

Editors

Timothy J. Barth
Michael Griebel
David E. Keyes
Risto M. Nieminen
Dirk Roose
Tamar Schlick

More information about this series at <http://www.springer.com/series/5151>

Svein Linge • Hans Petter Langtangen

Programming for Computations - Python

A Gentle Introduction to Numerical
Simulations with Python 3.6

Second Edition

 Springer Open

Svein Linge
Fac of Tech, Natural Sci & Maritime Sci
University of South-Eastern Norway
Porsgrunn, Norway

Hans Petter Langtangen
Simula Research Laboratory BioComp
Lysaker, Norway



ISSN 1611-0994 ISSN 2197-179X (electronic)
Texts in Computational Science and Engineering
ISBN 978-3-030-16876-6 ISBN 978-3-030-16877-3 (eBook)
<https://doi.org/10.1007/978-3-030-16877-3>

Mathematics Subject Classification (2010): 26-01, 34A05, 34A30, 34A34, 39-01, 40-01, 65D15, 65D25, 65D30, 68-01, 68N01, 68N19, 68N30, 70-01, 92D25, 97-04, 97U50

This book is an open access publication.

© The Editor(s) (if applicable) and The Author(s) 2020

Open Access This book is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this book are included in the book's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the book's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To My Family

Thanks to my dear wife, Katrin, and our lovely children, Stian, Mia, and Magnus, for their love, support, and patience. I am a very lucky man.

To Hans Petter

Dear friend and coauthor, it is so sad you are no longer among us.¹ Thank you for everything. I dedicate this second edition of our book to you.

Porsgrunn, Norway
June 2018

Svein Linge

¹ Professor Hans Petter Langtangen (https://en.wikipedia.org/wiki/Hans_Petter_Langtangen) passed away with cancer on the 10th of October, 2016.

Preface

Computing, in the sense of doing mathematical calculations, is a skill that mankind has developed over thousands of years. Programming, on the other hand, is in its infancy, with a history that spans a few decades only. Both topics are vastly comprehensive and usually taught as separate subjects in educational institutions around the world, especially at the undergraduate level. This book is about the *combination* of the two, because computing today becomes so much more powerful when combined with programming.

Most universities and colleges implicitly require students to specialize in computer science if they want to learn the craft of programming, since other student programs usually do not offer programming to an extent demanded for really mastering this craft. Common arguments claim that it is sufficient with a brief introduction, that there is not enough room for learning programming in addition to all other must-have subjects, and that there is so much software available that few really need to program themselves. A consequence is that engineering students often graduate with shallow knowledge about programming, unless they happened to choose the computer science direction.

We think this is an unfortunate situation. There is no doubt that practicing engineers and scientists need to know their pen-and-paper mathematics. They must also be able to run off-the-shelf software for important standard tasks and will certainly do that a lot. Nevertheless, the benefits of mastering programming are many.

Why Learn Programming?

1. Ready-made software is limited to handling certain standard problems. What do you do when the problem at hand is not covered by the software you bought? Fortunately, a lot of modern software systems are extensible via programming. In fact, many systems demand parts of the problem specification (e.g., material models) to be specified by computer code.
2. With programming skills, you may extend the flexibility of existing software packages by combining them. For example, you may integrate packages that do not speak to each other from the outset. This makes the work flow simpler, more efficient, and more reliable, and it puts you in a position to attack new problems.

3. It is easy to use excellent ready-made software the wrong way. The insight in programming and the mathematics behind is fundamental for understanding complex software, avoiding pitfalls, and becoming a safe user.
4. Bugs (errors in computer code) are present in most larger computer programs (also in the ones from the shop!). What do you do when your ready-made software gives unexpected results? Is it a bug, is it the wrong use, or is it the mathematically correct result? Experience with programming of mathematics gives you a good background for answering these questions. The one who can program can also make tailored code for a simplified problem setting and use that to verify the computations done with off-the-shelf software.
5. Lots of skilled people around the world solve computational problems by writing their own code and offering those for free on the Internet. To take advantage of this truly great source of software in a reliable way, one must normally be able to understand and possibly modify computer code offered by others.
6. It is recognized worldwide that students struggle with mathematics and physics. Too many find such subjects difficult and boring. With programming, we can execute the good old subjects in a brand new way! According to the authors' own experience, students find it much more motivating and enlightening when programming is made an integrated part of mathematics and physical science courses. In particular, the problem being solved can be much more realistic than when the mathematics is restricted to what you can do with pen and paper.
7. Finally, we launch our most important argument for learning computer programming: the *algorithmic thinking* that comes with the process of writing a program for a computational problem enforces a thorough understanding of both the problem and the solution method. We can simply quote the famous Norwegian computer scientist Kristen Nygaard: "Programming is understanding."

In the authors' experience, programming is an excellent pedagogical tool for understanding mathematics: "You think you know when you can learn, are more sure when you can write, even more when you can teach, but certain when you can program" (Alan Perlis, computer scientist, 1922–1990). Consider, for example, integration. A numerical method for integration has a much stronger focus on what the integral actually is and means compared to analytical methods, where much time and effort must be devoted to integration by parts, integration by substitution, etc. Moreover, when programming the numerical integration formula, it becomes evident that it works for "all" mathematical functions and that the implementation should be in terms of a *general* function applicable to "all" integrals. In this way, students learn to recognize a special problem as belonging to a class of problems (e.g., integration, differential equations, root finding), for which we have general numerical methods implemented in a widely applicable software. When they write this software, as we do in this book, they learn how to generalize and increase the abstraction level of the mathematical problem. When they use this software, they learn how a special case should be attacked by general methods and software for the class of problems that comprises the special case at hand. This is the power of mathematics in a nutshell, and it is paramount that students understand this way of thinking.

Target Audience and Background Knowledge This book was written for students, teachers, engineers, and scientists who know *nothing* about programming and numerical methods from before but who seek a *minimum* of the fundamental skills required to get started with programming as a tool for solving scientific and engineering problems. Some knowledge of one- and multivariable calculus is assumed. The basic programming concepts are presented in Chaps. 1–5 (about 150 pages), before practical applications of these concepts are demonstrated in important mathematical subjects addressed in the remaining parts of the book (Chaps. 6–9). Each chapter is followed by a set of exercises that covers a wide range of application areas, e.g., biology, geology, statistics, physics, and mathematics. The exercises were particularly designed to bring across important points from the text.

Learning the very basics of programming should not take long, but as with any other craft, mastering the skill requires continued and extensive practice. Some beginning practice is gained through Chaps. 6–9, but the authors strongly emphasize that this is only a start. Students should continue to practice programming in subsequent courses, while those who exercise self-study should keep up the learning process through continued application of the craft. The book is a good starting point when teaching computer programming as an integrated part of standard university courses in mathematics and natural science. In our experience, such an integration is doable and indeed rewarding.

Numerical Methods An overall goal with this book is to motivate computer programming as a very powerful tool for doing mathematics. All examples are related to mathematics and its use in engineering and science. However, to solve mathematical problems through computer programming, we need numerical methods. Explaining basic numerical methods is therefore an integral part of the book. Our choice of topics is governed by what is most needed in science and engineering, as well as in the teaching of applied natural science courses. Mathematical models are then central, with differential equations constituting the most frequent type of models. Consequently, the numerical focus in this book is on differential equations. As soft pedagogical starters for the programming of mathematics, we have chosen the topics of numerical integration and root finding. We remark that the book is deliberately brief on numerical methods. This is because our focus is on *implementing* numerical algorithms, and to develop reliable, working programs, the programmer must be confident about the basic ideas of the numerical approximations involved.

The Computer Language: Python We have chosen to use the programming language Python, because this language gives a very compact and readable code that closely resembles the mathematical recipe for solving the problem at hand. Python also has a gentle learning curve.

Other computer languages, like Fortran, C, and C++, have a strong position in science and engineering. During the last two decades, however, there has been a significant shift in popularity from these compiled languages to more high-level and easier-to-read languages, for instance, MATLAB, Python, R, Maple, Mathematica, and IDL. This latter class of languages is computationally less efficient but superior with respect to overall human problem-solving efficiency. This book emphasizes

how to think like a programmer, rather than focusing on technical language details. Thus, the book should put the reader in a good position for learning other programming languages later, including the classic ones: Fortran, C, and C++.

How This Book Is Different There are numerous texts on computer programming and numerical methods, so how does the present one differ from the existing literature? Compared to standard books on numerical methods, our book has a much stronger emphasis on the craft of programming and on verification. We want to give students a thorough understanding of how to think about programming as a problem-solving method and how to provide convincing evidence for program correctness.

Even though there are lots of books on numerical methods where many algorithms have a corresponding computer implementation (see, e.g., [1, 3–6, 10, 15–17, 20, 23, 25, 27–31]—the latter two apply Python), it is often assumed that the reader “can program” beforehand. The present book teaches the craft of structured programming along with the fundamental ideas of numerical methods. In this book, unit testing and corresponding test functions are introduced early on. We also put much emphasis on coding algorithms as *functions*, as opposed to “flat programs,” which often dominate in the literature and among practitioners. Functions are reusable because they utilize the general formulation of a mathematical algorithm such that it becomes applicable to a large class of problems.

There are also numerous books on computer programming, but not many that really emphasize how to *think* about programming in the context of numerical methods and scientific applications. One such book is [11], which gives a comprehensive introduction to Python programming and the thinking about programming as a computer scientist.

Sometimes, however, one needs a text like the present one. It does not go so deep into language-specific details, but rather targets the shortest path to reliable mathematical problem-solving through programming. With this attitude in mind, a lot of topics were left out of the present book, simply because they were not *strictly* needed in the mathematical problem-solving process. Examples of such topics are object-oriented programming and Python dictionaries (of which the latter omission is possibly subject to more debate). If you find the present book too shallow, [11] might be the right choice for you. That source should also work nicely as a more in-depth successor of the present text.

Whenever the need for a *structured introduction to programming* arises in science and engineering courses, the present book may be your option, either for self-study or for use in organized teaching. The thinking, habits, and practice covered herein will put readers in a firm position for utilizing and understanding the power of computers for problem-solving in science and engineering.

Changes to the First Edition

1. All code is now in Python version 3.6 (the previous edition was based on Python version 2.7).
2. In the first edition, the introduction to programming was basically covered in 50 pages by Chap. 1 (*The First Few Steps*) and Chap. 2 (*Basic Constructions*). This *is* enough to get going, but many readers soon want more details. In this second edition, these two chapters have therefore been extended and split up into five

chapters. Explanations are now more complete, previous examples have been modified, new examples have been added, and more. In particular, the importing of code is now elaborated on in a greater detail, so is the making of modules. Also, Sect. 4.2 is new, illustrating the important stepwise strategy of code writing through a dedicated example. The five first chapters now cover about 150 pages that explain, in a brief and simple manner, all the code basics required to follow the remaining parts of the book.

3. The new Chap. 6 (*Computing Integrals and Testing Code*) and Chap. 7 (*Solving Nonlinear Algebraic Equations*) are seen as gentle first applications of programming to problem-solving in mathematics. Both these chapters now precede the mathematically more challenging Chaps. 8 and 9, which treat basic numerical solving of ODEs and PDEs, respectively (the chapter *Solving Nonlinear Algebraic Equations* was, in the first edition, the final chapter of the book, but it seems more appropriate to let it act as a “warm-up” chapter, together with the new Chap. 6, for the two final chapters on differential equation solving).
4. Section 8.1 (*Filling a Water Tank: Two Cases*) is new, particularly written for readers who lack experience with differential equations.
5. Section 8.5 (*Rate of Convergence*) is new, explaining convergence rate related to differential equations.
6. New exercises have been added, e.g., on Fibonacci numbers, the Leapfrog method, Adams-Bashforth methods, and more.
7. Errors and typos have been corrected, and many explanations have been reformulated throughout.

Supplementary Materials All program and data files referred to herein are available from the book’s (2nd edition) primary web site:

https://github.com/slgit/prog4comp_2.

Acknowledgments We want to thank all students who attended the courses *FM1006 Modelling and simulation of dynamic systems*, *FM1115 Scientific Computing*, *FB1012 Mathematics*, and *FB2112 Physics* at the University of South-Eastern Norway over the last 5–10 years. They worked their way through early versions of this text and gave us constructive and positive feedback that helped us correct errors and improve the book in so many ways. Special acknowledgment goes to Guandong Kou, Edirisinghe V. P. J. Manjula, and Yapi Donatien Achou for their careful reading of the manuscript (first edition) and constructive suggestions for improvement. The constructive feedback and good suggestions received from Om Prakash Chapagain (second edition) is also highly appreciated. We thank all our good colleagues at the University of South-Eastern Norway, the University of Oslo, and Simula Research Laboratory for their continued support and interest, for the enlightening discussions, and for providing such an inspiring environment for teaching and science.

Special thanks go to Prof. Kent-Andre Mardal, the University of Oslo, for his insightful comments and suggestions.

The authors would also like to thank the Springer team with Dr. Martin Peters, Thanh-Ha Le Thi, and Leonie Kunz for the effective editorial and production process.

This text was written in the [DocOnce](#)¹ [12] markup language.

Lysaker, Norway
December 2015
Porsgrunn, Norway
June 2018

Hans Petter Langtangen

Svein Linge

¹ <https://github.com/hplgit/doconce>.

Abstract

This second edition of the book presents computer programming as a key method for solving mathematical problems and represents a major revision: all code is now written in Python version 3.6 (the first edition was based on Python version 2.7). The first two chapters of the previous edition have been extended and split up into five new chapters, thus expanding the introduction to programming from 50 to 150 pages. Throughout, explanations are now more complete, previous examples have been modified, and new sections, examples, and exercises have been added. Also, errors and typos have been corrected. The book was inspired by the Springer book TCSE 6, *A Primer on Scientific Programming with Python* (by Langtangen), but the style is more accessible and concise in keeping with the needs of engineering students. The book outlines the shortest possible path from no previous experience with programming to a set of skills that allows the students to write simple programs for solving common mathematical problems with numerical methods in engineering and science courses. The emphasis is on generic algorithms, clean design of programs, use of functions, and automatic tests for verification.

Contents

1	The First Few Steps	1
1.1	What Is a Program? And What Is Programming?	1
1.1.1	Installing Python	4
1.2	A Python Program with Variables	5
1.2.1	The Program	5
1.2.2	Dissecting the Program	6
1.2.3	Why Use Variables?	9
1.2.4	Mathematical Notation Versus Coding	10
1.2.5	Write and Run Your First Program	10
1.3	A Python Program with a Library Function	12
1.4	Importing from Modules and Packages	14
1.4.1	Importing for Use <i>Without</i> Prefix	14
1.4.2	Importing for Use <i>with</i> Prefix	17
1.4.3	Imports with Name Change	18
1.4.4	Importing from Packages	18
1.4.5	The Modules/Packages Used in This Book	19
1.5	A Python Program with Vectorization and Plotting	19
1.6	Plotting, Printing and Input Data	21
1.6.1	Plotting with Matplotlib	21
1.6.2	Printing: The String Format Method	27
1.6.3	Printing: The f-String	31
1.6.4	User Input	32
1.7	Error Messages and Warnings	33
1.8	Concluding Remarks	34
1.8.1	Programming Demands You to Be Accurate!	34
1.8.2	Write Readable Code	35
1.8.3	Fast Code or Slower and Readable Code?	35
1.8.4	Deleting Data No Longer in Use	36
1.8.5	Code Lines That Are Too Long	36
1.8.6	Where to Find More Information?	36
1.9	Exercises	37

2	A Few More Steps	39
2.1	Using Python Interactively	39
2.1.1	The IPython Shell	39
2.1.2	Command History	40
2.1.3	TAB Completion	40
2.2	Variables, Objects and Expressions	41
2.2.1	Choose Descriptive Variable Names	41
2.2.2	Reserved Words	41
2.2.3	Assignment	42
2.2.4	Object Type and Type Conversion	42
2.2.5	Automatic Type Conversion	43
2.2.6	Operator Precedence	44
2.2.7	Division—Quotient and Remainder	45
2.2.8	Using Parentheses	45
2.2.9	Round-Off Errors	45
2.2.10	Boolean Expressions	46
2.3	Numerical Python Arrays	47
2.3.1	Array Creation and Array Elements	47
2.3.2	Indexing an Array from the End	50
2.3.3	Index Out of Bounds	50
2.3.4	Copying an Array	50
2.3.5	Slicing an Array	51
2.3.6	Two-Dimensional Arrays and Matrix Computations	52
2.4	Random Numbers	54
2.5	Exercises	56
3	Loops and Branching	59
3.1	The for Loop	59
3.1.1	Example: Printing the 5 Times Table	59
3.1.2	Characteristics of a Typical for Loop	60
3.1.3	Combining for Loop and Array	62
3.1.4	Using the range Function	63
3.1.5	Using break and continue	64
3.2	The while Loop	65
3.2.1	Example: Finding the Time of Flight	65
3.2.2	Characteristics of a Typical while Loop	66
3.3	Branching (if, elif and else)	68
3.3.1	Example: Judging the Water Temperature	68
3.3.2	The Characteristics of Branching	70
3.3.3	Example: Finding the Maximum Height	70
3.3.4	Example: Random Walk in Two Dimensions	72
3.4	Exercises	74
4	Functions and the Writing of Code	79
4.1	Functions: How to Write Them?	79
4.1.1	Example: Writing Our First Function	79
4.1.2	Characteristics of a Function Definition	80
4.1.3	Functions and the Main Program	82
4.1.4	Local Versus Global Variables	83

4.1.5	Calling a Function Defined with Positional Parameters . . .	83
4.1.6	A Function with Two Return Values	85
4.1.7	Calling a Function Defined with Keyword Parameters	85
4.1.8	A Function with Another Function as Input Argument	86
4.1.9	Lambda Functions	87
4.1.10	A Function with Several Return Statements	87
4.2	Programming as a Step-Wise Strategy	88
4.2.1	Making a Times Tables Test	89
4.2.2	The 1st Version of Our Code	90
4.2.3	The 2nd Version of Our Code	91
4.2.4	The 3rd Version of Our Code	93
4.3	Exercises	97
5	Some More Python Essentials	103
5.1	Lists and Tuples: Alternatives to Arrays	103
5.2	Exception Handling	106
5.2.1	The Fourth Version of Our Times Tables Program	106
5.3	Symbolic Computations	111
5.3.1	Numerical Versus Symbolic Computations	111
5.3.2	SymPy: Some Basic Functionality	112
5.3.3	Symbolic Calculations with Some Other Tools	112
5.4	Making Our Own Module	113
5.4.1	A Naive Import	114
5.4.2	A Module for Vertical Motion	115
5.4.3	Module or Program?	119
5.5	Files: Read and Write	122
5.6	Measuring Execution Time	123
5.6.1	The <code>timeit</code> Module	123
5.7	Exercises	125
6	Computing Integrals and Testing Code	131
6.1	Basic Ideas of Numerical Integration	132
6.2	The Composite Trapezoidal Rule	134
6.2.1	The General Formula	135
6.2.2	A General Implementation	136
6.2.3	A Specific Implementation: What’s the Problem?	139
6.3	The Composite Midpoint Method	142
6.3.1	The General Formula	143
6.3.2	A General Implementation	144
6.3.3	Comparing the Trapezoidal and the Midpoint Methods . . .	145
6.4	Vectorizing the Functions	146
6.4.1	Vectorizing the Midpoint Rule	146
6.4.2	Vectorizing the Trapezoidal Rule	147
6.4.3	Speed up Gained with Vectorization	148
6.5	Rate of Convergence	148
6.6	Testing Code	150
6.6.1	Problems with Brief Testing Procedures	150
6.6.2	Proper Test Procedures	151

6.6.3	Finite Precision of Floating-Point Numbers	153
6.6.4	Constructing Unit Tests and Writing Test Functions	155
6.7	Double and Triple Integrals	157
6.7.1	The Midpoint Rule for a Double Integral	157
6.7.2	The Midpoint Rule for a Triple Integral	161
6.7.3	Monte Carlo Integration for Complex-Shaped Domains	163
6.8	Exercises	169
7	Solving Nonlinear Algebraic Equations	175
7.1	Brute Force Methods	176
7.1.1	Brute Force Root Finding	177
7.1.2	Brute Force Optimization	179
7.1.3	Model Problem for Algebraic Equations	180
7.2	Newton's Method	181
7.2.1	Deriving and Implementing Newton's Method	181
7.2.2	Making a More Efficient and Robust Implementation	184
7.3	The Secant Method	188
7.4	The Bisection Method	190
7.5	Rate of Convergence	192
7.6	Solving Multiple Nonlinear Algebraic Equations	195
7.6.1	Abstract Notation	195
7.6.2	Taylor Expansions for Multi-Variable Functions	195
7.6.3	Newton's Method	196
7.6.4	Implementation	197
7.7	Exercises	198
8	Solving Ordinary Differential Equations	203
8.1	Filling a Water Tank: Two Cases	205
8.1.1	Case 1: Piecewise Constant Rate	205
8.1.2	Case 2: Continuously Increasing Rate	207
8.1.3	Reformulating the Problems as ODEs	209
8.2	Population Growth: A First Order ODE	210
8.2.1	Derivation of the Model	211
8.2.2	Numerical Solution: The Forward Euler (FE) Method	213
8.2.3	Programming the FE Scheme; the Special Case	217
8.2.4	Understanding the Forward Euler Method	219
8.2.5	Programming the FE Scheme; the General Case	220
8.2.6	A More Realistic Population Growth Model	221
8.2.7	Verification: Exact Linear Solution of the Discrete Equations	224
8.3	Spreading of Disease: A System of First Order ODEs	225
8.3.1	Spreading of Flu	225
8.3.2	A FE Method for the System of ODEs	228
8.3.3	Programming the FE Scheme; the Special Case	229
8.3.4	Outbreak or Not	230
8.3.5	Abstract Problem and Notation	232
8.3.6	Programming the FE Scheme; the General Case	232
8.3.7	Time-Restricted Immunity	235

8.3.8	Incorporating Vaccination	236
8.3.9	Discontinuous Coefficients: A Vaccination Campaign	237
8.4	Oscillating 1D Systems: A Second Order ODE	239
8.4.1	Derivation of a Simple Model	240
8.4.2	Numerical Solution	241
8.4.3	Programming the FE Scheme; the Special Case	242
8.4.4	A Magic Fix of the Numerical Method	243
8.4.5	The Second-Order Runge-Kutta Method (or Heun’s Method)	246
8.4.6	Software for Solving ODEs	249
8.4.7	The Fourth-Order Runge-Kutta Method	254
8.4.8	More Effects: Damping, Nonlinearity, and External Forces	257
8.4.9	Illustration of Linear Damping	260
8.4.10	Illustration of Linear Damping with Sinusoidal Excitation	262
8.4.11	Spring-Mass System with Sliding Friction	264
8.4.12	A Finite Difference Method; Undamped, Linear Case	266
8.4.13	A Finite Difference Method; Linear Damping	268
8.5	Rate of Convergence	269
8.5.1	Asymptotic Behavior of the Error	270
8.5.2	Computing the Convergence Rate	270
8.5.3	Test Function: Convergence Rates for the FE Solver	272
8.6	Exercises	273
9	Solving Partial Differential Equations	287
9.1	Example: Temperature Development in a Rod	288
9.1.1	A Particular Case	289
9.2	Finite Difference Methods	290
9.2.1	Reduction of a PDE to a System of ODEs	290
9.2.2	Construction of a Test Problem with Known Discrete Solution	293
9.2.3	Implementation: Forward Euler Method	293
9.2.4	Animation: Heat Conduction in a Rod	295
9.2.5	Vectorization	299
9.2.6	Using Odespy to Solve the System of ODEs	299
9.2.7	Implicit Methods	300
9.3	Exercises	303
A	Installation and Use of Python	311
A.1	Recommendation: Install Anaconda and Odespy	311
A.2	Required Software	311
A.3	Anaconda and Spyder	313
A.3.1	Spyder on Mac	313
A.3.2	Installation of Additional Packages	313
A.4	How to Write and Run a Python Program	314
A.4.1	Spyder	314
A.4.2	Text Editors	315

A.4.3	Terminal Windows.....	315
A.4.4	Using a Plain Text Editor and a Terminal Window	315
A.5	Python with Jupyter Notebooks and Web Services.....	316
References	317
Index	319

List of Exercises

Exercise 1.1: Error Messages	37
Exercise 1.2: Volume of a Cube	37
Exercise 1.3: Area and Circumference of a Circle	37
Exercise 1.4: Volumes of Three Cubes	37
Exercise 1.5: Average of Integers	38
Exercise 1.6: Formatted Print to Screen	38
Exercise 2.1: Interactive Computing of Volume	56
Exercise 2.2: Interactive Computing of Circumference and Area	56
Exercise 2.3: Update Variable at Command Prompt	56
Exercise 2.4: Multiple Statements on One Line	56
Exercise 2.5: Boolean Expression—Even or Odd Number?	57
Exercise 2.6: Plotting Array Data	57
Exercise 2.7: Switching Values	57
Exercise 2.8: Drawing Random Numbers	58
Exercise 3.1: A for Loop with Errors	74
Exercise 3.2: The range Function	74
Exercise 3.3: A while Loop with Errors	74
Exercise 3.4: while Loop Instead of for Loop	74
Exercise 3.5: Compare Integers a and b	74
Exercise 3.6: Area of Rectangle Versus Circle	75
Exercise 3.7: Frequency of Random Numbers	75
Exercise 3.8: Game 21	75
Exercise 3.9: Simple Search: Verification	76
Exercise 3.10: Sort Array with Numbers	76
Exercise 3.11: Compute π	76
Exercise 4.1: Errors with Colon, Indent, etc.	97
Exercise 4.2: Reading Code 1	97
Exercise 4.3: Reading Code 2	98
Exercise 4.4: Functions for Circumference and Area of a Circle	98
Exercise 4.5: Function for Adding Vectors	98
Exercise 4.6: Function for Area of a Rectangle	99
Exercise 4.7: Average of Integers	99
Exercise 4.8: When Does Python Check Function Syntax?	99
Exercise 4.9: Find Crossing Points of Two Graphs	99

Exercise 4.10: Linear Interpolation	99
Exercise 4.11: Test Straight Line Requirement	100
Exercise 4.12: Fit Straight Line to Data	100
Exercise 4.13: Fit Sines to Straight Line	101
Exercise 5.1: Nested for Loops and Lists	125
Exercise 5.2: Exception Handling: Divisions in a Loop	125
Exercise 5.3: Taylor Series, sympy and Documentation	125
Exercise 5.4: Fibonacci Numbers	126
Exercise 5.5: Read File: Total Volume of Boxes	127
Exercise 5.6: Area of a Polygon	128
Exercise 5.7: Count Occurrences of a String in a String	128
Exercise 5.8: Compute Combinations of Sets	129
Exercise 6.1: Hand Calculations for the Trapezoidal Method	169
Exercise 6.2: Hand Calculations for the Midpoint Method	169
Exercise 6.3: Compute a Simple Integral	169
Exercise 6.4: Hand-Calculations with Sine Integrals	169
Exercise 6.5: Make Test Functions for the Midpoint Method	169
Exercise 6.6: Explore Rounding Errors with Large Numbers	169
Exercise 6.7: Write Test Functions for $\int_0^4 \sqrt{x} dx$	170
Exercise 6.8: Rectangle Methods	170
Exercise 6.9: Adaptive Integration	171
Exercise 6.10: Integrating x Raised to x	171
Exercise 6.11: Integrate Products of Sine Functions	172
Exercise 6.12: Revisit Fit of Sines to a Function	172
Exercise 6.13: Derive the Trapezoidal Rule for a Double Integral	173
Exercise 6.14: Compute the Area of a Triangle by Monte Carlo Integration	174
Exercise 7.1: Understand Why Newton's Method Can Fail	198
Exercise 7.2: See If the Secant Method Fails	198
Exercise 7.3: Understand Why the Bisection Method Cannot Fail	199
Exercise 7.4: Combine the Bisection Method with Newton's Method	199
Exercise 7.5: Write a Test Function for Newton's Method	199
Exercise 7.6: Halley's Method and the Decimal Module	199
Exercise 7.7: Fixed Point Iteration	200
Exercise 7.8: Solve Nonlinear Equation for a Vibrating Beam	201
Exercise 8.1: Restructure a Given Code	273
Exercise 8.2: Geometric Construction of the Forward Euler Method	273
Exercise 8.3: Make Test Functions for the Forward Euler Method	273
Exercise 8.4: Implement and Evaluate Heun's Method	274
Exercise 8.5: Find an Appropriate Time Step; Logistic Model	274
Exercise 8.6: Find an Appropriate Time Step; SIR Model	274
Exercise 8.7: Model an Adaptive Vaccination Campaign	274
Exercise 8.8: Make a SIRV Model with Time-Limited Effect of Vaccination	275
Exercise 8.9: Refactor a Flat Program	275
Exercise 8.10: Simulate Oscillations by a General ODE Solver	275
Exercise 8.11: Compute the Energy in Oscillations	276

Exercise 8.12: Use a Backward Euler Scheme for Population Growth	276
Exercise 8.13: Use a Crank-Nicolson Scheme for Population Growth	277
Exercise 8.14: Understand Finite Differences via Taylor Series	277
Exercise 8.15: The Leapfrog Method	279
Exercise 8.16: The Runge-Kutta Third Order Method	280
Exercise 8.17: The Two-Step Adams-Bashforth Method	280
Exercise 8.18: The Three-Step Adams-Bashforth Method	282
Exercise 8.19: Use a Backward Euler Scheme for Oscillations	282
Exercise 8.20: Use Heun's Method for the SIR Model	283
Exercise 8.21: Use Odespy to Solve a Simple ODE	283
Exercise 8.22: Set up a Backward Euler Scheme for Oscillations	284
Exercise 8.23: Set up a Forward Euler Scheme for Nonlinear and Damped Oscillations	284
Exercise 8.24: Solving a Nonlinear ODE with Backward Euler	285
Exercise 8.25: Discretize an Initial Condition	285
Exercise 9.1: Simulate a Diffusion Equation by Hand	303
Exercise 9.2: Compute Temperature Variations in the Ground	303
Exercise 9.3: Compare Implicit Methods	304
Exercise 9.4: Explore Adaptive and Implicit Methods	305
Exercise 9.5: Investigate the θ Rule	305
Exercise 9.6: Compute the Diffusion of a Gaussian Peak	306
Exercise 9.7: Vectorize a Function for Computing the Area of a Polygon	307
Exercise 9.8: Explore Symmetry	307
Exercise 9.9: Compute Solutions as $t \rightarrow \infty$	308
Exercise 9.10: Solve a Two-Point Boundary Value Problem	309