Preface

Welcome to the second edition! This is a book that is intended to be used to teach programming to introductory students. There is material here for intro CS, but also for Science and other disciplines. I still believe that programming is an essential skill for all professionals and especially academics in the 21st century and I have tried to make that clear in the contents of this book.

There are two new chapters and some seriously revised ones. First, the book exclusively uses the Pygame library. The Glib module has been updated but is no longer used in this book. This means that Chapters 7, 9, and 12 are quite different from those in the previous edition. Also, Pygame no longer supports video, so rather than build a new module from scratch, video is not discussed.

The new Chapter 14 concerns *parsing*. This can be a more advanced topic, but parsing is a good thing to know about for many reasons, not the least of which is to deal with user input effectively. The main example is a programming language for which a parser (and compiler) will be written. The language was developed for this book and is called *PyJ*: it is a small subset of the *Julia* language, which in turn is a variation on Python designed for efficiency.

The new Chapter 15 involves graphical input. Here a paint-type program will be developed, so as to clarify ideas in mouse input and graphical output. The resulting program (*Mondrean*) is actually usable for making drawings.

I use a "just-in-time" approach, meaning that I try to present new information just before or just after the reader needs it. As a result, there are a lot of examples,

and those examples were carefully selected to fit into the place they reside in the text. Not too soon, and not too late.

I believe in object-oriented programming. My master's thesis in the late 1970s was on that subject, and I cut my teeth on Simula, was there when C++ was created, and knew the creator of Java. I do not believe that object-oriented programming is the only solution, though, and realized early that good objects can only be devised by someone who can already program. I am therefore not an "objects first" teacher. I am a "whatever works best" teacher.

A lot of my examples involve games. That's because undergraduate students play games. They understand them better than, say, accounting or inventory systems, which have been typical early assignments. I believe in presenting students' assignments that are interesting. Not all students like games, and certainly not computer games, but a large number do. And they come to a game assignment with prior knowledge of the genre.

I have taught computer science for 26 years, and then moved to the arts. That's because of many things, but my experience teaching in a Drama department and more recently in the Art department has helped me immensely in understanding the role of computing and programming in general. I strongly feel that every student in a university should know how to write, and know how to program a computer. If you can't understand the computer, you are at the whim of programmers who, unseen in downtown high-rises and basements, who dictate how the world will work by default. The (sometimes poor) design decisions made, and the lack of attention paid to human needs results in actual policy being formed, and that is simply wrong. It's not always true that the code is bad, but when it is, it can have far reaching consequences.

Here is a truth: *nobody wants to run your program*. What they want is to get their work done, or play their game, or send their email. If you are an excellent programmer then you will enable that, and nobody will know your name. But nobody will curse your code either. The truth is that good code is invisible. It simply allows things to flow smoothly. Bad code is memorable. It interferes, makes people frustrated and angry. If you believe in karma, then I know what you would prefer.

You see, software (any computer program) is ubiquitous. Cars, phones, fridges, television, and almost everything in our society is computerized. Decisions

made about how a program is to be built tend to live on, and even after many modifications can affect how people use that device or system. Creating good software means making a productive and happy civilization. It sounds trite, but if you think about it I'm sure you will agree.

Python is a great language for beginning programmers. It is easy to write the first programs, because the conceptual overhead is small. That is, there's no need to understand what 'void' or 'public' means at the outset. Python does a lot of things for a programmer. Do you want something sorted? It's a part of the language. Lists and hash tables (dictionaries) are a part of the language. You can write classes, but do not have to, so it can be taught *objects first* or not. The required indentation means that it is much harder to place code incorrectly in loops or if statements. There are hundreds of reasons why Python is a great idea.

And it is free. This book was written using Python version 3.4, and with the *PyCharm* API. The modules used that require download are few, but include PyGame and tweepy. All free.

Overview of Chapters

Here's a breakdown of the book, for instructors. It can be used to teach computer science majors or science students who wish to have a competency in programming.

Chapter 0: Historical and technological material on computers. Binary numbers, the fetch-execute cycle. This chapter can be skipped in some syllabi.

Chapter 1: Problem solving with a computer; breaking a problem down so it can be solved. The Python system. Some simple programs involving games that introduce variables, expressions, print, types, and the **if** statement.

Chapter 2: Repetition in programming: **while** and **for** statements. Random numbers. Counting loops, nested loops. *Drawing a histogram*. Exceptions (**try-except**)

Chapter 3: Strings and string operations. Tuples, their definition, and use. Lists and list comprehension. Editing, slices. The *bytes* type. And set types. Example: the game of *craps*.

Chapter 4: Functions: modular programming. Defining a function, calling a function. Parameters, including default parameters, and scope. Return values.

Recursion. *The Game of Sticks*. Variable parameter lists, assigning a function to a variable. Find the maximum of a mathematical function. Modules. *Game of Nim*.

Chapter 5: Files. What is a file and how are they represented? Properties of files. File exceptions. Input, output, append, **open**, **close**. Comma separated value (CSV) files. Game of *Jeopardy*. The **with** statement.

Chapter 6: Classes and object orientation. What is an object and what is a class? Types and classes. Python class structure. Creating instances, **__init__** and **self**. Encapsulation. Examples: *deck of playing cards*; a *bouncing ball*; *Cat-a-pult*. Designing with classes. Subclasses and inheritance. Video game objects. Duck typing.

Chapter 7: Graphics. The *Pygame* module. Drawing window; color representation, pixels. Drawing lines, curves, and polygons. Filling. Drawing text. Example: *Histogram*, *Pie chart*. Images and image display, getting and setting pixels. *Thresholding*. Generative art.

Chapter 8: Data and information. Python dictionaries. *Latin to English translator*. Arrays, formatted text, formatted input/output. *Meteorite landing data*. Non-text files and the *struct* module. *High score file* example. Random access. Image and sound file types.

Chapter 9: Digital media: Using the mouse and the keyboard. Animation. *Space shuttle control console* example. Transparent colors. Sound: playing sound files, volume, pause. Pygame module for sound.

Chapter 10: Basic algorithms in computer science. Sorting (selection, merge) and searching (linear, binary). Timing code execution. Generating random numbers; cryptography; data compression (including Huffman codes and RLE); hashing.

Chapter 11: Programming for Science. Roots of equations; differentiation and integration. Optimization (minimum and maximum) and curve fitting (regression). Evolutionary algorithms. Longest common subsequence or edit distance.

Chapter 12: Writing *good* code. A walk through two major projects: a word processor written as procedural code and a *breakout* game written as object-oriented code. A collection of effective rules for writing good code.

Chapter 13: Dealing with real world interfaces, which tend to be defined for you. Examples are Email (send and receive), FTP, inter-process communication (client-server), Twitter, calling other languages like C++.

Chapter 14: Parsing. Introduction to grammars and BNF. Parsing data. A small compiler for a small language.

Chapter 15: Graphical Interaction. Using the mouse in complicated ways. Drawing, erasing, modifying images.

Chapter Coverage for Different Majors

A **computer science** introduction could use most chapters, depending on the background of the students, but Chapters 0, 7, 9, and / or 11 could be omitted.

An **introduction to programming for science** could omit Chapters 0, 10, and 12.

Chapter 13 is always optional, but is interesting as it explains how social media software works under the interface.

Basic **introduction to programming for non-science** should include Chapters 0, 1, 2, 3, 4, 5, and 7.

Companion Files *(A disc is included in the physical book or files are available for downloading from the publisher by writing to info@merclearning.com.)*

The accompanying disc contains useful material for each chapter.

- Selected exercises are solved, including working code when that is a part of the solution.
- All significant examples are provided as Python code files, which can be compiled and executed, and can be modified as exercises or class projects. This includes sample data files when appropriate.
- All figures are available as images, in full color.

Instructor Ancillaries

- Solutions to almost all of the programming exercises given in the text.
- MS PowerPoint *lectures* provided for an entire semester (35 files) including some new examples and short videos.

- All of the Python code that appears in the books has been executed, and all complete programs are provided as .py files. Some of the numerous programming examples (over 100) that are explored in the book and for which working code is included:
  - ***An interactive breakout game***
  - ***The Game of Nim***
  - ***A text formatting system***
  - ***Plotting histograms and pie charts***
  - ***Reading Twitter feeds***
  - ***Play*** Jeopardy ***Using a CSV Data Set***
  - ***Sending and receiving Email***
  - ***A simple Latin to English translator***
  - ***Cryptography***
  - ***Rock-Paper-Scissors***
- Hundreds of answered multiple choice quiz and sample *examination questions* in MS Word files that can be edited and used in various ways.

Dedicated Website

Please consider contributing material to the on-line community at *https://sites.google.com/site/pythonparker/* and do have fun. If you don't then you're doing it wrong.

J. Parker
February 2021