Sams Teach Yourself
Beginning Programming
in 24 Hours

Fourth Edition
Learn computer programming in just 24 one-hour lessons

Free Sample Chapter
Share with Others
Contents at a Glance

Introduction ................................................................. xvii

Part I: Start Programming Today

HOUR 1 Hands-On Programming ........................................ 1
  2 Process and Techniques .............................................. 17
  3 Designing a Program ................................................. 33
  4 Getting Input and Displaying Output ............................. 49
  5 Data Processing with Numbers and Words....................... 63

Part II: Programming Fundamentals

HOUR 6 Controlling Your Programs .................................... 81
  7 Debugging Tools .................................................... 97
  8 Structured Techniques ............................................. 109
  9 Programming Algorithms ......................................... 123

Part III: Java and Object-Oriented Programming

HOUR 10 Programming with Java ..................................... 151
  11 Java’s Details ...................................................... 167
  12 Java has Class ...................................................... 185

Part IV: Web Development with HTML and JavaScript

HOUR 13 HTML5 and CSS3 ............................................. 201
  14 JavaScript ........................................................... 217
  15 Having Fun with JavaScript ....................................... 233
  16 JavaScript and AJAX ............................................... 247

Part V: Other Programming Languages

HOUR 17 SQL ................................................................. 263
  18 Scripting with PHP ............................................... 277
  19 Programming with C and C++ ................................. 309
# Table of Contents

Introduction xvii

**Part I: Start Programming Today**

**HOUR 1: Hands-On Programming**  1
  - Get Ready to Program ................................................. 1
  - What a Computer Program Does .................................. 2
  - Common Programming Misconceptions ............................ 3
  - Many Programs Already Exist ........................................ 5
  - Programmers Are in Demand .......................................... 5
  - The Real Value of Programs ........................................... 6
  - Users Generally Don’t Own Programs ............................... 6
  - Giving Computers Programs ......................................... 6
  - Your First Program ...................................................... 8
  - Clarifying Comments .................................................. 9
  - Entering Your Own Program .......................................... 11
  - Summary .................................................................... 13
  - Q&A ........................................................................ 13
  - Workshop .................................................................. 14

**HOUR 2: Process and Techniques**  17
  - Understanding the Need for Programs ......................... 17
  - Programs, Programs Everywhere ................................. 20
  - Programs as Directions .............................................. 20
  - Summary .................................................................... 30
  - Q&A ........................................................................ 30
  - Workshop .................................................................. 30

**HOUR 3: Designing a Program**  33
  - The Need for Design .................................................. 33
  - User–Programmer Agreement ....................................... 34
### Table of Contents

Write Clear Programs ........................................... 104  
Additional Debugging Techniques .......................... 106  
Summary .......................................................... 106  
Q&A ................................................................... 107  
Workshop ......................................................... 107

**HOUR 8: Structured Techniques** 109  
Structured Programming ...................................... 109  
Packaging Your Python Code into Functions .......... 115  
Testing the Program ............................................ 118  
Profiling Code ................................................... 119  
Getting Back to Programming .............................. 120  
Summary .......................................................... 120  
Q&A ................................................................... 120  
Workshop ......................................................... 121

**HOUR 9: Programming Algorithms** 123  
Counters and Accumulators ................................. 124  
Python Lists ...................................................... 127  
Accumulators for Total ....................................... 130  
Swapping Values .............................................. 131  
Sorting ............................................................ 133  
Searching Lists ................................................ 137  
Taking Functions Further ................................... 144  
Nested Loops ..................................................... 148  
Summary .......................................................... 148  
Q&A ................................................................... 148  
Workshop ......................................................... 149

**Part III: Java and Object-Oriented Programming**

**HOUR 10: Programming with Java** 151  
Introducing Java ................................................ 152  
Java Provides Executable Content ....................... 154  
Seamless Execution ......................................... 155  
Multi-Platform Executable Content .................... 155
# Table of Contents

A Quick HTML Primer .......................................................... 207  
Using CSS to Control How Your Text Looks ............................ 210  
Including Graphics in a Website with HTML ............................ 213  
Summary ............................................................................... 214  
Q&A .................................................................................. 214  
Workshop ........................................................................... 215

**HOUR 14: JavaScript** ......................................................... 217  
Getting Started with JavaScript .................................................. 218  
Using Comments in JavaScript .................................................... 218  
Entering Your First JavaScript Program ........................................ 218  
Printing to the Screen with JavaScript ......................................... 221  
Variables in JavaScript ................................................................ 222  
Getting Keyboard Data with `prompt` ........................................ 222  
Comparing Data with `if` .......................................................... 227  
Looping Statements ................................................................... 227  
Summary ............................................................................... 227  
Q&A .................................................................................. 230  
Workshop ........................................................................... 231

**HOUR 15: Having Fun with JavaScript** ................................. 233  
Rotating Images on a Page ........................................................ 233  
Capturing the Position of the Mouse ........................................... 239  
Adding a Repeating News Ticker to Your Website ................. 241  
Summary ............................................................................... 244  
Q&A .................................................................................. 245  
Workshop ........................................................................... 245

**HOUR 16: JavaScript and AJAX** ......................................... 247  
Introducing AJAX .................................................................... 247  
Using `XMLHttpRequest` ....................................................... 251  
Creating a Simple AJAX Library ............................................. 253  
Creating an AJAX Quiz Using the Library ............................... 254  
Summary ............................................................................... 259  
Q&A .................................................................................. 259  
Workshop ........................................................................... 260
Part V: Other Programming Languages

**HOUR 17: SQL**
- Relational Databases ................................................. 263
- Basic SQL Queries ...................................................... 266
- Retrieving Records from a Database ............................ 266
- Inserting and Updating Database Records .................. 269
- Deleting Records from a Database .............................. 271
- Adding, Deleting, or Modifying the Fields in an Existing Table 272
- Summary ......................................................................... 273
- Q&A .................................................................................. 274
- Workshop ......................................................................... 274

**HOUR 18: Scripting with PHP**
- What You Need for PHP Programming ......................... 278
- Basic Structures in PHP Scripts ..................................... 279
- Looping ............................................................................ 284
- The Building Blocks of PHP: Variables, Data Types, and Operators 286
- Using and Creating Functions in PHP ......................... 295
- Working with Objects in PHP ......................................... 300
- Common Uses of PHP ....................................................... 304
- Summary ......................................................................... 305
- Q&A .................................................................................. 305
- Workshop ......................................................................... 306

**HOUR 19: Programming with C and C++**
- Introducing C ................................................................. 309
- What You Need for C and C++ Programming ............... 311
- Looking at C ..................................................................... 311
- C Data ............................................................................... 313
- C Functions ...................................................................... 314
- C Operators ..................................................................... 321
- C Control Statements Mimic Python ............................ 321
- Learning C++ ................................................................. 322
- Object Terminology ....................................................... 322
<table>
<thead>
<tr>
<th>Table of Contents</th>
<th>xi</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fundamental Differences Between C and C++</strong></td>
<td>323</td>
</tr>
<tr>
<td>Introducing C++ Objects</td>
<td>324</td>
</tr>
<tr>
<td>Things to Come</td>
<td>329</td>
</tr>
<tr>
<td>Summary</td>
<td>331</td>
</tr>
<tr>
<td>Q&amp;A</td>
<td>331</td>
</tr>
<tr>
<td>Workshop</td>
<td>331</td>
</tr>
<tr>
<td><strong>HOUR 20: Programming with Visual Basic 2019</strong></td>
<td>333</td>
</tr>
<tr>
<td>Reviewing the Visual Basic Screen</td>
<td>333</td>
</tr>
<tr>
<td>Creating a Simple Application from Scratch</td>
<td>335</td>
</tr>
<tr>
<td>Other Visual Basic Programming Considerations</td>
<td>342</td>
</tr>
<tr>
<td>Your Next Step</td>
<td>344</td>
</tr>
<tr>
<td>Summary</td>
<td>345</td>
</tr>
<tr>
<td>Q&amp;A</td>
<td>345</td>
</tr>
<tr>
<td>Workshop</td>
<td>345</td>
</tr>
<tr>
<td><strong>HOUR 21: C# and the .NET Core</strong></td>
<td>347</td>
</tr>
<tr>
<td>Understanding the Purpose of .NET</td>
<td>347</td>
</tr>
<tr>
<td>The Common Language Runtime</td>
<td>348</td>
</tr>
<tr>
<td>The Framework Class Library</td>
<td>349</td>
</tr>
<tr>
<td>Parallel Computing Platform</td>
<td>350</td>
</tr>
<tr>
<td>Dynamic Language Runtime</td>
<td>350</td>
</tr>
<tr>
<td>The C# Language</td>
<td>350</td>
</tr>
<tr>
<td>Summary</td>
<td>359</td>
</tr>
<tr>
<td>Q&amp;A</td>
<td>359</td>
</tr>
<tr>
<td>Workshop</td>
<td>360</td>
</tr>
<tr>
<td><strong>Part VI: The Business of Programming</strong></td>
<td></td>
</tr>
<tr>
<td><strong>HOUR 22: How Companies Program</strong></td>
<td>361</td>
</tr>
<tr>
<td>Data Processing and Information Technology Departments</td>
<td>361</td>
</tr>
<tr>
<td>Computer-Related Jobs</td>
<td>365</td>
</tr>
<tr>
<td>Job Titles</td>
<td>366</td>
</tr>
<tr>
<td>Structured Walkthroughs</td>
<td>371</td>
</tr>
<tr>
<td>Putting a Program into Production</td>
<td>372</td>
</tr>
</tbody>
</table>
About the Author

**Greg Perry** is a speaker and writer in both the programming and applications sides of computing. He is known for bringing programming topics down to the beginner’s level. Perry has been a programmer and trainer for two decades. He received his first degree in computer science and then earned a Master’s degree in corporate finance. Besides writing, he consults and lectures across the country, including at the acclaimed Software Development programming conferences. Perry is the author of more than 75 other computer books. In his spare time, he gives lectures on traveling in Italy, his second-favorite place to be.

**Dean Miller** is a writer and editor with more than 20 years of experience in both the publishing and licensed consumer products businesses. Over the years, he has created or helped shape a number of bestselling books and series, including *Sams Teach Yourself in 21 Days*, *Sams Teach Yourself in 24 Hours*, and the *Unleashed* series, all from Sams Publishing. He has written or cowritten 15 books on computer programming and professional wrestling and is still looking for a way to combine the two into one strange amalgam.
Dedication

Dean: To Fran, Margaret, John, and Alice—Thanks for being the absolute best family someone could ask for.

Acknowledgments

Greg: My thanks go to all my friends at Pearson. Most writers would refer to them as editors; to me, they are friends. I want all my readers to understand this: The people at Pearson care about you most of all. The things they do result from their concern for your knowledge and enjoyment. On a more personal note, my beautiful bride, Jayne; my mother Bettye Perry; and my friends, who wonder how I find the time to write, all deserve credit for supporting my need to write.

Dean: I’d like to thank Greg Perry for creating outstanding book that continues to educate generations of new computer programmers. It’s been a highlight of my career to work with him as both his editor and co-author over the years. Thanks to Kim Spenceley for working with me to create this new edition. I appreciate the amazing work Lori Lyons, Kitty Wilson, and the production team at Pearson put into this book. Special thanks to my technical reviewers, John Fonte and Michael Garcia, for improving the quality of the book with their thorough reads. On a personal level, I have to thank my three children, John, Alice, and Maggie, and my wife Fran for their unending patience and support.
This page intentionally left blank
Learning how to program computers is easier than you might think. If you approach computers with hesitation, if you cannot even spell PC, if you have tried your best to avoid the subject altogether but can do so no longer, the book you now hold contains support that you can depend on in troubled computing times.

This 24-hour tutorial does more than explain programming. This tutorial does more than describe the difference between JavaScript, C++, and Java. This tutorial does more than teach you what programming is all about. This tutorial is a training tool that you can use to develop proper programming skills. The aim of this text is to introduce you to programming using professionally recognized principles, while keeping things simple at the same time. It is not this text’s singular goal to teach you a programming language (although you will be writing programs before you finish it). This text’s goal is to give you the foundation to become the best programmer you can be.

These 24 one-hour lessons delve into proper program design principles. You’ll not only learn how to program, but also how to prepare for programming. This tutorial also teaches you how companies program and explains what you have to do to become a needed resource in a programming position. You’ll learn about various programming job titles and what to expect if you want to write programs for others. You’ll explore many issues related to online computing and learn how to address the needs of the online programming community.

**Who Should Use This Book?**

The title of this book says it all. If you have never programmed a computer, if you don’t even like them at all, or if updating the operating system of your phone throws you into fits, take three sighs of relief! This text was written for you so that, within 24 hours, you will understand the nature of computer programs and you will have written programs.

This book is aimed at three different groups of people:

- Individuals who know nothing about programming but who want to know what programming is all about.
Companies that want to train nonprogramming computer users for programming careers.

Schools—both for introductory language classes and for systems analysis and design classes—that want to promote good coding design and style and that want to offer an overview of the life of a programmer.

Readers who seem tired of the plethora of quick-fix computer titles cluttering today's shelves will find a welcome reprieve here. The book you now hold talks to newcomers about programming without talking down to them.

**What This Book Will Do for You**

In the next 24 hours, you will learn something about almost every aspect of programming. The following topics are discussed in depth throughout this 24-hour tutorial:

- The hardware and software related to programming
- The history of programming
- Programming languages
- The business of programming
- Programming jobs
- Program design
- Internet programming
- The future of programming

**Can This Book Really Teach Programming in 24 Hours?**

In a word, yes. You can master each chapter in one hour or less. (By the way, chapters are referred to as “hours” or “lessons” in the rest of this book.) The material is balanced with mountains of shortcuts and methods that will make your hours productive and hone your programming skills more and more with each hour. Although some chapters are longer than others, many of the shorter chapters cover more detailed or more difficult issues than the shorter ones. A true attempt was made to make each hour learnable in an hour. Exercises at the end of each hour will provide feedback about the skills you learned.
Conventions Used in This Book

This book uses several common conventions to help teach programming topics. Here is a summary of those typographical conventions:

- Commands and computer output appear in a special monospaced computer font. Sometimes a line of code will be too long to fit on one line in this book. The code continuation symbol (➥) indicates that the line continues.
- Words you type also appear in the monospaced computer font.
- If a task requires you to select from a menu, the book separates menu commands with a comma. Therefore, this book uses File, Save As to select the Save As option from the File menu.

In addition to typographical conventions, the following special elements are included to set off different types of information to make it easily recognizable.

TRY IT YOURSELF

The best way to learn how to program is to jump right in and start programming. These Try it Yourself sections will teach you a simple concept or method to accomplish a goal programmatically. The listing will be easy to follow and then the programs’ output will be displayed along with coverage of key points in the program. To really get some practice, try altering bits of the code in each of these sections in order to see what your tweaks accomplish.

NOTE

Special notes augment the material you read in each hour. These notes clarify concepts and procedures.

TIP

You’ll find numerous tips that offer shortcuts and solutions to common problems.

CAUTION

The cautions warn you about pitfalls. Reading them will save you time and trouble.
Programmers learn to develop patience early in their programming careers. They learn that proper design is critical to a successful program. Perhaps you have heard the term systems analysis and design. This is the name given to the practice of analyzing a problem and then designing a program from that analysis. Complete books and college courses have been dedicated to systems analysis and design. Of course, you want to get back to hands-on programming—and you’ll be doing that very soon. However, to be productive at hands-on programming, you need to understand the importance of design. This chapter covers program design highlights, letting you see what productive computer programmers go through before writing programs.

The highlights of this hour include the following:

- Understanding the importance of program design
- Mastering the three steps required to write programs
- Using output definition
- Comparing top-down and bottom-up designs
- Seeing how flowcharts and pseudocode are making room for RAD
- Preparing for the final step in the programming process

The Need for Design

A builder who begins to build a house doesn’t pick up a hammer and begin on the kitchen’s frame. A designer must design the new house before anything can begin to be built. As you will soon see, a program should also be designed before it is written.

A builder must first find out what the purchasers of the house want. Nothing can be built unless the builder has an end result in mind. Therefore, the buyers of the house must meet with an architect. They tell the architect what they want the house to look like. The architect helps the buyers decide by telling them what is possible and what isn’t. During this initial stage, the price is always a factor that requires the designers and the purchasers to reach compromise agreements.
After the architect completes the plans for the house, the builder must plan the resources needed to build the house. Only after the design of the house is finished, the permits are filed, the money is in place, the materials are purchased, and the laborers are hired can any physical building begin. As a matter of fact, the more effort the builder puts into these preliminary requirements, the faster the house can actually be built.

The problem with building a house before it is properly designed is that the eventual owners may want changes made after it is too late to change them. It is very difficult to add a bathroom in the middle of two bedrooms after the house is completed. The goal is to get the owners to agree with the builder on the design of the house prior to construction. When the specifications are agreed on by all the parties involved, there is little room for disagreement later. The clearer the initial plans are, the fewer problems down the road because all parties agreed on the same house plans.

Sure, this is not a book on house construction, but this example provides a good analogy for writing programs of any great length. You should not go to the keyboard and start typing instructions into the editor before designing the program any more than a builder should pick up a hammer before the house plans are finalized.

**TIP**
The more up-front design work that you do, the faster you will finish the final program.

Thanks to computer technology, a computer program is easier to modify than a house. If you leave out a routine that a user wanted, you can add it later more easily than a builder can add a room to a finished house. Nevertheless, adding something to a program is never as easy as designing the program correctly the first time.

**User–Programmer Agreement**

Suppose you accept a job as a programmer for a small business that wants to create sales and inventory software. (After you’ve gone through these 24 hours, you’ll understand programming better, and you’ll even learn how to write programs in Python or be able to switch to another language.) The changes that the owners want sound simple. They want you to write some interactive Python routines that enable them to look at existing inventory and to print what products have sold in the past day, week, month, or year.

So, you listen to what they want, you agree to a price for your services, you get an advance payment, you plan out the software, and you go to your home office to begin the work. After some grueling months of work, you bring your masterpiece program back to show the owners.

“Looks good,” they say. “But where is the report that breaks down credit card versus cash purchases? Where can we check in-store versus warehouse inventory? Where does the program list
the products we’ve back-ordered and that are unavailable? Why can’t the program total sales tax we’ve collected anywhere?”

You’ve just learned a painful lesson about user–programmer agreements. The users did a lousy job at explaining what they wanted. In fairness to them, you didn’t do a great job at pulling out of them what they needed. Both of you thought you knew what you were supposed to do, and neither knew in reality. You realize that the price you quoted them originally will pay for about 10% of the work this project requires.

Before you start a job and before you price a job, you must know what your users want. Learning this is part of the program design experience. You need to know every detail before you’ll be able to price your service accurately and before you’ll be able to make customers happy.

NOTE
Proper user–programmer agreement is vital for all areas of programming, not just for contract programmers. If you work for a corporation as a programmer, you also will need to have detailed specifications before you can begin your work. Other corporate users who will use the system must sign off on what they want so that everybody knows up front what is expected. If the user comes back to you later and asks why you didn’t include a feature, you will be able to answer, “Because we never discussed that feature. You approved specifications that never mentioned that feature.”

The program maintenance that takes place after the program is written, tested, and distributed is one of the most time-consuming aspects of the programming process. Programs are continually updated to reflect new user needs. Sometimes, if the program is not designed properly before it is written, the user will not want the program until it does exactly what the user wants it to do.

Computer consultants learn early to get the user’s acceptance—and even the user’s signature—on a program’s design before the programming begins. If both the user and the programmers agree on what to do, there is little room for argument when the final program is presented. Company resources are limited; there is no time to add something later that should have been in the system all along.

Steps to Design

There are three fundamental steps you should perform when you have a program to write:

1. Define the output and data flows.
2. Develop the logic to get to that output.
3. Write the program.
Notice that writing the program is the last step in writing the program. This is not as silly as it sounds. Remember that physically building the house is the last stage of building the house; proper planning is critical before any actual building can start. You will find that writing and typing in the lines of a program is one of the easiest parts of the programming process. If your design is well thought out, the program practically writes itself; typing it in becomes almost an afterthought to the whole process.

**Step 1: Define the Output and Data Flows**

Before beginning a program, you must have a firm idea of what the program should produce and what data is needed to produce that output. Just as a builder must know what the house should look like before beginning to build it, a programmer must know what the output is going to be before writing the program. Anything that the program produces and the user sees is considered output that you must define. You must know what every screen in the program should look like and what will be on every page of every printed report.

Some programs are rather small, but without knowing where you’re heading, you might take longer to finish the program than you would if you first determined the output in detail. Suppose you wanted to add a Python-based program that allowed a small business to record and store customer contact information. To start, you should make a list of all fields that the program is to produce onscreen. You would not only list each field but also describe the fields. Table 3.1 details the fields on the program’s window.

**TABLE 3.1** Fields that your contact management program might display

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contacts</td>
<td>Scrolling list</td>
<td>Displays the list of contacts</td>
</tr>
<tr>
<td>Name</td>
<td>Text field</td>
<td>Holds contact’s name</td>
</tr>
<tr>
<td>Address</td>
<td>Text field</td>
<td>Holds contact’s address</td>
</tr>
<tr>
<td>City</td>
<td>Text field</td>
<td>Holds contact’s city</td>
</tr>
<tr>
<td>State</td>
<td>Text field</td>
<td>Holds contact’s state</td>
</tr>
<tr>
<td>Zip</td>
<td>Text field</td>
<td>Holds contact’s zip code</td>
</tr>
<tr>
<td>Home Phone #</td>
<td>Text field</td>
<td>Holds contact’s phone number</td>
</tr>
<tr>
<td>Cell Phone #</td>
<td>Text field</td>
<td>Holds contact’s mobile number</td>
</tr>
<tr>
<td>Email</td>
<td>Text field</td>
<td>Holds contact’s email address</td>
</tr>
<tr>
<td>Stage</td>
<td>Fixed, scrolling list</td>
<td>Displays a list of possible stages this contact might reside in, such as being offered a special follow-up call or perhaps the initial contact</td>
</tr>
<tr>
<td>Notes</td>
<td>Text field</td>
<td>Miscellaneous notes about the contact, such as whether the contact has bought from the company before</td>
</tr>
</tbody>
</table>
### Field Type Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter Contacts</td>
<td>Fixed, scrolling list</td>
<td>Enables the user to search for groups of contacts based on the stage the contacts are in so that the user can see a list of all contacts who have been sent a mailing</td>
</tr>
<tr>
<td>Edit</td>
<td>Function</td>
<td>Enables the user to modify an existing contact</td>
</tr>
<tr>
<td>Add</td>
<td>Function</td>
<td>Enables the user to add a new contact</td>
</tr>
<tr>
<td>Delete</td>
<td>Function</td>
<td>Enables the user to delete an existing contact</td>
</tr>
</tbody>
</table>

Many of the fields you list in an output definition may be obvious. The field called Name obviously will hold and display a contact’s name. Being obvious is okay. Keep in mind that if you write programs for other people, as you often will do, you must get approval of your program’s parameters. One of the best ways to begin is to make a list of all the intended program’s fields and make sure that the user agrees that everything is there. Perhaps your client has specific interests, like wanting the Twitter handle of contacts as well. By communicating with your client, you will get a better idea of what you need to add to the program.

As you’ll see later this hour, in the section “Rapid Application Development,” you’ll be able to use programs to put together a model of the actual output screen that your users can see. With the model and with your list of fields, you have double verification that the program contains exactly what the user wants.

Input windows such as the Contacts program data-entry screen are part of your output definition. This may seem contradictory, but input screens require that your program place fields on the screen, and you should plan where these input fields must go.

The output definition is more than a preliminary output design. It gives you insight into what data elements the program should track, compute, and produce. Defining the output also helps you gather all the input you need to produce the output.

**CAUTION**

Some programs produce a huge amount of output. Don’t skip this first all-important step in the design process just because there is a lot of output. With more output, it becomes more important for you to define it. Defining the output is relatively easy—sometimes even downright boring and time-consuming. The time you need to define the output can take as long as typing in the program. You will lose that time and more, however, if you shrug off the output definition at the beginning.

The output definition consists of many pages of details. You must be able to specify all the details of a problem before you know what output you need. Even command buttons and scrolling list boxes are output because the program will display these items.
In Hour 1, “Hands-On Programming,” you learned that data goes into a program, and the program outputs meaningful information. You should inventory all the data that goes into a program. If you're using Python to make a customer contact program, you need to know what specific data the owners want to collect from the users. Define what each piece of data is. Perhaps the owners want to ask customers whether they want to submit a name and email address for the weekly sales email blast. Does the company want any additional data from the user, such as physical address, age, and income?

Object-Oriented Design

Throughout this 24-hour tutorial, you will learn what object-oriented programming (OOP) is all about. Basically, OOP turns data values, such as names and prices, into objects that can take on a life of their own inside programs. Part III, “Java and Object-Oriented Programming,” covers the basics of OOP.

A few years ago, some OOP experts developed a process for designing OOP programs called object-oriented design (OOD). OOD made an advanced science out of specifying data to be gathered in a program and defining that data in a way that was appropriate for the special needs of OOP programmers. Grady Booch was one of the founders of OOD. His specifications from almost three decades ago continue to help OOP programmers collect data for the applications they are about to write and to turn that data into objects for programs.

In Hour 4, “Getting Input and Displaying Output,” you’ll learn how to put these ideas into a program. You will learn how a program asks for data and produces information on the screen. This I/O (input/output) process is the most critical part of an application. You want to capture all data required and in an accurate way.

Something is still missing in all this design discussion. You understand the importance of gathering data. You understand the importance of knowing where you’re headed by designing the output. But how do you go from data to output? That’s the next step in the design process: You need to determine what processing will be required to produce the output from the input (data). You must be able to generate proper data flows and calculations so that your program manipulates that data and produces the correct output. The final sections of this hour discuss ways to develop the centerpiece—the logic for your programs.

All output screens, printed reports, and data-entry screens must be defined in advance so you know exactly what is required of your programs. You must also decide what data to keep in files and the format of your data files. As you progress in your programming education, you will learn ways to lay out data files in appropriate formats.

When capturing data, you want to gather data from users in a way that is reasonable, requires little time, and has prompts that request the data in a friendly and unobtrusive manner. Prototyping (discussed next) and rapid application development can help.
Prototyping

In the days of expensive hardware and costly computer usage time, the process of system design was, in some ways, more critical than it is today. The more time you spent designing your code, the smoother the costly hands-on programming became. This is far less true today because computers are inexpensive, and you have much more freedom to change your mind and add program options than before. Yet the first part of this hour was spent in great detail explaining why up-front design is critical.

The primary problem many new programmers have today is that they do absolutely no design work. That’s why many problems take place, such as the one mentioned earlier this hour about the company that wanted far more in its program than the programmer ever dreamed of.

Although the actual design of output, data, and even the logic in the body of the program itself is much simpler to work with, given the power and low cost of today’s computing tools, you still must maintain an eagle eye toward developing an initial design with agreed-upon output from your users. You must also know all the data that your program is to collect before you begin your coding. If you don’t, you will have a frustrating time as a contract programmer or as a corporate programmer because you’ll constantly be playing catch-up with what the users actually want and failed to tell you about.

One of the benefits of the Windows operating system is its visual nature. Before Windows, programming tools were limited to text-based design and implementation. Designing a user’s screen today means starting with a programming language such as Visual Basic, drawing the screen, and dragging to the screen objects that the user will interact with, such as an OK button. Therefore, you can quickly design prototype screens that you can send to the user. A prototype is a model, and a prototype screen models what the final program’s screen will look like. After the user sees the screens that he or she will interact with, the user will have a much better feel for whether you understand the needs of the program.

Many Windows programming languages, such as Visual C++ and Visual Basic, include prototyping tools. For comparison, Figure 3.1 shows the Visual Basic development screen. The language covered in these early chapters, Python, is more likely to help you behind the scenes, working with the data and analyzing it as needed. You can certainly perform input and output functions with Python, but if you are developing a Windows application, other languages are more appropriate, such as what you see in Figure 3.1. The screen looks rather busy, but the important things to look for are the Toolbox and the output design window. To place controls such as command buttons and text boxes on the form that serves as the output window, the programmer only has to drag that control from the Toolbox window to the form. So, to build a program’s output, the programmer only has to drag as many controls as needed to the form and does not have to write a single line of code in the meantime.
Once you place controls on a form window with a programming tool such as Visual Basic, you can do more than show the form to your users. You actually can compile the form, just as you would a program, and let your user interact with the controls. When the user is able to work with the controls, even though nothing happens as a result, the user is better able to tell if you understand the goals of the program. The user often notices if there is a missing piece of the program and can also offer suggestions to make the program flow more easily from a user’s point of view.

CAUTION
A prototype is often only an empty shell that cannot do anything except simulate user interaction until you tie its pieces together with code. Your job as a programmer has only just begun once you get approval on the screens, but the screens are the first place to begin because you must understand what your users want in order to know how to proceed.
Rapid Application Development

A more advanced program design tool used for defining output, data flows, and logic itself is called rapid application development, or RAD for short. RAD is the process of quickly placing controls on a form—not unlike you just saw done with Visual Basic—connecting those controls to data, and accessing pieces of prewritten code to put together a fully functional application without writing a single line of code. In a way, programming systems such as Visual Basic are fulfilling many goals of RAD. When you place controls on a form, as you’ll see done in far more detail in Hour 20, “Programming with Visual Basic 2012,” the Visual Basic system handles all the programming needed for that control. You don’t ever have to write anything to make a command button act like a command button should. Your only goal is to determine how many command buttons your program needs and where they are to go.

But these tools cannot read your mind. RAD tools do not know that, when the user clicks a certain button, a report is supposed to print. Programmers are still needed to connect all these things to each other and to data, and programmers are needed to write the detailed logic so that the program processes data correctly. Before these kinds of program development tools appeared, programmers had to write thousands of lines of code, often in the C programming language, just to produce a simple Windows program. At least now the controls and the interface are more rapidly developed. Perhaps someday a RAD tool will be sophisticated enough to develop the logic also. But in the meantime, don’t quit your day job if your day job is programming, because you’re still in demand.

TIP
Teach your users how to prototype their own screens! Programming knowledge is not required to design the screens. Your users, therefore, will be able to show you exactly what they want. The prototyped screens are interactive as well. That is, your users will be able to click the buttons and enter values in the fields even though nothing happens as a result of that use. The idea is to let your users try the screens for a while to make sure they are comfortable with the placement and appearance of the controls.

Top-Down Program Design

For large projects, many programming staff members find that a top-down design helps them focus on what a program needs and helps them detail the logic required to produce the program’s results. Top-down design is the process of breaking down a problem into more and more detail until you finalize all the details. With top-down design, you produce the details needed to accomplish a programming task.
The problem with top-down design is that programmers tend not to use it. They tend to design from the opposite direction (called bottom-up design). When you ignore top-down design, you impose a heavy burden on yourself to remember every detail that will be needed; with top-down design, the details fall out on their own. You don’t have to worry about the petty details if you follow a strict top-down design because the process of top-down design takes care of producing the details.

TIP
One of the keys to top-down design is that it forces you to put off the details until later. Top-down design forces you to think in terms of the overall problem for as long as possible. Top-down design keeps you focused. If you use bottom-up design, it is easy to lose sight of the forest for the trees. You get to the details too fast and lose sight of your program’s primary objectives.

Top-down design involves a three-step process:

1. Determine the overall goal.
2. Break that goal into two, three, or more detailed parts. Don’t add too many details, or you might leave things out.
3. Keep repeating steps 1 and 2—and put off the details as long as possible—until you cannot reasonably break down the problem any further.

You can learn about top-down design more easily by relating it to a common real-world problem before looking at a computer problem. Top-down design is not just for programming problems. Once you master top-down design, you can apply it to any part of your life that you must plan in detail. Perhaps the most detailed event that a person can plan is a wedding. Therefore, a wedding is the perfect place to see top-down design in action.

What is the first thing you must do to have a wedding? First, find a prospective spouse. (You’ll need a different book for help with that.) When it comes time to plan the wedding, the top-down design is the best way to approach the event. The way not to plan a wedding is to worry about the details first, yet this is the way most people plan a wedding. They start thinking about the dresses, the organist, the flowers, and the cake to serve at the reception. The biggest problem with trying to cover all these details from the beginning is that you lose sight of so much; it is too easy to forget a detail until it’s too late. The details of bottom-up design get in your way.

What is the overall goal of a wedding? Thinking in the most general terms possible, “Have a wedding” is about as general as it can get. If you were in charge of planning a wedding, the general goal of “Have a wedding” would put you right on target. Assume that “Have a wedding” is the highest-level goal.
NOTE

The overall goal keeps you focused. Despite its redundant nature, “Have a wedding” keeps out details such as planning the honeymoon. If you don’t put a fence around the exact problem you are working on, you’ll get mixed up with details and, more importantly, you’ll forget some details. If you’re planning both a wedding and a honeymoon, you should do two top-down designs or include the honeymoon trip in the top-level general goal. This wedding plan includes the event of the wedding—the ceremony and reception—but doesn’t include any honeymoon details. (Leave the honeymoon details to your spouse so you can be surprised. After all, you have enough to do with the wedding plans, right?)

Now that you know where you’re heading, begin by breaking down the overall goal into two or three details. For instance, what about the colors of the wedding, what about the guest list, what about paying the officiant... oops, too many details! The idea of top-down design is to put off the details for as long as possible. Don’t get in a hurry. When you find yourself breaking the current problem into more than three or four parts, you are rushing the top-down design. Put off the details. Basically, you can break down “Have a wedding” into the following two major components: the ceremony and the reception.

The next step of top-down design is to repeat the same process with the new components. The ceremony is made up of the people and the location. The reception includes the food, the people, and the location. The ceremony’s people include the guests, the wedding party, and the workers (officiant, organist, and so on—but those details come a little later).

TIP

Don’t worry about the time order of the details yet. The goal of top-down design is to produce every detail you need (eventually), not to put those details into any order. You must know where you are heading and exactly what is required before considering how those details relate to each other and which ones come first.

Eventually, you will have several pages of details that cannot be broken down any further. For instance, you’ll probably end up with the details of the reception food, such as peanuts for snacking. (If you start out listing those details, however, you could forget many of them.)

Now move to a more computerized problem; assume that you are assigned the task of writing a payroll program for a company. What would that payroll program require? You could begin by listing the payroll program’s details, such as:

- Print payroll checks.
- Calculate federal taxes.
- Calculate state taxes.
What is wrong with this approach? If you said that the details were coming too early, you are correct. The perfect place to start is at the top. The most general goal of a payroll program might be “Perform the payroll.” This overall goal keeps other details out of this program (no general ledger processing will be included, unless part of the payroll system updates a general ledger file) and keeps you focused on the problem at hand.

Consider Figure 3.2. This might be the first page of the payroll’s top-down design. Any payroll program has to include some mechanism for entering, deleting, and changing employee information such as address, city, state, zip code, number of exemptions, and so on. What other details about the employees do you need? At this point, don’t answer that question. The design is not ready for all those details.

![Figure 3.2: The first page of the payroll program's top-down design would include the highest level of details.](image-url)

There is a long way to go before you finish with the payroll top-down design, but Figure 3.2 is the first step. You must keep breaking down each component until the details finally appear.

Only when you and the user gather all the necessary details through top-down design can you decide what is going to comprise those details.

**Step 2: Develop the Logic**

After you and the user agree to the goals and output of the program, the rest is up to you. Your job is to use that output definition to decide how to make a computer produce the output. You have broken down the overall problem into detailed instructions that the computer can carry out. This doesn’t mean you are ready to write the program—quite the contrary. You are now ready to develop the logic that produces that output.

The output definition goes a long way toward describing what the program is supposed to do. Now you must decide how to accomplish the job. You must order the details that you have so they operate in a time-ordered fashion. You must also decide which decisions your program must make and the actions produced by each of those decisions.
Throughout the rest of this 24-hour tutorial, you’ll learn the final two steps of developing programs. You will gain insight into how programmers write and test a program after developing the output definition and getting the user’s approval on the program’s specifications.

CAUTION
Only after learning to program can you learn to develop the logic that goes into a program, yet you must develop some logic before writing programs to be able to move from the output and data definition stage to the program code. This “chicken before the egg” syndrome is common for newcomers to programming. When you begin to write your own programs, you’ll have a much better understanding of logic development.

In the past, users would use tools such as flowcharts and pseudocode to develop program logic. A flowchart is shown in Figure 3.3. It is said that a picture is worth a thousand words, and the flowchart provides a pictorial representation of program logic. The flowchart doesn’t include all the program details but represents the general logic flow of the program. If your flowchart is correctly drawn, writing the actual program becomes a matter of rote. After the final program is completed, the flowchart can act as documentation for the program.

Flowcharts are made up of industry-standard symbols. Plastic flowchart symbol outlines, called flowchart templates, are still available at office supply stores to help you draw better-looking flowcharts instead of relying on freehand drawing. There are also some programs that guide you through the creation of a flowchart and enable you to print flowcharts on your printer.

Although some still use flowcharts today, RAD and other development tools have virtually eliminated flowcharts except for depicting isolated parts of a program’s logic for documentation purposes. Even in its heyday in the 1960s and 1970s, flowcharting did not completely catch on. Some companies preferred another method for logic description called pseudocode, sometimes called structured English, which involves writing logic using sentences of text instead of the diagrams used in flowcharting.

Pseudocode doesn’t have any programming language statements in it, but it also is not free-flowing English. It is a set of rigid English words that allow for the depiction of logic you see so often in flowcharts and programming languages. As with flowcharts, you can write pseudocode for anything, not just computer programs. A lot of instruction manuals use a form of pseudocode to illustrate the steps needed to assemble parts. Pseudocode offers a rigid description of logic that tries to leave little room for ambiguity.
Here is the logic for the payroll problem in pseudocode form. Notice that you can read the text, yet it is not a programming language. The indentation helps keep track of which sentences go together. The pseudocode is readable by anyone, even by people unfamiliar with flowcharting symbols:

For each employee:
   If the employee worked 0 to 40 hours then
      net pay equals hours worked times rate.
   Otherwise,
      if the employee worked between 40 and 50 hours then
         net pay equals 40 times the rate;
         add to that (hours worked - 40) times the rate times 1.5.
      Otherwise,
         net pay equals 40 times the rate;
         add to that 10 times the rate times 1.5;
         add to that (hours worked - 50) times twice the rate.
   Deduct taxes from the net pay.
Print the paycheck.
Step 3: Writing the Code

The program writing takes the longest to learn. After you learn to program, however, the actual programming process takes less time than the design if your design is accurate and complete. The nature of programming requires that you learn some new skills. The next few hourly lessons will teach you a lot about programming languages and will help train you to become a better coder so that your programs will not only achieve the goals they are supposed to achieve but also will be simple to maintain.

Summary

A builder doesn’t build a house before designing it, and a programmer should not write a program without designing it either. Too often, programmers rush to the keyboard without thinking through the logic. A badly designed program results in lots of bugs and maintenance. This hour describes how to ensure that your program design matches the design that the user wants. After you complete the output definition, you can organize the program’s logic using top-down design, flowcharts, and pseudocode.

The next hour focuses on training you in your first computer language, Python.

Q&A

Q. At what point in the top-down design should I begin to add details?

A. Put off the details as long as possible. If you were designing a program to produce sales reports, you would not enter the printing of the final report total until you had completed all the other report design tasks. The details fall out on their own when you can no longer break a task into two or more other tasks.

Q. Once I break the top-down design into its lowest-level details, don’t I also have the pseudocode details?

A. The top-down enables you to determine all the details your program will need. The top-down design doesn’t, however, put those details into their logical execution order. The pseudocode dictates the executing logic of your program and determines when things happen, the order in which they happen, and when they stop happening. The top-down design simply determines everything that might happen in the program. Instead of using pseudocode, however, you should consider getting a RAD tool that will help you move more quickly from the design to the finished, working program. Today’s RAD systems are still rather primitive, and you’ll have to add much of the code yourself.
Workshop
The quiz questions are provided for your further understanding.

Quiz
1. Why does proper design often take longer than writing the program code?
2. Where does a programmer first begin determining the user’s requirements?
3. True or false: Proper top-down design forces you to put off details as long as possible.
4. How does top-down design differ from pseudocode?
5. What is the purpose of RAD?
6. True or false: You do not have to add code to any system that you design with RAD.
7. Which uses symbols: a flowchart or pseudocode?
8. True or false: You can flowchart both program logic as well as real-world procedures.
9. True or false: Your user will help you create a program’s output if you let the user work with an output prototype.
10. What is the final step of the programming process (before testing the final result)?

Answers
1. The more thorough the design, the more quickly the programming staff can write the program.
2. A programmer often begins defining the output of the proposed system.
3. True
4. Top-down design enables a program designer to incrementally generate all aspects of a program’s requirements. Pseudocode enables you to specify the logic of a program once the program’s design has been accomplished using tools such as top-down design.
5. RAD provides a way to rapidly develop systems and move quickly from the design stage to a finished product. RAD tools are not yet advanced enough to handle most programming tasks, although RAD can make designing systems easier than designing without RAD tools.
6. False. RAD requires quite a bit of programming in many instances once its work is done.
7. A flowchart uses symbols.
8. True
9. True
10. The final step of programming is writing the program code.
Input and output are the cornerstones that enable programs to interact with the outside world. In the previous hour, you learned how important the output process is to programming because through output, your program displays information. A program must get some of its data and control from the user’s input, so learning how to get the user’s responses is critical as well.

The highlights of this hour include the following:

- Displaying output in Python
- Printing multiple occurrences per line
- Separating output values
- Using variables to hold information
- Getting data in Python
- Prompting for data input
- Sending output to your printer

**Printing to the Screen with Python**

In Python, the primary method for displaying output on the screen is to use the `print()` function. You’ve already seen the `print()` function in action in the programs presented in the first two hours of the book. Almost every program you write will output some data to the screen. Your users must be able to see results and read messages from the programs that they run.

**NOTE**

In programming, a *function* is a collection of programming statements that perform a specific task. When programming, if you find yourself needing to do the same thing over and over again, you will save time by creating a function. Most programming languages include a series of predefined functions for output, input, and many mathematical operations. Some of Python’s built-in functions are covered in this book, but there are many more available. The Python functions that you learn in this book generally have comparable functions in other programming languages; once you learn one, it should be pretty easy to understand other similar functions in other languages.
The output to the screen in most programs is a combination of unchanging and changing information. Luckily, the `print()` function can handle both. The following statements show some examples:

```python
print('2 + 3 = ', 2+3)
print('Math is fun!')
```

These statements produce the following output:

```
2 + 3 = 5
Math is fun!
```

Remember that with the `print()` function in Python, you need to put what you plan to print in the parentheses. Without that, you will not get a result; instead, your code will generate an error message. You may be wondering about the information between the parentheses in the lines of code. There’s a string of characters between the two single quote marks in both, and that first single quote tells Python “print all characters you see from here on out until you get to the second, closing single quote mark.” The quotation marks are not printed; they mark the string to be printed. But what if you want to print quotation marks? Python has an easy solution. If you enclose your string to be printed in double quote marks, you can then include the single quotation mark as something to print. For example, if you changed the second line to the line:

```python
print("Isn't math fun?")
```

the output would be:

```
Isn't math fun?
```

Whether you use single or double quotation marks, understand that numbers and mathematical expressions will print as is inside the string. Python will not do any math within a string. If you write:

```python
print('2 + 3')
```

Python doesn’t print `5` (the result of `2 + 3`). Because quotation marks enclose the expression, the expression prints exactly as it appears inside the quotation marks. However, as you can see in the second half of the first statement, if you print an expression without the quotation marks, Python prints the result of the calculated expression:

```python
print(5 + 7)
```

prints `12`. 

NOTE

Don't worry too much about understanding the calculations in this hour's programs. Hour 5, “Data Processing with Numbers and Words,” explains how to form calculations in Python.

Here is the output you see if you run the program in Listing 4.1:

The area of a circle with a radius of 3 is
28.2744
The area of one-half that circle is
14.1372

Note that in Python, each time you call the print() function, it begins its output on a new line. You can also force the output to a second line by using the newline character (\n). For the newline character to work, it must be typed as the backlash character immediately followed by an n. For example, in the previous code, if you wanted the output in the first statement to span two lines but didn't want to write two print() statements, you could alter the code to:

print("The area of a circle \nwith a radius of 3 is ");

and the output of that line of code would be:

The area of a circle
with a radius of 3 is
Storing Data

As its definition implies, data processing refers to a program processing data. That data must somehow be stored in memory while a program processes it. In Python programs, as in most other languages’ programs, you must store data in variables. You can think of a variable as if it were a box inside your computer holding a data value. The value might be a number, a character, or a string of characters.

NOTE
Data is stored inside memory locations. Variables keep you from having to remember which memory locations hold your data. Instead of remembering a specific storage location (called an address), you only have to remember the name of the variables you create. The variable is like a box that holds data, and the variable name is a label for that box that lets you know what’s inside.

Your programs can have as many variables as you need. Variables have names associated with them. You don’t have to remember which internal memory location holds data; you can attach names to variables to make them easier to remember. For instance, Sales is much easier to remember than the 4,376th memory location.

You can use almost any name you want, provided that you follow these naming rules:

- Variable names must begin with an alphabetic character such as a letter.
- Variable names can be as long as you need them to be.
- Uppercase and lowercase variable names differ; MyName and MYNAME refer to two different variables.
- After the first alphabetic character, variable names can contain numbers and underscores.

CAUTION
Avoid strange variable names. Try to name variables so that their names help describe the kind of data being stored. Balanc19 is a much better variable name for an accountant’s 2019 balance value than Xly96a, although Python doesn’t care which one you use.

Here are some examples of valid and invalid variable names:

<table>
<thead>
<tr>
<th>Valid</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sales04</td>
<td>Sales-04</td>
</tr>
<tr>
<td>MyRate</td>
<td>My$Rate</td>
</tr>
<tr>
<td>ActsRecBal</td>
<td>5ActsRec</td>
</tr>
<tr>
<td>row</td>
<td>if</td>
</tr>
</tbody>
</table>
CAUTION
Don’t assign a variable the same name as a Python statement, or Python will issue an invalid variable name error message.

Variables can hold numbers or character strings. A character string usually consists of one or more characters, such as a word, a name, a sentence, or an address. Python lets you hold numbers or strings in your variables.

Assigning Values
Many Python program statements use variable names. Often, Python programs do little more than store values in variables, change variables, calculate with variables, and output variable values.

When you are ready to store a data value, you must name a variable to put it in. You must use an assignment statement to store values in your program variables. The assignment statement includes an equal sign (=). Here are two sample assignment statements:

```
sales = 956.34
salesperson = "Tina Grant"
```

TIP
If you learn another language, it may require that you use a keyword to first declare a variable, so keep that in mind.

Think of the equal sign in an assignment statement as a left-pointing arrow. Whatever is on the right side of the equal sign is sent to the left side to be stored in the variable there. Figure 4.1 shows how the assignment statement works.

![Diagram](A=23)

**FIGURE 4.1**
The assignment statement stores values in variables.
If you want to store character string data in a variable, you must enclose the string inside either single or double quotation marks. Here is how you store the phrase Python programmer in a variable named \texttt{myJob}:

\begin{verbatim}
myJob = "Python programmer" # Enclose strings in quotation marks
\end{verbatim}

After you put values in variables, they stay there for the entire run of the program or until you put something else in them. A variable can hold only one value at a time. Therefore, the two statements:

\begin{verbatim}
age = 67;
age = 27;
\end{verbatim}

result in \texttt{age} holding 27 because that was the last value stored there. The variable \texttt{age} cannot hold both values.

You can also assign values of one variable to another and perform math on the numeric variables. Here is code that stores the result of a calculation in a variable and then uses that result in another calculation:

\begin{verbatim}
pi = 3.1416;
radius = 3;
area = pi * radius * radius;
halfArea = area / 2;
\end{verbatim}

\textbf{TRY IT YOURSELF}

When you are looking to print the values stored in variables, print the variable names without quotes around them. Listing 4.2 contains code similar to Listing 4.1, but instead of printing calculated results directly, the program first stores calculations in variables and prints the variables’ values.

\textbf{LISTING 4.2 Calculating the area of a circle with variables}

\begin{verbatim}
# Filename AreaHalf2.py
# program that calculates and prints the area
# of a circle and half circle

pi = 3.14159; # mathematical value of PI
radius = 3; # radius of the circle
\end{verbatim}
Getting Keyboard Data with `input()`

So far, the programs you've created have used specific pieces of information and data coded right into the programs. Even variables have been defined with specific values, such as the radius of the circle in Listing 4.1. While this is interesting, it's ultimately limiting. To make programs more valuable, you need to get information from your user.

The `input()` function is sort of the opposite of `print()`. The input function receives values from the keyboard. You can then assign the values typed by the user to variables. In the previous section, you learned how to assign values to variables. You used the assignment statement because you knew the actual values. However, you often don't know all the data values when you write a program.

Think of a medical reception program that tracks patients as they enter the doctor's office. The programmer has no idea who will walk in next and so cannot assign patient names to variables. The patient names can be stored in variables only when the program is run.

When a program reaches a prompt call, it creates a dialog box that stays until the user types a value and clicks or taps the OK button. Here is an input:

```python
input("What is your favorite color?");
```

When program execution reaches this statement, the computer displays a dialog box or prompt with the message you type in the quotation marks. The dialog box is a signal to the user that something is being asked, and a response is desired. The more clear you make the statement you send to the prompt, the easier it will be for the user to enter the correct information.

The program in Listing 4.3 is a third attempt at the area of a circle program, but this time the user gets to enter the radius of the circle. Now that the user can enter the radii of different-sized circles, this program has far more value.
It might start to get a little dull to keep writing variations of the same program, but making just subtle changes to your code to achieve the same or slightly different results is a great way to understand new commands and techniques.

**LISTING 4.3 Using `input` to get the value of a circle's radius**

```python
# Filename AreaHalf3.py
# program that calculates and prints the area
# of a circle and half circle
pi = 3.14159 # mathematical value of PI
radius = float(input("Enter a circle's radius: ")) # get radius
# calculate the area of the whole circle
area = pi * radius * radius
print("The area of a circle with a radius of", radius, "is %.2f" % area);
print("The area of a half circle is %.2f" % (area/2));
```

If the user runs this program, the prompt statement produces the dialog box featured in Figure 4.2.

![Figure 4.2](image)

The program will not advance until the user enters a value and then presses Enter.

The statement to get the input needs to be examined a bit:

```python
radius = float(input("Enter a circle's radius: "))
```

You are using the `input()` function to get the value the user wants and will be assigning it to the variable `radius`. But when users enter information, the computer makes no assumptions that what they have entered is a string of letters or a number. So, you have to tell Python to treat the information as a number—in this case, by putting `float()` around the input statement. This tells Python to treat whatever is inside the () as a floating-point number, a concept known as **casting**. This concept will be covered in more detail in later hours, but remember from Hour 1 that computers are dumb machines that do exactly what you tell them to do, so you have to tell them this specific variable is a number.

Once the user enters a value for the radius, the program proceeds as it did before, with a few differences. First, it shows the area of an entire circle and then the area of a half circle. But the `print()` function looks a little different than it did before:

```python
print("The area of a circle with a radius of that radius is %.2f" % area);
print("The area of a half circle is %.1f" % (area/2));
```
Inputting Strings

Unlike in many programming languages, a variable in Python can hold either a number or a string. Any type of variable, numeric or string, can be entered by the user through a prompt dialog box. For example, this line waits for the user to enter a string value:

```python
fname = input("What is your first name");
```

When the user types a name in response to the question, the name is put into the `fname` variable.

CAUTION

If the user only clicks or taps OK, without entering a value in response to the prompt, Python puts a value called `null` into the variable. A null value is a zero for numeric variables or an empty string for string variables. An empty string—a string variable with nothing in it—is literally zero characters long.

TRY IT YOURSELF

Listing 4.4 is a simple program that once again takes user input and again stores the information in variables. This time, you are prompting the user for strings (two of them).

**LISTING 4.4 Using `input` to get a user's first and last names**

```python
# Filename entername.py
# program that asks the user's first and last
```
TIP

Python’s ability to combine the string asking the user to enter information and the prompt for the data itself is not a feature all programming languages share. When you use other languages (such as C), you may have to have a separate output statement telling the user what you need and an input statement to receive the information.

This program gets two strings from the user—a first name and a last name—and then combines them in two different formats in print() statements. There are other ways to combine strings, as discussed in the next lesson. The other issue is that there is no checking to ensure that the user entered the correct information. With strings, the program accepts numbers and treats them as strings. So if your user enters Helga as their first name and 11 as their last name, Python will set the full name as Helga 11.

While numbers can be treated as strings, the opposite is not true (for strings being entered as numbers). In Listing 4.3, if the user enters a series of letters for the radius, the program returns an error. When you are writing programs that take input, you often need to ensure that the user has entered the expected value. This is known as data validation, and this topic is covered in more detail in Hour 6, “Controlling Your Programs.”

TRY IT YOURSELF

Listing 4.5 shows a program that a small store might use to compute totals at the cash register. The input functions in this program are required; only at runtime will the customer purchase values be known. As you can see, getting input at runtime is vital for real-world data processing.
You can use `input()` to simulate a cash register program for a small store

```python
# Filename: Storereg.py
# A more practical use of input and output
# Asks users for specific info on sold items

print("Welcome to Fran's Place!\n\n")
print("Let's proceed to checkout!\n")
# A series of statements to find out how much of each item has been purchased

candy = int(input("How many candy bars did they buy? "))
drinks = int(input("How many energy drinks did they buy? "))
gas = int(input("How many gallons of gas did they buy? "))

# This section will take each value and multiply it by the current cost per item

candytotal = candy * 1.25
drinktotal = drinks * 2.25
gastotal = gas * 2.879

total = candytotal + drinktotal + gastotal

tax = total * .0725

# Finally print the itemized receipt

print("\n\nItem  Qnt  Total")
print("-------------------------------")
print("Candy  \", candy, "  $\%.2f" % candytotal)
print("Drinks  \", drinks, "  $\%.2f" % drinktotal)
print("Gas  \", gas, "  $\%.2f" % gastotal)
print("-------------------------------")
print("Subtotal  $ \%.2f" % total)
print("Tax  $ \%.2f" % tax)
print("Total  $ \%.2f" % (total+tax))
print("\n\nHAVE A GREAT DAY!")
```
Figure 4.3 shows the output of this program. As you can see, this type of program could be helpful for a small store. Obviously, it is unlikely that a store would only have three items, but once you learn some additional features of Python (such as dictionaries), you can quickly and easily build a more robust set of data for any need you have, personally or professionally.

You might be wondering about the \t character that appears in several of the last 10 lines that print out the receipt. This is another example of a formatting character you can use in Python. When Python encounters a \t in a string, it tabs over (as in a word processor) before continuing to print. This can be extremely useful if you are looking to line up columns when printing out output, as this program does for the quantities and totals of items purchased. The `print()` statements also perform formatting you have seen before, including using \n to jump down a line and %.2f to limit the digits to the right of the decimal points to two, which is all you should see in a financial transaction. When asking for purchase amounts, this program has lines for each item in inventory. While this works, it can be inefficient. Later on, in Hour 6, you will learn some tricks to loop through identical or similar code lines with fewer total lines. This might not seem like a big deal when you’re only dealing with 3 products, but what if you had 20 or more? In such situations, you can really improve your coding efficiency by taking advantage of loops.

```
Welcome to Fran's Place!

Let's proceed to checkout!
How many candy bars did they buy? 4
How many energy drinks did they buy? 6
How many gallons of gas did they buy? 12

Item  Qt  Total
-------  ----  ------
     Candy  4  $5.00
     Drinks 6  $13.50
      Gas  12  $34.55
    _subtotal  $53.85
     Tax     $3.85
    Total   $56.89

HAVE A GREAT DAY!
```

**FIGURE 4.3** Running the cash register program produces this output.

**NOTE**

Again, there is a lot you can do with input and output in Python, but this lesson just covers programming basics. If you want to learn more, please pick up a tutorial devoted to the language; your programs will thank you if you do!
**Summary**

Proper input and output can mean the difference between a program that your users like to use and one they hate to use. If you properly label all input that you want so that you prompt your users through the input process, the users will have no questions about the proper format for your program.

The next hour describes in detail how to use Python to program calculations using variables and the mathematical operators, as well as some handy string-manipulation tricks.

**Q&A**

**Q.** How can I ensure users enter information in the proper format for my program?

**A.** As mentioned earlier in the hour, techniques known as data validation can check to make sure the information entered is expected. If it isn’t you can either generate an error message or give the user another chance to enter the information. Data validation is covered more in later hours, but it will become an important consideration of any program that features user interaction.

**Q.** Why don’t I have to tell Python what type of variable I want to use?

**A.** Python is just that smart! Actually, for most programming languages, you need to specify the type of variable, and if you try to put a different type of data in that variable, you can get an error or unpredictable results. Python changes the variable type on-the-fly, so you can use the same variable as a string in the beginning of the program and then a number later. This is not the best idea, however. You should keep your variables focused on a specific type and a specific job.

**Workshop**

The quiz questions are provided for your further understanding.

**Quiz**

1. What is a function?
2. How would you write a `print()` statement that prints the sum of 10 and 20?
3. Declare a variable named `movie` and assign to it the last movie you saw in theaters.
4. What character is used in `print()` statements to force a new line?
5. What is a variable?
6. What function is used to get information from a program’s user?
7. What is a prompt?
8. Write a simple program that asks the user for his or her birthday in three separate prompts—one for month, one for day, and one for year—and then combine the three into a Month date, year format that you print on the screen.

9. In Python, what does the /t character do?

**Answers**

1. A function is a collection of statements that perform a specific task.

2. `print(10 + 20)`

3. (Obviously, this should vary based on your most recent cinema-viewing experience.) For me:
   ```python
   movie = "Once Upon a Time in Hollywood"
   ```

4. The newline character is \n.

5. A variable is a named storage location.

6. The `input()` function

7. A prompt describes the information that a user is to type.

8. Here is one possible solution:
   ```python
   # Answer to Chapter 4, Question 8
   bYear = input("What year were you born? ")
   bMonth = input("What month were you born? ")
   bDay = input("What day were you born? ")
   print("You were born on", bMonth, bDay, ",", bYear, ")
   ```

9. It tabs over the input.
Index

Numbers

2D/3D graphics, Java, 159

A

abs() function, 77
abstraction, C++, 322
accumulators, 123, 130–131
accuracy in programming, 98–104
AI (Artificial Intelligence), 28
AJAX (Asynchronous JavaScript and XML), 158, 247
ajaxRequest function, 254
ajaxResponse function, 254
elements of, 249–250
Frameworks, 250
JavaScript Client, 248
JSON, 249
libraries, 250, 253–259
limitations of, 250–251

quizzes, creating with libraries, 254–255
HTML files, 255–256
JavaScript files, 257–258
testing, 258–259
requests, 248
server-side scripts, 248–249
XML, 249, 256
XMLHttpRequest, 247–248
awaiting responses to, 252
creating requests, 251
interpreting responses to, 252–253
opening URL, 251–252
sending requests, 252

algorithms, 124
accumulators, 123, 130–131
counters, 123–127
defined, 123
dictionaries, 127–129
functions and, 144–147
lists, 127
nested loops, 148
sorting data, 123, 133
ascending sort order, 133
bubble sorts, 133–137
class string data, 133
descending sort order, 133
subroutines, 144–147
swapping data, 131–132
ALTER TABLE statements, 272
Amazon.com, AJAX, 249
ambiguity in programming, 28–29
Anaconda, installing, 395–398
analysis/design jobs, 369–370
anchor tags, 213–214
and operator, 179
animation, web pages, 155
API (Application Programming Interface), Java database API, 159
applets, 153, 155, 157–160
applications
C# applications, creating, 352–355
buttons, 356
controls, 357–359
declaring variables, 357–358
guidelines, 356
labels, 356
naming variables, 356–357
compiled applications, 378–379
distributing, 377
cloud computing, 379
compiled applications, 378–379
mobile applications, 380
open-source software, 380
packaged applications, 378–379
software distributions, 377–378
Java, 153, 158
packaged applications, 378–379
Visual Basic applications, creating, 335–336
adding details, 337–339
aligning controls, 339
centering forms, 339
changing/assigning properties, 337
procedures, 341–344
properties of, 340
resizing form windows, 336
subroutines, 341
Windows applications
code modules, 334, 344
form files, 334
other files, 334
arguments, Java, 188, 195
arithmetic assignment operators, 174–175
arithmetic operators, 289
arrays, 74–75
Java, 172
JavaScript, 226–227
ascending sort order, 133
ASCII table, 65, 71–73
calculate values, 65–66
nonprinting characters, 73
assembly language, 309–310
assignment operators, 286, 289, 291–292. See also combined assignment operators
assignment statements, C, 314
automated testing, 385–386
logic errors, 99–101
origin of, 97–98
syntax errors, 99–101
built-in PHP functions, 295–296
buttons, C#, 356
buying programs, 18
bytecode, 155–157

C, 309
assignment statements, 314
built-in functions, 315–320
C++ versus
   I/O differences, 323–324
   name differences, 323
clear programs, writing, 104–105
command keywords, 310
comments, 314
compilers, 311
control statements, 321–322
cryptic nature of, 309–312, 314
grouping symbols, 312
header files, 313
#include statement, 312–313
main() function, 312, 320–321
operators, 310, 321
portability of, 315
preprocessor directives, 312–313
printf() function, 315–320
scanf() function, 318–320
stdio.h files, 313
strcpy() function, 317
strings, 313–314, 317
supported data types, 313–314
variables
   declaring, 314
   floating-point variables, 313
   integer variables, 313
writing functions, 320–321
C#, 350–351. See also .NET Framework
applications, creating, 352–355
buttons, 356
clearing programs, writing, 104–105
controls, 357–359
debugging and testing, 106
declaring variables, 357–358
guidelines, 356
labels, 356
naming variables, 356–357
DLS, 350
visual nature of, 355–359
C++, 309
abstraction, 322
C versus
   I/O differences, 323–324
   name differences, 323
clear programs, writing, 104–105
control statements, 321–322
cryptic nature of, 309–312
DLS, 350
visual nature of, 355–359
C++, 309
abstraction, 322
C versus
   I/O differences, 323–324
   name differences, 323
clear programs, writing, 104–105
control statements, 321–322
cryptic nature of, 309–312
DLS, 350
visual nature of, 355–359
called methods, 195–197
camel notation, 356
capital/lowercase letters in strings, 75
CASE (Computer-Aided Software Engineering) tools, 387–388
cash register program, input example, 58–60
centering forms, Visual Basic, 339
certificates
computer-related jobs, 366–367
site certificates, 160
CGI (Common Gateway Interface), 155
color literals, 168–169
color strings, 52–53, 133
color variables, 172
chargeback, IT/data process departments, 364–365
chr() function, 73
CI (Continuous Integration), 386
CIL (Common Intermediate Language), 349
class statements, 322
objects and, 325, 327–328
scope of, 328–329
string classes, 330
classes
arguments, methods and, 195
Calculator class, 188
data members, 193
inheritance, 193
methods, 193, 195
arguments and, 195
called methods, 195–197
doubleIt() method, 197
objects (PHP), 300
overview of, 192–194
subclasses (derived classes), 193
SwingCalculator class, 190
clear programs, writing, 104–105
cloud computing
applications, distributing, 379
cloud services, 379
private cloud storage, 379
public cloud storage, 379
CLR (Common Language Runtime), 348–349
COBOL (Common Business-Oriented Language), 97
code
blocks of code, 176–177
defined, 6–7
managed/unmanaged code, 348
profilers, 119–120
writing (designing programs), 47
code modules, Windows applications, 334, 344
collections, 159
combined assignment operators, 290–292. See also assignment operators
command tags (HTML), 205–206
commands, SQL queries, 266
comments, 11
C, 314
defined, 9
JavaScript, 218
PHP 281
placement of, 10–11
pound sign (#), 9–10
reasons for, 10
companies, programming departments, 361–363
chargeback, 364–365
contract programmers, 364–365
funny money, 364
jobs, 365–366
analysis/design jobs, 369–370
consulting jobs, 374–375
data entry jobs, 367–368
degrees/certificates, 366–367
egoless programmers, 372
front end/back end developers, 370–371
mobile developers, 370–371
programmer jobs, 368–369
putting programs into production, 372–374
security, 374
systems analysts, 369–370
titles, 366
training, 388–390
UI developers, 370–371
overhead, 363
resource allocation, 363
telecommuting, 364
comparing data
elif statements, 84, 86
else statements, 83–84
if statements, 81–84
decision symbols, 83
eilf statements, 86
nesting, 86
relational operators, 84–85
comparison operators, 175–176, 292
compiled applications, 378–379
compilers, 311
complex test expressions, creating with logical operators, 292–293
computer-related jobs, 365–366
analysis/design jobs, 369–370
consulting jobs, 374–375
data entry jobs, 367–368
degrees/certificates, 366–367
egoless programmers, 372
front end/back end developers, 370–371
job security, 374
mobile developers, 370–371
production, putting programs into, 372–374
programmer jobs, 368–369
systems analysts, 369–370
titles, 366
training, 388–390
UI developers, 370–371
computers make mistakes, programming myths, 4
concatenating (merging) strings, 63–64
concatenation operators, 289–290
conditional operator, 176
constants, 294–295. See also variables
constructs, 110–111
decisions (selections), 112–113
sequences, 111–114
consulting jobs, 374–375
contract programmers, 364–365
control statements in C, 321–322
cookies in user sessions, PHP and, 304
counters, 123–127
Create a New Project screen (Visual Basic), navigating, 334–335
CSS (Cascading Style Sheets), formatting text, 210–213
CTS (Common Time System), 348–349
customizing programs, 18–19

data
comparisons
eilf statements, 84, 86
else statements, 83–84, 86
if statements, 81–86
flow, defining (designing programs), 36–38
data gathering process, 38–40
listing fields, 36–37
prototyping, 38–41
RAD, 41
top-down program design, 41–44
gathering process (designing programs), 38–40
information versus, 2
processing, 2
sorting data, 123, 133
ascending sort order, 133
bubble sorts, 133–137
character string data, 133
descending sort order, 133
storage
private cloud storage, 379
public cloud storage, 379
variables, 52–55, 57
swapping data, 131–132
validation, 58
data entry jobs, 367–368
data members, 193
data processing/IT departments, 361–363
chargeback, 364–365
contract programmers, 364–365
funny money, 364
jobs, 365–366
analysis/design jobs, 369–370
consulting jobs, 374–375
data entry jobs, 367–368
degrees/certificates, 366–367
egoless programmers, 372
front end/back end developers, 370–371
mobile developers, 370–371
programmer jobs, 368–369
putting programs into production, 372–374
security, 374
systems analysts, 369–370
titles, 366
training, 388–390
UI developers, 370–371
overhead, 363
resource allocation, 363
telecommuting, 364
data types (PHP), 288
database API (Application Programming Interface), Java, 159
databases, PHP interaction with, 304
databases (relational), 263–264
deleting, 271–272
fields, 264–265
adding to tables, 272
deleting from tables, 273
modifying in tables, 273
records
deleting, 271
inserting, 269
narrowing data results, 267–269
retrieving, 266–269
updating, 270–271
SQL queries
ALTER TABLE statements, 272–273
common commands, 266
DELETE statements, 271
DROP DATABASE statements, 271–272
DROP TABLE statements, 271–272
INSERT INTO statements, 269
SELECT statements, 266–269
UPDATE statements, 270–271
WHERE clause, 267–269
tables, 264–266
adding fields to tables, 272
ALTER TABLE statements, 272–273
deleting, 271–272
deleting fields from tables, 273
modifying fields in tables, 273
debugging tools
accuracy in programming, 98–104
breakpoints, 106
bugs
common bugs, 98
logic errors, 99–101
origin of, 97–98
runtime errors, 102
syntax errors, 99–101
call stacks, 106
clear programs, writing, 104–105
IDE, 101
testing, 101–104
variables, 106
decision statements
elif statements, 84
else statements, 83–84, 86
if statements, 81–84
decision symbols, 83
elif statements, 86
nesting, 86
relational operators, 84–85
decision symbols, 83
decisions (selections), 112–113
decrement/increment operators, 173–174
defining PHP functions, 296–298
degrees/certificates, computer-related jobs, 366–367
DELETE statements, 271
deleting
records from relational databases, 271
relational databases, 271–272
tables from relational databases, 271–272
demand for programmers, 5
deprecate, defined, 153
derived classes (subclasses), 193
descending sort order, 133
design tools, 23
design/analysis jobs, 369–370
designing programs
data flow, defining, 36–38
data gathering process, 38–40
listing fields, 36–37
prototyping, 38–41
RAD, 41
top-down program design, 41–44
logic development, 44–46
need for design, 33–34
OOD, 38
OOP 38
output, defining, 36–38
data gathering process, 38–40
listing fields, 36–37
prototyping, 38–41
RAD, 41
top-down program design, 41–44
user-programmer agreements, 34–35
writing code, 47
desk checking, 118
detailed instructions, 20–23
dictionaries, 127–129
difficulties in programming, 5
digital signatures, 160
directions, programs as, 20–21
distributing applications, 377
cloud computing, 379
compiled applications, 378–379
mobile applications, 380
open-source software, 380
packaged applications, 378–379
software distributions, 377–378

DLR (Dynamic Language Runtime), 350
<DOCTYPE html> tags, 208
dot operators, C++ objects, 326
doublet() method, 197
do.while loops, 228–230, 285
downloading, Python, 393–395
drag and drop, Java, 159
drawstring() method, 164
DROP DATABASE statements, 271–272
DROP TABLE statements, 271–272

E

EBCDIC table, 66
echo statements, 279–280
egoless programmers, 372
elif statements, 84, 86
else statements, 83–84, 86, 177–179
endl object, 324
English (structured), logic development, 44–46
events
debugging
logic errors, 99–101
runtime errors, 102
syntax errors, 99–101
typing errors, 145
escape sequences, 168–169
executable content, Java, 154–157
experts, programming myths, 3–4
extraction operator, 323–324

F

Facebook, AJAX, 249
false/true literals, 168
FCL (Framework Class Library), 349–350
fields
listing (designing programs), 36
relational databases, 264–265
adding fields to tables, 272
deleting fields from tables, 273
modifying fields in tables, 273
first programs, writing, 8, 11–13, 218–221
floating-point literals, 168
floating-point variables, 171, 313
flowcharts, logic development, 44–46
flow control, PHP, 281
fonts/text, Visual Basic applications, creating, 338
form data and PHP, 304
form files, Windows applications, 334
formatting
\n, 60
\t, 60
Java statements, 177
text
CSS, 210–213
HTML, 208–213
FORTRAN (Formula Translation), 30

frameworks
AJAX, 250
FCL, 349–350

free-form programming
languages, 10

front end/back end developers, 370–371

functions
abs() function, 77
ajaxRequest function, 254
ajaxResponse function, 254
algorithms and, 144–147
chr() function, 73
creating, 115–118
defined, 49, 74
flow control functions, 281–283
input() function, 55–57
cash register program,
input example, 57–60
data validation, 58
strings, 57–60
main() function, 312,
320–321
math.atan() function, 78
math.exp() function, 78–79
math.floor() function, 76–77
math.log() function, 79
member functions (C++), 326–327
numeric functions, 76–79
operator overloading functions, 329–330
overview of, 74

PHP functions, 295
built-in PHP functions, 295–296
defining, 296–298
variable scope in functions, 298–300
print() function, 49–51
print() statements, 145
printf() function, 315–320
prototypes, 327
range() function, for loops, 91–93
round() function, 77
scanf() function, 318–320
strcpy() function, 317
string functions, 75–76
subroutines, 144–147
typing errors, preventing, 145

funny money, IT/data processing departments, 364

G

game-development and Java, 160
garbage collection, .Net framework, 349
global variables, 169–170, 287
Gmail, AJAX, 249
goto statements, 111
graphics/multimedia images in web pages, 205, 213
grouping symbols (C), 312
GUI (Graphical User Interface), Java, 190–191
guidelines, C#, 356

H

<h> tags, 209–210
hardware, industry standards, 389
<head> tags, 208
header files, 313

HTML (HyperText Markup Language), 30, 201–202
<body> tags, 208
<br> tags, 208–209
<!DOCTYPE html> tags, 208
<h> tags, 209–210
<head> tags, 208
<html> tags, 208
<img> tags, 213
<title> tags, 208
AJAX quizzes, creating, 255–256
anchor tags, 213–214
attribute tags, 204
command tags, 205–206
example of, 203–204,
206–207
formatting text, 208–213
graphics/multimedia images in web pages, 213
hyperlinks, 213–214
Java, 154
PHP and, 280–281
tag references/commands, 205
W3C standardization, 202
hyperlinks, 213–214
IDE (Integrated Development Environment), 384–385, 393–405
debugging tools, 101
Java, 165
Python IDE, 9
if statements, 81–84, 177–179, 281–282, 284
decision symbols, 83
JavaScript, 227
nesting, 86
relational operators, 84–85
if-else statements, 177–179
<img> tags, 213
import commands, Java, 162–163
#include statement, 312–313
incremental variables, 124
increment/decrement operators, 173–174
industry standards, 389
infinite loops, 114, 285
information systems/services, 361–363
chargeback, 364–365
contract programmers, 364–365
funny money, 364
jobs, 365–366
analysis/design jobs, 369–370
consulting jobs, 374–375
data entry jobs, 367–368
degrees/certificates, 366–367
egoless programmers, 372
front end/back end developers, 370–371
mobile developers, 370–371
programmer jobs, 368–369
putting programs into production, 372–374
security, 374
systems analysts, 369–370
titles, 366
training, 388–390
UI developers, 370–371
overhead, 363
resource allocation, 363
telecommuting, 364
information versus data, 2
inheritance
C++, 322, 330
Java classes, 193
init() method, 163
input() function, 55–57
cash register program, input example, 57–60
data validation, 58
strings, 57–60
input verification, if statements, 178
INSERT INTO statements, 269
inserting records into relational databases, 269
insertion operator, 323–324
installing
Anaconda, 395–398
Python, 395–405
instructions
detailed instructions, 20–23
saved instructions, programs as, 24–26
statements as, 9
instructor terminators, 286
integer literals, 168
integer variables, 170–171, 313
interactivity, adding to photos, 237–241
interpreting responses to requests, 252–253
I/O (Input/Output), C versus C++, 323–324
italicized text, 209
IT/data processing departments, 361–363
chargeback, 364–365
contract programmers, 364–365
funny money, 364
jobs, 365–366
analysis/design jobs, 369–370
consulting jobs, 374–375
data entry jobs, 367–368
degrees/certificates, 366–367
egoless programmers, 372
front end/back end developers, 370–371
mobile developers, 370–371
programmer jobs, 368–369
putting programs into production, 372–374
security, 374
systems analysts, 369–370
titles, 366
training, 388–390
UI developers, 370–371
overhead, 363
resource allocation, 363
telecommuting, 364
iteration/repetition (looping), 113–114, 284

J
Java, 30, 151–153, 164
2D/3D graphics, 159
applets, 153, 155, 157–160
applications, 153, 158
apps, 153
arguments, 188, 195
arrays, 172
BigDecimal class, 179
blocks of code, 176–177
bytecode, 155–157
C++ and, 160–161, 167, 169
classes
Calculator class, 188
data members, 193
inheritance, 193
methods, 193
overview of, 192–194
subclasses (derived classes), 193
SwingCalculator class, 190
collections, 159
data members, 193
database API, 159
drag and drop, 159
drawstring() method, 164
drawstring() method, 164
escape sequences, 168–169
evaluate the, 168–169
example of, 161–162
executable content, 154–155
game-development, 160
GUI, 190–191
HTML, 154
IDE, 165
import commands, 162–163
inheritance, 193
init() method, 163
interface, 158–159
JavaFX library, 191
JavaScript and, 158, 217
JDBC, 159
JVM, 156
literals, 167
Boolean literals, 168
character literals, 168–169
floating-point literals, 168
integer literals, 168
string literals, 169
true/false literals, 168
loops
for loops, 180–182
while loops, 179–180
methods, 193, 195
arguments and, 195
called methods, 195–197
doubleIt() method, 197
NetBeans, 165, 185–190
network support, 159
objects, 193–194
OOP 160–161, 191–192, 195
operators
arithmetic assignment operators, 174–175
comparison operators, 175–176
conditional operator, 176
increment/decrement operators, 173–174
and operator, 179
primary math operators, 173
packages, 162
paint() method, 163–164
public statements, 163
resize() method, 163–164
security, 159–160
Servlets, 153
setColor() method, 164
sound, 159
standalone Java applications, 158
statements
else statements, 177–179
formatting, 177
if statements, 177–179
if-else statements, 177–179
success of, 153
Swing object library, 191
timers, 159
usage summary, 157–158
variables
Boolean variables, 171
character variables, 172
floating-point variables, 171
global variables, 169–170
integer variables, 170–171
local variables, 169–170
object variables, 194
string variables, 172
updating, 175
VM, 156
JavaScript, 30, 218
advantages of, 217
AJAX, 158, 247
ajaxRequest function, 254
ajaxResponse function, 254
examples of, 249–250
frameworks, 250
JavaScript Client, 248
JSON, 249
libraries, 250, 253–259
limitations of, 250–251
quizzes, creating with libraries, 254–259
requests, 248
server-side scripts, 248–249
XML, 249
XMLHttpRequest, 247–248, 251–253
arrays, 226–227
comments, 218
do.while loops, 228–230
first programs, writing, 218–221
if statements, 227
Java and, 158, 217
for loops, 228
mouse events, 224–226, 239–241
news tickers (repeating), adding to websites, 241–244
photos
adding interactivity, 237–241
rotating on page, 233–236
printing to screen, 221
prompt method, 222–223
strings, 223
variables, 222
while loops, 228–230
JDBC (Java Database Connectivity), 159
jobs, computer-related, 365–366
analysis/design jobs, 369–370
consulting jobs, 374–375
data entry jobs, 367–368
degrees/certificates, 366–367
egoless programmers, 372
front end/back end developers, 370–371
job security, 374
mobile developers, 370–371
production, putting programs into, 372–374
programmer jobs, 368–369
systems analysts, 369–370
titles, 366
training, 388–390
UI developers, 370–371
jQuery, 250
JSON (JavaScript Object Notation), AJAX and, 249
Jupyter Notebook, naming programs, 11–12
JVM (Java Virtual Machines), 156

labels
C#, 356
Visual Basic forms, 338
LAN (Local Area Networks), 370–371
languages, 28–30
binary, 27
FORTRAN, 30
HTML, 30
Java, 30
JavaScript, 30
list of, 29
machine languages, 8
defined, 7
example of, 7–8
PHP, 30
programming languages,
free-form programming languages, 10
leading edge technology, industry standards, 389
libraries (AJAX), 250
creating, 253–254
quizzes, creating with libraries, 254–259
licenses (software), 18
listing fields (designing programs), 36–37
lists, 127, 137–138
binary searches, 141–143
sequential searches, 138–141
literals, 167
Boolean literals, 168
character literals, 168–169
floating-point literals, 168
integer literals, 168
string literals, 169
true/false literals, 168
local variables, 169–170
logic development (designing programs), 44–46
logic errors, 99–101
logical operators, 292–293
loops, 87, 113–114, 127. See also statements
do.while loops, 228–230, 285
for loops, 87–91, 180–182, 285
controlling, 91–93
JavaScript, 228
range() function, 91–93
infinite loops, 114, 285
iteration/repetition, 284
nested loops, 148, 286
PHP, 284–286
while loops, 93–94, 179–180, 228–230, 284–285
lowercase/capital letters in strings, 75

M

machine languages, 8. See also Python
defined, 7
example of, 7–8
main() function, 312, 320–321
MAMP (Macintosh, Apache, MySQL, Perl/Python/PHP), 278
managed/unmanaged code, .NET framework, 348
math
abs() function, 77
advanced math functions, 78–79
binary arithmetic, 69–72
math.atan() function, 78
math.exp() function, 78–79
math.floor() function, 76–77
math.log() function, 79
negate numbers, 70
operators, 67–69
two’s complement, 70
member functions (C++), 326–327
memory
extra memory, benefits of, 26
layout (typical), 26
OS and, 25–26
program-to-output process, 25
RAM, 24
MenuStrip control (Visual Basic), 342–343
merging (concatenating) strings, 63–64
messages, C++, 322–323
methods
called methods, 195–197
doubleIt() method, 197
Java, 193, 195
objects (PHP), 300, 302–303
MIS (Management Information Systems), 361–363, 386
chargeback, 364–365
contract programmers, 364–365
funny money, 364
jobs, 365–366
analysis/design jobs, 369–370
consulting jobs, 374–375
data entry jobs, 367–368
degrees/certificates, 366–367
egoless programmers, 372
front end/back end developers, 370–371
mobile developers, 370–371
programmer jobs, 368–369
putting programs into production, 372–374
security, 374
systems analysts, 369–370
titles, 366
training, 388–390
UI developers, 370–371
overhead, 363
resource allocation, 363
telecommuting, 364
mistakes, programming myths, 4
mobile applications, distributing, 380
mobile developers, 370–371
Morse code, 66
mouse events, JavaScript, 224–226, 239–241
multimedia images in web pages, 205, 213
multi-platform executable content, 155–157
myths about programming, 3
difficulties in programming, 5
experts, 3–4
mistakes, 4

N
naming
camel notation, 356
programs, 11–12
C, 323, 356–357
C++, 323
variables, 52–53
C#, 356–357
narrowing query data results, relational databases, 267–269
need for
design, 33–34
programmers, 20
programs, 17–20
negate numbers, 70
nested loops, 148, 286
nesting
elif statements, 86
else statements, 86
if statements, 86
NetBeans, 165, 185–190

.NET Core, 348
CIL, 349
CLR, 348–349
CTS, 348–349
FCL, 349–350
garbage collection, 349
parallel computing, 350
.NET Framework, 347. See also C#
CIL, 349
CLR, 348–349
CTS, 348–349
FCL, 349–350
garbage collection, 349
parallel computing, 350
purpose of, 347–348
networks
Java support, 159
LAN, 370–371
WAN, 370–371
newline character and C++, 324
news tickers (repeating), adding to websites, 241–244
null values, 57
number variable, 124
number-guessing game, counter variables, 125–127
numeric functions, 76–79

O
objects
C, 322–323
C++, 323
behavior, adding to objects, 326–328
class, 325
class statements, 325, 327–328
declaring, 324–325
dot operators, 326
endl object, 324
member access, 326
variables, 325–326
Java, 193–194
PHP, 300
classes, 300
creating, 300–301
methods, 300, 302–303
properties of, 302
obtaining programs, advantages/disadvantages, 18
OOD (Object-Oriented Design), 38
OOP (Object-Oriented Programming), 26, 38
opening, URL with XMLHttpRequest, 251–252
open-source software, distributing, 380
operators, 288–289
and operator, 179
arithmetic assignment operators, 174–175
arithmetic operators, 289
assignment operators, 286, 289, 291–292
C operators, 321
combined assignment operators, 290–292
comparison operators, 175–176, 292
concatenation operators, 289–290
declaration operator, 176
decrement/increment operators, 173–174
operators

extraction operator, 323–324
increment/decrement operators, 173–174
insertion operator, 323–324
logical operators, 292–293
math operators, 67–69
operator overloading functions, 329–330
post-decrement operators, 291–292
post-increment operators, 291–292
precedence, 67, 293–294
primary math operators, 173
relational operators, 84–85
OS (Operating Systems), memory, 25–26
other files, Windows applications, 334
output, 24
defining (designing programs), 36–38
data gathering process, 36–40
listing fields, 36–37
prototyping, 38–41
RAD, 41
top-down program design, 41–44
print() function, 49–51
program-to-output process, 25
variables, 52
character strings, 53
naming, 52–53
null values, 57
value assignments, 53–55
overhead, IT/data processing departments, 363
ownership of programs, 6

P
packaged applications, 378–379
packages, defined, 162
paint() method, 163–164
parallel computing, .NET Framework, 350
parallel testing, 119
people-years, writing programs, 19–20
photos
interactivity, adding to photos, 237–241
rotating on page, 233–236
Visual Basic applications, creating, 339
PHP (PHP: Hypertext Preprocessor), 30
arithmetic operators, 289
assignment operators, 286, 289
break statements, 283
built-in PHP functions, 295–296
combined assignment operators, 290
comments, 281
common uses of, 304–305
comparison operators, 292
complex test expressions, creating with logical operators, 292–293
concatenation operators, 289–290
constants, 294–295.
See also variables
data types, 288
databases, interacting with, 304
development of, 277
echo statements, 279–280
flow control, 281–283
form data, 304
functions
defined, 295
defining *separate entry, 296–298
variable scope in functions, 298–300
HTML and, 280–281
if statements, 281–282, 284
instructor terminators, 286
logical operators, 292–293
loops, 284–286
do.while loops, 285
for loops, 285–286
nested loops, 286
objects, 300
classes, 300
creating, 300–301
methods, 300, 302–303
properties of, 302
operators
defined, 288–289
precedence, 293–294
post-decrement operators, 291–292
post-increment operators, 291–292
ternary operators, 283
print() statements, 279–280
requirements, 278–279
switch statements, 282–283
user sessions, cookies in, 304
variables, 286–287. See also constants
global variables, 287
incrementing/decrementing automatically, 291–292
scope in functions, 298–300
superglobal variables, 287
while loops, 284–285
polymorphism, C++, 323, 330
post-decrement operators, 291–292
post-increment operators, 291–292
pound sign (#), comments, 9–10
precedence (operator), 293–294
pre-existing programs, 5
preprocessor directives, 312–313
primary math operators, 173
print() function, 49–51
print() statements, 145, 279–280
printf() function, 315–320
printing to screen, JavaScript, 221
private cloud storage, 379
procedures, Visual Basic applications, 341–344
process of programming, 8
production, putting programs into, 372–374
profilers, 119–120, 383–384
programmer jobs, 368–369
programming languages (free-form), 10
programs/programming, 26, 383
accuracy in programming, 98–104
ambiguity in programming, 28–29
art or science, 26–27
buying programs, 18
CASE tools, 387–388
CI, 386
code, defined, 6–7
comments, 11
defined, 9
placement of, 10–11
pound sign (#), 9–10
reasons for, 10
common myths about programming, 3
difficulties in programming, 5
experts, 3–4
mistakes, 4
customizing programs, 18–19
defined, 2
demand for programmers, 5
design tools, 23
designing programs
defining data flow, 36–44
defining output, 36–44
logic development, 44–46
need for design, 33–34
user-programmer agreements, 34–35
detailed code, 47
directions, programs as, 20–21
first programs, writing, 8, 11–13
giving computers programs, 6
IDE, 384–385
languages, 29–30
machine languages, defined, 7
memory
benefits of extra memory, 26
layout (typical), 26
OS and, 25–26
program-to-output process, 25
MIS, 386
naming programs, 11–12
need for programmers, 20
need for programs, 17–20
OOD, 38
OOP, 38
output
defined, 24
program-to-output process, 25
ownership of programs, 6
pre-existing programs, 5
process of programming, 8
production, putting programs into, 372–374
profilers, 383–384
program-to-output process, 25
resource editors, 384
role of, 2–3
saved instructions, programs as, 24–26
science or art, 26–27
statements, defined, 9
structured programming, 26, 109–110
beta testing, 118–119
constructs, 110–114
decisions (selections), 112–113
desk checking, 118
functions, 115–118
looping (repetition/iteration), 113–114
parallel testing, 119
profilers, 119–120
roots of, 110
sequences, 111–112
testing, 118–119
structured walkthroughs, 371–372
switches, 27
syntax, defined, 6–7
testing, 101–104, 385–386
UML: data modeling, 388
user-programmer agreements, 34–35
value of programs, 6
version control, 374
writing programs, 18–20
projects (Visual Basic), defined, 335
prompt method, 222–223
Prototype, 250
prototypes, functions, 327
prototyping (designing programs), 38–41
pseudocode (structured English)
logic development, 44–46
structured programming decisions (selections), 112–113
looping (repetition/iteration), 113–114
sequences, 112
public cloud storage, 379
public/private keys, 160
public statements, Java, 163
Python
Anaconda, installing, 395–398
arrays, 74–75
caller statements, 52–53
debugging tools
breakpoints, 106
bugs, common, 98
bugs, origin of, 97–98
call stacks, 106
IDE, 101
logic errors, 99–101
runtime errors, 102
syntax errors, 99–101
testing, 101–104
watch variables, 106
writing clear programs, 104–105
detailed instructions, 22–23
downloading, 393–395
elf statements, 84
else statements, 83–84, 86
first programs, writing, 11–13
formatting
\n, 60
\t, 60
functions
abs() function, 77
chr() function, 73
creating, 115–118
declared, 74
input() function, 55–60
math.atan() function, 78
math.exp() function, 78–79
math.floor() function, 76–77
math.log10() function, 79
numeric functions, 76–79
overview of, 74
print() function, 49–51
range() function, 91–93
round() function, 77
string functions, 75–76
IDE, 9, 393–405
if statements, 81–84
decision symbols, 83
elif statements, 86
nesting, 86
relational operators, 84–85
installing, 395–405
loops, 87
for loops, 87–93
while loops, 93–94
math operators, 67–69
naming programs, 11–12
strings
ASCII table, 65–67
capital/lowercase letters, 75
functions, 75–76
inputting, 57–60
merging (concatenating) strings, 63–64
replacing parts of, 75–76
Unicode characters, 72–73
variables, 52
character strings, 53
debugging, 106
naming, 52–53
null values, 57
value assignments, 53–55
watch variables, 106
whitespace, 9–10
wrapping text, 9

Q

QA (Quality Assurance) testing, 385–386
queries (SQL)
ALTER TABLE statements, 272–273
common commands, 266
DELETE statements, 271
DROP DATABASE statements, 271–272
DROP TABLE statements, 271–272
INSERT INTO statements, 269
SELECT statements, 270–271
WHERE clause, 267–269
relational databases, 263–264
deleting, 271–272
fields, 264–265
adding to tables, 272
deleting from tables, 273
modifying in tables, 273
records
deleting, 271
inserting, 269
narrowing data results, 267–269
retrieving, 266–269
updating, 270–271
SQL queries
ALTER TABLE statements, 272–273
common commands, 266
DELETE statements, 271
DROP DATABASE statements, 271–272
DROP TABLE statements, 271–272
INSERT INTO statements, 269
SELECT statements, 266–269
WHERE clause, 267–269
requests
creating, 251
repeating news tickers, adding to websites, 241–244
replacement/iteration (looping), 113–114, 284
requests
creating, 251
responses to requests, awaiting, 252
sending, 252
XMLHttpRequest, interpreting, 252–253
resize() method, 163–164
R

RAD (Rapid Application Development), 41
RAM (Random Access Memory), 24
range() function, for loops, 91–93
records (relational databases)
deleting, 271
inserting, 269
narrowing data results, 267–269
retrieving, 266–269
updating, 270–271
requests
creating, 251
responses to requests, awaiting, 252
sending, 252
XMLHttpRequest, interpreting, 252–253
resize() method, 163–164
resource allocation, IT/data processing departments, 363
resource editors, 384
responses to requests
    awaiting, 252
    interpreting, 252–253
retrieving records from relational databases, 266–269
reusability, C++, 323
rotating photos on page, 233–236
round() function, 77
runtime errors, 102

S
saved instructions, programs as, 24–26
scanf() function, 318–320
screens, printing to, 221
scripting (AJAX)
    remote scripting, 247–248
    server-side scripts, 248–249
searching lists, 137–138
    binary searches, 141–143
    sequential searches, 138–141
security
    applets, 159–160
    digital signatures, 160
    Java, 159–160
    job security, 374
    public/private keys, 160
    site certificates, 160
SELECT statements, 266–269
selections (decisions), 112–113
sending requests, 252
sequences, 111–114
sequential searches, 138–141
server-side scripts (AJAX), 248–249
setters, 153
setColor() method, 164
signatures (digital), 160
site certificates, 160
software
    CASE tools, 387–388
    distributing
        issues with distributions, 377–378
        open-source software, 380
    industry standards, 389
    licenses, 18
    version control, 374, 380–381
sorting data, 123, 133
    ascending sort order, 133
    bubble sorts, 133–137
    character string data, 133
    descending sort order, 133
sound, Java, 159
source code
    comments, 11
        defined, 9
        placement of, 10–11
        pound sign (#), 9–10
        reasons for, 10
        defined, 7
    first programs, writing, 8, 11–13
    machine languages, 8
        defined, 7
        example of, 7–8
naming programs, 11–12
process of programming, 8
statements, defined, 9
whitespace, 9–10
spacing text. See whitespace
spaghetti code, 111
SQL (Structured Query Language)
    queries
        ALTER TABLE statements, 272–273
        common commands, 266
        DELETE statements, 271
        DROP DATABASE statements, 271–272
        DROP TABLE statements, 271–272
        INSERT INTO statements, 269
        SELECT statements, 266–269
        UPDATE statements, 270–271
        WHERE clause, 267–269
    relational databases
        adding fields to tables, 272
        deleting fields from tables, 273
        modifying fields in tables, 273
    standalone Java applications, 158
statements. See also loops
        ALTER TABLE statements, 272–273
        assignment statements, 314
        break statements, 283
        call statements, 344
class statements, 322
objects and, 
325, 327–328
scope of, 328–329
string classes, 330
control statements in C, 
321–322
defined, 9
DELETE statements, 271
DROP DATABASE statements, 
271–272
DROP TABLE statements, 
271–272
echo statements, 279–280
else statements, 177–179
formatting, 177
if statements, 177–179, 227, 
281–282, 284
if-else statements, 177–179
#include statement, 312–313
INSERT INTO statements, 269
lists, 127
binary searches, 141–143
searching, 137–143
sequential searches, 
138–141
print() statements, 145, 
279–280
public statements, 163
SELECT statements, 266–269
switch statements, 282–283
UPDATE statements, 270–271
WHERE clause, 267–269
stdio.h files, 313
storing data
private cloud storage, 379
public cloud storage, 379
variables, 52
character strings, 53
listing, 52–53
null values, 57
value assignments, 53–55
strcpy() function, 317
strings
ASCII table, 65
character values, 65–66
C and, 313–314, 317
capital/lowercase letters, 75
classes, 330
functions, 75–76
inputting, 57–60
JavaScript, 223
literals, 169
merging (concatenating) strings, 63–64
replacing parts of, 75–76
variables, 172
structured English (pseudocode),
logic development, 44–46
structured programming, 26, 
109–110
constructs, 110–111
decisions (selections), 
112–113
looping (repetition/iteration), 113–114
sequences, 111–112
functions, creating, 115–118
roots of, 110
testing, 118
beta testing, 118–119
desk checking, 118
parallel testing, 119
profilers, 119–120
structured walkthroughs, 371–372
subclasses (derived classes), 193
subroutines, 144–147, 341
superglobal variables, 287
swapping data, 131–132
Swing object library, 191
SwingCalculator class, 190
switch statements, 282–283
switches, 27
syntax
defined, 6–7
errors, 99–101
systems analysts, 369–370
tables (relational databases), 
264–266
ALTER TABLE statements, 
272–273
deleting, 271–272
fields
adding to tables, 272
deleting from tables, 273
modifying in tables, 273
tag references/commands 
(HTML), 205
telecommuting, IT/data processing 
departments, 364
ternary operators, 283
test expressions (complex),
creating with logical operators, 
292–293
testing
automated testing, 385–386
beta testing, 118–119
desk checking, 118
parallel testing, 119
profilers, 119–120
programs/programming, 101–104
QA testing, 385–386
structured programming, 118–119
test-driven development, 385–386

U
UI (User Interface) developers, 370–371
UML (Unified Modeling Language): data modeling, 388
underlined text, 209
Unicode characters, 72–73
unmanaged/managed code, .NET framework, 348
UPDATE statements, 270–271
upgrading
Java variables, 175
records in relational databases, 270–271
URL (Uniform Resource Layers), opening with XMLHttpRequest, 251–252
user sessions, cookies in, 304
user-programmer agreements, 34–35

V
validating data, 58
value assignments, variables, 53–55
value of programs, 6
variables, 52. See also constants
accumulators, 123, 130–131
character strings, 52–53
counters, 123–127
C# variables,
declaring, 357–358
debugging, 106
decrementing variables, 291–292
floating-point variables, 313
global variables, 287
incremental variables, 124, 291–292
integer variables, 313
Java variables
Boolean variables, 171
character variables, 172
floating-point variables, 171
global variables, 169–170
integer variables, 170–171
local variables, 169–170
string variables, 172
updating, 175
JavaScript, 222
lists, 127
binary searches, 141–143
searching, 137–143
sequential searches, 138–141
naming, 52–53, 356–357
null values, 57
number variable, 124
object variables, 194
PHP, 286–287, 298–300
superglobal variables, 287
swapping data, 131–132
value assignments, 53–55
watch variables, 106
validating input, if statements, 178
version control, 374, 380–381
Visual Basic, 333, 344–345
applications, creating, 335–336

boldfaced text, 209
formatting
CSS, 210–213
HTML, 208–213
italicized text, 209
underlined text, 209
Visual Basic applications, creating, 338
whitespace, 9–10
wrapping text, 9
text editors, writing first programs, 11–13
tickers (repeating news), adding to websites, 241–244
timers, Java, 159
<title> tags, 208
titles, computer-related jobs, 366
top-down program design, 41–44
training, 388–390
twips, Visual Basic windows, 337
two’s complement, 70
type safety, 350
typing errors, preventing, 145
adding details, 337–339
aligning controls, 339
centering forms, 339
changing/assigning properties, 337
labels, 338
photos, 339
procedures, 341–344
properties of, 340
resizing form windows, 336
subroutines, 341
text/fonts, 338
call statements, 344
Create a New Project screen, navigating, 334–335
DLR, 350
labels in forms, 338
language behind, procedures, 344
MenuStrip control, 342–343
pixels, 337–338, 340
projects, defined, 335
twips, 337
Visual Studio
.NET Core, 348
CIL, 349
CLR, 348–349
CTS, 348–349
FCL, 349–350
garbage collection, 349
parallel computing, 350
versions of, 335
Visual Basic. See separate entry
VM (Virtual Machines), JVM, 156

W

W3C (World Wide Web Consortium), HTML standardization, 202
walkthroughs (structured), 371–372
WAMP (Windows, Apache, MySQL, PHP), 278
WAN (Wide Area Networks), 370–371
watch variables, 106
web pages, See also HTML animation, 155
CGI, 155
displaying, 205, 207
formatting, text, 208–213
graphics/multimedia images, 205, 213
photos
adding interactivity, 237–241
rotating on page, 233–236
websites, repeating news tickers, 241–244
WHERE clause, SQL SELECT statements, 267–269
while loops, 93–94, 179–180, 228–230, 284–285
whitespace, 9–10
Windows applications
code modules, 334, 344
form files, 334
other files, 334

Windows Form Application, 335
word processors, 9, 25
wrapping text, 9
writing
code (designing programs), 47
functions in C, 320–321
programs, 18–19
clarity in, 104–105
JavaScript, 218–221
people-years, 19–20

X - Y - Z

XAMPP (Cross-Platform, Apache, MariaDB, PHP, Perl), 278
XML (Extensible Markup Language), 249
AJAX, 158, 247
ajaxRequest function, 254
ajaxResponse function, 254
examples of, 249–250
frameworks, 250
JavaScript Client, 248
JSON, 249
libraries, 250, 253–259
limitations of, 250–251
quizzes, creating with libraries, 254–259
requests, 248
server-side scripts, 248–249
XMLHttpRequest, 247–248,
251
awaiting responses to requests, 252
creating requests, 251
interpreting responses to requests, 252–253
sending requests, 252
URL, 251–252