Preface: How to Use the Instructor's Manual

The Instructor's Manual for ***Fundamentals of Python: First Programs*** is created to enrich a one-semester course on the fundamental concepts and techniques of Python. Ideas for teaching are presented in a chapter-by-chapter format. Each section includes:

- Lecture and Class Presentation Ideas
- Teaching Tips
- Quick Quizzes
- Topics for Class Discussion
- Additional Student Projects
- Additional Resources
- Key Terms

The lecture ideas closely follow the format of the book and provide a range of tips to enhance your classroom presentation. Discussion topics are meant to be thought-provoking and may require your students to solve a problem or work through a new direction in computer ethics. Quiz questions, student projects, additional Internet resources, and key terms give you options to direct each student's hands-on learning assignments.

Please also review the sample syllabus (included as a separate file) for additional suggestions to help you design and structure your course.

We hope that you have an engaging course experience and welcome your feedback on this manual or on any of our titles. Thank you for selecting our book.

Best wishes,
Course Technology Computer Science Team
computerscience@cengage.com

Chapter 1

Introduction

At a Glance

Instructor's Manual Table of Contents

Lecture Notes

Overview

Chapter 1 describes the basic features of an algorithm. Students learn how hardware and software collaborate in a computer's architecture. A brief history of computing is provided. Finally, students learn how to compose and run a simple Python program.

Objectives

After completing this chapter, students will be able to:

- Describe the basic features of an algorithm
- Explain how hardware and software collaborate in a computer's architecture
- Give a brief history of computing
- Compose and run a simple Python program

Teaching Tips

Two Fundamental Ideas of Computer Science: Algorithms and Information Processing

1. Explain that computer science focuses on a broad set of interrelated ideas; two of the most basic of these ideas are algorithms and information processing.

Algorithms

1. Introduce a simple algorithm in class. You may use the subtraction algorithm provided in the book, a cooking recipe, or any other sequence of instructions that your students can relate to. Stress that when these instructions are carried out, the *computing agent* is a human being.

Teaching Tip	For more information on the history of algorithms, ask your students to visit http://www.scriptol.com/programming/algorithm-history.php.

2. Provide a formal definition of the term *algorithm*.

3. List and explain the four features of an algorithm, specified on Pages 3 and 4 of the text.

Information Processing

1. Describe the role of information processing, introducing the terms *data*, *input* and *output*.

The Structure of a Modern Computer System

1. Explain that a modern computer system consists of *hardware* and *software*. Note that software is represented as *programs* in particular *programming languages*.

Computer Hardware

1. Use Figure 1.1 to describe the role of each of the most important hardware components of a modern computer system. Note that not shown in the figure are the computer *ports*, which enable computers to connect to *networks* and to other devices.

2. Use Figure 1.2 to help explain how the computer's *primary* or *internal* or *RAM* memory works, explaining that information is stored as patterns of *binary digits*.

3. Explain the concept of a *processor*, or CPU, and how it is structured to perform simple operations.

4. Explain the difference between internal and *external* or *secondary memory*. Note that examples of the latter are *magnetic media*, *semiconductor media*, and *optical media*, and give examples of each of these media.

Computer Software

1. Explain the concept of computer software as a program stored in memory that can be executed later. Point out that software is stored in *machine code* and is loaded into the memory by a *loader*.

2. Explain the concept of *system software*, specifying that the most important example of system software is the *operating system*, which includes a *file system*, one or more *user interfaces*, a *terminal-based interface*, and sometimes also a *graphical user interface* (*GUI*).

3. Explain the concept of *application software*, specifying how it is created. Be sure to introduce the terms *high-level programming languages*, *text editor*, *translator*, *syntax errors*, *run-time system*, *interpreter*, and *virtual machine.*

Teaching Tip	For more on the history of computer software, visit http://www.computerhistory.org/timeline/?category=sl.

4. Use Figure 1.3 to describe the role of the computer software used during the coding process.

Quick Quiz 1

1. What is an algorithm?
   Answer: Informally, an algorithm is like a recipe. It provides a set of instructions that tells us how to do something, such as make change, bake bread, or put together a piece of furniture. More precisely, an algorithm describes a process that ends with a solution to a problem.

2. The part of a computer that is responsible for processing data is the _____ (CPU).
   Answer: central processing unit

3. True or False: A program stored in computer memory must be represented in binary digits, which is also known as machine code.
   Answer: True

4. A modern _____ (GUI) organizes the monitor screen around the metaphor of a desktop, with windows containing icons for folders, files, and applications.
   Answer: graphical user interface

A Not-So-Brief History of Computing Systems

1. Use Figure 1.4 to provide a brief overview of some of the major developments in the history of computing.

Before Electronic Digital Computers

1. Briefly note some of the major developments in the history of computing before 1940. Some important names to mention are: Pascal, Leibnitz, Jacquard, Babbage, Hollerith, Boole, and Turing.

The First Electronic Digital Computers (1940–1950)

1. Briefly note some of the major developments in the history of computing that took place during the 1940s. Note that a few *mainframe computers* were developed in this period (e.g., Mark I, ENIAC, ABC, Colossus).

2. Stress that John von Neumann developed the first memory-stored programs.

The First Programming Languages (1950–1965)

1. Describe the evolution of the first programming languages, from *assembly languages* to *high-level programming languages* like FORTRAN, LISP, and COBOL. Be sure to explain the terms *keypunch machine*, *card reader*, *assembler*, *compiler*, *interpreter*, *artificial intelligence*, and *abstraction*.

Teaching Tip	For a brief history of computer languages and their evolution, visit http://www.scriptol.org/history.php.

Integrated Circuits, Interaction, and Timesharing (1965–1975)

1. Explain how the invention of the *transistor*, and later of the *integrated circuit*, led to the construction of smaller, faster, less expensive hardware components.

2. Introduce Moore's Law.

Teaching Tip	You can find a graph that illustrates Moore's Law at http://arstechnica.com/hardware/news/2008/09/moore.ars.

3. Note that computer processing evolved from *batch processing* to *time-sharing* to *concurrent processing*.

Personal Computing and Networks (1975–1990)

1. Describe the contribution of Engelbart to the creation of the first personal computers.

2. Note some other important developments, such as the Altair, the MS-DOS, the Ethernet, and ARPANET's evolution into the Internet.

Consultation, Communication, and Ubiquitous Computing (1990–Present)

1. Note some of the major developments of modern computing, like *optical storage media*, *virtual reality*, and *hypermedia*.

2. Provide a brief overview of the World Wide Web, making sure to explain the terms *Web server*, *Web browser*, and *Web client*.

Quick Quiz 2

1. In the late 1930s, _____, a mathematician and electrical engineer at M.I.T., wrote a classic paper titled "A Symbolic Analysis of Relay and Switching Circuits."
   Answer: Claude Shannon

2. True or False: In the early 1970s, computer scientists realized that a symbolic notation could be used instead of machine code, and the first assembly languages appeared.
   Answer: False

3. What is Moore's Law?
   Answer: This prediction states that the processing speed and storage capacity of hardware will increase and its cost will decrease by approximately a factor of 2 every 18 months.

4. By the mid 1980s, the _____ had grown into what we now call the Internet, connecting computers owned by large institutions, small organizations, and individuals all over the world.
   Answer: ARPANET

Getting Started with Python Programming

1. Note that Guido van Rossum invented the Python programming language in the early 1990s.

2. Explain that Python is a high-level, general-purpose programming language for solving problems on modern computer systems.

Teaching Tip	Python is an interpreted language. Are your students familiar with other interpreted languages? Ask them to discuss their experiences in class.

Running Code in the Interactive Shell

1. Use Figure 1.6 and one or more real examples to show students how to write and run code in the Python Shell.

2. Demonstrate to students how they can get help through the command prompt or through the drop-down menu in the Python shell.

3. Explain and demonstrate how to quit the Python shell.

Input, Processing, and Output

1. Explain the concept of programming language syntax as it applies to Python.

2. Define the concept of a variable, and explain how one can be created to store user input.

3. Define the concept of type conversion functions, and explain how they can be used to make the user input suitable for the program's needs.

4. Use a few examples to show students how to perform simple input and output operations in Python.

Editing, Saving, and Running a Script

1. Explain the concept of a script as it applies to Python programs.

2. Use Figures 1.7 and 1.8 to explain how to edit, save, and run scripts in an IDLE window.

3. Define the concept of a program library, and explain how such libraries can be used.

Behind the Scenes: How Python Works

1. Use Figure 1.9 to describe how Python code is interpreted and executed in the *Python Virtual Machine* (*PVM*).

Teaching Tip	A good Python programming tutorial is available at http://docs.python.org/tut/.

Detecting and Correcting Syntax Errors

1. Explain what the *syntax* of a programming language is.

2. Use one or more examples to stress that when Python encounters a syntax error in a program, it halts execution with an error message.

Quick Quiz 3

1. The easiest way to open a Python shell is to launch the _____.
   Answer: IDLE

2. What is a shell prompt?
   Answer: A shell window contains an opening message followed by the special symbol >>>, called a shell prompt.

3. True or False: In Python, to begin the next output on the same line as the previous one, you can place a colon at the end of the earlier `print` statement.
   Answer: False

4. The term _____ refers to the rules for forming sentences in a language.
   Answer: syntax

Class Discussion Topics

1. Ask your students to talk about any previous programming experience they might have had. What other programming languages are they familiar with?

2. Students familiar with other programming languages may be surprised that Python provides an interactive shell that can be used to test simple statements and expressions.

Do they think other programming languages should provide similar functionality? Why or why not?

Additional Projects

1. Ask students to do some research and create a timetable that describes the evolution of computer languages from machine language to the present day. The table does not have to be comprehensive, but it should include approximately ten major languages. For each language, they should include the creation date and author(s), a brief description, and the state of its use today.

2. Ask your students how they would like their final class grade to be determined in terms of the relative weight given to projects, weekly assignments, midterm, final, etc. Then, ask them to design an algorithm to calculate their final grade using the percentages they have chosen.

Additional Resources

1. History and evolution of computer languages:
   www.scriptol.org/history.php

2. Algorithm:
   http://www.wisegeek.com/what-is-an-algorithm.htm

3. Software:
   http://www.wordiq.com/definition/Software

4. Python Tutorial:
   http://docs.python.org/tut/

Key Terms

- **abacus:** An early computing device that allowed users to perform simple calculations by moving beads along wires.
- **abstraction:** A simplified view of a task or data structure that ignores complex detail.
- **algorithm:** A finite sequence of instructions that, when applied to a problem, will solve it.
- **Analytical Engine:** A general-purpose computer designed in the nineteenth century by Charles Babbage, but never completed by him.
- **application software:** Programs that allow human users to accomplish specialized tasks, such as word processing or database management.
- **argument:** A value or expression passed in a function or method call.
- **artificial intelligence:** A field of computer science whose goal is to build machines that can perform tasks that require human intelligence.
- **assembler:** A program that translates an assembly language program to machine code.

- **assembly language:** A computer language that allows the programmer to express operations and memory addresses with mnemonic symbols.
- **batch processing:** The scheduling of multiple programs so that they run in sequence on the same computer.
- **binary digit:** A digit, either 0 or 1, in the binary number system. Program instructions are stored in memory using a sequence of binary digits. See also bit.
- **bit:** A binary digit.
- **bitmap:** A data structure used to represent the values and positions of points on a computer screen or image.
- **bit-mapped display screen:** A type of display screen that supports the display of graphics and images.
- **byte:** A sequence of bits used to encode a character in memory.
- **byte code:** The kind of object code generated by a Python compiler and interpreted by a Python virtual machine. Byte code is platform independent.
- **card reader:** A device that inputs information from punched cards into the memory of a computer.
- **central processing unit (CPU):** A major hardware component that consists of the arithmetic/logic unit and the control unit. Also sometimes called a **processor**.
- **coding:** The process of writing executable statements that are part of a program to solve a problem. See also implementation.
- **compiler:** A computer program that automatically converts instructions in a high-level language to byte code or machine language.
- **computing agent:** The entity that executes instructions in an algorithm.
- **concurrent processing:** The simultaneous performance of two or more tasks.
- **data:** The symbols that are used to represent information in a form suitable for storage, processing, and communication.
- **execute:** To carry out the instructions of a program.
- **external (or secondary) memory:** Memory that can store large quantities of data permanently.
- **file system:** Software that organizes data on secondary storage media.
- **GUI (graphical user interface):** A means of communication between human beings and computers that uses a pointing device for input and a bitmapped screen for output. The bitmap displays images of windows and window objects such as buttons, text fields, and drop-down menus. The user interacts with the interface by using the mouse to directly manipulate the window objects. See also window object.
- **hardware:** The computing machine and its support devices.
- **high-level programming language:** Any programming language that uses words and symbols to make it relatively easy to read and write a program. See also assembly language and machine language.
- **HTML (HyperText Markup Language):** A programming language that allows the user to create pages for the World Wide Web.
- **hypermedia:** A data structure that allows the user to access different kinds of information (text, images, sound, video, applications) by traversing links.
- **hypertext:** A data structure that allows the user to access different chunks of text by traversing links.
- **identifiers:** Words that must be created according to a well-defined set of rules but can have any meaning subject to these rules.
- **IDE (integrated development environment):** A set of software tools that allows you to edit, compile, run, and debug programs within one user interface.

- **information processing:** The transformation of one piece of information into another piece of information.
- **input:** Data obtained by a program during its execution.
- **input device:** A device that provides information to the computer. Typical input devices are a mouse, keyboard, disk drive, microphone, and network port. See also I/O device and output device.
- **integrated circuit:** The arrangement of computer hardware components in a single, miniaturized unit.
- **internal memory:** See main memory
- **interpreter:** A program that translates and executes another program.
- **I/O device:** Any device that allows information to be transmitted to or from a computer. See also input device and output device.
- **keypunch machine:** An early input device that allowed the user to enter programs and data onto punched cards.
- **library:** A collection of methods and data organized to perform a set of related tasks.
- **linear:** An increase of work or memory in direct proportion to the size of a problem.
- **loader:** A system software tool that places program instructions and data into the appropriate memory locations before program start-up.
- **machine code:** The language used directly by the computer in all its calculations and processing. Also called machine language.
- **magnetic storage media:** Any media that allow data to be stored as patterns in a magnetic field.
- **main (primary or internal) memory:** The high-speed internal memory of a computer, also referred to as random access memory (RAM). See also memory and secondary memory.
- **mainframe:** Large computers typically used by major companies and universities.
- **memory:** The ordered sequence of storage cells that can be accessed by address. Instructions and variables of an executing program are temporarily held here. See also main memory and secondary memory.
- **memory location:** A storage cell that can be accessed by address.
- **microcomputer:** A computer capable of fitting on a laptop or desktop, generally used by one person at a time.
- **Moore's Law:** A hypothesis that states that the processing speed and storage capacity of computers will increase by a factor of two every 18 months.
- **network:** A collection of resources that are linked together for communication.
- **newline character:** A special character ('\n') used to indicate the end of a line of characters in a string or a file stream.
- **operating system:** A large program that allows the user to communicate with the hardware and performs various management tasks.
- **optical storage media:** Devices such as CDs and DVDs that store data permanently and from which the data are accessed by using laser technology.
- **output:** Information that is produced by a program.
- **output device:** A device that allows you to see the results of a program. Typically, it is a monitor, printer, speaker, or network port.
- **port:** A channel through which several clients can exchange data with the same server or with different servers.
- **primary memory:** See main memory
- **processor:** See Central Processing Unit (CPU)
- **program:** A set of instructions that tells the machine (the hardware) what to do.

- **program library:** A collection of operations and data organized to perform a set of related tasks.
- **programming language:** A formal language that computer scientists use to give instructions to the computer.
- **PVM (Python Virtual Machine):** A program that interprets Python byte codes and executes them.
- **RAM (random access memory):** High-speed memory where programs and their data reside during program execution.
- **run-time system:** Software that supports the execution of a program.
- **script:** A Python program that can be launched from a computer's operating system.
- **secondary (external) memory:** An auxiliary device for memory, usually a disk or magnetic tape.
- **semiconductor storage media:** Devices, such as flash sticks, that use solid state circuitry to store data permanently.
- **shell:** A program that allows users to enter and run Python program expressions and statements interactively.
- **software:** Programs that make the machine (the hardware) do something, such as word processing, database management, or games.
- **software reuse:** The process of building and maintaining software systems out of existing software components.
- **solid-state device:** An electronic device, typically based on a transistor, which has no moving parts.
- **source code:** The program text as viewed by the human being who creates or reads it, prior to compilation.
- **statement:** An individual instruction in a program.
- **syntax:** The rules for constructing well-formed programs in a language. Also, the rules for forming sentences in a language.
- **syntax error:** An error in spelling, punctuation, or placement of certain key symbols in a program. See also compilation error, design error, and run time error.
- **system software:** The programs that allow users to write and execute other programs, including operating systems such as Windows and Mac OS.
- **Terminal-based interface:** A user interface that allows the user to enter input from a keyboard and view output as text in a window. Also called a terminal-based interface.
- **text editor:** A program that allows the user to enter text, such as a program, and save it in a file.
- **time sharing:** The scheduling of multiple programs so that they run concurrently on the same computer.
- **time-sharing operating system:** A computer system that can run multiple programs in such a manner that its users have the illusion that they are running simultaneously.
- **transistor:** A device with no moving parts that can hold an electromagnetic signal and that is used to build computer circuitry for memory and a processor.
- **translator:** A program that converts a program written in one language to an equivalent program in another language.
- **type conversion functions:** A function that takes one type of data as an argument and returns the same data represented in another type.
- **user interface:** The part of a software system that handles interaction with users.
- **variable:** A memory location, referenced by an identifier, whose value can be changed during execution of a program.
- **variable identifier:** A name used to reference a variable.

- **virtual machine:** A software tool that behaves like a high-level computer.
- **virtual reality:** A technology that allows a user to interact with a computer-generated environment, usually simulating movement in three dimensions.
- **Web browser:** Software used to view information on the Web.
- **Web client:** Software on a user's computer that makes requests for resources from the Web.
- **Web server:** Software on a computer that responds to requests for resources and makes them available on the Web.
- **window:** A rectangular area of a computer screen that can contain window objects. Windows typically can be resized, minimized, maximized, zoomed, or closed.

Chapter 2

Software Development, Data Types, and Expressions

At a Glance

Instructor's Manual Table of Contents

Lecture Notes

Overview

Chapter 2 covers topics related to software development, data types, and expressions. Students first learn about the basic phases of the software development process. Next, they learn how to use strings, integer, and floating point numbers in Python. Simple as well as mixed-mode arithmetic expressions are covered. Students will also learn how to initialize and use variables with appropriate names. Students learn about modules, how to import functions from library modules, and how to call functions with arguments and use the returned values appropriately. Finally, they learn how to construct a simple Python program that performs inputs, calculations, and outputs, as well as how to use docstrings to document Python programs.

Objectives

After completing this chapter, students will be able to:

- Describe the basic phases of software development: analysis, design, coding, and testing
- Use strings for the terminal input and output of text
- Use integers and floating point numbers in arithmetic operations
- Construct arithmetic expressions
- Initialize and use variables with appropriate names
- Import functions from library modules
- Call functions with arguments and use returned values appropriately
- Construct a simple Python program that performs inputs, calculations, and outputs
- Use docstrings to document Python programs

Teaching Tips

The Software Development Process

1. Explain that there are several approaches to *software development*, the *waterfall model* being one of them.

2. Use Figure 2.1 to describe the role of each of the phases of the waterfall model.

3. Stress that modern software development is usually *incremental* and *iterative*. Introduce the term *prototype*.

4. Note that programs rarely work as hoped the first time they are run. Use Figure 2.2 to stress that it is important to perform extensive and careful testing in all of the development phases.

5. Use Figure 2.3 to explain that the cost of developing software is not spread equally over the phases.

Teaching Tip	For more information on the software development process, visit http://www.selectbs.com/analysis-and-design/what-is-a-software-development-process.

Case Study: Income Tax Calculator

1. Guide students as they step through this case study.

Request

1. Explain that the customer requests a program that computes a person's income tax.

Analysis

1. Stress that the inclusion of an interface at this point (see Figure 2.4) is a good idea because it allows the customer and the programmer to discuss the program's intended behavior in a context understandable to both.

Design

1. Ask your students to try to code the algorithm before they take a look at the code provided in the next section.

2. Explain the concept of *pseudocode* and how it can be used to create a design regardless of the specific programming language that will be used for implementation.

Implementation (Coding)

1. Explain that pseudocode and other design elements make it easier for a programmer to create a program that carries out the required computation. Pseudocode also allows for greater coding accuracy.

2. Stress that it is always important to provide descriptive comments to document a program.

Testing

1. Did the program work as expected the first time? Ask students to list the errors they had (if any) and describe how they fixed them.

2. Explain the purpose of testing, making sure to stress the importance of thorough testing and of planning the testing process.

3. Explain how to test a program using the IDLE window.

4. Explain the difference between *syntax errors* and *logic* or *design errors*. Explain the term *correct program*, distinguishing it from a program with logic or design errors.

5. Explain the purpose of a *test suite*, and how it represents all inputs. Use Table 2.1 to demonstrate the creation of a test suite and to explain how to determine what values should be included in a test suite for a specific program.

Strings, Assignment, and Comments

1. Use this section to introduce the use of strings for the output of text and the documentation of Python programs.

Data Types

1. Introduce the terms *data type*, *literal*, and *numeric data type*, emphasizing the differences and relationship between them. Table 2.2 lists some Python data types.

String Literals

1. Explain the term string literal and use real examples to illustrate how to create string literals in Python.

2. Introduce the term *empty string*, and be sure to distinguish the empty string from a string containing only white spaces.

Teaching Tip	For more information about string literals, visit http://docs.python.org/ref/strings.html.

3. Show students how to create strings that span multiple-lines, and introduce them to the *newline character*, \n.

Escape Sequences

1. Define the term *escape sequence*, and use Table 2.3 to introduce some of the most useful escape sequences in Python.

String Concatenation

1. Using one or more examples, demonstrate how to perform string concatenation in Python using the + operator.

2. Explain that the * operator allows you to build a string by repeating another string a given number of times.

Variables and the Assignment Statement

1. Remind students the meaning of the term *variable*.

2. Explain the naming rules that apply to variables in Python. Provide several examples to illustrate each of these rules.

Teaching Tip	CamelCase is a standard identifier naming convention for several programming languages. For more information, visit http://www.c2.com/cgi/wiki?CamelCase.

3. Introduce the term *symbolic constant*, and note that programmers usually use all uppercase letters to name symbolic constants.

4. Use one or more examples to show how to write an *assignment statement* in Python. Introduce the term *variable reference* and explain the difference between *defining* or *initializing* a variable and variable references.

5. Explain the purposes variables have in a program. Be sure to explain the term *abstraction* in this context.

Program Comments and Docstrings

1. Explain the purpose and importance of *program comments*.

2. Provide examples of how to include *docstrings* and *end-of-line* comments in Python. Stress that using these types of comments appropriately is important.

3. Review the guidelines for creating good documentation of code. You can use the list on Page 53 of the text as a guide.

Teaching Tip	The pydoc module can be used to display information about a Python object, including its docstring. For more information, visit http://epydoc.sourceforge.net/docstrings.html.

Quick Quiz 1

1. Modern software development is usually incremental and _____.
   Answer: iterative

2. What is a correct program?
   Answer: A correct program produces the expected output for any legitimate input.

3. True or False: In programming, a data type consists of a set of values and a set of operations that can be performed on those values.
   Answer: True

4. The newline character _____ is called an escape sequence.
   Answer: \n

Numeric Data Types and Character Sets

1. Explain that the following section provides a brief overview of numeric data types and their cousins, character sets.

Integers

1. Explain how integer literals are written in Python.

Floating-Point Numbers

1. Explain that real numbers have infinite precision, and point out that this is not possible on a computer due to memory limits.

2. Use Table 2.4 to introduce the use of *floating-point* numbers in Python. Explain that these numbers can be written using *decimal notation* or *scientific notation*.

Teaching Tip	Complex numbers are also supported in Python. For more information, visit http://docs.python.org/library/stdtypes.html.

Character Sets

1. Explain that in Python, character literals look just like string literals and are of the string type. Point out that character literals belong to several different *character sets*, among them the *ASCII set* and the *Unicode set*.

2. Provide a brief overview of the ASCII character set, using Table 2.5 as a guide.

Teaching Tip	You can find more information about Python's Unicode support at http://www.amk.ca/python/howto/unicode.

3. Use a few examples to show how to convert characters to and from ASCII using the `ord` and `chr` functions.

Expressions

1. Explain that *expressions* provide an easy way to perform operations on data values to produce other values.

2. Note that when entered at the Python shell prompt, an expression's operands are evaluated first. The operator is then applied to these values to compute the final value of the expression.

Arithmetic Expressions

1. Introduce the term *arithmetic expression*. Use Table 2.6 to describe the arithmetic operators available in Python.

2. Explain the difference between binary operators and unary operators. Give examples of each type of operator.

3. Briefly list the *precedence rules* that apply to arithmetic operators, using the list on Pages 58-59 as a guide. Be sure to explain the meaning of the terms *left associative* and *right associative*, and to point out that you can use parentheses to change the order of evaluation in an arithmetic expression.

4. Use Table 2.7 to show a few examples of how arithmetic expressions are evaluated. Introduce the term *semantic error* and clearly explain the difference between a semantic error and a syntax error.

5. Stress that when both operands of an expression are of the same numeric type, the resulting value is of that type; when each operand is of a different type, the resulting value is of the more general type.

6. Explain how the backslash character \ can be used to break an expression onto multiple lines.

Mixed-Mode Arithmetic and Type Conversions

1. Use a few examples to show how *mixed-mode arithmetic* can be problematic. Show how to use a *type conversion function* (see Table 2.8) to solve this problem.

2. Stress that the `int` function converts a `float` to an `int` by truncation, not by rounding. Show how to use the `round` function in cases when rounding is desirable.

3. Use one or more examples to show that type conversion also occurs in the construction of strings from numbers and other strings. Explain how to use the `str` function to solve this problem.

4. Explain that Python is a *strongly typed programming language*.

Quick Quiz 2

1. Real numbers have _____ precision, which means that the digits in the fractional part can continue forever.
   Answer: infinite

2. True or False: In the 1960s, the original ASCII set encoded each keyboard character and several control characters using the integers from 0 through 255.
   Answer: False

3. True or False: The precedence rules you learned in algebra apply during the evaluation of arithmetic expressions in Python.
   Answer: True

4. Exponentiation and assignment operations are right _____, so consecutive instances of these are evaluated from right to left.
   Answer: associative

Using Functions and Modules

1. Explain that Python includes many useful functions, which are organized in libraries of code called *modules*.

Teaching Tip	Python's global module index is available at http://docs.python.org/modindex.html.

Teaching Tip	"A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module is imported somewhere." *Reference:* http://docs.python.org/tut/node8.html

Calling Functions: Arguments and Return Values

1. Introduce the terms *function*, *argument/parameter* (*optional* and *required*), and *returning a value*.

2. Define the concept of a *default behavior* of a function, and explain how this may be changed by calling the function using optional arguments.

3. Use an example to show how to obtain more information about a function by using `help`.

The `math` Module

1. Show how to use functions in the `math` module, both by importing the whole module and by importing individual resources.

The Main Module

1. Explain that, like any module, the *main module* can be imported. Show how this is equivalent to importing a Python script as a module.

2. Use the example provided in the book to show how to import the main module created in the case study of this chapter.

Program Format and Structure

1. Provide some guidance on how a typical Python program should look. Use the bullet points on Pages 67-68 as a guide.

Running a Script from a Terminal Command Prompt

1. Use Figures 2.6 and 2.7 to show how to run a script from a terminal command prompt.

2. Note that Python installations enable you to launch Python scripts by double-clicking the files from the OS's file browser. Explain what the fly-by-window problem is, and how to solve it: add an input statement at the end of the script that pauses until the user presses the ENTER or RETURN key.

Quick Quiz 3

1. What is a module?
   Answer: Python includes many useful functions, which are organized in libraries of code called modules.

2. A(n) _____ is a chunk of code that can be called by name to perform a task.
   Answer: function

3. True or False: Arguments are also known as literals.
   Answer: False

4. The statement _____ would import all of the `math` module's resources.
   Answer: `from math import *`

Class Discussion Topics

1. Are your students familiar with other software development models? If so, ask them to discuss their similarities and differences with the waterfall model. Which one do they like best?

2. Have your students used functions in other languages?

Additional Projects

1. Ask students to do some research and find out what other escape sequences are available in Python. They should compile a list of each of these with a short description of their functions.

2. Python provides other useful modules besides the `math` module. Ask students to do some research and find one or two other modules that they think will be useful in future projects. They should provide a brief description of each module, and include a list of two to five functions that belong to each module.

Additional Resources

1. String Literals:
   http://docs.python.org/ref/strings.html

2. Python Docstrings:
   http://epydoc.sourceforge.net/docstrings.html

3. Unicode HOWTO:
   http://www.amk.ca/python/howto/unicode

4. Global Module Index:
   http://docs.python.org/modindex.html

5. Module tutorial:
   http://docs.python.org/tutorial/modules.html

Key Terms

- **abstraction:** A simplified view of a task or data structure that ignores complex detail.
- **analysis:** The phase of the software life cycle in which the programmer describes what the program will do.
- **Argument(s):** A value or expression passed in a method call.
- **arithmetic expression:** A sequence of operands and operators that computes a value.

- **ASCII character set:** The American Standard Code for Information Interchange ordering for a character set.
- **assignment operator:** The symbol =, which is used to give a value to a variable.
- **assignment statement:** A method of giving values to variables.
- **character set:** The list of characters available for data and program statements.
- **coding:** The process of writing executable statements that are part of a program to solve a problem. See also implementation.
- **comments:** Nonexecutable statements used to make a program more readable.
- **concatenation:** An operation in which the contents of one data structure are placed after the contents of another data structure.
- **correct program:** A program that produces an expected output for any legitimate input.
- **data type(s):** A set of values and operations on those values.
- **default behavior:** Behavior that is expected and provided under normal circumstances.
- **design:** The phase of the software life cycle in which the programmer describes how the program will accomplish its tasks.
- **design error:** An error such that a program runs, but unexpected results are produced. Also referred to as a logic error.
- **docstring:** A sequence of characters enclosed in triple quotation marks (“““) that Python uses to document program components such as modules, classes, methods, and functions.
- **empty string:** A string that contains no characters.
- **end-of-line comment:** Part of a single line of text in a program that is not executed, but serves as documentation for readers.
- **escape sequence:** A sequence of two characters in a string, the first of which is /. The sequence stands for another character, such as the tab or newline.
- **expression:** A description of a computation that produces a value.
- **float:** A Python data type used to represent numbers with a decimal point, for example, a real number or a floating-point number.
- **floating-point number:** A data type that represents real numbers in a computer program.
- **function(s):** A chunk of code that can be treated as a unit and called to perform a task.
- **Implementation:** The phase of the software life cycle in which the program is coded in a programming language.
- **integer:** A positive or negative whole number, or the number 0. The magnitude of an integer is limited by a computer’s memory.
- **integer arithmetic operations:** Operations allowed on data of type `int`. These include the operations of addition, subtraction, multiplication, division, and modulus to produce integer answers.
- **left associative:** The property of an operator such that repeated applications of it are evaluated from left to right (first to last).
- **literal:** An element of a language that evaluates to itself, such as 34 or “hi there.”.
- **logic error:** See design error
- **main module:** The software component that contains the point of entry or start-up code of a program.
- **mixed-mode arithmetic:** Expressions containing data of different types; the values of these expressions will be of either type, depending on the rules for evaluating them.
- **module(s):** An independent program component that can contain variables, functions, and classes.

- **newline character:** A special character (‘\n’) used to indicate the end of a line of characters in a string or a file stream.
- **optional arguments:** Arguments to a function or method that may be omitted.
- **Parameter(s):** See argument(s)
- **precedence rules:** Rules that govern the order in which operators are applied in expressions.
- **prototype:** A trimmed-down version of a class or software system that still functions and allows the programmer to study its essential features.
- **pseudocode:** A stylized half-English, half-code language written in English but suggesting program code.
- **required arguments:** Arguments that must be supplied by the programmer when a function or method is called.
- **returning a value:** The process whereby a function or method makes the value that it computes available to the rest of the program.
- **scientific notation:** The representation of a floating-point number that uses a decimal point and an exponent to express its value.
- **semantic error:** A type of error that occurs when the computer cannot carry out the instruction specified.
- **semantics:** The rules for interpreting the meaning of a program in a language.
- **software development life cycle (SDLC):** The process of development, maintenance, and demise of a software system. Phases include analysis, design, coding, testing/verification, maintenance, and obsolescence.
- **string(s) (string literals):** One or more characters, enclosed in double quotation marks, used as a constant in a program.
- **strongly-typed programming language:** A language in which the types of operands are checked prior to applying an operator to them, and which disallows such applications, either at run time or at compile time, when operands are not of the appropriate type.
- **symbolic constant:** A name that receives a value at program start-up and whose value cannot be changed.
- **test suite:** A set of test cases that exercise the capabilities of a software component.
- **type conversion function:** A function that takes one type of data as an argument and returns the same data represented in another type.
- **Unicode:** A character set that uses 16 bits to represent over 65,000 possible characters. These include the ASCII character set as well as symbols and ideograms in many international languages.
- **variable:** A memory location, referenced by an identifier, whose value can be changed during execution of a program.
- **variable reference:** The process whereby the computer looks up and returns the value of a variable.
- **waterfall model:** A series of steps in which a software system trickles down from analysis to design to implementation. See also software development life cycle.

Chapter 3

Control Statements

At a Glance

Instructor's Manual Table of Contents

Lecture Notes

Overview

Chapter 3 describes the different control statements available in Python. Students learn how to write `for` and `while` loops; students then learn to use these loops to repeat actions, traverse lists, count up and down, and generate random numbers. Selection statements are also covered, including one-way selection statements (`if`), two-way selection statements (`if-else`), and multi-way selection statements.

Objectives

After completing this chapter, students will be able to:

- Write a loop to repeat a sequence of actions a fixed number of times
- Write a loop to traverse the sequence of characters in a string
- Write a loop that counts down and a loop that counts up
- Write an entry-controlled loop that halts when a condition becomes false
- Use selection statements to make choices in a program
- Construct appropriate conditions for condition-controlled loops and selection statements
- Use logical operators to construct compound Boolean expressions
- Use a selection statement and a break statement to exit a loop that is not entry-controlled

Teaching Tips

Definite Iteration: The `for` Loop

1. Introduce the terms: *loop*, *pass*/*iteration*, *definite iteration*, and *indefinite iteration*.

Teaching Tip	For more information on Python loops, visit: http://en.wikibooks.org/wiki/Python_Programming/Loops.

Executing a Statement a Given Number of Times

1. Note that Python's *`for` loop* is the control statement that most easily supports definite iteration.

2. Describe the syntax of a `for` loop; identify its header and body and explain the structure of each. Stress that the statements in the loop body must be indented and aligned in the same column.

3. Use a few examples to show how to write a `for` loop.

Count-Controlled Loops

1. Use a few examples to show how to create count-controlled `for` loops.

2. Point out that the second (or only) argument of `range` should be greater by one than the desired upper bound of the count.

3. Give examples of how user input can be used to determine the for loop range.

Augmented Assignment

1. Explain the term *augmented assignment operations* and provide examples of such operations. Stress that the augmented assignment operators have the same precedence as the assignment operator.

Loop Errors: Off-by-One Error

1. Explain the term *off-by-one error*, pointing out that these are logic errors and not syntax errors. Note that this type of error is one of the most common types of errors in `for` loops. Use an example to illustrate this type of error.

Traversing the Contents of a Data Sequence

1. Explain that `for` loops can iterate over any sequence of elements, not only a numeric sequence. Provide a few examples to show how to write `for` loops that traverse list elements and characters in a string.

Specifying the Steps in the Range

1. Describe how a third variant of the `range` function can be used to specify a *step value*, and provide one or more examples to show how this kind of `for` loop is written.

Loops That Count Down

1. Use an example to note that one can provide a negative third argument to the `range` function in order to create loops that count down.

Formatting Text for Output

1. Use the examples provided in the book to show how to specify a *field width* and nicely display output in *tabular format*.

2. Describe the roles of *format strings* and the *format operator* (%), and explain how to use them to format strings, integers, and floating-point numbers.

Teaching Tip	You can find some useful information about string formatting operations in Python at: http://docs.python.org/lib/typesseq-strings.html, section 5.6.2.

Case Study: An Investment Report

1. Guide students as they step through this section.

Request

1. Explain that students are requested to write a program that computes an investment report.

Analysis

1. Provide an overview of the input the program should receive (as listed in the bullet points on Page 87) and the values the program should compute.

2. Use Figure 3.1 to describe the output that the program should generate. Can students figure out how will they use the format operator?

Design

1. Explain to students the four principal parts of the program.

2. Point out that the pseudocode for this algorithm is very simple—students should only look at the pseudocode provided in the book to check their own design.

3. Explain the term *prototype* as it applies to designing a program.

Implementation (Coding)

1. Review the code provided in the book with students. Point out the descriptive choices of variable names, as well as the end-of-line comments explaining the various computation stages.

2. Make sure students understand how the format operator (%) is used in the code.

Testing

1. Explain to students that when testing a program that contains a loop, they must first make sure that the number of iterations of the loop is correct.

2. Point out that the second phase of testing is to examine the effect of different inputs on the results, including the format of the results.

3. Show students the sample data set provided in Table 3.1, and explain why this is a good data set.

Quick Quiz 1

1. What is an iteration?
   Answer: In a loop, each repetition of the action is known as a pass or an iteration.

2. True or False: When two arguments are supplied to `range`, the count ranges from the first argument to the second argument minus 1.
   Answer: True

3. True or False: The third argument of the `range` function specifies the upper bound of the count.
   Answer: False

4. The total number of data characters and additional spaces for a given datum in a formatted string is called its _____.
   Answer: field width

Selection: `if` and `if-else` Statements

1. Explain that *selection statements* allow a computer to make choices based on a *condition*.

Teaching Tip	For more information on loops and conditionals in Python, read: www.sthurlow.com/python/lesson04/.

The Boolean Type, Comparisons, and Boolean Expressions

1. Explain that the *Boolean data type* consists of two values: true and false.

2. Point out that Boolean expressions can be created in a number of ways, including variables bound to Boolean values, function calls that return Boolean values, and comparisons.

3. Use Table 3.2 and a few examples to show how to use comparison operators to create Boolean expressions. Be sure to point out to students that to check whether two terms are equal you must use `==` and not `=`, which means assignment, and that use of the wrong operator is often a cause of logic errors in programs that contain comparisons.

4. Explain to students the order in which comparison operators are applied in complex expressions, pointing out the order of precedence of these operators.

`if-else` Statements

1. Describe the syntax of *`if-else` statements* and provide a few examples to demonstrate how to write *two-way selection statements*.

2. Use Figure 3.2 to describe the semantics of this statement, and stress that the condition must be a Boolean expression. Emphasize that each sequence of statements must be indented at least one space beyond the `if` and `else`.

One-Way Selection Statements

1. Use Figure 3.3 to describe the semantics of the `if` (*one-way selection*) statement.

2. Describe its syntax, and use a few examples to show how to use one-way selection statements.

Multi-Way `if` Statements

1. Use the grading example provided in the book (see Table 3.3) to show how to write *multi-way selection statements* in Python.

2. Explain the syntax of a multi-way `if` statement, emphasizing the use of the term `elif` when entering each alternate condition statement.

Logical Operators and Compound Boolean Expressions

1. Use the example provided in the book to show how to use *logical operators* to create *compound Boolean expressions*, which can then be used to write simpler selection statements.

2. Provide an overview of the *truth tables* for the `and`, `or`, and `not` logical operators, using Figure 3.4 as a guide.

3. Stress that the logical operators are evaluated after comparisons but before the assignment operator. Note that the `not` operator has higher precedence than `and` and `or` operators. Table 3.4 shows the operator precedence (from highest to lowest) of the arithmetic, logical, and assignment operators.

Short-Circuit Evaluation

1. Explain what *short-circuit evaluation* is, and use one or more examples to show how this feature can be useful for avoiding errors like division by zero.

Teaching Tip	For more information on short-circuit evaluation, read: http://www.freenetpages.co.uk/hp/alan.gauld/tutfctnl.htm.

Testing Selection Statements

1. Provide some tips on how to effectively test selection statements, including making sure to consider the program's behavior for all possible branches or alternatives of a selection and examining the conditions.

Quick Quiz 2

1. In Python, _____ literals can be written in several ways, but most programmers prefer the use of the standard values `True` and `False`.
   Answer: Boolean

2. What is a two-way selection statement?
   Answer: The `if-else` statement is the most common type of selection statement. It is also called a two-way selection statement, because it directs the computer to make a choice between two alternative courses of action.

3. True or False: Python includes all three Boolean or logical operators, `and`, `or`, and `xor`.
   Answer: False

4. True or False: There are times when short-circuit evaluation is advantageous.
   Answer: True

Conditional Iteration: The `while` Loop

1. Explain that the `while` loop can be used to describe conditional iteration. Introduce the term *sentinel*.

The Structure and Behavior of a `while` Loop

1. Describe the syntax of a `while` loop, identifying its *continuation condition*. Stress that improper setting of the continuation condition may lead to an infinite loop.

2. Use Figure 3.5 and an example to describe the semantics of a `while` loop. Introduce the term *loop control variable*.

3. Explain why a `while` loop is also called an *entry-control loop*.

Count Control with a `while` Loop

1. Use one or more examples to show how to create count control loops using a `while`.

2. Explain why using a while loop for a count controlled loop can potentially be a source of errors in loop logic.

The `while` True Loop and the `break` Statement

1. Use one or more examples to show how it is sometimes convenient to create `while True` loops in which the loop's *termination condition* is indicated using a selection statement within the loop.

2. Note that some computer scientists believe `while True` loops go against the nature of the `while` statement. Explain that an alternative to a `while True` loop is to use a Boolean variable (i.e., a flag) to control the loop.

Teaching Tip	For more information on the `break` statement, read: http://docs.python.org/ref/break.html.

Teaching Tip	Besides the `break` statement, `continue` statements are also useful in loops. For more information, visit: http://docs.python.org/ref/continue.html.

Random Numbers

1. Use the simple guessing game example provided in the book to show how to use the `random` module and the `randint` function to generate *random numbers*.

Teaching Tip	For more information on random number generators, visit: http://en.wikipedia.org/wiki/Random_number_generator.

Loop Logic, Errors, and Testing

1. Provide some tips on how to effectively test `while` loops.

2. Explain that the `Control+c` key combination can be used to halt a loop that appears to be infinite during testing.

Case Study: Approximating Square Roots

1. Guide students as they step through this section.

Request

1. Explain to students that they are asked to write a program that computes square roots.

Analysis

1. Describe the user interface of the program to be created.

Design

1. Spend some time explaining Newton's square root approximation algorithm.

Teaching Tip	For more information on Newton's and other square root approximation algorithms, visit: http://mathworld.wolfram.com/NewtonsMethod.html.

Implementation (Coding)

1. Show students the code for the program, going over each statement and paying specific attention to the `while` loop.

2. You may ask students to modify the program so that it counts how many iterations the loop performs. How does the number change if the tolerance is modified?

Testing

1. Explain the steps required to test this program: testing how the program functions for valid and invalid inputs, as well as using Python's own most accurate estimate of the square root provides a benchmark for assessing the correctness of our own algorithm.

Quick Quiz 3

1. What is conditional iteration?
   Answer: Conditional iteration requires that a condition be tested within the loop to determine whether the loop should continue.

2. The `while` loop is also called a(n) _____ loop, because its condition is tested at the top of the loop.
   Answer: entry-control

3. True or False: A `while` loop cannot be used for a count-controlled loop.
   Answer: False

4. A(n) _____ statement causes an exit from the loop that contains it.
   Answer: `break`

Class Discussion Topics

1. Are students familiar with the repeat-until loops available in some programming languages? If so, ask them to compare and contrast these types of loops with the `while` loop studied in this chapter.

2. Are students familiar with the `switch` (or `case`) selection statements available in some programming languages? If so, ask them if they think `switch` statements are more convenient than multi-way `if` statements (or the other way around).

Additional Projects

1. Ask your students to write a menu-driven program that calculates the total price for a picnic lunch that the user is purchasing for a group of friends. The user is first asked to enter her budget for the lunch. She has the option of buying apples, cheese, and bread. Set the price per apple, price per pound of cheese, and price per loaf of bread in constant variables. Use a nested repetition/selection structure to ask the user what type of item and how much of each item she would like to purchase. Keep a running total of the items purchased inside the loop. Exit the loop if the total has exceeded the user's budget. In addition, provide a sentinel value that allows the user to exit the purchasing loop at any time.

2. Ask students to write a program that uses a loop to calculate the *gcd* of two positive numbers using the Euclidean method. (http://mathworld.wolfram.com/EuclideanAlgorithm.html).

Additional Resources

1. Python Programming/Loops:
   http://en.wikibooks.org/wiki/Python_Programming/Loops

2. Python Tutorial: More Control Flow Tools:
   http://docs.python.org/tut/node6.html

3. A Beginner's Python Tutorial: Loops and Conditionals:
   www.sthurlow.com/python/lesson04/

4. The `break` Statement:
   http://docs.python.org/ref/break.html

Key Terms

- **augmented assignment:** An assignment operation that performs a designated operation, such as addition, before storing the result in a variable.

- **Boolean expression:** An expression whose value is either true or false. See also compound Boolean expression and simple Boolean expression.
- **compound Boolean expression:** Refers to the complete expression when logical connectives and negation are used to generate Boolean values. See also Boolean expression and simple Boolean expression.
- **continuation condition:** A Boolean expression that is checked to determine whether or not to continue iterating within a loop. If this expression is True, iteration continues.
- **control statement:** A statement that allows the computer to repeat or select an action.
- **count-controlled loop:** A loop that stops when a counter variable reaches a specified limit.
- **definite iteration:** The process of repeating a given action a preset number of times.
- **field width:** The number of columns used for the output of text
- **`for` loop:** A structured loop used to traverse a sequence.
- **format operator `%`:**
- **format string:** A special syntax within a string that allows the programmer to specify the number of columns within which data are placed in a string.
- **`if-else` statement:** A selection statement that allows a program to perform alternative actions based on a condition.
- **indefinite iteration:** The process of repeating a given action until a condition stops the repetition.
- **infinite loop:** A loop in which the controlling condition is not changed in such a manner to allow the loop to terminate.
- **Iteration:** See loop
- **logical operator:** Either of the logical connective operators `and`, `or`, or `not`.
- **loop(s):** A type of statement that repeatedly executes a set of statements. See also control statements.
- **loop body:** The action(s) performed on each iteration through a loop.
- **loop header:** Information at the beginning of a loop that includes the conditions for continuing the iteration process.
- **multi-way selection statement:** Code description of the process of testing several conditions and responding accordingly**.** See also extended if statement
- **off-by-one error:** Usually seen with loops, this error shows up as a result that is one less or one greater than the expected value.
- **prototype:** A trimmed down version of a class or software system that still functions and allows the programmer to study its essential features.
- **selection statement:** A control statement that selects some particular logical path based on the value of an expression. Also referred to as a conditional statement.
- **sentinel value (sentinel):** A special value that indicates the end of a set of data or of a process.
- **short-circuit evaluation:** The process by which a compound Boolean expression halts evaluation and returns the value of the first subexpression that evaluates to true, in the case of or, or false, in the case of and.
- **simple Boolean expression:** An expression in which two numbers or variable values are compared using a single relational operator. See also Boolean expression and compound Boolean expression.
- **step value:** The amount by which a counter is incremented or decremented in a count-controlled loop.
- **termination condition:** A Boolean expression that is checked to determine whether or not to stop iterating within a loop. If this expression is True, iteration stops.

- **truth table:** A means of listing all of the possible values of a Boolean expression.
- **`while` loop(s):** A pretest loop that examines a Boolean expression before causing a statement to be executed.

Chapter 4

Strings and Text Files

At a Glance

Instructor's Manual Table of Contents

Lecture Notes

Overview

Chapter 4 provides an introduction to strings and text files in Python. Students learn how use and manipulate strings, including some of the most useful string methods. Next they learn to create, read from, and write to text files. Finally, they learn how to use library functions to access and navigate a file system.

Objectives

After completing this chapter, students will be able to:

- Access individual characters in a string
- Retrieve a substring from a string
- Search for a substring in a string
- Convert a string representation of a number from one base to another base
- Use string methods to manipulate strings
- Open a text file for output and write strings or numbers to the file
- Open a text file for input and read strings or numbers from the file
- Use library functions to access and navigate a file system

Teaching Tips

Accessing Characters and Substrings in Strings

1. Explain to students that in this section the internal structure of a string is examined. Make sure to introduce the term *substring*.

Teaching Tip	For a brief introduction to text processing in Python, read: www.ibm.com/developerworks/linux/library/l-python5.html.

The Structure of Strings

1. Explain that a string is a *data structure* consisting of zero or more characters. Use the example provided in the book and Figure 4.1 to describe the structure of a string in Python. Stress that the numbering of the string's characters starts at 0.

2. Stress that a string is an *immutable data structure*.

3. Note that the length of a string is the number of characters it contains. A string with no characters has a length of zero (0).

The Subscript Operator

1. Describe the syntax of the *subscript operator*. Introduce the term *index*. Note that the index is usually in the range of 0 and (length of the string – 1). Explain why you can use negative indexes in Python.

2. Provide a few examples so that students become familiar with how the subscript operator works with strings.

3. Provide an example to explain that the subscript operator is useful when working with a count controlled loop.

Slicing for Substrings

1. Use a few examples to show how Python's subscript operator can be used to obtain a substring through a process called *slicing*.

Testing for a Substring with the `in` Operator

1. Use a couple of examples to show how to use the `in` operator to test for a substring within a string.

Quick Quiz 1

1. What is a data structure?
   Answer: A data structure is a compound unit that consists of several smaller pieces of data.

2. True or False: The string is an immutable data structure.
   Answer: True

3. True or False: To extract a substring, the programmer places a semicolon (;) in the subscript.
   Answer: False

4. The operator _____ returns `True` if the target string is somewhere in the search string; otherwise, it returns `False`.
   Answer: `in`

Data Encryption

1. Explain that by using *sniffing software*, an attacker can easily observe data crossing a network. Stress that the problem is even more pervasive in wireless networks.

2. Note that *data encryption* can be used to protect information transmitted on networks. Briefly describe how encryption works, introducing some important terms like *keys*, *encrypt*, *decrypt*, *cipher text*, and *plain text*.

3. Explain how a *Caesar cipher* works, and use the code provided in the book to show how the encryption and decryption functionality can be implemented in Python.

4. Explain why a Caesar cipher is easy to break using modern computers. Provide a brief introduction to more complex encryption methods, like the *block cipher* described in the book.

Teaching Tip	For more information about the Caesar cipher, read: http://www.cs.trincoll.edu/~crypto/historical/substitution.html, and http://www.cs.trincoll.edu/~crypto/historical/caesar.html.

Strings and Number Systems

1. Explain that, because *binary numbers* can be long strings of 0s and 1s, computer scientists often use other number systems, such as *octal* (base eight) and *hexadecimal* (base 16) as shorthand for these numbers. Point out that the number system people use in their day-to-day life is the *decimal*, or base ten, number system.

2. Demonstrate the syntax for identifying the number system being used.

Teaching Tip	You can find an online number system conversion tool at: www.cstc.org/data/resources/60/convtop.html.

The Positional System for Representing Numbers

1. Use a few examples and Figure 4.2 to explain that in *positional notation*, a digit has a *positional value* that is determined by raising the base to the power of the position ($base^{position}$).

2. Explain how the quantity represented by a number is calculated by multiplying a digit by its positional value and adding the results. Explain why this doesn't work for base 11 or higher.

Converting Binary to Decimal

1. Use one or more examples to show students how to manually convert a binary number (*bit string*) to its decimal equivalent.

2. Use the code provided in the book to show how to implement binary-to-decimal conversion in Python.

Converting Decimal to Binary

1. Use the code provided in the book to show how to implement decimal-to-binary conversion in Python.

Conversion Shortcuts

1. Use Table 4.1 and one or more examples to describe a quicker way to compute the decimal value of a bit string.

Octal and Hexadecimal Numbers

1. Use Figure 4.3 to show how to convert from octal to binary. Explain that to convert binary to octal, you begin at the right and factor the bits into groups of three bits each. Point out that the reverse process is carried out to convert from binary to octal.

2. Explain the structure of a hexadecimal number. Point out that the quantities 10…15 are represented by the letters A…F.

3. Use Figure 4.4 to show how to convert from hexadecimal to binary. Explain that to convert binary to hex, you begin at the right and factor the bits into groups of four and look up the corresponding hex digits. Point out that the reverse process is carried out to convert from binary to hexadecimal.

String Methods

1. Explain that Python includes a set of string operations called *methods* that make tasks like counting the words in a single sentence easy.

2. Explain that a method is always called with a given data value called an *object*. Stress that a method knows about the internal state of the object with which it is called. Demonstrate the syntax of calling a method:
`<object_name>.<method_name>(<argument1>, <argument2>…<argumentN>)`

3. Point out that methods can receive arguments and return a value.

4. Note that in Python, all data values are objects and every data type includes a set of methods to use with objects of that type.

5. Use one or more examples to show students how to use the string methods available in Python.

6. Provide a brief overview of each of the string methods listed in Table 4.2.

Teaching Tip	For a complete list of the string methods available in Python, visit: http://docs.python.org/lib/string-methods.html.

Quick Quiz 2

1. What is positional notation?
   Answer: All of the number systems we have examined use positional notation—that is, the value of each digit in a number is determined by the digit's position in the number. In other words, each digit has a positional value. The positional value of a digit is determined by raising the base of the system to the power specified by the position ($base^{position}$)

2. True or False: The octal system uses a base of 8 and the digits 0…7.
   Answer: True

3. True or False: Each digit in the hexadecimal number is equivalent to three digits in the binary number.
   Answer: False

4. Unlike a function, a method is always called with a given data value called a(n) _____, which is placed before the method name in the call.
   Answer: object

Text Files

1. Explain that a text file is a software object that stores data on a permanent medium such as a disk or CD.

2. Briefly describe the advantages of using text files for input in a program (versus interactive human input).

Text Files and Their Format

1. Show an example of a text file, and note that you can easily create one in a text editor.

2. Stress that all data output to or input from a text file must be strings.

Writing Text to a File

1. Use an example to show how to write to a text file using a `file` object and the `open`, `write`, and `close` methods. Stress that failure to close an output file can result in data being lost.

2. Explain what happens if you write to a file that does not previously exist, or if you write to an existing file.

Teaching Tip	For more information on `file` objects, read: http://docs.python.org/lib/bltin-file-objects.html.

Writing Numbers to a File

1. Note that the `file` method `write` expects a string as an argument. For this reason, other types of data must be converted to strings before being written to the output file.

2. Use the example provided in the book to show how to use `str` to convert a number to a string before writing it to a file.

Reading Text from a File

1. Use an example to show how to read from a text file using a `file` object and the `open` and `read` methods. Point out that if the specified pathname is not accessible, Python raises an error.

2. Stress that after input is finished, `read` returns an empty string.

3. Explain that in order to re-read a file it must be reopened.

4. Introduce the `readline` method, and explain how it can be used to read a specific number of lines from a file.

5. Demonstrate how to use `for` and `while` loops to read the contents of a file.

Reading Numbers from a File

1. Use the examples provided in the book to show how to read numbers from a text file. Point out that you must know the type of white spaces used in order to efficiently process the numbers read from a file.

2. Use Table 4.3 to briefly review the `file` methods studied in this section.

Accessing and Manipulating Files and Directories on Disk

1. Explain why it is important to include error recovery in Python programs that interact with files.

2. Use Tables 4.4 and 4.5 and the code snippets provided in the book to show how to use some system functions that are useful when working with files and directories on disk.

Case Study: Text Analysis

1. Explain the *Flesch Index* and what it measures.

Request

1. Explain to students that they are asked to write a program that computes the Flesch index and grade level for text stored in a text file.

Analysis

1. Use Table 4.6 to provide an overview of the items used in the text-analysis program and their definitions.

Design

1. Use Table 4.7 to briefly describe each of the tasks in the text analysis program.

Implementation (Coding)

1. Ask students whether they made any mistakes while typing the code in their computers. Were their mistakes syntax or logic errors?

Testing

1. Explain that *bottom-up* testing can be used to test the text-analysis program. Introduce the term *driver*.

Teaching Tip	Bottom-up testing: "An integration testing technique that tests the low-level components first using test drivers for those components that have not yet been developed to call the low-level components for test." For more information, read: http://foldoc.org/?bottom-up+testing.

Quick Quiz 3

1. Data can be output to a text file using a(n) _____ object.
   Answer: `file`

2. True or False: You can close a file with the `append` method.
   Answer: False

3. True or False: The `file` method `write` expects a string as an argument.
   Answer: True

4. After input is finished, another call to `read` would return a(n) _____ to indicate that the end of the file has been reached.

Answer: empty string

Class Discussion Topics

1. Are students familiar with string manipulation functions in other programming languages? If so, ask them to compare how strings are manipulated in one or more other languages with respect to how they are manipulated in Python.

2. Ask students to brainstorm for situations in which they would need to use each of the functions listed in Tables 4.4 and 4.5 for file/directory manipulation.

Additional Projects

1. Ask students to do some research on other string methods available in Python. They should compile a list of at least five other useful methods that were not studied in this chapter.

2. Ask students to do some research on how brute force attacks and frequency analysis attacks can be used to break a Caesar cipher. Have them summarize their findings in a one-page write-up.

Additional Resources

1. String Methods:
   http://docs.python.org/lib/string-methods.html

2. Python String Methods:
   http://python.about.com/od/pythonstandardlibrary/a/string-methods.htm

3. Charming Python: Text Processing in Python:
   www.ibm.com/developerworks/linux/library/l-python5.html

4. File Objects:
   http://docs.python.org/lib/bltin-file-objects.html

Key Terms

- **binary digit:** A digit, either 0 or 1, in the binary number system. Program instructions are stored in memory using a sequence of binary digits. See also bit.
- **bit:** A binary digit.
- **block cipher:** An encryption method that replaces characters with other characters located in a two-dimensional grid of characters.

- **bottom-up testing:** A method of coding and testing a program that starts with lower-level modules and a test driver module.
- **Caesar cipher:** An encryption method that replaces characters with other characters a given distance away in the character set.
- **data decryption:** The process of translating encrypted data to a form that can be used.
- **data encryption:** The process of transforming data so that others cannot use it.
- **data structure:** A compound unit consisting of several data values.
- **driver:** A function used to test other functions.
- **encryption:** The process of transforming data so that others cannot use it.
- **file:** A data structure that resides on a secondary storage medium accessing/manipulating.
- **immutable data structure:** a data structure whose internal data or state cannot be changed
- **index:** The relative position of a component of a linear data structure or collection.
- **key(s):** An item that is associated with a value and which is used to locate that value in a collection.
- **method(s):** A chunk of code that can be treated as a unit and invoked by name. A method is called with an object or class.
- **object:** A collection of data and operations, in which the data can be accessed and modified only by means of the operations.
- **positional notation:** The type of representation used in based number systems, in which the position of each digit denotes a power in the system's base.
- **slicing:** An operation that returns a subsection of a sequence, for example, a sublist or a substring.
- **substring:** A string that represents a segment of another string.
- **text files:** Files that contain characters and are readable and writable by text editors.

Chapter 5

Lists and Dictionaries

At a Glance

Instructor's Manual Table of Contents

Lecture Notes

Overview

Chapter 5 describes lists and dictionaries. Students learn to construct, use, and manipulate lists. They also learn to define simple functions that expect parameters and return values. Finally, they learn to construct, use, and manipulate dictionaries.

Objectives

After completing this chapter, students will be able to:

- Construct lists and access items in those lists
- Use methods to manipulate lists
- Perform traversals of lists to process items in the lists
- Define simple functions that expect parameters and return values
- Construct dictionaries and access entries in those dictionaries
- Use methods to manipulate dictionaries
- Decide whether a list or a dictionary is an appropriate data structure for a given application

Teaching Tips

Lists

1. Introduce the terms: *list* and *item*/*element*.

2. Provide some examples of real-life lists. You can use the bullet points on Page 160 in the text as a guide.

3. Explain that each item in a list has a unique *index* that specifies its position (from 0 to length – 1).

Teaching Tip	For more on Python lists, visit: http://diveintopython.org/native_data_types/lists.html.

List Literals and Basic Operators

1. Provide a few examples to show students what list literals look like. Explain that when an element is an expression or variable, its value is included in the list (not the expression itself).

2. Use a couple of examples to show how to create lists using the `range` and `list` functions.

3. Use several examples to show how the `len` function and the subscript, concatenation (+), and equality (==) operators work with lists.

4. Show how to print lists, both by printing the whole list at once and by iterating through the list in a loop that prints each element individually.

5. Use one or more examples to show how the `in` operator can be used to detect the presence or absence of an element in a list.

6. Use Table 5.1 to review the operators and functions covered in this section.

Replacing an Element in a List

1. Explain that a list is *mutable*, which means that the list itself maintains its identity but its *state*—its length and its contents—can change.

2. Use the examples provided in the book to show how to use the subscript operator to replace an element in a list. Stress that the subscript is used to reference the target of the assignment – the position of the replaced element within the list.

3. Explain that a sublist of elements may be replaced by using the slice operator, and demonstrate using the example on Page 165 of the text.

List Methods for Inserting and Removing Elements

1. Use Table 5.2 to explain that the `list` type includes several methods for inserting and removing elements. Use the examples provided in the book to show how to use these methods.

2. Stress the difference between the list `extend` and `append` methods.

Teaching Tip	In Python, lists can be used as stacks and as queues. For more information, read: http://docs.python.org/tut/node7.html.

Searching a List

1. Remind students that the `in` operator can be used to determine whether or not an element is present in a list.

2. Provide an example to illustrate how to use the method `index` to locate an element's position in a list. Be sure to point out that because the `index` method raises an error if the element is not found in the list, it is good practice to use the `in` operator to check whether an element is present prior to using the `index` method to locate that element.

Sorting a List

1. Provide an example to illustrate how to use the `sort` method to arrange the elements of a list in ascending order.

Mutator Methods and the Value `None`

1. Explain that *mutator* methods usually do not return a value. Use an example to show that in these cases, Python automatically returns the special value `None`.

Aliasing and Side Effects

1. Use the example provided on Page 169 of the book and Figure 5.1 to show how the mutable property of lists leads to interesting phenomena. Make sure students understand what *side effect* and *aliasing* are in the context of lists.

2. Use the example provided on Page 170 of the book and Figure 5.2 to show that aliasing can be avoided by copying the contents of an object instead of copying the reference to the object. Show how the slicing operator can be used to simplify this process.

Equality: Object Identity and Structural Equivalence

1. Use the example provided on Page 171 of the book and Figure 5.3 to define the terms *identity* and *structural equivalence* as they apply to variables in Python. Show that the equality operator (`==`) tests for structural equivalence, not for object identity. Show how to use the `is` operator to test for object identity.

Example: Using a List to Find the Median of a Set of Numbers

1. Use the example provided in the book to show how to use a list to find the *median* of a set of numbers. Make sure that students use this example to understand how different methods can be applied to lists in order to achieve a desired result.

Tuples

1. Explain that a *tuple* resembles a list, but it is immutable. Provide a few examples to illustrate how to create and use tuples, stressing the syntax differences between creating lists and creating tuples.

2. Note that most of the operators and functions used with lists can be used in a similar fashion with tuples.

Teaching Tip	"Assigning a tuple to a several different variables is called *tuple unpacking*, while assigning multiple values to a tuple in one variable is called *tuple packing*. When unpacking a tuple, or performing multiple assignment, you must have the same number of variables being assigned to as values being assigned." *Reference:* http://en.wikibooks.org/wiki/Python_Programming/Tuples.

Quick Quiz 1

1. What is a list in Python?
   Answer: In Python, a list is written as a sequence of data values separated by commas. The entire sequence is enclosed in square brackets (`[` and `]`).

2. True or False: The `is` operator can be used to detect the presence or absence of a given element in a list.
   Answer: False

3. True or False: A list is immutable.
   Answer: False

4. What is a tuple?
   Answer: A tuple is a type of sequence that resembles a list—although unlike a list, a tuple is immutable. You indicate a tuple literal in Python by enclosing its elements in parentheses instead of square brackets.

Defining Simple Functions

1. Explain that defining functions allows programmers to organize their code in existing scripts more effectively.

The Syntax of Simple Function Definitions

1. Describe the syntax of a simple function definition. Use the `square(x)` example provided in the book to show how to define and use a simple value-returning function.

2. Stress that it is important to provide a docstring that contains information about what the function does.

3. Note that a function can be defined in a Python shell, but that it is more convenient to define it in an IDLE window.

4. Point out that a function must be defined in a script before it is called in that script.

Parameters and Arguments

1. Explain the difference between parameters and arguments.

Teaching Tip	Python allows the programmer to provide default parameters in the header of a function definition. For more information, read: http://docs.python.org/ref/function.html.

The `return` Statement

1. Explain that you should place a `return` statement at each exit point of a function when the function should explicitly return a value. Describe the syntax of this statement. Point out that a function may include more than one `return` statement, for instance when an `if-else` statement is used in the function.

2. Stress that if a function contains no `return` statement, Python transfers control to the caller after the last statement in the function's body is executed and the special value `None` is automatically returned.

Boolean Functions

1. Explain that a *Boolean function* usually tests its argument for the presence or absence of some property.

2. Use the example provided in the book to show how to define and use a Boolean function.

Defining a `main` Function

1. Describe the role of the `main` function. Stress that the definition of `main` and other functions can appear in no particular order in the script, as long as `main` is called at the end of the script.

2. Use an example to illustrate how to define and use a `main` function.

Case Study: Generating Sentences

1. Guide students as they step through this section.

Request

1. Ask students to write a program that generates sentences.

Analysis

1. Use Table 5.3 to explain that the program created in this section will generate sentences from a simplified subset of English.

Design

1. Explain why it is important to assign the task of generating each phrase to a separate function. Briefly describe the role of the `sentence`, `nounPhrase`, and `main` functions.

Implementation (Coding)

1. Stress that the variables for the data should be initialized just below the `import` statement.

Testing

1. Describe the difference between the bottom-up and top-down testing approaches. Note that a wise programmer can mix bottom-up and top-down testing as needed.

Quick Quiz 2

1. Why is it useful to define your own functions?
   Answer: Some scripts might be useful enough to package as functions that can be used in other scripts. Moreover, defining your own functions allows you to organize your code in existing scripts more effectively.

2. True or False: The function's header contains the function's name and a parenthesized list of argument names.
   Answer: True

3. The header of a function consists of the reserved word _____, followed by the function's name, followed by a parenthesized list of parameters, followed by a colon.
   Answer: `def`

4. A(n) _____ function usually tests its argument for the presence or absence of some property.
   Answer: Boolean

Dictionaries

1. Use an example to explain that a dictionary organizes information by *association*, not by position. Note that data structures organized by association are also called *tables* or *association lists*.

2. Explain that in Python, a *dictionary* associates a set of *keys* with data values.

Teaching Tip	Here are some useful resources with more information about Python dictionaries: • http://diveintopython.org/getting_to_know_python/dictionaries.html • www.developer.com/article.php/630721 • http://en.wikibooks.org/wiki/Python_Programming/Dictionaries

Dictionary Literals

1. Provide several examples to show how to create dictionary literals in Python. Note that `{}` is an empty dictionary.

2. Explain that in a dictionary literal, the keys can be data of any immutable type, including other data structures.

Adding Keys and Replacing Values

1. Use a few examples to show how to use the subscript operator (`[]`) to add a new key/value pair to a dictionary, as well as to replace a value at an existing key.

Accessing Values

1. Use a few examples to show how to use the subscript operator (`[]`) to obtain the value associated with a key.

2. Explain that if the existence of a key is uncertain, you can test for it using the dictionary method `has_key`. Use an example to show that an easier strategy is to use the method `get`, and explain the syntax of this method.

Removing Keys

1. Provide a couple of examples to show how you can use the method `pop` to delete an entry from a dictionary.

2. Explain what happens if the key provided to the `pop` method is absent from the dictionary when the `pop` method is used with one argument. Next explain when it is used with two arguments.

Traversing a Dictionary

1. Provide examples to show how to traverse a dictionary using a loop and the method `items` (which returns a list of tuples).

2. Explain how the method `keys` can be used to help generate a specific order for traversing a dictionary.

3. Use Table 5.4 to briefly describe the dictionary operations studied in this chapter.

Example: The Hexadecimal System Revisited

1. Use the example provided in the book to show how to use a dictionary to create a hex-to-binary *lookup table* to aid in the conversion process.

Example: Finding the Mode of a List of Values

1. Use the code provided in the book to show how to use a dictionary to find the *mode* of a list of values.

Teaching Tip	Besides lists, tuples, and dictionaries, Python also includes a data type for sets. For more information, read: http://docs.python.org/tut/node7.html#SECTION007400000000000000000.

Case Study: Nondirective Psychotherapy

1. Guide students as they step through this section.

Teaching Tip	Nondirective psychotherapy is also called *client-centered* or *person-centered psychotherapy*. For more information, view the Britannica Online Encyclopedia entry on the subject: http://www.britannica.com/EBchecked/topic/417641/nondirective-psychotherapy.

Request

1. Ask students to write a program that emulates a nondirective psychotherapist.

Analysis

1. Use Figure 5.4 to describe the user interface of the program.

Design

1. Explain why it is a good idea to create a set of collaborating functions that share a common data pool.

Implementation (Coding)

1. Ask students to enter and run the `doctor.py` script.

Testing

1. Describe a few situations in which the program does not work as expected. Note that, with a little work, you can make the replies more realistic.

Quick Quiz 3

1. What is a dictionary?
   Answer: A dictionary organizes information by association, not position. In computer science, data structures organized by association are also called tables or association lists. In Python, a dictionary associates a set of keys with data values.

2. True or False: A dictionary associates a set of indexes with data values.
   Answer: False

3. True or False: `[]` is an empty dictionary.
   Answer: False

4. If the existence of a dictionary key is uncertain, the programmer can test for it using the dictionary method `has_key`. However, an easier strategy is to use the method _____.
   Answer: `get`

Class Discussion Topics

1. Ask your students to brainstorm for three situations in which they would use a list, three situations in which they would use a tuple, and three situations in which they would use a dictionary.

2. If your students are familiar with other programming languages, ask them if those languages provide data structures with functionality similar to that of Python's dictionaries.

Teaching Tip	A dictionary is like a *hash* in Perl, a *Hashtable* in Java, and an instance of the *Scripting.Dictionary* object in Visual Basic.

Additional Projects

1. In addition to lists, tuples, and dictionaries, Python also includes a data type for sets. Ask students to do some research to learn about this data structure and how to use it.

2. Lists can be used as stacks and queues. Ask students to write scripts that use a list as a stack and as a queue, respectively. Tip: you may ask them to read the "Data Structures" section of the Python Tutorial before working in this exercise. (http://docs.python.org/tut/node7.html).

Additional Resources

1. Python Tutorial: Data Structures: http://docs.python.org/tut/node7.html

2. Dive Into Python: Introducing Lists: http://diveintopython.org/native_data_types/lists.html

3. Python Programming/Tuples: http://en.wikibooks.org/wiki/Python_Programming/Tuples

4. Python Tutorial: Function Definitions: http://docs.python.org/ref/function.html

5. Python Programming/Dictionaries: http://en.wikibooks.org/wiki/Python_Programming/Dictionaries

Key Terms

- **alias:** A situation in which two or more names in a program can refer to the same memory location. An alias can cause subtle side effects.
- **association:** A pair of items consisting of a key and a value.
- **dictionary:** A data structure that allows the programmer to access items by specifying key values.
- **element:** A value that is stored in an array or a collection. Also known as item.
- **function(s):** A chunk of code that can be treated as a unit and called to perform a task.
- **function heading:** The portion of a function implementation containing the function's name and parameter names.
- **grammar:** The set of rules for constructing sentences in a language.
- **identity:** The property of an object that it is the same thing at different points in time, even though the values of its attributes might change.
- **index:** The relative position of a component of a linear data structure or collection.
- **key(s):** An item that is associated with a value and which is used to locate that value in a collection.
- **logical structure:** The organization of the components in a data structure, independent of their organization in computer memory.
- **mutator:** A method used to change the value of an attribute of an object.
- **natural ordering:** The placement of data items relative to each other by some internal criteria, such as numeric value or alphabetical value.
- **`None` value:** A special value that indicates that no object can be accessed.

- **object identity:** The property of an object that it is the same thing at different points in time, even though the values of its attributes might change.
- **side effect:** A change in a variable that is the result of some action taken in a program, usually from within a method.
- **state:** The set of all the values of the variables of a program at any point during its execution.
- **structural equivalence:** A criterion of equality between two distinct objects in which one or more of their attributes are equal.
- **tuple:** A linear, immutable collection.
- **value:** An item that is associated with a key and is located by a key in a collection.

Chapter 6

Design with Functions

At a Glance

Instructor's Manual Table of Contents

Lecture Notes

Overview

Chapter 6 describes the role of functions in structuring code in a program. Students learn to employ top-down design to assign tasks to functions. Through several examples, students learn to use and define functions, including the use of optional parameters. Advanced topics like recursion, namespaces, and higher-order functions are covered, too.

Objectives

After completing this chapter, students will be able to:

- Explain why functions are useful in structuring code in a program
- Employ top-down design to assign tasks to functions
- Define a recursive function
- Explain the use of the namespace in a program and exploit it effectively
- Define a function with required and optional parameters
- Use higher-order functions for mapping, filtering, and reducing

Teaching Tips

Functions as Abstraction Mechanisms

1. Introduce the concept of *abstraction*, and note that effective designers must invent useful abstractions to control complexity.

Teaching Tip	"In computer science, abstraction is a mechanism and practice to reduce and factor out details so that one can focus on a few concepts at a time." *Reference:* http://en.wikipedia.org/wiki/Abstraction_(computer_science).

Functions Eliminate Redundancy

1. Use the example provided in the book to show how functions serve as abstraction mechanisms by eliminating redundant, or repetitious, code. Discuss the importance of eliminating redundancy in order to easily test and debug code.

Functions Hide Complexity

1. Use the code of the `sum` function introduced in the previous sub-section to explain that functions serve as abstraction mechanisms by hiding complicated details, including length of code and number of interacting components.

Functions Support General Methods with Systematic Variations

1. Note that an algorithm is a *general method* for solving a class of problems.

2. Stress that algorithms should be general enough to provide a solution to many *problem instances*, and explain that a function should provide a general method with systematic variations.

3. Explain that a function's arguments provide the means for varying the problem instances that the function's algorithm solves.

Functions Support the Division of Labor

1. Explain that in a computer program, functions can enforce a division of labor. Each function should perform a single coherent task. Each of the tasks required by a system can be assigned to a function.

Teaching Tip	Point out that functions also allow for division of labor between multiple programmers working on a joint project. The function names must be set in advance, but after that one programmer can write a given function and all other programmers can use that function without knowing its exact implementation.

Problem Solving with Top-Down Design

1. Introduce the terms *top-down design*, *problem decomposition*, and *stepwise refinement*.

Teaching Tip	"Top-down design was promoted in the 1970s by IBM researcher Harlan Mills and Niklaus Wirth." *Reference:* http://en.wikipedia.org/wiki/Top-down.

The Design of the Text-Analysis Program

1. Use Figure 6.1 to describe how the text-analysis program of Chapter 4 could have been structured in terms of programmer-defined functions, specifying the arguments each function receives and the value it returns.

The Design of the Sentence-Generator Program

1. Use Figure 6.2 to show how the sentence-generator program of Chapter 5 used a top-down design that flows out of the top-down structure of the grammar.

The Design of the Doctor Program

1. Use Figure 6.3 to show how the doctor program of Chapter 5 used a *responsibility-driven design*.

2. Use the functions reply and changePerson of the doctor program to provide guidelines indicating when a problem should be decomposed into multiple functions and when it should be solved by a single function.

Quick Quiz 1

1. What is an abstraction?
   Answer: An abstraction hides detail and thus allows a person to view many things as just one thing. We use abstractions to refer to the most common tasks in everyday life.

2. True or False: Functions serve as abstraction mechanisms by hiding complicated details.
   Answer: True

3. True or False: The individual problems that make up a class of problems are known as problem variations.
   Answer: False

4. What is top-down design?
   Answer: One popular design strategy for programs of any significant size and complexity is called top-down design. This strategy starts with a global view of the entire problem and breaks the problem into smaller, more manageable subproblems—a process known as problem decomposition. As each subproblem is isolated, its solution is assigned to a function.

Design with Recursive Functions

1. Introduce the terms *recursive design* and *recursive function*.

Defining a Recursive Function

1. Explain that a recursive function is a function that calls itself. Explain that to avoid infinite recursion, recursive functions need to have a *base case* and a *recursive step*.

2. Use the Python code provided in the book to show how to convert `displayRange` into a recursive function, thereby clarifying how recursive functions are used and how loop functions can often be converted into recursive functions. Introduce the term *recursive call*.

3. Use the recursive version of the `sum` function to explain that most recursive functions expect at least one argument.

Tracing a Recursive Function

1. Use the example provided in the book to show how to trace a recursive function and to demonstrate why recursive functions fill up the stack faster than linear functions.

Using Recursive Definitions to Construct Recursive Functions

1. Explain that a *recursive definition* consists of equations that state what a value is for one or more base cases and one or more recursive cases.

2. Use the Fibonacci example provided in the book to show how to use a recursive definition to construct a recursive function.

Recursion in Sentence Structure

1. Use the example provided in the book to explain that recursive solutions can often flow from the structure of a problem. In this case, a noun phrase can be modified by a prepositional phrase, which also contains another noun phrase.

2. Introduce the term *indirect recursion*. Explain how to make sure that indirect recursion does not go on forever.

Teaching Tip	While the depth of indirect recursion may vary, indirect recursion requires the same attention to base cases as direct recursion. For more information, visit: www.cs.sfu.ca/~tamaras/recursion/Direct_vs_Indirect.html.

Infinite Recursion

1. Explain that *infinite recursion* arises when programmer fails to specify the base case or to reduce the size of the problem in a way that terminates the recursive process.

2. Use an example to show that in the event of infinite recursion, the Python virtual machine will eventually run out of memory resources to manage the process.

The Costs and Benefits of Recursion

1. Use Figure 6.4 to describe the costs of using recursive functions. Introduce the terms *call stack* and *stack frame*.

2. Provide some insight on why it is sometimes more convenient to develop recursive solutions despite the hidden costs.

Teaching Tip	For more information on mastering recursive programming, read: http://www.ibm.com/developerworks/linux/library/l-recurs.html.

Case Study: Gathering Information from a File System

1. Guide students as they step through this section.

Request

1. Ask students to write a program that allows the user to obtain information about the file system.

Analysis

1. Have your students taken an operating systems course before? If not, you will have to introduce several terms required to understand this case study, like *root directory*, *path*, and *parent* (of a node in a file system).

Design

1. Explain that the program can be structured according to two sets of tasks: those concerned with implementing a menu-driven command processor and those concerned with executing the commands. Outline the advantages of this structure.

Implementation (Coding)

1. Stress that the functions used in the program lead to a division of labor.

Quick Quiz 2

1. What is a recursive function?
   Answer: A recursive function is a function that calls itself.

2. A recursive _____ consists of equations that state what a value is for one or more base cases and one or more recursive cases.
   Answer: definition

3. True or False: Inefficient recursion arises when the programmer fails to specify the base case or to reduce the size of the problem in a way that terminates the recursive process.
   Answer: False

4. What is a stack frame?
   Answer: At program startup, the PVM reserves an area of memory named a call stack. For each call of a function, the PVM must allocate a small chunk of memory called a stack frame. In this type of storage, the system places the values of the arguments and the return address for the particular function call. Space for the function call's return value is also reserved in its stack frame. When a call returns or completes its execution, the return address is used to locate the next instruction in the caller's code and the memory for the stack frame is deallocated.

Managing a Program's Namespace

1. Explain that a program's *namespace* is the set of its variables and their values.

Module Variables, Parameters, and Temporary Variables

1. Use the sample code provided in the book to describe the difference between *module variables*, *temporary variables*, *parameters*, and *method names*.

Scope

1. Define the term *scope* as it applies to programs, comparing it to the scope of a word in a sentence.

2. Use one or more examples to explain that module variables, parameters, and temporary variables have different scope.

3. Stress that a function can reference a module variable but cannot assign a new value to it under normal circumstances.

Teaching Tip	Python has two built-in functions, `locals` and `globals`, which provide dictionary-based access to local and global variables. For more information, visit: www.faqs.org/docs/diveintopython/dialect_locals.html.

Lifetime

1. Explain the following with respect to the *lifetime* of variables in Python: when a variable comes into existence, storage is allocated for it; when it goes out of existence, storage is reclaimed by the PVM.

2. Demonstrate how the concept of lifetime explains the fact that a function can reference a module variable but cannot assign a new value to it.

Default (Keyword) Arguments

1. Use the examples provided in the book to show how to define and use *default* or *keyword* arguments.

2. Explain that the default arguments that follow a function call can be supplied in two ways: *by position* and *by keyword*.

Teaching Tip	A note on default arguments: "The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls." *Reference:* http://docs.python.org/tut/node6.html

Higher-Order Functions (Advanced Topic)

1. Explain what a *higher-order function* is. Stress that a higher-order function separates the task of transforming each data value from the logic of accumulating the results.

Teaching Tip	Python supports multiple programming paradigms (primarily object oriented, imperative, and functional). For more information on how to use Python as a functional programming language, see the second reference listed below. *References:* http://en.wikipedia.org/wiki/Python_(programming_language) http://gnosis.cx/publish/programming/charming_python_13.html

Functions as First-Class Data Objects

1. Explain that functions are *first-class data objects* in Python. Use the examples provided in the book to show how this functionality can be very useful, particularly when trying to separate different sections of code and reduce redundancy.

Mapping

1. Use one or more examples to show how Python supports *mapping* of functions.

2. Point out that the `map` function returns a map object, which must then be converted to a list or other desired object to be viewed and used.

Filtering

1. Use one or more examples to explain that when *filtering*, a function called a *predicate* is applied to each value in a list. Explain what happens when the predicate returns True and when it returns False.

Reducing

1. Use one or more examples to explain that when *reducing*, we take a list of values and repeatedly apply a function to accumulate a single data value.

Using `lambda` to Create Anonymous Functions

1. Use the examples provided in the book to show how to use `lambda` to create *anonymous functions* in Python. Be sure to specify the syntax restrictions of `lambda`.

Creating Jump Tables

1. Use the example provided in the book to show how to create a *jump table* (i.e., a dictionary of functions keyed by command names).

Teaching Tip	"In computer programming, a branch table (sometimes known as a jump table) is a term used to describe an efficient method of transferring program control (branching) to another part of a program (or a different program that may have been dynamically loaded) using a table of branch instructions. The branch table construction is commonly used when programming in assembly language but may also be generated by a compiler." *Reference:* http://en.wikipedia.org/wiki/Branch_table

Quick Quiz 3

1. What is a program's namespace?
   Answer: A program's namespace is the set of its variables and their values.

2. True or False: When module variables are introduced in a program, they are immediately given a value.
   Answer: True

3. True or False: In a program, the context that gives a name a meaning is called its lifetime.
   Answer: False

4. Explain what the following statement means: "In Python, functions can be treated as first-class data objects."

   Answer: This means that they can be assigned to variables (as they are when they are defined), passed as arguments to other functions, returned as the values of other functions, and stored in data structures such as lists and dictionaries.

Class Discussion Topics

1. Have students used top-down design principles before? If so, ask them to discuss their experiences in class.

2. Are your students familiar with any functional programming language? If so, ask them to compare the use of functions as first class objects in that language with those in Python.

Additional Projects

1. Convert Euclid's algorithm to find the greatest common divisor (GCD) of two positive integers (Project 3.8) to a recursive function named `gcd`.

2. Ask your students to do some research on functional programming and lambda calculus. Can Python be used as a functional programming language? They should document their findings in a 1-2 page report.

Additional Resources

1. Python Scopes and Name Spaces:
   www.network-theory.co.uk/docs/pytut/PythonScopesandNameSpaces.html

2. Default Argument Values:
   http://docs.python.org/tut/node6.html#SECTION00671000000000000000

3. Functional Programming in Python:
   http://gnosis.cx/publish/programming/charming_python_13.html

Key Terms

- **abstraction:** A simplified view of a task or data structure that ignores complex detail.
- **anonymous function:** A function without a name, constructed in Python using `lambda`.
- **base case:** The condition in a recursive algorithm that is tested to halt the recursive process.
- **call:** Any reference to a function or method by an executable statement. Also referred to as invoke.
- **call stack:** The trace of function or method calls that appears when Python raises an exception during program execution.
- **default argument:** Also called a keyword argument. A special type of parameter that is automatically given a value if the caller does not supply one.
- **filtering:** The successive application of a Boolean function to a list of arguments that returns a list of the arguments that make this function return True.
- **first-class data objects:** Data objects that can be passed as arguments to functions and returned as their values.
- **general method:** A method that solves a class of problems, not just one individual problem.
- **grammar:** The set of rules for constructing sentences in a language.

- **higher-order function:** A function that expects another function as an argument and/or returns another function as a value.
- **indirect recursion:** A recursive process that results when one function calls another, which results at some point in a second call to the first function.
- **infinite recursion:** In a running program, the state that occurs when a recursive function cannot reach a stopping state.
- **jump table:** A dictionary that associates command names with functions that are invoked when those functions are looked up in the table.
- **`lambda`:** The mechanism by which an anonymous function is created.
- **lifetime:** The time during which a data object, function call, or method call exists.
- **mapping:** The successive application of a function to a list of arguments that returns a list of results.
- **namespace(s):** The set of all of a program's variables and their values.
- **path:** A sequence of edges that allows one vertex to be reached from another.
- **predicate:** A function that returns a Boolean value.
- **problem decomposition:** The process of breaking a problem into subproblems.
- **problem instance:** An individual problem that belongs to a class of problems.
- **recursion:** The process of a subprogram calling itself. A clearly defined stopping state must exist.
- **recursive call:** The call of a function that already has a call waiting in the current chain of function calls.
- **recursive definition:** A set of statements in which at least one statement is defined in terms of itself.
- **recursive design:** The process of decomposing a problem into subproblems of exactly the same form that can be solved by the same algorithm.
- **recursive function:** A function that calls itself.
- **recursive step:** A step in the recursive process that solves a similar problem of smaller size and eventually leads to a termination of the process.
- **reducing:** The application of a function to a list of its arguments to produce a single value.
- **responsibility-driven design:** The assignment of roles and responsibilities to different actors in a program.
- **root directory:** The top-level directory in a file system.
- **scope:** The area of program text in which the value of a variable is visible.
- **stack frame:** An area of computer memory reserved for local variables and parameters of method calls. Also known as activation record and run-time stack.
- **stepwise refinement:** The process of repeatedly subdividing tasks into subtasks until each subtask is easily accomplished. See also structured programming and top-down design.
- **structure chart:** A graphical method of indicating the relationship between modules when designing the solution to a problem.
- **temporary variable:** A variable that is introduced in the body of a function or method for the use of that subroutine only.
- **top-down design:** A method for coding by which the programmer starts with a top-level task and implements subtasks. Each subtask is then subdivided into smaller subtasks. This process is repeated until each remaining subtask is easily coded.

Chapter 8

Design with Classes

At a Glance

Instructor's Manual Table of Contents

Lecture Notes

Overview

Chapter 8 provides an introduction to objects and classes with Python. Students learn about object-oriented design and how to create object-oriented programs in Python. Important OO concepts like constructors, instance variables, methods, inheritance, polymorphism, and encapsulation are introduced. Special topics like exception handling (`try-exception`) and how to transfer objects to and from files (pickling/unpickling) are introduced as well.

Objectives

After completing this chapter, students will be able to:

- Determine the attributes and behavior of a class of objects required by a program
- List the methods, including their parameters and return types, that realize the behavior of a class of objects
- Choose the appropriate data structures to represent the attributes of a class of objects
- Define a constructor, instance variables, and methods for a class of objects
- Recognize the need for a class variable and define it
- Define a method that returns the string representation of an object
- Define methods for object equality and comparisons
- Exploit inheritance and polymorphism when developing classes
- Transfer objects to and from files

Teaching Tips

Getting Inside Objects and Classes

1. Provide a brief introduction to *object-oriented programming*, objects, and classes, stressing that objects are abstractions that package their state and methods in a single entity which can be referenced with a name.

2. Provide an overview of the components of a class definition. Refer to the bullet points on Pages 294-295.

Teaching Tip	Note for C/C++/Java/C# Programmers (taken from www.ibiblio.org/g2swap/byteofpython/read/oops.html): "Note that even integers are treated as objects (of the int class). This is unlike C++ and Java (before version 1.5) where integers are primitive native types. See `help(int)` for more details on the class. C# and Java 1.5 programmers will be familiar with this concept since it is similar to the boxing and unboxing concept."

Teaching Tip	For an introduction to object-oriented programming with Python, read: www.voidspace.org.uk/python/articles/OOP.shtml.

A First Example: The `Student` Class

1. Use the sample code provided in the book to explain the role of the `Student` class for a course-management application that needs to represent information about students in a course.

2. Use Table 8.1 to describe the interface of the `Student` class.

3. Describe the syntax of a simple class definition. Stress that the class name is typically capitalized.

4. Explain that Python classes are organized in a tree-like *class hierarchy*. Introduce the terms: `object` (class), *subclass*, and *parent class*.

5. Use the `student.py` code to show how to define a class in Python.

Teaching Tip	For more information on how to define classes in Python, read: http://diveintopython.org/object_oriented_framework/defining_classes.html.

Docstrings

1. Explain that docstrings can appear at three levels: module, just after the class header, and after each method header.

2. Note that entering `help(Student)` at the shell prints the documentation for the class and all of its methods.

Method Definitions

1. Explain that method definitions are indented below the class header.

2. Note the similarities and differences between function definitions and method definitions.

3. Explain that each method definition **must** include a first parameter named `self`, which allows the interpreter to bind the parameter to the object used to call the method.

The `__init__` Method and Instance Variables

1. Use an example to explain the role of the `__init__` method. Explain that this method is the class's *constructor* and is run automatically when a user instantiates the class. Point out that the syntax of this method is fixed and that it must begin and end with two consecutive underscores.

2. Describe the role of *instance variables*. Stress that their scope is the entire class definition.

Teaching Tip	"Some people regard it as a Python 'wart' that we have to include `self`. Java includes it automatically and calls it `this`. The main argument in favour of `self` is the Pythonic principle explicit is better than implicit. This way we can see exactly where all our variable names come from." *Reference:* www.voidspace.org.uk/python/articles/OOP.shtml.

The `__str__` Method

1. Explain that `__str__` builds and returns a string representation of an object's state.

2. Note that when the `str` or `print` functions are called with an object, that object's `__str__` method is automatically invoked.

Accessors and Mutators

1. Remind students what *accessor* and *mutator* methods are. Use one or more examples to help students understand these concepts.

The Lifetime of Objects

1. Explain that the lifetime of an object's instance variables is the lifetime of that object.

2. Use the sample code provided in the book to explain that an object becomes a candidate for the graveyard when it can no longer be referenced. Introduce the term *garbage collector*.

Rules of Thumb for Defining a Simple Class

1. Introduce and explain each of the rules of thumb for defining a simple class listed in Page 302 of the book. You may add your own tips or ask students with previous OO experience to add their own.

Case Study: Playing the Game of Craps

1. Guide students as they step through this section.

Request

1. Ask students to write a program that allows the user to play and study the game of craps.

Analysis

1. Explain to students how they can use the nouns in the problem description to determine what types of classes are required to solve a problem. Introduce the role of the `Player` and `Die` classes.

2. Show and explain the sample user interface provided in the book.

Design

1. Use Table 8.2 to describe the interfaces of the `Player` and `Die` classes.

2. Review the pseudocode of the play method, and make sure students understand how this code captures the logic of playing a game of craps and tracking its results.

Implementation (Coding)

1. Review the code in Pages 306-309 of the book, specifying that the `Die` class is defined in a file named `die.py`. The `Player` class and the top-level functions are defined in a file named `craps.py`.

Quick Quiz 1

1. Why is an object considered an abstraction?
   Answer: Like functions, objects are abstractions. A function packages an algorithm in a single operation that can be called by name. An object packages a set of data values—its state—and a set of operations—its methods—in a single entity that can be referenced with a name. This makes an object a more complex abstraction than a function. To get inside a function, you must view the code contained in its definition. To get inside an object, you must view the code contained in its class. A class definition is like a blueprint for each of the objects of that class.

2. True or False: In a class, all of the method definitions are indented below the class header.
   Answer: True

3. True or False: Within the class definition, the names of instance variables must begin with `this`.
   Answer: False

4. Methods that allow a user to modify an object's state are called _____.
   Answer: mutator methods

Data-Modeling Examples

1. Provide students of an overview of the data modeling examples that are discussed in this section.

Rational Numbers

1. Use the sample code provided in the book to explain why it would be useful to implement a class for *rational numbers*. Introduce the term *overloading*.

2. Briefly go over the `Rational` class definition found on Pages 310-311 in the book.

3. Explain that methods that are not intended to be in a class's interface are typically given names that begin with the _ symbol.

Rational Number Arithmetic and Operator Overloading

1. Use Tables 8.3, 8.4, and the `__add__` example to explain how the built-in arithmetic operators can be overloaded.

2. Stress that *operator overloading* is an abstraction mechanism.

Teaching Tip	For more information on operator overloading in Python, read: www.learningpython.com/2008/06/21/operator-overload-learn-how-to-change-the-behavior-of-equality-operators/.

Comparison Methods

1. Use the example provided in the book to show how to overload and use the various comparison operators. Stress the importance of overloading these operators using the appropriate comparison logic.

2. Point out that once an implementer of a class has defined methods for == and <, the remaining methods can be defined in terms of these two methods.

3. Stress that if new class objects are comparable and the comparison methods are included in that class, other built-in methods—such as the `sort` method for lists—will be able to use the new class objects correctly.

Equality and the `__eq__` Method

1. Use the example provided in the book to show how to overload and use the `__eq__` method.

2. Point out that this method should also be applicable when comparing objects of two different types.

3. Note that one should include `__eq__` in any class where a comparison for equality uses a criterion other than object identity.

Savings Accounts and Class Variables

1. Use Table 8.5 to describe the interface for the `SavingsAccount` class.

2. Explain the concept of a *class variable*. Tell students why such a variable would be useful in the `SavingsAccount` class. Demonstrate how a class variable is introduced and how it is referenced.

3. Briefly step through the code of this class, pointing out the class variable and the use balance as an optional argument.

4. Briefly outline the guidelines for using class variables.

Putting the Accounts into a Bank

1. Use the sample code provided in the book and Table 8.6 to describe the functionality of the `Bank` class. Briefly step through the code of this class. Explain the choice of a directory to represent the collection of accounts.

Using `pickle` for Permanent Storage of Objects

1. Introduce the term *pickling*, and use the `save` example provided in the book to show how to use `pickle` for permanent storage of objects.

Input of Objects and the `try-except` Statement

1. Describe the syntax of the `try-except` statement and how it functions when an exception is generated.

2. Use the code on Page 321 of the book to show how to use a `try-expect` statement in a method.

Teaching Tip	For more information on errors and exceptions in Python, read: http://docs.python.org/tut/node10.html.

Playing Cards

1. Use the sample code provided in the book to show the functionality of the `Card` class. Explain why there is no need to include any methods other than `__str__` for the string representation. Briefly step through the code of this class.

2. Use Table 8.7 and the sample code provided in the book to describe the functionality of the `Deck` class. Briefly step through the code of this class.

Case Study: An ATM

1. Guide students as they step through this section.

Request

1. Ask students to write a program that simulates a simple ATM.

Analysis

1. Use Figure 8.1 to show sample terminal-based interface for the program.

2. The *class diagram* in Figure 8.2 shows the relationships among the classes. Explain how to interpret such diagrams.

3. Introduce the *model/view* pattern and explain how it will be used in this program.

Design

1. Use Table 8.8 to describe the interface of the `ATM` class. Be sure to explain why the `run` method is the only method in this class that does not begin with the _ symbol.

Implementation (Coding)

1. Review the code in `atm.py` with students. Explain that the code defines the `ATM` class, instantiates a `Bank` and an `ATM`, and executes the ATM's `run` method. Explain how the `run` method is used to call other methods as requested by the user.

Quick Quiz 2

1. True or False: The code `x + y` is actually shorthand for the code `y.__add__(x)`.
   Answer: False

2. To overload the % arithmetic operator, you need to define a new method using _____ as the method name.
   Answer: `__mod__`

3. What is operator overloading?
   Answer: Operator overloading is another example of an abstraction mechanism. In this case, programmers can use operators with single, standard meanings even though the underlying operations vary from data type to data type.

4. What is a class variable?
   Answer: A class variable is visible to all instances of a class and does not vary from instance to instance. While it normally behaves like a constant, in some situations a class variable can be modified. But when it is, the change takes effect for the entire class.

Structuring Classes with Inheritance and Polymorphism

1. Explain that most object-oriented languages require the programmer to master the following techniques: *data encapsulation*, *inheritance*, and *polymorphism*.

2. Stress that while inheritance and polymorphism are built into Python, its syntax does not enforce data encapsulation.

Teaching Tip	"Python does not really support encapsulation because it does not support data hiding through private and protected members. However some pseudo-encapsulation can be done. If an identifier begins with a double underline, i.e. `__a`, then it can be referred to within the class itself as `self.__a`, but outside of the class, it is named `instance._classname__a`. Therefore, while it can prevent accidents, this pseudo-encapsulation cannot really protect data from hostile code." *Reference:* www.astro.ufl.edu/~warner/prog/python.html.

Inheritance Hierarchies and Modeling

1. Use Figure 8.3 to introduce the role of *inheritance hierarchies*.

2. Explain that, in Python, all classes automatically extend the built-in `object` class.

3. Explain how inheritance provides an abstraction mechanism that allows programmers to avoid writing redundant code.

Example: A Restricted Savings Account

1. Use the `RestrictedSavingsAccount` class example to show how to extend an existing class while overloading some of the methods of the parent class.

Example: The Dealer and a Player in the Game of Blackjack

1. Use Figure 8.4 to describe the role of the classes in the blackjack game application.

2. Use the sample code provided in the book to show how the `Player` class can be used to simulate a blackjack player.

3. Use the `Dealer` class example to show how to extend an existing class while overloading some of the methods of the parent class. Step through the code of this class, explaining it to students.

4. Use the sample code provided in the book to show how the `Blackjack` class can be used to set up and manage the interactions with the user. Step through the code of this class, explaining it to students.

Polymorphic Methods

1. Explain that we subclass when two classes share a substantial amount of *abstract behavior*. Point out that a subclass usually adds something extra, such as a new method or data attribute, to the state provided by its superclass.

2. Describe the role of *polymorphic methods* in enabling the two subclasses to have the same interface while performing different operations when the polymorphic method is called.

Abstract Classes

1. Use Figure 8.5 to explain that an *abstract class* includes data and methods common to its subclasses but is never instantiated. Explain how an abstract class differs from a *concrete class*, which can be instantiated.

Teaching Tip	"Abstract classes and interfaces are not supported in Python. In Python, there is no difference between an abstract class and a concrete class. Abstract classes create a template for other classes to extend and use. Instances can not be created of abstract classes but they are very useful when working with several objects that share many characteristics. For instance, when creating a database of people, one could define the abstract class `Person`, which would contain basic attributes and functions common to all people in the database. Then child classes such as `SinglePerson`, `MarriedCouple`, or `Athlete` could be created by extending `Person` and adding appropriate variables and functions. The database could then be told to expect every entry to be an object of type `Person` and thus any of the child classes would be a valid entry. In Python, you could create a class `Person` and extend it with the child classes listed above, but you could not prevent someone from instantiating the `Person` class." *Reference:* www.astro.ufl.edu/~warner/prog/python.html.

The Costs and Benefits of Object-Oriented Programming

1. Analyze the advantages and disadvantages of *imperative programming*, *procedural programming*, *functional programming*, and *object-oriented programming*.

2. Explain that with OO programming, although well-designed objects decrease the likelihood that a system will break when changes are made within a component, this technique can sometimes be overused.

Quick Quiz 3

1. _____ restricts the manipulation of an object's state by external users to a set of method calls.
   Answer: Data encapsulation

2. _____ allows a class to automatically reuse and extend the code of similar but more general classes.
   Answer: Inheritance

3. What is polymorphism?
   Answer: Polymorphism is an object-oriented technique that allows several different classes to use the same general method names.

4. True or False: In Python, all classes automatically extend the built-in `object` class.
   Answer: True

Class Discussion Topics

1. Are your students familiar with exceptions in C++ or Java? If so, ask them to compare the exception mechanisms provided by those languages to the one provided by Python (`try-except` statement).

2. Ask your students if they are familiar with other object-oriented languages. Do those languages provide support for data encapsulation? How?

Additional Projects

1. Ask students to do some research on multiple inheritance and Python's support for this object-oriented feature.

2. Some card games may need to use the Jokers in a deck of cards. Modify the `Card` class so that it supports Jokers. Modify the `Deck` class by adding a `hasJokers` method that can be used to test if the current deck has Jokers or not.

Additional Resources

1. Python Basics:
   www.astro.ufl.edu/~warner/prog/python.html

2. Introduction to OOP with Python:
   www.voidspace.org.uk/python/articles/OOP.shtml

3. Defining Classes:
   http://diveintopython.org/object_oriented_framework/defining_classes.html

4. An Introduction to Python: Classes:
   www.penzilla.net/tutorials/python/classes/

5. Classes:
   http://docs.python.org/tut/node11.html

6. Errors and Exceptions:
   http://docs.python.org/tut/node10.html

Key Terms

- **abstract class:** A class that defines attributes and methods for subclasses, but is never instantiated.
- **accessor:** A method used to examine an attribute of an object without changing it.

- **behavior:** The set of actions that a class of objects supports.
- **class:** A description of the attributes and behavior of a set of computational objects.
- **class diagram:** A graphical notation that describes the relationships among the classes in a software system.
- **class variable:** A variable that is visible to all instances of a class and is accessed by specifying the class name.
- **concrete class:** A class that can be instantiated.
- **constructor:** A method that is run when an object is instantiated, usually to initialize that object's instance variables. This method is named _init_ in Python.
- **encapsulation:** The process of hiding and restricting access to the implementation details of a data structure.
- **garbage collection:** The automatic process of reclaiming memory when the data of a program no longer need it.
- **helper:** A method or function used within the implementation of a module or class but not used by clients of that module or class.
- **inheritance:** The process by which a subclass can reuse attributes and behavior defined in a superclass. See also subclass and superclass.
- **instance variable:** Storage for data in an instance of a class.
- **model/view/controller pattern (MVC):** A design plan in which the roles and responsibilities of the system are cleanly divided among data management (model), user interface display (view), and user event-handling (controller) tasks.
- **mutator:** A method used to change the value of an attribute of an object.
- **object-oriented programming:** The construction of software systems that define classes and rely on inheritance and polymorphism.
- **overloading:** The process of using the same operator symbol or identifier to refer to many different functions.
- **parent:** The immediate superclass of a class.
- **pickling:** The process of converting an object to a form that can be saved to permanent file storage.
- **polymorphism:** The property of one operator symbol or method identifier having many meanings. See also overloading.
- **procedural programming:** A style of programming that decomposes a program into a set of methods or procedures.
- **subclass:** A class that inherits attributes and behaviors from another class.
- **superclass:** The class from which a subclass inherits attributes and behavior.
- **Unified Modeling Language (UML):** A graphical notation for describing a software system in various phases of development.