A\_Primer\_Scientific\_Programming\_Python\_5E\_Hans\_c01

Hans Petter Langtangen A Primer on Scientific Programming with Python 5th Edition

p32

**Preface**

The aim of this book is to teach computer programming using examples from mathematics and the natural sciences. We have chosen to use the Python programming language because it combines remarkable expressive power with very clean, simple, and compact syntax. Python is easy to learn and very well suited for an introduction to computer programming. Python is also quite similar to MATLAB and a good language for doing mathematical computing. It is easy to combine Python with compiled languages, like Fortran, C, and C++, which are widely used languages for scientific computations.

El objetivo de este libro es enseñar programación informática utilizando ejemplos de matemáticas y ciencias naturales. Hemos elegido utilizar el lenguaje de programación Python porque combina un notable poder expresivo con una sintaxis muy limpia, simple y compacta. Python es fácil de aprender y muy adecuado para una introducción a la programación informática. **Python también es bastante similar a MATLAB y es un buen lenguaje para realizar cálculos matemáticos**. Es fácil combinar Python con lenguajes compilados, como Fortran, C y C++, que son lenguajes ampliamente utilizados para cálculos científicos.

The examples in this book integrate programming with applications to mathematics, physics, biology, and finance. The reader is expected to have knowledge of basic one-variable calculus as taught in mathematics-intensive programs in high schools. It is certainly an advantage to take a university calculus course in parallel, preferably containing both classical and numerical aspects of calculus. Although not strictly required, a background in high school physics makes many of the examples more meaningful.

**Los ejemplos de este libro integran la programación con aplicaciones a las matemáticas, la física, la biología y las finanzas.** Se espera que el lector tenga conocimientos de cálculo básico de una variable tal como se enseña en los programas intensivos de matemáticas en las escuelas secundarias. Sin duda, es una ventaja realizar un curso universitario de cálculo en paralelo, que preferiblemente contenga aspectos tanto clásicos como numéricos del cálculo. Aunque no es estrictamente necesario, tener experiencia en física en la escuela secundaria hace que muchos de los ejemplos sean más significativos.

Many introductory programming books are quite compact and focus on listing functionality of a programming language. However, learning to program is learning how to think as a programmer. This book has its main focus on the thinking process, or equivalently: programming as a problem solving technique. That is why most of the pages are devoted to case studies in programming, where we define a problem and explain how to create the corresponding program. New constructions and programming styles (what we could call theory) is also usually introduced via examples. Particular attention is paid to verification of programs and to finding errors. These topics are very demanding for mathematical software, because the unavoidable numerical approximation errors are possibly mixed with programming mistakes.

Muchos libros de introducción a la programación son bastante compactos y se centran en enumerar la funcionalidad de un lenguaje de programación. Sin embargo, aprender a programar es aprender a pensar como programador. **Este libro se centra principalmente en el proceso de pensamiento, o equivalentemente: la programación como técnica de resolución de problemas.** Es por eso que la mayoría de las páginas están dedicadas a estudios de casos de programación, donde definimos un problema y explicamos cómo crear el programa correspondiente. También se suelen introducir nuevas construcciones y estilos de programación (lo que podríamos llamar teoría) a través de ejemplos. Se presta especial atención a la verificación de programas y a la búsqueda de errores. Estos temas son muy exigentes para el software matemático, porque los inevitables errores de aproximación numérica posiblemente se mezclan con errores de programación.

By studying the many examples in the book, I hope readers will learn how to think right and thereby write programs in a quicker and more reliable way. Remember, nobody can learn programming by just reading – one has to solve a large amount of exercises hands on. The book is therefore full of exercises of various types: modifications of existing examples, completely new problems, or debugging of given programs.

Al estudiar los numerosos ejemplos del libro, espero que los lectores aprendan a pensar correctamente y así escribir programas de una forma más rápida y fiable. Recuerde, nadie puede aprender a programar simplemente leyendo: hay que resolver una gran cantidad de ejercicios de forma práctica. Por tanto, el libro está lleno de ejercicios de varios tipos: modificaciones de ejemplos existentes, problemas completamente nuevos o depuración de programas determinados.

To work with this book, I recommend using Python version 2.7. For Chaps. 5–9

and Appendices A–E, **you need the NumPy and Matplotlib packages**, preferably

also the **IPython** and **SciTools** packages, and for Appendix G, **Cython** is required.

Other packages used in the text are **nose** and **sympy**. Section H.1 has more information on how you can get access to Python and the mentioned packages.

There is a web page associated with this book, **http://hplgit.github.io/scipro-**

**primer**, containing all the example programs from the book as well as information

on installation of the software on various platforms.

Python version 2 or 3? A common problem among Python programmers is to choose between version 2 or 3, which at the time of this writing means choosing between version 2.7 and 3.5. A common recommendation is to go for Python 3, because this is the version that will be further developed in the future. However, there is a problem that much useful mathematical software in Python has not yet been ported to Python 3. Therefore, Python version 2.7 is the most popular version for doing scientific computing, and that is why also this book applies version 2.7.

A widely used strategy for software developers who want to write Python code that works with both versions, is to develop a common version for Python 2 and 3. For the programs in this book, a common version can easily be produced by first developing for version 2.7 and then automatically convert the code by running the **futurize** program. Section 4.10 demonstrates how this is done in simple cases.

The Python 2.7 code in this book sticks to all modern constructions that are backported from version 3 such that the code becomes as close as possible to the equivalent Python 3 code. At any time, you can just run futurize to see the differences between your Python 2.7 version and the corresponding Python 3.5 version.

Contents Chapter 1 introduces variables, objects, modules, and text formatting through examples concerning evaluation of mathematical formulas. Chapter 2 presents programming with while and for loops as well as lists, including nested lists. The next chapter deals with two other fundamental concepts in programming: functions and if-else tests.

How to read data into programs and deal with errors in input are the subjects of Chap. 4. Chapter 5 introduces arrays and array computing (including vectorization) and how this is used for plotting y D f .x/ curves and making animation of curves. Many of the examples in the first five chapters are strongly related. Typically, formulas from the first chapter are used to produce tables of numbers in the second chapter. Then the formulas are encapsulated in functions in the third chapter. In the next chapter, the input to the functions are fetched from the command line, and validity checks of the input are added. The formulas are then shown as graphs in Chap. 5. After having studied Chaps. 1–5, the reader should have enough knowledge of programming to solve mathematical problems by what many refer to as

“MATLAB-style” programming.

Chapter 6 explains how to work with dictionaries and strings, especially for interpreting text data in files and storing the extracted information in flexible data structures. Class programming, including user-defined types for mathematical computations (with overloaded operators), is introduced in Chap. 7. Chapter 8 deals with random numbers and statistical computing with applications to games and random walks. Object-oriented programming, in the meaning of class hierarchies and inheritance, is the subject of Chap. 9. The key examples here deal with building toolkits for numerical differentiation and integration as well as graphics.

Appendix A introduces mathematical modeling, using sequences and difference equations. Only programming concepts from Chaps. 1–5 are used in this appendix, the aim being to consolidate basic programming knowledge and apply it to mathematical problems. Some important mathematical topics are introduced via difference equations in a simple way: Newton’s method, Taylor series, inverse functions, and dynamical systems.

Appendix B deals with functions on a mesh, numerical differentiation, and numerical integration. A simple introduction to ordinary differential equations and their numerical treatment is provided in Appendix C. Appendix D shows how a complete project in physics can be solved by mathematical modeling, numerical methods, and programming elements from Chaps. 1–5. This project is a good example on problem solving in computational science, where it is necessary to integrate physics, mathematics, numerics, and computer science.

How to create software for solving ordinary differential equations, using both function-based and object-oriented programming, is the subject of Appendix E. The material in this appendix brings together many parts of the book in the context of physical applications and differential equations.

Appendix F is devoted to the art of debugging, and in fact problem solving in general. Speeding up numerical computations in Python by migrating code to C via Cython is exemplified in Appendix G. Finally, Appendix H deals with various more advanced technical topics.

Most of the examples and exercises in this book are quite short. However, many of the exercises are related, and together they form larger projects, for example on Fourier Series (3.21, 4.21, 4.22, 5.41, 5.42), numerical integration (3.11, 3.12, 5.49, 5.50, A.12), Taylor series (3.37, 5.32, 5.39, A.14, A.15, 7.23), piecewise constant functions (3.29–3.33, 5.34, 5.47, 5.48, 7.19–7.21), inverse functions (E.17–E.20), falling objects (E.8, E.9, E.38, E.39), oscillatory population growth (A.19, A.21, A.22, A.23), epidemic disease modeling (E.41–E.48), optimization and finance (A.24, 8.42, 8.43), statistics and probability (4.24, 4.25, 8.23, 8.24), hazard games (8.8–8.14), random walk and statistical physics (8.32–8.40), noisy data analysis (8.44–8.46), numerical methods (5.25–5.27, 7.8, 7.9, A.9, 7.22, 9.15–9.17, E.30–E.37), building a calculus calculator (7.34, 9.18, 9.19), and creating a toolkit for simulating vibrating engineering systems (E.50–E.55).

Chapters 1–9 together with Appendices A and E have from 2007 formed the core of an introductory first semester bachelor course on scientific programming at the University of Oslo (INF1100, 10 ECTS credits).

Changes from the fourth to the fifth edition Substantial changes were introduced in the fourth edition, and the fifth edition is primarily a consolidation of those changes. Many typos have been corrected and many explanations and exercises have been improved. The emphasis on unit tests and test functions, especially in exercises, is stronger than in the previous edition. Symbolic computation with the aid of SymPy is used to a larger extent and integrated with numerical computing throughout the book. All classes are now new-style (instead of old-style/classic as in previous editions). Examples on Matplotlib do not use the pylab module anymore, but pyplot and MATLAB-like syntax is still favored to ease the transition between Python and MATLAB. The concept of closures is more explicit than in earlier editions (see the new Sect. 7.1.7) since this is a handy and popular construction much used in the scientific Python community. We also discuss the difference between Python 2 and 3 and demonstrate how to use the future module to write code that runs under both versions.

The most substantial new material in the fifth edition appears toward the end of Chap. 5 and regards high-performance computing, linear algebra, and visualization of scalar and vector fields. Although this material is not used elsewhere in the book, many readers have requested basic recipes when going from one to two variables or from vectors to matrices later when solving more advanced problems and using the book as their programming reference. The new matrial in Chap. 5 was written jointly with Dr. Øyvind Ryan.

Contents

1 Computing with Formulas . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.1 The First Programming Encounter: a Formula . . . . . . . . . . . . . 1

1.1.1 Using a Program as a Calculator . . . . . . . . . . . . . . . . . 2

1.1.2 About Programs and Programming . . . . . . . . . . . . . . . 2

1.1.3 Tools for Writing Programs . . . . . . . . . . . . . . . . . . . . 3

1.1.4 Writing and Running Your First Python Program . . . . . . 4

1.1.5 Warning About Typing Program Text . . . . . . . . . . . . . . 5

1.1.6 Verifying the Result . . . . . . . . . . . . . . . . . . . . . . . . 6

1.1.7 Using Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.1.8 Names of Variables . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.1.9 Reserved Words in Python . . . . . . . . . . . . . . . . . . . . 7

1.1.10 Comments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

1.1.11 Formatting Text and Numbers . . . . . . . . . . . . . . . . . . 9

1.2 Computer Science Glossary . . . . . . . . . . . . . . . . . . . . . . . . 12

1.3 Another Formula: Celsius-Fahrenheit Conversion . . . . . . . . . . 16

1.3.1 Potential Error: Integer Division . . . . . . . . . . . . . . . . . 16

1.3.2 Objects in Python . . . . . . . . . . . . . . . . . . . . . . . . . . 17

1.3.3 Avoiding Integer Division . . . . . . . . . . . . . . . . . . . . . 18

1.3.4 Arithmetic Operators and Precedence . . . . . . . . . . . . . 20

1.4 Evaluating Standard Mathematical Functions . . . . . . . . . . . . . 20

1.4.1 Example: Using the Square Root Function . . . . . . . . . . 20

1.4.2 Example: Computing with sinh x . . . . . . . . . . . . . . . . 23

1.4.3 A First Glimpse of Rounding Errors . . . . . . . . . . . . . . 23

1.5 Interactive Computing . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

1.5.1 Using the Python Shell . . . . . . . . . . . . . . . . . . . . . . 25

1.5.2 Type Conversion . . . . . . . . . . . . . . . . . . . . . . . . . . 26

1.5.3 IPython . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

1.6 Complex Numbers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

1.6.1 Complex Arithmetics in Python . . . . . . . . . . . . . . . . . 30

1.6.2 Complex Functions in Python . . . . . . . . . . . . . . . . . . 31

1.6.3 Unified Treatment of Complex and Real Functions . . . . . 31

1.7 Symbolic Computing . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

1.7.1 Basic Differentiation and Integration . . . . . . . . . . . . . . 33

1.7.2 Equation Solving . . . . . . . . . . . .

xii Contents

1.7.3 Taylor Series and More . . . . . . . . . . . . . . . . . . . . . . 35

1.8 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

1.8.1 Chapter Topics . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

1.8.2 Example: Trajectory of a Ball . . . . . . . . . . . . . . . . . . 39

1.8.3 About Typesetting Conventions in This Book . . . . . . . . . 40

1.9 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

2 Loops and Lists . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

2.1 While Loops . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

2.1.1 A Naive Solution . . . . . . . . . . . . . . . . . . . . . . . . . . 51

2.1.2 While Loops . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

2.1.3 Boolean Expressions . . . . . . . . . . . . . . . . . . . . . . . . 54

2.1.4 Loop Implementation of a Sum . . . . . . . . . . . . . . . . . 56

2.2 Lists . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

2.2.1 Basic List Operations . . . . . . . . . . . . . . . . . . . . . . . 57

2.2.2 For Loops . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

2.3 Alternative Implementations with Lists and Loops . . . . . . . . . . 62

2.3.1 While Loop Implementation of a for Loop . . . . . . . . . . 62

2.3.2 The Range Construction . . . . . . . . . . . . . . . . . . . . . . 63

2.3.3 For Loops with List Indices . . . . . . . . . . . . . . . . . . . . 64

2.3.4 Changing List Elements . . . . . . . . . . . . . . . . . . . . . . 65

2.3.5 List Comprehension . . . . . . . . . . . . . . . . . . . . . . . . 66

2.3.6 Traversing Multiple Lists Simultaneously . . . . . . . . . . . 66

2.4 Nested Lists . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

2.4.1 A table as a List of Rows or Columns . . . . . . . . . . . . . 67

2.4.2 Printing Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

2.4.3 Extracting Sublists . . . . . . . . . . . . . . . . . . . . . . . . . 70

2.4.4 Traversing Nested Lists . . . . . . . . . . . . . . . . . . . . . . 72

2.5 Tuples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

2.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75

2.6.1 Chapter Topics . . . . . . . . . . . . . . . . . . . . . . . . . . . 75

2.6.2 Example: Analyzing List Data . . . . . . . . . . . . . . . . . . 78

2.6.3 How to Find More Python Information . . . . . . . . . . . . . 80

2.7 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

3 Functions and Branching . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91

3.1 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91

3.1.1 Mathematical Functions as Python Functions . . . . . . . . . 91

3.1.2 Understanding the Program Flow . . . . . . . . . . . . . . . . 93

3.1.3 Local and Global Variables . . . . . . . . . . . . . . . . . . . . 94

3.1.4 Multiple Arguments . . . . . . . . . . . . . . . . . . . . . . . . 96

3.1.5 Function Argument or Global Variable? . . . . . . . . . . . . 97

3.1.6 Beyond Mathematical Functions . . . . . . . . . . . . . . . . . 98

3.1.7 Multiple Return Values . . . . . . . . . . . . . . . . . . . . . . 99

3.1.8 Computing Sums . . . . . . . . . . . . . . . . . . . . . . . . . . 100

3.1.9 Functions with No Return Values . . . . . . . . . . . . . . . . 101

3.1.10 Keyword Arguments . . . . . . . . . . . . . . . . . . . . . . . . 103

3.1.11 Doc Strings . . . . . . .

Contents xiii

3.1.12 Functions as Arguments to Functions . . . . . . . . . . . . . . 107

3.1.13 The Main Program . . . . . . . . . . . . . . . . . . . . . . . . . 109

3.1.14 Lambda Functions . . . . . . . . . . . . . . . . . . . . . . . . . 110

3.2 Branching . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110

3.2.1 If-else Blocks . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111

3.2.2 Inline if Tests . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113

3.3 Mixing Loops, Branching, and Functions in Bioinformatics

Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113

3.3.1 Counting Letters in DNA Strings . . . . . . . . . . . . . . . . 114

3.3.2 Efficiency Assessment . . . . . . . . . . . . . . . . . . . . . . . 118

3.3.3 Verifying the Implementations . . . . . . . . . . . . . . . . . . 120

3.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121

3.4.1 Chapter Topics . . . . . . . . . . . . . . . . . . . . . . . . . . . 121

3.4.2 Example: Numerical Integration . . . . . . . . . . . . . . . . . 123

3.5 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127

4 User Input and Error Handling . . . . . . . . . . . . . . . . . . . . . . . . 149

4.1 Asking Questions and Reading Answers . . . . . . . . . . . . . . . . 150

4.1.1 Reading Keyboard Input . . . . . . . . . . . . . . . . . . . . . . 150

4.2 Reading from the Command Line . . . . . . . . . . . . . . . . . . . . 151

4.2.1 Providing Input on the Command Line . . . . . . . . . . . . . 151

4.2.2 A Variable Number of Command-Line Arguments . . . . . 152

4.2.3 More on Command-Line Arguments . . . . . . . . . . . . . . 153

4.3 Turning User Text into Live Objects . . . . . . . . . . . . . . . . . . . 154

4.3.1 The Magic Eval Function . . . . . . . . . . . . . . . . . . . . . 154

4.3.2 The Magic Exec Function . . . . . . . . . . . . . . . . . . . . . 158

4.3.3 Turning String Expressions into Functions . . . . . . . . . . 160

4.4 Option-Value Pairs on the Command Line . . . . . . . . . . . . . . . 161

4.4.1 Basic Usage of the Argparse Module . . . . . . . . . . . . . . 162

4.4.2 Mathematical Expressions as Values . . . . . . . . . . . . . . 163

4.5 Reading Data from File . . . . . . . . . . . . . . . . . . . . . . . . . . . 165

4.5.1 Reading a File Line by Line . . . . . . . . . . . . . . . . . . . 166

4.5.2 Alternative Ways of Reading a File . . . . . . . . . . . . . . . 167

4.5.3 Reading a Mixture of Text and Numbers . . . . . . . . . . . . 169

4.6 Writing Data to File . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 171

4.6.1 Example: Writing a Table to File . . . . . . . . . . . . . . . . 171

4.6.2 Standard Input and Output as File Objects . . . . . . . . . . . 173

4.6.3 What is a File, Really? . . . . . . . . . . . . . . . . . . . . . . . 176

4.7 Handling Errors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 179

4.7.1 Exception Handling . . . . . . . . . . . . . . . . . . . . . . . . 180

4.7.2 Raising Exceptions . . . . . . . . . . . . . . . . . . . . . . . . . 183

4.8 A Glimpse of Graphical User Interfaces . . . . . . . . . . . . . . . . 185

4.9 Making Modules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 188

4.9.1 Example: Interest on Bank Deposits . . . . . . . . . . . . . . 188

4.9.2 Collecting Functions in a Module File . . . . . . . . . . . . . 189

4.9.3 Test Block . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 190

4.9.4 Verification of the Module Code . . . . . . . . . . . . . . . . . 192

4.9.5 Getting Input Data . . . . . . . . . . . . . . .

\\

xiv Contents

4.9.6 Doc Strings in Modules . . . . . . . . . . . . . . . . . . . . . . 195

4.9.7 Using Modules . . . . . . . . . . . . . . . . . . . . . . . . . . . 196

4.9.8 Distributing Modules . . . . . . . . . . . . . . . . . . . . . . . . 199

4.9.9 Making Software Available on the Internet . . . . . . . . . . 200

4.10 Making Code for Python 2 and 3 . . . . . . . . . . . . . . . . . . . . . 201

4.10.1 Basic Differences Between Python 2 and 3 . . . . . . . . . . 201

4.10.2 Turning Python 2 Code into Python 3 Code . . . . . . . . . . 202

4.11 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 204

4.11.1 Chapter Topics . . . . . . . . . . . . . . . . . . . . . . . . . . . 204

4.11.2 Example: Bisection Root Finding . . . . . . . . . . . . . . . . 208

4.12 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 216

5 Array Computing and Curve Plotting . . . . . . . . . . . . . . . . . . . . 227

5.1 Vectors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 228

5.1.1 The Vector Concept . . . . . . . . . . . . . . . . . . . . . . . . 228

5.1.2 Mathematical Operations on Vectors . . . . . . . . . . . . . . 229

5.1.3 Vector Arithmetics and Vector Functions . . . . . . . . . . . 231

5.2 Arrays in Python Programs . . . . . . . . . . . . . . . . . . . . . . . . 232

5.2.1 Using Lists for Collecting Function Data . . . . . . . . . . . 232

5.2.2 Basics of Numerical Python Arrays . . . . . . . . . . . . . . . 233

5.2.3 Computing Coordinates and Function Values . . . . . . . . . 235

5.2.4 Vectorization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 236

5.3 Curve Plotting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 238

5.3.1 MATLAB-Style Plotting with Matplotlib . . . . . . . . . . . 238

5.3.2 Matplotlib; Pyplot Prefix . . . . . . . . . . . . . . . . . . . . . 243

5.3.3 SciTools and Easyviz . . . . . . . . . . . . . . . . . . . . . . . . 244

5.3.4 Making Animations . . . . . . . . . . . . . . . . . . . . . . . . 249

5.3.5 Making Videos . . . . . . . . . . . . . . . . . . . . . . . . . . . 254

5.3.6 Curve Plots in Pure Text . . . . . . . . . . . . . . . . . . . . . . 255

5.4 Plotting Difficulties . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 256

5.4.1 Piecewisely Defined Functions . . . . . . . . . . . . . . . . . . 256

5.4.2 Rapidly Varying Functions . . . . . . . . . . . . . . . . . . . . 259

5.5 More Advanced Vectorization of Functions . . . . . . . . . . . . . . 260

5.5.1 Vectorization of StringFunction Objects . . . . . . . . . . . . 260

5.5.2 Vectorization of the Heaviside Function . . . . . . . . . . . . 261

5.5.3 Vectorization of a Hat Function . . . . . . . . . . . . . . . . . 265

5.6 More on Numerical Python Arrays . . . . . . . . . . . . . . . . . . . . 267

5.6.1 Copying Arrays . . . . . . . . . . . . . . . . . . . . . . . . . . . 267

5.6.2 In-Place Arithmetics . . . . . . . . . . . . . . . . . . . . . . . . 268

5.6.3 Allocating Arrays . . . . . . . . . . . . . . . . . . . . . . . . . . 269

5.6.4 Generalized Indexing . . . . . . . . . . . . . . . . . . . . . . . 269

5.6.5 Testing for the Array Type . . . . . . . . . . . . . . . . . . . . 270

5.6.6 Compact Syntax for Array Generation . . . . . . . . . . . . . 271

5.6.7 Shape Manipulation . . . . . . . . . . . . . . . . . . . . . . . . 271

5.7 High-Performance Computing with Arrays . . . . . . . . . . . . . . 272

5.7.1 Scalar Implementation . . . . . . . . . . . . . . . . . . . . . . . 272

5.7.2 Vectorized Implementation . . . . . . . . . . . . . . . . . . . . 273

5.7.3 Memory-Saving Implementation . .

Contents xv

5.7.4 Analysis of Memory Usage . . . . . . . . . . . . . . . . . . . . 275

5.7.5 Analysis of the CPU Time . . . . . . . . . . . . . . . . . . . . 276

5.8 Higher-Dimensional Arrays . . . . . . . . . . . . . . . . . . . . . . . . 277

5.8.1 Matrices and Arrays . . . . . . . . . . . . . . . . . . . . . . . . 277

5.8.2 Two-Dimensional Numerical Python Arrays . . . . . . . . . 278

5.8.3 Array Computing . . . . . . . . . . . . . . . . . . . . . . . . . . 281

5.8.4 Matrix Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . 282

5.9 Some Common Linear Algebra Operations . . . . . . . . . . . . . . 283

5.9.1 Inverse, Determinant, and Eigenvalues . . . . . . . . . . . . . 283

5.9.2 Products . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 283

5.9.3 Norms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 284

5.9.4 Sum and Extreme Values . . . . . . . . . . . . . . . . . . . . . 284

5.9.5 Indexing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 286

5.9.6 Transpose and Upper/Lower Triangular Parts . . . . . . . . . 286

5.9.7 Solving Linear Systems . . . . . . . . . . . . . . . . . . . . . . 287

5.9.8 Matrix Row and Column Operations . . . . . . . . . . . . . . 287

5.9.9 Computing the Rank of a Matrix . . . . . . . . . . . . . . . . 288

5.9.10 Symbolic Linear Algebra . . . . . . . . . . . . . . . . . . . . . 289

5.10 Plotting of Scalar and Vector Fields . . . . . . . . . . . . . . . . . . . 292

5.10.1 Installation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 292

5.10.2 Surface Plots . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 293

5.10.3 Parameterized Curve . . . . . . . . . . . . . . . . . . . . . . . . 293

5.10.4 Contour Lines . . . . . . . . . . . . . . . . . . . . . . . . . . . . 294

5.10.5 The Gradient Vector Field . . . . . . . . . . . . . . . . . . . . . 294

5.11 Matplotlib . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 296

5.11.1 Surface Plots . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 296

5.11.2 Contour Plots . . . . . . . . . . . . . . . . . . . . . . . . . . . . 297

5.11.3 Vector Field Plots . . . . . . . . . . . . . . . . . . . . . . . . . . 299

5.12 Mayavi . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 299

5.12.1 Surface Plots . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 300

5.12.2 Contour Plots . . . . . . . . . . . . . . . . . . . . . . . . . . . . 303

5.12.3 Vector Field Plots . . . . . . . . . . . . . . . . . . . . . . . . . . 303

5.12.4 A 3D Scalar Field and Its Gradient Field . . . . . . . . . . . . 304

5.12.5 Animations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 306

5.13 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 307

5.13.1 Chapter Topics . . . . . . . . . . . . . . . . . . . . . . . . . . . 307

5.13.2 Example: Animating a Function . . . . . . . . . . . . . . . . . 308

5.14 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 313

6 Dictionaries and Strings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 333

6.1 Dictionaries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 333

6.1.1 Making Dictionaries . . . . . . . . . . . . . . . . . . . . . . . . 334

6.1.2 Dictionary Operations . . . . . . . . . . . . . . . . . . . . . . . 334

6.1.3 Example: Polynomials as Dictionaries . . . . . . . . . . . . . 336

6.1.4 Dictionaries with Default Values and Ordering . . . . . . . . 338

6.1.5 Example: Storing File Data in Dictionaries . . . . . . . . . . 341

6.1.6 Example: Storing File Data in Nested Dictionaries . . . . . 342

xvi Contents

6.1.7 Example: Reading and Plotting Data Recorded at Specific

Dates . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 347

6.2 Strings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 351

6.2.1 Common Operations on Strings . . . . . . . . . . . . . . . . . 351

6.2.2 Example: Reading Pairs of Numbers . . . . . . . . . . . . . . 355

6.2.3 Example: Reading Coordinates . . . . . . . . . . . . . . . . . 358

6.3 Reading Data from Web Pages . . . . . . . . . . . . . . . . . . . . . . 360

6.3.1 About Web Pages . . . . . . . . . . . . . . . . . . . . . . . . . . 361

6.3.2 How to Access Web Pages in Programs . . . . . . . . . . . . 362

6.3.3 Example: Reading Pure Text Files . . . . . . . . . . . . . . . 363

6.3.4 Example: Extracting Data from HTML . . . . . . . . . . . . 365

6.3.5 Handling Non-English Text . . . . . . . . . . . . . . . . . . . . 366

6.4 Reading and Writing Spreadsheet Files . . . . . . . . . . . . . . . . . 369

6.4.1 CSV Files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 369

6.4.2 Reading CSV Files . . . . . . . . . . . . . . . . . . . . . . . . . 370

6.4.3 Processing Spreadsheet Data . . . . . . . . . . . . . . . . . . . 371

6.4.4 Writing CSV Files . . . . . . . . . . . . . . . . . . . . . . . . . 372

6.4.5 Representing Number Cells with Numerical Python Arrays 373

6.4.6 Using More High-Level Numerical Python Functionality . 374

6.5 Examples from Analyzing DNA . . . . . . . . . . . . . . . . . . . . . 375

6.5.1 Computing Frequencies . . . . . . . . . . . . . . . . . . . . . . 375

6.5.2 Analyzing the Frequency Matrix . . . . . . . . . . . . . . . . . 382

6.5.3 Finding Base Frequencies . . . . . . . . . . . . . . . . . . . . . 385

6.5.4 Translating Genes into Proteins . . . . . . . . . . . . . . . . . 388

6.5.5 Some Humans Can Drink Milk, While Others Cannot . . . 393

6.6 Making Code that is Compatible with Python 2 and 3 . . . . . . . . 394

6.6.1 More Basic Differences Between Python 2 and 3 . . . . . . 394

6.6.2 Turning Python 2 Code into Python 3 Code . . . . . . . . . . 396

6.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 396

6.7.1 Chapter Topics . . . . . . . . . . . . . . . . . . . . . . . . . . . 396

6.7.2 Example: A File Database . . . . . . . . . . . . . . . . . . . . 398

6.8 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 402

7 Introduction to Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 409

7.1 Simple Function Classes . . . . . . . . . . . . . . . . . . . . . . . . . . 409

7.1.1 Challenge: Functions with Parameters . . . . . . . . . . . . . 410

7.1.2 Representing a Function as a Class . . . . . . . . . . . . . . . 412

7.1.3 The Self Variable . . . . . . . . . . . . . . . . . . . . . . . . . . 417

7.1.4 Another Function Class Example . . . . . . . . . . . . . . . . 419

7.1.5 Alternative Function Class Implementations . . . . . . . . . 420

7.1.6 Making Classes Without the Class Construct . . . . . . . . . 422

7.1.7 Closures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 424

7.2 More Examples on Classes . . . . . . . . . . . . . . . . . . . . . . . . 426

7.2.1 Bank Accounts . . . . . . . . . . . . . . . . . . . . . . . . . . . 426

7.2.2 Phone Book . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 428

7.2.3 A Circle . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 430

7.3 Special Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 432

7.3.1 The Call Special Method . .

Contents xvii

7.3.2 Example: Automagic Differentiation . . . . . . . . . . . . . . 433

7.3.3 Example: Automagic Integration . . . . . . . . . . . . . . . . 438

7.3.4 Turning an Instance into a String . . . . . . . . . . . . . . . . 440

7.3.5 Example: Phone Book with Special Methods . . . . . . . . . 441

7.3.6 Adding Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . 443

7.3.7 Example: Class for Polynomials . . . . . . . . . . . . . . . . . 443

7.3.8 Arithmetic Operations and Other Special Methods . . . . . 449

7.3.9 Special Methods for String Conversion . . . . . . . . . . . . . 449

7.4 Example: Class for Vectors in the Plane . . . . . . . . . . . . . . . . 451

7.4.1 Some Mathematical Operations on Vectors . . . . . . . . . . 451

7.4.2 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . 452

7.4.3 Usage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 454

7.5 Example: Class for Complex Numbers . . . . . . . . . . . . . . . . . 455

7.5.1 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . 455

7.5.2 Illegal Operations . . . . . . . . . . . . . . . . . . . . . . . . . . 457

7.5.3 Mixing Complex and Real Numbers . . . . . . . . . . . . . . 457

7.5.4 Dynamic, Static, Strong, Weak, and Duck Typing . . . . . . 459

7.5.5 Special Methods for “Right” Operands . . . . . . . . . . . . . 460

7.5.6 Inspecting Instances . . . . . . . . . . . . . . . . . . . . . . . . 461

7.6 Static Methods and Attributes . . . . . . . . . . . . . . . . . . . . . . . 463

7.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 464

7.7.1 Chapter Topics . . . . . . . . . . . . . . . . . . . . . . . . . . . 464

7.7.2 Example: Interval Arithmetic . . . . . . . . . . . . . . . . . . 466

7.8 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 470

8 Random Numbers and Simple Games . . . . . . . . . . . . . . . . . . . . 489

8.1 Drawing Random Numbers . . . . . . . . . . . . . . . . . . . . . . . . 489

8.1.1 The Seed . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 490

8.1.2 Uniformly Distributed Random Numbers . . . . . . . . . . . 491

8.1.3 Visualizing the Distribution . . . . . . . . . . . . . . . . . . . . 492

8.1.4 Vectorized Drawing of Random Numbers . . . . . . . . . . . 493

8.1.5 Computing the Mean and Standard Deviation . . . . . . . . . 494

8.1.6 The Gaussian or Normal Distribution . . . . . . . . . . . . . . 496

8.2 Drawing Integers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 497

8.2.1 Random Integer Functions . . . . . . . . . . . . . . . . . . . . 498

8.2.2 Example: Throwing a Die . . . . . . . . . . . . . . . . . . . . . 498

8.2.3 Drawing a Random Element from a List . . . . . . . . . . . . 501

8.2.4 Example: Drawing Cards from a Deck . . . . . . . . . . . . . 502

8.2.5 Example: Class Implementation of a Deck . . . . . . . . . . 504

8.3 Computing Probabilities . . . . . . . . . . . . . . . . . . . . . . . . . . 507

8.3.1 Principles of Monte Carlo Simulation . . . . . . . . . . . . . 507

8.3.2 Example: Throwing Dice . . . . . . . . . . . . . . . . . . . . . 508

8.3.3 Example: Drawing Balls from a Hat . . . . . . . . . . . . . . 511

8.3.4 Random Mutations of Genes . . . . . . . . . . . . . . . . . . . 513

8.3.5 Example: Policies for Limiting Population Growth . . . . . 519

8.4 Simple Games . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 522

8.4.1 Guessing a Number . . . . . . . . . . . . . . . . . . . . . . . . . 522

8.4.2 Rolling Two Dice . . . . . . . . . . . . . .

xviii Contents

8.5 Monte Carlo Integration . . . . . . . . . . . . . . . . . . . . . . . . . . 526

8.5.1 Derivation of Monte Carlo Integration . . . . . . . . . . . . . 526

8.5.2 Implementation of Standard Monte Carlo Integration . . . . 528

8.5.3 Area Computing by Throwing Random Points . . . . . . . . 531

8.6 Random Walk in One Space Dimension . . . . . . . . . . . . . . . . 534

8.6.1 Basic Implementation . . . . . . . . . . . . . . . . . . . . . . . 534

8.6.2 Visualization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 535

8.6.3 Random Walk as a Difference Equation . . . . . . . . . . . . 536

8.6.4 Computing Statistics of the Particle Positions . . . . . . . . . 536

8.6.5 Vectorized Implementation . . . . . . . . . . . . . . . . . . . . 537

8.7 Random Walk in Two Space Dimensions . . . . . . . . . . . . . . . . 539

8.7.1 Basic Implementation . . . . . . . . . . . . . . . . . . . . . . . 539

8.7.2 Vectorized Implementation . . . . . . . . . . . . . . . . . . . . 541

8.8 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 542

8.8.1 Chapter Topics . . . . . . . . . . . . . . . . . . . . . . . . . . . 542

8.8.2 Example: Random Growth . . . . . . . . . . . . . . . . . . . . 544

8.9 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 549

9 Object-Oriented Programming . . . . . . . . . . . . . . . . . . . . . . . . 567

9.1 Inheritance and Class Hierarchies . . . . . . . . . . . . . . . . . . . . 567

9.1.1 A Class for Straight Lines . . . . . . . . . . . . . . . . . . . . . 568

9.1.2 A First Try on a Class for Parabolas . . . . . . . . . . . . . . 569

9.1.3 A Class for Parabolas Using Inheritance . . . . . . . . . . . . 569

9.1.4 Checking the Class Type . . . . . . . . . . . . . . . . . . . . . 571

9.1.5 Attribute vs Inheritance: has-a vs is-a Relationship . . . . . 572

9.1.6 Superclass for Defining an Interface . . . . . . . . . . . . . . 574

9.2 Class Hierarchy for Numerical Differentiation . . . . . . . . . . . . . 576

9.2.1 Classes for Differentiation . . . . . . . . . . . . . . . . . . . . 577

9.2.2 Verification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 579

9.2.3 A flexible Main Program . . . . . . . . . . . . . . . . . . . . . 581

9.2.4 Extensions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 582

9.2.5 Alternative Implementation via Functions . . . . . . . . . . . 585

9.2.6 Alternative Implementation via Functional Programming . 586

9.2.7 Alternative Implementation via a Single Class . . . . . . . . 587

9.3 Class Hierarchy for Numerical Integration . . . . . . . . . . . . . . . 589

9.3.1 Numerical Integration Methods . . . . . . . . . . . . . . . . . 589

9.3.2 Classes for Integration . . . . . . . . . . . . . . . . . . . . . . . 590

9.3.3 Verification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 594

9.3.4 Using the Class Hierarchy . . . . . . . . . . . . . . . . . . . . 595

9.3.5 About Object-Oriented Programming . . . . . . . . . . . . . 597

9.4 Class Hierarchy for Making Drawings . . . . . . . . . . . . . . . . . 599

9.4.1 Using the Object Collection . . . . . . . . . . . . . . . . . . . 600

9.4.2 Example of Classes for Geometric Objects . . . . . . . . . . 609

9.4.3 Adding Functionality via Recursion . . . . . . . . . . . . . . 614

9.4.4 Scaling, Translating, and Rotating a Figure . . . . . . . . . . 618

9.5 Classes for DNA Analysis . . . . . . . . . . . . . . . . . . . . . . . . . 620

9.5.1 Class for Regions . . . . . . . . . . . . . . . . . . . . . . . . . . 620

9.5.2 Class for Genes . . . . . .

Contents xix

9.5.3 Subclasses . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 626

9.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 627

9.6.1 Chapter Topics . . . . . . . . . . . . . . . . . . . . . . . . . . . 627

9.6.2 Example: Input Data Reader . . . . . . . . . . . . . . . . . . . 629

9.7 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 635

A Sequences and Difference Equations . . . . . . . . . . . . . . . . . . . . . 645

A.1 Mathematical Models Based on Difference Equations . . . . . . . . 646

A.1.1 Interest Rates . . . . . . . . . . . . . . . . . . . . . . . . . . . . 647

A.1.2 The Factorial as a Difference Equation . . . . . . . . . . . . . 649

A.1.3 Fibonacci Numbers . . . . . . . . . . . . . . . . . . . . . . . . . 650

A.1.4 Growth of a Population . . . . . . . . . . . . . . . . . . . . . . 651

A.1.5 Logistic Growth . . . . . . . . . . . . . . . . . . . . . . . . . . . 652

A.1.6 Payback of a Loan . . . . . . . . . . . . . . . . . . . . . . . . . 654

A.1.7 The Integral as a Difference Equation . . . . . . . . . . . . . 655

A.1.8 Taylor Series as a Difference Equation . . . . . . . . . . . . . 657

A.1.9 Making a Living from a Fortune . . . . . . . . . . . . . . . . . 658

A.1.10Newton’s Method . . . . . . . . . . . . . . . . . . . . . . . . . . 659

A.1.11The Inverse of a Function . . . . . . . . . . . . . . . . . . . . . 663

A.2 Programming with Sound . . . . . . . . . . . . . . . . . . . . . . . . . 665

A.2.1 Writing Sound to File . . . . . . . . . . . . . . . . . . . . . . . 666

A.2.2 Reading Sound from File . . . . . . . . . . . . . . . . . . . . . 667

A.2.3 Playing Many Notes . . . . . . . . . . . . . . . . . . . . . . . . 667

A.2.4 Music of a Sequence . . . . . . . . . . . . . . . . . . . . . . . . 668

A.3 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 671

B Introduction to Discrete Calculus . . . . . . . . . . . . . . . . . . . . . . . 683

B.1 Discrete Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 683

B.1.1 The Sine Function . . . . . . . . . . . . . . . . . . . . . . . . . 684

B.1.2 Interpolation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 685

B.1.3 Evaluating the Approximation . . . . . . . . . . . . . . . . . . 686

B.1.4 Generalization . . . . . . . . . . . . . . . . . . . . . . . . . . . . 687

B.2 Differentiation Becomes Finite Differences . . . . . . . . . . . . . . 688

B.2.1 Differentiating the Sine Function . . . . . . . . . . . . . . . . 689

B.2.2 Differences on a Mesh . . . . . . . . . . . . . . . . . . . . . . . 690

B.2.3 Generalization . . . . . . . . . . . . . . . . . . . . . . . . . . . . 692

B.3 Integration Becomes Summation . . . . . . . . . . . . . . . . . . . . . 693

B.3.1 Dividing into Subintervals . . . . . . . . . . . . . . . . . . . . 693

B.3.2 Integration on Subintervals . . . . . . . . . . . . . . . . . . . . 695

B.3.3 Adding the Subintervals . . . . . . . . . . . . . . . . . . . . . . 696

B.3.4 Generalization . . . . . . . . . . . . . . . . . . . . . . . . . . . . 697

B.4 Taylor Series . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 699

B.4.1 Approximating Functions Close to One Point . . . . . . . . . 699

B.4.2 Approximating the Exponential Function . . . . . . . . . . . 699

B.4.3 More Accurate Expansions . . . . . . . . . . . . . . . . . . . . 700

B.4.4 Accuracy of the Approximation . . . . . . . . . . . . . . . . . 702

B.4.5 Derivatives Revisited . . . . . . . . . . . . . . . . . . . . . . . . 704

B.4.6 More Accurate Difference Approximations . . . .

xx Contents

B.4.7 Second-Order Derivatives . . . . . . . . . . . . . . . . . . . . . 707

B.5 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 709

C Introduction to differential equations . . . . . . . . . . . . . . . . . . . . 715

C.1 The simplest case . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 716

C.2 Exponential Growth . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 718

C.3 Logistic Growth . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 723

C.4 A Simple Pendulum . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 724

C.5 A Model for the Spreading of a Disease . . . . . . . . . . . . . . . . 727

C.6 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 729

D A Complete Differential Equation Project . . . . . . . . . . . . . . . . . 731

D.1 About the Problem: Motion and Forces in Physics . . . . . . . . . . 731

D.1.1 The Physical Problem . . . . . . . . . . . . . . . . . . . . . . . 731

D.1.2 The Computational Algorithm . . . . . . . . . . . . . . . . . . 733

D.1.3 Derivation of the Mathematical Model . . . . . . . . . . . . . 734

D.1.4 Derivation of the Algorithm . . . . . . . . . . . . . . . . . . . 736

D.2 Program Development and Testing . . . . . . . . . . . . . . . . . . . . 737

D.2.1 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . 737

D.2.2 Callback Functionality . . . . . . . . . . . . . . . . . . . . . . . 740

D.2.3 Making a Module . . . . . . . . . . . . . . . . . . . . . . . . . . 742

D.2.4 Verification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 743

D.3 Visualization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 746

D.3.1 Simultaneous Computation and Plotting . . . . . . . . . . . . 746

D.3.2 Some Applications . . . . . . . . . . . . . . . . . . . . . . . . . 748

D.3.3 Remark on Choosing t . . . . . . . . . . . . . . . . . . . . . 749

D.3.4 Comparing Several Quantities in Subplots . . . . . . . . . . . 750

D.3.5 Comparing Approximate and Exact Solutions . . . . . . . . 751

D.3.6 Evolution of the Error as t Decreases . . . . . . . . . . . . 752

D.4 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 755

E Programming of Differential Equations . . . . . . . . . . . . . . . . . . . 757

E.1 Scalar Ordinary Differential Equations . . . . . . . . . . . . . . . . . 758

E.1.1 Examples on Right-Hand-Side Functions . . . . . . . . . . . 758

E.1.2 The Forward Euler Scheme . . . . . . . . . . . . . . . . . . . . 759

E.1.3 Function Implementation . . . . . . . . . . . . . . . . . . . . . 760

E.1.4 Verifying the Implementation . . . . . . . . . . . . . . . . . . 761

E.1.5 From Discrete to Continuous Solution . . . . . . . . . . . . . 763

E.1.6 Switching Numerical Method . . . . . . . . . . . . . . . . . . 764

E.1.7 Class Implementation . . . . . . . . . . . . . . . . . . . . . . . 764

E.1.8 Logistic Growth via a Function-Based Approach . . . . . . 769

E.1.9 Logistic Growth via a Class-Based Approach . . . . . . . . . 769

E.2 Systems of Ordinary Differential Equations . . . . . . . . . . . . . . 772

E.2.1 Mathematical Problem . . . . . . . . . . . . . . . . . . . . . . . 773

E.2.2 Example of a System of ODEs . . . . . . . . . . . . . . . . . . 774

E.2.3 Function Implementation . . . . . . . . . . . . . . . . . . . . . 775

E.2.4 Class Implementation . . . . . . . . . . . . . . . . . . . . . . . 777

E.3 The ODESolver Class Hierar

Contents xxi

E.3.1 Numerical Methods . . . . . . . . . . . . . . . . . . . . . . . . 779

E.3.2 Construction of a Solver Hierarchy . . . . . . . . . . . . . . . 780

E.3.3 The Backward Euler Method . . . . . . . . . . . . . . . . . . . 783

E.3.4 Verification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 785

E.3.5 Example: Exponential Decay . . . . . . . . . . . . . . . . . . . 787

E.3.6 Example: The Logistic Equation with Problem and Solver

Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 789

E.3.7 Example: An Oscillating System . . . . . . . . . . . . . . . . 797

E.3.8 Application 4: The Trajectory of a Ball . . . . . . . . . . . . 799

E.3.9 Further Developments of ODESolver . . . . . . . . . . . . . . 801

E.4 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 802

F Debugging . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 835

F.1 Using a Debugger . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 835

F.2 How to Debug . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 838

F.2.1 A Recipe for Program Writing and Debugging . . . . . . . . 838

F.2.2 Application of the Recipe . . . . . . . . . . . . . . . . . . . . . 841

F.2.3 Getting Help from a Code Analyzer . . . . . . . . . . . . . . 853

G Migrating Python to Compiled Code . . . . . . . . . . . . . . . . . . . . . 857

G.1 Pure Python Code for Monte Carlo Simulation . . . . . . . . . . . . 857

G.1.1 The Computational Problem . . . . . . . . . . . . . . . . . . . 858

G.1.2 A Scalar Python Implementation . . . . . . . . . . . . . . . . 858

G.1.3 A Vectorized Python Implementation . . . . . . . . . . . . . . 859

G.2 Migrating Scalar Python Code to Cython . . . . . . . . . . . . . . . . 860

G.2.1 A Plain Cython Implementation . . . . . . . . . . . . . . . . . 860

G.2.2 A Better Cython Implementation . . . . . . . . . . . . . . . . 863

G.3 Migrating Code to C . . . . . . . . . . . . . . . . . . . . . . . . . . . . 865

G.3.1 Writing a C Program . . . . . . . . . . . . . . . . . . . . . . . . 865

G.3.2 Migrating Loops to C Code via F2PY . . . . . . . . . . . . . 866

G.3.3 Migrating Loops to C Code via Cython . . . . . . . . . . . . 867

G.3.4 Comparing Efficiency . . . . . . . . . . . . . . . . . . . . . . . 868

H Technical Topics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 871

H.1 Getting Access to Python . . . . . . . . . . . . . . . . . . . . . . . . . 871

H.1.1 Required Software . . . . . . . . . . . . . . . . . . . . . . . . . 871

H.1.2 Installing Software on Your Laptop: Mac OS X and

Windows . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 872

H.1.3 Anaconda and Spyder . . . . . . . . . . . . . . . . . . . . . . . 873

H.1.4 VMWare Fusion Virtual Machine . . . . . . . . . . . . . . . . 874

H.1.5 Dual Boot on Windows . . . . . . . . . . . . . . . . . . . . . . 877

H.1.6 Vagrant Virtual Machine . . . . . . . . . . . . . . . . . . . . . 877

H.2 How to Write and Run a Python Program . . . . . . . . . . . . . . . 878

H.2.1 The Need for a Text Editor . . . . . . . . . . . . . . . . . . . . 878

H.2.2 Terminal Windows . . . . . . . . . . . . . . . . . . . . . . . . . 880

H.3 The SageMathCloud and Wakari Web Services . . . . . . . . . . . . 880

H.3.1 Basic Intro to SageMathCloud . . . . . . . . . . . . . . . . . . 880

H.3.2 Basic Intro to Wakari . . . . . . . . .

xxii Contents

H.3.3 Installing Your Own Python Packages . . . . . . . . . . . . . 881

H.4 Writing IPython Notebooks . . . . . . . . . . . . . . . . . . . . . . . . 882

H.4.1 A Simple Program in the Notebook . . . . . . . . . . . . . . . 882

H.4.2 Mixing Text, Mathematics, Code, and Graphics . . . . . . . 882

H.5 Different Ways of Running Python Programs . . . . . . . . . . . . . 884

H.5.1 Executing Python Programs in iPython . . . . . . . . . . . . . 884

H.5.2 Executing Python Programs in Unix . . . . . . . . . . . . . . 884

H.5.3 Executing Python Programs in Windows . . . . . . . . . . . . 885

H.5.4 Executing Python Programs in Mac OS X . . . . . . . . . . . 887

H.5.5 Making a Complete Stand-Alone Executable . . . . . . . . . 887

H.6 Doing Operating System Tasks in Python . . . . . . . . . . . . . . . 888

H.7 Variable Number of Function Arguments . . . . . . . . . . . . . . . . 891

H.7.1 Variable Number of Positional Arguments . . . . . . . . . . . 891

H.7.2 Variable Number of Keyword Arguments . . . . . . . . . . . 894

H.8 Evaluating Program Efficiency . . . . . . . . . . . . . . . . . . . . . . 896

H.8.1 Making Time Measurements . . . . . . . . . . . . . . . . . . . 896

H.8.2 Profiling Python Programs . . . . . . . . . . . . . . . . . . . . 898

H.9 Software Testing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 899

H.9.1 Requirements of the Test Function . . . . . . . . . . . . . . . 900

H.9.2 Writing the Test Function; Precomputed Data . . . . . . . . 900

H.9.3 Writing the Test Function; Exact Numerical Solution . . . . 901

H.9.4 Testing of Function Robustness . . . . . . . . . . . . . . . . . 902

H.9.5 Automatic Execution of Tests . . . . . . . . . . . . . . . . . . 904

References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 907

Index . . . . . . . . . . . . . . . .

ppp32

**C01 Computing with Formulas**

Our first examples on computer programming involve programs that evaluate mathematical formulas. You will learn how to write and run a Python program, how to work with variables, how to compute with mathematical functions such as ex and sin x, and how to use Python for interactive calculations.

—-------------------------------------------------------

Nuestros primeros ejemplos de programación de computadoras involucran programas que evalúan **fórmulas matemáticas**. Aprenderá cómo **escribir y ejecutar un programa de Python**, cómo trabajar con **variables**, cómo **calcular con funciones matemáticas** como e^x, y sin x, y cómo usar Python para cálculos interactivos.

—-------------------------------------------------------

We assume that you are somewhat familiar with computers so that you know what files and folders are (another frequent word for folder is directory), how you move between folders, how you change file and folder names, and how you write text and save it in a file.

—-------------------------------------------------------

Suponemos que está algo familiarizado con las computadoras para que sepa qué archivos y carpetas son (otra palabra frecuente para **carpeta** es **directorio**), cómo se mueve entre carpetas, cómo cambia los nombres de archivos y carpetas, y cómo escribe texto y lo guarda. en un **archivo**.

—-------------------------------------------------------

All the program examples associated with this chapter can be downloaded as a tarfile or zipfile from the web page http://hplgit.github.com/scipro-primer. I strongly recommend you to visit this page, download and pack out the files. The examples are organized in a folder tree with src as root. Each subfolder corresponds to a particular chapter. For example, the subfolder formulas contains the program examples associated with this first chapter. The relevant subfolder name is listed at the beginning of every chapter.

—-------------------------------------------------------

Todos los ejemplos de programas asociados con este capítulo se pueden descargar como un **archivo tar o zip** desde la página web **http://hplgit.github.com/scipro-primer**. Le recomiendo encarecidamente que visite esta página, descargue y empaquete los archivos. Los ejemplos están organizados en un árbol de carpetas con src como raíz. Cada subcarpeta corresponde a un capítulo en particular. Por ejemplo, la subcarpeta fórmulas contiene los ejemplos de programas asociados con este primer capítulo. El nombre de la subcarpeta correspondiente aparece al principio de cada capítulo.

—-------------------------------------------------------

The folder structure with example programs can also be directly accessed in a GitHub repository1 on the web. You can click on the formulas folder to see all the examples from the present chapter. Clicking on a filename shows a nicely typeset version of the file. The file can be downloaded by first clicking Raw to get the plain text version of the file, and then right-clicking in the web page and choosing Save As. . . .

—-------------------------------------------------------

También se puede acceder directamente a la estructura de carpetas con programas de ejemplo en un repositorio de **GitHub1** en la web. Puede hacer clic en la carpeta de fórmulas para ver todos los ejemplos del presente capítulo. Al hacer clic en un nombre de archivo, se muestra una versión muy bien compuesta del archivo. El archivo se puede descargar haciendo clic primero en Raw para obtener la versión de texto sin formato del archivo y luego haciendo clic con el botón derecho en la página web y eligiendo Guardar como. . . .

—-------------------------------------------------------

**1.1 The First Programming Encounter: a Formula**

The first formula we shall consider concerns the vertical motion of a ball thrown up in the air. From Newton’s second law of motion one can set up a mathematical model for the motion of the ball and find that the vertical position of the ball, called y, varies with time t according to the following formula:

—-------------------------------------------------------

La primera **fórmula** que consideraremos se refiere al **movimiento vertical** de una pelota lanzada al aire. A partir de la **segunda ley de movimiento de Newton**, se puede establecer un **modelo matemático** para el movimiento de la pelota y encontrar que la posición vertical de la pelota, llamada **y**, varía con el tiempo **t** de acuerdo con la siguiente fórmula:

—-------------------------------------------------------

$$

y(t) = v\_0 t - \frac{1}{2} g t^2

$$

Here, $v\_0$ is the initial velocity of the ball, g is the acceleration of gravity, and t is time. Observe that the y axis is chosen such that the ball starts at y D 0 when t D 0. The above formula neglects air resistance, which is usually small unless v0 is large, see Exercise 1.11.

—-------------------------------------------------------

Aquí, $v\_0$ es la velocidad inicial de la pelota, g es la aceleración de la gravedad y t es el tiempo. Observe que el eje y se elige de modo que la pelota comience en y =0 cuando t = 0. La fórmula anterior ignora la resistencia del aire, que suele ser pequeña a menos que $v\_0$ sea grande; consulte el ejercicio 1.11.

—-------------------------------------------------------

To get an overview of the time it takes for the ball to move upwards and return to y = 0 again, we can look for solutions to the equation y = 0:

—-------------------------------------------------------

Para obtener una visión general del tiempo que tarda la pelota en moverse hacia arriba y volver a y = 0 nuevamente, podemos buscar soluciones a la ecuación y = 0:

—-------------------------------------------------------

$$

v\_0t - \frac{1}{2} g t^2 = t(v\_0 - \frac{1}{2} gt) = 0 \Rightarrow t = 0 or t = \frac{2v\_0}{g}

$$

That is, the ball returns after 2v0=g seconds, and it is therefore reasonable to restrict the interest of (1.1) to t 2 Œ0; 2v0=g.

—-------------------------------------------------------

Es decir, la pelota regresa después de $2v\_0/g$ segundos y, por lo tanto, es razonable restringir el interés de (1.1) a $t \in [0, 2v\_0/g]$.

—-------------------------------------------------------

**1.1.1 Using a Program as a Calculator**

Our first program will evaluate (1.1) for a specific choice of v0, g, and t. Choosing v0 D 5 m/s and g D 9:81 m/s2 makes the ball come back after t D 2v0=g 1 s. This means that we are basically interested in the time interval Œ0; 1. Say we want to compute the height of the ball at time t D 0:6 s. From (1.1) we have

—-------------------------------------------------------

Nuestro primer programa evaluará (1.1) para una elección específica de $v\_0, g y t$. Elegir $v\_0 =5 m/s$ y g = 9:81 $m/s^2$ hace que la pelota regrese después de $t = 2v\_0/g \approx 1 s$. Esto significa que estamos básicamente interesados en el intervalo de tiempo Œ0; 1. Digamos que queremos calcular la altura de la pelota en el tiempo t = 0:6 s. De (1.1) tenemos

—-------------------------------------------------------

$$

y = 5 \dot 0.6 - \frac{1}{2} \dot 9:81 \dot 0.6^2

$$

This arithmetic expression can be evaluated and its value can be printed by a very simple one-line Python program:

**print 5\*0.6 - 0.5\*9.81\*0.6\*\*2**

The four standard arithmetic operators are written as +, -, \*, and / in Python and most other computer languages. The exponentiation employs a double asterisk notation in Python, e.g., 0:62 is written as 0.6\*\*2. Our task now is to create the program and run it, and this will be described next.

—-----------------------------------------

Los **cuatro operadores aritméticos estándar** se escriben como +, -, \* y / en Python y en la mayoría de los demás lenguajes informáticos. La exponenciación emplea una notación de doble asterisco en Python, por ejemplo, $0.6^2$ se escribe como 0,6\*\*2. Nuestra tarea ahora es crear el programa y ejecutarlo, y esto se describirá a continuación.

—----------------------------------------

**1.1.2 About Programs and Programming**

A computer program is just a sequence of instructions to the computer, written in a computer language. Most computer languages look somewhat similar to English, but they are very much simpler. The number of words and associated instructions is very limited, so to perform a complicated operation we must combine a large number of different types of instructions. The program text, containing the sequence of instructions, is stored in one or more files. The computer can only do exactly what the program tells the computer to do.

—----------------------------------------

Un **programa de computadora** es solo una secuencia de instrucciones para la computadora, escritas en un lenguaje de computadora. La mayoría de los lenguajes de programación se parecen un poco al inglés, pero son mucho más simples. El número de palabras e instrucciones asociadas es muy limitado, por lo que para realizar una operación complicada debemos combinar una gran cantidad de diferentes tipos de instrucciones. El texto del programa, que contiene la secuencia de instrucciones, se almacena en uno o más archivos. La computadora solo puede hacer exactamente lo que el programa le dice que haga.

—----------------------------------------

Another perception of the word program is a file that can be run (“doubleclicked”) to perform a task. Sometimes this is a file with textual instructions (which is the case with Python), and sometimes this file is a translation of all the program text to a more efficient and computer-friendly language that is quite difficult to read for a human. All the programs in this chapter consist of short text stored in a single file. Other programs that you have used frequently, for instance Firefox or Internet Explorer for reading web pages, consist of program text distributed over a large number of files, written by a large number of people over many years. One single file contains the machine-efficient translation of the whole program, and this is normally the file that you double-click on when starting the program. In general, the word program means either this single file or the collection of files with textual instructions.

—----------------------------------------

Otra percepción de la palabra programa es un archivo que se puede ejecutar ("doble clic") para realizar una tarea. A veces, este es un archivo con instrucciones textuales (como es el caso de Python) y, a veces, este archivo es una traducción de todo el texto del programa a un lenguaje más eficiente y fácil de usar que es bastante difícil de leer para un ser humano. Todos los programas de este capítulo consisten en texto breve almacenado en un solo archivo. Otros programas que ha utilizado con frecuencia, por ejemplo Firefox o Internet Explorer para leer páginas web, consisten en texto de programa distribuido en una gran cantidad de archivos, escritos por una gran cantidad de personas durante muchos años. Un solo archivo contiene la traducción eficiente de la máquina de todo el programa, y este es normalmente el archivo en el que hace doble clic al iniciar el programa. En general, la palabra **programa** significa este archivo único o la colección de archivos con instrucciones textuales.

—----------------------------------------

Programming is obviously about writing programs, but this process is more than writing the correct instructions in a file. First, we must understand how a problem can be solved by giving a sequence of instructions to the computer. This is one of the most difficult things with programming. Second, we must express this sequence of instructions correctly in a computer language and store the corresponding text in a file (the program). This is normally the easiest part. Third, we must find out how to check the validity of the results. Usually, the results are not as expected, and we need to a fourth phase where we systematically track down the errors and correct them. Mastering these four steps requires a lot of training, which means making a large number of programs (exercises in this book, for instance!) and getting the programs to work.

—----------------------------------------

Obviamente, la programación se trata de escribir programas, pero este proceso es más que escribir las instrucciones correctas en un archivo. **Primero**, debemos entender cómo se puede resolver un problema dando una secuencia de instrucciones a la computadora. Esta es una de las cosas más difíciles con la programación. En **segundo** lugar, debemos expresar correctamente esta secuencia de instrucciones en un lenguaje informático y almacenar el texto correspondiente en un archivo (el programa). Esta es normalmente la parte más fácil. En **tercer** lugar, debemos averiguar cómo comprobar la validez de los resultados. Por lo general, los resultados no son los esperados y necesitamos una **cuarta** fase en la que sistemáticamente rastreamos los errores y los corregimos. **Dominar estos cuatro pasos** requiere mucho entrenamiento, lo que significa hacer una gran cantidad de programas (¡ejercicios en este libro, por ejemplo!) y hacer que los programas funcionen.

—----------------------------------------

**1.1.3 Tools for Writing Programs**

There are three alternative types of tools for writing Python programs:

* a plain text editor

Una editora de texto sin formato

* an integrated development environment (IDE) a text editor

un entorno de desarrollo integrado (IDE) un editor de texto

* an IPython notebook with

un cuaderno IPython con

What you choose depends on how you access Python. Section H.1 contains information on the various possibilities to install Python on your own computer, access a pre-installed Python environment on a computer system at an institution, or access Python in cloud services through your web browser. Based on teaching this and previous books to more than 3000 students, my recommendations go as follows.

Lo que elija depende de cómo acceda a Python. La sección H.1 contiene información sobre las diversas posibilidades de instalar Python en su propia computadora, acceder a un entorno de Python preinstalado en un sistema informático en una institución o acceder a Python en servicios en la nube a través de su navegador web. Basándome en enseñar este libro y los anteriores a más de 3000 estudiantes, mis recomendaciones son las siguientes.

* If you use this book in a course, the instructor has probably made a choice for how you should access Python – follow that advice.

Si utiliza este libro en un curso, el instructor probablemente haya elegido cómo debe acceder a Python; siga ese consejo.

* If you are a student at a university where Linux is the dominating operating system, install a virtual machine with Ubuntu on your own laptop and do all your scientific work in Ubuntu. Write Python programs in a text editor like Gedit, Atom, Sublime Text, Emacs, or Vim, and run programs in a terminal window (the gnome-terminal is recommended).

Si eres estudiante en una universidad donde Linux es el sistema operativo dominante, instala una máquina virtual con Ubuntu en tu propia computadora portátil y realiza todo tu trabajo científico en Ubuntu. Escriba programas Python en un editor de texto como **Gedit, Atom, Sublime Text, Emacs o Vim,** y ejecute programas en una ventana de terminal (se recomienda el terminal **gnome**).

* If you are a student a university where Windows is the dominating operating system, and you are a Windows user yourself, install Anaconda. Write and run Python programs in Spyder.

Si eres estudiante de una universidad donde Windows es el sistema operativo dominante y eres usuario de Windows, instala **Anaconda**. Escriba y ejecute programas Python en **Spyder**.

* If you are uncertain how much you will program with Python and primarily want to get a taste of Python programming first, access Python in the cloud, e.g., through the Wakari site.

Si no está seguro de cuánto programará con Python y principalmente desea probar primero la programación en Python, acceda a Python en la nube, por ejemplo, a través del sitio **Wakari**.

* If you want Python on your Mac and you are experienced with compiling and linking software in the Mac OS X environment, install Anaconda on the Mac. Write and run programs in Spyder, or use a text editor like Atom, TextWrangler, Emacs, or Vim, and run programs in the Terminal application. If you are not very familiar with building software on the Mac, and with environment variables like PATH, it will be easier in the long run to access Python in Ubuntu through a virtual machine.

Si desea Python en su Mac y tiene experiencia en la compilación y vinculación de software en el entorno Mac OS X, instale Anaconda en Mac. Escriba y ejecute programas en Spyder, o use un editor de texto como Atom, TextWrangler, Emacs o Vim, y ejecute programas en la aplicación Terminal. Si no está muy familiarizado con la creación de software en Mac y con variables de entorno como PATH, a la larga será más fácil acceder a Python en Ubuntu a través de una **máquina virtual.**

**1.1.4 Writing and Running Your First Python Program**

I assume that you have made a decision on how to access Python, which dictates whether you will be writing programs in a text editor or in an IPython notebook. What you write will be the same – the difference lies in how you run the program. Sections H.2 and H.4 briefly describe how to write programs in a text editor, run them in a terminal window or in Spyder, and how to operate an IPython notebook. I recommend taking a look at that material before proceeding. Open up your chosen text editor and write the following line:

Supongo que ha tomado una decisión sobre cómo acceder a Python, lo que dicta si escribirá programas en un editor de texto o en un **cuaderno IPython**. Lo que escriba será lo mismo; la diferencia radica en cómo ejecuta el programa. Las secciones H.2 y H.4 describen brevemente cómo escribir programas en un editor de texto, ejecutarlos en una ventana de terminal o en Spyder y cómo operar una computadora portátil IPython. Recomiendo echar un vistazo a ese material antes de continuar. Abra el editor de texto elegido y escriba la siguiente línea:

print 5\*0.6 - 0.5\*9.81\*0.6\*\*2

This is a complete Python program for evaluating the formula (1.2). Save the line to a file with name ball1.py.

The action required to run this program depends on what type of tool you use for running programs:

* terminal window: move to the folder where ball1.py is located and type **python ball1.py**
* IPython notebook: click on the “play” button to execute the cell
* Spyder: choose Run from the Run pull-down menu

The output is 1.2342 and appears

* right after the python ball1.py command in a terminal window
* right after the program line (cell) in the IPython notebook
* in the lower right window in Spyder

We remark that there are other ways of running Python programs in the terminal window, see **Appendix H.5**.

Suppose you want to evaluate (1.1) for $v\_0 = 1$ and t = 0.1. This is easy: move the cursor to the editor window, edit the program text to

print 1\*0.1 - 0.5\*9.81\*0.1\*\*2

Run the program again in Spyder or re-execute the cell in an IPython notebook. If you use a plain text editor, always remember to save the file after editing it, then move back to the terminal window and run the program as before:

**Terminal> python ball1.py**

0.05095

The result of the calculation has changed, as expected.

**Typesetting of operating system commands**

We use the prompt Terminal> in this book to indicate commands in a Unix or DOS/PowerShell terminal window. The text following the Terminal> prompt must be a valid operating system command. You will likely see a different prompt in the terminal window on your machine, perhaps something reflecting your username or the current folder.

Usamos el símbolo Terminal> en este libro para indicar comandos en una ventana de terminal Unix o DOS/PowerShell. El texto que sigue al mensaje Terminal> debe ser un comando válido del sistema operativo. Es probable que vea un mensaje diferente en la ventana del terminal de su máquina, tal vez algo que refleje su nombre de usuario o la carpeta actual.

**1.1.5 Warning About Typing Program Text**

Even though a program is just a text, there is one major difference between a text in a program and a text intended to be read by a human. When a human reads a text, she or he is able to understand the message of the text even if the text is not perfectly precise or if there are grammar errors. If our one-line program was expressed as

Aunque un programa es sólo un texto, existe una diferencia importante entre un texto en un programa y un texto destinado a ser leído por un humano. Cuando un humano lee un texto, es capaz de comprender el mensaje del texto incluso si el texto no es perfectamente preciso o si hay errores gramaticales. Si nuestro programa unifilar se expresara como

write 5\*0.6 - 0.5\*9.81\*0.6^2

most humans would interpret write and print as the same thing, and many would also interpret 6^2 as 62. In the Python language, however, write is a grammar error and 6^2 means an operation very different from the exponentiation 6\*\*2. Our communication with a computer through a program must be perfectly precise without a single grammar or logical error. The famous computer scientist Donald Knuth put it this way:

la mayoría de los humanos interpretarían escribir e imprimir como lo mismo, y muchos también interpretarían 6^2 como 62. Sin embargo, en el lenguaje Python, escribir es un error gramatical y 6^2 significa una operación muy diferente de la exponenciación 6\*\* 2. Nuestra comunicación con un ordenador a través de un programa debe ser perfectamente precisa y sin un solo error gramatical o lógico. El famoso informático Donald Knuth lo expresó de esta manera:

Programming demands significantly higher standard of accuracy. Things don’t simply have to make sense to another human being, they must make sense to a computer. Donald Knuth [11, p. 18], 1938-.

That is, the computer will only do exactly what we tell it to do. Any error in the program, however small, may affect the program. There is a chance that we will never notice it, but most often an error causes the program to stop or produce wrong results. The conclusion is that computers have a much more pedantic attitude to language than what (most) humans have.

Es decir, el ordenador sólo hará exactamente lo que le indiquemos. Cualquier error en el programa, por pequeño que sea, puede afectar al programa. Existe la posibilidad de que nunca lo notemos, pero la mayoría de las veces un error hace que el programa se detenga o produzca resultados incorrectos. La conclusión es que las computadoras tienen una actitud mucho más pedante hacia el lenguaje que la que tienen (la mayoría) los humanos.

Now you understand why any program text must be carefully typed, paying attention to the correctness of every character. If you try out program texts from this book, make sure that you type them in exactly as you see them in the book. Blanks, for instance, are often important in Python, so it is a good habit to always count them and type them in correctly. Any attempt not to follow this advice will cause you frustrations, sweat, and maybe even tears.

Ahora comprende por qué el texto de cualquier programa debe escribirse con cuidado, prestando atención a la exactitud de cada carácter. Si prueba los textos del programa de este libro, asegúrese de escribirlos exactamente como los ve en el libro. Los espacios en blanco, por ejemplo, suelen ser importantes en Python, por lo que es un buen hábito contarlos siempre y escribirlos correctamente. Cualquier intento de no seguir este consejo te provocará frustraciones, sudor y tal vez incluso lágrimas.

**1.1.6 Verifying the Result**

We should always carefully control that the output of a computer program is correct. You will experience that in most of the cases, at least until you are an experienced programmer, the output is wrong, and you have to search for errors. In the present application we can simply use a calculator to control the program. Setting t = 0.6 and $v\_0 = 5$ in the formula, the calculator confirms that 1.2342 is the correct solution to our mathematical problem.

Siempre debemos controlar cuidadosamente que la salida de un programa de computadora sea correcta. Experimentarás que en la mayoría de los casos, al menos hasta que seas un programador experimentado, el resultado es incorrecto y tendrás que buscar errores. En la presente aplicación podemos simplemente usar una calculadora para controlar el programa. Estableciendo t = 0,6 y $v\_0 = 5$ en la fórmula, la calculadora confirma que 1,2342 es la solución correcta a nuestro problema matemático.

**1.1.7 Using Variables**

When we want to evaluate y(t) for many values of t, we must modify the t value at two places in our program. Changing another parameter, like v0, is in principle straightforward, but in practice it is easy to modify the wrong number. Such modifications would be simpler to perform if we express our formula in terms of variables, i.e., symbols, rather than numerical values. Most programming languages, Python included, have variables similar to the concept of variables in mathematics. This means that we can define v0, g, t, and y as variables in the program, initialize the former three with numerical values, and combine these three variables to the desired right-hand side expression in (1.1), and assign the result to the variable y. The alternative version of our program, where we use variables, may be written as this text:

Cuando queremos evaluar y(t) para muchos valores de t, debemos modificar el valor de t en dos lugares de nuestro programa. Cambiar otro parámetro, como v0, en principio es sencillo, pero en la práctica es fácil modificar el número incorrecto. Tales modificaciones serían más sencillas de realizar si expresamos nuestra fórmula en **términos de variables**, es decir, **símbolos**, en lugar de valores numéricos. La mayoría de los lenguajes de programación, incluido Python, tienen variables similares al concepto de variables en matemáticas. Esto significa que podemos definir v0, g, t e y como variables en el programa, inicializar las tres primeras con valores numéricos y combinar estas tres variables con la expresión deseada del lado derecho en (1.1) y asignar el resultado. a la variable y. La versión alternativa de nuestro programa, donde usamos variables, se puede escribir como este texto:

v0 = 5

g = 9.81

t = 0.6

y = v0\*t - 0.5\*g\*t\*\*2

print y

Variables in Python are defined by setting a name (here v0, g, t, or y) equal to a numerical value or an expression involving already defined variables.

Las variables en Python se definen estableciendo un nombre (aquí v0, g, t o y) igual a un valor numérico o una expresión que involucra variables ya definidas.

Note that this second program is much easier to read because it is closer to the mathematical notation used in the formula (1.1). The program is also safer to modify, because we clearly see what each number is when there is a name associated with it. In particular, we can change t at one place only (the line t = 0.6) and not two as was required in the previous program.

Tenga en cuenta que este segundo programa es mucho más fácil de leer porque se acerca más a la notación matemática utilizada en la fórmula (1.1). El programa también es más seguro de modificar, porque vemos claramente qué es cada número cuando hay un nombre asociado a él. En particular, podemos cambiar t en un solo lugar (la línea t = 0,6) y no en dos como se requería en el programa anterior.

We store the program text in a file ball2.py. Running the program results in the correct output 1.2342.

**1.1.8 Names of Variables**

Introducing variables with descriptive names, close to those in the mathematical problem we are going to solve, is considered important for the readability and reliability (correctness) of the program. Variable names can contain any lower or upper case letter, the numbers from 0 to 9, and underscore, but the first character cannot be a number. Python distinguishes between upper and lower case, so X is always different from x. Here are a few examples on alternative variable names in the present example:

Introducir variables con nombres descriptivos, cercanos a los del problema matemático que vamos a resolver, se considera importante para la legibilidad y confiabilidad (corrección) del programa. Los nombres de las variables pueden contener cualquier letra minúscula o mayúscula, los números del 0 al 9 y un guión bajo, pero el primer carácter no puede ser un número. Python distingue entre mayúsculas y minúsculas, por lo que X siempre es diferente de x. A continuación se muestran algunos ejemplos de nombres de variables alternativos en el presente ejemplo:

initial\_velocity = 5

acceleration\_of\_gravity = 9.81

TIME = 0.6

VerticalPositionOfBall = initial\_velocity\*TIME - \ 0.5\*acceleration\_of\_gravity\*TIME\*\*2

print VerticalPositionOfBall

With such long variables names, the code for evaluating the formula becomes so long that we have decided to break it into two lines. This is done by a backslash at the very end of the line (make sure there are no blanks after the backslash!).

Con nombres de variables tan largos, el código para evaluar la fórmula se vuelve tan largo que hemos decidido dividirlo en dos líneas. Esto se hace mediante una barra invertida al final de la línea (¡asegúrese de que no queden espacios en blanco después de la barra invertida!).

In this book we shall adopt the convention that variable names have lower case letters where words are separated by an underscore. Whenever the variable represents a mathematical symbol, we use the symbol or a good approximation to it as variable name. For example, y in mathematics becomes y in the program, and v0 in mathematics becomes v0 in the program. A close resemblance between mathematical symbols in the description of the problem and variables names is important for easy reading of the code and for detecting errors. This principle is illustrated by the code snippet above: even if the long variable names explain well what they represent, checking the correctness of the formula for y is harder than in the program that employs the variables v0, g, t, and y0.

En este libro adoptaremos la convención de que los nombres de variables tienen letras minúsculas donde las palabras están separadas por un guión bajo. Siempre que la variable representa un símbolo matemático, usamos el símbolo o una buena aproximación a él como nombre de variable. Por ejemplo, y en matemáticas se convierte en y en el programa y v0 en matemáticas se convierte en v0 en el programa. Una estrecha semejanza entre los símbolos matemáticos en la descripción del problema y los nombres de las variables es importante para facilitar la lectura del código y detectar errores. Este principio se ilustra con el fragmento de código anterior: incluso si los nombres largos de las variables explican bien lo que representan, verificar la exactitud de la fórmula para y es más difícil que en el programa que emplea las variables v0, g, t e y0.

For all variables where there is no associated precise mathematical description and symbol, one must use descriptive variable names which explain the purpose of the variable. For example, if a problem description introduces the symbol D for a force due to air resistance, one applies a variable D also in the program. However, if the problem description does not define any symbol for this force, one must apply a descriptive name, such as air\_resistance, resistance\_force, or drag\_force.

Para todas las variables en las que no existe una descripción matemática precisa ni un símbolo asociado, se deben utilizar nombres de variables descriptivos que expliquen el propósito de la variable. Por ejemplo, si la descripción de un problema introduce el símbolo D para una fuerza debida a la resistencia del aire, se aplica una variable D también en el programa. Sin embargo, si la descripción del problema no define ningún símbolo para esta fuerza, se debe aplicar un nombre descriptivo, como resistencia\_aire, fuerza\_resistencia o fuerza\_arrastre.

**How to choose variable names**

* Use the same variable names in the program as in the mathematical description of the problem you want to solve.
* For all variables without a precise mathematical definition and symbol, use a carefully chosen descriptive name.

**1.1.9 Reserved Words in Python**

Certain words are reserved in Python because they are used to build up the Python language. These reserved words cannot be used as variable names: and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, with, while, and yield. If you wish to use a reserved word as a variable name, it is common to an underscore at the end. For example, if you need a mathematical quantity in the program, you may work with lambda\_ as variable name. See Exercise 1.16 for examples on legal and illegal variable names.

Ciertas palabras están reservadas en Python porque se utilizan para desarrollar el lenguaje Python. Estas palabras reservadas no se pueden utilizar como nombres de variables: y, como, afirmar, romper, clase, continuar, def, del, elif, demás, excepto, Falso, finalmente, para, desde, global, si, importar, en, es, lambda, Ninguno, no local, no, o, pasar, aumentar, devolver, Verdadero, intentar, con, mientras y ceder. Si desea utilizar una palabra reservada como nombre de variable, es común que aparezca un guión bajo al final. Por ejemplo, si necesita una cantidad matemática en el programa, puede trabajar con lambda\_ como nombre de variable. Consulte el ejercicio 1.16 para ver ejemplos de nombres de variables legales e ilegales.

Program files can have freelya chosen name, but stay away from names that coincide with keywords or module names in Python. For instance, do not use math.py, time.py, random.py, os.py, sys.py, while.py, for.py, if.py, class.py, or def.py.

Los archivos de programa pueden tener un nombre elegido libremente, pero manténgase alejado de nombres que coincidan con palabras clave o nombres de módulos en Python. Por ejemplo, no utilice math.py, time.py, random.py, os.py, sys.py, while.py, for.py, if.py, class.py o def.py.

**1.1.10 Comments**

Along with the program statements it is often informative to provide some comments in a natural human language to explain the idea behind the statements. Comments in Python start with the # character, and everything after this character on a line is ignored when the program is run. Here is an example of our program with explanatory comments:

Junto con las declaraciones del programa, suele ser informativo proporcionar algunos comentarios en un lenguaje humano natural para explicar la idea detrás de las declaraciones. Los comentarios en Python comienzan con el carácter # y todo lo que sigue a este carácter en una línea se ignora cuando se ejecuta el programa. Aquí tienes un ejemplo de nuestro programa con comentarios explicativos:

# Program for computing the height of a ball in vertical motion.

v0 = 5 # initial velocity

g = 9.81 # acceleration of gravity

t = 0.6 # time

y = v0\*t - 0.5\*g\*t\*\*2 # vertical position

print y

This program and the initial version in Sect. 1.1.7 are identical when run on the computer, but for a human the latter is easier to understand because of the comments.

Este programa y la versión inicial en la Sección. 1.1.7 son idénticos cuando se ejecutan en la computadora, pero para un humano este último es más fácil de entender debido a los comentarios.

Good comments together with well-chosen variable names are necessary for any program longer than a few lines, because otherwise the program becomes difficult to understand, both for the programmer and others. It requires some practice to write really instructive comments. Never repeat with words what the program statements already clearly express. Use instead comments to provide important information that is not obvious from the code, for example, what mathematical variable names mean, what variables are used for, a quick overview of a set of forthcoming statements, and general ideas behind the problem solving strategy in the code.

Buenos comentarios junto con nombres de variables bien elegidos son necesarios para cualquier programa de más de unas pocas líneas, porque de lo contrario el programa resulta difícil de entender, tanto para el programador como para otros. Se requiere algo de práctica para escribir comentarios realmente instructivos. Nunca repita con palabras lo que las declaraciones del programa ya expresan claramente. Utilice en su lugar comentarios para proporcionar información importante que no sea obvia en el código, por ejemplo, qué significan los nombres de las variables matemáticas, para qué se utilizan las variables, una descripción general rápida de un conjunto de declaraciones futuras e ideas generales detrás de la estrategia de resolución de problemas en el código. código.

**Remark** If you use non-English characters in your comments, Python will complain with error messages like

SyntaxError: Non-ASCII character ’\xc3’ in file ... but no encoding declared; see http://www.python.org/peps/pep-0263.html for details

Non-English characters are allowed if you put the following magic line in the program before such characters are used:

Se permiten caracteres no ingleses si coloca la siguiente línea mágica en el programa antes de utilizar dichos caracteres:

**# -\*- coding: utf-8 -\*-**

(Yes, this is a comment, but it is not ignored by Python!) More information on non-English characters and encodings like UTF-8 is found in Sect. 6.3.5.

**1.1.11 Formatting Text and Numbers**

Instead of just printing the numerical value of y in our introductory program, we now want to write a more informative text, typically something like

En lugar de simplemente imprimir el valor numérico de y en nuestro programa introductorio, ahora queremos escribir un texto más informativo, típicamente algo como

At t=0.6 s, the height of the ball is 1.23 m.

where we also have control of the number of digits (here y is accurate up to centimeters only).

**Printf syntax** The output of the type shown above is accomplished by a print statement combined with some technique for formatting the numbers. The oldest and most widely used such technique is known as printf formatting (originating from the function printf in the C programming language). For a newcomer to programming, the syntax of printf formatting may look awkward, but it is quite easy to learn and very convenient and flexible to work with. The printf syntax is used in a lot of other programming languages as well.

Sintaxis de Printf La salida del tipo mostrado arriba se logra mediante una declaración de impresión combinada con alguna técnica para formatear los números. La técnica más antigua y más utilizada se conoce como formato printf (que se origina en la función printf en el lenguaje de programación C). Para un recién llegado a la programación, la sintaxis del formato printf puede parecer incómoda, pero es bastante fácil de aprender y muy conveniente y flexible para trabajar. La sintaxis printf también se utiliza en muchos otros lenguajes de programación.

The sample output above is produced by this statement using printf syntax:

print ’At t=%g s, the height of the ball is %.2f m.’ % (t, y)

Let us explain this line in detail. The print statement prints a string: everything that is enclosed in quotes (either single: ’, or double: ") denotes a string in Python. The string above is formatted using printf syntax. This means that the string has various “slots”, starting with a percentage sign, here %g and %.2f, where variables in the program can be put in. We have two “slots” in the present case, and consequently two variables must be put into the slots. The relevant syntax is to list the variables inside standard parentheses after the string, separated from the string by a percentage sign.

Expliquemos esta línea en detalle. La declaración print imprime una cadena: todo lo que está entre comillas (ya sea simple: ' o doble: ") denota una cadena en Python. La cadena anterior está formateada usando la sintaxis printf. Esto significa que la cadena tiene varias “ranuras”, comenzando con un signo de porcentaje, aquí %g y %.2f, donde se pueden colocar las variables del programa. En el presente caso tenemos dos “ranuras” y, en consecuencia, se deben colocar dos variables en las ranuras. La sintaxis relevante es para enumerar las variables dentro de paréntesis estándar después de la cadena, separadas de la cadena por un signo de porcentaje.

The first variable, t, goes into the first “slot”. This “slot” has a format specification %g, where the percentage sign marks the slot and the following character, g, is a format specification. The g format instructs the real number to be written as compactly as possible. The next variable, y, goes into the second “slot”. The format specification here is .2f, which means a real number written with two digits after the decimal place. The f in the .2f format stands for float, a short form for floating-point number, which is the term used for a real number on a computer.

La primera variable, t, va al primer “espacio”. Esta “ranura” tiene una especificación de formato %g, donde el signo de porcentaje marca la ranura y el siguiente carácter, g, es una especificación de formato. El formato g indica que el número real se escriba de la forma más compacta posible. La siguiente variable, y, va al segundo “espacio”. La especificación de formato aquí es .2f, lo que significa un número real escrito con dos dígitos después del decimal. La f en el formato .2f significa flotante, una forma abreviada de número de punto flotante, que es el término utilizado para un número real en una computadora.

For completeness we present the whole program, where text and numbers are mixed in the output:

v0 = 5

g = 9.81

t = 0.6

y = v0\*t - 0.5\*g\*t\*\*2

print ’At t=%g s, the height of the ball is %.2f m.’ % (t, y)

The program is found in the file ball\_print1.py in the src/formulas folder of the collection of programs associated with this book.

There are many more ways to specify formats. For example, e writes a number in scientific notation, i.e., with a number between 1 and 10 followed by a power of 10, as in 1:2432 103. On a computer such a number is written in the form 1.2432e-03. Capital E in the exponent is also possible, just replace e by E, with the result 1.2432E-03.

For decimal notation we use the letter f, as in %f, and the output number then appears with digits before and/or after a comma, e.g., 0.0012432 instead of 1.2432E-03. With the g format, the output will use scientific notation for large or small numbers and decimal notation otherwise. This format is normally what gives most compact output of a real number. A lower case g leads to lower case e in scientific notation, while upper case G implies E instead of e in the exponent.

One can also specify the format as 10.4f or 14.6E, meaning in the first case that a float is written in decimal notation with four decimals in a field of width equal to 10 characters, and in the second case a float written in scientific notation with six decimals in a field of width 14 characters.

Here is a list of some important printf format specifications (the program printf\_demo.py exemplifies many of the constructions):

Format Meaning

%s a string

%d an integer

%0xd an integer in a field of with x, padded with leading zeros

%f decimal notation with six decimals

%e compact scientific notation, e in the exponent

%E compact scientific notation, E in the exponent

%g compact decimal or scientific notation (with e)

%G compact decimal or scientific notation (with E)

%xz format z right-adjusted in a field of width x

%-xz format z left-adjusted in a field of width x

%.yz format z with y decimals

%x.yz format z with y decimals in a field of width x

%% the percentage sign % itself

For a complete specification of the possible printf-style format strings, follow the link from the item printf-style formatting in the index2 of the Python Standard Library online documentation.

Para obtener una especificación completa de las posibles cadenas de formato de estilo printf, siga el enlace del elemento formato de estilo printf en el índice 2 de la documentación en línea de la biblioteca estándar de Python.

We may try out some formats by writing more numbers to the screen in our program (the corresponding file is ball\_print2.py):

v0 = 5

g = 9.81

t = 0.6

y = v0\*t - 0.5\*g\*t\*\*2

print """

At t=%f s, a ball with

initial velocity v0=%.3E m/s

is located at the height %.2f m. """ % (t, v0, y)

Observe here that we use a triple-quoted string, recognized by starting and ending with three single or double quotes: ’’’ or """. Triple-quoted strings are used for text that spans several lines.

In the print statement above, we print t in the f format, which by default implies six decimals; v0 is written in the .3E format, which implies three decimals and the number spans as narrow field as possible; and y is written with two decimals in decimal notation in as narrow field as possible. The output becomes

Terminal Terminal> python ball\_print2.py

At t=0.600000 s, a ball with

initial velocity v0=5.000E+00 m/s

is located at the height 1.23 m.

You should look at each number in the output and check the formatting in detail.

**Format string syntax** Python offers all the functionality of the printf format and much more through a different syntax, often known as format string syntax. Let us illustrate this syntax on the one-line output previously used to show the printf construction. The corresponding format string syntax reads

print ’At t={t:g} s, the height of the ball is {y:.2f} m.’.format( t=t, y=y)

The “slots” where variables are inserted are now recognized by curly braces rather than a percentage sign. The name of the variable is listed with an optional colon and format specifier of the same kind as was used for the printf format. The various variables and their values must be listed at the end as shown. This time the “slots” have names so the sequence of variables is not important.

The multi-line example is written as follows in this alternative format:

print """

At t={t:f} s, a ball with

initial velocity v0={v0:.3E} m/s

is located at the height {y:.2f} m.

""".format(t=t, v0=v0, y=y)

**The newline character** We often want a computer program to write out text that spans several lines. In the last example we obtained such output by triple-quoted strings. We could also use ordinary single-quoted strings and a special character for indicating where line breaks should occur. This special character reads \n, i.e., a backslash followed by the letter n. The two print statements

print """y(t) is

the position of

our ball."""

print ’y(t) is\nthe position of\nour ball’

result in identical output:

y(t) is

the position of

our ball.

**1.2 Computer Science Glossary**

It is now time to pick up some important words that programmers use when they talk about programming: algorithm, application, assignment, blanks (whitespace), bug, code, code segment, code snippet, debug, debugging, execute, executable, implement, implementation, input, library, operating system, output, statement, syntax, user, verify, and verification. These words are frequently used in English in lots of contexts, yet they have a precise meaning in computer science.

Program and code are interchangeable terms. A code/program segment is a collection of consecutive statements from a program. Another term with similar meaning is code snippet. Many also use the word application in the same meaning as program and code. A related term is source code, which is the same as the text that constitutes the program. You find the source code of a program in one or more text files. (Note that text files normally have the extension .txt, while program files have an extension related to the programming language, e.g., .py for Python programs. The content of a .py file is, nevertheless, plain text as in a .txt file.)

We talk about running a program, or equivalently executing a program or executing a file. The file we execute is the file in which the program text is stored. This file is often called an executable or an application. The program text may appear in many files, but the executable is just the single file that starts the whole program when we run that file. Running a file can be done in several ways, for instance, by double-clicking the file icon, by writing the filename in a terminal window, or by giving the filename to some program. This latter technique is what we have used so far in this book: we feed the filename to the program python. That is, we execute a Python program by executing another program python, which interprets the text in our Python program file.

The term library is widely used for a collection of generally useful program pieces that can be applied in many different contexts. Having access to good libraries means that you do not need to program code snippets that others have already programmed (most probable in a better way!). There are huge numbers of Python libraries. In Python terminology, the libraries are composed of modules and packages. Section 1.4 gives a first glimpse of the math module, which contains a set of standard mathematical functions for sin x, cos x, ln x, ex , sinh x, sin1 x, etc. Later, you will meet many other useful modules. Packages are just collections of modules. The standard Python distribution comes with a large number of modules and packages, but you can download many more from the Internet, see in particular www.python.org/pypi. Very often, when you encounter a programming task that is likely to occur in many other contexts, you can find a Python module where the job is already done. To mention just one example, say you need to compute how many days there are between two dates.

This is a non-trivial task that lots of other programmers must have faced, so it is not a big surprise that Python comes with a module datetime to do calculations with dates. The recipe for what the computer is supposed to do in a program is called algorithm. In the examples in the first couple of chapters in this book, the algorithms are so simple that we can hardly distinguish them from the program text itself, but later in the book we will carefully set up an algorithm before attempting to implement it in a program. This is useful when the algorithm is much more compact than the resulting program code. The algorithm in the current example consists of three steps:

* initialize the variables v0, g, and t with numerical values,
* evaluate y according to the formula (1.1),
* print the y value to the screen.

The Python program is very close to this text, but some less experienced programmers may want to write the tasks in English before translating them to Python.

The implementation of an algorithm is the process of writing and testing a program. The testing phase is also known as verification: After the program text is written we need to verify that the program works correctly. This is a very important step that will receive substantial attention in the present book. Mathematical software produce numbers, and it is normally quite a challenging task to verify that the numbers are correct.

An error in a program is known as a bug, and the process of locating and removing bugs is called debugging. Many look at debugging as the most difficult and challenging part of computer programming. We have in fact devoted Appendix F to the art of debugging in this book. The origin of the strange terms bug and debugging can be found in Wikipedia3.

Programs are built of statements. There are many types of statements:

v0 = 3

is an assignment statement, while

print y

is a print statement. It is common to have one statement on each line, but it is possible to write multiple statements on one line if the statements are separated by semi-colon. Here is an example:

v0 = 3; g = 9.81; t = 0.6

y = v0\*t - 0.5\*g\*t\*\*2

print y

Although most newcomers to computer programming will think they understand the meaning of the lines in the above program, it is important to be aware of some major differences between notation in a computer program and notation in mathematics. When you see the equality sign = in mathematics, it has a certain interpretation as an equation (x C2 D 5) or a definition (f .x/ D x2 C1). In a computer program, however, the equality sign has a quite different meaning, and it is called an assignment. The right-hand side of an assignment contains an expression, which is a combination of values, variables, and operators. When the expression is evaluated, it results in a value that the variable on the left-hand side will refer to. We often say that the right-hand side value is assigned to the variable on the lefthand side. In the current context it means that we in the first line assign the number 3 to the variable v0, 9.81 to g, and 0.6 to t. In the next line, the right-hand side expression v0\*t - 0.5\*g\*t\*\*2 is first evaluated, and the result is then assigned to the y variable. Consider the assignment statement

y = y + 3

This statement is mathematically false, but in a program it just means that we evaluate the right-hand side expression and assign its value to the variable y. That is, we first take the current value of y and add 3. The value of this operation is assigned to y. The old value of y is then lost. You may think of the = as an arrow, y <- y+3, rather than an equality sign, to illustrate that the value to the right of the arrow is stored in the variable to the left of the arrow. In fact, the R programming language for statistical computing actually applies an arrow, many old languages (like Algol, Simula, and Pascal) used := to explicitly state that we are not dealing with a mathematical equality. An example will illustrate the principle of assignment to a variable:

y = 3

print y

y = y + 4

print y

y = y\*y

print y

Running this program results in three numbers: 3, 7, 49. Go through the program and convince yourself that you understand what the result of each statement becomes. A computer program must have correct syntax, meaning that the text in the program must follow the strict rules of the computer language for constructing statements. For example, the syntax of the print statement is the word print, followed by one or more spaces, followed by an expression of what we want to print (a Python variable, text enclosed in quotes, a number, for instance). Computers are very picky about syntax! For instance, a human having read all the previous pages may easily understand what this program does,

myvar = 5.2

prinnt Myvar

but the computer will find two errors in the last line: prinnt is an unknown instruction and Myvar is an undefined variable. Only the first error is reported (a syntax error), because Python stops the program once an error is found. All errors that Python finds are easy to remove. The difficulty with programming is to remove the rest of the errors, such as errors in formulas or the sequence of operations. Blanks may or may not be important in Python programs. In Sect. 2.1.2 you will see that blanks are in some occasions essential for a correct program. Around = or arithmetic operators, however, blanks do not matter. We could hence write our program from Sect. 1.1.7 as

v0=3;g=9.81;t=0.6;y=v0\*t-0.5\*g\*t\*\*2;print y

This is not a good idea because blanks are essential for easy reading of a program code, and easy reading is essential for finding errors, and finding errors is the difficult part of programming. The recommended layout in Python programs specifies one blank around =, +, and -, and no blanks around \*, /, and \*\*. Note that the blank after print is essential: print is a command in Python and printy is not recognized as any valid command. (Python will complain that printy is an undefined variable.) Computer scientists often use the term whitespace when referring to a blank. (To be more precise, blank is the character produced by the space bar on the keyboard, while whitespace denotes any character(s) that, if printed, do not print ink on the paper: a blank, a tabulator character (produced by backslash followed by t), or a newline character (produced by backslash followed by n). (The newline character is explained in Sect. 1.1.11.)

When we interact with computer programs, we usually provide some information to the program and get some information out. It is common to use the term input data, or just input, for the information that must be known on beforehand. The result from a program is similarly referred to as output data, or just output. In our example, v0, g, and t constitute input, while y is output. All input data must be assigned values in the program before the output can be computed. Input data can be explicitly initialized in the program, as we do in the present example, or the data can be provided by the user through keyboard typing while the program is running (see Chap. 4). Output data can be printed in the terminal window, as in the current example, displayed as graphics on the screen, as done in Sect. 5.3, or stored in a file for later access, as explained in Sect. 4.6.

The word user usually has a special meaning in computer science: It means a human interacting with a program. You are a user of a text editor for writing Python programs, and you are a user of your own programs. When you write programs, it is difficult to imagine how other users will interact with the program. Maybe they provide wrong input or misinterpret the output. Making user-friendly programs is very challenging and depends heavily on the target audience of users. The author had the average reader of the book in mind as a typical user when developing programs for this book.

A central part of a computer is the operating system. This is actually a collection of programs that manages the hardware and software resources on the computer. There are three dominating operating systems today on computers: Windows, Mac OS X, and Linux. In addition, we have Android and iOS for handheld devices. Several versions of Windows have appeared since the 1990s: Windows 95, 98, 2000, ME, XP, Vista, Windows 7, and Windows 8. Unix was invented already in 1970 and comes in many different versions. Nowadays, two open source implementations of Unix, Linux and Free BSD Unix, are most common. The latter forms the core of the Mac OS X operating system on Macintosh machines, while Linux exists in slightly different flavors: Red Hat, Debian, Ubuntu, and OpenSuse to mention the most important distributions. We will use the term Unix in this book as a synonym for all the operating systems that inherit from classical Unix, such as Solaris, Free BSD, Mac OS X, and any Linux variant. As a computer user and reader of this book, you should know exactly what operating system you have.

The user’s interaction with the operation system is through a set of programs. The most widely used of these enable viewing the contents of folders or starting other programs. To interact with the operating system, as a user, you can either issue commands in a terminal window or use graphical programs. For example, for viewing the file contents of a folder you can run the command ls in a Unix terminal window or dir in a DOS (Windows) terminal window. The graphical alternatives are many, some of the most common are Windows Explorer on Windows, Nautilus and Konqueror on Unix, and Finder on Mac. To start a program, it is common to double-click on a file icon or write the program’s name in a terminal window.

1.3 Another Formula: Celsius-Fahrenheit Conversion

Our next example involves the formula for converting temperature measured in Celsius degrees to the corresponding value in Fahrenheit degrees:

$$

F = \frac{9}{5} C + 32 (1.3)

$$

In this formula, C is the amount of degrees in Celsius, and F is the corresponding temperature measured in Fahrenheit. Our goal now is to write a computer program that can compute F from (1.3) when C is known.

1.3.1 Potential Error: Integer Division

**Straightforward coding of the formula** A straightforward attempt at coding the formula (1.3) goes as follows:

C = 21

F = (9/5)\*C + 32

print F

The parentheses around 9/5 are not strictly needed, i.e., (9/5)\*C is computationally identical to 9/5\*C, but parentheses remove any doubt that 9/5\*C could mean 9/(5\*C). Section 1.3.4 has more information on this topic. When run under Python version 2.x, the program prints the value 53. You can find the program in the file c2f\_v1.py in the src/formulas folder in the folder tree of example programs from this book (downloaded from http://hplgit.github. com/scipro-primer). The v1 part of the name stands for version 1. Throughout this book, we will often develop several trial versions of a program, but remove the version number in the final version of the program.

**Verifying the results** Testing the correctness is easy in this case since we can evaluate the formula on a calculator: 9 5 21 C 32 is 69.8, not 53. What is wrong? The formula in the program looks correct!

**Float and integer division** The error in our program above is one of the most common errors in mathematical software and is not at all obvious for a newcomer to programming. In many computer languages, there are two types of divisions: float division and integer division. Float division is what you know from mathematics: 9/5 becomes 1.8 in decimal notation. Integer division a=b with integers (whole numbers) a and b results in an integer that is truncated (or mathematically, rounded down). More precisely, the result is the largest integer c such that bc a. This implies that 9=5 becomes 1 since 15 D 5 9 while 25 D 10 > 9. Another example is 1=5, which becomes 0 since 05 1 (and 15>1). Yet another example is 16=6, which results in 2 (try 26 and 3 6 to convince yourself). Many computer languages, including Fortran, C, C++, Java, and Python version 2, interpret a division operation a/b as integer division if both operands a and b are integers. If either a or b is a real (floating-point) number, a/b implies the standard mathematical float division. Other languages, such as MATLAB and Python version 3, interprets a/b as float division even if both operands are integers, or complex division if one of the operands is a complex number.

The problem with our program is the coding of the formula (9/5)\*C + 32. This formula is evaluated as follows. First, 9/5 is calculated. Since 9 and 5 are interpreted by Python as integers (whole numbers), 9/5 is a division between two integers, and Python version 2 chooses by default integer division, which results in 1. Then 1 is multiplied by C, which equals 21, resulting in 21. Finally, 21 and 32 are added with 53 as result. We shall very soon present a correct version of the temperature conversion program, but first it may be advantageous to introduce a frequently used term in Python programming: object.

1.3.2 Objects in Python

When we write

C = 21

Python interprets the number 21 on the right-hand side of the assignment as an integer and creates an int (for integer) object holding the value 21. The variable C acts as a name for this int object. Similarly, if we write C = 21.0, Python recognizes 21.0 as a real number and therefore creates a float (for floating-point) object holding the value 21.0 and lets C be a name for this object. In fact, any assignment statement has the form of a variable name on the left-hand side and an object on the right-hand side. One may say that Python programming is about solving a problem by defining and changing objects.

At this stage, you do not need to know what an object really is, just think of an int object as a collection, say a storage box, with some information about an integer number. This information is stored somewhere in the computer’s memory, and with the name C the program gets access to this information. The fundamental issue right now is that 21 and 21.0 are identical numbers in mathematics, while in a Python program 21 gives rise to an int object and 21.0 to a float object. There are lots of different object types in Python, and you will later learn how to create your own customized objects. Some objects contain a lot of data, not just an integer or a real number. For example, when we write

print ’A text with an integer %d and a float %f’ % (2, 2.0)

a str (string) object, without a name, is first made of the text between the quotes and then this str object is printed. We can alternatively do this in two steps:

s = ’A text with an integer %d and a float %f’ % (2, 2.0)

print s

1.3.3 Avoiding Integer Division

As a quite general rule of thumb, one should be careful to avoid integer division when programming mathematical formulas. In the rare cases when a mathematical algorithm does make use of integer division, one should use a double forward slash, //, as division operator, because this is Python’s way of explicitly indicating integer division. Python version 3 has no problem with unintended integer division, so the problem only arises with Python version 2 (and many other common languages for scientific computing). There are several ways to avoid integer division with the plain / operator. The simplest remedy in Python version 2 is to write

from \_\_future\_\_ import division

This import statement must be present in the beginning of every file where the / operator always shall imply float division. Alternatively, one can run a Python program someprogram.py from the command line with the argument -Qnew to the Python interpreter:

Terminal Terminal> python -Qnew someprogram.py

A more widely applicable method, also in other programming languages than Python version 2, is to enforce one of the operands to be a float object. In the current example, there are several ways to do this:

F = (9.0/5)\*C + 32

F = (9/5.0)\*C + 32

F = float(C)\*9/5 + 32

In the first two lines, one of the operands is written as a decimal number, implying a float object and hence float division. In the last line, float(C)\*9 means float times int, which results in a float object, and float division is guaranteed. A related construction,

F = float(C)\*(9/5) + 32

does not work correctly, because 9/5 is evaluated by integer division, yielding 1, before being multiplied by a float representation of C (see next section for how compound arithmetic operations are calculated). In other words, the formula reads F=C+32, which is wrong. We now understand why the first version of the program does not work and what the remedy is. A correct program is

C = 21

F = (9.0/5)\*C + 32

print F

Instead of 9.0 we may just write 9. (the dot implies a float interpretation of the number). The program is available in the file c2f.py. Try to run it – and observe that the output becomes 69.8, which is correct.

**Locating potential integer division** Running a Python program with the -Qwarnall argument, say

Terminal Terminal> python -Qwarnall someprogram.py

will print out a warning every time an integer division expression is encountered in Python version 2.

**Remark** We could easily have run into problems in our very first programs if we instead of writing the formula 1 2gt 2 as 0.5\*g\*t\*\*2 wrote (1/2)\*g\*t\*\*2. This term would then always be zero

1.3.4 Arithmetic Operators and Precedence

Formulas in Python programs are usually evaluated in the same way as we would evaluate them mathematically. Python proceeds from left to right, term by term in an expression (terms are separated by plus or minus). In each term, power operations such as ab, coded as a\*\*b, has precedence over multiplication and division. As in mathematics, we can use parentheses to dictate the way a formula is evaluated. Below are two illustrations of these principles.

* 5/9+2\*a\*\*4/2: First 5/9 is evaluated (as integer division, giving 0 as result), then a4 (a\*\*4) is evaluated, then 2 is multiplied with a4, that result is divided by 2, and the answer is added to the result of the first term. The answer is therefore a\*\*4.
* 5/(9+2)\*a\*\*(4/2): First 5 9C2 is evaluated (as integer division, yielding 0), then 4/2 is computed (as integer division, yielding 2), then a\*\*2 is calculated, and that number is multiplied by the result of 5/(9+2). The answer is thus always zero.

As evident from these two examples, it is easy to unintentionally get integer division in formulas. Although integer division can be turned off in Python, we think it is important to be strongly aware of the integer division concept and to develop good programming habits to avoid it. The reason is that this concept appears in so many common computer languages that it is better to learn as early as possible how to deal with the problem rather than using a Python-specific feature to remove the problem.

1.4 Evaluating Standard Mathematical Functions

Mathematical formulas frequently involve functions such as sin, cos, tan, sinh, cosh, exp, log, etc. On a pocket calculator you have special buttons for such functions. Similarly, in a program you also have ready-made functionality for evaluating these types of mathematical functions. One could in principle write one’s own program for evaluating, e.g., the sin.x/ function, but how to do this in an efficient way is a non-trivial topic. Experts have worked on this problem for decades and implemented their best recipes in pieces of software that we should reuse. This section tells you how to reach sin, cos, and similar functions in a Python context.

1.4.1 Example: Using the Square Root Function

**Problem** Consider the formula for the height y of a ball in vertical motion, with initial upward velocity v0:

yc D v0t 1 2 gt 2 ;

where g is the acceleration of gravity and t is time. We now ask the question: How long time does it take for the ball to reach the height yc ? The answer is straightforward to derive. When y D yc we have

yc D v0t 1 2 gt

We recognize that this equation is a quadratic equation, which we must solve with respect to t. Rearranging,

1 2 gt 2 v0t C yc D 0;

and using the well-known formula for the two solutions of a quadratic equation, we find

t1 D v0 q v2 0 2gyc =g; t2 D v0 C q v2 0 2gyc =g : (1.4)

There are two solutions because the ball reaches the height yc on its way up .t D t1) and on its way down (t D t2 > t1).

**The program** To evaluate the expressions for t1 and t2 from (1.4) in a computer program, we need access to the square root function. In Python, the square root function and lots of other mathematical functions, such as sin, cos, sinh, exp, and log, are available in a module called math. We must first import the module before we can use it, that is, we must write import math. Thereafter, to take the square root of a variable a, we can write math.sqrt(a). This is demonstrated in a program for computing t1 and t2:

v0 = 5

g = 9.81

yc = 0.2

import math

t1 = (v0 - math.sqrt(v0\*\*2 - 2\*g\*yc))/g

t2 = (v0 + math.sqrt(v0\*\*2 - 2\*g\*yc))/g

print ’At t=%g s and %g s, the height is %g m.’ % (t1, t2, yc)

The output from this program becomes

At t=0.0417064 s and 0.977662 s, the height is 0.2 m.

You can find the program as the file ball\_yc.py in the src/formulas folder.

**Two ways of importing a module** The standard way to import a module, say math, is to write

import math

and then access individual functions in the module with the module name as prefix as in

x = math.sqrt(y)

People working with mathematical functions often find math.sqrt(y) less pleasing than just sqrt(y). Fortunately, there is an alternative import syntax that allows us to skip the module name prefix. This alternative syntax has the form from module import function. A specific example is

from math import sqrt

Now we can work with sqrt directly, without the math. prefix. More than one function can be imported:

from math import sqrt, exp, log, sin

Sometimes one just writes

from math import \*

to import all functions in the math module. This includes sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, exp, log (base e), log10 (base 10), sqrt, as well as the famous numbers e and pi. Importing all functions from a module, using the asterisk (\*) syntax, is convenient, but this may result in a lot of extra names in the program that are not used. It is in general recommended not to import more functions than those that are really used in the program. Nevertheless, the convenience of the compact from math import \* syntax occasionally wins over the general recommendation among practitioners – and in this book. With a from math import sqrt statement we can write the formulas for the roots in a more pleasing way:

t1 = (v0 - sqrt(v0\*\*2 - 2\*g\*yc))/g

t2 = (v0 + sqrt(v0\*\*2 - 2\*g\*yc))/g

**Import with new names** Imported modules and functions can be given new names in the import statement, e.g.,

import math as m

# m is now the name of the math module

v = m.sin(m.pi)

from math import log as ln

v = ln(5)

from math import sin as s, cos as c, log as ln

v = s(x)\*c(x) + ln(x)

In Python, everything is an object, and variables refer to objects, so new variables may refer to modules and functions as well as numbers and strings. The examples above on new names can also be coded by introducing new variables explicitly:

m = math

ln = m.log

s = m.sin

c = m.cos

1.4.2 Example: Computing with sinh x

Our next examples involve calling some more mathematical functions from the math module. We look at the definition of the sinh.x/ function:

sinh.x/ D 1 2 .ex ex/ : (1.5)

We can evaluate sinh.x/ in three ways: i) by calling math.sinh, ii) by computing the right-hand side of (1.5), using math.exp, or iii) by computing the right-hand side of (1.5) with the aid of the power expressions math.e\*\*x and math.e\*\*(-x). A program doing these three alternative calculations is found in the file 3sinh.py. The core of the program looks like this:

from math import sinh, exp, e, pi

x = 2\*pi

r1 = sinh(x)

r2 = 0.5\*(exp(x) - exp(-x))

r3 = 0.5\*(e\*\*x - e\*\*(-x))

print r1, r2, r3

The output from the program shows that all three computations give identical results:

267.744894041 267.744894041 267.744894041

1.4.3 A First Glimpse of Rounding Errors

The previous example computes a function in three different yet mathematically equivalent ways, and the output from the print statement shows that the three resulting numbers are equal. Nevertheless, this is not the whole story. Let us try to print out r1, r2, r3 with 16 decimals:

print ’%.16f %.16f %.16f’ % (r1,r2,r3)

This statement leads to the output

267.7448940410164369 267.7448940410164369 267.7448940410163232

Now r1 and r2 are equal, but r3 is different! Why is this so? Our program computes with real numbers, and real numbers need in general an infinite number of decimals to be represented exactly. The computer truncates the sequence of decimals because the storage is finite. In fact, it is quite standard to keep only 17 digits in a real number on a computer. Exactly how this truncation is done is not explained in this book, but you read more on Wikipedia4. For now the purpose is to notify the reader that real numbers on a computer often have a small error. Only a few real numbers can be represented exactly, the rest of the real numbers are only approximations. For this reason, most arithmetic operations involve inaccurate real numbers, resulting in inaccurate calculations. Think of the following two calculations: 1=4949 and 1=51 51. Both expressions are identical to 1, but when we perform the calculations in Python,

print ’%.16f %.16f’ % (1/49.0\*49, 1/51.0\*51)

the result becomes

0.9999999999999999 1.0000000000000000

The reason why we do not get exactly 1.0 as answer in the first case is because 1/49 is not correctly represented in the computer. Also 1/51 has an inexact representation, but the error does not propagate to the final answer. To summarize, errors in floating-point numbers may propagate through mathematical calculations and result in answers that are only approximations to the exact underlying mathematical values. The errors in the answers are commonly known as rounding errors. As soon as you use Python interactively as explained in the next section, you will encounter rounding errors quite often. Python has a special module decimal and the SymPy package has an alternative module mpmath, which allow real numbers to be represented with adjustable accuracy so that rounding errors can be made as small as desired (an example appears at the end of Sect. 3.1.12). However, we will hardly use such modules because approximations implied by many mathematical methods applied throughout this book normally lead to (much) larger errors than those caused by rounding.

1.5 Interactive Computing

A particular convenient feature of Python is the ability to execute statements and evaluate expressions interactively. The environments where you work interactively with programming are commonly known as Python shells. The simplest Python shell is invoked by just typing python at the command line in a terminal window. Some messages about Python are written out together with a prompt >>>, after which you can issue commands. Let us try to use the interactive shell as a calculator. Type in 3\*4.5-0.5 and then press the Return key to see Python’s response to this expression:

Terminal> python

Python 2.7.5+ (default, Sep 19 2013, 13:48:49) [GCC 4.8.1] on linux2 Type "help", "copyright", "credits" or "license" for more information.

>>> 3\*4.5-0.5

13.0

The text on a line after >>> is what we write (shell input) and the text without the >>> prompt is the result that Python calculates (shell output). It is easy, as explained below, to recover previous input and edit the text. This editing feature makes it convenient to experiment with statements and expressions.

1.5.1 Using the Python Shell

The program from Sect. 1.1.7 can be typed in line by line in the interactive shell:

>>> v0 = 5

>>> g = 9.81

>>> t = 0.6

>>> y = v0\*t - 0.5\*g\*t\*\*2

>>> print y

1.2342

We can now easily calculate an y value corresponding to another (say) v0 value: hit the up arrow key to recover previous statements, repeat pressing this key until the v0 = 5 statement is displayed. You can then edit the line, e.g., to

>>> v0 = 6

Press return to execute this statement. You can control the new value of v0 by either typing just v0 or print v0:

>>> v0

6

>>> print v0

6

The next step is to recompute y with this new v0 value. Hit the up arrow key multiple times to recover the statement where y is assigned, press the Return key, and write y or print y to see the result of the computation:

>>> y = v0\*t - 0.5\*g\*t\*\*2

>>> y

1.8341999999999996

>>> print y

1.8342

The reason why we get two slightly different results is that typing just y prints out all the decimals that are stored in the computer (16), while print y writes out y with fewer decimals. As mentioned in Sect. 1.4.3 computations on a computer often suffer from rounding errors. The present calculation is no exception. The correct answer is 1.8342, but rounding errors lead to a number that is incorrect in the 16th decimal. The error is here 4 1016.

1.5.2 Type Conversion

Often you can work with variables in Python without bothering about the type of objects these variables refer to. Nevertheless, we encountered a serious problem in Sect. 1.3.1 with integer division, which forced us to be careful about the types of objects in a calculation. The interactive shell is very useful for exploring types. The following example illustrates the type function and how we can convert an object from one type to another. First, we create an int object bound to the name C and check its type by calling type(C):

>>> C = 21

>>> type(C)

We convert this int object to a corresponding float object:

>>> C = float(C) # type conversion

>>> type(C) >>> C 21.0 In the statement C = float(C) we create a new object from the original object referred to by the name C and bind it to the same name C. That is, C refers to a different object after the statement than before. The original int with value 21 cannot be reached anymore (since we have no name for it) and will be automatically deleted by Python. We may also do the reverse operation, i.e., convert a particular float object to a corresponding int object: >>> C = 20.9 >>> type(C) >>> D = int(C) # type conversion >>> type(D) >>> D 20 # decimals are truncated :-/ In general, one can convert a variable v to type MyType by writing v=MyType(v), if it makes sense to do the conversion. In the last input we tried to convert a float to an int, and this operation implied stripping off the decimals. Correct conversion according to mathematical rounding rules can be achieved with help of the round function: >>> round(20.9) 21.0 >>> int(round(20.9)) 21

1.5.3 IPython There exist several improvements of the standard Python shell presented in Sect. 1.5. The author advocates IPython as the preferred interactive shell. You will then need to have IPython installed. Typing ipython in a terminal window starts the shell. The (default) prompt in IPython is not >>> but In [X]:, where X is the number of the present input command. The most widely used features of IPython are summarized below. Running programs Python programs can be run from within the shell: In [1]: run ball2.py 1.2342 This command requires that you have taken a cd to the folder where the ball2.py program is located and started IPython from there. On Windows you may, as an alternative to starting IPython from a DOS or PowerShell window, double click on the IPython desktop icon or use the Start menu. In that case, you must move to the right folder where your program is located. This is done by the os.chdir (change directory) command. Typically, you write something like In [1]: import os In [2]: os.chdir(r’C:\Documents and Settings\me\My Documents\div’) In [3]: run ball2.py if the ball2.py program is located in the folder div under My Documents of user me. Note the r before the quote in the string: it is required to let a backslash really mean the backslash character. If you end up typing the os.chdir command every time you enter an IPython shell, this command (and others) can be placed in a startup file such that they are automatically executed when you launch IPython. Inside IPython you can invoke any operating system command. This allows us to navigate to the right folder above using Unix or Windows (cd) rather than Python (os.chdir): In [1]: cd C:\Documents and Settings\me\My Documents\div In [3]: run ball2.py We recommend running all your Python programs from the IPython shell. Especially when something goes wrong, IPython can help you to examine the state of variables so that you become quicker to locate bugs. Typesetting convention for executing Python programs In the rest of the book, we just write the program name and the output when we illustrate the execution of a program: Terminal ball2.py 1.2342

You then need to write run before the program name if you execute the program in IPython, or if you prefer to run the program directly in a terminal window, you need to write python prior to the program name. Appendix H.5 describes various other ways to run a Python program. Quick recovery of previous output The results of the previous statements in an interactive IPython session are available in variables of the form \_iX (underscore, i, and a number X), where X is 1 for the last statement, 2 for the second last statement, and so forth. Short forms are \_ for \_i1, \_\_ for \_i2, and \_\_\_ for \_i3. The output from the In [1] input above is 1.2342. We can now refer to this number by an underscore and, e.g., multiply it by 10: In [2]: \_\*10 Out[2]: 12.341999999999999 Output from Python statements or expressions in IPython are preceded by Out[X] where X is the command number corresponding to the previous In [X] prompt. When programs are executed, as with the run command, or when operating system commands are run (as shown below), the output is from the operating system and then not preceded by any Out[X] label. The command history from previous IPython sessions is available in a new session. This feature makes it easy to modify work from a previous session by just hitting the up-arrow to recall commands and edit them as necessary. Tab completion Pressing the TAB key will complete an incompletely typed variable name. For example, after defining my\_long\_variable\_name = 4, write just my at the In [4]: prompt below, and then hit the TAB key. You will experience that my is immediately expanded to my\_long\_variable\_name. This automatic expansion feature is called TAB completion and can save you from quite some typing. In [3]: my\_long\_variable\_name = 4 In [4]: my\_long\_variable\_name Out[4]: 4 Recovering previous commands You can walk through the command history by typing Ctrl+p or the up arrow for going backward or Ctrl+n or the down arrow for going forward. Any command you hit can be edited and re-executed. Also commands from previous interactive sessions are stored in the command history. Running Unix/Windows commands Operating system commands can be run from IPython. Below we run the three Unix commands date, ls (list files), mkdir (make directory), and cd (change directory): In [5]: date Thu Nov 18 11:06:16 CET 2010 In [6]: ls myfile.py yourprog.py

In [7]: mkdir mytestdir In [8]: cd mytestdir If you have defined Python variables with the same name as operating system commands, e.g., date=30, you must write !date to run the corresponding operating system command. IPython can do much more than what is shown here, but the advanced features and the documentation of them probably do not make sense before you are more experienced with Python – and with reading manuals. Typesetting of interactive shells in this book In the rest of the book we will apply the >>> prompt in interactive sessions instead of the input and output prompts as used by default by IPython, simply because most Python books and electronic manuals use >>> to mark input in interactive shells. However, when you sit by the computer and want to use an interactive shell, we recommend using IPython, and then you will see the In [X] prompt instead of >>>. Notebooks A particularly interesting feature of IPython is the notebook, which allows you to record and replay exploratory interactive sessions with a mix of text, mathematics, Python code, and graphics. See Sect. H.4 for a quick introduction to IPython notebooks. 1.6 Complex Numbers Suppose x2 D 2. Then most of us are able to find out that x D p2 is a solution to the equation. The more mathematically interested reader will also remark that x D p2 is another solution. But faced with the equation x2 D 2, very few are able to find a proper solution without any previous knowledge of complex numbers. Such numbers have many applications in science, and it is therefore important to be able to use such numbers in our programs. On the following pages we extend the previous material on computing with real numbers to complex numbers. The text is optional, and readers without knowledge of complex numbers can safely drop this section and jump to Sect. 1.8. A complex number is a pair of real numbers a and b, most often written as aCbi, or a C ib, where i is called the imaginary unit and acts as a label for the second term. Mathematically, i D p1. An important feature of complex numbers is definitely the ability to compute square roots of negative numbers. For example, p2 D p2i (i.e., p2 p1). The solutions of x2 D 2 are thus x1 D Cp2i and x2 D p2i. There are rules for addition, subtraction, multiplication, and division between two complex numbers. There are also rules for raising a complex number to a real power, as well as rules for computing sin z, cos z, tan z, ez , ln z, sinh z, cosh z, tanh z, etc. for a complex number z D a C ib. We assume in the following that you are familiar with the mathematics of complex numbers, at least to the degree

encountered in the program examples. let u D a C bi and v D c C d i The following rules reflect complex arithmetics: u D v ) a D c; b D d u D a bi u a bi (complex conjugate) u C v D .a C c/ C .b C d /i u v D .a c/ C .b d /i uv D .ac bd/ C .bc C ad /i u=v D ac C bd c2 C d2 C bc ad c2 C d2 i juj D p a2 C b2 ei q D cos q C i sin q 1.6.1 Complex Arithmetics in Python Python supports computation with complex numbers. The imaginary unit is written as j in Python, instead of i as in mathematics. A complex number 23i is therefore expressed as (2-3j) in Python. We remark that the number i is written as 1j, not just j. Below is a sample session involving definition of complex numbers and some simple arithmetics: >>> u = 2.5 + 3j # create a complex number >>> v = 2 # this is an int >>> w = u + v # complex + int >>> w (4.5+3j) >>> a = -2 >>> b = 0.5 >>> s = a + b\*1j # create a complex number from two floats >>> s = complex(a, b) # alternative creation >>> s (-2+0.5j) >>> s\*w # complex\*complex (-10.5-3.75j) >>> s/w # complex/complex (-0.25641025641025639+0.28205128205128205j) A complex object s has functionality for extracting the real and imaginary parts as well as computing the complex conjugate:

>>> s.real -2.0 >>> s.imag 0.5 >>> s.conjugate() (-2-0.5j) 1.6.2 Complex Functions in Python Taking the sine of a complex number does not work: >>> from math import sin >>> r = sin(w) Traceback (most recent call last): File "", line 1, in ? TypeError: can’t convert complex to float; use abs(z) The reason is that the sin function from the math module only works with real (float) arguments, not complex. A similar module, cmath, defines functions that take a complex number as argument and return a complex number as result. As an example of using the cmath module, we can demonstrate that the relation sin.ai / D i sinh a holds: >>> from cmath import sin, sinh >>> r1 = sin(8j) >>> r1 1490.4788257895502j >>> r2 = 1j\*sinh(8) >>> r2 1490.4788257895502j Another relation, ei q D cos q C i sin q, is exemplified next: >>> q = 8 # some arbitrary number >>> exp(1j\*q) (-0.14550003380861354+0.98935824662338179j) >>> cos(q)+1j\*sin(q) (-0.14550003380861354+0.98935824662338179j) 1.6.3 Unified Treatment of Complex and Real Functions The cmath functions always return complex numbers. It would be nice to have functions that return a float object if the result is a real number and a complex object if the result is a complex number. The Numerical Python package has such versions of the basic mathematical functions known from math and cmath. By taking a from numpy.lib.scimath import \*

one obtains access to these flexible versions of mathematical functions. The functions also get imported by any of the statements from scipy import \* from scitools.std import \* A session will illustrate what we obtain. Let us first use the sqrt function in the math module: >>> from math import sqrt >>> sqrt(4) # float 2.0 >>> sqrt(-1) # illegal Traceback (most recent call last): File "", line 1, in ? ValueError: math domain error If we now import sqrt from cmath, >>> from cmath import sqrt the previous sqrt function is overwritten by the new one. More precisely, the name sqrt was previously bound to a function sqrt from the math module, but is now bound to another function sqrt from the cmath module. In this case, any square root results in a complex object: >>> sqrt(4) # complex (2+0j) >>> sqrt(-1) # complex 1j If we now take >>> from numpy.lib.scimath import \* we import (among other things) a new sqrt function. This function is slower than the versions from math and cmath, but it has more flexibility since the returned object is float if that is mathematically possible, otherwise a complex is returned: >>> sqrt(4) # float 2.0 >>> sqrt(-1) # complex 1j As a further illustration of the need for flexible treatment of both complex and real numbers, we may code the formulas for the roots of a quadratic function f .x/ D ax2 C bx C c:

>>> a = 1; b = 2; c = 100 # polynomial coefficients >>> from numpy.lib.scimath import sqrt >>> r1 = (-b + sqrt(b\*\*2 - 4\*a\*c))/(2\*a) >>> r2 = (-b - sqrt(b\*\*2 - 4\*a\*c))/(2\*a) >>> r1 (-1+9.94987437107j) >>> r2 (-1-9.94987437107j) Using the up arrow, we may go back to the definitions of the coefficients and change them so the roots become real numbers: >>> a = 1; b = 4; c = 1 # polynomial coefficients Going back to the computations of r1 and r2 and performing them again, we get >>> r1 -0.267949192431 >>> r2 -3.73205080757 That is, the two results are float objects. Had we applied sqrt from cmath, r1 and r2 would always be complex objects, while sqrt from the math module would not handle the first (complex) case. 1.7 Symbolic Computing Python has a package SymPy for doing symbolic computing, such as symbolic (exact) integration, differentiation, equation solving, and expansion of Taylor series, to mention some common operations in mathematics. We shall here only give a glimpse of SymPy in action with the purpose of drawing attention to this powerful part of Python. For interactive work with SymPy it is recommended to either use IPython or the special, interactive shell isympy, which is installed along with SymPy itself. Below we shall explicitly import each symbol we need from SymPy to emphasize that the symbol comes from that package. For example, it will be important to know whether sin means the sine function from the math module, aimed at real numbers, or the special sine function from sympy, aimed at symbolic expressions. 1.7.1 Basic Differentiation and Integration The following session shows how easy it is to differentiate a formula v0t 1 2gt 2 with respect to t and integrate the answer to get the formula back:

>>> from sympy import ( ... symbols, # define symbols for symbolic math ... diff, # differentiate expressions ... integrate, # integrate expressions ... Rational, # define rational numbers ... lambdify, # turn symbolic expr. into Python functions ... ) >>> t, v0, g = symbols(’t v0 g’) >>> y = v0\*t - Rational(1,2)\*g\*t\*\*2 >>> dydt = diff(y, t) >>> dydt -g\*t + v0 >>> print ’acceleration:’, diff(y, t, t) # 2nd derivative acceleration: -g >>> y2 = integrate(dydt, t) >>> y2 -g\*t\*\*2/2 + t\*v0 Note here that t is a symbolic variable (not a float as it is in numerical computing), and y (like y2) is a symbolic expression (not a float as it would be in numerical computing). A very convenient feature of SymPy is that symbolic expressions can be turned into ordinary Python functions via lambdify. (Python functions are introduced in Chap. 3, but when discussing SymPy here in the present chapter, it is very natural to explain how lambdify can transform symbolic expressions back to ordinary numerical Python expressions.) Let us take the dydt expression above and turn it into a Python function v(t, v0, g) for numerical computing: >>> v = lambdify([t, v0, g], # arguments in v dydt) # symbolic expression >>> v(t=0, v0=5, g=9.81) 5 >>> v(2, 5, 9.81) -14.62 >>> 5 - 9.81\*2 # control the previous calculation -14.62 1.7.2 Equation Solving A linear equation defined through an expression e that is zero, can be solved by solve(e, t), if t is the unknown (symbol) in the equation. Here we may find the roots of y D 0: >>> from sympy import solve >>> roots = solve(y, t) >>> roots [0, 2\*v0/g] We can easily check the answer by inserting the roots in y. Inserting an expression e2 for e1 in some expression e is done by e.subs(e1, e2). In our case we check that

>>> y.subs(t, roots[0]) 0 >>> y.subs(t, roots[1]) 0 1.7.3 Taylor Series and More A Taylor polynomial of order n for an expression e in a variable t around the point t0 is computed by e.series(t, t0, n). Testing this on et and esin.t / gives >>> from sympy import exp, sin, cos >>> f = exp(t) >>> f.series(t, 0, 3) 1 + t + t\*\*2/2 + O(t\*\*3) >>> f = exp(sin(t)) >>> f.series(t, 0, 8) 1 + t + t\*\*2/2 - t\*\*4/8 - t\*\*5/15 - t\*\*6/240 + t\*\*7/90 + O(t\*\*8) Output of mathematical expressions in the LATEX typesetting system is possible: >>> from sympy import latex >>> print latex(f.series(t, 0, 7)) ’1 + t + \frac{t^{2}}{2} - \frac{t^{4}}{8} - \frac{t^{5}}{15} - \frac{t^{6}}{240} + \mathcal{O}\left(t^{7}\right)’ Finally, we mention that there are tools for expanding and simplifying expressions: >>> from sympy import simplify, expand >>> x, y = symbols(’x y’) >>> f = -sin(x)\*sin(y) + cos(x)\*cos(y) >>> simplify(f) cos(x + y) >>> expand(sin(x+y), trig=True) # requires a trigonometric hint sin(x)\*cos(y) + sin(y)\*cos(x) Later chapters utilize SymPy where it can save some algebraic work, but this book is almost exclusively devoted to numerical computing. 1.8 Summary 1.8.1 Chapter Topics Programs must be accurate! A program is a collection of statements stored in a text file. Statements can also be executed interactively in a Python shell. Any error in any statement may lead to termination of the execution or wrong results. The computer does exactly what the programmer tells the computer to do!

Variables The statement some\_variable = obj defines a variable with the name some\_variable which refers to an object obj. Here obj may also represent an expression, say a formula, whose value is a Python object. For example, 1+2.5 involves the addition of an int object and a float object, resulting in a float object. Names of variables can contain upper and lower case English letters, underscores, and the digits from 0 to 9, but the name cannot start with a digit. Nor can a variable name be a reserved word in Python. If there exists a precise mathematical description of the problem to be solved in a program, one should choose variable names that are in accordance with the mathematical description. Quantities that do not have a defined mathematical symbol, should be referred to by descriptive variables names, i.e., names that explain the variable’s role in the program. Well-chosen variable names are essential for making a program easy to read, easy to debug, and easy to extend. Well-chosen variable names also reduce the need for comments. Comment lines Everything after # on a line is ignored by Python and used to insert free running text, known as comments. The purpose of comments is to explain, in a human language, the ideas of (several) forthcoming statements so that the program becomes easier to understand for humans. Some variables whose names are not completely self-explanatory also need a comment. Object types There are many different types of objects in Python. In this chapter we have worked with the following types. Integers (whole numbers, object type int): x10 = 3 XYZ = 2 Floats (decimal numbers, object type float): max\_temperature = 3.0 MinTemp = 1/6.0 Strings (pieces of text, object type str): a = ’This is a piece of text\nover two lines.’ b = "Strings are enclosed in single or double quotes." c = """Triple-quoted strings can span several lines. """ Complex numbers (object type complex): a = 2.5 + 3j real = 6; imag = 3.1 b = complex(real, imag)

Operators Operators in arithmetic expressions follow the rules from mathematics: power is evaluated before multiplication and division, while the latter two are evaluated before addition and subtraction. These rules are overridden by parentheses. We suggest using parentheses to group and clarify mathematical expressions, also when not strictly needed. -t\*\*2\*g/2 -(t\*\*2)\*(g/2) # equivalent -t\*\*(2\*g)/2 # a different formula! a = 5.0; b = 5.0; c = 5.0 a/b + c + a\*c # yields 31.0 a/(b + c) + a\*c # yields 25.5 a/(b + c + a)\*c # yields 1.6666666666666665 Particular attention must be paid to coding fractions, since the division operator / often needs extra parentheses that are not necessary in the mathematical notation for fractions (compare a bCc with a/(b+c) and a/b+c). Common mathematical functions The math module contains common mathematical functions for real numbers. Modules must be imported before they can be used. The three types of alternative module import go as follows: # Import of module - functions requires prefix import math a = math.sin(math.pi\*1.5) # Import of individual functions - no prefix in function calls from math import sin, pi a = sin(pi\*1.5) # Import everything from a module - no prefix in function calls from math import \* a = sin(pi\*1.5) Print To print the result of calculations in a Python program to a terminal window, we apply the print command, i.e., the word print followed by a string enclosed in quotes, or just a variable: print "A string enclosed in double quotes" print a Several objects can be printed in one statement if the objects are separated by commas. A space will then appear between the output of each object: >>> a = 5.0; b = -5.0; c = 1.9856; d = 33 >>> print ’a is’, a, ’b is’, b, ’c and d are’, c, d a is 5.0 b is -5.0 c and d are 1.9856 33

The printf syntax enables full control of the formatting of real numbers and integers: >>> print ’a=%g, b=%12.4E, c=%.2f, d=%5d’ % (a, b, c, d) a=5, b= -5.0000E+00, c=1.99, d= 33 Here, a, b, and c are of type float and formatted as compactly as possible (%g for a), in scientific notation with 4 decimals in a field of width 12 (%12.4E for b), and in decimal notation with two decimals in as compact field as possible (%.2f for c). The variable d is an integer (int) written in a field of width 5 characters (%5d). Be careful with integer division! A common error in mathematical computations is to divide two integers, because this results in integer division (in Python 2). Any number written without decimals is treated as an integer. To avoid integer division, ensure that every division involves at least one real number, e.g., 9/5 is written as 9.0/5, 9./5, 9/5., or 9/5.0. In expressions with variables, a/b, ensure that a or b is a float object, and if not (or uncertain), do an explicit conversion as in float(a)/b to guarantee float division. If integer division is desired, use a double slash: a//b. Python 3 treats a/b as float division also when a and b are integers. Complex numbers Values of complex numbers are written as (X+Yj), where X is the value of the real part and Y is the value of the imaginary part. One example is (4-0.2j). If the real and imaginary parts are available as variables r and i, a complex number can be created by complex(r, i). The cmath module must be used instead of math if the argument is a complex variable. The numpy package offers similar mathematical functions, but with a unified treatment of real and complex variables. Terminology Some Python and computer science terms briefly covered in this chapter are object: anything that a variable (name) can refer to, such as a number, string, function, or module (but objects can exist without being bound to a name: print ’Hello!’ first makes a string object of the text in quotes and then the contents of this string object, without a name, is printed) variable: name of an object statement: an instruction to the computer, usually written on a line in a Python program (multiple statements on a line must be separated by semicolons) expression: a combination of numbers, text, variables, and operators that results in a new object, when being evaluated assignment: a statement binding an evaluated expression (object) to a variable (name) algorithm: detailed recipe for how to solve a problem by programming code: program text (or synonym for program) implementation: same as code

executable: the file we run to start the program verification: providing evidence that the program works correctly debugging: locating and correcting errors in a program 1.8.2 Example: Trajectory of a Ball Problem What is the trajectory of a ball that is thrown or kicked with an initial velocity v0 making an angle with the horizontal? This problem can be solved by basic high school physics as you are encouraged to do in Exercise 1.13. The ball will follow a trajectory y D f .x/ through the air where f .x/ D x tan 1 2v2 0 gx2 cos2 C y0 : (1.6) In this expression, x is a horizontal coordinate, g is the acceleration of gravity, v0 is the size of the initial velocity that makes an angle with the x axis, and .0; y0/ is the initial position of the ball. Our programming goal is to make a program for evaluating (1.6). The program should write out the value of all the involved variables and what their units are. We remark that the formula (1.6) neglects air resistance. Exercise 1.11 explores how important air resistance is. For a soft kick (v0 D 30 km/h) of a football, the gravity force is much larger than the air resistance, but for a hard kick, air resistance may be as important as gravity. Solution We use the SI system and assume that v0 is given in km/h; g D 9:81m/s2; x, y, and y0 are measured in meters; and in degrees. The program has naturally four parts: initialization of input data, import of functions and from math, conversion of v0 and to m/s and radians, respectively, and evaluation of the right-hand side expression in (1.6). We choose to write out all numerical values with one decimal. The complete program is found in the file trajectory.py: g = 9.81 # m/s\*\*2 v0 = 15 # km/h theta = 60 # degrees x = 0.5 # m y0 = 1 # m print """\ v0 = %.1f km/h theta = %d degrees y0 = %.1f m x = %.1f m\ """ % (v0, theta, y0, x) from math import pi, tan, cos # Convert v0 to m/s and theta to radians v0 = v0/3.6 theta = theta\*pi/180

y = x\*tan(theta) - 1/(2\*v0\*\*2)\*g\*x\*\*2/((cos(theta))\*\*2) + y0 print ’y = %.1f m’ % y The backslash in the triple-quoted multi-line string makes the string continue on the next line without a newline. This means that removing the backslash results in a blank line above the v0 line and a blank line between the x and y lines in the output on the screen. Another point to mention is the expression 1/(2\*v0\*\*2), which might seem as a candidate for unintended integer division. However, the conversion of v0 to m/s involves a division by 3.6, which results in v0 being float, and therefore 2\*v0\*\*2 being float. The rest of the program should be self-explanatory at this stage in the book. We can execute the program in IPython or an ordinary terminal window and watch the output: Terminal v0 = 15.0 km/h theta = 60 degrees y0 = 1.0 m x = 0.5 m y = 1.6 m 1.8.3 About Typesetting Conventions in This Book This version of the book applies different design elements for different types of “computer text”. Complete programs and parts of programs (snippets) are typeset with a light blue background. A snippet looks like this: a = sqrt(4\*p + c) print ’a =’, a A complete program has an additional, slightly darker frame: C = 21 F = (9.0/5)\*C + 32 print F As a reader of this book, you may wonder if a code shown is a complete program you can try out or if it is just a part of a program (a snippet) so that you need to add surrounding statements (e.g., import statements) to try the code out yourself. The appearance of a vertical line to the left or not will then quickly tell you what type of code you see. An interactive Python session is typeset as >>> from math import \* >>> p = 1; c = -1.5 >>> a = sqrt(4\*p + c)

Running a program, say ball\_yc.py, in the terminal window, followed by some possible output is typeset as Terminal ball\_yc.py At t=0.0417064 s and 0.977662 s, the height is 0.2 m. Recall from Sect. 1.5.3 that we just write the program name. A real execution demands prefixing the program name by python in a terminal window, or by run if you run the program from an interactive IPython session. We refer to Appendix H.5 for more complete information on running Python programs in different ways. Sometimes just the output from a program is shown, and this output appears as plain computer text: h = 0.2 order=0, error=0.221403 order=1, error=0.0214028 order=2, error=0.00140276 order=3, error=6.94248e-05 order=4, error=2.75816e-06 Files containing data are shown in a similar way in this book: date Oslo London Berlin Paris Rome Helsinki 01.05 18 21.2 20.2 13.7 15.8 15 01.06 21 13.2 14.9 18 24 20 01.07 13 14 16 25 26.2 14.5 Style guide for Python code This book presents Python code that is (mostly) in accordance with the official Style Guide for Python Code5, known in the Python community as PEP8. Some exceptions to the rules are made to make code snippets shorter: multiple imports on one line and less blank lines. 1.9 Exercises About solving exercises There is only one way to learn programming: you have to program yourself. This means that you have to do a lot of exercises! Reading this book is necessary to learn about the Python syntax and studying the examples in depth is necessary to grasp how to think about programming and solving problems. But the main effort in the learning process is your work with exercises or your own programming projects. Solving an exercise is a three-stage procedure. First, you have to study the text in the exercise carefully to understand what the problem is about. Programming exercises, especially in this book, are about a problem setting that has to be thoroughly understood before it makes sense to understand the specific questions in

the exercise. The second phase is to write the program. The more efforts you put into the first phase, the easier it will be to find the right statements and write the code. The third and final stage is to test the program and remove errors (known as debugging and verification from Sect. 1.2). This is by far the greatest challenge for beginners. Very often, especially for newcomers to programming, it boils down to writing out the result of every statement and checking these results carefully by playing computer with pen and paper. Beginners often underestimate the amount of work required in the first and third stage and instead try to do the second stage (i.e., write the program) as quickly as possible. The more work you put into the first stage, the easier it will be to find an example in this book or elsewhere that is similar to the exercise and that can help you get started. And the more work you put into stage three up front, with constructing a test case, the better your understanding of the statements will be and the fewer errors you will commit. Experience will prove that all these assertions are right! Most exercises are associated with a filename, e.g., myexer. If the answer to the exercise is a Python program, you should store the program in a file myexer.py. If the answer can be an explanation, you may store it in a plain text file, myexer.txt, or write the text in a word processor and produce a PDF file (myexer.pdf). When you hand in exercises to teaching assistants, it is often a requirement that a trial run of the program is inserted at the end of the code. This means that you run some case with known result, direct the output to a file result, Terminal Terminal> python myprogram.py > result and copy the contents of result to a triple-quoted string with appropriate comments after the statements of the program. Here is an example of a program with its trial run inserted: F = 69.8 # Fahrenheit degrees C = (5.0/9)\*(F - 32) # Corresponding Celsius degrees print C ’’’ Trial run (correct result is 21): python f2c.py 21.0 ’’’ The trial run demonstrates that the program runs and produces correct results in a test case. Exercise 1.1: Compute 1+1 The first exercise concerns some very basic mathematics and programming: assign the result of 1C1 to a variable and print the value of that variable. Filename: 1plus1.

Exercise 1.2: Write a Hello World program Almost all books about programming languages start with a very simple program that prints the text Hello, World! to the screen. Make such a program in Python. Filename: hello\_world. Exercise 1.3: Derive and compute a formula Can a newborn baby in Norway expect to live for one billion (109) seconds? Write a Python program for doing arithmetics to answer the question. Filename: seconds2years. Exercise 1.4: Convert from meters to British length units Make a program where you set a length given in meters and then compute and write out the corresponding length measured in inches, in feet, in yards, and in miles. Use that one inch is 2.54 cm, one foot is 12 inches, one yard is 3 feet, and one British mile is 1760 yards. For verification, a length of 640 meters corresponds to 25196.85 inches, 2099.74 feet, 699.91 yards, or 0.3977 miles. Filename: length\_conversion. Exercise 1.5: Compute the mass of various substances The density of a substance is defined as % D m=V , where m is the mass of a volume V . Compute and print out the mass of one liter of each of the following substances whose densities in g/cm3 are found in the file src/files/densities.dat6 : iron, air, gasoline, ice, the human body, silver, and platinum. Filename: 1liter. Exercise 1.6: Compute the growth of money in a bank Let p be a bank’s interest rate in percent per year. An initial amount A has then grown to A 1 C p 100 n after n years. Make a program for computing how much money 1000 euros have grown to after three years with 5 percent interest rate. Filename: interest\_rate. Exercise 1.7: Find error(s) in a program Suppose somebody has written a simple one-line program for computing sin.1/: x=1; print ’sin(%g)=%g’ % (x, sin(x)) Create this program and try to run it. What is the problem? Filename: find\_errors\_sin1. Exercise 1.8: Type in program text Type the following program in your editor and execute it. If your program does not work, check that you have copied the code correctly

from math import pi h = 5.0 # height b = 2.0 # base r = 1.5 # radius area\_parallelogram = h\*b print ’The area of the parallelogram is %.3f’ % area\_parallelogram area\_square = b\*\*2 print ’The area of the square is %g’ % area\_square area\_circle = pi\*r\*\*2 print ’The area of the circle is %.3f’ % area\_circle volume\_cone = 1.0/3\*pi\*r\*\*2\*h print ’The volume of the cone is %.3f’ % volume\_cone Filename: formulas\_shapes. Exercise 1.9: Type in programs and debug them Type these short programs in your editor and execute them. When they do not work, identify and correct the erroneous statements. a) Does sin2 .x/ C cos2.x/ D 1? from math import sin, cos x = pi/4 1\_val = math.sin^2(x) + math.cos^2(x) print 1\_VAL b) Compute s in meters when s D v0t C 1 2at 2, with v0 D 3 m/s, t D 1 s, a D 2 m/s2 . v0 = 3 m/s t = 1 s a = 2 m/s\*\*2 s = v0.t + 0,5.a.t\*\*2 print s c) Verify these equations: .a C b/2 D a2 C 2ab C b2 .a b/2 D a2 2ab C b2 a = 3,3 b = 5,3 a2 = a\*\*2 b2 = b\*\*2

eq1\_sum = a2 + 2ab + b2 eq2\_sum = a2 - 2ab + b2 eq1\_pow = (a + b)\*\*2 eq2\_pow = (a - b)\*\*2 print ’First equation: %g = %g’,%(eq1\_sum, eq1\_pow) print ’Second equation: %h = %h’,%(eq2\_pow, eq2\_pow) Filename: find\_errors\_programs. Exercise 1.10: Evaluate a Gaussian function The bell-shaped Gaussian function, f .x/ D 1 p2 s exp 1 2 x m s 2 ; (1.7) is one of the most widely used functions in science and technology. The parameters m and s>0 are prescribed real numbers. Make a program for evaluating this function when m D 0, s D 2, and x D 1. Verify the program’s result by comparing with hand calculations on a calculator. Filename: gaussian1. Remarks The function (1.7) is named after Carl Friedrich Gauss7, 1777–1855, who was a German mathematician and scientist, now considered as one of the greatest scientists of all time. He contributed to many fields, including number theory, statistics, mathematical analysis, differential geometry, geodesy, electrostatics, astronomy, and optics. Gauss introduced the function (1.7) when he analyzed probabilities related to astronomical data. Exercise 1.11: Compute the air resistance on a football The drag force, due to air resistance, on an object can be expressed as Fd D 1 2 CD%AV 2; (1.8) where % is the density of the air, V is the velocity of the object, A is the crosssectional area (normal to the velocity direction), and CD is the drag coefficient, which depends heavily on the shape of the object and the roughness of the surface. The gravity force on an object with mass m is Fg D mg, where g D 9:81 m s2. We can use the formulas for Fd and Fg to study the importance of air resistance versus gravity when kicking a football. The density of air is % D 1:2 kg m3. We have A D a2 for any ball with radius a. For a football, a D 11 cm and the mass is 0.43 kg. The drag coefficient CD varies with the velocity and can be taken as 0.4. Make a program that computes the drag force and the gravity force on a football. Write out the forces with one decimal in units of Newton (N D kg m=s2). Also print the ratio of the drag force and the gravity force. Define CD, %, A, V , m, g,

Fd , and Fg as variables, and put a comment with the corresponding unit. Use the program to calculate the forces on the ball for a hard kick, V D 120 km=h and for a soft kick, V D 30 km=h (it is easy to mix inconsistent units, so make sure you compute with V expressed in m=s). Filename: kick. Exercise 1.12: How to cook the perfect egg As an egg cooks, the proteins first denature and then coagulate. When the temperature exceeds a critical point, reactions begin and proceed faster as the temperature increases. In the egg white, the proteins start to coagulate for temperatures above 63 ıC, while in the yolk the proteins start to coagulate for temperatures above 70 ıC. For a soft boiled egg, the white needs to have been heated long enough to coagulate at a temperature above 63 ıC, but the yolk should not be heated above 70 ıC. For a hard boiled egg, the center of the yolk should be allowed to reach 70 ıC. The following formula expresses the time t it takes (in seconds) for the center of the yolk to reach the temperature Ty (in Celsius degrees): t D M2=3c1=3 K2.4=3/2=3 ln 0:76 To Tw Ty Tw : (1.9) Here, M, , c, and K are properties of the egg: M is the mass, is the density, c is the specific heat capacity, and K is thermal conductivity. Relevant values are M D 47 g for a small egg and M D 67 g for a large egg, D 1:038 g cm3, c D 3:7 J g1 K1, and K D 5:4103 W cm1 K1. Furthermore, Tw is the temperature (in C degrees) of the boiling water, and To is the original temperature (in C degrees) of the egg before being put in the water. Implement the formula in a program, set Tw D 100 ıC and Ty D 70 ıC, and compute t for a large egg taken from the fridge (To D 4 ıC) and from room temperature (To D 20 ıC). Filename: egg. Exercise 1.13: Derive the trajectory of a ball The purpose of this exercise is to explain how Equation (1.6) for the trajectory of a ball arises from basic physics. There is no programming in this exercise, just physics and mathematics. The motion of the ball is governed by Newton’s second law: Fx D max (1.10) Fy D may (1.11) where Fx and Fy are the sum of forces in the x and y directions, respectively, ax and ay are the accelerations of the ball in the x and y directions, and m is the mass of the ball. Let .x.t/; y.t// be the position of the ball, i.e., the horizontal and vertical coordinate of the ball at time t. There are well-known relations between acceleration, velocity, and position: the acceleration is the time derivative of the velocity, and the velocity is the time derivative of the position. Therefore we have

that ax D d2x dt 2 ; (1.12) ay D d2y dt 2 : (1.13) If we assume that gravity is the only important force on the ball, Fx D 0 and Fy D mg. Integrate the two components of Newton’s second law twice. Use the initial conditions on velocity and position, d dt x.0/ D v0 cos ; (1.14) d dt y.0/ D v0 sin ; (1.15) x.0/ D 0; (1.16) y.0/ D y0; (1.17) to determine the four integration constants. Write up the final expressions for x.t/ and y.t/. Show that if D =2, i.e., the motion is purely vertical, we get the formula (1.1) for the y position. Also show that if we eliminate t, we end up with the relation (1.6) between the x and y coordinates of the ball. You may read more about this type of motion in a physics book, e.g., [15]. Filename: trajectory. Exercise 1.14: Find errors in the coding of formulas Some versions of our program for calculating the formula (1.3) are listed below. Find the versions that will not work correctly and explain why in each case. C = 21; F = 9/5\*C + 32; print F C = 21.0; F = (9/5)\*C + 32; print F C = 21.0; F = 9\*C/5 + 32; print F C = 21.0; F = 9.\*(C/5.0) + 32; print F C = 21.0; F = 9.0\*C/5.0 + 32; print F C = 21; F = 9\*C/5 + 32; print F C = 21.0; F = (1/5)\*9\*C + 32; print F C = 21; F = (1./5)\*9\*C + 32; print F Filename: find\_errors\_division. Exercise 1.15: Explain why a program does not work Figure out why the following program does not work: C = A + B A = 3 B = 2 print C

Exercise 1.16: Find errors in Python statements Try the following statements in an interactive Python shell. Explain why some statements fail and correct the errors. 1a = 2 a1 = b x = 2 y = X + 4 # is it 6? from Math import tan print tan(pi) pi = "3.14159’ print tan(pi) c = 4\*\*3\*\*2\*\*3 \_ = ((c-78564)/c + 32)) discount = 12% AMOUNT = 120.- amount = 120$ address = hpl@simula.no and = duck class = ’INF1100, gr 2" continue\_ = x > 0 rev = fox = True Norwegian = [’a human language’] true = fox is rev in Norwegian Hint It is wise to test the values of the expressions on the right-hand side, and the validity of the variable names, separately before you put the left- and right-hand sides together in statements. The last two statements work, but explaining why goes beyond what is treated in this chapter. Filename: find\_errors\_syntax. Exercise 1.17: Find errors in the coding of a formula Given a quadratic equation, ax2 C bx C c D 0; the two roots are x1 D b C pb2 4ac 2a ; x2 D b p b2 4ac 2a : (1.18) What are the problems with the following program? a = 2; b = 1; c = 2 from math import sqrt q = b\*b - 4\*a\*c q\_sr = sqrt(q) x1 = (-b + q\_sr)/2\*a x2 = (-b - q\_sr)/2\*a print x1, x2 Correct the program so that it solves the given equation. Filename: find\_errors\_roots.

Exercise 1.18: Find errors in a program What is the problem in the following program? from math import pi, tan tan = tan(pi/4) tan2 = tan(pi/3) print tan, tan2 Filename: find\_errors\_tan.