```
import numpy
from ODESolver import RungeKutta4

def rhs(u, t):
    R = 1
    return alpha*u*(1 - u/R)
```

$$\frac{du}{dt} = \alpha u(1-u)$$

$$u(0) = 0.1$$

$$R = 1$$

$$\alpha = 0.2$$

TEXTS IN COMPUTATIONAL SCIENCE AND ENGINEERING 6

Hans Petter Langtangen

A Primer on Scientific Programming with Python

Fifth Edition

Editorial Board
T. J. Barth
M. Griebel
D. E. Keyes
R. M. Nieminen
D. Roose
T. Schlick

Copyrighted material

Hans Petter Langtangen
Simula Research Laboratory
Fornebu, Norway

On leave from:

Department of Informatics
University of Oslo
Oslo, Norway

ISSN 1611-0994
Texts in Computational Science and Engineering
ISBN 978-3-662-49886-6 ISBN 978-3-662-49887-3 (eBook)
DOI 10.1007/978-3-662-49887-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2016945366

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

© Springer-Verlag Berlin Heidelberg 2009, 2011, 2012, 2014, 2016
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer Berlin Heidelberg

Copyrighted material

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 34

Copyrighted material

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 . . . 105

Copyrighted material

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 . . . 193

Copyrighted material

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 . . . 273

Copyrighted material

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

Copyrighted material

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 . . . 432

Copyrighted material

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 . . . 523

Copyrighted material

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 . . . 621

Copyrighted material

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.1.10 Newton's Method . . . 659
A.1.11 The 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 . . . 705

Copyrighted material

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 Δt . . . 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 Δt 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 Hierarchy . . . 779

Copyrighted material

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 . . . 881

Copyrighted material

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 909

Copyrighted material

List of Exercises

Exercise 1.1: Compute 1+1 . . . 42
Exercise 1.2: Write a Hello World program . . . 43
Exercise 1.3: Derive and compute a formula . . . 43
Exercise 1.4: Convert from meters to British length units . . . 43
Exercise 1.5: Compute the mass of various substances . . . 43
Exercise 1.6: Compute the growth of money in a bank . . . 43
Exercise 1.7: Find error(s) in a program . . . 43
Exercise 1.8: Type in program text . . . 43
Exercise 1.9: Type in programs and debug them . . . 44
Exercise 1.10: Evaluate a Gaussian function . . . 45
Exercise 1.11: Compute the air resistance on a football . . . 45
Exercise 1.12: How to cook the perfect egg . . . 46
Exercise 1.13: Derive the trajectory of a ball . . . 46
Exercise 1.14: Find errors in the coding of formulas . . . 47
Exercise 1.15: Explain why a program does not work . . . 47
Exercise 1.16: Find errors in Python statements . . . 48
Exercise 1.17: Find errors in the coding of a formula . . . 48
Exercise 1.18: Find errors in a program . . . 49
Exercise 2.1: Make a Fahrenheit-Celsius conversion table . . . 82
Exercise 2.2: Generate an approximate Fahrenheit-Celsius conversion table . 82
Exercise 2.3: Work with a list . . . 82
Exercise 2.4: Generate odd numbers . . . 82
Exercise 2.5: Compute the sum of the first n integers . . . 82
Exercise 2.6: Compute energy levels in an atom . . . 82
Exercise 2.7: Generate equally spaced coordinates . . . 83
Exercise 2.8: Make a table of values from a formula . . . 83
Exercise 2.9: Store values from a formula in lists . . . 83
Exercise 2.10: Simulate operations on lists by hand . . . 84
Exercise 2.11: Compute a mathematical sum . . . 84
Exercise 2.12: Replace a while loop by a for loop . . . 84
Exercise 2.13: Simulate a program by hand . . . 85
Exercise 2.14: Explore Python documentation . . . 85
Exercise 2.15: Index a nested list . . . 85
Exercise 2.16: Store data in lists . . . 86

Copyrighted material

Exercise 2.17: Store data in a nested list . . . 86
Exercise 2.18: Values of boolean expressions . . . 86
Exercise 2.19: Explore round-off errors from a large number of inverse operations . . . 87
Exercise 2.20: Explore what zero can be on a computer . . . 87
Exercise 2.21: Compare two real numbers with a tolerance . . . 87
Exercise 2.22: Interpret a code . . . 88
Exercise 2.23: Explore problems with inaccurate indentation . . . 88
Exercise 2.24: Explore punctuation in Python programs . . . 89
Exercise 2.25: Investigate a for loop over a changing list . . . 89
Exercise 3.1: Implement a simple mathematical function . . . 127
Exercise 3.2: Implement a simple mathematical function with a parameter . . . 127
Exercise 3.3: Explain how a program works . . . 128
Exercise 3.4: Write a Fahrenheit-Celsius conversion functions . . . 128
Exercise 3.5: Write a test function for Exercise 3.4 . . . 128
Exercise 3.6: Given a test function, write the function . . . 128
Exercise 3.7: Evaluate a sum and write a test function . . . 129
Exercise 3.8: Write a function for solving $ax^2 + bx + c = 0$ . . . 129
Exercise 3.9: Implement the sum function . . . 129
Exercise 3.10: Compute a polynomial via a product . . . 130
Exercise 3.11: Integrate a function by the Trapezoidal rule . . . 130
Exercise 3.12: Derive the general Midpoint integration rule . . . 131
Exercise 3.13: Make an adaptive Trapezoidal rule . . . 132
Exercise 3.14: Simulate a program by hand . . . 133
Exercise 3.15: Debug a given test function . . . 134
Exercise 3.16: Compute the area of an arbitrary triangle . . . 134
Exercise 3.17: Compute the length of a path . . . 135
Exercise 3.18: Approximate π . . . 135
Exercise 3.19: Compute the area of a polygon . . . 135
Exercise 3.20: Write functions . . . 136
Exercise 3.21: Approximate a function by a sum of sines . . . 136
Exercise 3.22: Implement a Gaussian function . . . 137
Exercise 3.23: Wrap a formula in a function . . . 137
Exercise 3.24: Write a function for numerical differentiation . . . 137
Exercise 3.25: Implement the factorial function . . . 137
Exercise 3.26: Compute velocity and acceleration from 1D position data . . . 138
Exercise 3.27: Find the max and min values of a function . . . 138
Exercise 3.28: Find the max and min elements in a list . . . 139
Exercise 3.29: Implement the Heaviside function . . . 139
Exercise 3.30: Implement a smoothed Heaviside function . . . 139
Exercise 3.31: Implement an indicator function . . . 140
Exercise 3.32: Implement a piecewise constant function . . . 140
Exercise 3.33: Apply indicator functions . . . 141
Exercise 3.34: Test your understanding of branching . . . 141
Exercise 3.35: Simulate nested loops by hand . . . 141
Exercise 3.36: Rewrite a mathematical function . . . 142
Exercise 3.37: Make a table for approximations of $\cos x$ . . . 142
Exercise 3.38: Use None in keyword arguments . . . 143

Copyrighted material

Exercise 3.39: Write a sort function for a list of 4-tuples 143
Exercise 3.40: Find prime numbers 145
Exercise 3.41: Find pairs of characters 145
Exercise 3.42: Count substrings 145
Exercise 3.43: Resolve a problem with a function 145
Exercise 3.44: Determine the types of some objects 146
Exercise 3.45: Find an error in a program 146
Exercise 4.1: Make an interactive program 216
Exercise 4.2: Read a number from the command line 216
Exercise 4.3: Read a number from a file 216
Exercise 4.4: Read and write several numbers from and to file 217
Exercise 4.5: Use exceptions to handle wrong input 217
Exercise 4.6: Read input from the keyboard 217
Exercise 4.7: Read input from the command line 217
Exercise 4.8: Try MSWord or LibreOffice to write a program 218
Exercise 4.9: Prompt the user for input to a formula 218
Exercise 4.10: Read parameters in a formula from the command line 218
Exercise 4.11: Use exceptions to handle wrong input 218
Exercise 4.12: Test validity of input data 219
Exercise 4.13: Raise an exception in case of wrong input 219
Exercise 4.14: Evaluate a formula for data in a file 219
Exercise 4.15: Write a function given its test function 219
Exercise 4.16: Compute the distance it takes to stop a car 220
Exercise 4.17: Look up calendar functionality 221
Exercise 4.18: Use the StringFunction tool 221
Exercise 4.19: Why we test for specific exception types 221
Exercise 4.20: Make a complete module 221
Exercise 4.21: Organize a previous program as a module 222
Exercise 4.22: Read options and values from the command line 222
Exercise 4.23: Check if mathematical identities hold 222
Exercise 4.24: Compute probabilities with the binomial distribution 224
Exercise 4.25: Compute probabilities with the Poisson distribution 224
Exercise 5.1: Fill lists with function values 313
Exercise 5.2: Fill arrays; loop version 313
Exercise 5.3: Fill arrays; vectorized version 313
Exercise 5.4: Plot a function 313
Exercise 5.5: Apply a function to a vector 313
Exercise 5.6: Simulate by hand a vectorized expression 313
Exercise 5.7: Demonstrate array slicing 314
Exercise 5.8: Replace list operations by array computing 314
Exercise 5.9: Plot a formula 314
Exercise 5.10: Plot a formula for several parameters 314
Exercise 5.11: Specify the extent of the axes in a plot 314
Exercise 5.12: Plot exact and inexact Fahrenheit-Celsius conversion formulas 314
Exercise 5.13: Plot the trajectory of a ball 314
Exercise 5.14: Plot data in a two-column file 315
Exercise 5.15: Write function data to file 315
Exercise 5.16: Plot data from a file 316

Copyrighted material

Exercise 5.17: Write table to file . . . 316
Exercise 5.18: Fit a polynomial to data points . . . 317
Exercise 5.19: Fit a polynomial to experimental data . . . 318
Exercise 5.20: Read acceleration data and find velocities . . . 318
Exercise 5.21: Read acceleration data and plot velocities . . . 319
Exercise 5.22: Plot a trip's path and velocity from GPS coordinates . . . 319
Exercise 5.23: Vectorize the Midpoint rule for integration . . . 320
Exercise 5.24: Vectorize a function for computing the area of a polygon . . . 320
Exercise 5.25: Implement Lagrange's interpolation formula . . . 321
Exercise 5.26: Plot Lagrange's interpolating polynomial . . . 321
Exercise 5.27: Investigate the behavior of Lagrange's interpolating polynomials 322
Exercise 5.28: Plot a wave packet . . . 322
Exercise 5.29: Judge a plot . . . 322
Exercise 5.30: Plot the viscosity of water . . . 323
Exercise 5.31: Explore a complicated function graphically . . . 323
Exercise 5.32: Plot Taylor polynomial approximations to sin x . . . 323
Exercise 5.33: Animate a wave packet . . . 324
Exercise 5.34: Animate a smoothed Heaviside function . . . 324
Exercise 5.35: Animate two-scale temperature variations . . . 324
Exercise 5.36: Use non-uniformly distributed coordinates for visualization . . . 325
Exercise 5.37: Animate a sequence of approximations to π . . . 325
Exercise 5.38: Animate a planet's orbit . . . 325
Exercise 5.39: Animate the evolution of Taylor polynomials . . . 326
Exercise 5.40: Plot the velocity profile for pipeflow . . . 327
Exercise 5.41: Plot sum-of-sines approximations to a function . . . 327
Exercise 5.42: Animate the evolution of a sum-of-sine approximation to a function . . . 327
Exercise 5.43: Plot functions from the command line . . . 328
Exercise 5.44: Improve command-line input . . . 328
Exercise 5.45: Demonstrate energy concepts from physics . . . 328
Exercise 5.46: Plot a w-like function . . . 328
Exercise 5.47: Plot a piecewise constant function . . . 329
Exercise 5.48: Vectorize a piecewise constant function . . . 329
Exercise 5.49: Visualize approximations in the Midpoint integration rule . . . 329
Exercise 5.50: Visualize approximations in the Trapezoidal integration rule . . . 330
Exercise 5.51: Experience overflow in a function . . . 330
Exercise 5.52: Apply a function to a rank 2 array . . . 331
Exercise 5.53: Explain why array computations fail . . . 331
Exercise 5.54: Verify linear algebra results . . . 331
Exercise 6.1: Make a dictionary from a table . . . 402
Exercise 6.2: Explore syntax differences: lists vs. dicts . . . 402
Exercise 6.3: Use string operations to improve a program . . . 403
Exercise 6.4: Interpret output from a program . . . 403
Exercise 6.5: Make a dictionary . . . 403
Exercise 6.6: Make a nested dictionary . . . 404
Exercise 6.7: Make a nested dictionary from a file . . . 404
Exercise 6.8: Make a nested dictionary from a file . . . 404
Exercise 6.9: Compute the area of a triangle . . . 405

Copyrighted material

Exercise 6.10: Compare data structures for polynomials 405
Exercise 6.11: Compute the derivative of a polynomial 405
Exercise 6.12: Specify functions on the command line 405
Exercise 6.13: Interpret function specifications 406
Exercise 6.14: Compare average temperatures in cities 407
Exercise 6.15: Generate an HTML report with figures 407
Exercise 6.16: Allow different types for a function argument 408
Exercise 6.17: Make a function more robust 408
Exercise 6.18: Find proportion of bases inside/outside exons 408
Exercise 7.1: Make a function class 470
Exercise 7.2: Add a data attribute to a class 471
Exercise 7.3: Add functionality to a class 471
Exercise 7.4: Make classes for a rectangle and a triangle 471
Exercise 7.5: Make a class for quadratic functions 472
Exercise 7.6: Make a class for straight lines 472
Exercise 7.7: Flexible handling of function arguments 472
Exercise 7.8: Wrap functions in a class 473
Exercise 7.9: Flexible handling of function arguments 473
Exercise 7.10: Deduce a class implementation 474
Exercise 7.11: Implement special methods in a class 474
Exercise 7.12: Make a class for summation of series 474
Exercise 7.13: Apply a numerical differentiation class 475
Exercise 7.14: Implement an addition operator 475
Exercise 7.15: Implement in-place += and -= operators 476
Exercise 7.16: Implement a class for numerical differentiation 476
Exercise 7.17: Examine a program 477
Exercise 7.18: Modify a class for numerical differentiation 477
Exercise 7.19: Make a class for the Heaviside function 478
Exercise 7.20: Make a class for the indicator function 478
Exercise 7.21: Make a class for piecewise constant functions 479
Exercise 7.22: Speed up repeated integral calculations 479
Exercise 7.23: Apply a class for polynomials 480
Exercise 7.24: Find a bug in a class for polynomials 480
Exercise 7.25: Implement subtraction of polynomials 480
Exercise 7.26: Test the functionality of pretty print of polynomials 480
Exercise 7.27: Vectorize a class for polynomials 481
Exercise 7.28: Use a dict to hold polynomial coefficients 481
Exercise 7.29: Extend class Vec2D to work with lists/tuples 482
Exercise 7.30: Extend class Vec2D to 3D vectors 483
Exercise 7.31: Use NumPy arrays in class Vec2D 483
Exercise 7.32: Impreciseness of interval arithmetics 484
Exercise 7.33: Make classes for students and courses 484
Exercise 7.34: Find local and global extrema of a function 484
Exercise 7.35: Find the optimal production for a company 485
Exercise 8.1: Flip a coin times 549
Exercise 8.2: Compute a probability 550
Exercise 8.3: Choose random colors 550
Exercise 8.4: Draw balls from a hat 550

Copyrighted material

Exercise 8.5: Computing probabilities of rolling dice 550
Exercise 8.6: Estimate the probability in a dice game 550
Exercise 8.7: Compute the probability of hands of cards 551
Exercise 8.8: Decide if a dice game is fair 551
Exercise 8.9: Adjust a game to make it fair 551
Exercise 8.10: Make a test function for Monte Carlo simulation 551
Exercise 8.11: Generalize a game 551
Exercise 8.12: Compare two playing strategies 552
Exercise 8.13: Investigate strategies in a game 552
Exercise 8.14: Investigate the winning chances of some games 552
Exercise 8.15: Compute probabilities of throwing two dice 553
Exercise 8.16: Vectorize flipping a coin 553
Exercise 8.17: Vectorize a probablility computation 553
Exercise 8.18: Throw dice and compute a small probability 553
Exercise 8.19: Is democracy reliable as a decision maker? 553
Exercise 8.20: Difference equation for random numbers 555
Exercise 8.21: Make a class for drawing balls from a hat 555
Exercise 8.22: Independent versus dependent random numbers 555
Exercise 8.23: Compute the probability of flipping a coin 556
Exercise 8.24: Simulate binomial experiments 557
Exercise 8.25: Simulate a poker game 557
Exercise 8.26: Estimate growth in a simulation model 557
Exercise 8.27: Investigate guessing strategies 557
Exercise 8.28: Vectorize a dice game 558
Exercise 8.29: Compute π by a Monte Carlo method 558
Exercise 8.30: Compute π by a Monte Carlo method 558
Exercise 8.31: Compute π by a random sum 558
Exercise 8.32: 1D random walk with drift 558
Exercise 8.33: 1D random walk until a point is hit 558
Exercise 8.34: Simulate making a fortune from gaming 559
Exercise 8.35: Simulate pollen movements as a 2D random walk 559
Exercise 8.36: Make classes for 2D random walk 559
Exercise 8.37: 2D random walk with walls; scalar version 560
Exercise 8.38: 2D random walk with walls; vectorized version 560
Exercise 8.39: Simulate mixing of gas molecules 561
Exercise 8.40: Simulate slow mixing of gas molecules 561
Exercise 8.41: Guess beer brands 561
Exercise 8.42: Simulate stock prices 562
Exercise 8.43: Compute with option prices in finance 562
Exercise 8.44: Differentiate noise measurements 563
Exercise 8.45: Differentiate noisy signals 564
Exercise 8.46: Model noise in a time signal 565
Exercise 8.47: Speed up Markov chain mutation 566
Exercise 9.1: Demonstrate the magic of inheritance 635
Exercise 9.2: Make polynomial subclasses of parabolas 635
Exercise 9.3: Implement a class for a function as a subclass 636
Exercise 9.4: Create an alternative class hierarchy for polynomials 636
Exercise 9.5: Make circle a subclass of an ellipse 636

Copyrighted material

Exercise 9.6: Make super- and subclass for a point 636
Exercise 9.7: Modify a function class by subclassing 637
Exercise 9.8: Explore the accuracy of difference formulas 637
Exercise 9.9: Implement a subclass 637
Exercise 9.10: Make classes for numerical differentiation 638
Exercise 9.11: Implement a new subclass for differentiation 638
Exercise 9.12: Understand if a class can be used recursively 638
Exercise 9.13: Represent people by a class hierarchy 638
Exercise 9.14: Add a new class in a class hierarchy 639
Exercise 9.15: Compute convergence rates of numerical integration methods . 640
Exercise 9.16: Add common functionality in a class hierarchy 641
Exercise 9.17: Make a class hierarchy for root finding 641
Exercise 9.18: Make a calculus calculator class 641
Exercise 9.19: Compute inverse functions 642
Exercise 9.20: Make line drawing of a person; program 643
Exercise 9.21: Make line drawing of a person; class 643
Exercise 9.22: Animate a person with waving hands 643
Exercise A.1: Determine the limit of a sequence 671
Exercise A.2: Compute π via sequences 672
Exercise A.3: Reduce memory usage of difference equations 672
Exercise A.4: Compute the development of a loan 672
Exercise A.5: Solve a system of difference equations 672
Exercise A.6: Modify a model for fortune development 672
Exercise A.7: Change index in a difference equation 673
Exercise A.8: Construct time points from dates 673
Exercise A.9: Visualize the convergence of Newton's method 674
Exercise A.10: Implement the secant method 674
Exercise A.11: Test different methods for root finding 675
Exercise A.12: Make difference equations for the Midpoint rule 675
Exercise A.13: Compute the arc length of a curve 675
Exercise A.14: Find difference equations for computing $\sin x$ 676
Exercise A.15: Find difference equations for computing $\cos x$ 677
Exercise A.16: Make a guitar-like sound 677
Exercise A.17: Damp the bass in a sound file 677
Exercise A.18: Damp the treble in a sound file 678
Exercise A.19: Demonstrate oscillatory solutions of the logistic equation . . . 678
Exercise A.20: Automate computer experiments 679
Exercise A.21: Generate an HTML report 679
Exercise A.22: Use a class to archive and report experiments 680
Exercise A.23: Explore logistic growth interactively 681
Exercise A.24: Simulate the price of wheat 681
Exercise B.1: Interpolate a discrete function 709
Exercise B.2: Study a function for different parameter values 710
Exercise B.3: Study a function and its derivative 710
Exercise B.4: Use the Trapezoidal method 711
Exercise B.5: Compute a sequence of integrals 711
Exercise B.6: Use the Trapezoidal method 712
Exercise B.7: Compute trigonometric integrals 712

Copyrighted material

Exercise B.8: Plot functions and their derivatives . . . 713
Exercise B.9: Use the Trapezoidal method . . . 714
Exercise C.1: Solve a nonhomogeneous linear ODE . . . 729
Exercise C.2: Solve a nonlinear ODE . . . 729
Exercise C.3: Solve an ODE for $y(x)$ . . . 729
Exercise C.4: Experience instability of an ODE . . . 730
Exercise C.5: Solve an ODE with time-varying growth . . . 730
Exercise D.1: Model sudden movements of the plate . . . 755
Exercise D.2: Write a callback function . . . 756
Exercise D.3: Improve input to the simulation program . . . 756
Exercise E.1: Solve a simple ODE with function-based code . . . 802
Exercise E.2: Solve a simple ODE with class-based code . . . 802
Exercise E.3: Solve a simple ODE with the ODEsolver hierarchy . . . 802
Exercise E.4: Solve an ODE specified on the command line . . . 802
Exercise E.5: Implement a numerical method for ODEs . . . 803
Exercise E.6: Solve an ODE for emptying a tank . . . 803
Exercise E.7: Solve an ODE for the arc length . . . 804
Exercise E.8: Simulate a falling or rising body in a fluid . . . 804
Exercise E.9: Verify the limit of a solution as time grows . . . 805
Exercise E.10: Scale the logistic equation . . . 806
Exercise E.11: Compute logistic growth with time-varying carrying capacity . 806
Exercise E.12: Solve an ODE until constant solution . . . 807
Exercise E.13: Use a problem class to hold data about an ODE . . . 807
Exercise E.14: Derive and solve a scaled ODE problem . . . 808
Exercise E.15: Clean up a file to make it a module . . . 809
Exercise E.16: Simulate radioactive decay . . . 809
Exercise E.17: Compute inverse functions by solving an ODE . . . 809
Exercise E.18: Make a class for computing inverse functions . . . 810
Exercise E.19: Add functionality to a class . . . 810
Exercise E.20: Compute inverse functions by interpolation . . . 811
Exercise E.21: Code the 4th-order Runge-Kutta method; function . . . 811
Exercise E.22: Code the 4th-order Runge-Kutta method; class . . . 811
Exercise E.23: Compare ODE methods . . . 811
Exercise E.24: Code a test function for systems of ODEs . . . 812
Exercise E.25: Code Heun's method for ODE systems; function . . . 812
Exercise E.26: Code Heun's method for ODE systems; class . . . 812
Exercise E.27: Implement and test the Leapfrog method . . . 812
Exercise E.28: Implement and test an Adams-Bashforth method . . . 813
Exercise E.29: Solve two coupled ODEs for radioactive decay . . . 813
Exercise E.30: Implement a 2nd-order Runge-Kutta method; function . . . 813
Exercise E.31: Implement a 2nd-order Runge-Kutta method; class . . . 813
Exercise E.32: Code the iterated midpoint method; function . . . 814
Exercise E.33: Code the iterated midpoint method; class . . . 814
Exercise E.34: Make a subclass for the iterated midpoint method . . . 815
Exercise E.35: Compare the accuracy of various methods for ODEs . . . 815
Exercise E.36: Animate how various methods for ODEs converge . . . 815
Exercise E.37: Study convergence of numerical methods for ODEs . . . 815
Exercise E.38: Find a body's position along with its velocity . . . 816

Copyrighted material

Exercise E.39: Add the effect of air resistance on a ball 816
Exercise E.40: Solve an ODE system for an electric circuit 817
Exercise E.41: Simulate the spreading of a disease by a SIR model 817
Exercise E.42: Introduce problem and solver classes in the SIR model 819
Exercise E.43: Introduce vaccination in a SIR model 820
Exercise E.44: Introduce a vaccination campaign in a SIR model 820
Exercise E.45: Find an optimal vaccination period 821
Exercise E.46: Simulate human-zombie interaction 821
Exercise E.47: Simulate a zombie movie 823
Exercise E.48: Simulate a war on zombies 824
Exercise E.49: Explore predator-prey population interactions 824
Exercise E.50: Formulate a 2nd-order ODE as a system 825
Exercise E.51: Solve $\ddot{u} + u = 0$ 826
Exercise E.52: Make a tool for analyzing oscillatory solutions 826
Exercise E.53: Implement problem, solver, and visualizer classes 827
Exercise E.54: Use classes for flexible choices of models 831
Exercise E.55: Apply software for oscillating systems 831
Exercise E.56: Model the economy of fishing 832

Copyrighted material

1 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 e^x and $\sin x$, and how to use Python for interactive calculations.

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.

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.

The folder structure with example programs can also be directly accessed in a GitHub repository[1] 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...*.

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:

$$y(t) = v_0 t - \frac{1}{2} g t^2 . \tag{1.1}$$

[1] http://tinyurl.com/pwyasaa

© Springer-Verlag Berlin Heidelberg 2016

H.P. Langtangen, *A Primer on Scientific Programming with Python*,
Texts in Computational Science and Engineering 6, DOI 10.1007/978-3-662-49887-3_1

Copyrighted material

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 $y = 0$ when $t = 0$. The above formula neglects air resistance, which is usually small unless v_0 is large, see Exercise 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 $y = 0$:

$$v_0 t - \frac{1}{2}gt^2 = t(v_0 - \frac{1}{2}gt) = 0 \quad \Rightarrow \quad t = 0 \text{ or } t = 2v_0/g\,.$$

That is, the ball returns after $2v_0/g$ seconds, and it is therefore reasonable to restrict the interest of (1.1) to $t \in [0, 2v_0/g]$.

1.1.1 Using a Program as a Calculator

Our first program will evaluate (1.1) for a specific choice of v_0, g, and t. Choosing $v_0 = 5$ m/s and $g = 9.81$ m/s^2 makes the ball come back after $t = 2v_0/g \approx 1$ s. This means that we are basically interested in the time interval $[0, 1]$. Say we want to compute the height of the ball at time $t = 0.6$ s. From (1.1) we have

$$y = 5 \cdot 0.6 - \frac{1}{2} \cdot 9.81 \cdot 0.6^2 \tag{1.2}$$

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., 0.6^2 is written as `0.6**2`.

Our task now is to create the program and run it, and this will be described next.

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.

Another perception of the word *program* is a file that can be run ("double-clicked") 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

Copyrighted material

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.

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.

1.1.3 Tools for Writing Programs

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

- a plain text editor
- an integrated development environment (IDE) with a text editor
- an IPython notebook

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.

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

Copyrighted material

- 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.

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:

```
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:

Copyrighted material

Terminal

```
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.

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

```
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 6^2. 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:

> *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.

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.

Copyrighted material

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.

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 v_0, 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:

```
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.

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.

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

Copyrighted material

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:

```
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!).

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 v_0 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`.

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`.

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 λ in the program, you may

Copyrighted material

work with lambda_ as variable name. See Exercise 1.16 for examples on legal and illegal variable names.

Program files can have a freely 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.

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:

```
# 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.

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.

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:

```
# -*- coding: utf-8 -*-
```

Copyrighted material

(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

```
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.

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. 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.

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)
```

Copyrighted material

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 $1.2432 \cdot 10^{-3}$. 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 index[2] of the Python Standard Library online documentation.

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
```

[2] http://docs.python.org/2/genindex.html

Copyrighted material

```
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

Copyrighted material

```
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 $\sin x$, $\cos x$, $\ln x$, e^x, $\sinh x$, $\sin^{-1} x$, 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

Copyrighted material

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 v_0, 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 Wikipedia[3].

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
```

[3] http://en.wikipedia.org/wiki/Software_bug#Etymology

Copyrighted material

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 ($x + 2 = 5$) or a definition ($f(x) = x^2 + 1$). 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 left-hand 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,

Copyrighted material

```
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, v_0, 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.

Copyrighted material

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:

$$F = \frac{9}{5}C + 32 \tag{1.3}$$

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
```

Copyrighted material

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: $\frac{9}{5} \cdot 21 + 32$ 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 $bc \leq a$. This implies that $9/5$ becomes 1 since $1 \cdot 5 = 5 \leq 9$ while $2 \cdot 5 = 10 > 9$. Another example is $1/5$, which becomes 0 since $0 \cdot 5 \leq 1$ (and $1 \cdot 5 > 1$). Yet another example is $16/6$, which results in 2 (try $2 \cdot 6$ and $3 \cdot 6$ 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
```

Copyrighted material

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
```

Copyrighted material

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 $\frac{1}{2}gt^2$ as `0.5*g*t**2` wrote `(1/2)*g*t**2`. This term would then always be zero!

Copyrighted material

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 a^b, 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 a^4 (`a**4`) is evaluated, then 2 is multiplied with a^4, 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 $\frac{5}{9+2}$ 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 v_0:

$$y_c = v_0 t - \frac{1}{2} g t^2,$$

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 y_c? The answer is straightforward to derive. When $y = y_c$ we have

$$y_c = v_0 t - \frac{1}{2} g t^2 .$$

Copyrighted material

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

$$\frac{1}{2}gt^2 - v_0t + y_c = 0,$$

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

$$t_1 = \left(v_0 - \sqrt{v_0^2 - 2gy_c}\right)/g, \quad t_2 = \left(v_0 + \sqrt{v_0^2 - 2gy_c}\right)/g\,. \tag{1.4}$$

There are two solutions because the ball reaches the height y_c on its way up ($t = t_1$) and on its way down ($t = t_2 > t_1$).

The program To evaluate the expressions for t_1 and t_2 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 t_1 and t_2:

```
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

Copyrighted material

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
```

Copyrighted material

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:

$$\sinh(x) = \frac{1}{2}\left(e^{x} - e^{-x}\right) . \tag{1.5}$$

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 Wikipedia[4]. For now the

[4] http://en.wikipedia.org/wiki/Floating_point_number

Copyrighted material

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: $1/49 \cdot 49$ and $1/51 \cdot 51$. 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
```

Copyrighted material

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 $4 \cdot 10^{-16}$.

Copyrighted material

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)
<type 'int'>
```

We convert this int object to a corresponding float object:

```
>>> C = float(C)   # type conversion
>>> type(C)
<type 'float'>
>>> 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)
<type 'float'>
>>> D = int(C)      # type conversion
>>> type(D)
<type 'int'>
>>> 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
```

Copyrighted material

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
```

Copyrighted material

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
```

Copyrighted material

```
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 $x^2 = 2$. Then most of us are able to find out that $x = \sqrt{2}$ is a solution to the equation. The more mathematically interested reader will also remark that $x = -\sqrt{2}$ is another solution. But faced with the equation $x^2 = -2$, 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 $a+bi$, or $a + ib$, where i is called the imaginary unit and acts as a label for the second term. Mathematically, $i = \sqrt{-1}$. An important feature of complex numbers is definitely the ability to compute square roots of negative numbers. For example, $\sqrt{-2} = \sqrt{2}i$ (i.e., $\sqrt{2}\sqrt{-1}$). The solutions of $x^2 = -2$ are thus $x_1 = +\sqrt{2}i$ and $x_2 = -\sqrt{2}i$.

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 $\sin z$, $\cos z$, $\tan z$, e^z, $\ln z$, $\sinh z$, $\cosh z$, $\tanh z$, etc. for a complex number $z = a + ib$. We assume in the following that you are familiar with the mathematics of complex numbers, at least to the degree

Copyrighted material

encountered in the program examples.

$$\text{let } u = a + bi \text{ and } v = c + di$$

The following rules reflect complex arithmetics:

$$
\begin{aligned}
u = v \quad &\Rightarrow \quad a = c,\ b = d \\
-u &= -a - bi \\
u^* &\equiv a - bi \quad \text{(complex conjugate)} \\
u + v &= (a + c) + (b + d)i \\
u - v &= (a - c) + (b - d)i \\
uv &= (ac - bd) + (bc + ad)i \\
u/v &= \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2} i \\
|u| &= \sqrt{a^2 + b^2} \\
e^{iq} &= \cos q + i \sin q
\end{aligned}
$$

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 $2-3i$ 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:

Copyrighted material

```
>>> 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 "<input>", 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 $\sin(ai) = i \sinh a$ holds:

```
>>> from cmath import sin, sinh
>>> r1 = sin(8j)
>>> r1
1490.4788257895502j
>>> r2 = 1j*sinh(8)
>>> r2
1490.4788257895502j
```

Another relation, $e^{iq} = \cos q + i \sin q$, 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 *
```

Copyrighted material

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 "<input>", 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 $f(x) = ax^2 + bx + c$:

Copyrighted material

```
>>> 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 $v_0t - \frac{1}{2}gt^2$ with respect to t and integrate the answer to get the formula back:

Copyrighted material

```
>>> 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 $y = 0$:

```
>>> 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

Copyrighted material

```
>>> 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 e^t and $e^{\sin(t)}$ 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!

Copyrighted material

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)
```

Copyrighted material

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 $\frac{a}{b+c}$ 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
```

Copyrighted material

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

Copyrighted material

- 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 v_0 making an angle θ 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 $y = f(x)$ through the air where

$$f(x) = x \tan\theta - \frac{1}{2v_0^2}\frac{gx^2}{\cos^2\theta} + y_0\,. \tag{1.6}$$

In this expression, x is a horizontal coordinate, g is the acceleration of gravity, v_0 is the size of the initial velocity that makes an angle θ with the x axis, and $(0, y_0)$ 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 ($v_0 = 30$ 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 v_0 is given in km/h; $g = 9.81\text{m/s}^2$; x, y, and y_0 are measured in meters; and θ in degrees. The program has naturally four parts: initialization of input data, import of functions and π from `math`, conversion of v_0 and θ 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
```

Copyrighted material

```
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)
```

Copyrighted material

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 Code[5], 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

[5] http://www.python.org/dev/peps/pep-0008/

Copyrighted material

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 $1+1$ to a variable and print the value of that variable.
Filename: `1plus1`.

Copyrighted material

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 (10^9) 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 $\varrho = m/V$, 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/cm^3 are found in the file `src/files/densities.dat`[6]: 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

$$A\left(1 + \frac{p}{100}\right)^n$$

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.

[6] http://tinyurl.com/pwyasaa/files/densities.dat

Copyrighted material

```
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 $\sin^2(x) + \cos^2(x) = 1$?

```
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 $s = v_0 t + \frac{1}{2}at^2$, with $v_0 = 3$ m/s, $t = 1$ s, $a = 2$ m/s^2.

```
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:

$$(a + b)^2 = a^2 + 2ab + b^2$$
$$(a - b)^2 = a^2 - 2ab + b^2$$

```
a = 3,3   b = 5,3
a2 = a**2
b2 = b**2
```

Copyrighted material

```
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,

$$f(x) = \frac{1}{\sqrt{2\pi}\, s} \exp\left[-\frac{1}{2}\left(\frac{x-m}{s}\right)^2\right], \tag{1.7}$$

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 $m = 0$, $s = 2$, and $x = 1$. 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 Gauss[7], 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

$$F_d = \frac{1}{2} C_D \varrho A V^2, \tag{1.8}$$

where ϱ is the density of the air, V is the velocity of the object, A is the cross-sectional area (normal to the velocity direction), and C_D 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 $F_g = mg$, where $g = 9.81\,\mathrm{m\,s^{-2}}$.

We can use the formulas for F_d and F_g to study the importance of air resistance versus gravity when kicking a football. The density of air is $\varrho = 1.2$ kg m^{-3}. We have $A = \pi a^2$ for any ball with radius a. For a football, $a = 11$ cm and the mass is 0.43 kg. The drag coefficient C_D 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 ($\mathrm{N} = \mathrm{kg\,m/s^2}$). Also print the ratio of the drag force and the gravity force. Define C_D, ϱ, A, V, m, g,

[7] http://en.wikipedia.org/wiki/Carl_Gauss

Copyrighted material

F_d, and F_g as variables, and put a comment with the corresponding unit. Use the program to calculate the forces on the ball for a hard kick, $V = 120$ km/h and for a soft kick, $V = 30$ 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 °C, while in the yolk the proteins start to coagulate for temperatures above 70 °C. For a soft boiled egg, the white needs to have been heated long enough to coagulate at a temperature above 63 °C, but the yolk should not be heated above 70 °C. For a hard boiled egg, the center of the yolk should be allowed to reach 70 °C.

The following formula expresses the time t it takes (in seconds) for the center of the yolk to reach the temperature T_y (in Celsius degrees):

$$t = \frac{M^{2/3} c \rho^{1/3}}{K \pi^2 (4\pi/3)^{2/3}} \ln \left[0.76 \frac{T_o - T_w}{T_y - T_w} \right] . \tag{1.9}$$

Here, M, ρ, c, and K are properties of the egg: M is the mass, ρ is the density, c is the specific heat capacity, and K is thermal conductivity. Relevant values are $M = 47$ g for a small egg and $M = 67$ g for a large egg, $\rho = 1.038\,\mathrm{g\,cm^{-3}}$, $c = 3.7\ \mathrm{J\,g^{-1}\,K^{-1}}$, and $K = 5.4 \cdot 10^{-3}\,\mathrm{W\,cm^{-1}\,K^{-1}}$. Furthermore, T_w is the temperature (in C degrees) of the boiling water, and T_o is the original temperature (in C degrees) of the egg before being put in the water. Implement the formula in a program, set $T_w = 100$ °C and $T_y = 70$ °C, and compute t for a large egg taken from the fridge ($T_o = 4$ °C) and from room temperature ($T_o = 20$ °C).
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:

$$F_x = ma_x \tag{1.10}$$

$$F_y = ma_y \tag{1.11}$$

where F_x and F_y are the sum of forces in the x and y directions, respectively, a_x and a_y 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

Copyrighted material

that

$$a_x = \frac{d^2x}{dt^2}, \tag{1.12}$$

$$a_y = \frac{d^2y}{dt^2}. \tag{1.13}$$

If we assume that gravity is the only important force on the ball, $F_x = 0$ and $F_y = -mg$.

Integrate the two components of Newton's second law twice. Use the initial conditions on velocity and position,

$$\frac{d}{dt}x(0) = v_0 \cos\theta, \tag{1.14}$$

$$\frac{d}{dt}y(0) = v_0 \sin\theta, \tag{1.15}$$

$$x(0) = 0, \tag{1.16}$$

$$y(0) = y_0, \tag{1.17}$$

to determine the four integration constants. Write up the final expressions for $x(t)$ and $y(t)$. Show that if $\theta = \pi/2$, 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
```

Filename: `find_errors_vars`.

Copyrighted material

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,

$$ax^2 + bx + c = 0,$$

the two roots are

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}. \tag{1.18}$$

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`.

Copyrighted material

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`.

Copyrighted material

Loops and Lists

2

This chapter explains how repetitive tasks in a program can be automated by loops. We also introduce list objects for storing and processing collections of data with a specific order. Loops and lists, together with functions and `if` tests from Chap. 3, lay the fundamental programming foundation for the rest of the book. The programs associated with the chapter are found in the folder `src/looplist`[1].

2.1 While Loops

Our task now is to print out a conversion table with Celsius degrees in the first column of the table and the corresponding Fahrenheit degrees in the second column. Such a table may look like this:

```
 -20  -4.0
 -15   5.0
 -10  14.0
  -5  23.0
   0  32.0
   5  41.0
  10  50.0
  15  59.0
  20  68.0
  25  77.0
  30  86.0
  35  95.0
  40 104.0
```

2.1.1 A Naive Solution

The formula for converting C degrees Celsius to F degrees Fahrenheit is $F = 9C/5 + 32$. Since we know how to evaluate the formula for one value of C, we can just repeat these statements as many times as required for the table above. Using

[1] http://tinyurl.com/pwyasaa/looplist

© Springer-Verlag Berlin Heidelberg 2016

H.P. Langtangen, *A Primer on Scientific Programming with Python*,
Texts in Computational Science and Engineering 6, DOI 10.1007/978-3-662-49887-3_2

Copyrighted material

three statements per line in the program, for compact layout of the code, we can write the whole program as

```
C = -20;  F = 9.0/5*C + 32;  print C, F
C = -15;  F = 9.0/5*C + 32;  print C, F
C = -10;  F = 9.0/5*C + 32;  print C, F
C =  -5;  F = 9.0/5*C + 32;  print C, F
C =   0;  F = 9.0/5*C + 32;  print C, F
C =   5;  F = 9.0/5*C + 32;  print C, F
C =  10;  F = 9.0/5*C + 32;  print C, F
C =  15;  F = 9.0/5*C + 32;  print C, F
C =  20;  F = 9.0/5*C + 32;  print C, F
C =  25;  F = 9.0/5*C + 32;  print C, F
C =  30;  F = 9.0/5*C + 32;  print C, F
C =  35;  F = 9.0/5*C + 32;  print C, F
C =  40;  F = 9.0/5*C + 32;  print C, F
```

Running this program (which is stored in the file `c2f_table_repeat.py`), demonstrates that the output becomes

```
-20 -4.0
-15 5.0
-10 14.0
-5 23.0
0 32.0
5 41.0
10 50.0
15 59.0
20 68.0
25 77.0
30 86.0
35 95.0
40 104.0
```

This output suffers from somewhat ugly formatting, but that problem can quickly be fixed by replacing `print C, F` by a `print` statement based on printf formatting. We will return to this detail later.

The main problem with the program above is that lots of statements are identical and repeated. First of all it is boring to write this sort of repeated statements, especially if we want many more C and F values in the table. Second, the idea of the computer is to automate repetition. Therefore, all computer languages have constructs to efficiently express repetition. These constructs are called *loops* and come in two variants in Python: `while` loops and `for` loops. Most programs in this book employ loops, so this concept is extremely important to learn.

2.1.2 While Loops

The `while` loop is used to repeat a set of statements as long as a condition is true. We shall introduce this kind of loop through an example. The task is to generate the rows of the table of C and F values. The C value starts at -20 and is incremented

Copyrighted material

by 5 as long as $C \leq 40$. For each C value we compute the corresponding F value and write out the two temperatures. In addition, we also add a line of dashes above and below the table.

The list of tasks to be done can be summarized as follows:

- Print line with dashes
- $C = -20$
- While $C \leq 40$:
  - $F = \frac{9}{5}C + 32$
  - Print C and F
  - Increment C by 5
- Print line with dashes

This is the *algorithm* of our programming task. The way from a detailed algorithm to a fully functioning Python code can often be made very short, which is definitely true in the present case:

```
print '------------------'      # table heading
C = -20                         # start value for C
dC = 5                          # increment of C in loop
while C <= 40:                  # loop heading with condition
    F = (9.0/5)*C + 32          # 1st statement inside loop
    print C, F                  # 2nd statement inside loop
    C = C + dC                  # 3rd statement inside loop
print '------------------'      # end of table line (after loop)
```

A very important feature of Python is now encountered: the block of statements to be executed in each pass of the `while` loop must be indented. In the example above the block consists of three lines, and all these lines must have exactly the same indentation. Our choice of indentation in this book is four spaces. The first statement whose indentation coincides with that of the `while` line marks the end of the loop and is executed after the loop has terminated. In this example this is the final `print` statement. You are encouraged to type in the code above in a file, indent the last line four spaces, and observe what happens (you will experience that lines in the table are separated by a line of dashes: `----`).

Many novice Python programmers forget the colon at the end of the `while` line – this colon is essential and marks the beginning of the indented block of statements inside the loop. Later, we will see that there are many other similar program constructions in Python where there is a heading ending with a colon, followed by an indented block of statements.

Programmers need to fully understand what is going on in a program and be able to simulate the program by hand. Let us do this with the program segment above. First, we define the start value for the sequence of Celsius temperatures: `C = -20`. We also define the increment `dC` that will be added to `C` inside the loop. Then we enter the loop condition `C <= 40`. The first time `C` is `-20`, which implies that `C <= 40` (equivalent to $C \leq 40$ in mathematical notation) is true. Since the loop condition is true, we enter the loop and execute all the indented statements. That is, we compute `F` corresponding to the current `C` value, print the temperatures, and

Copyrighted material

increment C by dC. For simplicity, we have used a plain print C, F without any formatting so the columns will not be aligned, but this can easily be fixed later.

Thereafter, we enter the second pass in the loop. First we check the condition: C is -15 and C <= 40 is still true. We execute the statements in the indented loop block, C becomes -10, this is still less than or equal to 40, so we enter the loop block again. This procedure is repeated until C is updated from 40 to 45 in the final statement in the loop block. When we then test the condition, C <= 40, this condition is no longer true, and the loop is terminated. We proceed with the next statement that has the same indentation as the while statement, which is the final print statement in this example.

Newcomers to programming are sometimes confused by statements like

```
C = C + dC
```

This line looks erroneous from a mathematical viewpoint, but the statement is perfectly valid computer code, because we first evaluate the expression on the right-hand side of the equality sign and then let the variable on the left-hand side refer to the result of this evaluation. In our case, C and dC are two different int objects. The operation C+dC results in a new int object, which in the assignment C = C+dC is bound to the name C. Before this assignment, C was already bound to an int object, and this object is automatically destroyed when C is bound to a new object and there are no other names (variables) referring to this previous object (if you did not get this last point, just relax and continue reading!).

Since incrementing the value of a variable is frequently done in computer programs, there is a special short-hand notation for this and related operations:

```
C += dC   # equivalent to C = C + dC
C -= dC   # equivalent to C = C - dC
C *= dC   # equivalent to C = C*dC
C /= dC   # equivalent to C = C/dC
```

2.1.3 Boolean Expressions

In our first example on a while loop, we worked with a condition C <= 40, which evaluates to either true or false, written as True or False in Python. Other comparisons are also useful:

```
C == 40    # C equals 40
C != 40    # C does not equal 40
C >= 40    # C is greater than or equal to 40
C >  40    # C is greater than 40
C <  40    # C is less than 40
```

Not only comparisons between numbers can be used as conditions in while loops: any expression that has a boolean (True or False) value can be used. Such expressions are known as *logical* or *boolean* expressions.

The keyword not can be inserted in front of the boolean expression to change the value from True to False or from False to True. To evaluate not C ==

Copyrighted material

```
x = 1.2   # assign some value
N = 25    # maximum power in sum
k = 1
s = x
sign = 1.0
import math

while k < N:
    sign = - sign
    k = k + 2
    term = sign*x**k/math.factorial(k)
    s = s + term

print 'sin(%g) = %g (approximation with %d terms)' % (x, s, N)
```

The best way to understand such a program is to simulate it by hand. That is, we go through the statements, one by one, and write down on a piece of paper what the state of each variable is.

When the loop is first entered, `k < N` implies `1 < 25`, which is `True` so we enter the loop block. There, we compute `sign = -1.0`, `k = 3`, `term = -1.0*x**3/(3*2*1))` (note that `sign` is float so we always have `float` divided by `int`), and `s = x - x**3/6`, which equals the first two terms in the sum. Then we test the loop condition: `3 < 25` is `True` so we enter the loop block again. This time we obtain `term = 1.0*x**5/math.factorial(5)`, which correctly implements the third term in the sum. At some point, `k` is updated to from 23 to 25 inside the loop and the loop condition then becomes `25 < 25`, which is `False`, implying that the program jumps over the loop block and continues with the `print` statement (which has the same indentation as the `while` statement).

2.2 Lists

Up to now a variable has typically contained a single number. Sometimes numbers are naturally grouped together. For example, all Celsius degrees in the first column of our table from Sect. 2.1.2 could be conveniently stored together as a group. A Python *list* can be used to represent such a group of numbers in a program. With a variable that refers to the list, we can work with the whole group at once, but we can also access individual elements of the group. Figure 2.1 illustrates the difference between an `int` object and a list object. In general, a list may contain a sequence of arbitrary objects in a given order. Python has great functionality for examining and manipulating such sequences of objects, which will be demonstrated below.

2.2.1 Basic List Operations

To create a list with the numbers from the first column in our table, we just put all the numbers inside square brackets and separate the numbers by commas:

```
C = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
```

Copyrighted material

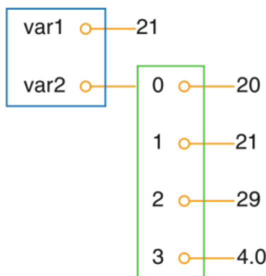

Fig. 2.1 Illustration of two variables: `var1` refers to an `int` object with value 21, created by the statement `var1 = 21`, and `var2` refers to a `list` object with value `[20, 21, 29, 4.0]`, i.e., three `int` objects and one `float` object, created by the statement `var2 = [20, 21, 29, 4.0]`

The variable `C` now refers to a `list` object holding 13 list *elements*. All list elements are in this case `int` objects.

Every element in a list is associated with an *index*, which reflects the position of the element in the list. The first element has index 0, the second index 1, and so on. Associated with the `C` list above we have 13 indices, starting with 0 and ending with 12. To access the element with index 3, i.e., the fourth element in the list, we can write `C[3]`. As we see from the list, `C[3]` refers to an `int` object with the value -5.

Elements in lists can be deleted, and new elements can be inserted anywhere. The functionality for doing this is built into the list object and accessed by a dot notation. Two examples are `C.append(v)`, which appends a new element `v` to the end of the list, and `C.insert(i,v)`, which inserts a new element `v` in position number `i` in the list. The number of elements in a list is given by `len(C)`. Let us exemplify some list operations in an interactive session to see the effect of the operations:

```
>>> C = [-10, -5, 0, 5, 10, 15, 20, 25, 30]     # create list
>>> C.append(35)                     # add new element 35 at the end
>>> C                                # view list C
[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35]
```

Two lists can be added:

```
>>> C = C + [40, 45]                 # extend C at the end
>>> C
[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

What adding two lists means is up to the list object to define, and not surprisingly, addition of two lists is defined as appending the second list to the first. The result of `C + [40,45]` is a new list object, which we then assign to `C` such that this name refers to this new list. In fact, every object in Python and everything you can do with it is defined by programs made by humans. With the techniques of class programming (see Chap. 7) you can create your own objects and define (if desired) what it means to add such objects. All this gives enormous power in the hands of

Copyrighted material

programmers. As one example, you can define your own list object if you are not satisfied with the functionality of Python's own lists.

New elements can be inserted anywhere in the list (and not only at the end as we did with `C.append`):

```
>>> C.insert(0, -15)          # insert new element -15 as index 0
>>> C
[-15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

With `del C[i]` we can remove an element with index `i` from the list `C`. Observe that this changes the list, so `C[i]` refers to another (the next) element after the removal:

```
>>> del C[2]                  # delete 3rd element
>>> C
[-15, -10, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> del C[2]                  # delete what is now 3rd element
>>> C
[-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> len(C)                    # length of list
11
```

The command `C.index(10)` returns the index corresponding to the first element with value `10` (this is the 4th element in our sample list, with index 3):

```
>>> C.index(10)               # find index for an element (10)
3
```

To just test if an object with the value `10` is an element in the list, one can write the boolean expression `10 in C`:

```
>>> 10 in C                   # is 10 an element in C?
True
```

Python allows negative indices, which leads to indexing from the right. As demonstrated below, `C[-1]` gives the last element of the list `C`. `C[-2]` is the element before `C[-1]`, and so forth.

```
>>> C[-1]                     # view the last list element
45
>>> C[-2]                     # view the next last list element
40
```

Building long lists by writing down all the elements separated by commas is a tedious process that can easily be automated by a loop, using ideas from Sect. 2.1.4. Say we want to build a list of degrees from -50 to 200 in steps of 2.5 degrees. We then start with an empty list and use a `while` loop to append one element at a time:

Copyrighted material

```
C = []
C_value = -50
C_max = 200
while C_value <= C_max:
     C.append(C_value)
     C_value += 2.5
```

In the next sections, we shall see how we can express these six lines of code with just one single statement.

There is a compact syntax for creating variables that refer to the various list elements. Simply list a sequence of variables on the left-hand side of an assignment to a list:

```
>>> somelist = ['book.tex', 'book.log', 'book.pdf']
>>> texfile, logfile, pdf = somelist
>>> texfile
'book.tex'
>>> logfile
'book.log'
>>> pdf
'book.pdf'
```

The number of variables on the left-hand side must match the number of elements in the list, otherwise an error occurs.

A final comment regards the syntax: some list operations are reached by a dot notation, as in `C.append(e)`, while other operations requires the list object as an argument to a function, as in `len(C)`. Although `C.append` for a programmer behaves as a function, it is a function that is reached through a list object, and it is common to say that `append` is a *method* in the list object, not a function. There are no strict rules in Python whether functionality regarding an object is reached through a method or a function.

2.2.2 For Loops

The nature of for loops When data are collected in a list, we often want to perform the same operations on each element in the list. We then need to walk through all list elements. Computer languages have a special construct for doing this conveniently, and this construct is in Python and many other languages called a `for` loop. Let us use a `for` loop to print out all list elements:

```
degrees = [0, 10, 20, 40, 100]
for C in degrees:
    print 'list element:', C
print 'The degrees list has', len(degrees), 'elements'
```

The `for C in degrees` construct creates a loop over all elements in the list `degrees`. In each pass of the loop, the variable `C` refers to an element in the list, starting with `degrees[0]`, proceeding with `degrees[1]`, and so on, before ending

Copyrighted material

with the last element degrees[n-1] (if n denotes the number of elements in the list, len(degrees)).

The for loop specification ends with a colon, and after the colon comes a block of statements that does something useful with the current element. Each statement in the block must be indented, as we explained for while loops. In the example above, the block belonging to the for loop contains only one statement. The final print statement has the same indentation (none in this example) as the for statement and is executed as soon as the loop is terminated.

As already mentioned, understanding all details of a program by following the program flow by hand is often a very good idea. Here, we first define a list degrees containing 5 elements. Then we enter the for loop. In the first pass of the loop, C refers to the first element in the list degrees, i.e., the int object holding the value 0. Inside the loop we then print out the text 'list element:' and the value of C, which is 0. There are no more statements in the loop block, so we proceed with the next pass of the loop. C then refers to the int object 10, the output now prints 10 after the leading text, we proceed with C as the integers 20 and 40, and finally C is 100. After having printed the list element with value 100, we move on to the statement after the indented loop block, which prints out the number of list elements. The total output becomes

```
list element: 0
list element: 10
list element: 20
list element: 40
list element: 100
The degrees list has 5 elements
```

Correct indentation of statements is crucial in Python, and we therefore strongly recommend you to work through Exercise 2.23 to learn more about this topic.

Making the table Our knowledge of lists and for loops over elements in lists puts us in a good position to write a program where we collect all the Celsius degrees to appear in the table in a list Cdegrees, and then use a for loop to compute and write out the corresponding Fahrenheit degrees. The complete program may look like this:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
for C in Cdegrees:
    F = (9.0/5)*C + 32
    print C, F
```

The print C, F statement just prints the value of C and F with a default format, where each number is separated by one space character (blank). This does not look like a nice table (the output is identical to the one shown in Sect. 2.1.1. Nice formatting is obtained by forcing C and F to be written in fields of fixed width and with a fixed number of decimals. An appropriate printf format is %5d (or %5.0f) for C and %5.1f for F. We may also add a headline to the table. The complete program becomes:

Copyrighted material

2.3.3 For Loops with List Indices

Instead of iterating over a list directly with the construction

```
for element in somelist:
    ...
```

we can equivalently iterate of the list indices and index the list inside the loop:

```
for i in range(len(somelist)):
    element = somelist[i]
    ...
```

Since `len(somelist)` returns the length of `somelist` and the largest legal index is `len(somelist)-1`, because indices always start at 0, `range(len(somelist))` will generate all the correct indices: `0`, `1`, ..., `len(somelist)-1`.

Programmers coming from other languages, such as Fortran, C, C++, Java, and C#, are very much used to `for` loops with integer counters and usually tend to use `for i in range(len(somelist))` and work with `somelist[i]` inside the loop. This might be necessary or convenient, but if possible, Python programmers are encouraged to use `for element in somelist`, which is more elegant to read.

Iterating over loop indices is useful when we need to process two lists simultaneously. As an example, we first create two `Cdegrees` and `Fdegrees` lists, and then we make a list to write out a table with `Cdegrees` and `Fdegrees` as the two columns of the table. Iterating over a loop index is convenient in the final list:

```
Cdegrees = []
n = 21
C_min = -10
C_max = 40
dC = (C_max - C_min)/float(n-1)  # increment in C
for i in range(0, n):
     C = -10 + i*dC
     Cdegrees.append(C)

Fdegrees = []
for C in Cdegrees:
    F = (9.0/5)*C + 32
    Fdegrees.append(F)

for i in range(len(Cdegrees)):
    C = Cdegrees[i]
    F = Fdegrees[i]
    print '%5.1f %5.1f' % (C, F)
```

Instead of appending new elements to the lists, we can start with lists of the right size, containing zeros, and then index the lists to fill in the right values. Creating a list of length `n` consisting of zeros (for instance) is done by

```
somelist = [0]*n
```

Copyrighted material

With this construction, the program above can use `for` loops over indices everywhere:

```
n = 21
C_min = -10
C_max = 40
dC = (C_max - C_min)/float(n-1)  # increment in C

Cdegrees = [0]*n
for i in range(len(Cdegrees)):
     Cdegrees[i] = -10 + i*dC

Fdegrees = [0]*n
for i in range(len(Cdegrees)):
    Fdegrees[i] = (9.0/5)*Cdegrees[i] + 32

for i in range(len(Cdegrees)):
    print '%5.1f %5.1f' % (Cdegrees[i], Fdegrees[i])
```

Note that we need the construction `[0]*n` to create a list of the right length, otherwise the index `[i]` will be illegal.

2.3.4 Changing List Elements

We have two seemingly alternative ways to traverse a list, either a loop over elements or over indices. Suppose we want to change the `Cdegrees` list by adding 5 to all elements. We could try

```
for c in Cdegrees:
    c += 5
```

but this loop leaves `Cdegrees` unchanged, while

```
for i in range(len(Cdegrees)):
    Cdegrees[i] += 5
```

works as intended. What is wrong with the first loop? The problem is that `c` is an ordinary variable, which refers to a list element in the loop, but when we execute `c += 5`, we let `c` refer to a new `float` object (`c+5`). This object is never inserted in the list. The first two passes of the loop are equivalent to

```
c = Cdegrees[0]   # automatically done in the for statement
c += 5
c = Cdegrees[1]   # automatically done in the for statement
c += 5
```

The variable `c` can only be used to read list elements and never to change them. Only an assignment of the form

Copyrighted material

```
Cdegrees[i] = ...
```

can change a list element.

There is a way of traversing a list where we get both the index and an element in each pass of the loop:

```
for i, c in enumerate(Cdegrees):
    Cdegrees[i] = c + 5
```

This loop also adds 5 to all elements in the list.

2.3.5 List Comprehension

Because running through a list and for each element creating a new element in another list is a frequently encountered task, Python has a special compact syntax for doing this, called *list comprehension*. The general syntax reads

```
newlist = [E(e) for e in list]
```

where `E(e)` represents an expression involving element `e`. Here are three examples:

```
Cdegrees = [-5 + i*0.5 for i in range(n)]
Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]
C_plus_5 = [C+5 for C in Cdegrees]
```

List comprehensions are recognized as a `for` loop inside square brackets and will be frequently exemplified throughout the book.

2.3.6 Traversing Multiple Lists Simultaneously

We may use the `Cdegrees` and `Fdegrees` lists to make a table. To this end, we need to traverse both arrays. The `for element in list` construction is not suitable in this case, since it extracts elements from one list only. A solution is to use a `for` loop over the integer indices so that we can index both lists:

```
for i in range(len(Cdegrees)):
    print '%5d %5.1f' % (Cdegrees[i], Fdegrees[i])
```

It happens quite frequently that two or more lists need to be traversed simultaneously. As an alternative to the loop over indices, Python offers a special nice syntax that can be sketched as

```
for e1, e2, e3, ... in zip(list1, list2, list3, ...):
    # work with element e1 from list1, element e2 from list2,
    # element e3 from list3, etc.
```

Copyrighted material

The `zip` function turns n lists (`list1, list2, list3, ...`) into one list of n-tuples, where each n-tuple `(e1,e2,e3,...)` has its first element (`e1`) from the first list (`list1`), the second element (`e2`) from the second list (`list2`), and so forth. The loop stops when the end of the shortest list is reached. In our specific case of iterating over the two lists `Cdegrees` and `Fdegrees`, we can use the `zip` function:

```
for C, F in zip(Cdegrees, Fdegrees):
    print '%5d %5.1f' % (C, F)
```

It is considered more *Pythonic* to iterate over list elements, here `C` and `F`, rather than over list indices as in the `for i in range(len(Cdegrees))` construction.

2.4 Nested Lists

Nested lists are list objects where the elements in the lists can be lists themselves. A couple of examples will motivate for nested lists and illustrate the basic operations on such lists.

2.4.1 A table as a List of Rows or Columns

Our table data have so far used one separate list for each column. If there were n columns, we would need n list objects to represent the data in the table. However, we think of a table as *one* entity, not a collection of n columns. It would therefore be natural to use one argument for the whole table. This is easy to achieve using a *nested list*, where each entry in the list is a list itself. A table object, for instance, is a list of lists, either a list of the row elements of the table or a list of the column elements of the table. Here is an example where the table is a list of two columns, and each column is a list of numbers:

```
Cdegrees = range(-20, 41, 5)   # -20, -15, ..., 35, 40
Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]

table = [Cdegrees, Fdegrees]
```

(Note that any value in $[41, 45]$ can be used as second argument (stop value) to `range` and will ensure that 40 is included in the range of generate numbers.)

With the subscript `table[0]` we can access the first element in `table`, which is nothing but the `Cdegrees` list, and with `table[0][2]` we reach the third element in the first element, i.e., `Cdegrees[2]`.

However, tabular data with rows and columns usually have the convention that the underlying data is a nested list where the first index counts the rows and the second index counts the columns. To have `table` on this form, we must construct `table` as a list of `[C, F]` pairs. The first index will then run over rows `[C, F]`. Here is how we may construct the nested list:

Copyrighted material

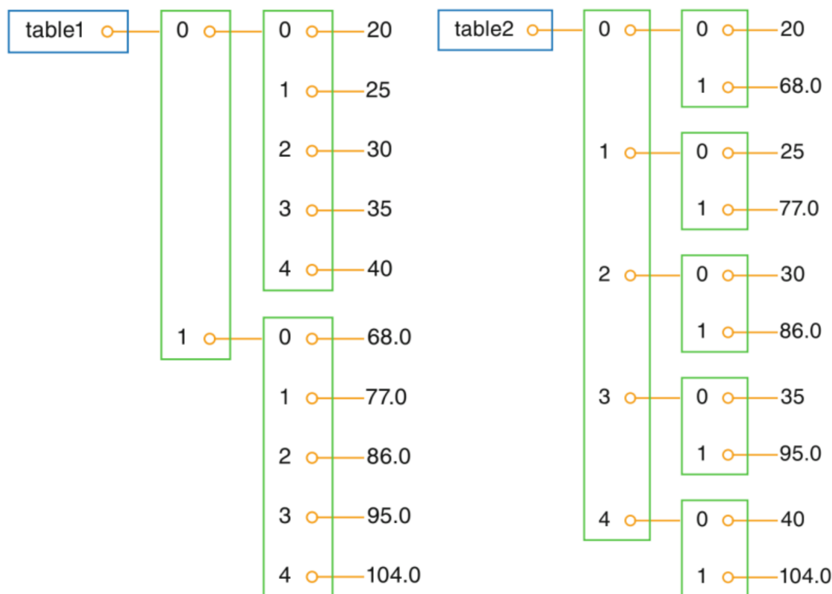

Fig. 2.2 Two ways of creating a table as a nested list. Left: table of columns `C` and `F`, where `C` and `F` are lists. Right: table of rows, where each row `[C, F]` is a list of two floats

```
table = []
for C, F in zip(Cdegrees, Fdegrees):
    table.append([C, F])
```

We may shorten this code segment by introducing a list comprehension:

```
table = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]
```

This construction loops through pairs `C` and `F`, and for each pass in the loop we create a list element `[C, F]`.

The subscript `table[1]` refers to the second element in `table`, which is a `[C, F]` pair, while `table[1][0]` is the `C` value and `table[1][1]` is the `F` value. Figure 2.2 illustrates both a list of columns and a list of pairs. Using this figure, you can realize that the first index looks up an element in the outer list, and that this element can be indexed with the second index.

2.4.2 Printing Objects

Modules for pretty print of objects We may write `print table` to immediately view the nested list `table` from the previous section. In fact, any Python object `obj` can be printed to the screen by the command `print obj`. The output is usually one line, and this line may become very long if the list has many elements. For example, a long list like our `table` variable, demands a quite long line when printed.

```
[[-20, -4.0], [-15, 5.0], [-10, 14.0], ............., [40, 104.0]]
```

Copyrighted material

We can also slice the second index, or both indices:

```
>>> table[4:7][0:2]
[[0, 32.0], [5, 41.0]]
```

Observe that `table[4:7]` makes a list `[[0, 32.0], [5, 41.0], [10, 50.0]]` with three elements. The slice `[0:2]` acts on this sublist and picks out its first two elements, with indices 0 and 1.

Sublists are always copies of the original list, so if you modify the sublist the original list remains unaltered and vice versa:

```
>>> l1 = [1, 4, 3]
>>> l2 = l1[:-1]
>>> l2
[1, 4]
>>> l1[0] = 100
>>> l1              # l1 is modified
[100, 4, 3]
>>> l2              # l2 is not modified
[1, 4]
```

The fact that slicing makes a copy can also be illustrated by the following code:

```
>>> B = A[:]
>>> C = A
>>> B == A
True
>>> B is A
False
>>> C is A
True
```

The `B == A` boolean expression is `True` if all elements in `B` are equal to the corresponding elements in `A`. The test `B is A` is `True` if `A` and `B` are names for the same list. Setting `C = A` makes `C` refer to the same list object as `A`, while `B = A[:]` makes `B` refer to a copy of the list referred to by `A`.

Example We end this information on sublists by writing out the part of the `table` list of `[C, F]` rows (see Sect. 2.4) where the Celsius degrees are between 10 and 35 (not including 35):

```
>>> for C, F in table[Cdegrees.index(10):Cdegrees.index(35)]:
...     print '%5.0f %5.1f' % (C, F)
...
   10  50.0
   15  59.0
   20  68.0
   25  77.0
   30  86.0
```

You should always stop reading and convince yourself that you understand why a code segment produces the printed output. In this latter example, `Cdegrees.`

Copyrighted material

index(10) returns the index corresponding to the value 10 in the Cdegrees list. Looking at the Cdegrees elements, one realizes (do it!) that the for loop is equivalent to

```
for C, F in table[6:11]:
```

This loop runs over the indices 6, 7, ..., 10 in table.

2.4.4 Traversing Nested Lists

We have seen that traversing the nested list table could be done by a loop of the form

```
for C, F in table:
    # process C and F
```

This is natural code when we know that table is a list of [C, F] lists. Now we shall address more general nested lists where we do not necessarily know how many elements there are in each list element of the list.

Suppose we use a nested list scores to record the scores of players in a game: scores[i] holds a list of the historical scores obtained by player number i. Different players have played the game a different number of times, so the length of scores[i] depends on i. Some code may help to make this clearer:

```
scores = []
# score of player no. 0:
scores.append([12, 16, 11, 12])
# score of player no. 1:
scores.append([9])
# score of player no. 2:
scores.append([6, 9, 11, 14, 17, 15, 14, 20])
```

The list scores has three elements, each element corresponding to a player. The element no. g in the list scores[p] corresponds to the score obtained in game number g played by player number p. The length of the lists scores[p] varies and equals 4, 1, and 8 for p equal to 0, 1, and 2, respectively.

In the general case we may have n players, and some may have played the game a large number of times, making scores potentially a big nested list. How can we traverse the scores list and write it out in a table format with nicely formatted columns? Each row in the table corresponds to a player, while columns correspond to scores. For example, the data initialized above can be written out as

```
  12  16  11  12
   9
   6   9  11  14  17  15  14  20
```

In a program, we must use two *nested loops*, one for the elements in scores and one for the elements in the sublists of scores. The example below will make this clear.

Copyrighted material

There are two basic ways of traversing a nested list: either we use integer indices for each index, or we use variables for the list elements. Let us first exemplify the index-based version:

```
for p in range(len(scores)):
    for g in range(len(scores[p])):
        score = scores[p][g]
        print '%4d' % score,
    print
```

With the trailing comma after the print string, we avoid a newline so that the column values in the table (i.e., scores for one player) appear at the same line. The single `print` command after the loop over `c` adds a newline after each table row. The reader is encouraged to go through the loops by hand and simulate what happens in each statement (use the simple `scores` list initialized above).

The alternative version where we use variables for iterating over the elements in the `scores` list and its sublists looks like this:

```
for player in scores:
    for game in player:
        print '%4d' % game,
    print
```

Again, the reader should step through the code by hand and realize what the values of `player` and `game` are in each pass of the loops.

In the very general case we can have a nested list with many indices: `somelist[i1][i2][i3]....` To visit each of the elements in the list, we use as many nested `for` loops as there are indices. With four indices, iterating over integer indices look as

```
for i1 in range(len(somelist)):
    for i2 in range(len(somelist[i1])):
        for i3 in range(len(somelist[i1][i2])):
            for i4 in range(len(somelist[i1][i2][i3])):
                value = somelist[i1][i2][i3][i4]
                # work with value
```

The corresponding version iterating over sublists becomes

```
for sublist1 in somelist:
    for sublist2 in sublist1:
        for sublist3 in sublist2:
            for sublist4 in sublist3:
                value = sublist4
                # work with value
```

Copyrighted material

2.5 Tuples

Tuples are very similar to lists, but tuples cannot be changed. That is, a tuple can be viewed as a *constant list*. While lists employ square brackets, tuples are written with standard parentheses:

```
>>> t = (2, 4, 6, 'temp.pdf')    # define a tuple with name t
```

One can also drop the parentheses in many occasions:

```
>>> t = 2, 4, 6, 'temp.pdf'
>>> for element in 'myfile.txt', 'yourfile.txt', 'herfile.txt':
...     print element,
...
myfile.txt yourfile.txt herfile.txt
```

The `for` loop here is over a tuple, because a comma separated sequence of objects, even without enclosing parentheses, becomes a tuple. Note the trailing comma in the `print` statement. This comma suppresses the final newline that the `print` command automatically adds to the output string. This is the way to make several `print` statements build up one line of output.

Much functionality for lists is also available for tuples, for example:

```
>>> t = t + (-1.0, -2.0)           # add two tuples
>>> t
(2, 4, 6, 'temp.pdf', -1.0, -2.0)
>>> t[1]                           # indexing
4
>>> t[2:]                          # subtuple/slice
(6, 'temp.pdf', -1.0, -2.0)
>>> 6 in t                         # membership
True
```

Any list operation that changes the list will not work for tuples:

```
>>> t[1] = -1
...
TypeError: object does not support item assignment

>>> t.append(0)
...
AttributeError: 'tuple' object has no attribute 'append'

>>> del t[1]
...
TypeError: object doesn't support item deletion
```

Some list methods, like `index`, are not available for tuples. So why do we need tuples when lists can do more than tuples?

- Tuples protect against accidental changes of their contents.
- Code based on tuples is faster than code based on lists.

Copyrighted material

- Tuples are frequently used in Python software that you certainly will make use of, so you need to know this data type.

There is also a fourth argument, which is important for a data type called dictionaries (introduced in Sect. 6.1): tuples can be used as keys in dictionaries while lists can not.

2.6 Summary

2.6.1 Chapter Topics

While loops Loops are used to repeat a collection of program statements several times. The statements that belong to the loop must be consistently indented in Python. A `while` loop runs as long as a condition evaluates to `True`:

```
>>> t = 0; dt = 0.5; T = 2
>>> while t <= T:
...     print t
...     t += dt
...
0
0.5
1.0
1.5
2.0
>>> print 'Final t:', t, '; t <= T is', t <= T
Final t: 2.5 ; t <= T is False
```

Lists A list is used to collect a number of values or variables in an ordered sequence.

```
>>> mylist = [t, dt, T, 'mynumbers.dat', 100]
```

A list element can be any Python object, including numbers, strings, functions, and other lists, for instance.

The table below shows some important list operations (only a subset of these are explained in the present chapter).

Construction	Meaning
`a = []`	initialize an empty list
`a = [1, 4.4, 'run.py']`	initialize a list
`a.append(elem)`	add `elem` object to the end
`a + [1,3]`	add two lists
`a.insert(i, e)`	insert element `e` before index `i`
`a[3]`	index a list element
`a[-1]`	get last list element
`a[1:3]`	slice: copy data to sublist (here: index 1, 2)
`del a[3]`	delete an element (index 3)

Copyrighted material

2.6.2 Example: Analyzing List Data

Problem The file `src/misc/Oxford_sun_hours.txt`[2] contains data of the number of sun hours in Oxford, UK, for every month since January 1929. The data are already on a suitable nested list format:

```
[
[43.8, 60.5, 190.2, ...],
[49.9, 54.3, 109.7, ...],
[63.7, 72.0, 142.3, ...],
...
]
```

The list in every line holds the number of sun hours for each of the year's 12 months. That is, the first index in the nested list corresponds to year and the second index corresponds to the month number. More precisely, the double index `[i][j]` corresponds to year $1929 + \mathtt{i}$ and month $1 + \mathtt{j}$ (January being month number 1).

The task is to define this nested list in a program and do the following data analysis.

- Compute the average number of sun hours for each month during the total data period (1929–2009).
- Which month has the best weather according to the means found in the preceding task?
- For each decade, 1930–1939, 1940–1949, ..., 2000–2009, compute the average number of sun hours per day in January and December. For example, use December 1949, January 1950, ..., December 1958, and January 1959 as data for the decade 1950–1959. Are there any noticeable differences between the decades?

Solution Initializing the data is easy: just copy the data from the `Oxford_sun_hours.txt` file into the program file and set a variable name on the left hand side (the long and wide code is only indicated here):

```
data = [
[43.8, 60.5, 190.2, ...],
[49.9, 54.3, 109.7, ...],
[63.7, 72.0, 142.3, ...],
...
]
```

For task 1, we need to establish a list `monthly_mean` with the results from the computation, i.e., `monthly_mean[2]` holds the average number of sun hours for March in the period 1929–2009. The average is computed in the standard way: for each month, we run through all the years, sum up the values, and finally divide by the number of years (`len(data)` or $2009 - 1929 + 1$).

[2] http://tinyurl.com/pwyasaa/misc/Oxford_sun_hours.txt

Copyrighted material

```
monthly_mean = []
n = len(data)    # no of years
for m in range(12): # counter for month indices
    s = 0           # sum
    for y in data:  # loop over "rows" (first index) in data
        s += y[m]   # add value for month m
    monthly_mean.append(s/n)
```

An alternative solution would be to introduce separate variables for the monthly averages, say `Jan_mean`, `Feb_mean`, etc. The reader should as an exercise write the code associated with such a solution and realize that using the `monthly_mean` list is more elegant and yields much simpler and shorter code. Separate variables might be an okay solution for 2–3 variables, but not for as many as 12.

Perhaps we want a nice-looking printout of the results. This can elegantly be created by first defining a tuple (or list) of the names of the months and then running through this list in parallel with `monthly_mean`:

```
month_names = 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',\
              'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'
for name, value in zip(month_names, monthly_mean):
    print '%s: %.1f' % (name, value)
```

The printout becomes

```
Jan: 56.6
Feb: 72.7
Mar: 116.5
Apr: 153.2
May: 191.1
Jun: 198.5
Jul: 193.8
Aug: 184.3
Sep: 138.3
Oct: 104.6
Nov: 67.4
Dec: 52.4
```

Task 2 can be solved by pure inspection of the above printout, which reveals that June is the winner. However, since we are learning programming, we should be able to replace our eyes with some computer code to automate the task. The maximum value `max_value` of a list like `monthly_mean` is simply obtained by `max(monthly_mean)`. The corresponding index, needed to find the right name of the corresponding month, is found from `monthly_mean.index(max_value)`. The code for task 2 is then

```
max_value = max(monthly_mean)
month = month_names[monthly_mean.index(max_value)]
print '%s has best weather with %.1f sun hours on average' % \
      (month, max_value)
```

Task 3 requires us to first develop an algorithm for how to compute the decade averages. The algorithm, expressed with words, goes as follows. We loop over the

Copyrighted material

decades, and for each decade, we loop over its years, and for each year, we add the December data of the previous year and the January data of the current year to an accumulation variable. Dividing this accumulation variable by $10 \cdot 2 \cdot 30$ gives the average number of sun hours per day in the winter time for the particular decade. The code segment below expresses this algorithm in the Python language:

```
decade_mean = []
for decade_start in range(1930, 2010, 10):
    Jan_index = 0; Dec_index = 11  # indices
    s = 0
    for year in range(decade_start, decade_start+10):
        y = year - 1929  # list index
        print data[y-1][Dec_index] + data[y][Jan_index]
        s += data[y-1][Dec_index] + data[y][Jan_index]
    decade_mean.append(s/(20.*30))
for i in range(len(decade_mean)):
    print 'Decade %d-%d: %.1f' % \
          (1930+i*10, 1939+i*10, decade_mean[i])
```

The output becomes

```
Decade 1930-1939: 1.7
Decade 1940-1949: 1.8
Decade 1950-1959: 1.8
Decade 1960-1969: 1.8
Decade 1970-1979: 1.6
Decade 1980-1989: 2.0
Decade 1990-1999: 1.8
Decade 2000-2009: 2.1
```

The complete code is found in the file `sun_data.py`.

Remark The file `Oxford_sun_hours.txt` is based on data from the UK Met Office[3]. A Python program for downloading the data, interpreting the content, and creating a file like `Oxford_sun_hours.txt` is explained in detail in Sect. 6.3.3.

2.6.3 How to Find More Python Information

This book contains only fragments of the Python language. When doing your own projects or exercises you will certainly feel the need for looking up more detailed information on modules, objects, etc. Fortunately, there is a lot of excellent documentation on the Python programming language.

The primary reference is the official Python documentation website[4]: `docs.python.org`. Here you can find a Python tutorial, the very useful *Library Reference* [3], and a *Language Reference*, to mention some key documents. When you wonder what functions you can find in a module, say the `math` module, you can go to the Library Reference search for *math*, which quickly leads you to the official

[3] http://www.metoffice.gov.uk/climate/uk/stationdata/
[4] http://docs.python.org/index.html

Copyrighted material

documentation of the `math` module. Alternatively, you can go to the index of this document and pick the `math (module)` item directly. Similarly, if you want to look up more details of the printf formatting syntax, go to the index and follow the *printf-style formatting* index.

Warning

A word of caution is probably necessary here. Reference manuals are very technical and written primarily for experts, so it can be quite difficult for a newbie to understand the information. An important ability is to browse such manuals and dig out the key information you are looking for, without being annoyed by all the text you do not understand. As with programming, reading manuals efficiently requires a lot of training.

A tool somewhat similar to the Python Standard Library documentation is the `pydoc` program. In a terminal window you write

Terminal
```
Terminal> pydoc math
```

In IPython there are two corresponding possibilities, either

```
In [1]: !pydoc math
```

or

```
In [2]: import math
In [3]: help(math)
```

The documentation of the complete `math` module is shown as plain text. If a specific function is wanted, we can ask for that directly, e.g., `pydoc math.tan`. Since `pydoc` is very fast, many prefer `pydoc` over web pages, but `pydoc` has often less information compared to the web documentation of modules.

There are also a large number of books about Python. Beazley [1] is an excellent reference that improves and extends the information in the web documentation. The *Learning Python* book [17] has been very popular for many years as an introduction to the language. There is a special web page[5] listing most Python books on the market. Very few books target scientific computing with Python, but [4] gives an introduction to Python for mathematical applications and is more compact and advanced than the present book. It also serves as an excellent reference for the capabilities of Python in a scientific context. A comprehensive book on the use of Python for assisting and automating scientific work is [13].

Quick references, which list almost to all Python functionality in compact tabular form, are very handy. We recommend in particular the one by Richard Gruet[6] [6].

The website http://www.python.org/doc/ contains a list of useful Python introductions and reference manuals.

[5] http://wiki.python.org/moin/PythonBooks
[6] http://rgruet.free.fr/PQR27/PQR2.7.html

Copyrighted material

2.7 Exercises

Exercise 2.1: Make a Fahrenheit-Celsius conversion table
Write a Python program that prints out a table with Fahrenheit degrees 0, 10, 20, ..., 100 in the first column and the corresponding Celsius degrees in the second column.

Hint Modify the `c2f_table_while.py` program from Sect. 2.1.2.
Filename: `f2c_table_while`.

Exercise 2.2: Generate an approximate Fahrenheit-Celsius conversion table
Many people use an approximate formula for quickly converting Fahrenheit (F) to Celsius (C) degrees:

$$C \approx \hat{C} = (F - 30)/2 \tag{2.2}$$

Modify the program from Exercise 2.1 so that it prints three columns: F, C, and the approximate value $\hat{C}$.
Filename: `f2c_approx_table`.

Exercise 2.3: Work with a list
Set a variable `primes` to a list containing the numbers 2, 3, 5, 7, 11, and 13. Write out each list element in a `for` loop. Assign 17 to a variable `p` and add `p` to the end of the list. Print out the entire new list.
Filename: `primes`.

Exercise 2.4: Generate odd numbers
Write a program that generates all odd numbers from 1 to `n`. Set `n` in the beginning of the program and use a `while` loop to compute the numbers. (Make sure that if `n` is an even number, the largest generated odd number is `n-1`.)
Filename: `odd`.

Exercise 2.5: Compute the sum of the first *n* integers
Write a program that computes the sum of the integers from 1 up to and including n. Compare the result with the famous formula $n(n+1)/2$.
Filename: `sum_int`.

Exercise 2.6: Compute energy levels in an atom
The n-th energy level for an electron in a Hydrogen atom is given by

$$E_n = -\frac{m_e e^4}{8\epsilon_0^2 h^2} \cdot \frac{1}{n^2},$$

where $m_e = 9.1094 \cdot 10^{-31}$ kg is the electron mass, $e = 1.6022 \cdot 10^{-19}$ C is the elementary charge, $\epsilon_0 = 8.8542 \cdot 10^{-12}\,\mathrm{C^2\,s^2\,kg^{-1}\,m^-3}$ is the electrical permittivity of vacuum, and $h = 6.6261 \cdot 10^{-34}$ Js.

a) Write a Python program that calculates and prints the energy level E_n for $n = 1, \ldots, 20$.

Copyrighted material

Exercise 2.13: Simulate a program by hand
Consider the following program for computing with interest rates:

```
initial_amount = 100
p = 5.5  # interest rate
amount = initial_amount
years = 0
while amount <= 1.5*initial_amount:
    amount = amount + p/100*amount
    years = years + 1
print years
```

a) Use a pocket calculator or an interactive Python shell and work through the program calculations by hand. Write down the value of `amount` and `years` in each pass of the loop.
b) Set `p = 5` instead. Why will the loop now run forever? (Apply Ctrl+c, see Exercise 2.11, to stop a program with a loop that runs forever.) Make the program robust against such errors.
c) Make use of the operator += wherever possible in the program.
d) Explain with words what type of mathematical problem that is solved by this program.

Filename: `interest_rate_loop`.

Exercise 2.14: Explore Python documentation
Suppose you want to compute with the inverse sine function: $\sin^{-1} x$. How do you do that in a Python program?

Hint The `math` module has an inverse sine function. Find the correct name of the function by looking up the module content in the online Python Standard Library[7] document or use `pydoc`, see Sect. 2.6.3.
Filename: `inverse_sine`.

Exercise 2.15: Index a nested list
We define the following nested list:

```
q = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h']]
```

a) Index this list to extract 1) the letter `a`; 2) the list `['d', 'e', 'f']`; 3) the last element `h`; 4) the `d` element. Explain why `q[-1][-2]` has the value `g`.
b) We can visit all elements of `q` using this nested `for` loop:

```
for i in q:
    for j in range(len(i)):
        print i[j]
```

What type of objects are `i` and `j`?

Filename: `index_nested_list`.

[7] http://docs.python.org/2/library/

Copyrighted material

Exercise 2.16: Store data in lists
Modify the program from Exercise 2.2 so that all the F, C, and $\hat{C}$ values are stored in separate lists `F`, `C`, and `C_approx`, respectively. Then make a nested list `conversion` so that `conversion[i]` holds a row in the table: `[F[i], C[i], C_approx[i]]`. Finally, let the program traverse the `conversion` list and write out the same table as in Exercise 2.2.
Filename: `f2c_approx_lists`.

Exercise 2.17: Store data in a nested list
The purpose of this exercise is to store tabular data in two alternative ways, either as a list of columns or as a list of rows. In order to write out a nicely formatted table, one has to traverse the data, and the technique for traversal depends on how the tabular data is stored.

a) Compute two lists of t and y values as explained in Exercise 2.9. Store the two lists in a new *nested* list `ty1` such that `ty1[0]` and `ty1[1]` correspond to the two lists. Write out a table with t and y values in two columns by looping over the data in the `ty1` list. Each number should be written with two decimals.
b) Make a list `ty2` which holds each row in the table of t and y values (`ty1` is a list of table columns while `ty2` is a list of table rows, as explained in Sect. 2.4). Loop over the `ty2` list and write out the t and y values with two decimals each.

Filename: `ball_table3`.

Exercise 2.18: Values of boolean expressions
Explain the outcome of each of the following boolean expressions:

```
C = 41
C == 40
C != 40 and C < 41
C != 40 or  C < 41
not C == 40
not C > 40
C <= 41
not False
True and False
False or True
False or False or False
True and True and False
False == 0
True == 0
True == 1
```

Note
It makes sense to compare `True` and `False` to the integers 0 and 1, but not other integers (e.g., `True == 12` is `False` although the *integer* 12 evaluates to `True` in a boolean context, as in `bool(12)` or `if 12`).

Filename: `eval_bool`.

Copyrighted material

Exercise 2.19: Explore round-off errors from a large number of inverse operations
Maybe you have tried to hit the square root key on a calculator multiple times and then squared the number again an equal number of times. These set of inverse mathematical operations should of course bring you back to the starting value for the computations, but this does not always happen. To avoid tedious pressing of calculator keys, we can let a computer automate the process. Here is an appropriate program:

```
from math import sqrt
for n in range(1, 60):
    r = 2.0
    for i in range(n):
        r = sqrt(r)
    for i in range(n):
        r = r**2
    print '%d times sqrt and **2: %.16f' % (n, r)
```

Explain with words what the program does. Then run the program. Round-off errors are here completely destroying the calculations when `n` is large enough! Investigate the case when we come back to 1 instead of 2 by fixing an `n` value where this happens and printing out `r` in both `for` loops over `i`. Can you now explain why we come back to 1 and not 2?
Filename: `repeated_sqrt`.

Exercise 2.20: Explore what zero can be on a computer
Type in the following code and run it:

```
eps = 1.0
while 1.0 != 1.0 + eps:
    print '...............', eps
    eps = eps/2.0
print 'final eps:', eps
```

Explain with words what the code is doing, line by line. Then examine the output. How can it be that the "equation" $1 \neq 1 + \text{eps}$ is not true? Or in other words, that a number of approximately size 10^{-16} (the final `eps` value when the loop terminates) gives the same result as if `eps` were zero?
Filename: `machine_zero`.

Remarks The nonzero `eps` value computed above is called *machine epsilon* or *machine zero* and is an important parameter to know, especially when certain mathematical techniques are applied to control round-off errors.

Exercise 2.21: Compare two real numbers with a tolerance
Run the following program:

```
a = 1/947.0*947
b = 1
if a != b:
    print 'Wrong result!'
```

Copyrighted material

The lesson learned from this program is that one should never compare two floating-point objects directly using `a == b` or `a != b`, because round-off errors quickly make two identical mathematical values different on a computer. A better result is to test if `abs(a - b) < tol`, where `tol` is a very small number. Modify the test according to this idea.
Filename: `compare_floats`.

Exercise 2.22: Interpret a code
The function `time` in the module `time` returns the number of seconds since a particular date (called the Epoch, which is January 1, 1970, on many types of computers). Python programs can therefore use `time.time()` to mimic a stop watch. Another function, `time.sleep(n)` causes the program to pause for `n` seconds and is handy for inserting a pause. Use this information to explain what the following code does:

```
import time
t0 = time.time()
while time.time() - t0 < 10:
    print '....I like while loops!'
    time.sleep(2)
print 'Oh, no - the loop is over.'
```

How many times is the `print` statement inside the loop executed? Now, copy the code segment and change the < sign in the loop condition to a > sign. Explain what happens now.
Filename: `time_while`.

Exercise 2.23: Explore problems with inaccurate indentation
Type in the following program in a file and check carefully that you have exactly the same spaces:

```
C = -60; dC = 2
while C <= 60:
    F = (9.0/5)*C + 32
      print C, F
C = C + dC
```

Run the program. What is the first problem? Correct that error. What is the next problem? What is the cause of that problem? (See Exercise 2.11 for how to stop a hanging program.)
Filename: `indentation`.

Remarks The lesson learned from this exercise is that one has to be very careful with indentation in Python programs! Other computer languages usually enclose blocks belonging to loops in curly braces, parentheses, or begin-end marks. Python's convention with using solely indentation contributes to visually attractive, easy-to-read code, at the cost of requiring a pedantic attitude to blanks from the programmer.

Copyrighted material

Exercise 2.24: Explore punctuation in Python programs
Some of the following assignments work and some do not. Explain in each case why the assignment works/fails and, if it works, what kind of object `x` refers to and what the value is if we do a `print x`.

```
x = 1
x = 1.
x = 1;
x = 1!
x = 1?
x = 1:
x = 1,
```

Hint Explore the statements in an interactive Python shell.
Filename: `punctuation`.

Exercise 2.25: Investigate a for loop over a changing list
Study the following interactive session and explain in detail what happens in each pass of the loop, and use this explanation to understand the output.

```
>>> numbers = range(10)
>>> print numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> for n in numbers:
...     i = len(numbers)/2
...     del numbers[i]
...     print 'n=%d, del %d' % (n,i), numbers
...
n=0, del 5 [0, 1, 2, 3, 4, 6, 7, 8, 9]
n=1, del 4 [0, 1, 2, 3, 6, 7, 8, 9]
n=2, del 4 [0, 1, 2, 3, 7, 8, 9]
n=3, del 3 [0, 1, 2, 7, 8, 9]
n=8, del 3 [0, 1, 2, 8, 9]
```

Warning

The message in this exercise is to *never modify a list that we are looping over.* Modification is indeed technically possible, as shown above, but you really need to know what you are doing. Otherwise you will experience very strange program behavior.

Filename: `for_changing_list`.

Copyrighted material

3.1.2 Understanding the Program Flow

A programmer must have a deep understanding of the sequence of statements that are executed in the program and be able to simulate by hand what happens with a program in the computer. To help build this understanding, a debugger (see Sect. F.1) or the Online Python Tutor[2] are excellent tools. A debugger can be used for all sorts of programs, large and small, while the Online Python Tutor is primarily an educational tool for small programs. We shall demonstrate it here.

Below is a program `c2f.py` having a function and a `for` loop, with the purpose of printing out a table for conversion of Celsius to Fahrenheit degrees:

```
def F(C):
    F = 9./5*C + 32
    return F

dC = 10
C = -30
while C <= 50:
    print '%5.1f %5.1f' % (C, F(C))
    C += dC
```

We shall now ask the Online Python Tutor to visually explain how the program is executed. Go to http://www.pythontutor.com/visualize.html, erase the code there and write or paste the `c2f.py` file into the editor area. Click *Visualize Execution*. Press the forward button to advance one statement at a time and observe the evolution of variables to the right in the window. This demo illustrates how the program jumps around in the loop and up to the `F(C)` function and back again. Figure 3.1

Fig. 3.1 Screen shot of the Online Python Tutor and stepwise execution of the `c2f.py` program

[2] http://www.pythontutor.com/

Copyrighted material

gives a snapshot of the status of variables, terminal output, and what the current and next statements are.

Tip: How does a program actually work?
Every time you are a bit uncertain about the flow of statements in a program with loops and/or functions, go to http://www.pythontutor.com/visualize.html, paste in your program and see exactly what happens.

3.1.3 Local and Global Variables

Local variables are invisible outside functions Let us reconsider the F2(C) function from Sect. 3.1.1. The variable F_value is a *local* variable in the function, and a local variable does not exist outside the function, i.e., in the main program. We can easily demonstrate this fact by continuing the previous interactive session:

```
>>> c1 = 37.5
>>> s2 = F2(c1)
>>> F_value
...
NameError: name 'F_value' is not defined
```

This error message demonstrates that the surrounding program outside the function is not aware of F_value. Also the argument to the function, C, is a local variable that we cannot access outside the function:

```
>>> C
...
NameError: name 'C' is not defined
```

On the contrary, the variables defined outside of the function, like s1, s2, and c1 in the above session, are *global* variables. These can be accessed everywhere in a program, also inside the F2 function.

Local variables hide global variables Local variables are created inside a function and destroyed when we leave the function. To learn more about this fact, we may study the following session where we write out F_value, C, and some global variable r inside the function:

```
>>> def F3(C):
...     F_value = (9.0/5)*C + 32
...     print 'Inside F3: C=%s F_value=%s r=%s' % (C, F_value, r)
...     return '%.1f degrees Celsius corresponds to '\
...            '%.1f degrees Fahrenheit' % (C, F_value)
...
>>> C = 60    # make a global variable C
>>> r = 21    # another global variable
>>> s3 = F3(r)
```

Copyrighted material

```
Inside F3: C=21 F_value=69.8 r=21
>>> s3
'21.0 degrees Celsius corresponds to 69.8 degrees Fahrenheit'
>>> C
60
```

This example illustrates that there are two `C` variables, one global, defined in the main program with the value 60 (an `int` object), and one local, living when the program flow is inside the `F3` function. The value of this latter `C` is given in the call to the `F3` function (an `int` object). Inside the `F3` function the local `C` *hides* the global `C` variable in the sense that when we refer to `C` we access the local variable. (The global `C` can technically be accessed as `globals()['C']`, but one should avoid working with local and global variables with the same names at the same time!)

The Online Python Tutor gives a complete overview of what the local and global variables are at any point of time. For instance, in the example from Sect. 3.1.2, Fig. 3.1 shows the content of the three global variables `F`, `dC`, and `C`, along with the content of the variables that are in play in this call of the `F(C)` function: `C` and `F`.

How Python looks up variables

The more general rule, when you have several variables with the same name, is that Python first tries to look up the variable name among the local variables, then there is a search among global variables, and finally among built-in Python functions.

Example Here is a complete sample program that aims to illustrate the rule above:

```
print sum  # sum is a built-in Python function
sum = 500  # rebind the name sum to an int
print sum  # sum is a global variable

def myfunc(n):
    sum = n + 1
    print sum  # sum is a local variable
    return sum

sum = myfunc(2) + 1   # new value in global variable sum
print sum
```

In the first line, there are no local variables, so Python searches for a global value with name `sum`, but cannot find any, so the search proceeds with the built-in functions, and among them Python finds a function with name `sum`. The printout of `sum` becomes something like `<built-in function sum>`.

The second line rebinds the global name `sum` to an `int` object. When trying to access `sum` in the next `print` statement, Python searches among the global variables (no local variables so far) and finds one. The printout becomes `500`. The call `myfunc(2)` invokes a function where `sum` is a local variable. Doing a `print sum` in this function makes Python first search among the local variables, and since `sum`

Copyrighted material

is found there, the printout becomes 3 (and not 500, the value of the global variable sum). The value of the local variable sum is returned, added to 1, to form an int object with value 4. This int object is then bound to the global variable sum. The final print sum leads to a search among global variables, and we find one with value 4.

Changing global variables inside functions The values of global variables can be accessed inside functions, but the values cannot be changed unless the variable is declared as global:

```
a = 20; b = -2.5          # global variables

def f1(x):
    a = 21                # this is a new local variable
    return a*x + b

print a                   # yields 20

def f2(x):
    global a
    a = 21                # the global a is changed
    return a*x + b

f1(3); print a            # 20 is printed
f2(3); print a            # 21 is printed
```

Note that in the f1 function, a = 21 creates a local variable a. As a programmer you may think you change the global a, but it does not happen! *You are strongly encouraged to run the programs in this section in the Online Python Tutor*, which is an excellent tool to explore local versus global variables and thereby get a good understanding of these concepts.

3.1.4 Multiple Arguments

The previous F(C) and F2(C) functions from Sect. 3.1.1 are functions of one variable, C, or as we phrase it in computer science: the functions take one argument (C). Functions can have as many arguments as desired; just separate the argument names by commas.

Consider the mathematical function

$$y(t) = v_0 t - \frac{1}{2} g t^2,$$

where g is a fixed constant and v_0 is a physical parameter that can vary. Mathematically, y is a function of one variable, t, but the function values also depends on the value of v_0. That is, to evaluate y, we need values for t *and* v_0. A natural Python implementation is therefore a function with two arguments:

Copyrighted material

```
def yfunc(t, v0):
    g = 9.81
    return v0*t - 0.5*g*t**2
```

Note that the arguments `t` and `v0` are local variables in this function. Examples on valid calls are

```
y = yfunc(0.1, 6)
y = yfunc(0.1, v0=6)
y = yfunc(t=0.1, v0=6)
y = yfunc(v0=6, t=0.1)
```

The possibility to write `argument=value` in the call makes it easier to read and understand the call statement. With the `argument=value` syntax for all arguments, the sequence of the arguments does not matter in the call, which here means that we may put `v0` before `t`. When omitting the `argument=` part, the sequence of arguments in the call must perfectly match the sequence of arguments in the function definition. The `argument=value` arguments must appear after all the arguments where only `value` is provided (e.g., `yfunc(t=0.1, 6)` is illegal).

Whether we write `yfunc(0.1, 6)` or `yfunc(v0=6, t=0.1)`, the arguments are initialized as local variables in the function in the same way as when we assign values to variables:

```
    t = 0.1
    v0 = 6
```

These statements are not visible in the code, but a call to a function automatically initializes the arguments in this way.

3.1.5 Function Argument or Global Variable?

Since y mathematically is considered a function of one variable, t, some may argue that the Python version of the function, `yfunc`, should be a function of `t` only. This is easy to reflect in Python:

```
def yfunc(t):
    g = 9.81
    return v0*t - 0.5*g*t**2
```

The main difference is that `v0` now becomes a *global* variable, which needs to be initialized outside the function `yfunc` (in the main program) before we attempt to call `yfunc`. The next session demonstrates what happens if we fail to initialize such a global variable:

Copyrighted material

```
>>> def f(x):
...     return x, x**2, x**4
...
>>> s = f(2)
>>> s
(2, 4, 16)
>>> type(s)
<type 'tuple'>
>>> x, x2, x4 = f(2)    # store in separate variables
```

Note that storing multiple return values in separate variables, as we do in the last line, is actually the same functionality as we use for storing list (or tuple) elements in separate variables, see Sect. 2.2.1.

3.1.8 Computing Sums

Our next example concerns a Python function for calculating the sum

$$L(x;n) = \sum_{i=1}^{n} \frac{1}{i} \left(\frac{x}{1+x} \right)^{i} . \quad (3.1)$$

To compute a sum in a program, we use a loop and add terms to an accumulation variable inside the loop. Section 2.1.4 explains the idea. However, summation expressions with an integer counter, such as i in (3.1), are normally implemented by a `for` loop over the i counter and not a `while` loop as in Sect. 2.1.4. For example, the implementation of $\sum_{i=1}^{n} i^2$ is typically implemented as

```
s = 0
for i in range(1, n+1):
    s += i**2
```

For the specific sum (3.1) we just replace `i**2` by the right term inside the `for` loop:

```
s = 0
for i in range(1, n+1):
    s += (1.0/i)*(x/(1.0+x))**i
```

Observe the factors `1.0` used to avoid integer division, since `i` is `int` and `x` may also be `int`.

It is natural to embed the computation of the sum in a function that takes x and n as arguments and returns the sum:

```
def L(x, n):
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1.0+x))**i
    return s
```

Copyrighted material

Our formula (3.1) is not chosen at random. In fact, it can be shown that $L(x; n)$ is an approximation to $\ln(1 + x)$ for a finite n and $x \geq 1$. The approximation becomes exact in the limit

$$\lim_{n\to\infty} L(x; n) = \ln(1 + x) .$$

Computational significance of $L(x; n)$

Although we can compute $\ln(1 + x)$ on a calculator or by `math.log(1+x)` in Python, you may have wondered how such a function is actually calculated inside the calculator or the `math` module. In most cases this must be done via simple mathematical expressions such as the sum in (3.1). A calculator and the `math` module will use more sophisticated formulas than (3.1) for ultimate efficiency of the calculations, but the main point is that the numerical values of mathematical functions like $\ln(x)$, $\sin(x)$, and $\tan(x)$ are usually computed by sums similar to (3.1).

Instead of having our `L` function just returning the value of the sum, we could return additional information on the error involved in the approximation of $\ln(1+x)$ by $L(x; n)$. The size of the terms decreases with increasing n, and the first neglected term is then bigger than all the remaining terms, but not necessarily bigger than their sum. The first neglected term is hence an indication of the size of the total error we make, so we may use this term as a rough estimate of the error. For comparison, we could also return the exact error since we are able to calculate the ln function by `math.log`.

A new version of the `L(x, n)` function, where we return the value of $L(x; n)$, the first neglected term, and the exact error goes as follows:

```
def L2(x, n):
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1.0+x))**i
    value_of_sum = s
    first_neglected_term = (1.0/(n+1))*(x/(1.0+x))**(n+1)
    from math import log
    exact_error = log(1+x) - value_of_sum
    return value_of_sum, first_neglected_term, exact_error

# typical call:
value, approximate_error, exact_error = L2(x, 100)
```

The next section demonstrates the usage of the `L2` function to judge the quality of the approximation $L(x; n)$ to $\ln(1 + x)$.

3.1.9 Functions with No Return Values

Sometimes a function just performs a set of statements, without computing objects that are natural to return to the calling code. In such situations one can simply skip the `return` statement. Some programming languages use the terms *procedure* or *subroutine* for functions that do not return anything.

Copyrighted material

Let us exemplify a function without return values by making a table of the accuracy of the $L(x;n)$ approximation to $\ln(1 + x)$ from the previous section:

```
def table(x):
    print '\nx=%g, ln(1+x)=%g' % (x, log(1+x))
    for n in [1, 2, 10, 100, 500]:
        value, next, error = L2(x, n)
        print 'n=%-4d %-10g  (next term: %8.2e  '\
              'error: %8.2e)' % (n, value, next, error)
```

This function just performs a set of statements that we may want to run several times. Calling

```
table(10)
table(1000)
```

gives the output

```
x=10, ln(1+x)=2.3979
n=1    0.909091    (next term: 4.13e-01  error: 1.49e+00)
n=2    1.32231     (next term: 2.50e-01  error: 1.08e+00)
n=10   2.17907     (next term: 3.19e-02  error: 2.19e-01)
n=100  2.39789     (next term: 6.53e-07  error: 6.59e-06)
n=500  2.3979      (next term: 3.65e-24  error: 6.22e-15)

x=1000, ln(1+x)=6.90875
n=1    0.999001    (next term: 4.99e-01  error: 5.91e+00)
n=2    1.498       (next term: 3.32e-01  error: 5.41e+00)
n=10   2.919       (next term: 8.99e-02  error: 3.99e+00)
n=100  5.08989     (next term: 8.95e-03  error: 1.82e+00)
n=500  6.34928     (next term: 1.21e-03  error: 5.59e-01)
```

From this output we see that the sum converges much more slowly when x is large than when x is small. We also see that the error is an order of magnitude or more larger than the first neglected term in the sum. The functions `L`, `L2`, and `table` are found in the file `lnsum.py`.

When there is no explicit `return` statement in a function, Python actually inserts an invisible `return None` statement. `None` is a special object in Python that represents something we might think of as empty data or just "nothing". Other computer languages, such as C, C++, and Java, use the word *void* for a similar thing. Normally, one will call the `table` function without assigning the return value to any variable, but if we assign the return value to a variable, `result = table(500)`, `result` will refer to a `None` object.

The `None` value is often used for variables that should exist in a program, but where it is natural to think of the value as conceptually undefined. The standard way to test if an object `obj` is set to `None` or not reads

```
if obj is None:
    ...
if obj is not None:
    ...
```

Copyrighted material

One can also use `obj == None`. The `is` operator tests if two names refer to the same object, while == tests if the contents of two objects are the same:

```
>>> a = 1
>>> b = a
>>> a is b   # a and b refer to the same object
True
>>> c = 1.0
>>> a is c
False
>>> a == c   # a and c are mathematically equal
True
```

3.1.10 Keyword Arguments

Some function arguments can be given a default value so that we may leave out these arguments in the call. A typical function may look as

```
>>> def somefunc(arg1, arg2, kwarg1=True, kwarg2=0):
>>>     print arg1, arg2, kwarg1, kwarg2
```

The first two arguments, `arg1` and `arg2`, are *ordinary* or *positional* arguments, while the latter two are *keyword arguments* or *named arguments*. Each keyword argument has a name (in this example `kwarg1` and `kwarg2`) and an associated default value. The keyword arguments must always be listed after the positional arguments in the function definition.

When calling `somefunc`, we may leave out some or all of the keyword arguments. Keyword arguments that do not appear in the call get their values from the specified default values. We can demonstrate the effect through some calls:

```
>>> somefunc('Hello', [1,2])
Hello [1, 2] True 0
>>> somefunc('Hello', [1,2], kwarg1='Hi')
Hello [1, 2] Hi 0
>>> somefunc('Hello', [1,2], kwarg2='Hi')
Hello [1, 2] True Hi
>>> somefunc('Hello', [1,2], kwarg2='Hi', kwarg1=6)
Hello [1, 2] 6 Hi
```

The sequence of the keyword arguments does not matter in the call. We may also mix the positional and keyword arguments if we explicitly write `name=value` for all arguments in the call:

```
>>> somefunc(kwarg2='Hello', arg1='Hi', kwarg1=6, arg2=[1,2],)
Hi [1, 2] 6 Hello
```

Example: Function with default parameters Consider a function of t which also contains some parameters, here A, a, and ω:

$$f(t; A, a, \omega) = Ae^{-at}\sin(\omega t)\,. \tag{3.2}$$

Copyrighted material

We can implement f as a Python function where the independent variable t is an ordinary positional argument, and the parameters A, a, and ω are keyword arguments with suitable default values:

```
from math import pi, exp, sin

def f(t, A=1, a=1, omega=2*pi):
    return A*exp(-a*t)*sin(omega*t)
```

Calling `f` with just the `t` argument specified is possible:

```
v1 = f(0.2)
```

In this case we evaluate the expression $e^{-0.2}\sin(2\pi \cdot 0.2)$. Other possible calls include

```
v2 = f(0.2, omega=1)
v3 = f(1, A=5, omega=pi, a=pi**2)
v4 = f(A=5, a=2, t=0.01, omega=0.1)
v5 = f(0.2, 0.5, 1, 1)
```

You should write down the mathematical expressions that arise from these four calls. Also observe in the third line above that a positional argument, `t` in that case, can appear in between the keyword arguments if we write the positional argument on the keyword argument form `name=value`. In the last line we demonstrate that keyword arguments can be used as positional argument, i.e., the name part can be skipped, but then the sequence of the keyword arguments in the call must match the sequence in the function definition exactly.

Example: Computing a sum with default tolerance Consider the $L(x;n)$ sum and the Python implementations `L(x, n)` and `L2(x, n)` from Sect. 3.1.8. Instead of specifying the number of terms in the sum, n, it is better to specify a tolerance ε of the accuracy. We can use the first neglected term as an estimate of the accuracy. This means that we sum up terms as long as the absolute value of the next term is greater than ϵ. It is natural to provide a default value for ϵ:

```
def L3(x, epsilon=1.0E-6):
    x = float(x)
    i = 1
    term = (1.0/i)*(x/(1+x))**i
    s = term
    while abs(term) > epsilon:
        i += 1
        term = (1.0/i)*(x/(1+x))**i
        s += term
    return s, i
```

Here is an example involving this function to make a table of the approximation error as ϵ decreases:

Copyrighted material

Function input and output

It is a convention in Python that function arguments represent the input data to the function, while the returned objects represent the output data. We can sketch a general Python function as

```
def somefunc(i1, i2, i3, io4, io5, i6=value1, io7=value2):
    # modify io4, io5, io6; compute o1, o2, o3
    return o1, o2, o3, io4, io5, io7
```

Here `i1`, `i2`, `i3` are positional arguments representing input data; `io4` and `io5` are positional arguments representing input *and* output data; `i6` and `io7` are keyword arguments representing input and input/output data, respectively; and `o1`, `o2`, and `o3` are computed objects in the function, representing output data together with `io4`, `io5`, and `io7`. All examples later in the book will make use of this convention.

3.1.12 Functions as Arguments to Functions

Programs doing calculus frequently need to have functions as arguments in functions. For example, a mathematical function $f(x)$ is needed in Python functions for

- numerical root finding: solve $f(x) = 0$ approximately (Sects. 4.11.2 and A.1.10)
- numerical differentiation: compute $f'(x)$ approximately (Sects. B.2 and 7.3.2)
- numerical integration: compute $\int_a^b f(x)dx$ approximately (Sects. B.3 and 7.3.3)
- numerical solution of differential equations: $\frac{dx}{dt} = f(x)$ (Appendix E)

In such Python functions we need to have the $f(x)$ function as an argument `f`. This is straightforward in Python and hardly needs any explanation, but in most other languages special constructions must be used for transferring a function to another function as argument.

As an example, consider a function for computing the second-order derivative of a function $f(x)$ numerically:

$$f''(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}, \quad (3.3)$$

where h is a small number. The approximation (3.3) becomes exact in the limit $h \to 0$. A Python function for computing (3.3) can be implemented as follows:

```
def diff2nd(f, x, h=1E-6):
    r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)
    return r
```

The `f` argument is like any other argument, i.e., a name for an object, here a function object that we can call as we normally call functions. An application of `diff2nd` may be

Copyrighted material

```
def g(t):
    return t**(-6)

t = 1.2
d2g = diff2nd(g, t)
print "g''(%f)=%f" % (t, d2g)
```

The behavior of the numerical derivative as $h \to 0$ From mathematics we know that the approximation formula (3.3) becomes more accurate as h decreases. Let us try to demonstrate this expected feature by making a table of the second-order derivative of $g(t) = t^{-6}$ at $t = 1$ as $h \to 0$:

```
for k in range(1,15):
    h = 10**(-k)
    d2g = diff2nd(g, 1, h)
    print 'h=%.0e: %.5f' % (h, d2g)
```

The output becomes

```
h=1e-01: 44.61504
h=1e-02: 42.02521
h=1e-03: 42.00025
h=1e-04: 42.00000
h=1e-05: 41.99999
h=1e-06: 42.00074
h=1e-07: 41.94423
h=1e-08: 47.73959
h=1e-09: -666.13381
h=1e-10: 0.00000
h=1e-11: 0.00000
h=1e-12: -666133814.77509
h=1e-13: 66613381477.50939
h=1e-14: 0.00000
```

With $g(t) = t^{-6}$, the exact answer is $g''(1) = 42$, but for $h < 10^{-8}$ the computations give totally wrong answers! The problem is that for small h on a computer, rounding errors in the formula (3.3) blow up and destroy the accuracy. The mathematical result that (3.3) becomes an increasingly better approximation as h gets smaller and smaller does not hold on a computer! Or more precisely, the result holds until h in the present case reaches 10^{-4}.

The reason for the inaccuracy is that the numerator in (3.3) for $g(t) = t^{-6}$ and $t = 1$ contains subtraction of quantities that are almost equal. The result is a very small and inaccurate number. The inaccuracy is magnified by h^{-2}, a number that becomes very large for small h.

Switching from the standard floating-point numbers (`float`) to numbers with arbitrary high precision resolves the problem. Python has a module `decimal` that can be used for this purpose. The SymPy package comes with an alternative module `mpmath`, which also offers mathematical functions like `sin` and `cos` with arbitrary precision. The file `high_precision.py` solves the current problem using arithmetics also based on the `decimal` and `mpmath` modules. With 25 digits in `x` and `h`

Copyrighted material

inside the `diff2nd` function, we get accurate results for $h \leq 10^{-13}$ with `decimal`, while rounding errors show up for $h \geq 10^{10}$ with the `mpmath` module.

Nevertheless, for most practical applications of (3.3), a moderately small h, say $10^{-3} \leq h \leq 10^{-4}$, gives sufficient accuracy and then rounding errors from `float` calculations do not pose problems. Real-world science or engineering applications usually have many parameters with uncertainty, making the end result also uncertain, and formulas like (3.3) can then be computed with moderate accuracy without affecting the overall uncertainty in the answers.

3.1.13 The Main Program

In programs containing functions we often refer to a part of the program that is called the *main program*. This is the collection of all the statements outside the functions, plus the definition of all functions. Let us look at a complete program:

```
from math import *                  # in main

def f(x):                           # in main
    e = exp(-0.1*x)
    s = sin(6*pi*x)
    return e*s

x = 2                               # in main
y = f(x)                            # in main
print 'f(%g)=%g' % (x, y)           # in main
```

The main program here consists of the lines with a comment `in main`. The execution always starts with the first line in the main program. When a function is encountered, its statements are just used to define the function – nothing gets computed inside the function before we explicitly call the function, either from the main program or from another function. All variables initialized in the main program become global variables (see Sect. 3.1.3).

The program flow in the program above goes as follows:

- Import functions from the `math` module,
- define a function `f(x)`,
- define `x`,
- call `f` and execute the function body,
- define `y` as the value returned from `f`,
- print the string.

In point 4, we jump to the `f` function and execute the statement inside that function for the first time. Then we jump back to the main program and assign the `float` object returned from `f` to the `y` variable.

Readers who are uncertain about the program flow and the jumps between the main program and functions should use a debugger or the Online Python Tutor as explained in Sect. 3.1.2.

Copyrighted material

3.1.14 Lambda Functions

There is a quick one-line construction of functions that is often convenient to make Python code compact:

```
f = lambda x: x**2 + 4
```

This so-called *lambda function* is equivalent to writing

```
def f(x):
    return x**2 + 4
```

In general,

```
def g(arg1, arg2, arg3, ...):
    return expression
```

can be written as

```
g = lambda arg1, arg2, arg3, ...: expression
```

Lambda functions are usually used to quickly define a function as argument to another function. Consider, as an example, the `diff2nd` function from Sect. 3.1.12. In the example from that chapter we want to differentiate $g(t) = t^{-6}$ twice and first make a Python function `g(t)` and then send this `g` to `diff2nd` as argument. We can skip the step with defining the `g(t)` function and instead insert a lambda function as the `f` argument in the call to `diff2nd`:

```
d2 = diff2nd(lambda t: t**(-6), 1, h=1E-4)
```

Because lambda functions saves quite some typing, at least for very small functions, they are popular among many programmers.

Lambda functions may also take keyword arguments. For example,

```
d2 = diff2nd(lambda t, A=1, a=0.5: -a*2*t*A*exp(-a*t**2), 1.2)
```

3.2 Branching

The flow of computer programs often needs to branch. That is, if a condition is met, we do one thing, and if not, we do another thing. A simple example is a function defined as

$$f(x) = \begin{cases} \sin x, & 0 \le x \le \pi \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

In a Python implementation of this function we need to test on the value of x, which can be done as displayed below:

Copyrighted material

```
def f(x):
    if 0 <= x <= pi:
        value = sin(x)
    else:
        value = 0
    return value
```

3.2.1 If-else Blocks

The general structure of an `if-else` test is

```
if condition:
    <block of statements, executed if condition is True>
else:
    <block of statements, executed if condition is False>
```

When `condition` evaluates to `True`, the program flow branches into the first block of statements. If `condition` is `False`, the program flow jumps to the second block of statements, after the `else:` line. As with `while` and `for` loops, the block of statements are indented. Here is another example:

```
if C < -273.15:
    print '%g degrees Celsius is non-physical!' % C
    print 'The Fahrenheit temperature will not be computed.'
else:
    F = 9.0/5*C + 32
    print F
print 'end of program'
```

The two `print` statements in the `if` block are executed if and only if `C < -273.15` evaluates to `True`. Otherwise, we jump over the first two `print` statements and carry out the computation and printing of `F`. The printout of `end of program` will be performed regardless of the outcome of the `if` test since this statement is not indented and hence neither a part of the `if` block nor the `else` block.

The `else` part of an `if` test can be skipped, if desired:

```
if condition:
    <block of statements>
<next statement>
```

For example,

```
if C < -273.15:
    print '%s degrees Celsius is non-physical!' % C
F = 9.0/5*C + 32
```

In this case the computation of `F` will always be carried out, since the statement is not indented and hence not a part of the `if` block.

Copyrighted material

bioinformatics, a scientific discipline addressing the use of computers to search for, explore, and use information about genes, nucleic acids, and proteins.

The leading Python software for bioinformatics applications is BioPython[4]. The examples in this book (below and Sects. 6.5, 8.3.4, and 9.5) are simple illustrations of the type of problem settings and corresponding Python implementations that are encountered in bioinformatics. For real-world problem solving one should rather utilize BioPython, but the sections below act as an introduction to what is inside packages like BioPython.

We start with some very simple examples on DNA analysis that bring together basic building blocks in programming: loops, `if` tests, and functions.

3.3.1 Counting Letters in DNA Strings

Given some string `dna` containing the letters A, C, G, or T, representing the bases that make up DNA, we ask the question: how many times does a certain base occur in the DNA string? For example, if `dna` is ATGGCATTA and we ask how many times the base A occur in this string, the answer is 3.

A general Python implementation answering this problem can be done in many ways. Several possible solutions are presented below.

List iteration The most straightforward solution is to loop over the letters in the string, test if the current letter equals the desired one, and if so, increase a counter. Looping over the letters is obvious if the letters are stored in a list. This is easily done by converting a string to a list:

```
>>> list('ATGC')
['A', 'T', 'G', 'C']
```

Our first solution becomes

```
def count_v1(dna, base):
    dna = list(dna)  # convert string to list of letters
    i = 0            # counter
    for c in dna:
        if c == base:
            i += 1
    return i
```

String iteration Python allows us to iterate directly over a string without converting it to a list:

```
>>> for c in 'ATGC':
...     print c
A
T
G
C
```

[4] http://biopython.org

Copyrighted material

In fact, all built-in objects in Python that contain a set of elements in a particular sequence allow a `for` loop construction of the type `for element in object`.

A slight improvement of our solution is therefore to iterate directly over the string:

```
def count_v2(dna, base):
    i = 0 # counter
    for c in dna:
        if c == base:
            i += 1
    return i

dna = 'ATGCGGACCTAT'
base = 'C'
n = count_v2(dna, base)

# printf-style formatting
print '%s appears %d times in %s' % (base, n, dna)

# or (new) format string syntax
print '{base} appears {n} times in {dna}'.format(
    base=base, n=n, dna=dna)
```

We have here illustrated two alternative ways of writing out text where the value of variables are to be inserted in "slots" in the string.

Index iteration Although it is natural in Python to iterate over the letters in a string (or more generally over elements in a sequence), programmers with experience from other languages (Fortran, C and Java are examples) are used to `for` loops with an integer counter running over all indices in a string or array:

```
def count_v3(dna, base):
    i = 0 # counter
    for j in range(len(dna)):
        if dna[j] == base:
            i += 1
    return i
```

Python indices always start at 0 so the legal indices for our string become `0`, `1`, ..., `len(dna)-1`, where `len(dna)` is the number of letters in the string `dna`. The `range(x)` function returns a list of integers `0`, `1`, ..., `x-1`, implying that `range(len(dna))` generates all the legal indices for `dna`.

While loops The `while` loop equivalent to the last function reads

```
def count_v4(dna, base):
    i = 0 # counter
    j = 0 # string index
    while j < len(dna):
        if dna[j] == base:
            i += 1
        j += 1
    return i
```

Copyrighted material