Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

Preface

Copyright©2024 by Tim Teitelbaum; Most recent revision, 12/03/2024

Methodology Orientation

This is an introduction to computer programming aimed at the level of a first college course. It is also suitable as a monograph for people beyond the introductory level who are unfamiliar with its methodological content. It is a *methodology-oriented* introduction, and its subject is programming principles, not language features.

In elementary Physics, one doesn't start learning mechanics by studying one or another brand of springs and pulleys; rather, one learns Newton's Laws and how to apply them in arbitrary situations. Similarly, in this book, I eschew the study of any particular brand of programming language, opting instead to focus on fundamental laws formulated as rules of program composition.

I use a minimal programming language, one so small that it can be said to be universal. Programming skill is measured by the ease with which you can turn a problem statement into a working program, not by the number of language features you know. The methodology presented is not specific to a particular language; rather, it applies to programming, in general.

The notation used is a small subset of Python, but I hasten to repeat: The book is about programming, not programming in Python. For our purposes, all imperative programming languages, e.g., Java, Python, JavaScript, C/C++, etc., share a common core.

The notation is readily summarized in Chapter 2 Prerequisites. For students with a modicum of background, this chapter will be a succinct refresher that firms up prior knowledge, provides standard vocabulary, and establishes a common baseline for the rest of the book. Students with no background whatsoever can learn the material from the chapter, but may wish to supplement it, e.g., with one of the many excellent and free resources on the Internet. Instructors may wish to offer a lecture, or a few recitation sections, to bring everyone up to speed.

Precepts

A premise of the book is that much of programming can be reduced to a set of rules you can follow in cookbook fashion. The conceit is that programming can (almost) be algorithmic. You, the programmer, just follow the rules, and out will pop a program. And not just any program, but a reasonably good program, at that. You play the role of a computer, and just follow the *programming precepts* taught. The chapters teach the precepts, and illustrate how they are applied.

Precepts are written as imperatives, albeit they are couched in equivocating phrases such as "seek", "consider", "if possible", "prefer", etc. to allow for the possibility that other (perhaps contradictory) precepts take precedence. Thus, I straddle the gap between the fiction that coding can be deterministic (just follow the rules) and the fiction that coding is pure design (inexplicable creativity).

One of my themes is the use of *programming patterns*, short fragments of code that perform frequently needed tasks. These patterns arise so often that they are best mastered as if they were primitives of the programming language. Patterns are introduced and discussed throughout the book.

You are encouraged to learn each pattern so well that it becomes an atomic notion in your programming vocabulary. When, in the course of programming, you see the need to do something for which there is an established pattern, you should be able to recognize the pattern's applicability, and then immediately blast it into your program in one indivisible action. In the parlance of cognitive psychology, you should have *chunked* the pattern, and should no longer think of it in terms of its constituent parts.

The book's focus is synthesis, not analysis. Thus, no substantial code is presented as a *fait accompli* for interpretation. Rather, the essential content of the book is the stepwise development of solutions rather than the solutions *per se*.

In cases where more than one approach comes to mind, each will be considered, explored, and evaluated. A few examples are consequential *algorithms*. My purpose, however, is instruction in programming, not algorithms. As such, although an example may have a well-deserved reputation and a noteworthy asymptotic running-time complexity, these will be incidental to its use in illustrating how you might develop the program yourself.

Code is presented in a sequence of incremental steps that are displayed in numbered “movie” frames. Each coding “movie” starts with a specification in frame one, and ends with the finished product. That way, you are shown a recommended order of development, and not just the final program.

Much of the power of computers derives from fashioning conceptual hierarchies at varying levels of abstraction, ranging from high-level ideas to low-level details. The notion of a *specification* and its *implementation* is central in that activity.

A specification defines *what* must be accomplished (at one level of abstraction), and its *refinement* into an implementation defines *how* to accomplish that (in terms of lower-levels of abstraction). This process, known as *stepwise refinement*, is essential to the book's methodology.

The principle of *information hiding* is introduced early as a mechanism that allows a program's different levels of abstraction to be separated from one another. Information hiding, and the related concepts of *modularity* and *encapsulation*, are often presented as an aspect of *object-orientation*. In contrast, I present them as separate notions well before objects.

Because *objects* implicate too many language-specific details, I defer them until late, which allows the rest of the book to be truly language-independent.

Much of the power of a modern programming language comes from its *libraries*. If you plan to do any serious programming in a given language, you will surely want to master its libraries, and use them rather than "reinventing the wheel".

For pedagogical purposes, however, I focus on how to program in the base programming language, and largely ignore libraries. Although their use is eschewed, libraries are far from forgotten. In fact, the text anticipates the need for generic collections, and then implements a class `ArrayList` as a motivating examples for objects. It happens that Python's builtin `list` data type is very similar to `ArrayLists`, so this example helps to clarify what is going on "under the hood" with Python lists.

Thus, you are led right to the pearly gates of libraries, armed with the ability to read and understand library interface specifications, and to use them to advantage.

Throughout the book, I advocate a cautious approach to programming that is aimed at writing correct code from the start. But mistakes are inevitable.

Debugging code is like trying to find “a needle in a haystack”, and the topic is not realistically discussed in the context of short program segments. Accordingly, the subject is deferred until the final chapter, where I deliberately introduce bugs into the largest program example of the book, and then discuss how to find them.

Language-oriented introductions to programming tend toward being encyclopedic tomes; in contrast, I have aimed for a comparatively short, coherent, and digestible book. I have aspired to tell a compelling story, knitted together by interesting, nontrivial examples that are woven throughout --- a book that invites cover-to-cover reading.

The slide version of the book is even more succinct, and benefits from some of the unique advantages of the medium, e.g., animation of both code development and program execution.

My aim in writing this book is your proficiency in programming. I wish you well.

Tim Teitelbaum
Ithaca, NY