The Pragmatic Programmers

Practical Programming

Third Edition

An Introduction to Computer Science Using Python 3.6

Paul Gries
Jennifer Campbell
Jason Montojo
edited by Tammy Coron

What Readers Are Saying about
Practical Programming

I wish I could go back in time and give this book to my 10-year-old self when I first learned programming! It's so much more engaging, practical, and accessible than the dry introductory programming books that I tried (and often failed) to comprehend as a kid. I love the authors' hands-on approach of mixing explanations with code snippets that students can type into the Python prompt.

➤ **Philip Guo**
Creator of Online Python Tutor (www.pythontutor.com), Assistant Professor, Department of Cognitive Science, UCSD

Practical Programming delivers just what it promises: a clear, readable, usable introduction to programming for beginners. This isn't just a guide to hacking together programs. The book provides foundations to lifelong programming skills: a crisp, consistent, and visual model of memory and execution and a design recipe that will help readers produce quality software.

➤ **Steven Wolfman**
Professor of Teaching, Department of Computer Science, University of British Columbia

This excellent text reflects the authors' many years of experience teaching Python to beginning students. Topics are presented so that each leads naturally to the next, and common novice errors and misconceptions are explicitly addressed. The exercises at the end of each chapter invite interested students to explore computer science and programming language topics.

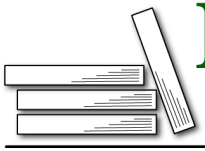
➤ **Kathleen Freeman**
Director of Undergraduate Studies, Department of Computer and Information Science, University of Oregon

Practical Programming, Third Edition

An Introduction to Computer Science Using Python 3.6

Paul Gries
Jennifer Campbell
Jason Montojo

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Development Editor: Tammy Coron
Indexing: Potomac Indexing
Copy Editor: Liz Welch
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-6805026-8-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2017

Contents

Acknowledgments

This book would be confusing and riddled with errors if it weren't for a bunch of awesome people who patiently and carefully read our drafts.

We had a great team of people provide technical reviews for this edition and previous editions: in no particular order, Frank Ruiz, Stefan Turalski, Stephen Wolff, Peter W.A. Wood, Steve Wolfman, Adam Foster, Owen Nelson, Arturo Martínez Peguero, C. Keith Ray, Michael Szamosi, David Gries, Peter Beens, Edward Branley, Paul Holbrook, Kristie Jolliffe, Mike Riley, Sean Stickle, Tim Ottinger, Bill Dudney, Dan Zingaro, and Justin Stanley. We also appreciate all the people who reported errata: your feedback was invaluable.

Greg Wilson started us on this journey when he proposed that we write a textbook, and he was our guide and mentor as we worked together to create the first edition of this book.

Finally, we would like to thank our editor Tammy Coron, who set up a workflow that made the tight timeline possible. Tammy, your gentle nudges kept us on track (squirrel!) and helped us complete this third edition in record time.

Preface

This book uses the Python programming language to teach introductory computer science topics and a handful of useful applications. You'll certainly learn a fair amount of Python as you work through this book, but along the way you'll also learn about issues that every programmer needs to know: ways to approach a problem and break it down into parts, how and why to document your code, how to test your code to help ensure your program does what you want it to, and more.

We chose Python for several reasons:

- *It is free and well documented.* In fact, Python is one of the largest and best-organized open source projects going.
- *It runs everywhere.* The reference implementation, written in C, is used on everything from cell phones to supercomputers, and it's supported by professional-quality installers for Windows, macOS, and Linux.
- *It has a clean syntax.* Yes, every language makes this claim, but during the several years that we have been using it at the University of Toronto, we have found that students make noticeably fewer "punctuation" mistakes with Python than with C-like languages.
- *It is relevant.* Thousands of companies use it every day: it is one of the languages used at Google, Industrial Light & Magic uses it extensively, and large portions of the game EVE Online are written in Python. It is also widely used by academic research groups.
- *It is well supported by tools.* Legacy editors like vi and Emacs all have Python editing modes, and several professional-quality IDEs are available. (We use IDLE, the free development environment that comes with a standard Python installation.)

Our Approach

We have organized the book into two parts. The first covers fundamental programming ideas: how to store and manipulate information (numbers, text, lists, sets, dictionaries, and files), how to control the flow of execution (conditionals and loops), how to organize code (functions and modules), how to ensure your code works (testing and debugging), and how to plan your program (algorithms).

The second part of the book consists of more or less independent chapters on more advanced topics that assume all the basic material has been covered. The first of these chapters shows how to create and manage your own types of information. It introduces object-oriented concepts such as encapsulation, inheritance, and polymorphism. The other chapters cover testing, databases, and graphical user interface construction.

Further Reading

Lots of other good books on Python programming exist. Some are accessible to novices, such as *Introduction to Computing and Programming in Python: A Multimedia Approach [GE13]* and *Python Programming: An Introduction to Computer Science [Zel03]*; others are for anyone with any previous programming experience (*How to Think Like a Computer Scientist: Learning with Python [DEM02]*, *Object-Oriented Programming in Python [GL07]*, and *Learning Python [Lut13]*). You may also want to take a look at *Python Education Special Interest Group (EDU-SIG) [Pyt11]*, the special interest group for educators using Python.

Python Resources

Information about a variety of Python books and other resources is available at http://wiki.python.org/moin/FrontPage.

After you have a good grasp of programming in Python, we recommend that you learn a second programming language. There are many possibilities, such as well-known languages like C, Java, C#, and Ruby. Python is similar in concept to those languages. However, you will likely learn more *and become a better programmer* if you learn a programming language that requires a different mindset, such as Racket,[1] Erlang,[2] or Haskell.[3] In any case, we strongly recommend learning a second programming language.

1. See http://www.ccs.neu.edu/home/matthias/HtDP2e/index.html.
2. See http://learnyousomeerlang.com.
3. See http://learnyouahaskell.com.

What You'll See

In this book, we'll do the following:

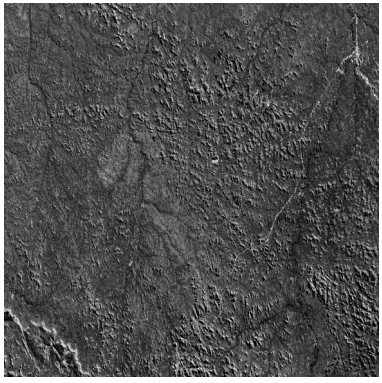
- We'll show you how to develop and use programs that solve real-world problems. Most of the examples will come from science and engineering, but the ideas can be applied to any domain.
- We'll start by teaching you the core features of Python. These features are included in most modern programming languages, so you can use what you learn no matter what you work on next.
- We'll also teach you how to think methodically about programming. In particular, we will show you how to break complex problems into simple ones and how to combine the solutions to those simpler problems to create complete applications.
- Finally, we'll introduce some tools that will help make your programming more productive, as well as some others that will help your applications cope with larger problems.

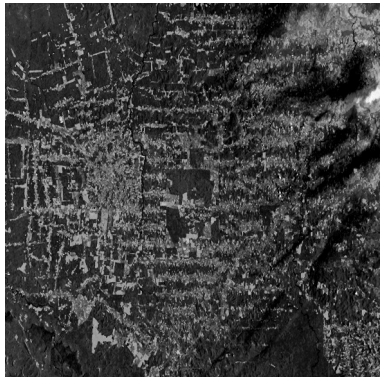
Online Resources

All the source code, errata, discussion forums, installation instructions, and exercise solutions are available at http://pragprog.com/book/gwpy3/practical-programming.

CHAPTER 1

What's Programming?

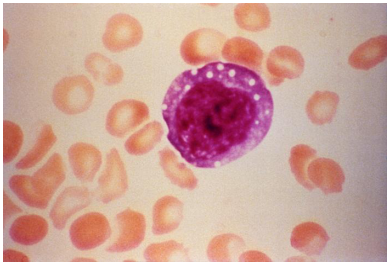




(Photo credit: NASA/Goddard Space Flight Center Scientific Visualization Studio)

Take a look at the pictures above. The first one shows forest cover in the Amazon basin in 1975. The second one shows the same area twenty-six years later. Anyone can see that much of the rainforest has been destroyed, but how much is "much"?

Now look at this:



(Photo credit: CDC)

Are these blood cells healthy? Do any of them show signs of leukemia? It would take an expert doctor a few minutes to tell. Multiply those minutes by the number of people who need to be screened. There simply aren't enough human doctors in the world to check everyone.

This is where computers come in. Computer programs can measure the differences between two pictures and count the number of oddly shaped platelets in a blood sample. Geneticists use programs to analyze gene sequences; statisticians, to analyze the spread of diseases; geologists, to predict the effects of earthquakes; economists, to analyze fluctuations in the stock market; and climatologists, to study global warming. More and more scientists are writing programs to help them do their work. In turn, those programs are making entirely new kinds of science possible.

Of course, computers are good for a lot more than just science. We used computers to write this book. Your smartphone is a pretty powerful computer; you've probably used one today to chat with friends, check your lecture notes, or look for a restaurant that serves pizza *and* Chinese food. Every day, someone figures out how to make a computer do something that has never been done before. Together, those "somethings" are changing the world.

This book will teach you how to make computers do what *you* want them to do. You may be planning to be a doctor, a linguist, or a physicist rather than a full-time programmer, but whatever you do, being able to program is as important as being able to write a letter or do basic arithmetic.

We begin in this chapter by explaining what programs and programming are. We then define a few terms and present some useful bits of information for course instructors.

Programs and Programming

A *program* is a set of instructions. When you write down directions to your house for a friend, you are writing a program. Your friend "executes" that program by following each instruction in turn.

Every program is written in terms of a few basic operations that its reader already understands. For example, the set of operations that your friend can understand might include the following: "Turn left at Darwin Street," "Go forward three blocks," and "If you get to the gas station, turn around—you've gone too far."

Computers are similar but have a different set of operations. Some operations are mathematical, like "Take the square root of a number," whereas others include "Read a line from the file named `data.txt`" and "Make a pixel blue."

The most important difference between a computer and an old-fashioned calculator is that you can "teach" a computer new operations by defining them in terms of old ones. For example, you can teach the computer that "Take the average" means "Add up the numbers in a sequence and divide by the sequence's size." You can then use the operations you have just defined to create still more operations, each layered on top of the ones that came before. It's a lot like creating life by putting atoms together to make proteins and then combining proteins to build cells, combining cells to make organs, and combining organs to make a creature.

Defining new operations and combining them to do useful things is the heart and soul of programming. It is also a tremendously powerful way to think about other kinds of problems. As Professor Jeannette Wing wrote in *Computational Thinking [Win06]*, computational thinking is about the following:

- *Conceptualizing, not programming.* Computer science isn't computer programming. Thinking like a computer scientist means more than being able to program a computer: it requires thinking at multiple levels of abstraction.
- *A way that humans, not computers, think.* Computational thinking is a way humans solve problems; it isn't trying to get humans to think like computers. Computers are dull and boring; humans are clever and imaginative. We humans make computers exciting. Equipped with computing devices, we use our cleverness to tackle problems we wouldn't dare take on before the age of computing and build systems with functionality limited only by our imaginations.
- *For everyone, everywhere.* Computational thinking will be a reality when it becomes so integral to human endeavors it disappears as an explicit philosophy.

We hope that by the time you have finished reading this book, you will see the world in a slightly different way.

What's a Programming Language?

Directions to the nearest bus station can be given in English, Portuguese, Mandarin, Hindi, and many other languages. As long as the people you're talking to understand the language, they'll get to the bus station.

In the same way, there are many programming languages, and they all can add numbers, read information from files, and make user interfaces with windows and buttons and scroll bars. The instructions look different, but

they accomplish the same task. For example, in the Python programming language, here's how you add 3 and 4:

```
3 + 4
```

But here's how it's done in the Scheme programming language:

```
(+ 3 4)
```

They both express the same idea—they just look different.

Every programming language has a way to write mathematical expressions, repeat a list of instructions a number of times, choose which of two instructions to do based on the current information you have, and much more. In this book, you'll learn how to do these things in the Python programming language. Once you understand Python, learning the next programming language will be much easier.

What's a Bug?

Pretty much everyone has had a program crash. A standard story is that you were typing in a paper when, all of a sudden, your word processor crashed. You had forgotten to save, and you had to start all over again. Old versions of Microsoft Windows used to crash more often than they should have, showing the dreaded "blue screen of death." (Happily, they've gotten a *lot* better in the past several years.) Usually, your computer shows some kind of cryptic error message when a program crashes.

What happened in each case is that the people who wrote the program told the computer to do something it couldn't do: open a file that didn't exist, perhaps, or keep track of more information than the computer could handle, or maybe repeat a task with no way of stopping other than by rebooting the computer. (Programmers don't mean to make these kinds of mistakes, they are just part of the programming process.)

Worse, some bugs don't cause a crash; instead, they give incorrect information. (This is worse because at least with a crash you'll notice that there's a problem.) As a real-life example of this kind of bug, the calendar program that one of the authors uses contains an entry for a friend who was born in 1978. That friend, according to the calendar program, had his 5,875,542nd birthday this past February. Bugs can be entertaining, but they can also be tremendously frustrating.

Every piece of software that you can buy has bugs in it. Part of your job as a programmer is to minimize the number of bugs and to reduce their severity. In order to find a bug, you need to track down where you gave the wrong

instructions, then you need to figure out the right instructions, and then you need to update the program without introducing other bugs.

Every time you get a software update for a program, it is for one of two reasons: new features were added to a program or bugs were fixed. It's always a game of economics for the software company: are there few enough bugs, and are they minor enough or infrequent enough in order for people to pay for the software?

In this book, we'll show you some fundamental techniques for finding and fixing bugs and also show you how to prevent them in the first place.

The Difference Between Brackets, Braces, and Parentheses

One of the pieces of terminology that causes confusion is what to call certain characters. Several dictionaries use these names, so this book does too:

() Parentheses

[] Brackets

{} Braces (Some people call these *curly brackets* or *curly braces*, but we'll stick to just *braces*.)

Installing Python

Installation instructions and use of the IDLE programming environment are available on the book's website: http://pragprog.com/titles/gwpy3/practical-programming.

CHAPTER 2

Hello, Python

Programs are made up of commands that tell the computer what to do. These commands are called *statements*, which the computer executes. This chapter describes the simplest of Python's statements and shows how they can be used to do arithmetic, which is one of the most common tasks for computers and also a great place to start learning to program. It's also the basis of almost everything that follows.

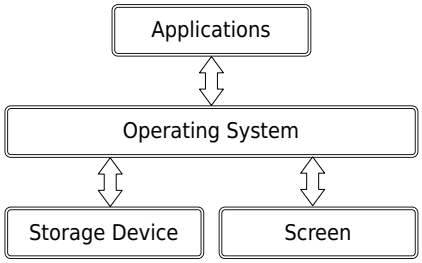
How Does a Computer Run a Python Program?

In order to understand what happens when you're programming, it helps to have have a mental model of how a computer executes a program.

The computer is assembled from pieces of hardware, including a *processor* that can execute instructions and do arithmetic, a place to store data such as a *hard drive*, and various other pieces, such as a screen, a keyboard, an Ethernet controller for connecting to a network, and so on.

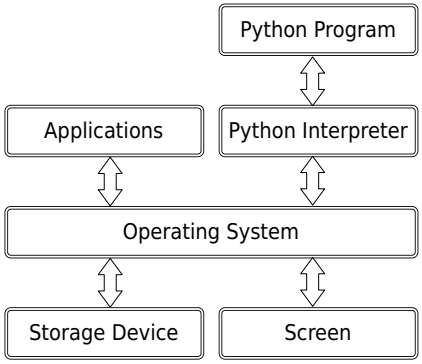
To deal with all these pieces, every computer runs some kind of *operating system*, such as Microsoft Windows, Linux, or macOS. An operating system, or OS, is a program; what makes it special is that it's the only program on the computer that's allowed direct access to the hardware. When any other application (such as your browser, a spreadsheet program, or a game) wants to draw on the screen, find out what key was just pressed on the keyboard, or fetch data from storage, it sends a request to the OS (see the top image on page 8).

This may seem like a roundabout way of doing things, but it means that only the people writing the OS have to worry about the differences between one graphics card and another and whether the computer is connected to a network through Ethernet or wireless. The rest of us—everyone analyzing



scientific data or creating 3D virtual chat rooms—only have to learn our way around the OS, and our programs will then run on thousands of different kinds of hardware.

Today, it's common to add another layer between the programmer and the computer's hardware. When you write a program in Python, Java, or Visual Basic, it doesn't run directly on top of the OS. Instead, another program, called an *interpreter* or *virtual machine*, takes your program and runs it for you, translating your commands into a language the OS understands. It's a lot easier, more secure, and more portable across operating systems than writing programs directly on top of the OS:



There are two ways to use the Python interpreter. One is to tell it to execute a Python program that is saved in a file with a .py extension. Another is to interact with it in a program called a *shell*, where you type statements one at a time. The interpreter will execute each statement when you type it, do what the statement says to do, and show any output as text, all in one window. We will explore Python in this chapter using a Python shell.

Install Python Now (If You Haven't Already)

If you haven't yet installed Python 3.6, please do so now. (Python 2 won't do; there are significant differences between Python 2 and Python 3, and this book uses Python 3.6.) Locate installation instructions on the book's website: http://pragprog.com/titles/gwpy3/practical-programming.

Programming requires practice: you won't learn how to program just by reading this book, much like you wouldn't learn how to play guitar just by reading a book on how to play guitar.

Python comes with a program called IDLE, which we use to write Python programs. IDLE has a Python shell that communicates with the Python interpreter and also allows you to write and run programs that are saved in a file.

We *strongly* recommend that you open IDLE and follow along with our examples. Typing in the code in this book is the programming equivalent of repeating phrases back to an instructor as you're learning to speak a new language.

Expressions and Values: Arithmetic in Python

You're familiar with mathematical expressions like `3 + 4` ("three plus four") and `2 - 3 / 5` ("two minus three divided by five"); each expression is built out of *values* like `2`, `3`, and `5` and *operators* like `+` and `-`, which combine their *operands* in different ways. In the expression `4 / 5`, the operator is "/" and the operands are `4` and `5`.

Expressions don't have to involve an operator: a number by itself is an expression. For example, we consider `212` to be an expression as well as a value.

Like any programming language, Python can *evaluate* basic mathematical expressions. For example, the following expression adds `4` and `13`:

```
>>> 4 + 13
17
```

The `>>>` symbol is called a *prompt*. When you opened IDLE, a window should have opened with this symbol shown; you don't type it. It is prompting you to type something. Here we typed `4 + 13`, and then we pressed the Return (or Enter) key in order to signal that we were done entering that *expression*. Python then evaluated the expression.

When an expression is evaluated, it produces a single value. In the previous expression, the evaluation of `4 + 13` produced the value `17`. When you type the expression in the shell, Python shows the value that is produced.