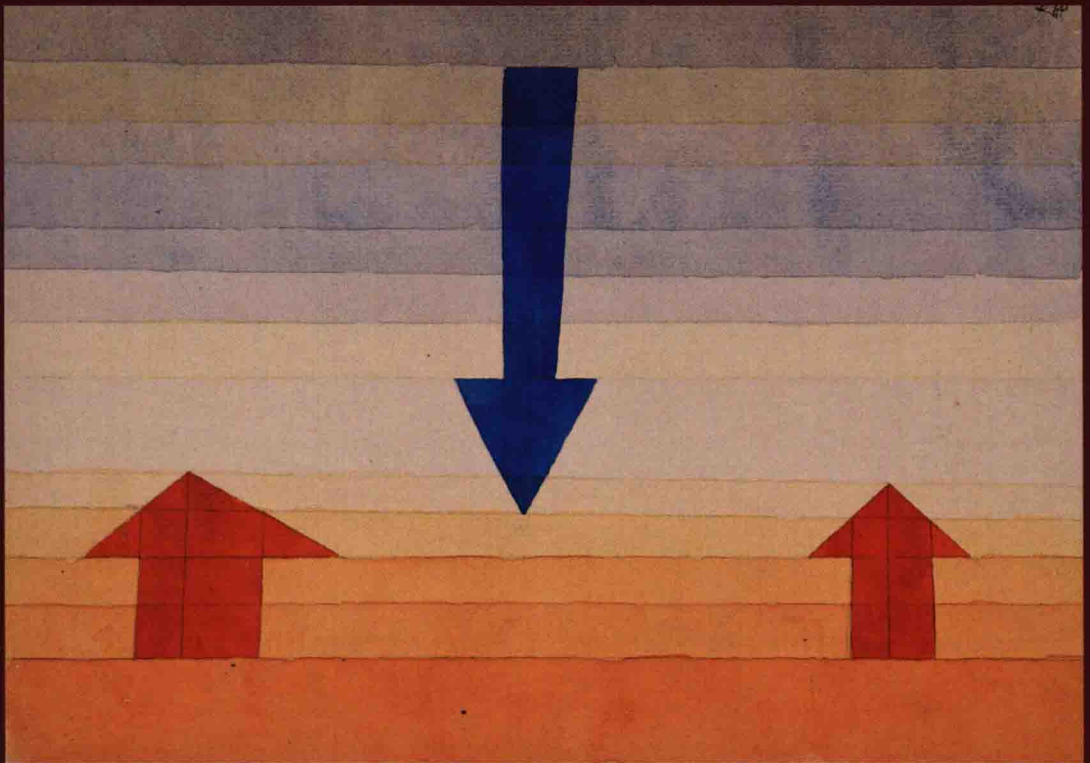


Second Edition

PYTHON FOR SCIENTISTS



JOHN M. STEWART

Python for Scientists

Second Edition

JOHN M. STEWART

Department of Applied Mathematics & Theoretical Physics
University of Cambridge



CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

4843/24, 2nd Floor, Ansari Road, Daryaganj, Delhi – 110002, India

79 Anson Road, #06–04/06, Singapore 079906

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning, and research at the highest international levels of excellence.

www.cambridge.org

Information on this title: www.cambridge.org/9781316641231

DOI: 10.1017/9781108120241

© John M. Stewart 2014, 2017

This publication is in copyright. Subject to statutory exception and to the provisions of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published 2014

Second edition 2017

Printed in the United Kingdom by TJ International Ltd. Padstow Cornwall

A catalogue record for this publication is available from the British Library.

Library of Congress Cataloging-in-Publication Data

Names: Stewart, John, 1943 July 1–

Title: Python for scientists / John M. Stewart, Department of Applied, Mathematics & Theoretical Physics, University of Cambridge.

Description: Second edition. | Cambridge, United Kingdom ; New York, NY, USA : Cambridge University Press, [2017] | Includes bibliographical references and index.

Identifiers: LCCN 2016049298 | ISBN 9781316641231 (paperback)

Subjects: LCSH: Science–Data processing. | Python (Computer program language)

Classification: LCC Q183.9 .S865 2017 | DDC 005.13/3–dc23

LC record available at <https://lcn.loc.gov/2016049298>

ISBN 978-1-316-64123-1 Paperback

Additional resources for this publication at www.cambridge.org/9781316641231

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party Internet Web sites referred to in this publication and does not guarantee that any content on such Web sites is, or will remain, accurate or appropriate.

Python for Scientists

Second Edition

Scientific Python is a significant public domain alternative to expensive proprietary software packages. This book teaches from scratch everything the working scientist needs to know using copious, downloadable, useful and adaptable code snippets. Readers will discover how easy it is to implement and test non-trivial mathematical algorithms and will be guided through the many freely available add-on modules. A range of examples, relevant to many different fields, illustrate the language's capabilities. The author also shows how to use pre-existing legacy code (usually in Fortran77) within the Python environment, thus avoiding the need to master the original code.

In this new edition, several chapters have been rewritten to reflect the *IPython* notebook style. With an extended index, an entirely new chapter discussing *SymPy* and a substantial increase in the number of code snippets, researchers and research students will be able to quickly acquire all the skills needed for using Python effectively.

Preface to the Second Edition

The motivation for writing this book, and the acknowledgements of the many who have assisted in its production, are included in the topics of the Preface to the first edition, which is reprinted after this one. Here I also need to adjoin thanks to the many readers who provided constructive criticisms, most of which have been incorporated in this revision. The purpose here is to explain why a second edition is needed. Superficially it might appear that very little has changed, apart from a new Chapter 7 which discusses *SymPy*, Python's own computer algebra system.

There is, however, a fundamental change, which permeates most of the latest version of this book. When the first edition was prepared, the reliable way to use the enhanced interpreter *IPython* was via the traditional "terminal mode". Preparations were under way for an enhanced "notebook mode", which looked then rather like the Mathematica notebook concept, except that it appeared within one's default web browser.¹ That project has now morphed into the Jupyter notebook. The notebook allows one to construct and distribute documents containing computer code (over forty languages are supported), equations, explanatory text, figures and visualizations. Since this is also perhaps the easiest software application for a beginner to develop Python experience, much of the book has been rewritten for the notebook user. In particular there is now a lightning course on how to use the notebook in Appendix A, and Chapter 2 has been extensively rewritten to demonstrate its properties. All of the material in the book now reflects, where appropriate, its use. For example, it allows *SymPy* to produce algebraic expressions whose format is unsurpassed by other computer algebra systems.

This change also affects the areas of interactive graphics and visual animations. Their demands are such that the standard Python two-dimensional graphics package *Matplotlib* is having difficulty in producing platform-independent results. Indeed, because of "improved" software upgrades, the code suggested for immediate on-screen animations in the first edition no longer works. However, the notebook concept has a subtle solution to resolve this impasse. Recall that the notebook window is your browser window, which uses modern HTML graphics. The consequent benefits are introduced in Chapter 6.

As a final enhancement, all but the most trivial code snippets listed in this book are now available in electronic form, as a notebook of course, but the website includes

¹ Internet access is neither required nor used.

HTML and PDF versions, see Section 1.2. The explanatory text surrounding the text is not included. For that you have to read the book, in hard copy or ebook format!

Note added in proof:

John died shortly after the completion of the Second Edition, and is much missed by colleagues, friends and family, especially the “Python widow”.

Preface to the First Edition

I have used computers as an aid to scientific research for over 40 years. During that time, hardware has become cheap, fast and powerful. However, software relevant to the working scientist has become progressively more complicated. My favourite textbooks on Fortran90 and C++ run to 1200 and 1600 pages respectively. And then we need documentation on mathematics libraries and graphics packages. A newcomer going down this route is going to have to invest significant amounts of time and energy in order to write useful programmes. This has led to the emergence of “scientific packages” such as Matlab® or Mathematica® which avoid the complications of compiled languages, separate mathematics libraries and graphics packages. I have used them and found them very convenient for executing the tasks envisaged by their developers. However, I also found them very difficult to extend beyond these boundaries, and so I looked for alternative approaches.

Some years ago, a computer science colleague suggested that I should take a look at Python. At that time, it was clear that Python had great potential but a very flaky implementation. It was, however, free and open-source, and was attracting what has turned out to be a very effective army of developers. More recently, their efforts have coordinated to produce a formidable package consisting of a small core language surrounded by a wealth of add-on libraries or *modules*. A select group of these can and do replicate the facilities of the conventional scientific packages. More importantly an informed, intelligent user of Python and its modules can carry out major projects usually entrusted to dedicated programmers using Fortran, C etc. There is a marginal loss of execution speed, but this is more than compensated for by the vastly telescoped development time. The purpose of this book is to explain to working scientists the utility of this relatively unknown resource.

Most scientists will have some computer familiarity and programming awareness, although not necessarily with Python, and I shall take advantage of this. Therefore, unlike many books which set out to “teach” a language, this one is not just a brisk trot through the reference manuals. Python has many powerful but unfamiliar facets, and these need more explanation than the familiar ones. In particular, if you encounter in this text a reference to the “beginner” or the “unwary”, it signifies a point which is not made clear in the documentation, and has caught out this author at least once.

The first seven chapters, plus Appendix A, cover almost everything the working scientist needs to know in order to get started in using Python effectively. My editor and some referees suggested that I should devote the second half of the book to problems in

a particular field. This would have led to a series of books, “Python for Biochemists”, “Python for Crystallographers”, . . . , all with a common first half. Instead I have chosen to cover just three topics, which, however, should be far more widely applicable in many different fields. Chapter 8 covers four radically different types of ordinary differential equations and shows how to use the various relevant black boxes, which are often Python wrappers around tried and trusted Fortran codes. The next chapter while ostensibly about pseudospectral approaches to evolutionary partial differential equations, actually covers a topic of great utility to many scientists, namely how to reuse legacy code, usually written in Fortran77, within Python at Fortran-like speeds, without understanding Fortran. The final chapter about solving very large linear systems via multigrid is also a case history in how to use object-oriented programming meaningfully in a scientific context. If readers look carefully and critically at these later chapters, they should gain the practical expertise to handle problems in their own field.

Acknowledgments are due to the many Python developers who have produced and documented a very useful tool, and also to the very many who have published code snippets on the web, a great aid to the tyro, such as this author. Many of my colleagues have offered valuable advice. Des Higham generously consented to my borrowing his ideas for the last quarter of Chapter 8. I am especially grateful to Oliver Rinne who read carefully and critically an early draft. At Cambridge University Press, my Production Editor, Jessica Murphy and my Copy Editor, Anne Rix have exhibited their customary expertise. Last but not least I thank the Department of Applied Mathematics and Theoretical Physics, Cambridge for continuing to offer me office space after my retirement, which has facilitated the production of this book.

Writing a serious book is not a trivial task and so I am rather more than deeply grateful for the near-infinite patience of *Mary*, the “Python-widow”, which made this book possible!

Contents

| | | |
|----------|---|-----------|
| | <i>Preface to the Second Edition</i> | page xiii |
| | <i>Preface to the First Edition</i> | xv |
| 1 | Introduction | 1 |
| | 1.1 Scientific Software | 1 |
| | 1.2 The Plan of This Book | 4 |
| | 1.3 Can Python Compete with Compiled Languages? | 8 |
| | 1.4 Limitations of This Book | 9 |
| | 1.5 Installing Python and Add-ons | 9 |
| 2 | Getting Started with <i>IPython</i> | 11 |
| | 2.1 Tab Completion | 11 |
| | 2.2 Introspection | 12 |
| | 2.3 History | 14 |
| | 2.4 Magic Commands | 14 |
| | 2.5 <i>IPython</i> in Action: An Extended Example | 15 |
| | 2.5.1 An <i>IPython</i> terminal workflow | 17 |
| | 2.5.2 An <i>IPython</i> notebook workflow | 17 |
| 3 | A Short Python Tutorial | 21 |
| | 3.1 Typing Python | 21 |
| | 3.2 Objects and Identifiers | 22 |
| | 3.3 Numbers | 24 |
| | 3.3.1 Integers | 24 |
| | 3.3.2 Real numbers | 24 |
| | 3.3.3 Boolean numbers | 26 |
| | 3.3.4 Complex numbers | 26 |
| | 3.4 Namespaces and Modules | 27 |
| | 3.5 Container Objects | 28 |
| | 3.5.1 <i>Lists</i> | 29 |
| | 3.5.2 <i>List</i> indexing | 30 |
| | 3.5.3 <i>List</i> slicing | 30 |
| | 3.5.4 <i>List</i> mutability | 31 |
| | 3.5.5 <i>Tuples</i> | 32 |

| | | |
|----------|---|-----------|
| 3.5.6 | <i>Strings</i> | 33 |
| 3.5.7 | <i>Dictionaries</i> | 33 |
| 3.6 | Python <i>if</i> Statements | 34 |
| 3.7 | Loop Constructs | 35 |
| 3.7.1 | The Python <i>for</i> loop | 35 |
| 3.7.2 | The Python <i>continue</i> statement | 37 |
| 3.7.3 | The Python <i>break</i> statement | 37 |
| 3.7.4 | <i>List</i> comprehensions | 38 |
| 3.7.5 | Python <i>while</i> loops | 39 |
| 3.8 | Functions | 39 |
| 3.8.1 | Syntax and scope | 40 |
| 3.8.2 | Positional arguments | 43 |
| 3.8.3 | Keyword arguments | 43 |
| 3.8.4 | Variable number of positional arguments | 43 |
| 3.8.5 | Variable number of keyword arguments | 44 |
| 3.8.6 | Python input/output functions | 44 |
| 3.8.7 | The Python <i>print</i> function | 45 |
| 3.8.8 | Anonymous functions | 47 |
| 3.9 | Introduction to Python Classes | 47 |
| 3.10 | The Structure of Python | 50 |
| 3.11 | Prime Numbers: A Worked Example | 51 |
| 4 | <i>NumPy</i> | 55 |
| 4.1 | One-Dimensional Arrays | 57 |
| 4.1.1 | Ab initio constructors | 57 |
| 4.1.2 | Look-alike constructors | 58 |
| 4.1.3 | Arithmetical operations on vectors | 59 |
| 4.1.4 | Ufuncs | 60 |
| 4.1.5 | Logical operations on vectors | 62 |
| 4.2 | Two-Dimensional Arrays | 65 |
| 4.2.1 | Broadcasting | 65 |
| 4.2.2 | Ab initio constructors | 66 |
| 4.2.3 | Look-alike constructors | 68 |
| 4.2.4 | Operations on arrays and ufuncs | 69 |
| 4.3 | Higher-Dimensional Arrays | 69 |
| 4.4 | Domestic Input and Output | 69 |
| 4.4.1 | Discursive output and input | 70 |
| 4.4.2 | <i>NumPy</i> text output and input | 71 |
| 4.4.3 | <i>NumPy</i> binary output and input | 72 |
| 4.5 | Foreign Input and Output | 73 |
| 4.5.1 | Small amounts of data | 73 |
| 4.5.2 | Large amounts of data | 73 |
| 4.6 | Miscellaneous Ufuncs | 74 |
| 4.6.1 | Maxima and minima | 74 |

| | | |
|----------|---|------------|
| 4.6.2 | Sums and products | 75 |
| 4.6.3 | Simple statistics | 75 |
| 4.7 | Polynomials | 75 |
| 4.7.1 | Converting data to coefficients | 76 |
| 4.7.2 | Converting coefficients to data | 76 |
| 4.7.3 | Manipulating polynomials in coefficient form | 76 |
| 4.8 | Linear Algebra | 76 |
| 4.8.1 | Basic operations on matrices | 76 |
| 4.8.2 | More specialized operations on matrices | 78 |
| 4.8.3 | Solving linear systems of equations | 79 |
| 4.9 | More <i>NumPy</i> and Beyond | 79 |
| 4.9.1 | <i>SciPy</i> | 80 |
| 4.9.2 | <i>SciKits</i> | 81 |
| 5 | Two-Dimensional Graphics | 82 |
| 5.1 | Introduction | 82 |
| 5.2 | Getting Started: Simple Figures | 83 |
| 5.2.1 | Front-ends | 83 |
| 5.2.2 | Back-ends | 83 |
| 5.2.3 | A simple figure | 84 |
| 5.2.4 | Interactive controls | 86 |
| 5.3 | Object-Oriented <i>Matplotlib</i> | 87 |
| 5.4 | Cartesian Plots | 88 |
| 5.4.1 | The <i>Matplotlib</i> <code>plot</code> function | 88 |
| 5.4.2 | Curve styles | 89 |
| 5.4.3 | Marker styles | 90 |
| 5.4.4 | Axes, grid, labels and title | 90 |
| 5.4.5 | A not-so-simple example: partial sums of Fourier series | 91 |
| 5.5 | Polar Plots | 93 |
| 5.6 | Error Bars | 94 |
| 5.7 | Text and Annotations | 95 |
| 5.8 | Displaying Mathematical Formulae | 96 |
| 5.8.1 | Non- \LaTeX users | 96 |
| 5.8.2 | \LaTeX users | 97 |
| 5.8.3 | Alternatives for \LaTeX users | 98 |
| 5.9 | Contour Plots | 98 |
| 5.10 | Compound Figures | 101 |
| 5.10.1 | Multiple figures | 101 |
| 5.10.2 | Multiple plots | 102 |
| 5.11 | Mandelbrot Sets: A Worked Example | 104 |
| 6 | Multi-Dimensional Graphics | 109 |
| 6.1 | Introduction | 109 |
| 6.1.1 | Multi-dimensional data sets | 109 |

| | | |
|----------|--|-----|
| 6.2 | The Reduction to Two Dimensions | 109 |
| 6.3 | Visualization Software | 110 |
| 6.4 | Example Visualization Tasks | 111 |
| 6.5 | Visualization of Solitary Waves | 111 |
| 6.5.1 | The <i>interactivity</i> task | 112 |
| 6.5.2 | The animation task | 113 |
| 6.5.3 | The movie task | 115 |
| 6.6 | Visualization of Three-Dimensional Objects | 116 |
| 6.7 | A Three-Dimensional Curve | 118 |
| 6.7.1 | Visualizing the curve with <i>mplot3d</i> | 118 |
| 6.7.2 | Visualizing the curve with <i>mlab</i> | 120 |
| 6.8 | A Simple Surface | 121 |
| 6.8.1 | Visualizing the simple surface with <i>mplot3d</i> | 121 |
| 6.8.2 | Visualizing the simple surface with <i>mlab</i> | 123 |
| 6.9 | A Parametrically Defined Surface | 124 |
| 6.9.1 | Visualizing Enneper's surface using <i>mplot3d</i> | 124 |
| 6.9.2 | Visualizing Enneper's surface using <i>mlab</i> | 125 |
| 6.10 | Three-Dimensional Visualization of a Julia Set | 126 |
| 7 | SymPy: A Computer Algebra System | 129 |
| 7.1 | Computer Algebra Systems | 129 |
| 7.2 | Symbols and Functions | 130 |
| 7.3 | Conversions from Python to <i>SymPy</i> and Vice Versa | 132 |
| 7.4 | Matrices and Vectors | 133 |
| 7.5 | Some Elementary Calculus | 134 |
| 7.5.1 | Differentiation | 134 |
| 7.5.2 | Integration | 134 |
| 7.5.3 | Series and limits | 136 |
| 7.6 | Equality, Symbolic Equality and Simplification | 136 |
| 7.7 | Solving Equations | 138 |
| 7.7.1 | Equations with one independent variable | 138 |
| 7.7.2 | Linear equations with more than one independent variable | 139 |
| 7.7.3 | More general equations | 141 |
| 7.8 | Solving Ordinary Differential Equations | 142 |
| 7.9 | Plotting from within <i>SymPy</i> | 144 |
| 8 | Ordinary Differential Equations | 150 |
| 8.1 | Initial Value Problems | 150 |
| 8.2 | Basic Concepts | 150 |
| 8.3 | The <i>odeint</i> Function | 153 |
| 8.3.1 | Theoretical background | 153 |
| 8.3.2 | The harmonic oscillator | 155 |
| 8.3.3 | The van der Pol oscillator | 158 |
| 8.3.4 | The Lorenz equations | 159 |

| | | |
|-----------|--|------------|
| 8.4 | Two-Point Boundary Value Problems | 161 |
| 8.4.1 | Introduction | 161 |
| 8.4.2 | Formulation of the boundary value problem | 162 |
| 8.4.3 | A simple example | 164 |
| 8.4.4 | A linear eigenvalue problem | 165 |
| 8.4.5 | A non-linear boundary value problem | 167 |
| 8.5 | Delay Differential Equations | 171 |
| 8.5.1 | A model equation | 172 |
| 8.5.2 | More general equations and their numerical solution | 173 |
| 8.5.3 | The logistic equation | 174 |
| 8.5.4 | The Mackey–Glass equation | 176 |
| 8.6 | Stochastic Differential Equations | 179 |
| 8.6.1 | The Wiener process | 179 |
| 8.6.2 | The Itô calculus | 181 |
| 8.6.3 | Itô and Stratonovich stochastic integrals | 184 |
| 8.6.4 | Numerical solution of stochastic differential equations | 185 |
| 9 | Partial Differential Equations: A Pseudospectral Approach | 192 |
| 9.1 | Initial Boundary Value Problems | 192 |
| 9.2 | Method of Lines | 193 |
| 9.3 | Spatial Derivatives via Finite Differencing | 193 |
| 9.4 | Spatial Derivatives by Spectral Techniques | 194 |
| 9.5 | The IVP for Spatially Periodic Problems | 196 |
| 9.6 | Spectral Techniques for Non-Periodic Problems | 199 |
| 9.7 | An Introduction to <code>f2py</code> | 201 |
| 9.7.1 | Simple examples with scalar arguments | 201 |
| 9.7.2 | Vector arguments | 203 |
| 9.7.3 | A simple example with multi-dimensional arguments | 204 |
| 9.7.4 | Undiscussed features of <code>f2py</code> | 206 |
| 9.8 | A Real-Life <code>f2py</code> Example | 206 |
| 9.9 | Worked Example: Burgers' Equation | 208 |
| 9.9.1 | Boundary conditions: the traditional approach | 208 |
| 9.9.2 | Boundary conditions: the penalty approach | 209 |
| 10 | Case Study: Multigrid | 213 |
| 10.1 | The One-Dimensional Case | 214 |
| 10.1.1 | Linear elliptic equations | 214 |
| 10.1.2 | Smooth and rough modes | 215 |
| 10.2 | The Tools of Multigrid | 215 |
| 10.2.1 | Relaxation methods | 215 |
| 10.2.2 | Residual and error | 218 |
| 10.2.3 | Prolongation and restriction | 219 |
| 10.3 | Multigrid Schemes | 220 |
| 10.3.1 | The two-grid algorithm | 221 |

| | | |
|--|--|-----|
| 10.3.2 | The V-cycle scheme | 222 |
| 10.3.3 | The full multigrid (FMG) scheme | 223 |
| 10.4 | A Simple Python Multigrid Implementation | 224 |
| 10.4.1 | Utility functions | 225 |
| 10.4.2 | Smoothing functions | 226 |
| 10.4.3 | Multigrid functions | 228 |
| Appendix A Installing a Python Environment | | 235 |
| A.1 | Installing Python Packages | 235 |
| A.2 | Communication with <i>IPython</i> Using the Jupyter Notebook | 237 |
| A.2.1 | Starting and stopping the notebook | 237 |
| A.2.2 | Working in the notebook | 238 |
| A.2.2.1 | Entering headers | 239 |
| A.2.2.2 | Entering Markdown text | 239 |
| A.2.2.3 | Converting notebooks to other formats | 240 |
| A.3 | Communication with <i>IPython</i> Using Terminal Mode | 240 |
| A.3.1 | Editors for programming | 240 |
| A.3.2 | The two-windows approach | 241 |
| A.3.3 | Calling the editor from within <i>IPython</i> | 242 |
| A.3.4 | Calling <i>IPython</i> from within the editor | 242 |
| A.4 | Communication with <i>IPython</i> via an IDE | 242 |
| A.5 | Installing Additional Packages | 243 |
| Appendix B Fortran77 Subroutines for Pseudospectral Methods | | 244 |
| References | | 250 |
| Hints for Using the Index | | 252 |
| Index | | 253 |

1 Introduction

The title of this book is “Python for Scientists”, but what does that mean? The dictionary defines “Python” as either (a) a non-venomous snake from Asia or Saharan Africa or (b) a computer scripting language, and it is the second option which is intended here. (What exactly this second definition means will be explained later.) By “scientist”, I mean anyone who uses quantitative models either to obtain conclusions by processing pre-collected experimental data or to model potentially observable results from a more abstract theory, **and** who asks “what if?”. What if I analyse the data in a different way? What if I change the model? Thus the term also includes economists, engineers and mathematicians among others, as well as the usual concept of scientists. Given the volume of potential data or the complexity (non-linearity) of many theoretical models, the use of computers to answer these questions is fast becoming mandatory.

Advances in computer hardware mean that immense amounts of data or ever more complex models can be processed at increasingly rapid speeds. These advances also mean reduced costs so that today virtually every scientist has access to a “personal computer”, either a desktop work station or a laptop, and the distinction between the two is narrowing quickly. It might seem to be a given that suitable software will also be available so that the “what if” questions can be answered readily. However, this turns out not always to be the case. A quick pragmatic reason is that, while there is a huge market for hardware improvements, scientists form a very small fraction of it and so there is little financial incentive to improve scientific software. But for scientists, this issue is important and we need to examine it in more detail.

1.1 Scientific Software

Before we discuss what is available, it is important to note that all computer software comes in one of two types: proprietary and open-source. The first is supplied by a commercial firm. Such organizations have both to pay wages and taxes and to provide a return for their shareholders. Therefore, they have to charge real money for their products, and, in order to protect their assets from their competitors, they do not tell the customer how their software works. Thus, the end-users have little chance of being able to adapt or optimize the product for their own use. Since wages and taxes are recurrent expenditures, the company needs to issue frequent charged-for updates and improvements (the *Danegeld effect*). Open-source software is available for free or at nominal cost (media,

postage etc.). It is usually developed by computer literate individuals, often working for universities or similar organizations, who provide the service for their colleagues. It is distributed subject to anti-copyright licences, which give nobody the right to copy-right it or to use it for commercial gain. Conventional economics might suggest that the gamut of open-source software should be inferior to its proprietary counterpart, or else the commercial organizations would lose their market. As we shall see, this is not necessarily the case.

Next we need to differentiate between two different types of scientific software. Computers operate according to a very limited and obscure set of instructions. A programming language is a somewhat less limited subset of human language in which sequences of instructions are written, usually by humans, to be read and understood by computers. The most common languages are capable of expressing very sophisticated mathematical concepts, albeit with a steep learning curve. Only a few language families, e.g., C and Fortran, have been widely accepted, but they come with many different dialects, e.g., Fortran77, Fortran90, Ansi C, C++ etc. Compilers then translate code written by humans into machine code which can be optimized for speed and then processed. As such, they are rather like Formula 1 racing cars. The best of them are capable of breathtakingly fast performance, but driving them is not intuitive and requires a great deal of training and experience. Note that compilers need to be supplemented by libraries of software packages which implement frequently used numerical algorithms, and graphics packages will usually be needed. Fast versatile library packages are usually expensive, although good public domain packages are starting to appear.

A racing car is not usually the best choice for a trip to the supermarket, where speed is not of paramount importance. Similarly, compiled languages are not always ideal for trying out new mathematical ideas. Thus for the intended readers of this book the direct use of compilers is likely to be unattractive, unless their use is mandatory. We therefore look at the other type of software, usually called “scientific packages”. Proprietary packages include Mathematica and Matlab, and open-source equivalents include Maxima, Octave, R and SciLab. They all operate in a similar fashion. Each provides its own idiosyncratic programming language in which problems are entered at a user interface. After a coherent group of statements, often just an individual statement, has been typed, the package writes equivalent core language code and compiles it on the fly. Thus errors and/or results can be reported immediately back to the user. Such packages are called “interpreters”, and older readers may remember, perhaps with mixed feelings, the BASIC language. For small projects, the slow operation compared with a fully compiled code is masked by the speed of current microprocessors, but it does become apparent on larger jobs.

These packages are attractive for at least two reasons. The first is their ability to post-process data. For example, suppose that x is a real variable and there exists a (possibly unknown) function $y(x)$. Suppose also that for an ordered set X of discrete instances of x we have computed a corresponding set Y of instances of y . Then a command similar to `plot(X, Y)` will display instantly a nicely formatted graph on the screen. Indeed, those generated by Matlab in particular can be of publication quality. A second advantage is the apparent ability of some of the proprietary packages to perform in addition some