.

A dictionary is sometimes called an *associative array* because its elements (values) are associated with keys instead of indices. The placement and lookup of an element within a dictionary uses a process known as *hashing*. A hash function maps a key to a location within the dictionary where the key's associated value resides. Python dictionaries are related to *hash tables* in computer science. See http://en.wikipedia.org/wiki/Hash_table for more information about hash functions and hash tables. The important thing to know about the hashing process is that it makes value lookup via a key very fast.

When treated as a Boolean expression, the empty dictionary (`{}`) is interpreted as `False`, and any other dictionary is considered `True`.

12.4 Using Dictionaries

You should use a dictionary when you need fast and convenient access to an element of a collection based on a search key rather than an index. Consider the problem of implementing a simple telephone contact list. Most people are very familiar with the names of their friends, family, and business contacts but can remember only a handful of telephone numbers. A contact list associates a name with a telephone number.

It would be inappropriate to place the names in a list and locate a name using the associated phone number as an index into the list. This look-up method is backwards—we do not want to find a name given a phone number; we want to look up a number based on a name. Besides, each phone number contains many digits, and we would not need or want to have a list with indices with values that large—most of the space in the list would be unused.

In our situation a person or company's name is a unique identifier for that contact. In this case the name is a *key* to that contact. A Python dictionary is the ideal data structure for mapping keys to values. A dictionary allows for the fast retrieval of a value given its associated key. Listing 12.6 (`phonelist.py`) uses a Python dictionary to implement a simple telephone contact database with a rudimentary command line interface.

Listing 12.6: `phonelist.py`

```
contacts = {} # The global telephone contact list

running = True

while running:
```

```

command = input('A)dd D)delete L)ook up Q)uit: ')
if command == 'A' or command == 'a' :
    name = input('Enter new name:')
    print('Enter phone number for', name, end=':')
    number = input()
    contacts[name] = number
elif command == 'D' or command == 'd':
    name = input('Enter name to delete :')
    del contacts[name]
elif command == 'L' or command == 'l':
    name = input('Enter name :')
    print(name, contacts[name])
elif command == 'Q' or command == 'q':
    running = False
elif command == 'dump': # Secret command
    print(contacts)
else:
    print(command, 'is not a valid command')

```

The following shows a sample run of Listing 12.6 (phonelist.py):

```

A)dd D)delete L)ook up Q)uit: a
Enter new name:Fred
Enter phone number for Fred:423-123-0134
A)dd D)delete L)ook up Q)uit: dump
{'Fred': '423-555-0134'}
A)dd D)delete L)ook up Q)uit: a
Enter new name:Wilma
Enter phone number for Wilma:423-453-0128
A)dd D)delete L)ook up Q)uit: l
Enter name :Wilma
Wilma 423-555-0128
A)dd D)delete L)ook up Q)uit: l
Enter name :Fred
Fred 423-555-0134
A)dd D)delete L)ook up Q)uit: dump
{'Wilma': '423-555-0128', 'Fred': '423-123-0134'}
A)dd D)delete L)ook up Q)uit: d
Enter name to delete :Wilma
A)dd D)delete L)ook up Q)uit: dump
{'Fred': '423-555-0134'}
A)dd D)delete L)ook up Q)uit: q

```

Listing 12.7 (translateif.py) translates some Spanish words into English.

Listing 12.7: translateif.py

```

word = '*' # Initial word to ensure loop entry
while word != '': # Loop until user presses return by itself
    # Obtain word from the user
    word = input('Enter Spanish word:')
    if word == 'uno':
        print('one')
    elif word == 'dos':
        print('two')

```

```

elif word == 'tres':
    print('three')
elif word == 'cuatro':
    print('four')
elif word == 'cinco':
    print('five')
elif word == 'seis':
    print('six')
elif word == 'siete':
    print('seven')
elif word == 'ocho':
    print('eight')
else:    # Unknown word
    print('???')

```

Listing 12.7 (`translateif.py`) can successfully translate eight Spanish words into English. If we wish to increase the program’s vocabulary, we must modify the program’s logic by adding another `elif` block for each new word. Listing 12.8 (`translatedictionary.py`) uses a dictionary to assist the translation.

Listing 12.8: `translatedictionary.py`

```

translator = {'uno': 'one',
              'dos': 'two',
              'tres': 'three',
              'cuatro': 'four',
              'cinco': 'five',
              'seis': 'six',
              'siete': 'seven',
              'ocho': 'eight'}

word = '*'
while word != '':    # Loop until user presses return by itself
    # Obtain word from the user
    word = input('Enter Spanish word:')
    if word in translator:
        print(translator[word])
    else:
        print('???')    # Unknown word

```

We do not need to touch the program’s logic at all to expand the program’s vocabulary; all we need do is add the appropriate *key:value* item to the dictionary. This is a significant difference if wish to include enough words to make the program practical.

12.5 Counting with Dictionaries

Dictionaries are useful for counting things. We have experience using variables to count; recall Listing 5.3 (`countup.py`), Listing 5.12 (`countvowels.py`), Listing 5.32 (`startree.py`), Listing 6.7 (`measureprimespeed.py`), Listing 7.15 (`treefunc.py`), or Listing 10.24 (`timeprimes.py`). These programs all have counted one thing at a time, so they each use just one counter variable. In general, we need to use a separate variable for each count we manage. The following code counts the number of negative and non-negative numbers in a list of numbers and returns a tuple with the results:

```
def count_neg_nonneg(nums):
```

```

# Initialize counters
neg_count, nonneg_count = 0, 0
for num in nums:
    if num < 0:
        neg_count += 1
    else:
        nonneg_count += 1
return neg_count, nonneg_count

```

Since we needed to count two different kinds of things, we had to use two separate counter variables.

What if we face a situation in which we must count multiple kinds of things, but we cannot know ahead of time how many kinds of things there will be to count? How can we determine how many counter variables to use in a program that attempts to solve such a problem?

The answer is this: We cannot know how many counter variables we will need, so we must use a different approach. If all the things we need to count are immutable objects, like numbers or strings, we can use the objects as keys in a dictionary and associate with each key a count. As a concrete example, Listing 12.9 (wordcount.py) reads the content of a text file containing words. After reading the file the program prints a count of each word. To simplify things, the text file contains only words with no punctuation. The user supplies the file name on the command line when launching the program (see Section 10.12).

Listing 12.9: wordcount.py

```

''' Uses a dictionary to count the number of occurrences of each
    word in a text file. '''

import sys      # For sys.argv global command line arguments list

def main():
    ''' Counts the words in a text file. '''
    if len(sys.argv) < 2:  # The user supply a file name?
        print('Usage: python wordcount <filename>')
        print('      where <filename> is the name of a text file.')
    else:  # User provided file name
        filename = sys.argv[1]
        counters = {}      # Initialize counting dictionary
        with open(filename, 'r') as f:  # Open the file for reading
            content = f.read()  # Read in content of the entire file
            words = content.split()  # Make list of individual words
            for word in words:
                word = word.upper()  # Make the word all caps
                if word not in counters:
                    counters[word] = 1  # First occurrence, add the counter
                else:
                    counters[word] += 1  # Increment existing counter
            # Report the counts for each word
            for word, count in counters.items():
                print(word, count)

if __name__ == '__main__':
    main()

```

The following paragraph appears in the the *Declaration of Independence of the United States* (all punctuation has been removed):

When in the Course of human events it becomes necessary for one people to dissolve the political bands which have connected them with another and to assume among the powers of the earth the separate and equal station to which the Laws of Nature and of Nature's God entitle them a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation

The following shows a sample run of Listing 12.9 (wordcount.py) when presented with the above text file:

```
THEY 1
COURSE 1
AMONG 1
OPINIONS 1
SEPARATE 1
DECENT 1
NATURE 1
NECESSARY 1
THAT 1
IMPEL 1
IN 1
NATURE'S 1
PEOPLE 1
ANOTHER 1
STATION 1
REQUIRES 1
CONNECTED 1
ASSUME 1
CAUSES 1
EVENTS 1
TO 5
A 1
GOD 1
FOR 1
POWERS 1
BANDS 1
IT 1
THE 9
EQUAL 1
DISSOLVE 1
RESPECT 1
EARTH 1
LAWS 1
WHEN 1
MANKIND 1
ONE 1
SHOULD 1
WHICH 3
POLITICAL 1
THEM 3
AND 3
OF 5
HUMAN 1
BECOMES 1
SEPARATION 1
WITH 1
ENTITLE 1
```

```
DECLARE 1
HAVE 1
```

In Listing 12.9 (`wordcount.py`), since we cannot predict what words will appear in the document, we cannot use a separate variable for each counter. Instead, we use the user's words as keys in a dictionary. For each key in the dictionary we associate an integer value that keeps track of the number of times the word appears in the file.

The following expression in Listing 12.9 (`wordcount.py`):

```
content.split()
```

exercises a method of the `str` class that separates the very long string composed of all the words in the file into separate strings. The `split` method divides the string based on whitespace (spaces, tabs, and newlines) and returns the individual words in a list. The following interactive sequence shows how the `split` method works:

```
>>> s = '   ABC  def GHI JKLM-nop, aaa  '
>>> s
'   ABC  def GHI JKLM-nop, aaa  '
>>> s.split()
['ABC', 'def', 'GHI', 'JKLM-nop,', 'aaa']
```

With no arguments, `str.split` method splits the string based on whitespace (spaces, tabs, and newlines). Whitespace separates the words to place in the list. The `split` function accepts an optional string parameter that contains the characters used to separate the words (or *tokens*); for example,

```
>>> x = 'ABC:xyz.122:prst'
>>> x
'ABC:xyz.122:prst'
>>> x.split()
['ABC:xyz.122:prst']
>>> x.split(':')
['ABC', 'xyz.122', 'prst']
```

12.6 Grouping with Dictionaries

Dictionaries are useful for grouping items. Like Listing 12.9 (`wordcount.py`), Listing 12.10 (`groupwords.py`) reads in the contents of a text file. Instead of counting the words, Listing 12.10 (`groupwords.py`) groups the words into lists based on the length (number of letters) in the word. All the words containing only one letter are in one list, all the words containing two letters are in another list, etc.

Listing 12.10: `groupwords.py`

```
''' Uses a dictionary to group the words in a text file according to
    their length (number of letters). '''

import sys      # For argv global command line arguments list

def main():
    ''' Group the words by length in a text file. '''
    if len(sys.argv) < 2:  # The use supply a file name?
```

```

    print('Usage: python groupwords.py <filename>')
    print('    where <filename> is the name of a text file.')
else: # User provided file name
    filename = sys.argv[1]
    groups = {} # Initialize grouping dictionary
    with open(filename, 'r') as f: # Open the file for reading
        content = f.read() # Read in content of the entire file
        words = content.split() # Make list of individual words
        for word in words:
            word = word.upper() # Make the word all caps
            # Compute the word's length
            size = len(word)
            if size in groups:
                if word not in groups[size]: # Avoid duplicates
                    groups[size] += [word] # Add the word to its group
            else:
                groups[size] = [word] # Add the word to a new group
    # Show the groups
    for size, group in groups.items():
        print(size, ': ', group)

if __name__ == '__main__':
    main()

```

The following shows a sample run of Listing 12.10 (groupwords.py) on our snippet from the *Declaration of Independence*:

```

1 : ['A']
2 : ['IN', 'OF', 'IT', 'TO']
3 : ['THE', 'FOR', 'ONE', 'AND', 'GOD']
4 : ['WHEN', 'HAVE', 'THEM', 'WITH', 'LAWS', 'THAT', 'THEY']
5 : ['HUMAN', 'BANDS', 'WHICH', 'AMONG', 'EARTH', 'EQUAL', 'IMPEL']
6 : ['COURSE', 'EVENTS', 'PEOPLE', 'ASSUME', 'POWERS', 'NATURE', 'DECENT', 'SHOULD', 'CAUSES']
7 : ['BECOMES', 'ANOTHER', 'STATION', 'ENTITLE', 'RESPECT', 'MANKIND', 'DECLARE']
8 : ['DISSOLVE', 'SEPARATE', 'NATURE'S', 'OPINIONS', 'REQUIRES']
9 : ['NECESSARY', 'POLITICAL', 'CONNECTED']
10 : ['SEPARATION']

```

Each key represents the length of all the strings in the list it oversees.

12.7 Implementing Mathematical Graphs

CAUTION! SECTION UNDER CONSTRUCTION

Dictionaries are ideal for representing relationships among things.

12.8 Keyword Arguments

The following function specifies formal parameters named a, b, and c:

```
def process(a, b, c):  
    print('a =', a, ' b =', b, ' c =', c)
```

The following code uses function process, passing actual parameters 2, x (that is, 14), and 10:

```
x = 14  
process(2, x, 10)
```

It prints

```
a = 2  b = 14  c = 10
```

The calling code assigns the value of the first actual parameter to the first formal parameter. It assigns the value of the second parameter to the second formal parameter. Finally, it assigns the value of the third actual parameter to the third formal parameter. By default, the association of actual parameter to formal parameter during a function invocation is strictly positional. This is the shortest, simplest way for the caller to pass parameters.

Python allows the caller to pass its actual parameters in any order using a technique known as *keyword arguments*. We introduced the end and sep keyword arguments for the print function in Section 2.8. In order to use keyword arguments, the caller must know the names of the function's formal parameters. Listing 12.11 (namedparams.py) shows how callers can use keyword parameters.

Listing 12.11: namedparams.py

```
def process(a, b, c):  
    print('a =', a, ' b =', b, ' c =', c)  
  
x = 14  
process(1, 2, 3)  
process(a=10, b=20, c=30)  
process(b=200, c=300, a=100)  
process(c=3000, a=1000, b=2000)  
process(10000, c=30000, b=20000)
```

Listing 12.11 (namedparams.py) prints the following:

```
a = 1  b = 2  c = 3  
a = 10 b = 20 c = 30  
a = 100 b = 200 c = 300  
a = 1000 b = 2000 c = 3000  
a = 10000 b = 20000 c = 30000
```

The statement

```
process(10000, c=30000, b=20000)
```

shows that keywords arguments may appear in the same call as non-keyword arguments, but in such mixed-parameter calls all non-keyword arguments must appear before any keyword arguments. The function invocation mechanism assigns the non-keyword arguments as usual: the first actual parameter to the first formal parameter, second actual parameter to the second formal parameter, etc. It assigns the keyword arguments that follow to the formal parameters of the same name.

We can define a function to require keyword parameters by prefixing a formal parameter with two asterisks (**). The following function requires the caller to pass three actual parameters named a, b, and c:

```
def f(**args):
    a = args['a']
    b = args['b']
    c = args['c']
    return 2*a*a + 3*b + c
```

Given this definition of `f`, a caller can invoke `f` as

```
x = f(a=4, b=11, c=3)
```

and the following statement:

```
x = f(4, 11, c)    # Illegal, f requires keyword arguments
```

is illegal.

The following function prints the names of the keyword arguments passed in by the caller:

```
def process(**args):
    for arg in args:
        print(arg)
```

This `process` function is designed to accept any keyword arguments the caller chooses to pass. Consider the following interactive sequence:

```
>>> def process(**args):
...     for arg in args:
...         print(arg, '-->', args[arg])
...     print('args =', args)
...
>>> process(num=5, x='Hello', value=True, zz=100)
num --> 5
zz --> 100
value --> True
x --> Hello
args = {'num': 5, 'zz': 100, 'value': True, 'x': 'Hello'}
```

As we can see, the formal parameter `**args` is really a dictionary. All the keyword arguments become keys and values in the dictionary. The parameter's name becomes a key, and the associated value is the caller's value assigned to the keyword argument name.

As with arbitrary argument lists, we can mix regular positional parameters with the special `**` keyword argument parameter. We actually can mix regular parameters, arbitrary argument lists, and keyword arguments as long as we use the proper order:

```
def f(x, y, z, *a, **b):
    pass
```

In this function `x`, `y`, and `z` are regular positional arguments, `a` is the arbitrary arguments tuple, and `b` is the keywords arguments dictionary. The positional arguments, if any, must appear before any arbitrary arguments and keyword arguments. The arbitrary arguments, if any, must appear after the positional arguments and before the keyword arguments. The keyword arguments, if any, must appear after the positional and arbitrary argument list parameters.

As with arbitrary argument list arguments, we can use keyword arguments on the caller side. Listing 12.12 (`callerkeyword.py`) shows how we can send a dictionary as a parameter to a function that expects regular positional parameters.

Listing 12.12: callerkeyword.py

```
def f(a, b, c):  
    print('a =', a, ' b =', b, ' c =', c)  
  
f(1, 2, 3)                      # Pass three parameters  
dict = {}  
dict['b'] = 22  
dict['a'] = 11  
dict['c'] = 33  
f(**dict)  
f(**{'a':10, 'b':20, 'c':30})  # Pass a dictionary
```

Listing 12.12 (callerkeyword.py) prints

```
a = 1  b = 2  c = 3  
a = 11 b = 22 c = 33  
a = 10 b = 20 c = 30
```

Observe that the caller must use the `**` prefix when passing the dictionary in the place of the expected positional parameters.

12.9 Sets

Python provides a data structure that represents a mathematical set. As with mathematical sets, we use curly braces (`{}`) in Python code to enclose the elements of a literal set. Python distinguishes between set literals and dictionary literals by the fact that all the items in a dictionary are colon-connected (`:`) key-value pairs, while the elements in a set are simply values. Unlike Python lists, sets are unordered and may contain no duplicate elements. The following interactive sequence demonstrates these set properties:

```
>>> S = {10, 3, 7, 2, 11}  
>>> S  
{2, 11, 3, 10, 7}  
>>> T = {5, 4, 5, 2, 4, 9}  
>>> T  
{9, 2, 4, 5}
```

Note the element ordering of the input is different from the ordering in the output. Also observe that sets do not admit duplicate elements.

We can make a set out of a list using the `set` conversion function:

```
>>> L = [10, 13, 10, 5, 6, 13, 2, 10, 5]  
>>> S = set(L)  
>>> L  
[10, 13, 10, 5, 6, 13, 2, 10, 5]  
>>> S  
{10, 2, 13, 5, 6}
```

As you can see, the element ordering is not preserved, and duplicate elements appear only once in the set.

Python set notation exhibits one important difference with mathematics: the expression `{}` does not represent the empty set. In order to use the curly braces for a set, the set must contain at least one element.

Operation	Mathematical Notation	Python Syntax	Type	Meaning
Union	$A \cup B$	<code>A B</code>	set	Elements in A or B or both
Intersection	$A \cap B$	<code>A & B</code>	set	Elements common to both A and B
Set Difference	$A - B$	<code>A - B</code>	set	Elements in A but not in B
Symmetric Difference	$A \oplus B$	<code>A ^ B</code>	set	Elements in A or B , but not both
Set Membership	$x \in A$	<code>x in A</code>	Boolean	x is a member of A
Set Membership	$x \notin A$	<code>x not in A</code>	Boolean	x is not a member of A

Table 12.2: Python set operations. Figure 12.1 illustrates how the set operations work.

The expression `set()` produces a set with no elements, and thus represents the empty set. Python reserves the `{}` notation for empty *dictionaries* (see Section 12.3).

Unlike in mathematics, all sets in Python must be finite. Python supports the standard mathematical set operations of intersection, union, set difference, and symmetric difference. Table 12.2 shows the Python syntax for these operations. Figure 12.1 illustrates how the set operations work. The following interactive

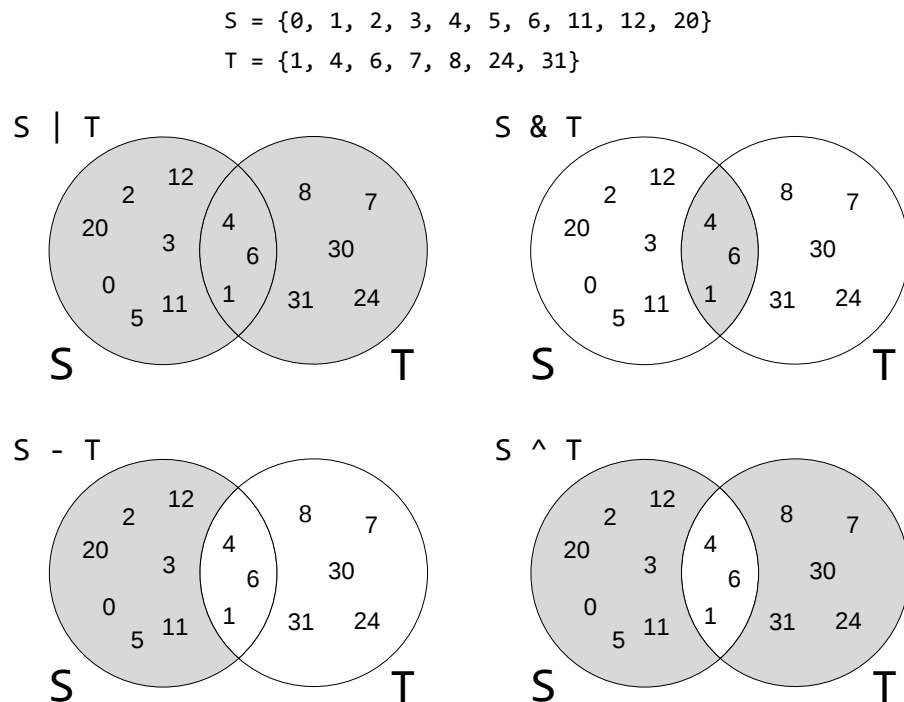


Figure 12.1: Python's set operations: union, intersection, set difference, and symmetric difference. The shaded area in each diagram indicates the elements that are included in the set that results from applying the indicated operator.

sequence computes the union and intersection and two sets and tests for set membership:

```
>>> S = {2, 5, 7, 8, 9, 12}
>>> T = {1, 5, 6, 7, 11, 12}
>>> S | T
{1, 2, 5, 6, 7, 8, 9, 11, 12}
>>> S & T
{12, 5, 7}
>>> 7 in S
True
>>> 11 in S
False
```

As with list comprehensions and generator expressions (Section 10.13), we can use *set comprehension* to build sets. The syntax is the same as for list comprehension, except we use curly braces rather than square brackets. The following interactive sequence constructs the set of perfect squares less than 100:

```
>>> S = {x**2 for x in range(10)}
>>> S
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
```

The displayed order of elements is not as nice as the list version, but, again, element ordering is meaningless with sets.

When treated as a Boolean expression, the empty set (`set()`) is interpreted as `False`, and any other set is considered `True`.

12.10 Set Quantification with `all` and `any`

Python provides functions named `all` and `any` that respectively correspond to mathematical *universal* and *existential quantification*. These fancy mathematical terms label relatively simple concepts. Universal quantification means that a particular property is true for all the elements of a set. Existential quantification means that at least one element in the set exhibits a particular property. In mathematics the $\forall$ symbol represents universal quantification, and the $\exists$ symbol represents existential quantification. The $\forall$ symbol usually is pronounced “for all,” and the $\exists$ symbol is read as “there exists.”

To see how we can use these quantifiers in a Python program, consider the set $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$. In the interactive shell we can type

```
>>> S = {1, 2, 3, 4, 5, 6, 7, 8}
>>> S
{1, 2, 3, 4, 5, 6, 7, 8}
```

To express in mathematics the fact that all the elements in set S are greater than zero, we can write $(\forall x \in S)(x > 0)$. This is a statement that is either true or false, and we can see that it is a true statement. In Python, we first will use a list comprehension to see which elements in S are greater than zero. We can do this by building a list of Boolean values by using a Boolean expression in the list comprehension:

```
>>> [x > 0 for x in S]
[True, True, True, True, True, True, True, True]
```

We can see that all the entries in this list are `True`, but the best way to determine this in code is to use Python’s `all` function:

```
>>> all([x > 0 for x in S])
True
```

The `all` function returns `True` if all the elements in a list, set, or other iterable possesses a particular quality. We do not need to create a list; a generator expression is better (note that parentheses replace the square brackets):

```
>>> all((x > 0 for x in S))
True
```

and in this case the inner parentheses are superfluous. We can write the expression as

```
>>> all(x > 0 for x in S)
True
```

The expression `all(x > 0 for x in S)` is Python's way of representing the mathematical predicate $(\forall x \in S) (x > 0)$.

The `any` function returns `True` if any element in a list, set, or other iterable possesses a particular quality. This means the `any` function represents the mathematical existential quantifier, $\exists$:

```
>>> any(x > 0 for x in S)
True
```

The expression `any(x > 0 for x in S)` is Python's way of representing the mathematical predicate $(\exists x \in S) (x > 0)$. Certainly if the property holds for all the elements in set S , there is at least one element for which it holds.

Are all the elements of S greater than 5?

```
>>> all(x > 5 for x in S)
False
```

The answer is false, of course, because the set contains 1, 2, 3, 4, and 5, none of which are greater than 5. There are some elements in S that are greater than 5:

```
>>> any(x > 5 for x in S)
True
```

The elements 6, 7, and 8 are all greater than 5. Is it true that the set contains an element greater than 10?

```
>>> any(x > 10 for x in S)
False
```

We can see that none of the elements in S are greater than 10. If none of the set's elements possess the particular property, it certainly cannot be true for all the elements in the set:

```
>>> all(x > 10 for x in S)
False
```

The `all` and `any` functions work with any iterable object: sets, lists, dictionaries, and generated sequences.

In most Python programming, sets play much smaller role than lists and dictionaries. Sets are most similar to lists, and the ordering of data is important in many applications. If order does not matter and all

elements are unique, the set type does offer a big advantage over the list type: testing for membership using `in` is much faster on sets than lists. Listing 12.13 (`setvslistaccess.py`) creates both a set and a list, each containing the first 1,000 perfect squares. It then searches both data structures for, and does nothing with, all the integers from 0 to 999,999. It reports the time required for the efforts.

Listing 12.13: `setvslistaccess.py`

```
# Data structure size
size = 1000

# Make a big set
S = {x**2 for x in range(size)}
# Make a big list
L = [x**2 for x in range(size)]

# Verify the type of S and L
print('Set:', type(S), ' List:', type(L))

from time import clock

# Search size
search_size = 1000000

# Time list access
start_time = clock()
for i in range(search_size):
    if i in L:
        pass
stop_time = clock()
print('List elapsed:', stop_time - start_time)

# Time set access
start_time = clock()
for i in range(search_size):
    if i in S:
        pass
stop_time = clock()
print('Set elapsed:', stop_time - start_time)
```

The results of Listing 12.13 (`setvslistaccess.py`) are dramatic. A run on one system reports:

```
Set: <class 'set'> List: <class 'list'>
List elapsed: 44.99767441164282
Set elapsed: 0.48652052551967984
```

The 1,000,000 list accesses required about three-quarters of a minute, while the set accesses needed less than one-half second. The set membership test was almost 100 times faster than the exact same test performed on the list.

Listing 12.9 (`wordcount.py`) grouped words from a text file according to their length. The program contained a check to avoid duplicate entries:

```
if size in groups:
    if word not in groups[size]: # Avoid duplicates
        groups[size] += [word] # Add the word to its group
```

```
else:
    groups[size] = [word]    # Add the word to a new group
```

We know now that if we used sets of words rather than lists of words we could have eliminated the check for duplicate entries.

```
if size in groups:
    groups[size] += {word}   # Add the word to its group
else:
    groups[size] = {word}    # Add the word to a new group
```

By removing this extra check we also remove the application of the `in` operator on a list. We have seen that testing for membership within a list is more costly than testing for membership within a set. Eliminating this check removes the potentially costly search for an element within a large list.

12.11 Summary

CAUTION! SECTION UNDER CONSTRUCTION

- A tuple is similar to a list in that it is an ordered collection of elements. Unlike a list, a tuple is immutable, meaning an executing program cannot modify the contents of a tuple object.
- As with lists, we can use an index within square brackets to access individual components within a tuple. As with lists, the first element of a tuple is at index zero.
- Python uses tuples to enable arbitrary argument lists.
- A formal parameter prefixed with an asterisk (*) represents a “hidden” tuple in which a caller can pack an arbitrary number of actual parameters.
- A dictionary is an associative container in which elements are accessed via a key rather than an index.
- A dictionary key must be an instance of an immutable type. Integers, floating-point numbers, strings, Booleans, and tuples all constitute valid key types, but lists, generators, dictionaries, and sets may not serve as dictionary keys.
- Python uses dictionaries to enable keyword arguments for function parameters.
- A formal parameter prefixed with two asterisks (**) indicates that the function requires keyword arguments.
- Python’s set type represents a mathematical set.
- The set type supports the intersection, union, set difference, and symmetric difference set operations.

12.12 Exercises

CAUTION! SECTION UNDER CONSTRUCTION

1. Add item

Chapter 13

Handling Exceptions

In our programming experience so far we have encountered several kinds of run-time exceptions, such as division by zero, accessing a list with an out-of-range index, and attempting to convert a non-number to an integer. To this point, all of our run-time exceptions have terminated the running program. Python provides a standard mechanism called *exception handling* that allows programmers to deal with these kinds of run-time exceptions and many more. Rather than always terminating the program's execution, an executing program can detect the problem when it arises and possibly execute code to correct the issue or mitigate it in other ways. This chapter explores handling exceptions in Python.

13.1 Motivation

Algorithm design can be tricky because the details are crucial. It may be straightforward to write an algorithm to solve a problem in the general case, but the designer may have to address a number of special cases within the problem for the algorithm to be correct. Some of these special cases might occur rarely and only under the most extraordinary circumstances. The algorithm must properly handle these exceptional cases to be truly robust; however, adding the necessary details to the algorithm may render it overly complex and difficult to construct correctly. Such an overly complex algorithm would be difficult for others to read and understand, and it would be harder to debug and extend.

Ideally, a developer would write the algorithm in its general form including any common special cases. Exceptional situations that should arise rarely, along with a strategy to handle them, could appear elsewhere, perhaps as an annotation to the algorithm. This approach would focus the algorithm on its routine activity and keep its rare behavior tucked out of sight until specifically needed.

Python's exception handling infrastructure allows programmers to cleanly separate the code that implements the focused algorithm from the code that deals with exceptional situations that the algorithm may face. This approach is more modular and encourages the development of code that is cleaner and easier to maintain and debug.

An *exception* is a special object that the executing program can create when it encounters an extraordinary situation. Such a situation almost always represents a problem, usually some sort of run-time error. Examples of exceptional situations include:

- attempting to read past the end of a file

- evaluating the expression `lst[i]` where `lst` is a list, and $i \geq \text{len}(\text{lst})$.
- attempting to convert a non-numeric string to a number, as in `int('Fred')`
- Attempting to read a variable that has not been defined
- attempting to read data from the network when the connection is lost (perhaps due to a server crash or the wire being unplugged from the port).

The algorithm may handle many of these potential problems itself. For example, a programmer can use an `if` statement to determine if a list index is within the bounds of a list. Fortunately, specific Python exceptions are available to cover problems such as these.

Exceptions represent a standard way to deal with run-time errors. In programming languages that do not support exception handling, programmers must devise their own ways of dealing with exceptional situations. One common approach is for functions to return an integer code that represents success or failure. For example, consider a function named `ReadFile` that is to open a file and read its contents. It returns an integer that is interpreted as follows:

- 0: Success; the function successfully opened and read the contents of the file
- 1: File not found error; the requested file does not exist
- 2: Permissions error; the program is not authorized to read the file
- 3: Device not ready error; for example, a DVD is not present in the drive
- 4: Media error; the program encountered bad sectors on the disk while reading the file
- 5: Some other file error

Notice that zero indicates success, and nonzero indicates failure. Calling code that uses the function may look like

```
if ReadFile("stats.data") == 0:
    # Code to execute if the file was read properly
else:
    # Code to execute if an error occurred while reading the file
```

The developers of `ReadFile` were looking toward the future, since any value above 4 represents some unspecified file error. They may add functionality to `ReadFile` in the future and devise more error codes (for example, 5 may mean illegal file format). Existing code that calls the updated `ReadFile` function will still work (5 > 4 just represents some kind of file error), but new calling code can explicitly check for a return value of 5 and act accordingly.

This kind of error handling has its limitations, however. The primary purpose of some functions is to return an integer result that is not an indication of an error (for example, the `int` function). Perhaps a string could be returned instead? Unfortunately, some functions naturally return strings (like the `str` function). Also, returning a string would not work for a function that naturally returns an integer as its result. A completely different type of exception handling technique would need to be developed for functions such as these.

The return-value-as-error-status approach can be cumbersome to use for complicated programming situations. Consider the situation where function A calls function B which calls function C which calls function D which calls our `ReadFile` function from above. We can visualize this call chain as

Class	Meaning
AttributeError	Object does not contain the specified field or method
ImportError	The <code>import</code> statement fails to find a specified module or name in that module
IndexError	A sequence (list, string, tuple) index is out of range
KeyError	Specified key does not appear in a dictionary
NameError	Specified local or global name does not exist
TypeError	Operation or function applied to an inappropriate type
ValueError	Operation or function applied to correct type but inappropriate value
ZeroDivisionError	Second operand of division or modulus operation is zero

Table 13.1: Common standard exception classes

$A \rightarrow B \rightarrow C \rightarrow D \rightarrow \text{ReadFile}$

Suppose function A is concerned about the file being opened correctly. The ReadFile function returns an error status, but this value is returned to function D, the function that calls ReadFile directly. If A really needs to know about how ReadFile worked, then all the functions in between in the call chain (B, C, and D) must also return an error status. The process essentially passes the error status of ReadFile back up the call chain to A. While this is inconvenient at best, it may be impossible in general. Suppose D's job is to read the data in the file (via ReadFile) and then pass each piece of data read to another function called Process. Now Process also returns an integer value that indicates its error status. If the data passed to Process is not of the proper format, it returns 1; otherwise, it returns 0. If function A needs to know specifics about why the data file was not properly read in and processed (was it a problem reading the file with ReadFile or a problem with the data format with Process?), it cannot distinguish the cause from the single error code passed up the call chain.

Another problem with an ad hoc approach to error handling like this one is that the error handling facilities developed by one programmer may be incompatible with those used by another. We need a comprehensive, uniform exception handling mechanism, and Python's exceptions provide such a framework. The proper use of Python's exception handling infrastructure leads to code that is logically cleaner and less prone to programming errors. The standard Python library uses exceptions, and programmers can create new exceptions that address issues specific to their particular problems. These exceptions all use a common syntax and are completely compatible with each other.

13.2 Common Standard Exceptions

We have encountered a number of Python's standard exception classes. Table 13.1 lists some of the more common exception classes.

The following interactive sequence provides an example of each of the exceptions shown in Table 13.1:

```
>>> from fractions import Fraction
>>> frac = Fraction(1, 2)
>>> print(frac.numerator)
1
>>> print(frac.numertor)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Fraction' object has no attribute 'numertor'
```

```
>>> from fraction import Fraction
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fraction'
>>> from fractions import Fractions
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: cannot import name 'Fractions'
>>> seq = [2, 5, 11]
>>> print(seq[1])
5
>>> print(seq[3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> d = {}
>>> d['Fred'] = 100
>>> print(d['Fred'])
100
>>> print(d['Freddie'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Freddie'
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> print(seq['Fred'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers, not str
>>> print(int('Fred'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Fred'
>>> print(1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Python contains many more standard exception classes than those shown in Table 13.1, and in Section 14.6 we will explore ways to create our own custom exception classes.

13.3 Handling Exceptions

Listing 13.1 (dividenumbers.py) computes the quotient of two integer values supplied by the user.

Listing 13.1: dividenumbers.py

```
num1, num2 = eval(input("Please enter two numbers: "))
print('{0} divided by {1} = {2}'.format(num1, num2, num1/num2))
```

In Listing 13.1 (dividenumbers.py), all is well until the user attempts to divide by zero:

```
Please enter two integers: 4, 0
Traceback (most recent call last):
  File "dividenumbers.py", line 2, in <module>
    print('{0} divided by {1} = {2}'.format(num1, num2, num1/num2))
ZeroDivisionError: division by zero
```

This program execution produces a `ZeroDivisionError` exception.

We can defend against the `ZeroDivisionError` exception with a conditional statement, as Listing 13.2 (checkforzero.py) shows.

Listing 13.2: checkforzero.py

```
num1, num2 = eval(input("Please enter two numbers: "))
if num2 != 0:
    print('{0} divided by {1} = {2}'.format(num1, num2, num1/num2))
else:
    print('Cannot divide by zero')
```

The solution expressed by Listing 13.2 (checkforzero.py) illustrates the concept of *look before you leap*, usually abbreviated as *LBYL* in the Python community. This programming idiom checks code that may misbehave before executing it. The LBYL idiom works well for code like that found in Listing 13.2 (checkforzero.py).

Consider Listing 13.3 (enterinteger.py) that asks the user for a small integer value.

Listing 13.3: enterinteger.py

```
val = int(input("Please enter a small positive integer: "))
print('You entered', val)
```

A typical run of Listing 13.3 (enterinteger.py) looks like

```
Please enter a small positive integer: 5
You entered 5
```

A user easily and innocently can thwart the programmer's original intentions, as the following sample run illustrates:

```
Please enter a small positive integer: five
Traceback (most recent call last):
  File "enterinteger.py", line 1, in <module>
    val = int(input("Please enter a small positive integer: "))
ValueError: invalid literal for int() with base 10: 'five'
```

For an English-speaking human, the response *five* should be just as acceptable as 5. The strings acceptable to the Python `int` function, however, can contain only numeric characters and an optional leading sign character (+ or -). The user's input causes the program to produce a run-time exception. As it stands, the program reacts to the exception by printing a message and terminating itself. As shown in the exception error report, the kind of exception that this execution example produces is a `ValueError` exception.

Unfortunately, any attempt to make Listing 13.3 (enterinteger.py) more robust via the LBYL idiom is not as easy as it is for Listing 13.2 (checkforzero.py). We basically need to determine if the arbitrary

string the user enters is acceptable to the `int` conversion function. The string must contain only the digit characters `'0'`, `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, `'6'`, `'7'`, `'8'`, or `'9'`, and it may contain a leading `'-'` or `'+'` character indicating the number's sign. Python's *regular expression* library is ideal for this purpose, but it is somewhat complicated and deserves an entire chapter devoted to its use. Short of using the regular expression library, the logic to ensure that the string is acceptable to the `int` function would be relatively complex.

An alternative to LBYL is *EAFP*, which stands for *easier to ask for forgiveness than permission*. The EAFP approach attempts to execute the potentially problematic code within a `try` statement. If the code raises an exception, the program's execution does not necessarily terminate; instead, the program's execution jumps to a different block within the `try` statement. Listing 13.4 (`enterintexcept.py`) wraps the code from Listing 13.3 (`enterinteger.py`) within a `try` statement to successfully defend against bad user input.

Listing 13.4: `enterintexcept.py`

```
try:
    val = int(input("Please enter a small positive integer: "))
    print('You entered', val)
except ValueError:
    print('Input not accepted')
```

The two statements between `try` and `except` constitute the `try` block. The statement after the `except` line represents an `except` block. If the user enters a string unacceptable to the `int` function, the `int` function will raise a `ValueError` exception. At this point the program will not complete the assignment statement nor will it execute the `print` statement that follows. Instead the program immediately will begin executing the code in the `except` block. This means if the user enters *five*, the program will print the message *Input not accepted*. If the user enters a convertible string like *5*, the program will complete the `try` block and bypass the code in the `except` block. Figure 13.1 contrasts the possible program execution flows within a `try/except` statement. We say the `except` block *handles* the exception raised in the `try` block. Another

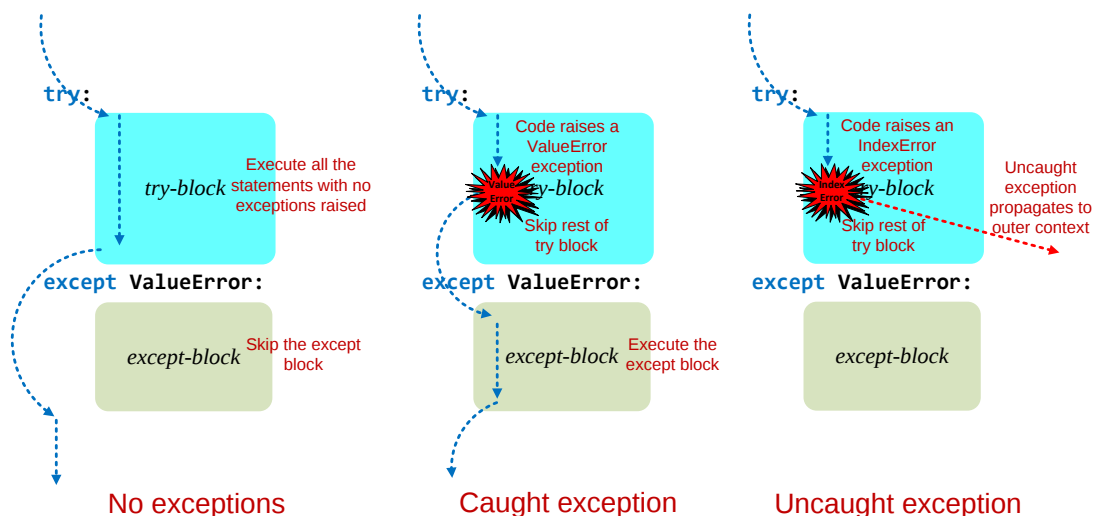


Figure 13.1: The possible program execution flows through a `try/except` statement.

common terminology used to describe the excepting handling process uses the *throw/catch* metaphor: the executing program *throws* an exception that an `except` block *catches*.

Listing 13.4 (`enterintexcept.py`) catches only exceptions of type `ValueError`. If for some reason the code within the `try` block of Listing 13.4 (`enterintexcept.py`) raises a different type of exception, this `try` statement is unable to catch it. In this case the program will behave as if the `try/except` statement were not there. Unless other exception handling code is present in the calling environment, the interpreter simply will terminate the program with an error message. Listing 13.5 (`otherexcept.py`) includes a statement that attempts to assign the value at index 2 of an empty list. Unfortunately, an empty list contains no values, so any index is outside of its range of indices.

Listing 13.5: `otherexcept.py`

```
try:
    val = int(input("Please enter a small positive integer: "))
    print('You entered', val)
    [][2] = 5    # Try to assign to a non-existent index of the empty list
except ValueError:
    print('Input not accepted')
```

Consider the following sample run of Listing 13.5 (`otherexcept.py`):

```
Please enter a small positive integer: 5
You entered 5
Traceback (most recent call last):
  File "enterintexcept.py", line 4, in <module>
    [][2] = 5    # Try to assign to a non-existent index of the empty list
IndexError: list assignment index out of range
```

We see that the `except` block in Listing 13.5 (`otherexcept.py`) is unable to catch the `IndexError` exception. Figure 13.1 illustrates this possible outcome as well.

13.4 Handling Multiple Exceptions

A `try` statement can have multiple `except` blocks. Each `except` block must catch a different type of exception object. Listing 13.6 (`multiexcept.py`) offers three `except` blocks. Its `try` statement specifically can catch `ValueError`, `IndexError`, and `ZeroDivisionError` exceptions.

Listing 13.6: `multiexcept.py`

```
import random

for i in range(10):    # Loop 10 times
    print('Beginning of loop iteration', i)
    try:
        r = random.randint(1, 3)    # r is pseudorandomly 1, 2, or 3
        if r == 1:
            print(int('Fred'))    # Try to convert a non-integer
        elif r == 2:
            [][2] = 5    # Try to assign to a non-existent index of the empty list
        else:
            print(3/0)    # Try to divide by zero
```

```
except ValueError:
    print('Cannot convert integer')
except IndexError:
    print('List index is out of range')
except ZeroDivisionError:
    print('Division by zero not allowed')

print('End of loop iteration', i)
```

Each time through the loop the code within the `try` block of Listing 13.6 (`multiexcept.py`) will raise one of three different exceptions based on the generated pseudorandom number. The program offers three `except` blocks. If the code in the `try` block raises one of the three types of exceptions, the program will execute the code in the matching `except` block. Only code in one of the three `except` blocks will execute as a result of the exception. The following shows a sample run of Listing 13.6 (`multiexcept.py`):

```
Beginning of loop iteration 0
List index is out of range
End of loop iteration 0
Beginning of loop iteration 1
Division by zero not allowed
End of loop iteration 1
Beginning of loop iteration 2
Cannot convert integer
End of loop iteration 2
Beginning of loop iteration 3
List index is out of range
End of loop iteration 3
Beginning of loop iteration 4
Cannot convert integer
End of loop iteration 4
Beginning of loop iteration 5
List index is out of range
End of loop iteration 5
Beginning of loop iteration 6
Division by zero not allowed
End of loop iteration 6
Beginning of loop iteration 7
List index is out of range
End of loop iteration 7
Beginning of loop iteration 8
List index is out of range
End of loop iteration 8
Beginning of loop iteration 9
Division by zero not allowed
End of loop iteration 9
```

Observe that only one `except` block executes each time through the loop.

If we need the exact code to handle more than one exception type, we can associate multiple types with a single `except` block by listing each exception type within a tuple. Listing 13.7 (`multihandle.py`) illustrates.

Listing 13.7: `multihandle.py`

```
import random
```

```
for i in range(10):    # Loop 10 times
    print('Beginning of loop iteration', i)
    try:
        r = random.randint(1, 3)    # r is pseudorandomly 1, 2, or 3
        if r == 1:
            print(int('Fred'))    # Try to convert a non-integer
        elif r == 2:
            [][2] = 5    # Try to assign to a non-existent index of the empty list
        else:
            print(3/0)    # Try to divide by zero
    except (ValueError, ZeroDivisionError):
        print('Problem with integer detected')
    except IndexError:
        print('List index is out of range')

    print('End of loop iteration', i)
```

The first `except` block in Listing 13.7 (`multihandle.py`) will catch both `ValueError` and `ZeroDivisionError` exceptions. As shown in Listing 13.7 (`multihandle.py`), parentheses must enclose a tuple specified in an `except` block.

In general it is better to bundle exception handlers rather than duplicating code over multiple handlers.

13.5 The Catch-all Handler

Consider Listing 13.8 (`missedexception.py`), a slight modification of Listing 13.6 (`multiexcept.py`).

Listing 13.8: `missedexception.py`

```
import random

for i in range(10):    # Loop 10 times
    print('Beginning of loop iteration', i)
    try:
        r = random.randint(1, 4)    # r is pseudorandomly 1, 2, 3, or 4
        if r == 1:
            print(int('Fred'))    # Try to convert a non-integer
        elif r == 2:
            [][2] = 5    # Try to assign to a non-existent index of the empty list
        elif r == 3:
            print({}[1])    # Try to use a non-existent key to get an item from a dictionary
        else:
            print(3/0)    # Try to divide by zero
    except ValueError:
        print('Cannot convert integer')
    except IndexError:
        print('List index is out of range')
    except ZeroDivisionError:
        print('Division by zero not allowed')

    print('End of loop iteration', i)
```

In addition to raising the three exceptions from Listing 13.6 (`multiexcept.py`), Listing 13.8 (`missedException.py`) can raise a `KeyError` exception. Unfortunately, Listing 13.8 (`missedException.py`) has no handler for the `KeyError` exception. The following output shows the results for one program run:

```
Beginning of loop iteration 0
Division by zero not allowed
End of loop iteration 0
Beginning of loop iteration 1
List index is out of range
End of loop iteration 1
Beginning of loop iteration 2
Traceback (most recent call last):
  File "missedException.py", line 12, in <module>
    print({}[1]) # Try to use a non-existent key to get an item from a dictionary
KeyError: 1
```

If we want our programs not to crash, we need to handle all possible exceptions that can arise. This is particularly important when we use libraries that we did not write. A program may execute code that only under very rare circumstances raises an exception. This situation may be so rare that it evades our thorough testing and only appears after we deploy the application to users. We need a handler that can catch *any* exception.

The Python `try` statement allows us to catch any previously uncaught exception with an untyped `except` block. Listing 13.9 (`catchallexcept.py`) shows how to use this “catch-all” exception to close the exception hole found in Listing 13.8 (`missedException.py`).

Listing 13.9: `catchallexcept.py`

```
import random

for i in range(10): # Loop 10 times
    print('Beginning of loop iteration', i)
    try:
        r = random.randint(1, 4) # r is pseudorandomly 1, 2, 3, or 4
        if r == 1:
            print(int('Fred')) # Try to convert a non-integer
        elif r == 2:
            [][2] = 5 # Try to assign to a non-existent index of the empty list
        elif r == 3:
            print({}[1]) # Try to use a non-existent key to get an item from a dictionary
        else:
            print(3/0) # Try to divide by zero
    except ValueError:
        print('Cannot convert integer')
    except IndexError:
        print('List index is out of range')
    except ZeroDivisionError:
        print('Division by zero not allowed')
    except:
        print('This program has encountered a problem')

    print('End of loop iteration', i)
```

A run of Listing 13.9 (`catchallexcept.py`) shows that it can run to completion:

```
List index is out of range
End of loop iteration 0
Beginning of loop iteration 1
Cannot convert integer
End of loop iteration 1
Beginning of loop iteration 2
This program has encountered a problem
End of loop iteration 2
Beginning of loop iteration 3
Cannot convert integer
End of loop iteration 3
Beginning of loop iteration 4
This program has encountered a problem
End of loop iteration 4
Beginning of loop iteration 5
Cannot convert integer
End of loop iteration 5
Beginning of loop iteration 6
Cannot convert integer
End of loop iteration 6
Beginning of loop iteration 7
Cannot convert integer
End of loop iteration 7
Beginning of loop iteration 8
Division by zero not allowed
End of loop iteration 8
Beginning of loop iteration 9
Division by zero not allowed
End of loop iteration 9
```

Listing 13.9 (catchallexcept.py) offers three `except` blocks. The untyped `except` block represents the “catch-all” handler that can catch any exception not caught by an earlier, typed `except` block within the `try` statement. If present, the catch-all, untyped `except` block must be the last `except` block in the `try` statement. It is a syntax error for the untyped `except` block to appear before a typed `except` block.

The catch-all handler is useful for catching exceptions from code outside the programmer’s control. Consider a program that uses a collection of functions and objects from a commercial library. The library vendor may release a newer version of the library containing bug fixes and performance enhancements. If the modifications to a function from that library make the function capable of raising a new type of exception that it did not before, callers handling only the exceptions from the previous version will crash when attempting to handle the new exception type. Callers that include a catch-all handler will not crash, but instead provide some default response.

13.6 Catching Exception Objects

The `except` blocks “catch” exception objects. We can inspect the object that an `except` block catches if we specify the object’s name with the `as` keyword. Listing 13.10 (exceptobject.py) shows how to use the `as` keyword to get access to the exception object raised by code in the `try` block.

Listing 13.10: exceptobject.py

```

import random

for i in range(10):    # Loop 10 times
    print('Beginning of loop iteration', i)
    try:
        r = random.randint(1, 4)    # r is pseudorandomly 1, 2, 3, or 4
        if r == 1:
            print(int('Fred'))    # Try to convert a non-integer
        elif r == 2:
            [][2] = 5    # Try to assign to a non-existent index of the empty list
        elif r == 3:
            print({}[1])    # Try to use a non-existent key to get an item from a dictionary
        else:
            print(3/0)    # Try to divide by zero
    except ValueError as e:
        print('Problem with value ==>', type(e), e)
    except IndexError as e:
        print('Problem with list ==>', type(e), e)
    except ZeroDivisionError as e:
        print('Problem with division ==>', type(e), e)
    except Exception as e:
        print('Problem with something ==>', type(e), e)

    print('End of loop iteration', i)

```

The last `except` block in Listing 13.10 (`exceptobject.py`) is the catch-all handler. Notice here we must specify a type: `Exception`. We must use this class name when an `except` block needs to catch any exception type *and* you need access the exception object. The type `Exception` is compatible with all the other exception types. (We will examine in Section 14.6 the concept of class inheritance which makes this type compatibility possible.) The following sample program run reveals what the exception objects print when sent to the `print` function:

```

Beginning of loop iteration 0
Problem with division ==> <class 'ZeroDivisionError'> division by zero
End of loop iteration 0
Beginning of loop iteration 1
Problem with list ==> <class 'IndexError'> list assignment index out of range
End of loop iteration 1
Beginning of loop iteration 2
Problem with division ==> <class 'ZeroDivisionError'> division by zero
End of loop iteration 2
Beginning of loop iteration 3
Problem with division ==> <class 'ZeroDivisionError'> division by zero
End of loop iteration 3
Beginning of loop iteration 4
Problem with division ==> <class 'ZeroDivisionError'> division by zero
End of loop iteration 4
Beginning of loop iteration 5
Problem with list ==> <class 'IndexError'> list assignment index out of range
End of loop iteration 5
Beginning of loop iteration 6
Problem with division ==> <class 'ZeroDivisionError'> division by zero
End of loop iteration 6
Beginning of loop iteration 7

```

```

Problem with value      ==> <class 'ValueError'> invalid literal for int() with base 10: 'Fre
End of loop iteration 7
Beginning of loop iteration 8
Problem with value      ==> <class 'ValueError'> invalid literal for int() with base 10: 'Fre
End of loop iteration 8
Beginning of loop iteration 9
Problem with division   ==> <class 'ZeroDivisionError'> division by zero
End of loop iteration 9

```

Observe how each exception object (e) prints a message that is meaningful for its particular exception.

13.7 The try Statement's Optional else Block

The Python `try` statement supports an optional `else` block. Its behavior is reminiscent of the `while` statement's `else` block (see Section 5.6). If the code within the `try` block does not produce an exception, no `except` blocks trigger, and the program's execution continues with code in the `else` block. Figure 13.2 contrasts the possible program execution flows within a `try/else` statement. The `else` block, if present,

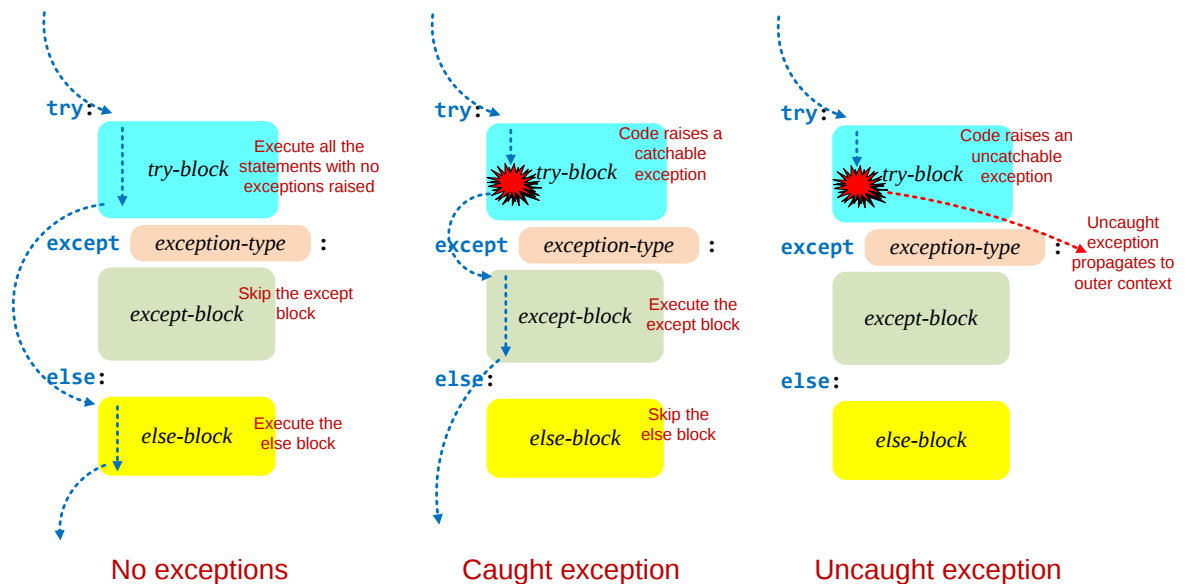


Figure 13.2: The possible program execution flows through a `try/else` statement.

must appear after all of the `except` blocks.

Since the code in the `else` block executes only if the code in the `try` block does not raise an exception, why not just append the code in the `else` block to the end of the code within the `try` block and eliminate the `else` block altogether? The code restructured in this way may not behave identically to the original code. Consider Listing 13.11 (`trynoelse.py`) which demonstrates the different behavior.

Listing 13.11: `trynoelse.py`

```

def fun1():
    try:
        print('try code')
    except:
        print('exception handling code')
    else:
        print('no exception raised code')
        x = int('a') # Raises an exception

def fun2():
    try:
        print('try code')
        print('no exception raised code')
        x = int('a') # Raises an exception
    except:
        print('exception handling code')

print('Calling fun2')
fun2()
print('-----')
print('Calling fun1')
fun1()

```

The fun1 function in Listing 13.11 (trynoelse.py) uses an `else` block, and fun2 moves the `else` code up into the `try` block. The program's output is

```

Calling fun2
try code
no exception raised code
exception handling code
-----
Calling fun1
try code
no exception raised code
Traceback (most recent call last):
  File "trynoelse.py", line 22, in <module>
    fun1()
  File "trynoelse.py", line 8, in fun1
    x = int('a') # Raises an exception
ValueError: invalid literal for int() with base 10: 'a'

```

If the code in the original `else` block can raise an exception, moving it up into the `try` block means that one of the `try` statement's `except` blocks could catch that exception. Leaving the code in the `else` block means that any exception it might raise cannot be caught by one of that `try` statement's `except` blocks.

13.8 finally block

A `try` statement may include an optional `finally` block. Code within a `finally` block always executes whether the `try` block raises an exception or not. A `finally` block usually contains “clean-up code” that must execute due to activity initiated in the `try` block. Figure 13.3 contrasts the possible program execution flows within a `try/except/finally` statement.

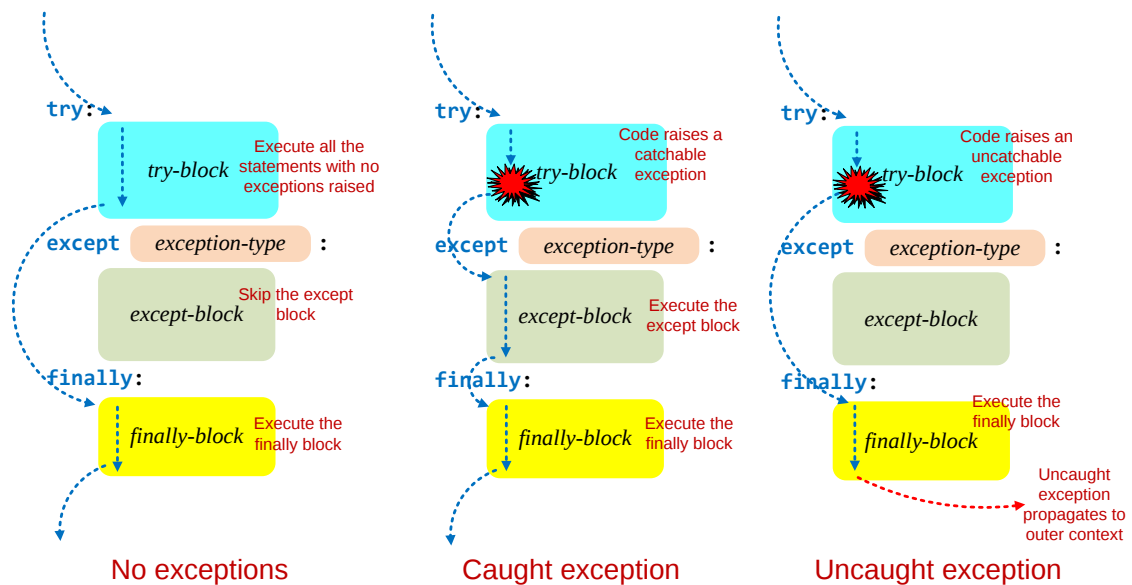


Figure 13.3: The possible program execution flows through a `try/except/finally` statement.

Consider Listing 13.12 (`riskyread.py`) which opens a file and reads its contents.

Listing 13.12: `riskyread.py`

```
# Sum the values in a text file containing numbers
sum = 0
f = open('mydata.dat')
for line in f:
    sum += eval(line)
f.close() # Close the file
print(sum)
```

Listing 13.12 (`riskyread.py`) does not use a `with/as` statement as recommended in Section 9.3, and so it will have a problem should an exception arise before the program executes the `f.close()` statement. The file could be corrupted and one or more of the lines could contain text that is not convertible to a number. Either of these problems would raise an exception before the `f.close` method executes.

Listing 13.13 (`tryfileread.py`) rectifies the problems with Listing 13.12 (`riskyread.py`) by adding exception handling with nested `try` statements.

Listing 13.13: `tryfileread.py`

```
# Sum the values in a text file containing numbers
try:
    f = open('mydata.dat')
except OSError:
    print('Could not open file')
else:
    # File opened properly
```

```
sum = 0
try:
    for line in f:
        sum += eval(line)
    f.close() # Close the file if no exception
except Exception as er:
    print(er) # Show the problem
    f.close() # Close the file if exception
print('sum =', sum)
```

Listing 13.13 (tryfileread.py) uses two `try` statements. The first `try` statement defends against an `OSError` exception. The operating system will prompt the open function to raise such an exception if it cannot satisfy the request; for example, the file may not exist in the current directory or the user may not have sufficient permissions to access the file. The program does not proceed if it cannot open the file for reading.

The code in the outer `try` statement's `else` block contains the interesting part. Once the file is open, more exceptions are possible. In particular, the text file may contain a string that does not evaluate to a number. The inner `try` statement includes a catch-all `except` block to handle any exception that may arise.

If the file `mydata.dat` contains the following:

```
5.5
2.0
6.1
```

Listing 13.13 (tryfileread.py) will print

```
sum = 13.6
```

If the file instead contains the following:

```
5.5
2.0
five
6.1
```

Listing 13.13 (tryfileread.py) will print

```
name 'five' is not defined
sum = 7.5
```

Finally, if no file named `mydata.dat` exists in the current directory, Listing 13.13 (tryfileread.py) will print

```
Could not open file
```

Observe that Listing 13.13 (tryfileread.py) contains two identical statements to close the file:

- If the execution makes it all the way to the end of the inner `try` block, it needs to close the file.
- If an exception arises in the inner `try` block, the exception handler must close the file.

This code duplication is undesirable, so we can use a `finally` block to consolidate the code, as shown in Listing 13.14 (filereadfinally.py).

Listing 13.14: filereadfinally.py

```
# Sum the values in a text file containing numbers
try:
    f = open('mydata.dat')
except OSError:
    print('Could not open file')
else:
    sum = 0
    try:
        for line in f:
            sum += eval(line)
    except Exception as er:
        print(er) # Show the problem
    finally:
        f.close() # Close the file
    print('sum =', sum)
```

Listing 13.14 (filereadfinally.py) behaves exactly like Listing 13.13 (tryfileread.py) contains.

Because of the design of the file class, the `with/as` statement takes care of the details of properly closing a file should an exception arise. The `with/as` statement, however, will not automatically handle any exceptions. We can remedy this with with another pair of nested `try` statements, as Listing 13.15 (betterfileread.py) shows.

Listing 13.15: betterfileread.py

```
# Sum the values in a text file containing numbers
try:
    with open('mydata.dat') as f:
        sum = 0
        try:
            for line in f:
                sum += eval(line)
        except Exception as er:
            print(er) # Show the problem
        print('sum =', sum)
except OSError:
    print('Could not open file')
```

Since Listing 13.15 (betterfileread.py) uses the `with/as` statement, the program does not explicitly call the file object's `close` method. The omission of the `f.close()` statement eliminates the need of the `finally` block.

The `try` keyword cannot appear without at least one of `except` or `finally`. This means the `except` blocks are optional. In fact, the following code fragment:

```
with open('data.dat') as f:    # f is a file object
    for line in f:             # Read each line as text
        print(line.strip())    # Remove trailing newline character
```

is roughly equivalent to

```
f = None    # No file object by default
try:
```

```

f = open('data.dat')    # f is a file object
else:
    for line in f:       # Read each line as text
        print(line.strip()) # Remove trailing newline character
finally:
    if f:
        f.close()       # Close the file, if open

```

Figure 13.4 compares the possible program execution flows within a `try/finally` statement.

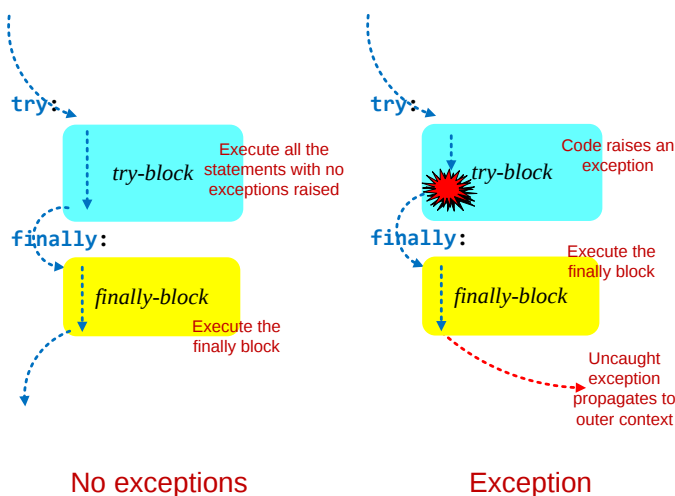


Figure 13.4: The possible program execution flows through a `try/finally` statement.

The `except` and `finally` blocks may not appear without an associated `try` block. An `else` block must be used in the context of a `try` statement (or `if`, `while`, or `for` statement).

As a final note about `finally`: the `finally` block, if present, must appear after all `except` blocks and after the `else` block.

13.9 Exception Handling Scope

Python's exception handling infrastructure is special because it can transcend the usual scoping rules for functions and objects. The exception handling examples we have seen so far have been simple programs where all the code is in the main executing module. The origin of the exception is within the code we can see. These examples have not demonstrated the true power of Python's exceptions. To get a better idea of the scope of exceptions, consider Listing 13.16 (`makeintegerlist.py`).

Listing 13.16: `makeintegerlist.py`

```

def get_int_in_range(low, high):
    ''' Obtains an integer value from the user. Acceptable values

```

```

        must fall within the specified range low...high. '''
    val = int(input())      # Can raise a ValueError
    while val < low or val > high:
        print('Value out of range, please try again:', end=' ')
        val = int(input()) # Can raise a ValueError
    return val

def create_list(n, min, max):
    ''' Allows the user to create a list of n elements consisting
        of integers in the range min...max '''
    result = []
    while n > 0: # Count down to zero
        print('Enter integer in the range {}...{}:'.format(min, max), end=' ')
        result.append(get_int_in_range(min, max))
        n -= 1
    return result

def main():
    ''' Create a list of two elements supplied by the user,
        each element in the range 10...20 '''
    lst = create_list(2, 10, 20)
    print(lst)

if __name__ == '__main__':
    main() # Invoke main

```

Listing 13.16 (makeintegerlist.py) provides two handy functions, `get_int_in_range` and `create_int_list`. The `get_int_in_range` function expects the user to enter an integer value that falls within the range of values specified by its parameters. It ensures the integer the user provides is in the correct range, but if the user enters a non-integer value, the function will raise an exception. The following shows the program's interaction with a well-behaved user:

```

Enter integer in the range 10...20: 12
Enter integer in the range 10...20: 14
[12, 14]

```

Figure 13.5 diagrams the function call sequences of this sample program execution.

The following shows how the program runs with a more creative user:

```

Enter integer in the range 10...20: 9+7
Traceback (most recent call last):
  File "makeintegerlist.py", line 27, in <module>
    main() # Invoke main
  File "makeintegerlist.py", line 23, in main
    lst = create_list(3, 10, 20)
  File "makeintegerlist.py", line 16, in create_list
    result.append(get_int_in_range(min, max))
  File "makeintegerlist.py", line 4, in get_int_in_range
    val = int(input()) # Can raise a ValueError
ValueError: invalid literal for int() with base 10: '9+7'

```

If the `get_int_in_range` function used the `eval` function instead of `int`, it would avoid this exception, but that would allow the function to accept floating-point values. Also, using the `eval` function would not help for the following input:

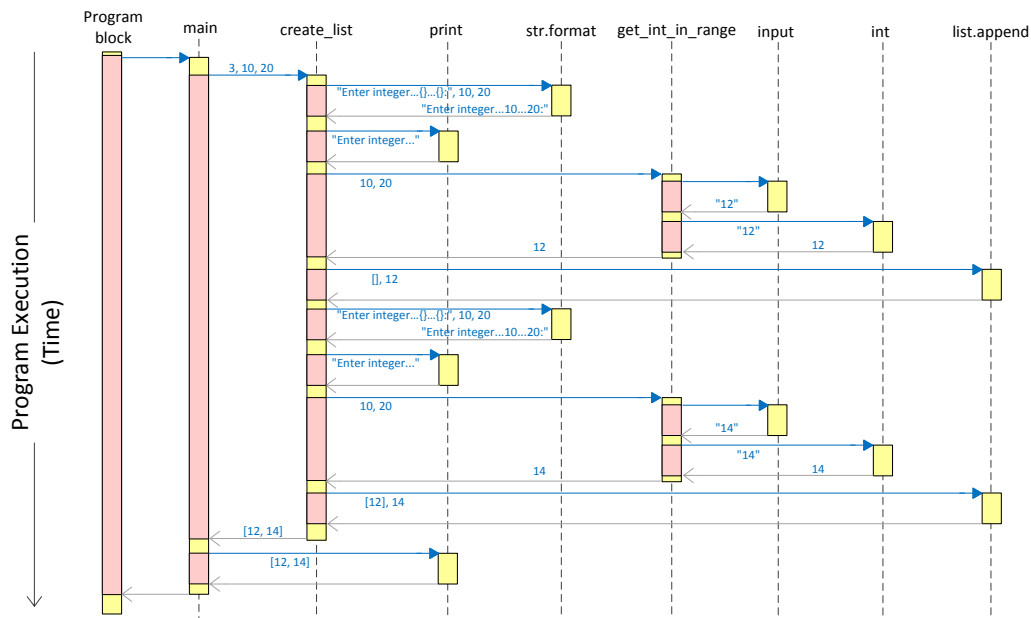


Figure 13.5: Function call sequence diagram for Listing 13.16 (makeintegerlist.py) running with the user providing the input 12 and 14.

```
Enter integer in the range 10...20: eleven
Traceback (most recent call last):
  File "makeintegerlist.py", line 27, in <module>
    main() # Invoke main
  File "makeintegerlist.py", line 23, in main
    lst = create_list(3, 10, 20)
  File "makeintegerlist.py", line 16, in create_list
    result.append(get_int_in_range(min, max))
  File "makeintegerlist.py", line 4, in get_int_in_range
    val = int(input()) # Can raise a ValueError
ValueError: invalid literal for int() with base 10: 'eleven'
```

An uncaught exception produces a *stack trace*. Python uses an area of the computer's memory known as the *stack* to help it control function and method invocations. The stack stores parameters, return values, and the point in the code where the program's execution should return when a function completes. An exception stack trace provides a snapshot of the stack that enables developers to reconstruct the chain of function and/or method calls that produced the exception.

We can read the stack trace from the top down. We see that the program (referenced in the stack trace as `<module>`) called the `main` function at line 27 in the source file `makeintegerlist.py`. In turn, a statement at line 23 in the `main` function invoked the `create_list` function. Code within the `create_list` function at line 16 called the `get_int_in_range` function. Finally, line 4 in the `get_int_in_range` function raised a `ValueError` exception when it called the `int` function with an invalid string literal (the value `'eleven'`).

that the user provided).

We say that an exception “unwinds” the stack, because, if uncaught, an exception will propagate back up the call chain. The propagation stops when it is caught by an exception handler. If the propagation progresses all the way back to the program block level, the interpreter will terminate the program’s execution.

Figure 13.6 provides the function call sequence diagram for case of Listing 13.16 (makeintegerlist.py) raising an exception and thereby terminating the program’s execution. In Listing 13.16 (makeintegerlist.py),

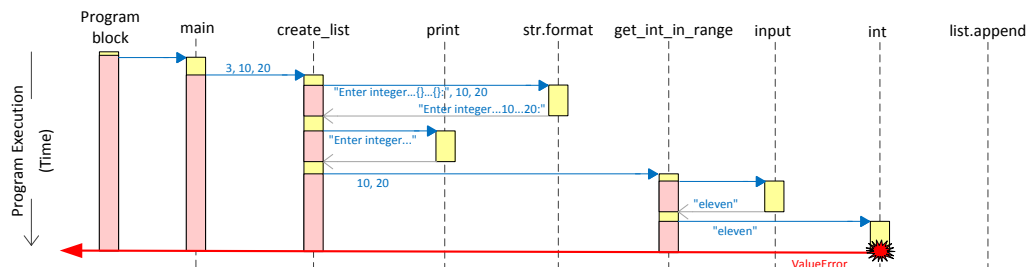


Figure 13.6: Function call sequence diagram for Listing 13.16 (makeintegerlist.py) raising a `ValueError` exception when the user enters *eleven*

`get_int_in_range`’s call to `int` function can raise a `ValueError` exception. To get to this point, the chain of function calls is

Program block → main → create_list → get_int_in_range → int

The red arrow in Figure 13.6 shows that the exception rises immediately up the call chain in the reverse order:

int → get_int_in_range → create_list → main → Program block

Any function in the call chain can catch the exception. In Listing 13.16 (makeintegerlist.py) we know that `int` can raise the exception, but which function should catch the exception? Should we handle the exception close to where it arises or farther up the call chain?

Handling an exception closer to the location that raises it generally gives us more options for corrective action. Consider the following addition of an exception handler to the `get_int_in_range` function:

```
def get_int_in_range(low, high):
    ''' Obtains an integer value from the user. Acceptable values
        must fall within the specified range low...high. '''
    need = True
    while need:
        try:
            val = int(input())          # Can raise a ValueError
            if val < low or val > high:
                print('Value out of range, please try again:', end=' ')
            else:
```

```

        need = False    # No need to continue in loop
    except ValueError:
        print('Value not a valid integer, please try again:', end=' ')
    return val

```

This version of `get_int_in_range` forces the user to enter a viable integer value. The handler is close to the source of the exception, and, therefore, can utilize local information to address the exception.

We could instead keep `get_int_in_range` in its original form and add the exception handler to the `create_list` function. This moves the handler one function up the call chain. The following is one possible rewrite of `create_list`:

```

def create_list(n, min, max):
    ''' Allows the user to create a list of n elements consisting
        of integers in the range min...max '''
    result = []
    while n > 0:    # Count down to zero
        try:
            print('Enter integer in the range {}...{}:'.format(min, max), end=' ')
            result.append(get_int_in_range(min, max))
            n -= 1
        except:
            print('Unable to add the value to the list')
    return result

```

Notice the catch-all `except` block rather than a `ValueError`-specific handler. This is because if we catch only the `ValueError` exception potentially raised by `get_int_in_range`, we may miss a different exception. Consider Listing 13.17 (`appendfail.py`) that attempts to add one billion references of the integer zero to a list.

Listing 13.17: `appendfail.py`

```

lst = []
for i in range(1000000000):
    lst.append(0)
print(len(lst))

```

On a 32-bit Windows system Listing 13.17 (`appendfail.py`) reports:

```

Traceback (most recent call last):
  File "appendfail.py", line 3, in <module>
    lst.append(0)
MemoryError

```

The number of elements in a list is limited by operating systems constraints. The `append` method of the `list` class will raise a `MemoryError` exception if an executing program attempts to exceed this limit. Listing 13.18 (`appendexcept.py`) adds an exception handler to provide more insight.

Listing 13.18: `appendexcept.py`

```

lst = []
try:
    for i in range(1000000000):

```

```

        lst.append(0)
except MemoryError:
    print('List too big')
print(len(lst))

```

On the same system as above, the program prints

```

List too big
107465551

```

It is conceivable that the first parameter sent to `create_list` could exceed the list length limit (although pity the poor user who would have to type in over 100,000,000 values only to be faced with a program that cannot continue!). The code within the `try` block of our revised `create_list` function has the potential to produce two different types of exceptions. The exception handler in `create_list` does not have access to the local information in the `get_int_in_range` function, so the corrective action must be more general to accommodate more reasons for the error.

The exception handler we added to `create_list` has a problem. It is structured so that if it catches an exception, the user can continue to try to enter in the expected number of values. In the unlikely event that the `append` method raises the `MemoryError` exception, the list can hold no more values, so the program will enter an infinite loop. The following version of `create_list` adds an extra `except` block to remedy this situation:

```

def create_list(n, min, max):
    ''' Allows the user to create a list of n elements consisting
        of integers in the range min...max '''
    result = []
    while n > 0:    # Count down to zero
        try:
            print('Enter integer in the range {}...{}:'.format(min, max), end=' ')
            result.append(get_int_in_range(min, max))
            n -= 1
        except ValueError:
            print('Value not an integer')
        except:
            print('Unable to add the value to the list')
            break    # List can hold no more, exit the loop
    return result

```

This version of `create_list` allows the user to continue building the list unless the `append` method fails.

We can go one step farther up the call chain to the main function:

```

def main():
    ''' Create a list of three elements supplied by the user,
        each element in the range 10...20 '''
    try:
        lst = create_list(3, 10, 20)
        print(lst)
    except:
        print('Cannot create a list')

```

The main function's reaction to the exception is even more general than that of `create_list`. Since `create_list` contains significant logic, `main` is powerless to attempt corrective action. Note that `main`

cannot simply add a loop to allow the user to continue trying to add numbers each time the code in the `try` block raises an exception; doing so would call `create_list` again, and, since `create_list` begins with the empty list, calling it again would lose any values the user entered before the exception appeared. The best main can do is report the program's inability to create the list.

Since the main function basically just wraps the call to `create_list`, we would not gain very much generality moving up to the program block level:

```
if __name__ == '__main__':
    try:
        main()      # Invoke main
    except:
        print('Cannot create a list')
```

In general, exception handlers located closer to the code originating the exception have access to more local information that is useful for implementing more focused corrective action. As the exception is uncaught and propagates up the function-call chain, this local information is lost, and corrective action, if any, becomes less focused and more generic.

13.10 Raising Exceptions

CAUTION! SECTION UNDER CONSTRUCTION

We have seen how to write code that reacts to exceptions. We know that certain functions, like `open` and `int` can raise exceptions. Also, statements that attempt to divide by zero or print an undefined variable will raise exceptions. The following statement raises a `ValueError` exception:

```
x = int('x')
```

The following statement is a more direct way to raise a `ValueError` exception:

```
raise ValueError()
```

The `raise` keyword raises an exception. Its argument, if present, must be an exception object. The class constructor for most exception objects accepts a string parameter that provides additional information to handlers:

```
raise ValueError('45')
```

In the interactive shell:

```
>>> raise ValueError()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError
>>> raise ValueError('45')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 45
>>> raise ValueError('This is a value error')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: This is a value error
```

This means we can write a custom version of the `int` function that behaves similar to the built-in `int` function, as Listing 13.19 (`nonnegconvert.py`) illustrates.

Listing 13.19: `nonnegconvert.py`

```
def non_neg_int(n):
    result = int(n)
    if result < 0:
        raise ValueError(result)
    return result

while True:
    x = non_neg_int(input('Please enter a non-negative integer:'))
    if x == 999: # Secret number exits loop
        break
    print('You entered', x)
```

The following shows a sample run of Listing 13.19 (`nonnegconvert.py`):

```
Please enter a non-negative integer:3
You entered 3
Please enter a non-negative integer:-3
Traceback (most recent call last):
  File "nonnegconvert.py", line 8, in <module>
    x = nonnegint(input('Please enter a non-negative integer:'))
  File "nonnegconvert.py", line 4, in nonnegint
    raise ValueError(result)
ValueError: -3
```

The built-in `int` function accepts well-formed negative integers with no problem, but our `non_neg_int` function is more selective. It does not accept valid negative integers; hence, it raises a value error. Listing 13.20 (`nonnegexcept.py`) catches the exception so the program does not crash.

Listing 13.20: `nonnegexcept.py`

```
def non_neg_int(n):
    ''' Converts argument n into a non-negative integer, if possible.
        Raises a ValueError if the argument is not convertible
        to a non-negative integer. '''
    result = int(n)
    if result < 0:
        raise ValueError(result)
    return result

while True:
    try:
        x = non_neg_int(input('Please enter a non-negative integer:'))
        if x == 999: # Secret number exits loop
            break
        print('You entered', x)
    except:
        print('The value you entered is not acceptable')
```

The following shows a sample run of Listing 13.20 (`nonnegexcept.py`):

```

Please enter a non-negative integer:3
You entered 3
Please enter a non-negative integer:-3
The value you entered is not acceptable
Please enter a non-negative integer:5
You entered 5
Please enter a non-negative integer:999

```

If one Python’s built-in exception types is not appropriate to describe the exception you need to raise, you can use the generic `Exception` class and provide a descriptive message to its constructor, as in

```
raise Exception('Cannot add non-integer to restricted list')
```

If raised and uncaught, the interpreter will print the following line at the end of the stack trace:

```
Exception: Cannot add non-integer to restricted list
```

In Chapter 14.6 we will see a better way to customize exceptions by designing our own custom exception classes that integrate seamlessly with Python’s exception handling infrastructure.

Sometimes it is appropriate for a function (or method) to catch an exception, take some action appropriate to its local context, and then re-raise the same exception so that the function’s caller can take further action if necessary. In essence, the function that catches the exception first administers “first aid” and then passes the exception up the call chain for more advanced, application-specific treatment and care. In Listing 13.21 (`reraise.py`), the `count_elements` function accepts a list, `lst`, presumed to contain only integers, and a Boolean function predicate. The predicate function parameter accepts a single argument and returns true or false based on whether or not its argument parameter has a certain property. The program calls `count_elements` with the `is_prime` function (is the argument a prime number?) and the `non_neg` (is the argument a non-negative integer). Both `is_prime` and `non_neg` can raise a `TypeError` exception.

Listing 13.21: `reraise.py`

```

def is_prime(n):
    ''' Returns True if non-negative integer n is prime;
        otherwise, returns false.
        Raises a TypeError exception if n is not
        an integer. '''
    from math import sqrt
    if n == 2:
        return True
    if n < 2 or n % 2 == 0:
        return False
    trial_factor = 3
    root = sqrt(n) + 1
    while trial_factor <= root:
        if n % trial_factor == 0:
            return False
        trial_factor += 2
    return True

def non_neg(n):
    ''' Determines if n is non-negative.
        Raises a ValueError if n is not an integer. '''
    return n >= 0

```

```

def count_elements(lst, predicate):
    ''' Counts the number of integers in list lst that are
        acceptable to a given predicate (Boolean function). '''
    count = 0
    for x in lst:
        try:
            if predicate(x):
                count += 1
        except:
            print(x, 'is a not an acceptable element')
            raise # Re-raise the exception
    return count

print(count_elements([3, -71, 22, -19, 2, 9], non_neg))
print(count_elements([2, 3, 4, 5, 6, 8, 9], is_prime))
print(count_elements([2, 4, '6', 8, 'x', 7], is_prime))

```

The `count_elements` function uses a `try` statement to defend against the possible exceptions raised by `is_prime` and `non_neg`. If it catches any type of exception, it prints a diagnostic message alerting the user that it found a non-integer element in the list. Neither the `is_prime` nor the `non_neg` function can provide such a specific error message. After catching the `TypeError` exception and reporting the problem, the `count_elements` function re-raises the same exception for the benefit of its caller.

Listing 13.21 (`reraise.py`) prints the following:

```

4
3
6 is a not an acceptable element
Traceback (most recent call last):
  File "countprimes.py", line 42, in <module>
    print(count_elements([2, 4, '6', 8, 'x', 7], lambda x: x % 10 == 0))
  File "countprimes.py", line 32, in count_elements
    if predicate(x):
  File "countprimes.py", line 42, in <lambda>
    print(count_elements([2, 4, '6', 8, 'x', 7], lambda x: x % 10 == 0))
TypeError: not all arguments converted during string formatting

```

The caller has information about its context that the called function would not have.

Re-raising an exception. Catch the exceptions you expect and re-raise or propagate up the exceptions you do not expect.

13.11 Using Exceptions

CAUTION! SECTION UNDER CONSTRUCTION

HERE

Exceptions should be reserved for uncommon errors. For example, the following code adds up all the elements in a list of numbers named `lst`:

```
sum = 0
for elem in lst:
    sum += elem
print("Sum =", sum)
```

This loop is fairly typical. Another approach uses exceptions:

```
sum = 0
int i = 0
try:
    while True:
        sum += lst[i]
        i += 1
except IndexError:
    pass
print("Sum =", sum)
```

Here, an `IndexError` exception interrupts the loop when the list access is out of bounds. The exception interrupts the statement

```
sum += lst[i]
```

in midstream, before the expression on the right side of the `+=` operator is fully evaluated. The result is that `sum`'s value is not incorrectly incremented.

Both approaches compute the same result. However, the second approach *always* raises and handles an exception. The exception definitely is **not** an uncommon occurrence.

Exceptions should not be used to dictate normal logical flow. While very useful for its intended purpose, the exception mechanism adds some overhead to program execution, especially when an exception is raised. This overhead is reasonable when exceptions are rare but not when exceptions are part of the program's normal execution.

Exceptions are valuable aids for careless or novice programmers. A careful programmer ensures that code accessing a list does not exceed the list's bounds. Another programmer's code may accidentally attempt to access `a[len(a)]`. A novice may believe `a[len(a)]` is a valid element. Since no programmer is perfect, exceptions provide a nice safety net.

As you develop more sophisticated programs you will find exceptions more compelling. You should analyze your code carefully to determine its limitations. Exceptions can be valuable for covering these limitations. The Python standard library uses exceptions extensively, so programs that make use of library functions and classes should properly handle the exceptions they can raise.

13.12 Summary

CAUTION! SECTION UNDER CONSTRUCTION

- Add summary items

13.13 Exercises

CAUTION! SECTION UNDER CONSTRUCTION

1. Add exercises

Chapter 14

Custom Types

We have examined many of Python’s built-in types. Some, like, integers, floating-point numbers, and Booleans are relatively simple, while others such as lists, tuples, dictionaries, sets, and exceptions are more complex. Python’s rich collection of built-in types enable us to write a wide variety of programs in diverse problem domains. Python also provides the ability for programmers to design their own custom types by which developers can craft data types that more closely model the problem at hand. This better alignment of software assets with the problem domain can expedite the development process.

14.1 Circle Objects

CAUTION! SECTION UNDER CONSTRUCTION

An object generally bundles data (fields) and functionality (methods). The fields and methods of an object comprise its members. The class of an object defines the object’s basic structure. As a simple concrete example, consider the familiar geometric circle, shown in Figure 14.1. Given a circle’s radius (r

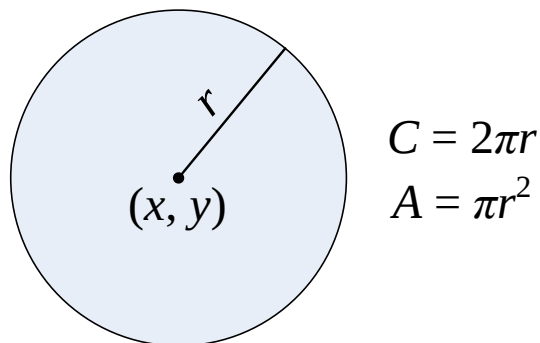


Figure 14.1: A circle with radius r centered at (x, y) . A represents the circle’s area, and C is its circumference

in Figure 14.1), we can compute the circle's area and circumference. The circle's center, (x,y) , establishes the circle's position. We will define a custom `Circle` class in Python from which we can create `Circle` instances (objects). The `Circle` class specifies what circle objects can do and how clients (other code) can interact with them.

What data must each `Circle` object maintain? Since circles can appear in various places, each circle has its own position. Just like in mathematics, we can specify the location of a `Circle` object by its (x,y) center. Also, since some circles are larger or smaller than others, each `Circle` should have its own radius. It is natural, then, for our `Circle` object to have a center field and a radius field.

Should our circle objects have an area field and/or a circumference field? Both area and circumference depend solely on a circle's radius, so if we include a radius field, both area and circumference would be redundant information. Besides, we easily can compute them as needed with simple formulas. We will implement area and circumference as methods in our `Circle` class.

As we finalize the design for our `Circle` class, we will add a few interesting points:

- Clients should be able to create a `Circle` object with a specified center point (a tuple of two numbers) and radius.
- An attempt to create a circle with a negative radius results in a `ValueError` exception.
- Clients can determine a `Circle` object's radius via a `get_radius` method.
- Clients can determine a `Circle` object's center via a `get_center` method.
- Clients can reposition the circle via a `move` method.
- Clients can increase the circle's radius by one via a `grow` method.
- Clients can decrease the circle's radius by one via a `shrink` method. At no time should the circle's radius fall below zero.

Our intention is to discourage clients from changing the fields (center and radius) of `Circle` objects directly; thus, we offer the `move`, `grow`, and `shrink` methods.

Based on our criteria, we will need the following methods in our `Circle` class:

- `__init__`: The name of all constructors in Python classes is `__init__`. Our `__init__` method must create and initialize the center and radius fields, and it must detect an attempt to make a `Circle` object with a negative radius. The client code must supply a center (a tuple consisting of two numbers) and a radius.
- `get_radius`: This method simply needs to return the value of the radius field. This method accepts no parameters.
- `get_center`: This method simply needs to return the value of the center field. This method accepts no parameters.
- `get_area`: This method needs to compute and return the area. This method accepts no parameters.
- `get_circumference`: This method needs to compute and return the circumference. This method accepts no parameters.
- `move`: This method repositions the `Circle` object's center. The client must provide a tuple consisting of two numbers. This tuple represents the new coordinates of the object's center.

- `grow`: This method increases the `Circle` object's radius by one unit. This method accepts no parameters.
- `shrink`: If the `Circle` object's radius is greater than zero, this method decreases its radius by one unit. This method does not change the radius if the radius is zero before the call. This method accepts no parameters.

Armed with this more detailed specification, we are ready to define our `Circle` class. Listing 14.1 (`circle.py`) contains the complete definition for `Circle`.

Listing 14.1: `circle.py`

```
class Circle:
    ''' Represents a geometric circle object '''
    def __init__(self, center, radius):
        ''' Initialize the center's center and radius '''
        # Disallow a negative radius
        if radius < 0:
            raise ValueError('Negative radius')
        self.center = center
        self.radius = radius

    def get_radius(self):
        ''' Return the radius of the circle '''
        return self.radius

    def get_center(self):
        ''' Return the coordinates of the center '''
        return self.center

    def get_area(self):
        ''' Compute and return the area of the circle '''
        from math import pi
        return pi*self.radius*self.radius

    def get_circumference(self):
        ''' Compute and return the circumference of the circle '''
        from math import pi
        return 2*pi*self.radius

    def move(self, pt):
        ''' Moves the center of the circle to point pt '''
        self.center = pt

    def grow(self):
        ''' Increases the radius of the circle '''
        self.radius += 1

    def shrink(self):
        ''' Decreases the radius of the circle;
            does not affect a circle with radius zero '''
        if self.radius > 0:
            self.radius -= 1
```

We see in Listing 14.1 (`circle.py`) that a class definition begins with the reserved word `class` followed by the name of the class and a colon (`:`) at the end of the line. As in function definitions, the body of the class is indented. The class body looks like a series of function definitions; however, since they appear within a class definition they are method definitions. As with functions, we can (and should) document classes and methods with docstrings.

Notice that each method definition has `self` for its first parameter. The language does not require the parameter's name to be `self` (it could be `x`, `obj`, or any valid identifier), but the universal convention in the Python programming world is to use the name `self`. The `self` parameter references the object on whose behalf the method is being invoked. In Turtle graphics, for example, the `Turtle` class definition contains a `forward` method definition that begins

```
def forward(self, distance):  
    # Details omitted . . .
```

If `t` is a reference to a `Turtle` object, we can move the turtle forward 100 pixels with the call

```
t.forward(100)
```

This method invocation assigns the actual parameter `t` to the formal parameter `self` and the actual parameter 100 to the formal parameter `distance`. In fact, we can rewrite the call as

```
Turtle.forward(t, 100)
```

In this way can see exactly that `t` corresponds to `self` and 100 corresponds to `distance`.

In the case of our `Circle` class, if `circ` refers to a `Circle` object, the call

```
circ.grow()
```

is equivalent to

```
Circle.grow(circ)
```

In the `Circle` class definition, the `__init__` method initializes a `Circle` object. The statement

```
circ = Circle((10, 3.4), 5)
```

creates a new `Circle` object with a center at (10,3.4) and radius 5. It invokes the `__init__` method to do the initialization work. The statement then assigns the variable `circ` to this new `Circle` object. In the definition of `__init__` we prefix field names with `self`. Any variables not prefixed with `self` are treated as normal local or global variables. In the following two statements in the constructor:

```
self.center = center  
self.radius = radius
```

`self.center` refers to the `center` field of the object under construction, and `center` on the right side of the assignment operator refers to the formal parameter. Similarly, `self.radius` refers to the `radius` field of the object, and `radius` on the right side is the parameter.

The methods `get_center` and `get_radius` are sometimes called *accessor methods*, or *getters*, as they give clients access to see the state of an object. In this case, clients can obtain a `Circle` object's center and radius via these methods. In contrast, the methods `move`, `grow`, and `shrink` are known as *mutator methods*, or *setters*, because they allow clients to modify the state of an object. Note that `grow` and `shrink` do not allow arbitrary changes; they allow clients to adjust a circle's radius only by one-unit increments.

The methods `get_area` and `get_circumference` are neither accessors nor mutators. They do not provide direct access to the data in a `Circle` object but rather provide indirect access via a computation (you could reverse engineer the result of either method to deduce the radius, but the `get_radius` method provides direct access).

Figure 14.2 provides a conceptual view of a `Circle` object. Note that each `Circle` object needs only

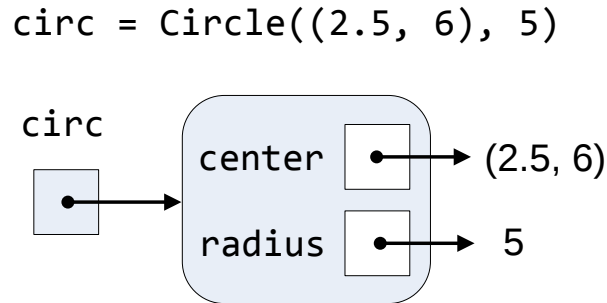


Figure 14.2: A conceptual illustration of a circle object. Each instance of the `Circle` class has its own center and radius fields.

store a reference to its fields. All `Circle` instances share the code for their methods. In the following code:

```
c1 = Circle((2, 4), 5)
c2 = Circle((0, 0), 1)
```

The objects referred by `c1` and `c2` have their own center and radius fields. The call

```
print(c1.get_radius())
```

passes a reference to `c1`'s object as the first (`self`) parameter to the method `get_radius`, while

```
print(c2.get_radius())
```

passes a reference to `c2`'s object as the first (`self`) parameter to the `get_radius` method. As you can see, there is no need for each object to have its own copy of the method code; however, every distinct object must maintain its own copies of its fields.

Listing 14.2 (`circlemaker.py`) exercises our new `Circle` class. It uses Turtle graphics (Section 9.5) to render `Circle` objects in a graphical window.

Listing 14.2: `circlemaker.py`

```
from turtle import Turtle, Screen, mainloop, delay, clear
from circle import Circle

t = Turtle()                # Global turtle
circ = Circle((0, 0), 100)  # Global circle object

def draw_circle(t, circle):
    x, y = circle.get_center() # Unpack center's coordinates
```

```

    radius = circle.get_radius()
    t.penup()                # Lift pen
    t.setposition(x, y)      # Move pen to (x,y)
    t.pendown()             # Place pen
    t.dot()                 # Draw a dot at the circle's center
    t.penup()              # Lift pen
    t.setposition(x, y - radius) # Position pen to draw rim of circle
    t.pendown()            # Place pen to draw
    t.circle(radius)        # Draw the circle
    t.penup()              # Lift pen

def do_click(x, y):
    circ.move((x, y))      # Move the circle to a new location
    redraw()

def do_up():
    circ.grow()            # Make the circle bigger
    redraw()

def do_down():
    circ.shrink()          # Make the circle smaller
    redraw()

def redraw():
    t.clear()              # Clear the drawing screen
    draw_circle(t, circ)   # Draw the circle object

def main():
    delay(0)              # Do not slowly trace drawing
    t.hideturtle()         # Make the turtle invisible
    screen = Screen()      # Create Screen object to receive user input
    screen.listen()        # Set focus for keystrokes
    screen.onclick(do_click) # Set mouse press handler
    screen.onkey(do_up, 'Up') # Set "up" cursor key handler
    screen.onkey(do_down, 'Down') # Set "down" cursor key handler
    mainloop()

if __name__ == '__main__':
    main()

```

Listing 14.2 (circlemaker.py) is an interactive program that allows the user to place the image of a Circle object in the window via a mouse click. Subsequent mouse clicks move the circle to the mouse position. Users can use the up  and down  cursor keys to enlarge and reduce the size of the circle.

The turtle module provides the Screen class. In Listing 14.2 (circlemaker.py), the screen variable references a Screen object. Screen objects accept user input from the pointing device and keyboard. The statement

```
screen = Screen()
```

creates the Screen object and assigns it to screen. The statement

```
screen.listen()
```

enables the graphical window to receive keystrokes from the user. The statement

```
screen.onclick(do_click)
```

calls the `onclick` method of the `Screen` class. Like functions (see Section 8.6), methods can accept functions and other methods as parameters. The `onclick` method accepts a function as a parameter. The `turtle` module provides the `Screen.onclick` method, and we must provide the function we send to it. The only restriction on the function we send to `onclick` is that the function must accept two integer parameters. In this case we pass it the `do_click` function. The purpose of the `onclick` function is to register a *callback function* with the Turtle graphics framework. The framework will “call back” the function when a certain event occurs. If the user clicks the mouse when the pointer is over the graphics window, the framework will call the function registered with `onclick` (in this case `do_click`) and pass to it the x and y coordinates of the mouse pointer at the time of the click. That is why this callback function requires a function that expects the two integer parameters. Note that our `do_click` function packs up into a tuple the x and y coordinates it receives and passes this tuple on to the global `Circle` object `circ`’s `move` method. We must use a global variable in the `do_click` function because we cannot pass in the `circ` object—remember, we do not call `do_click`, the framework does, and the framework only passes in the mouse pointer coordinates.

Similarly, the statements

```
screen.onkey(do_up, 'Up')
screen.onkey(do_down, 'Down')
```

assign callback functions to invoke when the user presses the up  and down  cursor keys. The callback functions accepted by the `Screen.onkey` method take no parameters. Consequently, neither `do_up` nor `do_down` accept parameters. The second parameter to the `onkey` method specifies which key event to associate with the given callback function.

Note that the `do_click`, `do_up`, and `do_down` functions modify the global `Circle` object `circ`’s state, but the `draw_circle` method simply uses the `Circle` object’s `get_center` and `get_radius` methods to be able to draw the circle using the Turtle object’s `circle` method. The `Turtle.circle` method draws a circle starting from the turtle’s current position—three o’clock on the circle—and it draws a counter-clockwise curve around the circle.

Figure 14.3 shows a sample run of Listing 14.2 (`circlemaker.py`).

14.2 Restricting Access to Members

The intention of the `Circle` class is that clients should not directly manipulate a `Circle` object’s center and radius fields. The constructor (`__init__`), `move`, `grow`, and `shrink` are the only methods that can influence the state of a `Circle` object; however, client code can access the fields directly:

```
>>> from circle import *
>>> c = Circle((0, 0), 4)
>>> c.get_radius()
4
>>> c.radius = 10
>>> c.get_radius()
10
```

As we can see, clients can alter the radius of a circle outside the facilities provided by the `grow` and `shrink` methods.

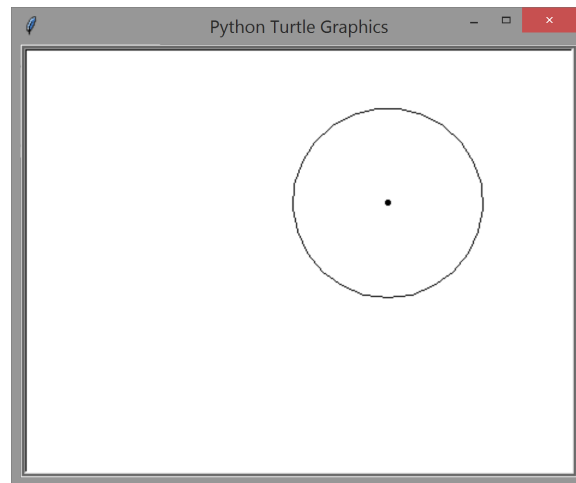


Figure 14.3: Using the custom Circle class in a graphical application

The convention in Python is to use a leading underscore to name fields not meant for general access by clients:

```
class MyClass:
    def __init__(self, num):
        self.public_var = num
        self._private_var = num
```

Here, clients are encouraged to access the `public_var` field of `MyClass` objects directly:

```
mc = MyClass(10)
mc.public_var = 12    # Acceptable access
```

but code like the following is discouraged:

```
mc = MyClass(10)
mc._private_var = 12  # Modifying a field not meant for public access
```

The statement that modifies the intended private field is not illegal. It is understood that a programmer doing so assumes all the risk of any problems that may arise from the action.

Python does support a naming scheme that makes it more difficult to access a field or method, in effect rendering it private to code outside the class. Python does not permit normal access with the dot (`.`) to members of an object with names that begin with two underscores. (The Python community often tersely refers to *double underscore* by the neologism *dunder*.) The following interactive sequence illustrates:

```
>>> class MyType:
...     def __init__(self):
...         self.__id = 2
...
>>> m = MyType()
>>> m.__id
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
AttributeError: 'MyType' object has no attribute '__id'
>>> m._MyType__id
2
```

As you can see, internally Python changes the name of a class member with a name that begins with double underscores by prefixing it with a single underscore followed by the class name. Python thus does not provide a way to truly protect object members from the outside world.

14.3 Bank Account Objects

Consider the task of writing a program that manages accounts for a bank. Real bank accounts must store a large amount of information: the account owner's name, address, social security number, account number, etc. To streamline our example, we will model bank accounts that maintain just three attributes:

- Every account has a unique identifier, the account number.
- Each account's owner has a name.
- Each account has a current balance.

We can define a class of bank account objects. Each account object would have an account number field (integer), a name field (string), and a balance field (integer number of cents). The bank account management application could store the accounts in a list and move the contents of the list to a file for persistent storage.

Listing 14.3 (bankaccounts.py) uses a `BankAccount` class to implement a simple database of customer bank accounts.

Listing 14.3: bankaccounts.py

```
class BankAccount:
    ''' Models a bank account '''
    def __init__(self, number, name, balance):
        ''' Initialize the fields of a bank account object.
            Disallows a negative initial balance. '''
        if balance < 0:
            raise ValueError('Negative initial balance')
        self.__account_number = number # Account number
        self.__name = name             # Customer name
        self.__balance = balance       # Funds available in the account

    def deposit(self, amount):
        ''' Add funds to the account. There is no limit
            to the size of the deposit. '''
        self.__balance += amount

    def withdraw(self, amount):
        ''' Remove funds from the account, if possible.
            Only completes the withdrawal successfully if
            there are enough funds in the account to
            fulfill the withdrawal.
            Return true if successful, false otherwise '''
        result = False # Unsuccessful by default
```

```

        if self.__balance - amount >= 0:
            self.__balance -= amount
            result = True # Success
        return result

    def __str__(self):
        ''' Returns the string representation for this object '''
        #return '[' + str(self.__account_number) + ', ' + str(
        #    self.__name) + ', ' + str(
        #    self.__balance) + ']'
        return '[{:>5}, {:<10} {:>7}]'.format(self.__account_number,
                                             self.__name, self.__balance)

# Global functions that use BankAccount objects

def open_database(filename, db):
    ''' Read account information from a given file and store it
        in the given list. '''
    with open(filename) as lines: # Open file to read
        for line in lines:
            line.strip()
            number, name, balance = eval(line)
            db.append(BankAccount(int(number), name, int(balance)))

def print_database(db):
    ''' Display the contents of the database '''
    for acct in db:
        print(acct) # Calls the __str__ method of acct

def main():
    # Simple bank account "database"
    database = []

    try:
        open_database('accountdata.text', database)
        print_database(database)
    except:
        print('Error in account database')

main()

```

Given a text file named accountdata.text containing the data

```

324, 'Fred',    34523
371, 'Wilma',  1263210
129, 'Betty',   78934
120, 'Barney',  247702
412, 'Pebbles', 12000
420, 'Bam-Bam', 10354
1038, 'George', 6733498

```

```
966, 'Jane',    9923912
1210, 'Judy',    83497
1300, 'Elroy',   50315
```

the program Listing 14.3 (bankaccounts.py) would produce the output

```
[ 324, Fred      34523 ]
[ 371, Wilma     1263210 ]
[ 129, Betty      78934 ]
[ 120, Barney     247702 ]
[ 412, Pebbles    12000 ]
[ 420, Bam-Bam    10354 ]
[ 1038, George    6733498 ]
[ 966, Jane       9923912 ]
[ 1210, Judy      83497 ]
[ 1300, Elroy     50315 ]
```

Clients interact with these bank account objects via the methods; thus, it is only through methods that clients may alter the state of a bank account object.

In the BankAccount methods

- The constructor (`__init__`) initializes the fields of a BankAccount object, but it disallows the creation of an object with a negative account balance.
- Clients may add funds via the `deposit` method. The `deposit` method does limit the amount of a deposit.
- The `withdraw` method prevents a client from withdrawing more money from an account than some specified minimum value. Withdrawals are not possible from an inactive account.

The following code will not work:

```
acct = BankAccount(31243, 'Joe', 1000)
acct.deposit(100)
acct.__balance -= 100    # Illegal
```

Clients instead must use the `withdraw` method. The `withdraw` method prevents actions such as

```
# New bank account object with $1,000.00 balance
acct = BankAccount(31243, 'Joe', 1000)
acct.withdraw(2000.00); # Method should disallow this operation
```

The operations of depositing and withdrawing funds are the responsibility of the object itself, not the client code. The attempt to withdraw the \$2,000 dollars will not alter the state of the object.

Consider a non-programming example. If I deposit \$1,000.00 dollars into a bank, the bank then has custody of my money. It is still my money, so I theoretically can reclaim it at any time. The bank stores money in its safe, and my money is in the safe as well. Suppose I wish to withdraw \$100 dollars from my account. Since I have \$1,000 total in my account, the transaction should be no problem. What is wrong with the following scenario:

1. Enter the bank.
2. Walk past the teller into a back room that provides access to the safe.

3. The door to the safe is open, so enter the safe and remove \$100 from a stack of \$20 bills.
4. Exit the safe and inform a teller that you got \$100 out of your account.
5. Leave the bank.

This is not the process a normal bank uses to handle withdrawals. In a perfect world where everyone is honest and makes no mistakes, all is well. In reality, many customers might be dishonest and intentionally take more money than they report. Even though I faithfully counted out my funds, perhaps some of the bills were stuck to each other and I made an honest mistake by picking up six \$20 bills instead of five. If I place the bills in my wallet with other money that already be present, I may never detect the error. Clearly a bank needs more controlled procedure for customer withdrawals.

When working with programming objects, in many situations it is advantageous to discourage client access to the internals of an object. Client code should not be able to easily modify directly bank account fields for various reasons, including:

- A withdrawal should not exceed the account balance.
- Federal laws dictate that deposits above a certain amount should be reported to the Internal Revenue Service, so a bank would not want customers to be able to add funds to an account in a way to circumvent this process. We could add this functionality to our `deposit` method.
- An account number should never change for a given account for the life of that account.

We know that naming the fields with a dunder prefix provides only mild security (Section 14.2). One check we can perform is to search client code for dunders. Behaving client code should be dunder-free.

14.4 Stopwatch Objects

In 6.3 we saw how to use the `clock` function to measure elapsed time during a program's execution. The following skeleton code fragment

```
seconds = clock()    # Record starting time
#
# Do something here that you wish to time
#
other = clock()      # Record ending time
print(other - seconds, "seconds")
```

can be adapted to any program, but we can make it more convenient if we wrap the functionality into an object. We can wrap all the messy details of the timing code into a convenient package. Consider the following client code that uses an object to keep track of the time:

```
timer = Stopwatch()  # Declare a stopwatch object

timer.start()        # Start timing

#
# Do something here that you wish to time
#
```

```
timer.stop()    # Stop the clock
print(timer.elapsed(), " seconds")
```

This code using a Stopwatch object is simpler. A programmer writes code using a Stopwatch in a similar way to using an actual stopwatch: push a button to start the clock (call the `start` method), push a button to stop the clock (call the `stop` method), and then read the elapsed time (use the result of the `elapsed` method). Programmers using a Stopwatch object in their code are much less likely to make a mistake because the details that make it work are hidden and inaccessible.

Given our experience designing our own types though Python classes, we now are adequately equipped to implement such a Stopwatch class. Listing 14.4 (`stopwatch.py`) defines the structure and capabilities of our Stopwatch objects.

Listing 14.4: stopwatch.py

```
from time import clock

class Stopwatch:
    ''' Provides stopwatch objects that that programmers
        can use to time the execution time of portions of
        a program. '''
    def __init__(self):
        ''' Makes a new stopwatch ready for timing. '''
        self.reset()

    def start(self):
        ''' Starts the stopwatch, unless it is already running.
            This method does not affect any time that may have
            already accumulated on the stopwatch. '''
        if not self.__running:
            self.__start_time = clock()
            self.__running = True # Clock now running
        else:
            print("Stopwatch already running")

    def stop(self):
        ''' Stops the stopwatch, unless it is not running.
            Updates the accumulated elapsed time. '''
        if self.__running:
            self.__elapsed += clock() - self.__start_time
            self.__running = False # Clock stopped
        else:
            print("Stopwatch not running")

    def reset(self):
        ''' Resets stopwatch to zero. '''
        self.__start_time = self.__elapsed = 0
        self.__running = False

    def elapsed(self):
        ''' Reveals the elapsed time since the stopwatch
            was last reset. '''
        if not self.__running:
            return self.__elapsed
        else:
```

```
print("Stopwatch must be stopped")
return None
```

Four methods are available to clients: `start`, `stop`, `reset`, and `elapsed`. A client does not have to worry about the “messy” detail of the arithmetic to compute the elapsed time.

Note that our design forces clients to stop a Stopwatch object before calling the `elapsed` method. Failure to do so results in a programmer-defined run-time error report. A variation on this design might allow a client to read the elapsed time without stopping the watch. This implementation allows a user to stop the stopwatch and resume the timing later without resetting the time in between.

Listing 14.5 (`bettersearchcompare.py`) is a rewrite of Listing 11.6 (`searchcompare.py`) that uses our Stopwatch object.

Listing 14.5: `bettersearchcompare.py`

```
def binary_search(lst, seek):
    """
    Returns the index of element seek in list lst,
    if seek is present in lst.
    lst must be in sorted order.
    Returns None if seek is not an element of lst.
    lst is the list in which to search.
    seek is the element to find.
    """
    first = 0          # Initially the first element in list
    last = len(lst) - 1 # Initially the last element in list
    while first <= last:
        # mid is middle of the list
        mid = first + (last - first + 1)//2 # Note: Integer division
        if lst[mid] == seek:
            return mid      # Found it
        elif lst[mid] > seek:
            last = mid - 1   # continue with 1st half
        else: # v[mid] < seek
            first = mid + 1  # continue with 2nd half
    return None           # Not there

def ordered_linear_search(lst, seek):
    """
    Returns the index of element seek in list lst,
    if seek is present in lst.
    lst must be in sorted order.
    Returns None if seek is not an element of lst.
    lst is the list in which to search.
    seek is the element to find.
    """
    i = 0
    n = len(lst)
    while i < n and lst[i] <= seek:
        if lst[i] == seek:
            return i        # Return position immediately
        i += 1
    return None             # Element not found
```

```

def test_searches(lst):
    from stopwatch import Stopwatch

    timer = Stopwatch()
    # Find each element using ordered linear search
    timer.start()    # Start the clock
    n = len(lst)
    for i in range(n):
        if ordered_linear_search(lst, i) != i:
            print("error")
    timer.stop()    # Stop the clock
    print("Linear elapsed time", timer.elapsed())

    # Find each element using binary search
    timer.reset()    # Reset the clock
    timer.start()    # Start the clock
    n = len(lst)
    for i in range(n):
        if binary_search(lst, i) != i:
            print("error")
    timer.stop()    # Stop the clock
    print("Binary elapsed time", timer.elapsed())

def main():
    SIZE = 20000
    test_list = list(range(SIZE))
    test_searches(test_list)

main()

```

This new, object-oriented version is simpler and more readable.

14.5 Automated Testing

We know that just because a program runs to completion without a run-time error does not imply that the program works correctly. We can detect logic errors in our code as we interact with the executing program. The process of exercising code to reveal errors or demonstrate the lack thereof is called *testing*. The informal testing that we have done up to this point has been adequate, but serious software development demands a more formal approach. We will see that good testing requires the same skills and creativity as programming itself.

Until relatively recently in the software development world, testing was often an afterthought. Testing was not perceived to be as glamorous as designing and coding. Poor testing led to buggy programs that frustrated users. Also, tests were written largely after the program's design and coding were complete. The problem with this approach is major design flaws may not be revealed until late in the development cycle. Changes late in the development process are invariably more expensive and difficult to deal with than changes earlier in the process.

Weaknesses in the standard approach to testing led to a new strategy: *test-driven development*. In test-driven development the testing is automated, and the design and implementation of good tests is just as important as the design and development of the actual program. In pure test-driven development, tests are developed *before* any application code is written, and any application code produced is immediately

subjected to testing.

Listing 14.6 (tester.py) defines the structure of a rudimentary test object.

Listing 14.6: tester.py

```
class Tester:
    def __init__(self):
        self.__error_count = self.__total_count = 0
        print("+-----")
        print("|  Testing                               |")
        print("+-----")

    def check_equals(self, msg, expected, actual):
        print("[", msg, "] ")
        self.__total_count += 1 # Count this test
        if expected == actual:
            print("OK")
        else:
            self.__error_count += 1 # Count this failed test
            print("*** Failed! Expected:", expected, " actual:", actual)

    def report_results(self):
        print("+-----")
        print("|", self.__total_count, "tests run")
        print("|", self.__total_count - self.__error_count, " passed")
        print("|", self.__error_count, " failed")
        print("+-----")
```

A simple test object keeps track of the number of tests performed and the number of failures. The client uses the test object to check the results of a computation against a predicted result.

Listing 14.7 (testliststuff.py) uses our Tester class.

Listing 14.7: testliststuff.py

```
from tester import Tester

# sort has a bug (it has yet to be written!)
def sort(lst):
    pass # Sort not yet implemented

# sum has a bug (misses first element)
def sum(lst):
    total = 0
    for i in range(1, len(lst)):
        total += lst[i]
    return total

def main():
    t = Tester() # Make a test object
    # Some test cases to test sort
    col = [4, 2, 3]
    sort(col)
    t.check_equals("Sort test #1", [2, 3, 4], col)
    col = [2, 3, 4]
```

```

    sort(col)
    t.check_equals("Sort test #2", [2, 3, 4], col)
    # Some test cases to test sum
    t.check_equals("Sum test #1", sum([0, 3, 4]), 7)
    t.check_equals("Sum test #2", sum([-3, 0, 5]), 2)

    t.report_results()

main()

```

The program's output is

```

+-----+
|  Testing
+-----+
[ Sort test #1 ]
*** Failed! Expected: [2, 3, 4]  actual: [4, 2, 3]
[ Sort test #2 ]
OK
[ Sum test #1 ]
OK
[ Sum test #2 ]
*** Failed! Expected: 5  actual: 2
+-----+
| 4 tests run
| 2 passed
| 2 failed
+-----+

```

Notice that the `sort` function has yet to be implemented, but we can test it anyway. The first test is bound to fail. The second test checks to see if our `sort` function will not disturb an already sorted vector, and we pass this test with no problem.

In the `sum` function, the programmer was careless and used 1 as the beginning index for the vector. Notice that the first test does not catch the error, since the element in the zeroth position (zero) does not affect the outcome. A tester must be creative and even devious to try and force the code under test to demonstrate its errors.

14.6 Class Inheritance

CAUTION! SECTION UNDER CONSTRUCTION

We can base a new class on an existing class using a technique known as *inheritance*. Recall our `Stopwatch` class we defined in Listing 14.4 (`stopwatch.py`). Our `Stopwatch` objects may be started and stopped as often as necessary without resetting the time. Support we need a stopwatch object that records the number of times the watch is started until it is reset. We can build our enhanced `Stopwatch` class from scratch, but it would more efficient to base our new class on the existing `Stopwatch` class. Listing 14.8 (`countingstopwatch.py`) defines our enhanced stopwatch objects.

Listing 14.8: `countingstopwatch.py`

```

from stopwatch import Stopwatch

```

```
class CountingStopwatch(Stopwatch):
    def __init__(self):
        # Allow superclass to do its initialization of the
        # inherited fields
        super().__init__()
        # Set number of starts to zero
        self.__count = 0

    def start(self):
        # Let superclass do its start code
        super().start()
        # Count this start message
        self.__count += 1

    def reset(self):
        # Let superclass reset the inherited fields
        super().reset()
        # Reset new field that the base class method does not know about
        self.__count = 0

    def count(self):
        return self.__count
```

The line

```
from stopwatch import Stopwatch
```

indicates that the code in this module will somehow use the Stopwatch class from Listing 14.4 (stopwatch.py). The line

```
class CountingStopwatch (Stopwatch):
```

defines a new class named CountingStopwatch, but this new class is based on the existing class Stopwatch. This single line means that the CountingStopwatch class *inherits* everything from the Stopwatch class. CountingStopwatch objects automatically will have start, stop, reset, and elapsed methods.

We say stopwatch is the *superclass* of CountingStopwatch. Another term for superclass is *base class*. CountingStopwatch is the *subclass* of Stopwatch, or, said another way, CountingStopwatch is a *derived class* of Stopwatch.

Even though a subclass inherits all the fields and methods of its superclass, a subclass may add new fields and methods and provide new code for an inherited method. The statement

```
super().__init__()
```

in the `__init__` method definition calls the constructor of the superclass. After executing the superclass constructor code, the subclass constructor defines and initializes the new `__count` field. The start and reset methods in CountingStopwatch similarly invoke the services of their counterparts in the superclass. The count method is a brand new method not found in the superclass.

Notice that the CountingStopwatch class has no apparent stop method. In fact, it inherits the stop method as is from Stopwatch.

Listing 14.9 (usecountingsw.py) provides some sample client code that uses the CountingStopwatch class.

Listing 14.9: usecountingsw.py

```
from countingstopwatch import CountingStopwatch
from time import sleep

timer = CountingStopwatch()
timer.start()
sleep(10) # Pause program for 10 seconds
timer.stop()
print("Time:", timer.elapsed(), "    Number:", timer.count())

timer.start()
sleep(5) # Pause program for 5 seconds
timer.stop()
print("Time:", timer.elapsed(), "    Number:", timer.count())

timer.start()
sleep(20) # Pause program for 20 seconds
timer.stop()
print("Time:", timer.elapsed(), "    Number:", timer.count())
```

Listing 14.9 (usecountingsw.py) produces

```
Time: 10.010378278632945    Number: 1
Time: 15.016618866378108    Number: 2
Time: 35.02881993198008     Number: 3
```

14.7 Custom Exceptions

CAUTION! SECTION UNDER CONSTRUCTION

14.8 Summary

CAUTION! SECTION UNDER CONSTRUCTION

- The `class` reserved word introduces a programmer-defined type.
- Variables of a class are called *objects* or *instances* of that class.
- The dot (.) operator is used to access elements of an object.
- A data member of a class is known as a *field*. Equivalent terms include *data member*, *instance variable*, and *attribute*.
- A function defined in a class that operates on objects of that class is called a *method*. Equivalent terms include *member function* and *operation*.
- Encapsulation and data hiding offers several benefits to programmers:

- Flexibility—class authors are free to change the private details of a class. Existing client code need not be changed to work with the new implementation.
 - Reducing programming errors—if client code cannot touch directly the hidden details of an object, the internal state of that object is completely under the control of the class author. With a well-designed class, clients cannot place the object in an ill-defined state (thus leading to incorrect program execution).
 - Hiding complexity—the hidden internals of an object might be quite complex, but clients cannot see and should not be concerned with those details. Clients need to know *what* an object can do, not *how* it accomplishes the task.
- A field with a name that begins with two underscores (`__`) is not meant to be used directly by clients.

14.9 Exercises

CAUTION! SECTION UNDER CONSTRUCTION

1. Given the definition of the geometric Point class, complete the function named distance:

```
def distance(r1, r2):
    # Details go here
```

that returns the distance between the two points passed as parameters.

2. Given the definition of the Rational number class, complete the following function named reduce:

```
def reduce(r):
    # Details go here
```

that returns the rational number that represents the parameter reduced to lowest terms; for example, the fraction 10/20 would be reduced to 1/2.

3. What is the purpose of the `__init__` method in a class?
4. What is the parameter named `self` that appears as the first parameter of a method?
5. Given the definition of the Rational number class, complete the following method named reduce:

```
class Rational:
    # Other details omitted here ...

    # Returns an object of the same value reduced
    # to lowest terms
    def reduce(self):
        # Details go here
```

that returns the rational number that represents the object reduced to lowest terms; for example, the fraction 10/20 would be reduced to 1/2.

6. Given the definition of the Rational number class, complete the following method named reduce:

```
class Rational:
    # Other details omitted here ...

    # Reduces the object to lowest terms
    def reduce(self):
        # Details go here
```

that reduces the object on whose behalf the method is called to lowest terms; for example, the fraction 10/20 would be reduced to 1/2.

7. Given the definition of the geometric Point class, add a method named distance:

```
class Point:
    # Other details omitted

    # Returns the distance from this point to the
    # parameter p
    double distance(self, p):
        # Details go here
```

that returns the distance between the point on whose behalf the method is called and the parameter p.

Index

None object, 257
 end keyword argument in print, 32
 lambda expressions, 215
 len function, 267
 list function, 270
 sep keyword argument in print, 34
 try block, 374

absolute value, 96
 accessor methods, 402
 accumulator, 107
 actual, 153
 algorithm, 58
 aliasing, 276
 aliasing, object, 255
 associative array, 353
 associative container, 349
 attributes, 235

base case, 208
 base class, 416
 binary search, 311
 block, 11
 body, 68
 Boolean, 65
 bugs, 54

callback function, 405
 calling code, 149
 catching an exception, 375
 chained assignment, 48
 class, 15, 235
 class constructor, 247
 client, 236
 client code, 149
 closure, 216
 code instrumentation, 329
 comma-separated list, 19
 command prompt, 288
 commutative, 272
 compiler, 2, 3
 concatenation, 16

conditional expression, 95
 context manager, 242
 control codes, 28

data, 235
 data persistence, 240
 debugger, 4
 default argument, 204
 default parameters, 204
 definite loop, 109
 delimiter, 14
 denominator, 247
 dependent function, 203
 derivative, 228
 derived class, 416
 dictionary, 349
 dictionary key, 349
 differentiation, 228
 docstring, 212
 documentation string, 212
 double negative, 74

EAFP, 374
 easier to ask for forgiveness than permission, 374
 elapsed time, 156
 Eratosthenes, 286
 escape symbol, 28
 exception, 369
 exception handling, 369
 existential quantification, 364
 expression, 13
 external documentation, 212

factorial, 205
 field, object, 399
 fields, 236
 files, 240
 floating-point numbers, 25
 formal, 153
 formatting string, 36
 function, 148
 function definition, 168

- function invocation, 168
- function `time.clock`, 156
- function `time.sleep`, 158
- function call, 149
- function coherence, 182
- function composition, 178, 186
- function definition parts, 168
- function invocation, 149
- functional composition, 30
- functional decomposition, 168
- functional independence, 203
- garbage, 257
- garbage collection, 257
- generator, 219
- generator comprehension, 293
- generator expression, 293
- getters, 402
- global variable, 199
- handling an exception, 374
- hash tables, 353
- hashing, 353
- Hoare, C. A. R., 306
- identifier, 23
- identifiers beginning with two underscores, 406
- immutable, 179
- in place list processing, 323
- indefinite loop, 110
- index, 239, 263
- inheritance, 415
- integer division, 42
- internal documentation, 212
- interpreter, 3, 4
- iterable, 111
- key, 349, 350
- keyword argument, 32
- keyword arguments, 360
- keywords, 24
- lambda calculus, 216
- lazy list, 295
- LBYL, 373
- length, 267
- linear search, 310
- list aliasing, 276
- list comprehension, 290
- list slicing, 280
- local variable, 173
- local variables, 199
- look before you leap, 373
- loop unrolling, 327
- member, object, 399
- members, 236
- Mersenne Twister, 159
- method call, 236
- method, object, 399
- methods, 235
- midpoint, 174
- module, 149
- modules, 147, 210
- modulus, 42
- monolithic code, 167
- multiplication table, 115
- mutable, 266, 276
- mutator methods, 402
- name collision, 162
- namespace, 162
- namespace pollution, 162
- nested, 80
- nested loop, 115
- newline, 28
- numerator, 247
- numerical differentiation, 228
- object aliasing, 255
- object oriented, 235
- operations, 236
- pass statement, 77
- permutation, 119, 325
- positional parameters, 36
- predicate, 65, 365
- private object members, 406
- profiler, 4
- pure function, 203
- qualified name, 162
- quantification, 364
- Quicksort, 306
- random access, 295
- read, eval, print loop, 14
- recursive, 205
- recursive case, 207
- reference counting, 257
- regular expression, 374
- relational operators, 66
- remainder, 42

- reserved words, 24
- run-time errors, 51
- run-time exceptions, 51
- scripting languages, 4
- selection sort, 301
- set comprehension, 364
- setters, 402
- short-circuit evaluation, 75
- Sieve of Eratosthenes, 286
- slice assignment, 281
- slicing, 280
- stack, 388
- stack trace, 388
- stack unwinding, 389
- statement, 5
- string, 14
- string formatter, 36
- subclass, 416
- subscript, 263
- superclass, 416
- symbolic differentiation, 228
- syntactic sugar, 248
- syntax error, 50
- terminal, 288
- test-driven development, 413
- testing, 413
- text files, 244
- texttttry statement, 374
- throwing an exception, 375
- times table, 115
- Tk toolkit, 250
- tkinter, 250
- token, 358
- translation phase, 50
- tuple, 19
- tuple assignment, 19
- tuple unpacking, 175
- Turtle graphics, 248
- type, 15
- unbound variable, 22
- undefined variable, 22
- underscore, beginning an identifier, 406
- universal quantification, 364
- unpacking a tuple, 175
- unwinding the stack, 389
- value, 350
- whitespace, 11
- yield from, 331