Greg Perry
Dean Miller

FOURTH EDITION

Learn computer programming in just 24 one-hour lessons

Sams Teach Yourself

Beginning Programming

in 24 Hours

Pearson

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Greg Perry
Dean Miller

Sams Teach Yourself Beginning Programming in 24 Hours

Fourth Edition

SAMS

Sams Teach Yourself Beginning Programming in 24 Hours, Fourth Edition

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearson.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Library of Congress Catalog Card Number: 2019951973

ISBN-13: 978-0-13-583670-5

ISBN-10: 0-13-583670-0

ScoutAutomatedPrintCode

Editor-in-Chief
Mark L. Taub

Acquisitions Editor
Kim Spenceley

Managing Editor
Sandra Schroeder

Development Editor
Chris Zahn

Senior Project Editor
Lori Lyons

Technical Editors
John Fonte
Michael Garcia

Production Manager
Aswini Kumar/codeMantra

Indexer
Ken Johnson

Proofreader
Abigail Manheim

Cover Designer
Chuti Prasertsith

Compositor
codeMantra

Contents at a Glance

Table of Contents

Part V: Other Programming Languages

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

Introduction

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.

HOUR 3
Designing a Program

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

Field	Type	Description
Contacts	Scrolling list	Displays the list of contacts
Name	Text field	Holds contact's name
Address	Text field	Holds contact's address
City	Text field	Holds contact's city
State	Text field	Holds contact's state
Zip	Text field	Holds contact's zip code
Home Phone #	Text field	Holds contact's phone number
Cell Phone #	Text field	Holds contact's mobile number
Email	Text field	Holds contact's email address
Stage	Fixed, scrolling list	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
Notes	Text field	Miscellaneous notes about the contact, such as whether the contact has bought from the company before

Field	Type	Description
Filter Contacts	Fixed, scrolling list	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
Edit	Function	Enables the user to modify an existing contact
Add	Function	Enables the user to add a new contact
Delete	Function	Enables the user to delete an existing contact

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.

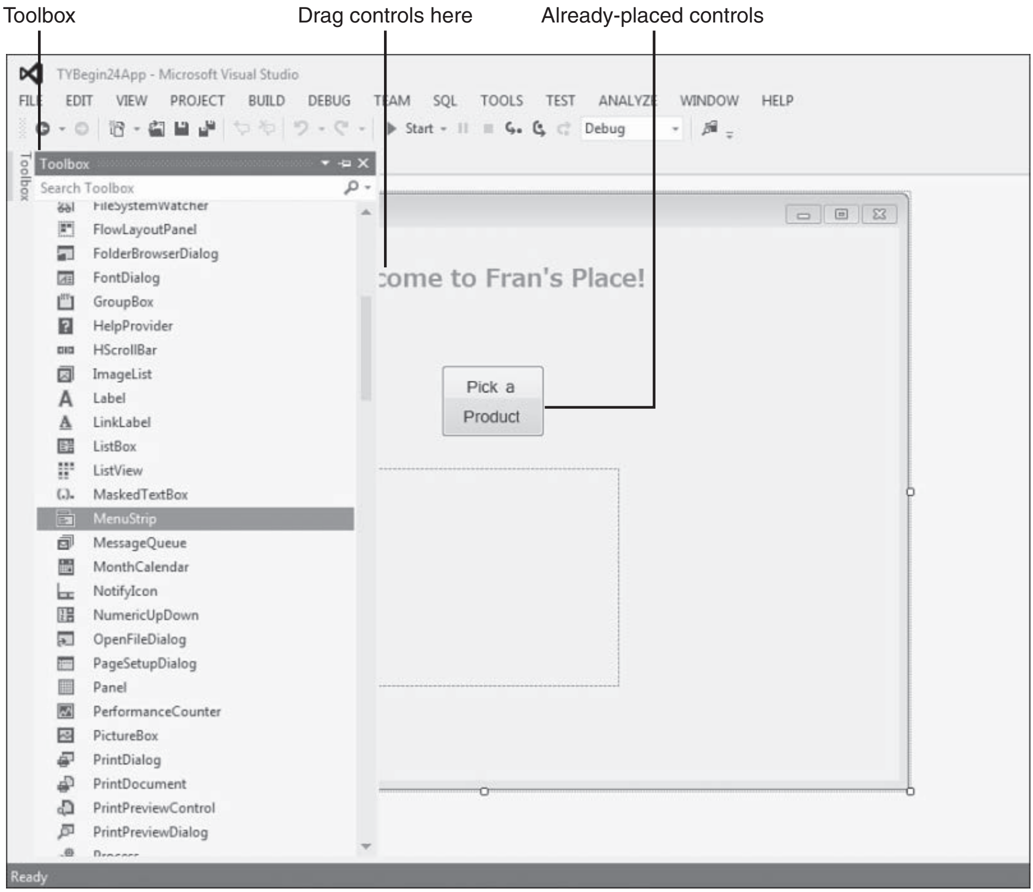

FIGURE 3.1
Program development systems such as Visual Basic provide tools that you can use to create output definitions visually.

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.

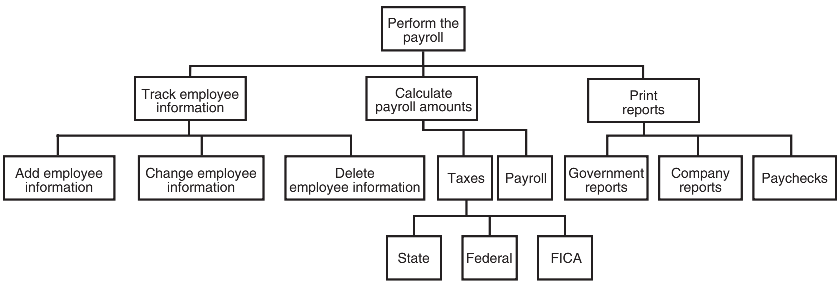

FIGURE 3.2
The first page of the payroll program's top-down design would include the highest level of details.

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.

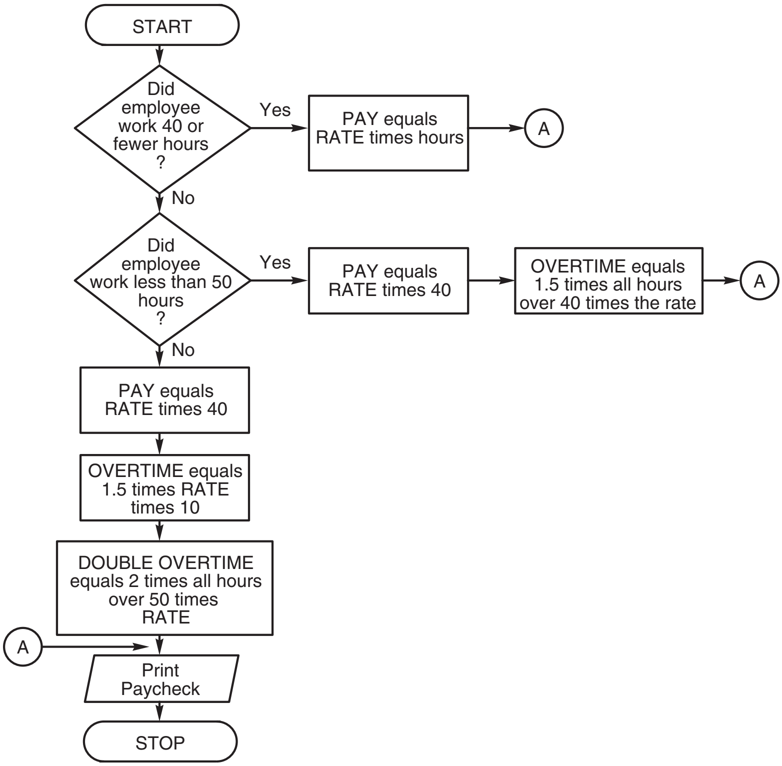

FIGURE 3.3
The flowchart depicts the payroll program's logic graphically.

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 indention 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.

HOUR 4

Getting Input and Displaying Output

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:

```
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:

```
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:

```
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:

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

prints 12.

TRY IT YOURSELF ▼

Consider the program in Listing 4.1. It prints the radius of a circle, as well as the area of the entire circle and half of the circle.

LISTING 4.1 Printing results of calculations

```
# Filename: AreaHalf.py
# Program that calculates and prints the area
# of a circle and half circle


print("The area of a circle with a radius of 3 is ");
print(3.1416 * 3 * 3);
print("The area of one-half that circle is ");
print((3.1416 * 3 * 3) / 2);
```

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 `X1y96a`, although Python doesn't care which one you use.

Here are some examples of valid and invalid variable names:

Valid	**Invalid**
`Sales04`	`Sales-04`
`MyRate`	`My$Rate`
`ActsRecBal`	`5ActsRec`
`row`	`if`

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.

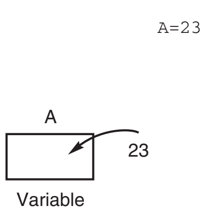

FIGURE 4.1
The assignment statement stores values in variables.

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 `myJob`:

```
myJob = "Python programmer" # Enclose strings in quotation marks
```

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:

```
age = 67;
age = 27;
```

result in `age` holding `27` because that was the last value stored there. The variable `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:

```
pi = 3.1416;
radius = 3;
area = pi * radius * radius;
halfArea = area / 2;
```

▼ 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.

LISTING 4.2 Calculating the area of a circle with variables

```
# 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
```

```
# calculate the area of the whole circle
area = pi * radius * radius;

print("The area of a circle with a radius of 3 is ", area);

print("The area of a half circle is ", area/2);
```

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:

```
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.

TRY IT YOURSELF

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.

NOTE

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

```
# 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.

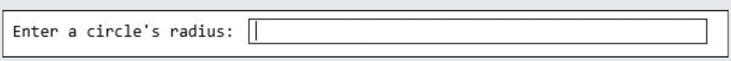

FIGURE 4.2
The program will not advance until the user enters a value and then presses Enter.

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

```
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:

```
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));
```

Now the output is a little different. Rather than just ending the two strings with `is` and then printing the number, you have `…%.2f` at the end of the first string and `%.1f` at the end of the second one. These are specific formatting instructions for Python. In the first case, you are telling Python to take the value after the second `%`—the one outside the string (the area)—and put it inside the string. However, the `.2` is telling Python to include only two places after the decimal point. So if you entered `2` as the radius, the area output would be `12.57`. Without the `.2` in the `%.2f` (that is, if your string ended with `%f`) the area output would be `12.566360`. It's a more exact answer, but it's also more awkward. Using this kind of formatting is useful for cleaner-looking output, especially when you are dealing with money. If you were trying to figure out how much sales tax you'd pay if the tax rate were 7% and the amount were $56.76, without this type of formatting help, you'd get an answer of $3.9732. But $3.97 is not only cleaner looking but correct. So sometimes you need to do this type of formatting.

The second line shows the same type of formatting, but with only one value to the right of the decimal place, showing that you can be as exact as you want. Without formatting, Python defaults to using six digits, but you can actually use formatting to print more than (or less than) six.

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:

```
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

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

```
# name and then displays it in a last, first format

# Ask the user for their first name
fname = input("What is your first name? ")

# Ask the user for their last name
lname = prompt("What is your last name? ")

print("First name first: ", fname, lname);
print("Last name first: ", lname, ",", fname);
```

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.

LISTING 4.5 **You can use** `Input` **to simulate a cash register program for a small store**

```
# 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!")
# 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
subtotal = candytotal + drinktotal + gastotal

# Don't forget sales tax! 7.25% in this example
tax = subtotal * .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" % subtotal)
print("Tax           $ %.2f" % tax)
print("Total         $ %.2f" % (subtotal+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    Qnt     Total
------------------------------
Candy   4       $5.00
Drinks  6       $13.50
Gas     12      $34.55
------------------------------
Subtotal        $ 53.05
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:

   ```
   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:

   ```
   # 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

A

C

D

G

H

I

J

M

P

Q

R

T

U

V

W

X - Y - Z