THIRD EDITION

PYTHON PROGRAMMING:

AN INTRODUCTION TO COMPUTER SCIENCE

JOHN ZELLE

FRANKLIN, BEEDLE
[INDEPENDENT PUBLISHERS SINCE 1985]

PYTHON PROGRAMMING

AN INTRODUCTION TO COMPUTER SCIENCE

THIRD EDITION

John M. Zelle

Wartburg College

Franklin, Beedle & Associates Inc. ❖ 2154 NE Broadway, Suite 100 ❖ Portland, Oregon 97232 ❖ 503/284-6348 ❖ www.fbeedle.com

Publisher — Tom Sumner (tsumner@fbeedle.com)
Editor — Brenda Jones
Production Associate — Jaron Ayres
Cover Photography — Jim Leisy ©2012

Printed in the U.S.A.

Library of Congress Cataloging-in-Publication data

Names: Zelle, John M., author.
Title: Python programming : an introduction to computer science / John M. Zelle, Wartburg College.
Description: Third edition. | Portland, Oregon : Franklin, Beedle & Associates Inc., [2016] | Includes bibliographical references and index.
Identifiers: LCCN 2016024338 | ISBN 9781590282755
Subjects: LCSH: Python (Computer program language)
Classification: LCC QA76.73.P98 Z98 2016 | DDC 005.13/3--dc23
LC record available at https://lccn.loc.gov/2016024338

Contents

Foreword

When the publisher first sent me a draft of this book, I was immediately excited. Disguised as a Python textbook, it is really an introduction to the fine art of programming, using Python merely as the preferred medium for beginners. This is how I have always imagined Python would be most useful in education: not as the only language, but as a first language, just as in art one might start learning to draw using a pencil rather than trying to paint in oil right away.

The author mentions in his preface that Python is near-ideal as a first programming language, without being a "toy language." As the creator of Python I don't want to take full credit for this: Python was derived from ABC, a language designed to teach programming in the early 1980s by Lambert Meertens, Leo Geurts, and others at CWI (National Research Institute for Mathematics and Computer Science) in Amsterdam. If I added anything to their work, it was making Python into a non-toy language, with a broad user base and an extensive collection of standard and third-party application modules.

I have no formal teaching experience, so I may not be qualified to judge its educational effectiveness. Still, as a programmer with nearly 30 years experience, reading through the chapters I am continuously delighted by the book's clear explanations of difficult concepts. I also like the many good excercises and questions which both test understanding and encourage thinking about deeper issues.

Reader of this book, congratulations! You will be well rewarded for studying Python. I promise you'll have fun along the way, and I hope you won't forget your first language once you have become a proficient software developer.

—*Guido van Rossum*

Preface

This book is designed to be used as a primary textbook in a college-level first course in computing. It takes a fairly traditional approach, emphasizing problem solving, design, and programming as the core skills of computer science. However, these ideas are illustrated using a non-traditional language, namely Python. In my teaching experience, I have found that many students have difficulty mastering the basic concepts of computer science and programming. Part of this difficulty can be blamed on the complexity of the languages and tools that are most often used in introductory courses. Consequently, this textbook was written with a single overarching goal: to introduce fundamental computer science concepts as simply as possible without being simplistic. Using Python is central to this goal.

Traditional systems languages such as C++, Ada, and Java evolved to solve problems in large-scale programming, where the primary emphasis is on structure and discipline. They were not designed to make writing small- or medium-scale programs easy. The recent rise in popularity of scripting (sometimes called "agile") languages, such as Python, suggests an alternative approach. Python is very flexible and makes experimentation easy. Solutions to simple problems are simply and elegantly expressed. Python provides a great laboratory for the neophyte programmer.

Python has a number of features that make it a near-perfect choice as a first programming language. The basic structures are simple, clean, and well designed, which allows students to focus on the primary skills of algorithmic thinking and program design without getting bogged down in arcane language details. Concepts learned in Python carry over directly to subsequent study of

systems languages such as C++ and Java. But Python is not a "toy language." It is a real-world production language that is freely available for virtually every programming platform and comes standard with its own easy-to-use integrated programming environment. The best part is that Python makes learning to program fun again.

Although I use Python as the language, teaching Python is not the main point of this book. Rather, Python is used to illustrate fundamental principles of design and programming that apply in any language or computing environment. In some places I have purposely avoided certain Python features and idioms that are not generally found in other languages. There are many good books about Python on the market; this book is intended as an introduction to computing. Besides using Python, there are other features of this book designed to make it a gentler introduction to computer science. Some of these features include:

- **Extensive use of computer graphics.** Students love working on programs that include graphics. This book presents a simple-to-use graphics package (provided as a Python module) that allows students both to learn the principles of computer graphics and to practice object-oriented concepts without the complexity inherent in a full-blown graphics library and event-driven programming.

- **Interesting examples.** The book is packed with complete programming examples to solve real problems.

- **Readable prose.** The narrative style of the book introduces key computer science concepts in a natural way as an outgrowth of a developing discussion. I have tried to avoid random facts or tangentially related sidebars.

- **Flexible spiral coverage.** Since the goal of the book is to present concepts simply, each chapter is organized so that students are introduced to new ideas in a gradual way, giving them time to assimilate an increasing level of detail as they progress. Ideas that take more time to master are introduced in early chapters and reinforced in later chapters.

- **Just-in-time object coverage.** The proper place for the introduction of object-oriented techniques is an ongoing controversy in computer science education. This book is neither strictly "objects early" nor "objects late," but gradually introduces object concepts after a brief initial grounding in the basics of imperative programming. Students learn multiple design

techniques, including top-down (functional decomposition), spiral (prototyping), and object-oriented methods. Additionally, the textbook material is flexible enough to accommodate other approaches.

- **Extensive end-of-chapter problems.** Exercises at the end of every chapter provide ample opportunity for students to reinforce their mastery of the chapter material and to practice new programming skills.

Changes in the Second and Third Editions

The first edition of the textbook has aged gracefully, and the approach it takes remains just as relevant now as when it was first published.

While fundamental principles do not change, the technology environment does. With the release of Python 3.0, updates to the original material became necessary. The second edition was basically the same as the original textbook, except that it was updated to use Python 3. Virtually every program example in the book had to be modified for the new Python. Additionally, to accommodate certain changes in Python (notably the removal of the string library), the material was reordered slightly to cover object terminology before discussing string processing. A beneficial side effect of this change was an even earlier introduction of computer graphics to pique student interest.

The third edition continues the tradition of updating the text to reflect new technologies while maintaining a time-tested approach to teaching introductory computer science. An important change to this edition is the removal of most uses of `eval` and the addition of a discussion of its dangers. In our increasingly connected world, it's never too early to begin considering computer security issues.

Several new graphics examples, developed throughout chapters 4–12, have been added to introduce new features of the graphics library that support animations, including simple video game development. This brings the text up to date with the types of final projects that are often assigned in modern introductory classes.

Smaller changes have been made throughout the text, including:

- Material on file dialogs has been added in Chapter 5.
- Chapter 6 has been expanded and reorganized to emphasize value-returning functions.

- Coverage has been streamlined and simplified to use IDLE (the standard "comes-with-Python" development environment) consistently. This makes the text more suitable for self-study as well as for use as a classroom textbook.

- Technology references have been updated.

- To further accommodate self-studiers, end-of-chapter solutions for this third edition are freely available online. Classroom instructors wishing to use alternative exercises can request those from the publisher. Self-studiers and instructors alike can visit https://fbeedle.com for details.

Coverage Options

In keeping with the goal of simplicity, I have tried to limit the amount of material that would not be covered in a first course. Still, there is probably more material here than can be covered in a typical one-semester introduction. My classes cover virtually all of the material in the first 12 chapters in order, though not necessarily covering every section in depth. One or two topics from Chapter 13 ("Algorithm Design and Recursion") are generally interspersed at appropriate places during the term.

Recognizing that different instructors prefer to approach topics in different ways, I have tried to keep the material relatively flexible. Chapters 1–4 ("Computers and Programs," "Writing Simple Programs," "Computing with Numbers," "Objects and Graphics") are essential introduction and should probably be covered in order. The initial portions of Chapter 5 ("Sequences: Strings, Lists, and Files") on string processing are also fundamental, but the later topics such as string formatting and file processing can be delayed until needed later on. Chapters 6–8 ("Defining Functions," "Decision Structures," and "Loop Structures and Booleans") are designed to stand independently and can be taken in virtually any order. Chapters 9–12 on design approaches are written to be taken in order, but the material in Chapter 11 ("Data Collections") could easily be moved earlier, should the instructor want to cover lists (arrays) before various design techniques. Instructors wishing to emphasize object-oriented design need not spend much time on Chapter 9. Chapter 13 contains more advanced material that may be covered at the end or interspersed at various places throughout the course.

Acknowledgments

My approach to CS1 has been influenced over the years by many fine textbooks that I have read and used for classes. Much that I have learned from those books has undoubtedly found its way into these pages. There are a few specific authors whose approaches have been so important that I feel they deserve special mention. A.K. Dewdney has always had a knack for finding simple examples that illustrate complex issues; I have borrowed a few of those and given them new legs in Python. I also owe a debt to wonderful textbooks from both Owen Astrachan and Cay Horstmann. The graphics library I introduce in Chapter 4 was directly inspired by my experience teaching with a similar library designed by Horstmann. I also learned much about teaching computer science from Nell Dale, for whom I was fortunate enough to serve as a TA when I was a graduate student at the University of Texas.

Many people have contributed either directly or indirectly to the production of this book. I have also received much help and encouragement from my colleagues (and former colleagues) at Wartburg College: Lynn Olson for his unflagging support at the very beginning; Josef Breutzmann, who supplied many project ideas; and Terry Letsche, who prepared PowerPoint slides for the first and third editions.

I want to thank the following individuals who read or commented on the manuscript for the first edition: Rus May, Morehead State University; Carolyn Miller, North Carolina State University; Guido Van Rossum, Google; Jim Sager, California State University, Chico; Christine Shannon, Centre College; Paul Tymann, Rochester Institute of Technology; Suzanne Westbrook, University of Arizona. I am grateful to Dave Reed at Capital University, who used early versions of the first edition, offered numerous insightful suggestions, and worked with Jeffrey Cohen at University of Chicago to supply alternate end-of-chapter exercises for this edition. Ernie Ackermann test drove the second edition at Mary Washington College. The third edition was test driven in classes by Theresa Migler at California Polytechnic State University in San Luis Obispo and my colleague Terry Letsche; and David Bantz provided feedback on a draft. Thanks to all for their valuable observations and suggestions.

I also want to acknowledge the fine folks at Franklin, Beedle, and Associates, especially Tom Sumner, Brenda Jones, and Jaron Ayres, who turned my pet project into a real textbook. This edition is dedicated to the memory of Jim Leisy, the founder of Franklin, Beedle and Associates, who passed away unex-

pectedly as the third edition was getting off the ground. Jim was an amazing man of unusually wide-ranging interests. It was his vision, guidance, relentless enthusiasm, and a fair bit of determined prodding, that ultimately molded me into a textbook author and made this book a success.

A special thanks also goes out to all my students, who have taught me so much about teaching, and to Wartburg College for giving me sabbatical support to work on the book. Last, but most importantly, I acknowledge my wife, Elizabeth Bingham, who has served as editor, advisor, and morale booster while putting up with me during my writing spells.

—JMZ

Chapter 1 Computers and Programs

Objectives

- To understand the respective roles of hardware and software in computing systems.
- To learn what computer scientists study and the techniques that they use.
- To understand the basic design of a modern computer.
- To understand the form and function of computer programming languages.
- To begin using the Python programming language.
- To learn about chaotic models and their implications for computing.

1.1 The Universal Machine

Almost everyone has used a computer at one time or another. Perhaps you have played computer games or used a computer to write a paper, shop online, listen to music, or connect with friends via social media. Computers are used to predict the weather, design airplanes, make movies, run businesses, perform financial transactions, and control factories.

Have you ever stopped to wonder what exactly a computer is? How can one device perform so many different tasks? These basic questions are the starting point for learning about computers and computer programming.

A modern computer can be defined as "a machine that stores and manipulates information under the control of a changeable program." There are two key elements to this definition. The first is that computers are devices for manipulating information. This means we can put information into a computer, and it can transform the information into new, useful forms, and then output or display the information for our interpretation.

Computers are not the only machines that manipulate information. When you use a simple calculator to add up a column of numbers, you are entering information (the numbers) and the calculator is processing the information to compute a running sum which is then displayed. Another simple example is a gas pump. As you fill your tank, the pump uses certain inputs: the current price of gas per gallon and signals from a sensor that reads the rate of gas flowing into your car. The pump transforms this input into information about how much gas you took and how much money you owe.

We would not consider either the calculator or the gas pump as full-fledged computers, although modern versions of these devices may actually contain embedded computers. They are different from computers in that they are built to perform a single, specific task. This is where the second part of our definition comes into the picture: Computers operate under the control of a changeable program. What exactly does this mean?

A *computer program* is a detailed, step-by-step set of instructions telling a computer exactly what to do. If we change the program, then the computer performs a different sequence of actions, and hence, performs a different task. It is this flexibility that allows your PC to be at one moment a word processor, at the next moment a financial planner, and later on, an arcade game. The machine stays the same, but the program controlling the machine changes.

Every computer is just a machine for *executing* (carrying out) programs. There are many different kinds of computers. You might be familiar with Macintoshes, PCs, laptops, tablets and smartphones, but there are literally thousands of other kinds of computers both real and theoretical. One of the remarkable discoveries of computer science is the realization that all of these different computers have the same power; with suitable programming, each computer can basically do all the things that any other computer can do. In this sense, the PC that you might have sitting on your desk is really a universal machine. It can do anything you want it to do, provided you can describe the task to be accomplished in sufficient detail. Now that's a powerful machine!

1.2 Program Power

You have already learned an important lesson of computing: *Software* (programs) rules the *hardware* (the physical machine). It is the software that determines what any computer can do. Without software, computers would just be expensive paperweights. The process of creating software is called *programming*, and that is the main focus of this book.

Computer programming is a challenging activity. Good programming requires an ability to see the big picture while paying attention to minute detail. Not everyone has the talent to become a first-class programmer, just as not everyone has the skills to be a professional athlete. However, virtually anyone *can* learn how to program computers. With some patience and effort on your part, this book will help you to become a programmer.

There are lots of good reasons to learn programming. Programming is a fundamental part of computer science and is, therefore, important to anyone interested in becoming a computer professional. But others can also benefit from the experience. Computers have become a commonplace tool in our society. Understanding the strengths and limitations of this tool requires an understanding of programming. Non-programmers often feel they are slaves of their computers. Programmers, however, are truly in control. If you want to become a more intelligent user of computers, then this book is for you.

Programming can also be loads of fun. It is an intellectually engaging activity that allows people to express themselves through useful and sometimes remarkably beautiful creations. Believe it or not, many people actually write computer programs as a hobby. Programming also develops valuable problem-solving skills, especially the ability to analyze complex systems by reducing them to interactions of understandable subsystems.

As you probably know, programmers are in great demand. More than a few liberal arts majors have turned a couple of computer programming classes into a lucrative career option. Computers are so commonplace in the business world today that the ability to understand and program computers might just give you the edge over your competition regardless of your occupation. When inspiration strikes, you could be poised to write the next killer app.

1.3 What Is Computer Science?

You might be surprised to learn that computer science is not the study of computers. A famous computer scientist named Edsger Dijkstra once quipped that

computers are to computer science what telescopes are to astronomy. The computer is an important tool in computer science, but it is not itself the object of study. Since a computer can carry out any process that we can describe, the real question is "What processes can we describe?" To put it another way, the fundamental question of computer science is simply "What can be computed?" Computer scientists use numerous techniques of investigation to answer this question. The three main ones are *design*, *analysis*, and *experimentation*.

One way to demonstrate that a particular problem can be solved is to actually design a solution. That is, we develop a step-by-step process for achieving the desired result. Computer scientists call this an *algorithm*. That's a fancy word that basically means "recipe." The design of algorithms is one of the most important facets of computer science. In this book you will find techniques for designing and implementing algorithms.

One weakness of design is that it can only answer the question "What is computable?" in the positive. If I can devise an algorithm, then the problem is solvable. However, failing to find an algorithm does not mean that a problem is unsolvable. It may mean that I'm just not smart enough, or I haven't hit upon the right idea yet. This is where analysis comes in.

Analysis is the process of examining algorithms and problems mathematically. Computer scientists have shown that some seemingly simple problems are not solvable by *any* algorithm. Other problems are *intractable*. The algorithms that solve these problems take too long or require too much memory to be of practical value. Analysis of algorithms is an important part of computer science; throughout this book we will touch on some of the fundamental principles. Chapter 13 has examples of unsolvable and intractable problems.

Some problems are too complex or ill-defined to lend themselves to analysis. In such cases, computer scientists rely on experimentation; they actually implement systems and then study the resulting behavior. Even when theoretical analysis is done, experimentation is often needed in order to verify and refine the analysis. For most problems, the bottom line is whether a working, reliable system can be built. Often we require empirical testing of the system to determine that this bottom line has been met. As you begin writing your own programs, you will get plenty of opportunities to observe your solutions in action.

I have defined computer science in terms of designing, analyzing, and evaluating algorithms, and this is certainly the core of the academic discipline. These days, however, computer scientists are involved in far-flung activities, all of which fall under the general umbrella of computing. Some examples

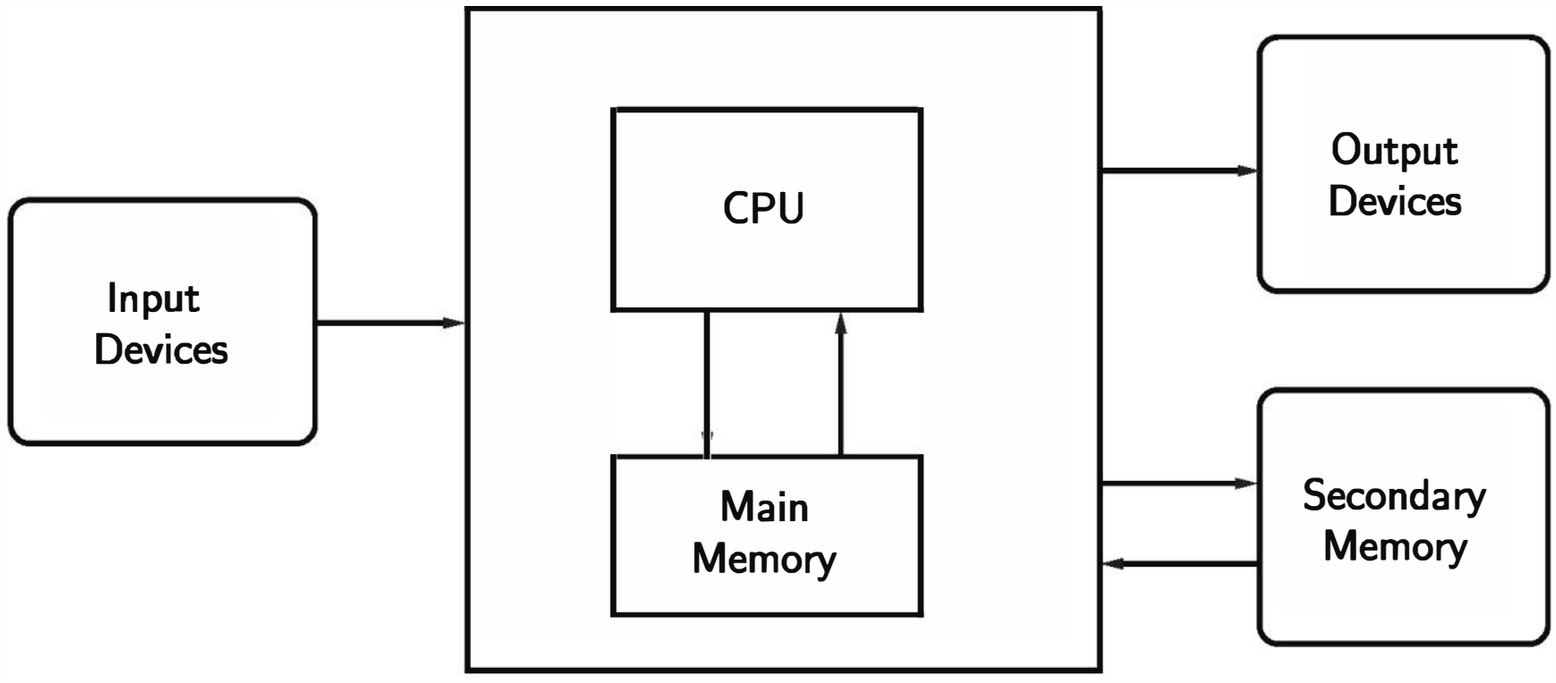

Figure 1.1: Functional view of a computer

include mobile computing, networking, human-computer interaction, artificial intelligence, computational science (using powerful computers to model scientific processes), databases and data mining, software engineering, web and multimedia design, music production, management information systems, and computer security. Wherever computing is done, the skills and knowledge of computer science are being applied.

1.4 Hardware Basics

You don't have to know all the details of how a computer works to be a successful programmer, but understanding the underlying principles will help you master the steps we go through to put our programs into action. It's a bit like driving a car. Knowing a little about internal combustion engines helps to explain why you have to do things like fill the gas tank, start the engine, step on the accelerator, and so on. You could learn to drive by just memorizing what to do, but a little more knowledge makes the whole process much more understandable. Let's take a moment to "look under the hood" of your computer.

Although different computers can vary significantly in specific details, at a higher level all modern digital computers are remarkably similar. Figure 1.1 shows a functional view of a computer. The *central processing unit* (CPU) is the "brain" of the machine. This is where all the basic operations of the computer are carried out. The CPU can perform simple arithmetic operations like adding two numbers and can also do logical operations like testing to see if two numbers are equal.

The memory stores programs and data. The CPU can directly access only information that is stored in *main memory* (called RAM for *Random Access Memory*). Main memory is fast, but it is also volatile. That is, when the power is turned off, the information in the memory is lost. Thus, there must also be some secondary memory that provides more permanent storage.

In a modern personal computer, the principal secondary memory is typically an internal hard disk drive (HDD) or a solid state drive (SSD). An HDD stores information as magnetic patterns on a spinning disk, while an SSD employs electronic circuits known as flash memory. Most computers also support removeable media for secondary memory such as USB memory "sticks" (also a form of flash memory) and DVDs (digital versatile discs), which store information as optical patterns that are read and written by a laser.

Humans interact with the computer through input and output devices. You are probably familiar with common devices such as a keyboard, mouse, and monitor (video screen). Information from input devices is processed by the CPU and may be shuffled off to the main or secondary memory. Similarly, when information needs to be displayed, the CPU sends it to one or more output devices.

So what happens when you fire up your favorite game or word processing program? First, the instructions that comprise the program are copied from the (more) permanent secondary memory into the main memory of the computer. Once the instructions are loaded, the CPU starts executing the program.

Technically the CPU follows a process called the *fetch-execute cycle*. The first instruction is retrieved from memory, decoded to figure out what it represents, and the appropriate action carried out. Then the next instruction is fetched, decoded, and executed. The cycle continues, instruction after instruction. This is really all the computer does from the time that you turn it on until you turn it off again: fetch, decode, execute. It doesn't seem very exciting, does it? But the computer can execute this stream of simple instructions with blazing speed, zipping through billions of instructions each second. Put enough simple instructions together in just the right way, and the computer does amazing things.

1.5 Programming Languages

Remember that a program is just a sequence of instructions telling a computer what to do. Obviously, we need to provide those instructions in a language that a computer can understand. It would be nice if we could just tell a computer what to do using our native language, like they do in science fiction

movies. ("Computer, how long will it take to reach planet Alphalpha at maximum warp?") Computer scientists have made great strides in this direction; you may be familiar with technologies such as Siri (Apple), Google Now (Android), and Cortana (Microsoft). But as anyone who has seriously useded such systems can attest, designing a computer program to fully understand human language is still an unsolved problem.

Even if computers could understand us, human languages are not very well suited for describing complex algorithms. Natural language is fraught with ambiguity and imprecision. For example, if I say "I saw the man in the park with the telescope," did I have the telescope, or did the man? And who was in the park? We understand each other most of the time only because all humans share a vast store of common knowledge and experience. Even then, miscommunication is commonplace.

Computer scientists have gotten around this problem by designing notations for expressing computations in an exact and unambiguous way. These special notations are called *programming languages*. Every structure in a programming language has a precise form (its *syntax*) and a precise meaning (its *semantics*). A programming language is something like a code for writing down the instructions that a computer will follow. In fact, programmers often refer to their programs as *computer code*, and the process of writing an algorithm in a programming language is called *coding*.

Python is one example of a programming language and is the language that we will use throughout this book.[1] You may have heard of some other commonly used languages, such as C++, Java, Javascript, Ruby, Perl, Scheme, or BASIC. Computer scientists have developed literally thousands of programming languages, and the languages themselves evolve over time yielding multiple, sometimes very different, versions. Although these languages differ in many details, they all share the property of having well-defined, unambiguous syntax and semantics.

All of the languages mentioned above are examples of *high-level* computer languages. Although they are precise, they are designed to be used and understood by humans. Strictly speaking, computer hardware can understand only a very low-level language known as *machine language*.

Suppose we want the computer to add two numbers. The instructions that the CPU actually carries out might be something like this:

[1]This edition of the text was developed and tested using Python version 3.4. Python 3.5 is now available. If you have an earlier version of Python installed on your computer, you should upgrade to the latest stable 3.x version to try out the examples.

```
load the number from memory location 2001 into the CPU
load the number from memory location 2002 into the CPU
add the two numbers in the CPU
store the result into location 2003
```

This seems like a lot of work to add two numbers, doesn't it? Actually, it's even more complicated than this because the instructions and numbers are represented in *binary* notation (as sequences of 0s and 1s).

In a high-level language like Python, the addition of two numbers can be expressed more naturally: `c = a + b`. That's a lot easier for us to understand, but we need some way to translate the high-level language into the machine language that the computer can execute. There are two ways to do this: a high-level language can either be *compiled* or *interpreted*.

A *compiler* is a complex computer program that takes another program written in a high-level language and translates it into an equivalent program in the machine language of some computer. Figure 1.2 shows a block diagram of the compiling process. The high-level program is called *source code*, and the resulting *machine code* is a program that the computer can directly execute. The dashed line in the diagram represents the execution of the machine code (also known as "running the program").

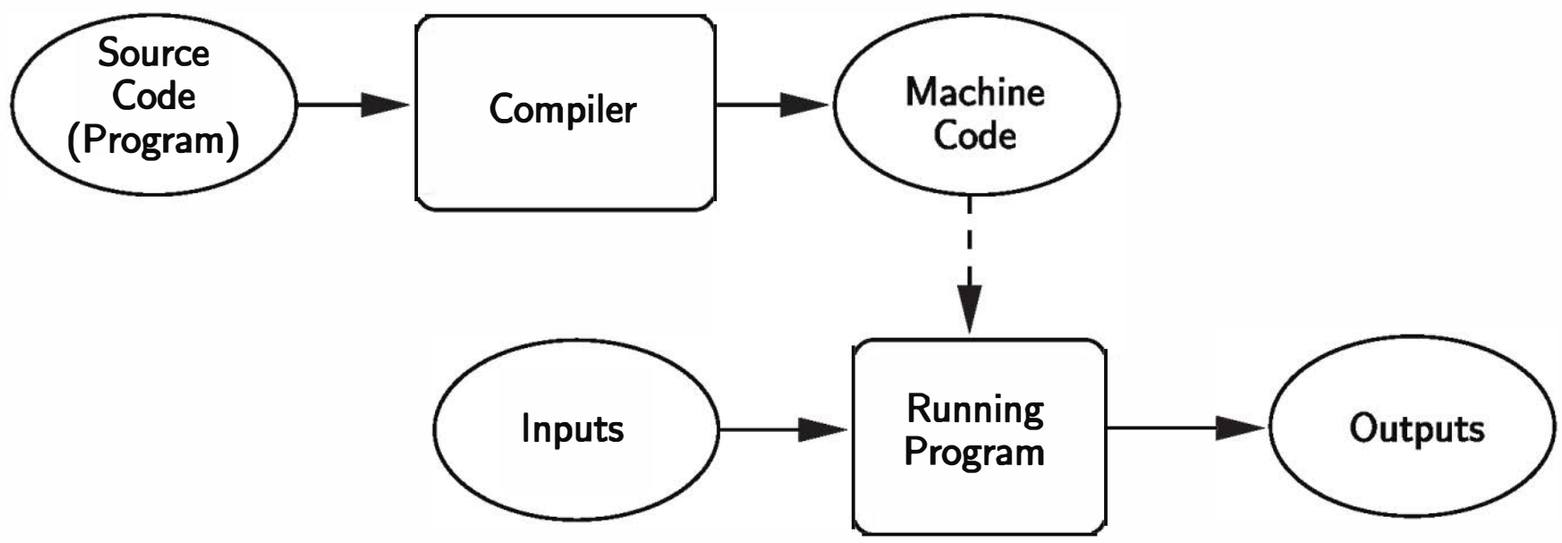

Figure 1.2: Compiling a high-level language

An *interpreter* is a program that simulates a computer that understands a high-level language. Rather than translating the source program into a machine language equivalent, the interpreter analyzes and executes the source code instruction by instruction as necessary. Figure 1.3 illustrates the process.

The difference between interpreting and compiling is that compiling is a one-shot translation; once a program is compiled, it may be run over and over again without further need for the compiler or the source code. In the interpreted

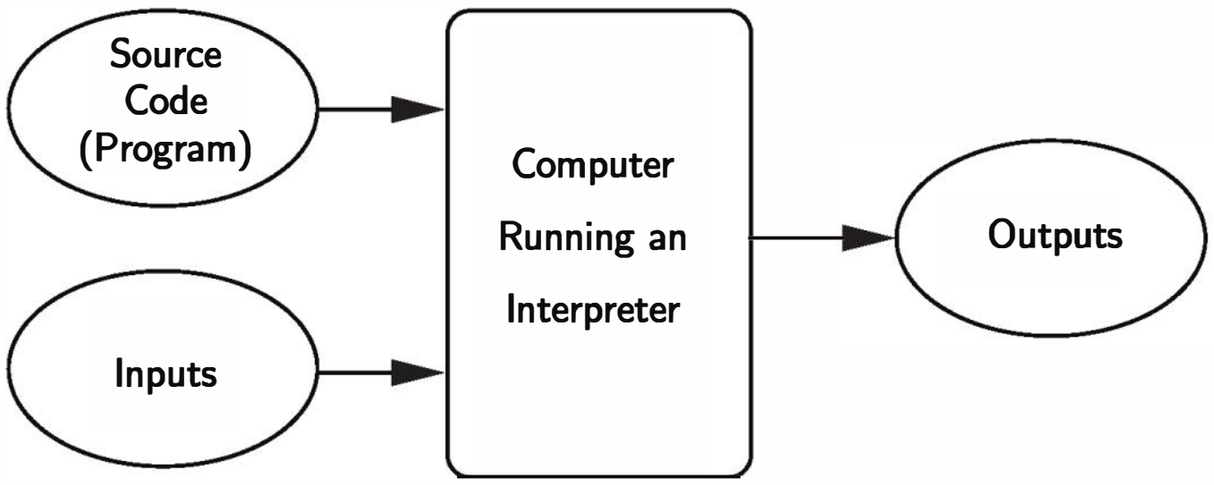

Figure 1.3: Interpreting a high-level language

case, the interpreter and the source are needed every time the program runs. Compiled programs tend to be faster, since the translation is done once and for all, but interpreted languages lend themselves to a more flexible programming environment as programs can be developed and run interactively.

The translation process highlights another advantage that high-level languages have over machine language: *portability*. The machine language of a computer is created by the designers of the particular CPU. Each kind of computer has its own machine language. A program for an Intel i7 Processor in your laptop won't run directly on an ARMv8 CPU in your smartphone. On the other hand, a program written in a high-level language can be run on many different kinds of computers as long as there is a suitable compiler or interpreter (which is just another program). As a result, I can run the exact same Python program on my laptop and my tablet; even though they have different CPUs, they both sport a Python interpreter.

1.6 The Magic of Python

Now that you have all the technical details, it's time to start having fun with Python. The ultimate goal is to make the computer do our bidding. To this end, we will write programs that control the computational processes inside the machine. You have already seen that there is no magic in this process, but in some ways programming *feels* like magic.

The computational processes inside the computer are like magical spirits that we can harness for our work. Unfortunately, those spirits only understand a very arcane language that we do not know. What we need is a friendly genie that can direct the spirits to fulfill our wishes. Our genie is a Python interpreter. We can give instructions to the Python interpreter, and it directs the underlying spirits