AI-Assisted Program Design

AI-Assisted Program Design:

An Introduction to Prompt Engineering Using Python and GitHub Copilot

By

Marco T. Morazán

Cambridge Scholars Publishing

AI-Assisted Program Design: An Introduction to Prompt Engineering Using Python and GitHub Copilot

By Marco T. Morazán

This book first published 2026

Cambridge Scholars Publishing

Lady Stephenson Library, Newcastle upon Tyne, NE6 2PA, UK

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-1-0364-6254-3
ISBN (Ebook): 978-1-0364-6255-0

Dedication

To Matthias, PLT, and everyone who tirelessly works to advance the program-by-design methodology in Computer Science Education.

To my students for asking me to teach a course on how to program using AI and coding assistants.

To Walter, Madrid, and Skyler for one day carrying forward the Morazán legacy of helping others to learn.

Marco

Contents

Preface **ix**

I Introduction to Python and Program Design **1**

1 Python Syntax and Built-In Data Types **3**
1.1 Variables and Function Names 5
1.2 Built-in Single-Value Data Types 6
1.2.1 Numeric Types 6
1.2.2 Boolean Type 8
1.3 Built-in Compound Types 9
1.3.1 Set Type 9
1.3.2 Sequence Types 11
1.3.3 Mapping Types 16
1.4 Control Flow 18
1.4.1 Conditional Statements 18
1.4.2 Loops 21
1.5 Functions 27
1.5.1 Function Definitions 27
1.5.2 Unit Testing 29
1.6 Classes and Objects 32
1.6.1 A Brief Introduction 32
1.6.2 Values in Python 36
1.7 What have we learned in this chapter? 38

2 Brief Introduction to VS Code **41**
2.1 VS Code Features 41
2.1.1 Graphical User Interface 42
2.1.2 Syntax Highlighting 43
2.1.3 Debugging Tool 44
2.1.4 Git Integration 47
2.1.5 Customization 47
2.1.6 Cross-Platform 48
2.2 Installing Python 48
2.3 Writing Your First Programs 50
2.3.1 Create a Workspace 50
2.3.2 Preparing for Your First Program 50
2.4 Design Recipe for Functions 51
2.5 Using the Design Recipe 53
2.5.1 The Area of a Right Triangle 53
2.5.2 The Number with the Largest Magnitude 57
2.6 What have we learned in this chapter? 60

3 Processing Compound Data of Finite Size **63**
3.1 Defining a Student 64
3.1.1 Defining the Student Class 64
3.1.2 Testing the Class 65
3.2 Student Function Template 67
3.3 Determining if a Student is on the Dean's List 68
3.3.1 Predicate Design 68
3.3.2 An Alternative Implementation 71
3.4 Union Types 73
3.4.1 Representing Data with Variety 73
3.4.2 Processing Data with Variety 74
3.4.3 Example: Processing Coordinates 76
3.4.4 The Coordinate Constructor 77
3.5 Using Separate Classes for Subtypes 85
3.5.1 Redefining Coordinates 85
3.5.2 Refining the Template for Functions on a Coordinate 86
3.5.3 Refining Distance to Origin 88
3.6 Polymorphic Dispatch 91
3.7 What have we learned in this chapter? 94

4 Processing Compound Data of Arbitrary Size 97
4.1 Processing Lists 98
4.1.1 Data Definition 98
4.1.2 Function Template 100
4.1.3 Summing a `(listof number)` 102
4.1.4 Computing the Average of a `(listof number)` . . 103
4.2 Ranges 107
4.2.1 Data Definition 107
4.2.2 Function Template 107
4.2.3 Determining List Membership in a Range of Indexes 108
4.3 Natural Numbers 113
4.3.1 Data Definition 113
4.3.2 Function Template 114
4.3.3 Computing $\mathtt{n}^2$ 114
4.3.4 Summing the First `n` Squares 118
4.4 Binary Trees 122
4.4.1 Data Definition 122
4.4.2 Function Template 123
4.4.3 Node Class Implementation 125
4.4.4 Determining Membership in a `(bintreeof natnum)` 126
4.5 What have we learned in this chapter? 129

5 Generative Recursion 131
5.1 Template for Functions Using Generative Recursion 132
5.2 Finding the Greatest Common Divisor 134
5.2.1 Finding the `gcd` Using Structural Recursion 134
5.2.2 Finding the `gcd` Using Generative Recursion 136
5.3 Creating a List of Ordered Lists of Length 2 140
5.3.1 Data Definition for a `2lonList` 140
5.3.2 Data Definition for a `2lon` 141
5.3.3 Constructing a `2lonList` from a `evenlon` 142
5.4 The Sierpiński Triangle 145
5.4.1 `Python`'s Turtle Module 147
5.4.2 Data Definitions and Design Idea 148
5.4.3 The `sierpinski` Function Definition 150
5.4.4 The Auxiliary Functions 151
5.5 What have we learned in this chapter? 157

6 Accumulative Recursion 159
6.1 Revisiting Summing a List of Numbers . . . 160
6.1.1 Design Idea . . . 161
6.1.2 Implementation . . . 163
6.2 Scoping . . . 165
6.3 Computing Fibonacci Numbers . . . 167
6.3.1 Refactoring `fib` Using Accumulators . . . 170
6.3.2 Implementation . . . 171
6.4 Finding a Path in a Directed Graph . . . 175
6.4.1 Design Idea . . . 176
6.4.2 The `findPathHelper` Function . . . 178
6.4.3 The `findPathFromAny` Function . . . 180
6.4.4 The `getNode` Function . . . 181
6.4.5 Alas! There is a Bug! . . . 182
6.5 Finding a Path Using an Accumulator . . . 184
6.5.1 Refactoring `findPath` . . . 185
6.5.2 Refactoring `findPathHelper` . . . 186
6.5.3 Refactoring `findPathFromAny` . . . 188
6.6 What have we learned in this chapter? . . . 191

7 Functional Abstraction 193
7.1 A Motivating Example . . . 193
7.2 List Comprehensions . . . 195
7.3 Lambda Expressions . . . 199
7.3.1 Syntax . . . 199
7.4 The Design Recipe for Abstraction . . . 201
7.5 Functional Abstraction Examples . . . 202
7.5.1 Abstracting Over the Testing Functions . . . 202
7.5.2 Abstracting Over Computing Functions . . . 203
7.6 Built-in Higher-Order Functions in `Python` . . . 207
7.6.1 The Function `map` . . . 207
7.6.2 The Function `filter` . . . 209
7.6.3 The Function `reduce` . . . 210
7.6.4 Find the Maximum of a `(listof number)` . . . 211
7.6.5 Computing `n!` . . . 212
7.7 What have we learned in this chapter? . . . 216

8 Imperative Programming 219
8.1 Is `f(x)` equal to `f(x)`? . . . 220
8.2 The Loss of Referential Transparency . . . 220
8.3 Reasoning About Mutations . . . 226
8.4 Revisiting the Greatest Common Denominator . . . 230
8.5 Computing `n!` . . . 235
8.5.1 Computing `n!` using structural recursion . . . 235
8.5.2 Computing `n!` using accumulative recursion . . . 236
8.5.3 Computing `n!` using a `while`-loop . . . 238
8.6 Sharing Values . . . 243
8.6.1 A Design Recipe for Mutators . . . 244
8.6.2 Phonebook Design . . . 245
8.7 What have we learned in this chapter? . . . 251

II Prompt Engineering 253

9 Coding Assistants 255
9.1 Large Language Models . . . 256
9.2 This is not Programming! . . . 257
9.3 Advantages of Using Coding Assistants . . . 261
9.4 Pitfalls of Using Coding Assistants . . . 265
9.4.1 Anthropomorphism . . . 265
9.4.2 Ethical Issues . . . 265
9.4.3 Development of Questionable Habits . . . 267
9.4.4 Overreliance . . . 268
9.5 What have we learned in this chapter? . . . 269

10 Program Design Using Prompts 271
10.1 Coding Assistants are Nondeterministic . . . 272
10.2 A Design Recipe for Using a Coding Assistant . . . 272
10.3 A Design Recipe for Prompts . . . 275
10.4 Composing Functions . . . 276
10.5 Golden Rectangles . . . 283
10.6 Swapping List Values . . . 294
10.7 What have we learned in this chapter? . . . 300

11 Prompt Engineering for Structural Recursion 301
11.1 Tetrahedral Numbers . . . 302
11.2 Asking a Coding Assistant for Advice . . . 309
11.2.1 Chat Interface . . . 309
11.2.2 Optimizing `tetra` and `tri` . . . 311
11.3 Computing π . . . 316
11.3.1 High-Precision Decimal Numbers in `Python` . . . 317
11.3.2 Chudnovsky Algorithm Version 0 . . . 318
11.3.3 The Factorial Function . . . 324
11.3.4 Chudnovsky Algorithm Version 1 . . . 327
11.4 Unjumbling Words . . . 337
11.4.1 Determining English Words . . . 338
11.4.2 General Design Idea and Data Definitions . . . 339
11.4.3 The Topmost Function . . . 341
11.4.4 Computing Permutations . . . 343
11.4.5 Inserting in Every Position in Every Word . . . 345
11.4.6 Inserting Everywhere in a Word . . . 347
11.4.7 Debugging the Design . . . 349
11.5 What have we learned in this chapter? . . . 355

12 Prompt Engineering for Generative Recursion 357
12.1 The Partial Digest Problem . . . 357
12.1.1 A Naive Design Idea . . . 359
12.1.2 A Practical Design Idea . . . 361
12.1.3 Implementation . . . 364
12.2 Newton's Method for Computing $\sqrt{\texttt{x}}$ . . . 374
12.2.1 Design Idea . . . 375
12.2.2 Implementation . . . 377
12.3 What have we learned in this chapter? . . . 383

13 Prompt Engineering for Accumulative Recursion 385
13.1 Computing Primes . . . 385
13.1.1 Design Idea . . . 386
13.1.2 Implementation . . . 388
13.2 Arithmetic Programming Language Evaluation . . . 391
13.2.1 Concrete Syntax for Arithmetic Expressions . . . 392
13.2.2 Abstract Syntax for Arithmetic Expressions . . . 394
13.2.3 Designing and Implementing the Parser . . . 399

13.2.4 Intentional and Extensional Equality 405
13.2.5 Unparsing 406
13.2.6 Evaluating Expressions 411
13.2.7 Environment Design and Implementation 413
13.2.8 Implementing the Evaluator 415
13.2.9 Function Application 419
13.3 What have we learned in this chapter? 425

14 Prompt Engineering for Imperative Programming 427
14.1 A Third Look at Summing a List of Numbers 428
14.2 Radix Sorting 436
14.2.1 Understanding Radix Sorting 436
14.2.2 Bucket Implementation 438
14.2.3 Sorting Implementation 448
14.3 What have we learned in this chapter? 471

Epilogue 473

Preface

Welcome to the world of prompt engineering and programming with a coding assistant! If you are reading this textbook, you must be excited about learning how to use a coding assistant as part of the programming process. Coding assistants are powerful software tools that attempt to leverage the power of Artificial Intelligence (AI) to help you solve problems and express solutions as programs. It is important to always keep in mind that coding assistants are software tools and, despite what you may read online or hear in the news, you should not think of them as being as powerful or more powerful than a skilled programmer. They are, indeed, powerful tools, but their efficacy depends on how well they are used. Using coding assistants effectively requires skills much like developing software using any programming language requires skills. Let us immediately dispel the notion that anyone can be a programmer by simply using a coding assistant. Nothing can be further from the truth. Using a coding assistant without being systematic does not make anyone a programmer.

Using a coding assistant requires three essential skills:

- Reading Code
- Analyzing Code
- Critiquing Code

Any programmer using a coding assistant must be able to read to understand a suggested answer. If you cannot read code then you will be unable to understand the algorithm behind the answer suggested by a coding assistant. It is equally important to be able to analyze code. A programmer using a coding assistant must be able to discern if the code suggested by a coding

assistant is efficient or is correct. There is no guarantee that the answers provided by a coding assistant are efficient and correct. Finally, it is essential for any programmer that uses a coding assistant to be able to critique code. A programmer must be able to discern if a proposed piece of code is, for example, safe or easy to extend as the software under development grows in complexity. Realizing, for instance, that a coding-assistant-developed website is vulnerable to attacks is unlikely to be acceptable. Critiquing the code generated by a coding assistant can help yourself and others to develop more robust software.

You may have read online or in a book, been told by a professor, or seen in a documentary that prompt engineering and using a coding assistant is an art and not a science. This book, in part, is written to debunk this misconception. An important idea found throughout this textbook is that *prompts are programs*. Admittedly, prompts are not programs written using the strict syntax of a programming language, but they are, nonetheless, programs, albeit written in a natural language (e.g., English). You may naturally ask how is this possible. To answer this question it is necessary to understand a little about how programs are evaluated. When you write a program using a programming language, the program you write is input to another program that parses and evaluates it to obtain a value or achieve an effect. Similarly, a prompt is input to another program, called a Large Language Model (`LLM`), that parses and evaluates it to predict, for example, the code a programmer needs or the answer to a question. Thus, the inescapable conclusion is that prompts are programs. Once we accept that prompts are programs, it follows that prompts need to be designed, validated, and refined just like a program written using a programming language. This textbook is about how to design, validate and refine prompts. To be able to do that requires knowing how to design, validate, and refine programs using a programming language. It is also important to keep in mind that coding assistants are trained and learn from vast amounts of software found on the internet. Much of this software is not carefully designed and, therefore, as software projects grow in size and complexity it becomes ever more likely that a coding assistant may not be able to cope with your needs. This means that the task of designing and implementing an algorithm falls on the programmers–as is always the case. A programmer may choose to design and implement a solution themselves or may choose to write carefully designed prompts to guide the coding assistant. Expecting a coding assistant to plan and carefully design a solution is an unreasonable expectation.

The next question is how are prompts designed to guide a coding assistant. Fortunately, in 2001, the German-American Computer Scientist Matthias Felleisen (1958–) and his Ph.D. students published a textbook, *How to Design Programs*, addressing how to teach systematic software development to beginners. Their approach is based on *design recipes* that take a programmer from a problem statement to a validated solution expressed as a program. A design recipe is a series of steps, each with a concrete outcome, that scaffolds the program design and implementation process. In essence, it outlines a step-by-step process through which solutions to problems are developed. Matthias and his co-authors made the science of systematic problem solving popular in Computer Science Education. Their approach has been used by a myriad of universities all over the world with great success. Building on the work of Matthias et al., the Honduran-American Computer Scientist (and author of this textbook), Marco T. Morazán (1967–), developed a 2-semester design-based curriculum for beginners and published two accompanying textbooks: *Animated Problem Solving* and *Animated Program Design*. These textbooks showcase the power of design recipes by introducing beginners to topics that are traditionally considered too advance for them such as distributed programming, nondeterministic programming (i.e., randomness in programming), heuristic-based searching, and systematic imperative programming using Hoare Logic. The success of the approach is affirmed in the smiles and the pride of every beginning student that develops a distributed multiplayer video game or a heuristic-based puzzle solving video game. Two years later, in 2024, Morazán successfully extended the use of design recipes to the realm of Theoretical Computer Science in his textbook *Programming-Based Formal Languages and Automata Theory*, where students can systematically design, run and experiment with state-based machines, grammars, and regular expressions using a domain-specific programming language, `FSM` (`F`unctional `S`tate `M`achines). The textbook makes theorems and Mathematics come alive by harnessing CS-student interest in programming.

This textbook brings the rich and successful history of systematic software development using design recipes to bear on the development of software using coding assistants. To this end, the textbook first introduces the program-by-design methodology using different problem-solving methodologies. Throughout this introduction, you will subtly learn how to write prompts. Following this introduction, coding assistants are used in full force along with design recipes to guide software development. The power of de-

sign recipes in conjunction with coding assistants is showcased by illustrating how nontrivial problems are solved.

To Students

You are about to embark on a fascinating journey exploring how to systematically use coding assistants to develop software. Like any programming endeavor, programming using a coding assistant is not easy. Do not believe the hype commonly found on the internet and in the news that coding assistants make programming easy. Anyone claiming that coding assistants make programming easy or that the world will soon no longer need programmers does not know what they are talking about. Programming and problem solving remain equally hard and enjoyable using a coding assistant as they are without using a coding assistant.

So, what is the hype all about? Coding assistants help problem solvers in significant ways. First, programmers no longer need to focus as much on the low-level details of writing programs using the strict syntax of a programming language. Coding assistants have become proficient at generating syntactically correct programs. This allows programmers to focus more on problem solving and on developing algorithms. This means, among other things, programmers can focus more on decomposing problems into subproblems to make finding a solution easier (i.e., divide and conquer), on devising a series of simpler problems that may be sequentially solved culminating in a solution to the original problem (i.e., iterative refinement), and on efficiency (i.e., the use of different data structures to make programs faster). Programmers can focus more on problem solving, but must always keep in mind that coding assistants are not infallible. It is still necessary to know the syntax of the programming language used to write software in order to check and understand code generated by a coding assistant. Second, programmers no longer need to struggle as much with cryptic error messages. Coding assistants have become proficient at explaining the meaning of such error messages. This does not mean, however, that coding assistants are proficient at explaining their own errors. A programmer will still need to point out mistakes to a coding assistant much like they point out mistakes to fellow programmers. In fact, coding assistants are notorious for generating unit tests that pass for code that is incorrect. A programmer must always be vigilant in this regard. Third, coding assistants can expose programmers to

different ways to solve a problem and to represent the data a program needs to manipulate. This is important, because it deeply influences how we think about problems. Fourth, a coding assistant can make the process of iterative refinement smoother. Coding assistants can be very proficient at refactoring code when given a clear prompt outlining how to refactor the code. That said, you must always remain vigilant, because coding assistants, as stated above, are not infallible and prompts are not always accurate or correct. These and other advantages of coding assistants are discussed in the pages of this textbook.

The use of coding assistants is also associated with significant pitfalls. These are also discussed in the pages of this textbook, but the one that ought to worry you the most is forfeiting your ability to think. It is very tempting to write a prompt that fails to outline a design idea for solving a problem and simply letting a coding assistant generate code for you. People outside of Computer Science, unfortunately, have taken to calling this programming. They do not know what they are talking about. It is not programming. It is plagiarism–a serious offense at any university. Plainly stated, no programmer deserves credit for software that they did not design. Do not use a program assistant to forfeit thinking about a problem and accept any pre-canned solution a coding assistant parrots back to you. Doing so means that you are cheating yourself out of a learning experience and that credit belongs to the person that wrote the pre-canned solution the coding assistant provides. Be aware that coding assistants are able to quickly provide working solutions to problems used to instruct beginners, but are significantly less successful at providing solutions to nontrivial problems without significant guidance from the programmer. As you progress through this textbook you may find problems that you consider easy to solve. That is excellent! However, be disciplined. Do not casually dismiss the material. Take advantage of easy problems to develop expertise in program design and in effectively using a coding assistant.

Finally, you need to be aware that coding assistants are nondeterministic. This means that the evaluation of a prompt may not always produce the same result. In practical terms, this means that when you evaluate the examples in the textbook on your computer, you may get very different results. This does not mean that you made a mistake nor does it mean that the textbook is wrong. It is simply the nature of LLMs. LLMs are constantly learning and this means that they may provide different answers to the same prompt. Think of it as a raffle: the same number is not always pulled out of the bag. When this occurs with examples in the textbook, continue reading the development in the textbook and try to understand the design process. Once you do this, take advantage of the different result obtained on your computer and try to continue to solve the problem using the techniques outlined in what you have read.

To Instructors

This textbook is intended as an introduction to prompt engineering for students that already have some programming experience. At a minimum, this means students that have successively traversed CS1 and CS2 or have equivalent experience. The textbook is not recommended for beginning students, because, to use a coding assistant, students must be able to read, analyze, and critique code. They must also understand the basic constructs of programming such as conditional expressions, forms of iteration, mutation, and classes. This is enough of a reason for not using coding assistants in a course for beginners. There are, however, other important reasons. Chief among them is the impairment of cognitive abilities. Coding assistants have become very good at providing solutions to problems used to instruct beginners with very little guidance from the programmer. This means that a beginner can type as a prompt *Write a function to compute n!* and the coding assistant will produce correct code. This does not help the student at all. The student has not thought about the problem, has not developed an understanding about how to solve the problem, and it is extremely likely that the student does not understand how the proposed solution works. As instructors, it is our responsibility to protect them from the dangers of a technology that can inhibit their cognitive development.

If you are an instructor that is unfamiliar with the program-by-design methodology, the use of this book likely means that many of the ideas you

instinctively use to program are made explicit. Be aware that this textbook is not like the textbooks from yesteryears that defined programming as learning the syntax of a programming language with callous disregard for problem solving and program design. Similarly, this textbook is not like the other `AI`-based programming textbooks on the market, at the time of writing, that focus on describing the features available in the interface of a coding assistant with callous disregard for problem solving and how to design prompts. This textbook is the first to integrate coding assistants into the program-by-design methodology. The good news is that with steady consistent work, the material ought to begin to flow smoothly. Keep in mind that some of the steps of the design recipes studied are later integrated into a design recipe from prompts. My advice is to always focus on design and less on getting code to run. By focusing on design during the development and the debugging process, correct running code will follow.

In the later chapters, the textbook tackles some nontrivial problems to teach students how to provide guidance to coding assistants and how to use a coding assistant as a colleague. These are the problems that will provide long-lasting lessons for students. These problems require students to bring all the program design, mathematical, and coding skills they have accumulated to bear on solving a problem. Be aware that solving these problems cannot and should not be done in 30 minutes. Some may take a few lectures. Make sure to bring your students along for the ride and do not rush this material. Keep in mind that effectively using coding assistants is as difficult as any programming task.

Finally, teaching a course about prompt engineering is inherently more difficult than teaching a traditional programming course. The reason is that coding assistants are nondeterministic. You cannot count on a coding assistant producing the same results found on the pages of this textbook. Some instructors opt to use slides and do not use live coding with a coding assistant in the classroom. Others, opt for live coding and explain that the results appearing in the textbook are different. You ought to use the technique you feel most comfortable with and as your expertise increases revisit your decision.

The Programming Language and IDE

The design techniques outlined in this textbook are both programming language and coding assistant agnostic. That is, the design recipes presented do not target a specific programming language nor do they target a specific coding assistant. They may be effectively applied using any programming language and any coding assistant.

That said, of course, a textbook best serves its readers by consistently using a single programming language and a single coding assistant. The question then becomes whether or not there is a programming language and IDE that makes the delivery of the material easier. To this end, an IDE that integrates a programming language and a coding assistant is likely the best choice. The IDE chosen is VS Code, because it seamlessly integrates GitHub Copilot.

The next decision is the programming language for instruction. The use of a functional programming language was seriously considered, because it is undeniable that it is the best medium for instructing beginners. However, there are a significant number of excellent textbooks that present program design using a functional programming language (see the textbooks outlined above). This means that it is time for a textbook that uses a different type of programming language for instruction in conjunction with the program-by-design methodology. To maximize the number of students and instructors that may be reached, the chosen language is Python. Python is already popular among CS instructors and its use makes it easier for instructors to adopt this textbook.

The Parts of the Book

The book is divided into two parts. The first part introduces the program design methodology using Python as the language for instruction. This part does not make direct use of a coding assistant. Readers would be well-advised to put GitHub Copilot on snooze as they work through this part of the textbook. This part covers the basics of Python syntax and its built-in types as well as the basics of using VS Code. In then proceeds into how to design solutions to problems processing compound data of finite size represented as objects. Following this, solving problems using compound data of arbitrary size (e.g., lists, ranges, natural numbers, and

binary trees) by exploiting the structure of the data is presented. This is followed with how to design functions that generate new instances of a given problem (that are hopefully easier to solve) and that use accumulators. The use of accumulators naturally leads to imperative programming. Before that, however, generic programming is discussed. This part ends with a chapter discussing the design of imperative programs (i.e., programs that use mutation). It introduces students to the use of loops and the concept of state. Throughout, students will be subtly learn how to write prompts, but will not evaluate prompts until the second part.

The second part of the textbook is dedicated to prompt engineering and the use of coding assistants. First, the advantages and disadvantages of using coding assistants are discussed. This discussion includes some fairly important ethical issues that are effectively addressed by requiring that prompts be designed and included as documentation for the software developed. One of the goals is to make it easier for others to the understand the solution to the problem. The following chapters introduce prompt design. At first, small simple problems that may not be familiar to most readers are tackled. The final chapters address prompt design for exploiting the structure of the data, for generating new instances of the same problem, for using accumulators, and for sequencing assignment statements. Each of these chapters addresses at least one nontrivial problem such as approximating π using the Chudnovsky algorithm from Mathematics, the partial digest problem from Computational Biology, the parsing and evaluation of arithmetic expressions from Programming Languages, and radix sorting to illustrate the difficulties encountered when designing in-place mutators using a coding assistant.

Acknowledgements

This textbook is the direct result of decades worth work done to develop and advance the program-by-design methodology to help students learn how to program. Without a doubt, this textbook would have never existed without the seminal work done by Matthias Felleisen, his then Ph.D. students (Robby Findler, Matthew Flatt, and Shriram Krishnamurthi), and the rest of Matthias' group lovingly known as `PLT`. It is an honor, my friends, to build my work on your shoulders, and I trust that the results honor you.

I must also extend a heart-felt thank you to my students at Seton Hall University for never failing to inspire me to do better. I clearly remember

when a large group of nervous students walked up to me and asked if I could teach a course on how to program using AI and a coding assistant. At first, I felt that was way too outside my wheelhouse. Nonetheless, I looked into it thinking I would point them to good sources. I quickly found out I was unable to do so and the seed for this textbook was planted. The result is a textbook that is significantly different, at the time of writing, from any other published textbook that addresses AI-based programming. One of my research assistants commented that I was crazy, because I was trying to bring order to a world that was chaotic by design (no pun intended). I leave it to the readers to judge how well I did.

Finally, I must thank my current research assistants, Andrés M. Garced, Sophia G. Turano, and David Anthony K. Fields, for their honest feedback on my ideas and their healthy skepticism about coding assistants. It kept me true to my mission and brought about my realization that prompts are programs and, therefore, need to be designed and validated. Furthermore, they helped me understand, once again, the old adage that states *there is no such thing as a free lunch*. The undisciplined use of coding assistants can quickly lead to cognitive atrophy and, in my opinion, that is not a free lunch. It is a very expensive lunch that is not worth the price. On the other hand, when a disciplined design-based use of coding assistants is pursued, the results can be quite enlightening and sharpen problem-solving skills.

Part I

Introduction to Python and Program Design

Chapter 1

Python Syntax and Built-In Data Types

Python is a general purpose programming language. That is, it is a programming language that is used to solve many different types of problems. Python is used every day, for example, to solve scientific, machine learning, data science, web development, prototyping, and video game development problems. Although used by programming professionals, it is popular in Computer Science education given that it is considered friendly to beginners. This means that it is considered easy to read and understand. In addition, there are many Python libraries available that expand its expressive power and, therefore, make it easier to design and implement programs.

Like any programming language, Python has syntax rules that need to be followed to write valid programs. The good news in that Python syntax aims to be simple and readable. Understanding its syntax is important, because it will allow you to understand the solution to a problem a program expresses. As a general principle, Python statements are executed in the order if which they are written. For instance, consider the following code snippet:

```
1        five = 5
2        six  = 6
```

There are two statements. The first (on line 1) mutates (or changes) the value of the variable five to 5. The second (on line 2) mutates the value

of the variable six to 6. The order in which the statements are written is important. First five is mutated and then, after five has been mutated, six is mutated. The order in which you sequence (or write) statements is important to understand the purpose of a program.

Unlike most programming languages that use parenthesis or curly braces to define code blocks, Python uses indentation to define code blocks. Therefore, code indentation is a detail you must be careful about to properly write and understand programs. For instance consider the code snippet:

```
1        if a > b:
2            max = a
3            min = b
```

On line 1, the value of a is compared with the value of b. The indentation indicates that lines 2 and 3 belong to the same code block that must be executed when the value of a is greater than the value of b. This code block first mutates the value of max to a and then mutates the value of min to b. When a is less than or equal to b, the values of max and min remain unchanged because the code block consisting of lines 2 and 3 is never executed. In contrast, consider the following code snippet:

```
1        if a > b:
2            max = a
3        min = b
```

Here, on line 1, the value of a is also compared with the value of b. Observe that the indentation of the statements that mutate max and min is different. This means they belong to different code blocks. The code block that must be executed when the value of a is greater than the value of b only mutates the value of max to a. Given its indentation, the statement on line 3 belongs to the same code block as the if-statement on line 1. The value of min, therefore, is always mutated to b after the if-statement is executed. Finally, consider the following code snippet:

```
1        if a > b:
2            max = a
3          min = b
```

This is not valid Python code. Python throws an error, because the indentation of the statement that mutates min does not match any previous

indentation. Python does not know when to line 3, because it does not belong to the same code block as line 2 nor does it belong to the same code block as line 1.

This chapter introduces the basic Python syntax rules. In addition, it introduces several built-in data types. The goal is to enable you to start writing and reading Python programs. Once these basic principles are understood, the rest of the chapters in this part of the book introduce fundamental design techniques for problem solving and program development.

1.1 Variables and Function Names

Variable and function names in Python have strict syntax rules that are outlined as follows:

- May contain letters (i.e., a–z and A–Z), numbers (i.e., 0–9), and underscores (i.e., _)
- May start with a letter or a underscore
- May not start with a number
- May not be a Python reserved word (e.g., `if`, `elif`, `while`, `in`, `def`, `import`, and `class`).

The following are examples of valid variable and function names:

```
length          height          interest_rate1
move_point_on_x      move_point_on_y
calculate_monthly_payment
```

In order to foster code readability and make it easier for others (and, in fact, yourself) to understand problem solutions, it is important use meaningful variable and function names. That is, such names ought to convey to the reader the purpose of the variable or the function. For instance, `length` is better than `l`, `height` is better than `h`, `move_point_on_x` is better than `mpox`, and `calculate_monthly_payment` is better than `cmp`. Admittedly, using meaningful names requires more typing, but that is a small price to pay for the gains in readability and in code maintenance. Imagine, for example, that you have written software to process loan applications that includes a function to calculate the loan's monthly payment. Which is a better name for

such a function: `calculate_monthly_payment` or `cmp`. Which of these names will make it easier for you or others to understand your code 6 months from now?

1.2 Built-in Single-Value Data Types

`Python` has several built-in data types. In this context, built-in means that `Python` knows about these types of data and offers functions to manipulate such data. Therefore, you may use built-in data types and the functions defined on them to design and write programs. The simplest data types represent data that has a single value. As we shall see later, such types may be used as building blocks to define compound data (i.e., data that has more than a single value). This section outlines some of `Python`'s single-vaue data types.

1.2.1 Numeric Types

As the name suggests, numeric types represent numbers. There is a variety of numbers in `Python` including:

int Represents the integers (e.g., 100 and -31)

float Represents the real numbers (which are called floating-point numbers) and always include a decimal point (e.g., 8.25 and -0.0087)

complex Represent the complex numbers, which have a real and an imaginary part (e.g., 24 + 4j and 7 - 2j)

`Python` includes several binary functions on numbers including all the arithmetic operations. To apply such functions, infix notation is used (i.e., the function goes between its two inputs). Some of the built-in operations are:

+ Adds two numbers. For example, 5 + 3 evaluates to 8.

- Subtracts two numbers. For example, 5 - 10 evaluates to -5.

***** Multiplies two numbers. For example, -7 * -10 evaluates to 70.