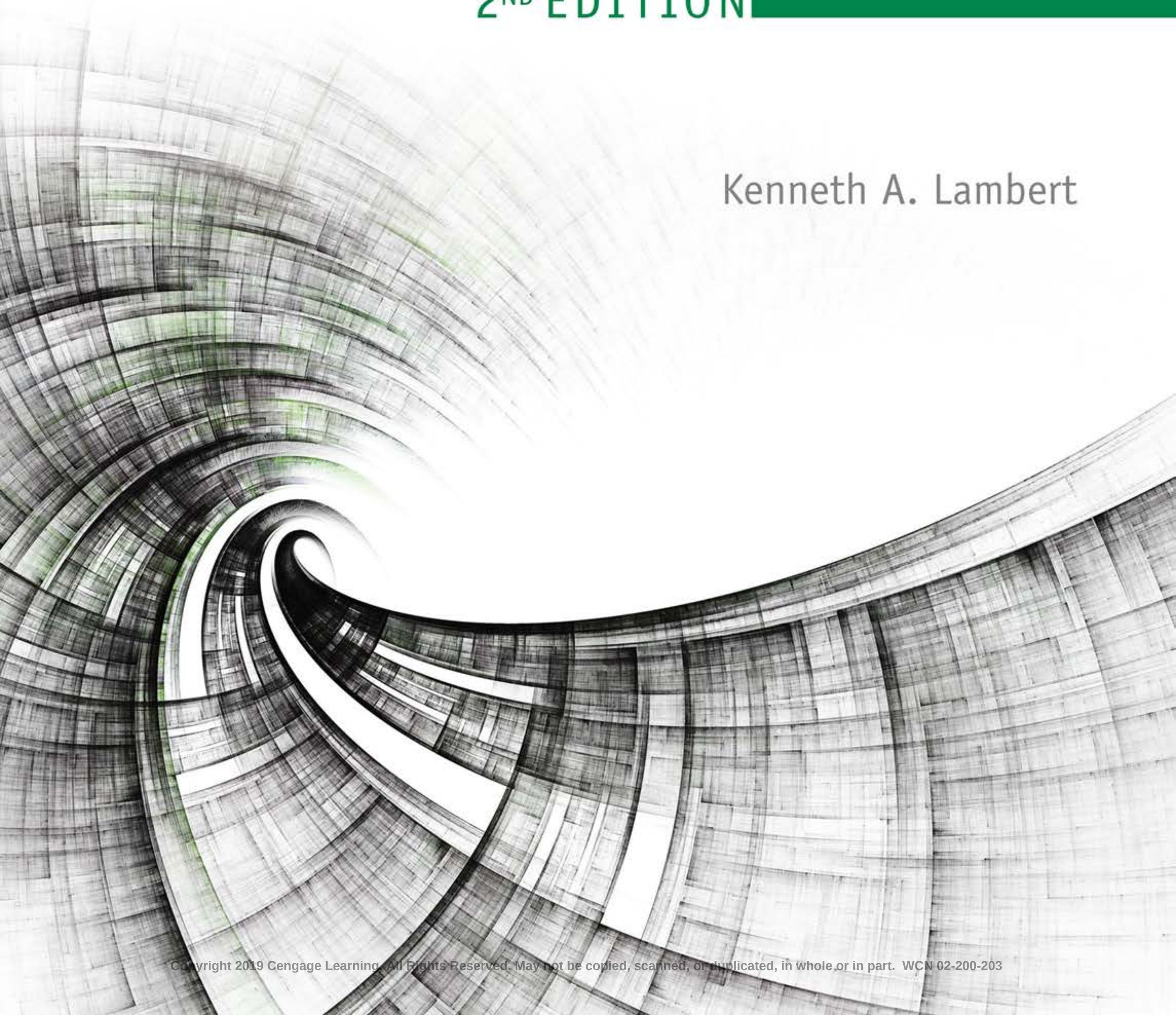
Fundamentals of

PYTHON DATA STRUCTURES

2ND EDITION

Kenneth A. Lambert



SECOND EDITION

FUNDAMENTALS OF PYTHON: DATA STRUCTURES

KENNETH A. LAMBERT

CENGAGE

Australia • Brazil • Mexico • Singapore • United Kingdom • United States

CENGAGE

***Fundamentals of Python: Data Structures*, Second Edition**

Kenneth A. Lambert

SVP, GM Skills & Global Product Management: Jonathan Lau

Product Team Manager: Kristin McNary

Product Manager: Chris Shortt

Product Assistant: Thomas Benedetto

Executive Director, Content Design: Marah Bellegarde

Director, Learning Design: Leigh Hefferon

Learning Designer: Kate Mason

Vice President, Strategic Marketing Services: Jennifer Baker

Marketing Director: Michele McTighe

Associate Marketing Manager: Cassie Cloutier

Director, Content Delivery: Patty Stephan

Senior Content Manager: Michelle Ruelos Cannistraci

Designer: Erin K. Griffin

Cover image: Digital_Art/Shutterstock.com

Service Provider/Compositor: SPi Global

Library of Congress Control Number: 2018956860

ISBN: 978-0-357-12275-4

Cengage

20 Channel Center Street

Boston, MA 02210

USA

Printed in the United States of America
Print Number: 01 Print Year: 2018

Table of Contents

Preface

Welcome to *Fundamentals of Python: Data Structures, 2nd Edition*. This text is intended for a second semester course in programming and problem solving with data structures. It covers the material taught in a typical Computer Science 2 course (CS2) at the undergraduate level. Although this book uses the Python programming language, you need only have a basic knowledge of programming in a high-level programming language before beginning Chapter 1.

What You'll Learn

The book covers four major aspects of computing:

1. **Programming basics**—Data types, control structures, algorithm development, and program design with functions are basic ideas that you need to master to solve problems with computers. You'll review these core topics in the Python programming language and employ your understanding of them to solve a wide range of problems.
2. **Object-Oriented Programming (OOP)**—Object-Oriented Programming is the dominant programming paradigm used to develop large software systems. You'll be introduced to the fundamental principles of OOP so that you can apply them successfully. Unlike other textbooks, this book helps you develop a professional-quality framework of collection classes to illustrate these principles.
3. **Data structures**—Most useful programs rely on data structures to solve problems. At the most concrete level, data structures include arrays and various types of linked structures. You'll use these data structures to implement various types of collection structures, such as stacks, queues, lists, trees, bags, sets, dictionaries, and graphs. You'll also learn to use complexity analysis to evaluate the space/time trade-offs of different implementations of these collections.
4. **Software development life cycle**—Rather than isolate software development techniques in one or two chapters, this book deals with them throughout in the context of numerous case studies. Among other things, you'll learn that coding a program is often not the most difficult or challenging aspect of problem solving and software development.

Why Python?

Computer technology and applications have become increasingly more sophisticated over the past three decades, and so has the computer science curriculum, especially at the introductory level. Today's students learn a bit of programming and problem solving and are then expected to move quickly into topics like software development, complexity analysis, and data structures that, 30 years ago, were relegated to advanced courses. In addition, the ascent of object-oriented programming as the dominant paradigm has led instructors and textbook authors to bring powerful, industrial-strength programming languages such as C++ and Java into the introductory curriculum. As a result, instead of experiencing the rewards and excitement of solving problems with computers, beginning computer science students often become overwhelmed by the combined tasks of mastering advanced concepts as well as the syntax of a programming language.

This book uses the Python programming language as a way of making the second course in computer science more manageable and attractive for students and instructors alike. Python has the following pedagogical benefits:

- Python has simple, conventional syntax. Python statements are very close to those of pseudocode algorithms, and Python expressions use the conventional notation found in algebra. Thus, you can spend less time dealing with the syntax of a programming language and more time learning to solve interesting problems.
- Python has safe semantics. Any expression or statement whose meaning violates the definition of the language produces an error message.
- Python scales well. It is easy for beginners to write simple programs in Python. Python also includes all the advanced features of a modern programming language, such as support for data structures and object-oriented software development, for use when they become necessary, especially in the second course in computer science
- Python is highly interactive. You can enter expressions and statements at an interpreter's prompts to try out experimental code and receive immediate feedback. You can also compose longer code segments and save them in script files to be loaded and run as modules or stand-alone applications.
- Python is general purpose. In today's context, this means that the language includes resources for contemporary applications, including media computing and web services.
- Python is free and is in widespread use in the industry. You can download Python to run on a variety of devices. There is a large Python user community, and expertise in Python programming has great resume value.

To summarize these benefits, Python is a comfortable and flexible vehicle for expressing ideas about computation, both for beginners and for experts. If you learn these ideas well in the first year, you should have no problems making a quick transition to other languages needed for courses later in the curriculum. Most importantly, you will spend less time staring at a computer screen and more time thinking about interesting problems to solve.

Organization of This Book

The approach in this book is easygoing, with each new concept introduced only when it is needed.

Chapter 1 provides a review of the features of Python programming that are needed to begin a second course in programming and problem solving in Python. The content of this chapter is organized so that you can skim it quickly if you have experience in Python programming, or you can dig a bit deeper to get up to speed in the language if you are new to Python.

Chapters 2 through 12 covers the major topics in a typical CS2 course, especially the specification, implementation, and application of abstract data types, with the collection types as the primary vehicle and focus. Along the way, you will be thoroughly exposed to object-oriented programming techniques and the elements of good software design. Other important CS2 topics include recursive processing of data, search and sort algorithms, and the tools used in software development, such as complexity analysis and graphical notations (UML) to document designs.

Chapter 2 introduces the concept of an abstract data type (ADT) and provides an overview of various categories of collection ADTs.

Chapters 3 and 4 explore the data structures used to implement most collections and the tools for analyzing their performance trade-offs. Chapter 3 introduces complexity analysis with big-O notation. Enough material is presented to enable you to perform simple analyses of the running time and memory usage of algorithms and data structures, using search and sort algorithms as examples. Chapter 4 covers the details of processing arrays and linear linked structures, the concrete data structures used to implement most collections. You'll learn the underlying models of computer memory that support arrays and linked structures and the time/space trade-offs that they entail.

Chapters 5 and 6 shift the focus to the principles of object-oriented design. These principles are used to organize a professional-quality framework of collection classes that will be covered in detail in later chapters.

Chapter 5 is concerned with the critical difference between interface and implementation. A single interface and several implementations of a bag collection are developed as a first example. Emphasis is placed on the inclusion of conventional methods in an interface, to allow different types of collections to collaborate in applications. For example, one such method creates an iterator, which allows you to traverse any collection with a simple loop. Other topics covered in this chapter include polymorphism and information hiding, which directly stem from the difference between interface and implementation.

Chapter 6 shows how class hierarchies can reduce the amount of redundant code in an object-oriented software system. The related concepts of inheritance, dynamic binding of method calls, and abstract classes are introduced here and used throughout the remaining chapters.

Armed with these concepts and principles, you'll then be ready to consider the other major collection ADTs, which form the subject of Chapters 7 through 12.

Chapters 7 through 9 present the linear collections, stacks, queues, and lists. Each collection is viewed first from the perspective of its users, who are aware only of an interface and a set of performance characteristics possessed by a chosen implementation. The use of each collection is illustrated with one or more applications, and then several implementations are developed, and their performance trade-offs are analyzed.

Chapters 10 through 12 present advanced data structures and algorithms as a transition to later courses in computer science. Chapter 10 discusses various tree structures, including binary search trees, heaps, and expression trees. Chapter 11 examines the implementation of the unordered collections, bags, sets, and dictionaries, using hashing strategies. Chapter 12 introduces graphs and graph-processing algorithms.

As mentioned earlier, this book is unique in presenting a professional-quality framework of collection types. Instead of encountering a series of apparently unrelated collections, you will explore the place of each collection in an integrated whole. This approach allows you to see what the collection types have in common as well as what makes each one unique. At the same time, you will be exposed to a realistic use of inheritance and class hierarchies, topics in object-oriented software design that are difficult to motivate and exemplify at this level of the curriculum.

Special Features

This book explains and develops concepts carefully, using frequent examples and diagrams. New concepts are then applied in complete programs to show how they aid in solving problems. The chapters place an early and consistent emphasis on good writing habits and neat, readable documentation.

The book includes several other important features:

- **Case studies**—These present complete Python programs ranging from the simple to the substantial. To emphasize the importance and usefulness of the software development life cycle, case studies are discussed in the framework of a user request, followed by analysis, design, implementation, and suggestions for testing, with well-defined tasks performed at each stage. Some case studies are extended in end-of-chapter programming projects.
- **Chapter summaries**—Each chapter after the first one ends with a summary of the major concepts covered in the chapter.
- **Key terms**—When a new term is introduced in the text, it appears in bold face. Definitions of the key terms are also collected in a glossary.
- **Exercises**—Most major sections of each chapter after the first one end with exercise questions that reinforce the reading by asking basic questions about the material in the section. After Chapter 2, each chapter ends with review questions.
- **Programming projects**—Each chapter ends with a set of programming projects of varying difficulty.

New in This Edition

The most obvious change in this edition is the addition of full color. All program examples include the color coding used in Python's IDLE, so students can easily identify program elements such as keywords, comments, and function, method, and class names. Learning objectives have been added to the beginning of each chapter. Several new figures have been added to illustrate concepts, and many programming projects have been added or reworked. A new section on iterators and higher-order functions has been added to Chapter 2. Finally, a new section on Lisp-like lists, recursive list processing, and functional programming has been added to Chapter 9.

Instructor Resources

MindTap

MindTap activities for *Fundamentals of Python: Data Structures* are designed to help students master the skills they need in today's workforce. Research shows employers need critical thinkers, troubleshooters, and creative problem-solvers to stay relevant in our fast-paced, technology-driven world. MindTap helps you achieve this with assignments and activities that provide hands-on practice and real-life relevance. Students are guided through assignments that help them master basic knowledge and understanding before moving on to more challenging problems.

All MindTap activities and assignments are tied to defined unit learning objectives. Hands-on coding labs provide real-life application and practice. Readings and dynamic visualizations support the lecture, while a post-course assessment measures exactly how much a class stands in terms of progress, engagement, and completion rates. Use the content and learning path as-is, or pick and choose how our materials will wrap around yours. You control what the students see and when they see it. Learn more at http://www.cengage.com/mindtap/.

Instructor Companion Site

The following teaching tools are available for download at the Companion Site for this text. Go to instructor.cengage.com and sign in to the instructor account. Search for the textbook and add the text to the instructor dashboard.

- **Instructor's Manual:** The Instructor's Manual that accompanies this textbook includes additional instructional material to assist in class preparation, including items such as Overviews, Chapter Objectives, Teaching Tips, Quick Quizzes, Class Discussion Topics, Additional Projects, Additional Resources, and Key Terms. A sample syllabus is also available.
- **Test Bank:** Cengage Testing Powered by Cognero is a flexible, online system that allows you to:
    - author, edit, and manage test bank content from multiple Cengage solutions
    - create multiple test versions in an instant
    - deliver tests from your LMS, your classroom, or wherever you want
- **PowerPoint Presentations:** This text provides PowerPoint slides to accompany each chapter. Slides may be used to guide classroom presentations, to make available to students for chapter review, or to print as classroom handouts. Files are provided for every figure in the text. Instructors may use the files to customize PowerPoint slides, illustrate quizzes, or create handouts.

- **Solutions:** Solutions to all programming exercises are available. If an input file is needed to run a programming exercise, it is included with the solution file.
- **Source Code:** The source code is available at www.cengage.com. If an input file is needed to run a program, it is included with the source code.

We Appreciate Your Feedback

We have tried to produce a high-quality text, but should you encounter any errors, please report them to lambertk@wlu.edu. A listing of errata, should they be found, as well as other information about the book, will be posted on the website http://home.wlu.edu/~lambertk/python/.

Acknowledgments

I would like to thank my friend, Martin Osborne, for many years of advice, friendly criticism, and encouragement on several of my book projects.

I would also like to thank my students in Computer Science 112 at Washington and Lee University for classroom testing this book over several semesters.

Finally, I would like to thank Kristin McNary, Product Team Manager; Chris Shortt, Product Manager; Maria Garguilo and Kate Mason, Learning Designers; Magesh Rajagopalan, Senior Project Manager; Danielle Shaw, Tech Editor; and especially Michelle Ruelos Cannistraci, Senior Content Manager, for handling all the details of producing this edition of the book.

About the Author

Kenneth A. Lambert is a professor of computer science and the chair of that department at Washington and Lee University. He has taught introductory programming courses for over 30 years and has been an active researcher in computer science education. Lambert has authored or coauthored a total of 28 textbooks, including a series of introductory C++ textbooks with Douglas Nance and Thomas Naps, a series of introductory Java textbooks with Martin Osborne, and a series of introductory Python textbooks.

Dedication

To Brenda Wilson, with love and admiration.
Kenneth A. Lambert
Lexington, VA

CHAPTER 1

Basic Python Programming

After completing this chapter, you will be able to:

- Write a simple Python program using its basic structure
- Perform simple input and output operations
- Perform operations with numbers such as arithmetic and comparisons
- Perform operations with Boolean values
- Implement an algorithm using the basic constructs of sequences of statements, selection statements, and loops
- Define functions to structure code
- Use built-in data structures such as strings, files, lists, tuples, and dictionaries
- Define classes to represent new types of objects
- Structure programs in terms of cooperating functions, data structures, classes, and modules

This chapter gives a quick overview of Python programming. It is intended to bring those new to or rusty in Python up to speed, but it does not pretend to be a thorough introduction to computer science or the Python programming language. For a more detailed treatment of programming in Python, see my book *Fundamentals of Python: First Programs, Second Edition* (Cengage Learning, 2019). For documentation on the Python programming language, visit *www.python.org*.

If your computer already has Python, check the version number by running the **`python`** or **`python3`** command at a terminal prompt. (Linux and Mac users first open a terminal window, and Windows users first open a DOS window.) You are best off using the most current version of Python available. Check for that at *www.python.org*, and download and install the latest version if necessary. You will need Python 3.0 or higher to run the programs presented in this book.

Basic Program Elements

Like all contemporary programming languages, Python has a vast array of features and constructs. However, Python is among the few languages whose basic program elements are quite simple. This section discusses the essentials to get you started in Python programming.

Programs and Modules

A Python program consists of one or more modules. A module is just a file of Python code, which can include statements, function definitions, and class definitions. A short Python program, also called a **script**, can be contained in one module. Longer, more complex programs typically include one main module and one or more supporting modules. The main module contains the starting point of program execution. Supporting modules contain function and class definitions.

An Example Python Program: Guessing a Number

Next, you'll see a complete Python program that plays a game of guess-the-number with the user. The computer asks the user to enter the lower and upper bounds of a range of numbers. The computer then "thinks" of a random number in that range and repeatedly asks the user to guess this number until the user enters a correct guess. The computer gives a hint to the user after each guess and displays the total number of guesses at the end of the process. The program includes several of the types of Python statements to be discussed later in this chapter, such as input statements, output statements, assignment statements, loops, and conditional statements. The program also includes a single function definition.

Here is the code for the program, in the file numberguess.py:

```
"""
Author: Ken Lambert
Plays a game of guess the number with the user.
"""
```

```
import random

def main():
    """Inputs the bounds of the range of numbers
    and lets the user guess the computer's number until
    the guess is correct."""
    smaller = int(input("Enter the smaller number: "))
    larger = int(input("Enter the larger number: "))
    myNumber = random.randint(smaller, larger)
    count = 0
    while True:
        count += 1
        userNumber = int(input("Enter your guess: "))
        if userNumber < myNumber:
            print("Too small")
        elif userNumber > myNumber:
            print("Too large")
        else:
            print("You've got it in", count, "tries!")
            break

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

Here is a trace of a user's interaction with the program:

```
Enter the smaller number: 1
Enter the larger number: 32
Enter your guess: 16
Too small
Enter your guess: 24
Too large
Enter your guess: 20
You've got it in 3 tries!
```

Note that the code and its trace appear in the colors black, blue, orange, and green. Python's IDLE uses color coding to help the reader recognize various types of program elements. The role of each color will be explained shortly.

Editing, Compiling, and Running Python Programs

You can run complete Python programs, including most of the examples presented, by entering a command in a terminal window. For example, to run the program contained in the file numberguess.py, enter the following command in most terminal windows:

```
python3 numberguess.py
```

To create or edit a Python module, try using Python's IDLE (short for Integrated DeveLopment Environment). To start IDLE, enter the idle or **idle3** command at a terminal prompt or launch its icon if it is available. You can also launch IDLE by double-clicking on a Python source code file (any file with a .py extension) or by right-clicking on the file and selecting Open or Edit with IDLE. Make sure that your system is set to open IDLE when files of this type are launched (this is the default on macOS but not on Windows).

IDLE gives you a shell window for interactively running Python expressions and statements. Using IDLE, you can move back and forth between editor windows and the shell window to develop and run complete programs. IDLE also formats your code and color-codes it.

When you open an existing Python file with IDLE, the file appears in an editor window, and the shell pops up in a separate window. To run a program, move the cursor into the editor window and press the F5 (function-5) key. Python compiles the code in the editor window and runs it in the shell window.

If a Python program appears to hang or not quit normally, you can exit by pressing Ctrl+C or closing the shell window.

Program Comments

A program comment is text ignored by the Python compiler but valuable to the reader as documentation. An end-of-line comment in Python begins with a **#** symbol and extends to the end of the current line. It is color-coded in red. For example:

```
# This is an end-of-line comment.
```

A multiline comment is a string enclosed in triple single quotes or triple double quotes. Such comments, which are colored green, are also called **docstrings**, to indicate that they can document major constructs within a program. The **numberguess** program shown earlier includes two doc strings. The first one, at the top of the program file, serves as a comment for the entire **numberguess** module. The second one, just below the header of the **main** function, describes what this function does. As we shall see shortly, docstrings play a critical role in giving help to a programmer within the Python shell.

Lexical Elements

The lexical elements in a language are the types of words or symbols used to construct sentences. As in all high-level programming languages, some of Python's basic symbols are keywords, such as **if**, **while**, and **def**, which are colored orange. Also included among lexical items are identifiers (names), literals (numbers, strings, and other built-in data structures), operators, and delimiters (quotation marks, commas, parentheses, square brackets, and braces). Among the identifiers are the names of built-in functions, which are colored purple.

Spelling and Naming Conventions

Python keywords and names are case-sensitive. Thus, **while** is a keyword, whereas **While** is a programmer-defined name. Python keywords are spelled in lowercase letters and are color-coded in orange in an IDLE window.

All Python names, other than those of built-in functions, are color-coded in black, except when they are introduced as function, class, or method names, in which case they appear in blue. A name can begin with a letter or an underscore (_), followed by any number of letters, underscores, or digits.

In this book, the names of modules, variables, functions, and methods are spelled in lowercase letters. With the exception of modules, when one of these names contains one or more embedded words, the embedded words are capitalized. The names of classes follow the same conventions but begin with a capital letter. When a variable names a constant, all the letters are uppercase, and an underscore separates any embedded words. Table 1-1 shows examples of these naming conventions.

Type of Name	Examples
Variable	`salary`, `hoursWorked`, `isAbsent`
Constant	`ABSOLUTE_ZERO`, `INTEREST_RATE`
Function or method	`printResults`, `cubeRoot`, `input`
Class	`BankAccount`, `SortedSet`

Table 1-1 Examples of Python Naming Conventions

Use names that describe their role in a program. In general, variable names should be nouns or adjectives (if they denote Boolean values), whereas function and method names should be verbs if they denote actions, or nouns or adjectives if they denote values returned.

Syntactic Elements

The syntactic elements in a language are the types of sentences (expressions, statements, definitions, and other constructs) composed from the lexical elements. Unlike most high-level languages, Python uses white space (spaces, tabs, or line breaks) to mark the syntax of many types of sentences. This means that indentation and line breaks are significant in Python code. A smart editor like Python's IDLE can help indent code correctly. The programmer need not worry about separating sentences with semicolons and marking blocks of sentences with braces. In this book, I use an indentation width of four spaces in all Python code.

Literals

Numbers (integers or floating-point numbers) are written as they are in other programming languages. The Boolean values **`True`** and **`False`** are keywords. Some data structures, such as strings, tuples, lists, and dictionaries, also have literals, as you will see shortly.

String Literals

You can enclose strings in single quotes, double quotes, or sets of three double quotes or three single quotes. The last notation is useful for a string containing multiple lines of text. Character values are single-character strings. The \ character is used to escape nongraphic characters such as the newline (**\n**) and the tab (**\t**), or the \ character itself. The next code segment, followed by the output, illustrates the possibilities.

```
print("Using double quotes")
print('Using single quotes')
print("Mentioning the word 'Python' by quoting it")
print("Embedding a\nline break with \\n")
print("""Embedding a
line break with triple quotes""")
```

Output:

```
Using double quotes
Using single quotes
Mentioning the word 'Python' by quoting it
Embedding a
line break with \n
Embedding a
line break with triple quotes
```

Operators and Expressions

Arithmetic expressions use the standard operators (+, –, *, /, %) and infix notation. The / operator produces a floating-point result with any numeric operands, whereas the // operator produces an integer quotient. The + operator means concatenation when used with collections, such as strings and lists. The ** operator is used for exponentiation.

The comparison operators <, <=, >, >=, ==, and != work with numbers and strings.

The == operator compares the internal contents of data structures, such as two lists, for structural equivalence, whereas the **is** operator compares two values for object identity. Comparisons return **True** or **False**.

The logical operators **and**, **or**, and **not** treat several values, such as 0, **None**, the empty string, and the empty list, as **False**. In contrast, most other Python values count as **True**.

The subscript operator, **[]**, used with collection objects, will be examined shortly.

The selector operator, '**.**', is used to refer to a named item in a module, class, or object.

The operators have the standard precedence (selector, function call, subscript, arithmetic, comparison, logical, assignment). Parentheses are used in the usual manner, to group subexpressions for earlier evaluation.

The ** and = operators are right associative, whereas the others are left associative.

Function Calls

Functions are called in the usual manner, with the function's name followed by a parenthesized list of arguments. For example:

```
min(5, 2)     # Returns 2
```

Python includes a few standard functions, such as **abs** and **round**. Many other functions are available by import from modules, as you will see shortly.

The `print` Function

The standard output function **print** displays its arguments on the console. This function allows a variable number of arguments. Python automatically runs the **str** function on each argument to obtain its string representation and separates each string with a space before output. By default, **print** terminates its output with a newline.

The input Function

The standard input function **input** waits for the user to enter text at the keyboard. When the user presses the Enter key, the function returns a string containing the characters entered. This function takes an optional string as an argument and prints this string, without a line break, to prompt the user for the input.

Type Conversion Functions and Mixed-Mode Operations

You can use some data type names as type conversion functions. For example, when the user enters a number at the keyboard, the **input** function returns a string of digits, not a numeric value. The program must convert this string to an **int** or a **float** before numeric processing. The next code segment inputs the radius of a circle, converts this string to a **float**, and computes and outputs the circle's area:

```
radius = float(input("Radius: "))
print("The area is", 3.14 * radius ** 2)
```

Like most other languages, Python allows operands of different numeric types in arithmetic expressions. In those cases, the result type is the same type as the most general operand type. For example, the addition of an **int** and a **float** produces a **float** as the result.

Optional and Keyword Function Arguments

Functions may allow optional arguments, which can be named with keywords when the function is called. For example, the **print** function by default outputs a newline after its arguments are displayed. To prevent this from happening, you can give the optional argument `end` a value of the empty string, as follows:

```
print("The cursor will stay on this line, at the end", end = "")
```

Required arguments have no default values. Optional arguments have default values and can appear in any order when their keywords are used, as long as they come after the required arguments.

For example, the standard function **round** expects one required argument, a rounded number, and a second, optional argument, the number of figures of precision. When the second argument is omitted, the function returns the nearest whole number (an **int**). When the second argument is included, the function returns a **float**. Here are some examples:

```
>>> round(3.15)
3

>>> round(3.15, 1)
3.2
```

In general, the number of arguments passed to a function when it is called must be at least the same number as its required arguments.

Standard functions and Python's library functions check the types of their arguments when the function is called. Programmer-defined functions can receive arguments of any type, including functions and types themselves.

Variables and Assignment Statements

A Python variable is introduced with an assignment statement. For example:

```
PI = 3.1416
```

sets `PI` to the value 3.1416. The syntax of a simple assignment statement is:

```
<identifier> = <expression>
```

Several variables can be introduced in the same assignment statement, as follows:

```
minValue, maxValue = 1, 100
```

To swap the values of the variables **a** and **b**, you write:

```
a, b = b, a
```

Assignment statements must appear on a single line of code, unless the line is broken after a comma, parenthesis, curly brace, or square bracket. When these options are unavailable, another means of breaking a line within a statement is to end it with the escape symbol \. You typically place this symbol before or after an operator in an expression. Here are some admittedly unrealistic examples:

```
minValue = min(100,
               200)
product = max(100, 200) \
          * 30
```

When you press Enter after a comma or the escape symbol, IDLE automatically indents the next line of code.

Python Data Typing

In Python, any variable can name a value of any type. Variables are not declared to have a type, as they are in many other languages; they are simply assigned a value.

Consequently, data type names almost never appear in Python programs. However, all values or objects have types. The types of operands in expressions are checked at run time, so type errors do not go undetected; however, the programmer does not have to worry about mentioning data types when writing code.

`import` Statements

The **`import`** statement makes visible to a program the identifiers from another module. These identifiers might name objects, functions, or classes. There are several ways to express an **`import`** statement. The simplest is to import the module name, as in:

```
import math
```

This makes any name defined in the math module available to the current module, by using the syntax **`math.<name>`**. Thus, **`math.sqrt(2)`** would return the square root of 2.

A second style of importing brings in a name itself, which you can use directly without the module name as a prefix:

```
from math import sqrt
print(sqrt(2))
```

You can import several individual names by listing them:

```
from math import pi, sqrt
print(sqrt(2) * pi)
```

You can import all names from a module using the * operator, but that is not usually considered good programming practice.

Getting Help on Program Components

Although the Python website at *www.python.org* has complete documentation for the Python language, help on most language components is also readily available within the Python shell. To access such help, just enter the function call **`help(<component>)`** at the shell prompt, where **`<component>`** is the name of a module, data type, function, or method. For example, **`help(abs)`** and **`help(math.sqrt)`** display documentation for the **`abs`** and **`math.sqrt`** functions, respectively. Calls of **`dir(int)`** and **`dir(math)`** list all the operations in the **`int`** type and **`math`** module, respectively. You can then run **`help`** to get help on one of these operations.