A Primer on Scientific Programming with Python

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Aug 21, 2014

Preface

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

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

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

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

To work with this book, I recommend using Python version 2.7. For Chapters 5-9 and Appendices A-E you need the NumPy and Matplotlib packages, preferably also the IPython and SciTools packages, and for Appendix G Cython is required. Other packages used occasionally in the text are nose and `sympy`. Section H.1 has more information on how you can get access to Python and the mentioned packages.

There is a web page associated with this book, `http://hplgit.github.com/scipro-primer`, containing all the example programs from the book as well as information on installation of the software on various platforms.

Python version 2 or 3? A common problem among Python programmers is to choose between version 2 or 3, which at the time of this writing means choosing between version 2.7 and 3.4. The general recommendation is to go for Python 3, because this is the version that will be developed in the future. However, there is still a problem that much useful mathematical software in Python has not yet been ported to Python 3. Therefore, scientific computing with Python still goes mostly with version 2. A widely used strategy for software developers who want to write Python code that works with both versions, is to develop for version 2.7, which is very close to what is found version 3.4, and then use the translation tool *2to3* to automatically translate from Python 2 to Python 3.

When using v2.7, you should employ the newest syntax and modules that make the differences between Python 2 and 3 very small. This strategy is adopted in the present book. Only two differences between versions 2 and 3 are expected to be significant for the programs in the book: `a/b` for integers `a` and `b` implies float division Python 3 and integer division in Python 2. Moreover, `print 'Hello'` in Python 2 must be turned into a function call `print('Hello')` in Python 3. None of these differences should lead to any annoying problems when future readers study the book's v2.7 examples, but program in Python 3. Anyway, running `2to3` on the example files generates the corresponding Python 3 code.

Contents. Chapter 1 introduces variables, objects, modules, and text formatting through examples concerning evaluation of mathematical formulas. Chapter 2 presents programming with `while` and `for` loops as well as lists, including nested lists. The next chapter deals with two other fundamental concepts in programming: functions and `if-else` tests. Successful further reading of the book demands that Chapters 1-3 are digested.

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

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

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

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

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

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

Most of the examples and exercises in this book are quite short. However, many of the exercises are related, and together they form larger projects, for example on Fourier Series (3.15, 4.20, 4.21, 5.39, 5.40), numerical integration (3.6, 3.7, 5.47, 5.48, A.12), Taylor series (3.31, 5.30, 5.37, A.14, A.15, 7.23), piecewise constant functions (3.23-3.27, 5.32, 5.45, 5.46, 7.19-7.21), inverse functions (E.17-E.20), falling objects (E.8, E.9, E.38, E.39), oscillatory population growth (A.19, A.21, A.22, A.23), epidemic disease modeling (E.41-E.48), optimization and finance (A.24, 8.39, 8.40), statistics and probability (4.23, 4.24, 8.21, 8.22), hazard games (8.8-8.13), random walk and statistical physics (8.30-8.37), noisy data analysis (8.41-8.43), numerical methods (5.23-5.25, 7.8, 7.9, A.9, 7.22, 9.15-9.17, E.30-E.37), building a calculus calculator (7.33, 9.18, 9.19), and creating a toolkit for simulating vibrating engineering systems (E.50-E.55).

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

Changes from the third to the fourth edition. A large number of the exercises have been reformulated and reorganized. Typically, longer exercises are divided into subpoints a), b), c), etc., various type of help is factored out in separate paragraphs marked with **Hint**, and information that puts the exercise into a broader context is placed at the end under the heading **Remarks**. Quite some related exercises have been merged.

Another major change is the enforced focus on testing and verification. Already as soon as functions are introduced in Chapter 3, we start verifying the implementations through test functions written according to the conventions in the nose testing framework. This is continued throughout the book and especially incorporated in the reformulated exercises. Testing is challenging in programs containing unknown approximation errors, so strategies for finding appropriate test problems have also become an integral part of the fourth edition.

Many chapters now refer to the Online Python Tutor for visualizing the program flow. This is a splendid tool for learning what happens with the variables and execution of statements in small programs. The `sympy` package for symbolic computing is a powerful tool in scientific programming and introduced already in Chapter 1. The sections in Chapter 4 have been reorganized, and the basic information on file reading and writing was moved from Chapter 6 to Chapter 4. The fourth edition clearly features three distinct parts: basic programming concepts in Chapters 1-5, more advanced programming concepts in Chapters 6-9, and programming for solving science problems in Appendix A-E.

Sections 4.9 and 4.10.2 have been rewritten to emphasize the importance of test functions. The information on how to make animations and videos in Sections 5.3.4 and 5.3.5 has undergone a substantial revision. Section 6.1.7 has been completely rewritten to better reflect how to work with data associated with dates.

Appendix E has been reworked so that function-based programming and object-oriented programming appear in separate sections. This allows reading the appendix and solving ODEs without knowledge of classes and inheritance. Much of the text in Appendix E has been rewritten and extended, the exercises are substantially revised, and several new exercises have been added.

Section H.1 is new and describes the various options for getting access to Python and its packages for scientific computations. This topic includes, e.g., installing software on personal laptops and writing notebooks in cloud services.

In addition to the mentioned changes, a large number of smaller updates, improved explanations, and correction of typos have been incorporated in the new edition. I am very thankful to all the readers, instructors, and students who have sent emails with corrections or suggestions for improvements.

The perhaps biggest change for me was to move the whole manuscript from LaTeX to DocOnce[1]. This move enables a much more flexible composition of topics for various purposes, and support for output in different formats such as LaTeX, HTML, Sphinx, Markdown, IPython notebooks, and MediaWiki. The chapters have been made more independent by repeating key knowledge, which opens up for meaningful reading of only parts of the book, even the most advanced parts.

Acknowledgments. This book was born out of stimulating discussions with my close colleague Aslak Tveito, and he started writing what is now Appendix B and C. The whole book project and the associated university course were critically dependent on Aslak's enthusiastic role back in 2007. The continuous support from Aslak regarding my book projects is much appreciated and contributes greatly to my strong motivation. Another key contributor in the early days was Ilmar Wilbers. He made extensive efforts with assisting the book project and establishing the university course INF1100. I feel that without Ilmar and his solutions to numerous technical problems the first edition of the book would never have been completed. Johannes H. Ring also deserves special acknowledgment for the development of the Easyviz graphics tool and for his careful maintenance and support of software associated with the book over the years.

Professor Loyce Adams studied the entire book, solved all the exercises, found numerous errors, and suggested many improvements. Her contributions are so much appreciated. More recently, Helmut Büch worked extremely carefully through all details in Chapters 1-6, tested the software, found many typos, and asked critical questions that led to

[1] `https://github.com/hplgit/doconce`

lots of significant improvements. I am so thankful for all his efforts and for his enthusiasm during the preparations of the fourth edition.

Special thanks go to Geir Kjetil Sandve for being the primary author of the computational bioinformatics examples in Sections 3.3, 6.5, 8.3.4, and 9.5, with contributions from Sveinung Gundersen, Ksenia Khelik, Halfdan Rydbeck, and Kai Trengereid.

Several people have contributed with suggestions for improvements of the text, the exercises, and the associated software. I will in particular mention Ingrid Eide, Ståle Zerener Haugnæss, Kristian Hiorth, Arve Knudsen, Tobias Vidarssønn Langhoff, Martin Vonheim Larsen, Kine Veronica Lund, Solveig Masvie, Håkon Møller, Rebekka Mørken, Mathias Nedrebø, Marit Sandstad, Helene Norheim Semmerud, Lars Storjord, Fredrik Heffer Valdmanis, and Torkil Vederhus. Hakon Adler is greatly acknowledged for his careful reading of early various versions of the manuscript. Many thanks go to the professors Fred Espen Bent, Ørnulf Borgan, Geir Dahl, Knut Mørken, and Geir Pedersen for formulating several exciting exercises from various application fields. I also appreciate the cover image made by my good friend Jan Olav Langseth.

This book and the associated course are parts of a comprehensive and successful reform at the University of Oslo, called *Computing in Science Education*. The goal of the reform is to integrate computer programming and simulation in all bachelor courses in natural science where mathematical models are used. The present book lays the foundation for the modern computerized problem solving technique to be applied in later courses. It has been extremely inspiring to work closely with the driving forces behind this reform, especially the professors Morten Hjorth-Jensen, Anders Malthe-Sørenssen, Knut Mørken, and Arnt Inge Vistnes.

The excellent assistance from the Springer system, in particular Martin Peters, Thanh-Ha Le Thi, Ruth Allewelt, Peggy Glauch-Ruge, Nadja Kroke, Thomas Schmidt, Patrick Waltemate, Donatas Akmanavicius, and Yvonne Schlatter, is highly appreciated, and ensured a smooth and rapid production of all editions of this book.

Oslo, March 2014 *Hans Petter Langtangen*

Contents

List of Exercises

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, how you move between folders, how you change file and folder names, and how you write text and save it in a file. Another frequent word for folder is directory.

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] `http://tinyurl.com/pwyasaa`

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:

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

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 $y = 0$:

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

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 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 an editor like Gedit, 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.
- 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 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.1.7 and H.1.9 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.2.

Suppose you want to evaluate (1.1) for $v_0 = 1$ and $t = 0.1$. This is easy: move the cursor to the editor window, edit the program text to

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

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

Terminal

```
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 [12, 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.

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 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 `v_0` 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 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 Section 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. For example,

```
s = 1.8   # Å. Ødegård recommended this value for s
```

Running this program results in the error message

```
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 -*-
```

(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 Section 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 comma. 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)
```

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 padded with `x` 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

print """
At t=%f s, a ball with
initial velocity v0=%.3E m/s
is located at the height %.2f m.
""" % (t, v0, y)
```

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

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

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

Terminal

```
Terminal> python ball_print2.py

At t=0.600000 s, a ball with
initial velocity v0=5.000E+00 m/s
is located at the height 1.23 m.
```

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

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

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

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

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

```
print """
At t={t:f} s, a ball with
initial velocity v0={v0:.3E} m/s
is located at the height {y:.2f} m.
""".format(t=t, v0=v0, y=y)
```

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

```
print """y(t) is
the position of
our ball."""

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

result in identical output:

```
y(t) is
the position of
our ball.
```

1.2 Computer science glossary

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

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

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

The term *library* is widely used for a collection of generally useful program pieces that can be applied in many different contexts. Having access to good libraries means that you do not need to program code snippets that others have already programmed (most probable in a better way!). There are huge numbers of Python libraries. In Python terminology, the libraries are composed of *modules* and *packages*. Section 1.4 gives a first glimpse of the `math` module, which contains a set of standard mathematical functions for $\sin x$, $\cos x$, $\ln x$, 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 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:

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

```
v0 = 3
```

is an *assignment* statement, while

```
print y
```

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

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

Although most newcomers to computer programming will think they understand the meaning of the lines in the above program, it is important to be aware of some major differences between notation in a computer program and notation in mathematics. When you see the equality sign `=` in mathematics, it has a certain interpretation as an equation ($x + 2 = 5$) 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,

```
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 Section 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 Section 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 Section 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 Chapter 4). Output data can be printed in the terminal window, as in the current example, displayed as graphics on the screen, as done in Section 5.3, or stored in a file for later access, as explained in Section 4.6.

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

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

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

1.3 Another formula: Celsius-Fahrenheit conversion

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

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

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
```

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
```

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

At this stage, you do not need to know what an object really is, just think of an `int` object as a collection, say a storage box, with some information about an integer number. This information is stored somewhere in the computer's memory, and with the name `C` the program gets access to this information. The fundamental issue right now is that 21 and 21.0 are identical numbers in mathematics, while in a Python program 21 gives rise to an `int` object and 21.0 to a `float` object.

There are lots of different object types in Python, and you will later learn how to create your own customized objects. Some objects contain a lot of data, not just an integer or a real number. For example, when we write

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

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

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

1.3.3 Avoiding integer division

As a quite general rule of thumb, one should be careful to avoid integer division when programming mathematical formulas. In the rare cases when a mathematical algorithm does make use of integer division, one should use a double forward slash, `//`, as division operator, because this is Python's way of explicitly indicating integer division.

Python version 3 has no problem with unintended integer division, so the problem only arises with Python version 2 (and many other common languages for scientific computing). There are several ways to avoid integer division with the plain `/` operator. The simplest remedy in Python version 2 is to write

```
from __future__ import division
```

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

Terminal

```
Terminal> python -Qnew someprogram.py
```

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

```
F = (9.0/5)*C + 32
F = (9/5.0)*C + 32
F = float(C)*9/5 + 32
```

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

A related construction,

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

does not work correctly, because `9/5` is evaluated by integer division, yielding 1, before being multiplied by a `float` representation of `C` (see next section for how compound arithmetic operations are calculated). In other words, the formula reads `F=C+32`, which is wrong.

We now understand why the first version of the program does not work and what the remedy is. A correct program is

```
C = 21
F = (9.0/5)*C + 32
print F
```

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

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

```
Terminal

Terminal> python -Qwarnall someprogram.py
```

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

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

1.3.4 Arithmetic operators and precedence

Formulas in Python programs are usually evaluated in the same way as we would evaluate them mathematically. Python proceeds from left to right, term by term in an expression (terms are separated by plus or minus). In each term, power operations such as 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

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

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

$$\frac{1}{2} g t^2 - v_0 t + y_c = 0,$$

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 us to skip the module name prefix. This alternative syntax has the form `from module import function`. A specific example is

```
from math import sqrt
```

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

```
from math import sqrt, exp, log, sin
```

Sometimes one just writes

```
from math import *
```

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

With a `from math import sqrt` statement we can write the formulas for the roots in a more pleasing way:

```
t1 = (v0 - sqrt(v0**2 - 2*g*yc))/g
t2 = (v0 + sqrt(v0**2 - 2*g*yc))/g
```

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

```
import math as m
# m is now the name of the math module
v = m.sin(m.pi)

from math import log as ln
v = ln(5)

from math import sin as s, cos as c, log as ln
v = s(x)*c(x) + ln(x)
```

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

```
m = math
ln = m.log
s = m.sin
c = m.cos
```

1.4.2 Example: Computing with $\sinh x$

Our next examples involve calling some more mathematical functions from the `math` module. We look at the definition of the $\sinh(x)$ function:

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

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 round-off 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 16 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 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 with 16 digits, 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
```

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

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 *round-off errors*. As soon as you use Python interactively as explained in the next section, you will encounter round-off errors quite often.

Python has a special module `decimal` which allows real numbers to be represented with adjustable accuracy so that round-off errors can be made as small as desired (an example appears at the end of Section 3.1.12). However, we will hardly use this module because approximations implied by many mathematical methods applied throughout this book normally lead to (much) larger errors than those caused by round-off.

1.5 Interactive computing

A particular convenient feature of Python is the ability to execute statements and evaluate expressions interactively. The environments where you work interactively with programming are commonly known as Python *shells*. The simplest Python shell is invoked by just typing `python` at the command line in a terminal window. Some messages about Python are written out together with a prompt >>>, after which you can issue commands. Let us try to use the interactive shell as a calculator. Type in `3*4.5-0.5` and then press the Return key to see Python's response to this expression:

```
Terminal> python
Python 2.7.5+ (default, Sep 19 2013, 13:48:49)
[GCC 4.8.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 3*4.5-0.5
13.0
```

The text on a line after >>> is what we write (shell input) and the text without the >>> prompt is the result that Python calculates (shell output). It is easy, as explained below, to recover previous input and edit the text. This editing feature makes it convenient to experiment with statements and expressions.

1.5.1 Using the Python shell

The program from Section 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 Section 1.4.3 computations on a computer often suffer from round-off errors. The present calculation is no exception. The correct answer is `1.8342`, but round-off errors lead to a number that is incorrect in the 16th decimal. The error is here $4 \cdot 10^{-16}$.

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 Section 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
```

1.5.3 IPython

There exist several improvements of the standard Python shell presented in Section 1.5. The author advocates IPython as the preferred interactive shell. You will then need to have IPython installed. Typing `ipython` in a terminal window starts the shell. The (default) prompt in IPython is not >>> but `In [X]:`, where `X` is the number of the present input command. The most widely used features of IPython are summarized below.

Running programs. Python programs can be run from within the shell:

```
In [1]: run ball2.py
1.2342
```

This command requires that you have taken a `cd` to the folder where the `ball2.py` program is located and started IPython from there.

On Windows you may, as an alternative to starting IPython from a DOS or PowerShell window, double click on the IPython desktop icon or use the Start menu. In that case, you must move to the right folder where your program is located. This is done by the `os.chdir` (change directory) command. Typically, you write something like

```
In [1]: import os
In [2]: os.chdir(r'C:\Documents and Settings\me\My Documents\div')
In [3]: run ball2.py
```

if the `ball2.py` program is located in the folder `div` under `My Documents` of user `me`. Note the `r` before the quote in the string: it is required to let a backslash really mean the backslash character. If you end up typing the `os.chdir` command every time you enter an IPython shell, this command (and others) can be placed in a *startup file* such that they are automatically executed when you launch IPython.

Inside IPython you can invoke any operating system command. This allows us to navigate to the right folder above using Unix or Windows (`cd`) rather than Python (`os.chdir`):

```
In [1]: cd C:\Documents and Settings\me\My Documents\div
In [3]: run ball2.py
```

We recommend running all your Python programs from the IPython shell. Especially when something goes wrong, IPython can help you to examine the state of variables so that you become quicker to locate bugs.

Typesetting convention for executing Python programs

In the rest of the book, we just write the program name and the output when we illustrate the execution of a program:

Terminal

```
ball2.py
1.2342
```

You then need to write `run` before the program name if you execute the program in IPython, or if you prefer to run the program directly in a terminal window, you need to write `python` prior to the program name. Appendix H.2 describes various other ways to run a Python program.

Quick recovery of previous output. The results of the previous statements in an interactive IPython session are available in variables of the form `_iX` (underscore, `i`, and a number `X`), where `X` is 1 for the last statement, 2 for the second last statement, and so forth. Short forms are `_` for `_i1`, `__` for `_i2`, and `___` for `_i3`. The output from the `In [1]` input above is `1.2342`. We can now refer to this number by an underscore and, e.g., multiply it by 10:

```
In [2]: _*10
Out[2]: 12.341999999999999
```

Output from Python statements or expressions in IPython are preceded by `Out[X]` where `X` is the command number corresponding to the previous `In [X]` prompt. When programs are executed, as with the `run` command, or when operating system commands are run (as shown below), the output is from the operating system and then not preceded by any `Out[X]` label.

The command history from previous IPython sessions is available in a new session. This feature makes it easy to modify work from a previous session by just hitting the up-arrow to recall commands and edit them as necessary.

Tab completion. Pressing the TAB key will complete an incompletely typed variable name. For example, after defining `my_long_variable_name = 4`, write just `my` at the `In [4]:` prompt below, and then hit the TAB key. You will experience that `my` is immediately expanded to `my_long_variable_name`. This automatic expansion feature is called TAB completion and can save you from quite some typing.

```
In [3]: my_long_variable_name = 4

In [4]: my_long_variable_name
Out[4]: 4
```

Recovering previous commands. You can walk through the command history by typing `Ctrl+p` or the up arrow for going backward or `Ctrl+n` or the down arrow for going forward. Any command you hit can be edited and re-executed. Also commands from previous interactive sessions are stored in the command history.

Running Unix/Windows commands. Operating system commands can be run from IPython. Below we run the three Unix commands `date`, `ls` (list files), `mkdir` (make directory), and `cd` (change directory):

```
In [5]: date
Thu Nov 18 11:06:16 CET 2010

In [6]: ls
myfile.py  yourprog.py
```

```
In [7]: mkdir mytestdir

In [8]: cd mytestdir
```

If you have defined Python variables with the same name as operating system commands, e.g., `date=30`, you must write `!date` to run the corresponding operating system command.

IPython can do much more than what is shown here, but the advanced features and the documentation of them probably do not make sense before you are more experienced with Python - and with reading manuals.

Typesetting of interactive shells in this book

In the rest of the book we will apply the >>> prompt in interactive sessions instead of the input and output prompts as used by default by IPython, simply because most Python books and electronic manuals use >>> to mark input in interactive shells. However, when you sit by the computer and want to use an interactive shell, we recommend using IPython, and then you will see the `In [X]` prompt instead of >>>.

Notebooks. A particularly interesting feature of IPython is the notebook, which allows you to record and replay exploratory interactive sessions with a mix of text, mathematics, Python code, and graphics. See Section H.1.9 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 Section 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 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:

```
>>> 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 *
```

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 $f(x) = ax^2 + bx + c$:

```
>>> 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.

Import of SymPy functionality is often done via `from sympy import *`, but below we shall explicitly import each symbol we need to emphasize that the symbol comes from SymPy. For example, it will be important to know whether `sin` means the sine function from the `math` module or the special sine function from `sympy` aimed at symbolic computing.

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:

```
>>> from sympy import (
...     symbols,   # define mathematical symbols for symbolic math
...     diff,      # differentiate expressions
...     integrate, # integrate expressions
...     Rational,  # define rational numbers
...     lambdify,  # turn symbolic expressions 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` 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`. Let us take the `dydt` expression above and turn it into a Python function `v(t, v0, g)`:

```
>>> v = lambdify([t, v0, g],  # arguments in v
                 dydt)        # symbolic expression
>>> v(0, 5, 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 and Taylor series

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 $y = 0$:

```
>>> # Find t values such that 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

```
>>> y.subs(t, roots[0])
0
>>> y.subs(t, roots[1])
0
```

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!

Variables. The statement

```
some_variable = obj
```

defines a variable with the name `some_variable` which refers to an object `obj`. Here `obj` may also represent an expression, say a formula, whose value is a Python object. For example, `1+2.5` involves the addition of an `int` object and a `float` object, resulting in a `float` object. Names of variables can contain upper and lower case English letters, underscores, and the digits from 0 to 9, but the name cannot start with a digit. Nor can a variable name be a reserved word in Python.

If there exists a precise mathematical description of the problem to be solved in a program, one should choose variable names that are in accordance with the mathematical description. Quantities that do not have a defined mathematical symbol, should be referred to by *descriptive* variables names, i.e., names that explain the variable's role in the program. Well-chosen variable names are essential for making a program easy to read, easy to debug, and easy to extend. Well-chosen variable names also reduce the need for comments.

Comment lines. Everything after # on a line is ignored by Python and used to insert free running text, known as *comments*. The purpose of comments is to explain, in a human language, the ideas of (several) forthcoming statements so that the program becomes easier to understand for humans. Some variables whose names are not completely self-explanatory also need a comment.

Object types. There are many different types of objects in Python. In this chapter we have worked with the following types.
Integers (whole numbers, object type `int`):

```
x10 = 3
XYZ = 2
```

Floats (decimal numbers, object type `float`):

```
max_temperature = 3.0
MinTemp = 1/6.0
```

Strings (pieces of text, object type `str`):

```
a = 'This is a piece of text\nover two lines.'
b = "Strings are enclosed in single or double quotes."
c = """Triple-quoted strings can
span
several lines.
"""
```

Complex numbers (object type `complex`):

```
a = 2.5 + 3j
real = 6; imag = 3.1
b = complex(real, imag)
```

Operators. Operators in arithmetic expressions follow the rules from mathematics: power is evaluated before multiplication and division, while the latter two are evaluated before addition and subtraction. These rules are overridden by parentheses. We suggest using parentheses to group and clarify mathematical expressions, also when not strictly needed.

```
-t**2*g/2
-(t**2)*(g/2)          # equivalent
-t**(2*g)/2            # a different formula!
```

```
a = 5.0; b = 5.0; c = 5.0
a/b + c + a*c         # yields 31.0
a/(b + c) + a*c       # yields 25.5
a/(b + c + a)*c       # yields 1.6666666666666665
```

Particular attention must be paid to coding fractions, since the division operator `/` often needs extra parentheses that are not necessary in the mathematical notation for fractions (compare $\frac{a}{b+c}$ 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
```

The printf syntax enables full control of the formatting of real numbers and integers:

```
>>> print 'a=%g, b=%12.4E, c=%.2f, d=%5d' % (a, b, c, d)
a=5, b= -5.0000E+00, c=1.99, d=   33
```

Here, `a`, `b`, and `c` are of type `float` and formatted as compactly as possible (`%g` for `a`), in scientific notation with 4 decimals in a field of width 12 (`%12.4E` for `b`), and in decimal notation with two decimals in as compact field as possible (`%.2f` for `c`). The variable `d` is an integer (`int`) written in a field of width 5 characters (`%5d`).

Be careful with integer division!

A common error in mathematical computations is to divide two integers, because this results in integer division (in Python 2).

- Any number written without decimals is treated as an integer. To avoid integer division, ensure that every division involves at least one real number, e.g., `9/5` is written as `9.0/5`, `9./5`, `9/5.`, or `9/5.0`.
- In expressions with variables, `a/b`, ensure that `a` or `b` is a `float` object, and if not (or uncertain), do an explicit conversion as in `float(a)/b` to guarantee float division.
- If integer division is desired, use a double slash: `a//b`.
- Python 3 treats `a/b` as float division also when `a` and `b` are integers.

Complex numbers. Values of complex numbers are written as `(X+Yj)`, where `X` is the value of the real part and `Y` is the value of the imaginary part. One example is `(4-0.2j)`. If the real and imaginary parts are available as variables `r` and `i`, a complex number can be created by `complex(r, i)`.

The `cmath` module must be used instead of `math` if the argument is a complex variable. The `numpy` package offers similar mathematical functions, but with a unified treatment of real and complex variables.

Terminology. Some Python and computer science terms briefly covered in this chapter are

- object: anything that a variable (name) can refer to, such as a number, string, function, or module (but objects can exist without being bound to a name: `print 'Hello!'` first makes a string object of the text in quotes and then the contents of this string object, without a name, is printed)
- variable: name of an object
- statement: an instruction to the computer, usually written on a line in a Python program (multiple statements on a line must be separated by semicolons)
- expression: a combination of numbers, text, variables, and operators that results in a new object, when being evaluated
- assignment: a statement binding an evaluated expression (object) to a variable (name)
- algorithm: detailed recipe for how to solve a problem by programming
- code: program text (or synonym for program)
- implementation: same as code
- executable: the file we run to start the program

- verification: providing evidence that the program works correctly
- debugging: locating and correcting errors in a program

1.8.2 Example: Trajectory of a ball

Problem. What is the trajectory of a ball that is thrown or kicked with an initial velocity 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

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

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 ($v_0 = 10$ km/h) of a football, the gravity force is about 120 times larger than the air resistance. 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

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 vertical line to the left:

```
C = 21
F = (9.0/5)*C + 32
print F
```

As a reader of this book, you may wonder if a code shown is a complete program you can try out or if it is just a part of a program (a snippet) so that you need to add surrounding statements (e.g., import statements) to try the code out yourself. The appearance of a vertical line to the left or not will then quickly tell you what type of code you see.

An interactive Python session is typeset as

```
>>> from math import *
>>> p = 1; c = -1.5
>>> a = sqrt(4*p + c)
```

Running a program, say `ball_yc.py`, in the terminal window, followed by some possible output is typeset as

Terminal

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

Recall from Section 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.2 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

What does it mean to solve an exercise? The solution to most of the exercises in this book is a Python program. To produce the solution, you first need understand the problem and what the program is supposed to do, and then you need to understand how to translate the problem description into a series of Python statements. Equally important is the verification (testing) of the program. A complete solution to a programming exercises therefore consists of two parts: 1) the program text and 2) a demonstration that the program works correctly. Some simple programs, like the ones in the first two exercises below, have so obviously correct output that the verification can just be to run the program and record the output.

In cases where the correctness of the output is not obvious, it is necessary to prove or bring evidence that the result is correct. This can be done through comparisons with calculations done separately on a calculator, or one can apply the program to a special simple test case

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

with known results. The requirement is to provide evidence to the claim that the program is without programming errors.

The sample run of the program to check its correctness can be inserted at the end of the program as a triple-quoted string. Alternatively, the output lines can be inserted as comments, but using a multi-line string requires less typing. (Technically, a string object is created, but not assigned to any name or used for anything in the program beyond providing useful information for the reader of the code.) One can do

Terminal

```
Terminal> python myprogram.py > result
```

and use a text editor to insert the file `result` inside the triple-quoted multi-line string. Here is an example on a run of a Fahrenheit to Celsius conversion program inserted at the end as a triple-quoted string:

```
F = 69.8               # Fahrenheit degrees
C = (5.0/9)*(F - 32)   # Corresponding Celsius degrees
print C

'''
Sample run (correct result is 21):
python f2c.py
21.0
'''
```

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.py`.

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.py`.

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.py`.

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.py`.

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.py`.

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.py`.

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?

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`

```
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.py`.

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 $s = v_0t + 0,5at^2$, 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

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: `sin2_plus_cos2.py`.

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.py`.

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, the mass is 0.43 kg, and C_D can be taken as 0.2.

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

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

($\mathrm{N} = \mathrm{kg\,m/s^2}$). Also print the ratio of the drag force and the gravity force. Define C_D, ϱ, A, V, m, g, 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, $V = 10$ km/h (it is easy to mix inconsistent units, so make sure you compute with V expressed in m/s). Filename: `kick.py`.

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.py`.

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 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., [16]. 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
```

Exercise 1.15: Explain why a program does not work

Find out why the following program does not work:

```
C = A + B
A = 3
B = 2
print C
```

Exercise 1.16: Find errors in Python statements

Try the following statements in an interactive Python shell. Explain why some statements fail and correct the errors.

```
1a = 2
a1 = b
x = 2
y = X + 4  # is it 6?
from Math import tan
print tan(pi)
pi = "3.14159’
print tan(pi)
c = 4**3**2**3
_ = ((c-78564)/c + 32))
discount = 12%
AMOUNT = 120.-
amount = 120$
address = hpl@simula.no
and = duck
class = ’INF1100, gr 2"
continue_ = x > 0
rev = fox = True
Norwegian = [’a human language’]
true = fox is rev in Norwegian
```

Hint. It is wise to test the values of the expressions on the right-hand side, and the validity of the variable names, separately before you put the left- and right-hand sides together in statements. The last two statements work, but explaining why goes beyond what is treated in this chapter.

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.py`.

2 Loops and lists

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 Chapter 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

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

for the table above. Using 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 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 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 == 40`, we first evaluate `C == 40`, for `C = 1` this is `False`, and then `not` turns the value into `True`. On the opposite, if `C == 40` is `True`, `not C == 40` becomes `False`. Mathematically it is easier to read `C != 40` than `not C == 40`, but these two boolean expressions are equivalent.

Boolean expressions can be combined with `and` and `or` to form new compound boolean expressions, as in

```
while x > 0 and y <= 1:
    print x, y
```

If `cond1` and `cond2` are two boolean expressions with values `True` or `False`, the compound boolean expression `cond1 and cond2` is `True` if both `cond1` and `cond2` are `True`. On the other hand, `cond1 or cond2` is `True` if at least one of the conditions, `cond1` or `cond2`, is `True`

Remark

In Python, `cond1 and cond2` or `cond1 or cond2` returns one of the operands and not just `True` or `False` values as in most other computer languages. The operands `cond1` or `cond2` can be expressions or objects. In case of expressions, these are first evaluated to an object before the compound boolean expression is evaluated. For example, `(5+1) or -1` evaluates to `6` (the second operand is not evaluated when the first one is `True`), and `(5+1) and -1` evaluates to `-1`.

Here are some more examples from an interactive session where we just evaluate the boolean expressions themselves without using them in loop conditions:

```
>>> x = 0;  y = 1.2
>>> x >= 0 and y < 1
False
>>> x >= 0 or y < 1
True
>>> x > 0 or y > 1
True
>>> x > 0 or not y > 1
False
>>> -1 < x <= 0   #  -1 < x and x <= 0
True
>>> not (x > 0 or y > 0)
False
```

In the last sample expression, `not` applies to the value of the boolean expression inside the parentheses: `x>0` is `False`, `y>0` is `True`, so the combined expression with `or` is `True`, and `not` turns this value to `False`.

The common boolean values in Python are `True`, `False`, `0` (false), and any integer different from zero (true). To see such values in action, we recommend doing Exercises 2.21 and 2.17.

Boolean evaluation of an object

All objects in Python can in fact be evaluated in a boolean context, and all are `True` except `False`, zero numbers, and empty strings, lists, and dictionaries:

```
>>> s = 'some string'
>>> bool(s)
True
>>> s = ''  # empty string
>>> bool(s)
False
>>> L = [1, 4, 6]
>>> bool(L)
True
>>> L = []
>>> bool(L)
False
>>> a = 88.0
>>> bool(a)
True
>>> a = 0.0
>>> bool(a)
False
```

Essentially, `if a` tests if `a` is a non-empty object or if it is non-zero value. Such constructions are frequent in Python code.

Erroneous thinking about boolean expressions is one of the most common sources of errors in computer programs, so you should be careful every time you encounter a boolean expression and check that it is correctly stated.

2.1.4 Loop implementation of a sum

Summations frequently appear in mathematics. For instance, the sine function can be calculated as a polynomial:

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots, \tag{2.1}$$

where $3! = 3 \cdot 2 \cdot 1$, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, etc., are factorial expressions. Computing $k! = k(k-1)(k-2)\cdots 2 \cdot 1$ is done by `math.factorial(k)`.

An infinite number of terms are needed on the right-hand side of (2.1) for the equality sign to hold. With a finite number of terms, we obtain an approximation to $\sin(x)$, which is well suited for being calculated in a program since only powers and the basic four arithmetic operations are involved. Say we want to compute the right-hand side of (2.1) for powers up to $N = 25$. Writing out and implementing each one of these terms is a tedious job that can easily be automated by a loop.

Computation of the sum in (2.1) by a `while` loop in Python, makes use of (i) a counter `k` that runs through odd numbers from 1 up to some given maximum power `N`, and (ii) a summation variable, say `s`, which accumulates the terms, one at a time. The purpose of each pass of the loop is to compute a new term and add it to `s`. Since the sign of each term alternates, we introduce a variable `sign` that changes between -1 and 1 in each pass of the loop.

The previous paragraph can be precisely expressed by this piece of Python code:

```
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 Section 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.

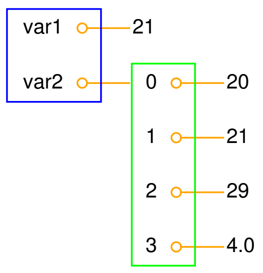


Fig. 2.1 Illustration of two variables: `var1` refers to an `int` object with value 21, created by the statement `var1 = 21`, and `var2` refers to a `list` object with value `[20, 21, 29, 4.0]`, i.e., three `int` objects and one `float` object, created by the statement `var2 = [20, 21, 29, 4.0]`.

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]
```

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 Chapter 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 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 Section 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:

```
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 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 2.22 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 Section 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:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
print '    C    F'
for C in Cdegrees:
    F = (9.0/5)*C + 32
    print '%5d %5.1f' % (C, F)
```

This code is found in the file `c2f_table_list.py` and its output becomes

```
    C    F
  -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
```

2.3 Alternative implementations with lists and loops

We have already solved the problem of printing out a nice-looking conversion table for Celsius and Fahrenheit degrees. Nevertheless, there are usually many alternative ways to write a program that solves a specific problem. The next paragraphs explore some other possible Python constructs and programs to store numbers in lists and print out tables. The various code snippets are collected in the program file `session.py`.

2.3.1 While loop implementation of a for loop

Any `for` loop can be implemented as a `while` loop. The general code

```
for element in somelist:
    <process element>
```

can be transformed to this `while` loop:

```
index = 0
while index < len(somelist):
    element = somelist[index]
    <process element>
    index += 1
```

In particular, the example involving the printout of a table of Celsius and Fahrenheit degrees can be implemented as follows in terms of a `while` loop:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
index = 0
print '    C    F'
while index < len(Cdegrees):
    C = Cdegrees[index]
    F = (9.0/5)*C + 32
    print '%5d %5.1f' % (C, F)
    index += 1
```

2.3.2 The range construction

It is tedious to write the many elements in the `Cdegrees` in the previous programs. We should use a loop to automate the construction of the `Cdegrees` list. The `range` construction is particularly useful in this regard:

- `range(n)` generates integers `0, 1, 2, ..., n-1`.
- `range(start, stop, step)` generates a sequence if integers `start`, `start+step`, `start+2*step`, and so on up to, *but not including*, `stop`. For example, `range(2, 8, 3)` returns 2 and 5 (and not 8), while `range(1, 11, 2)` returns 1, 3, 5, 7, 9.
- `range(start, stop)` is the same as `range(start, stop, 1)`.

A `for` loop over integers are written as

```
for i in range(start, stop, step):
    ...
```

We can use this construction to create a `Cdegrees` list of the values $-20, -15, \ldots, 40$:

```
Cdegrees = []
for C in range(-20, 45, 5):
    Cdegrees.append(C)

# or just
Cdegrees = range(-20, 45, 5)
```

Note that the upper limit must be greater than `40` to ensure that `40` is included in the range of integers.

Suppose we want to create `Cdegrees` as $-10, -7.5, -5, \ldots, 40$. This time we cannot use `range` directly, because `range` can only create integers and we have decimal degrees such as -7.5 and 1.5. In this more general case, we introduce an integer counter i and generate the C values by the formula $C = -10 + i \cdot 2.5$ for $i = 0, 1, \ldots, 20$. The following Python code implements this task:

```
Cdegrees = []
for i in range(0, 21):
     C = -10 + i*2.5
     Cdegrees.append(C)
```

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
```

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

```
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.
```

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]`.

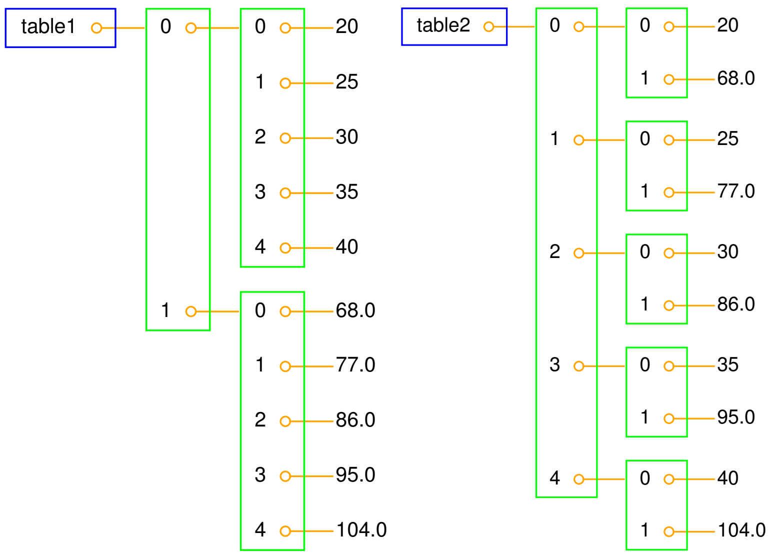


Fig. 2.2 Two ways of creating a table as a nested list. Left: table of columns `C` and `F`, where `C` and `F` are lists. Right: table of rows, where each row `[C, F]` is a list of two floats.

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:

```
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]]
```

Splitting the output over several shorter lines makes the layout nicer and more readable. The `pprint` module offers a *pretty print* functionality for this purpose. The usage of `pprint` looks like

```
import pprint
pprint.pprint(table)
```

and the corresponding 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]]
```

With this book comes a slightly modified `pprint` module having the name `scitools.pprint2`. This module allows full format control of the printing of the `float` objects in lists by specifying `scitools.pprint2.float_format` as a printf format string. The following example demonstrates how the output format of real numbers can be changed:

```
>>> import pprint, scitools.pprint2
>>> somelist = [15.8, [0.2, 1.7]]
>>> pprint.pprint(somelist)
[15.800000000000001, [0.20000000000000001, 1.7]]
>>> scitools.pprint2.pprint(somelist)
[15.8, [0.2, 1.7]]
>>> # default output is '%g', change this to
>>> scitools.pprint2.float_format = '%.2e'
>>> scitools.pprint2.pprint(somelist)
[1.58e+01, [2.00e-01, 1.70e+00]]
```

As can be seen from this session, the `pprint` module writes floating-point numbers with a lot of digits, in fact so many that we explicitly see the round-off errors. Many find this type of output is annoying and that the default output from the `scitools.pprint2` module is more like one would desire and expect.

The `pprint` and `scitools.pprint2` modules also have a function `pformat`, which works as the `pprint` function, but it returns a pretty formatted string rather than printing the string:

```
s = pprint.pformat(somelist)
print s
```

This last `print` statement prints the same as `pprint.pprint(somelist)`.

Manual printing. Many will argue that tabular data such as those stored in the nested `table` list are not printed in a particularly pretty way by the `pprint` module. One would rather expect pretty output to be a table with two nicely aligned columns. To produce such output we need to code the formatting manually. This is quite easy: we loop over each row, extract the two elements `C` and `F` in each row, and print these in fixed-width fields using the printf syntax. The code goes as follows:

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

2.4.3 Extracting sublists

Python has a nice syntax for extracting parts of a list structure. Such parts are known as *sublists* or *slices*:

`A[i:]` is the sublist starting with index `i` in `A` and continuing to the end of `A`:

```
>>> A = [2, 3.5, 8, 10]
>>> A[2:]
[8, 10]
```

`A[i:j]` is the sublist starting with index `i` in `A` and continuing up to and including index `j-1`. Make sure you remember that the element corresponding to index `j` is not included in the sublist:

```
>>> A[1:3]
[3.5, 8]
```

`A[:i]` is the sublist starting with index 0 in `A` and continuing up to and including the element with index `i-1`:

```
>>> A[:3]
[2, 3.5, 8]
```

`A[1:-1]` extracts all elements except the first and the last (recall that index `-1` refers to the last element), and `A[:]` is the whole list:

```
>>> A[1:-1]
[3.5, 8]
>>> A[:]
[2, 3.5, 8, 10]
```

In nested lists we may use slices in the first index, e.g.,

```
>>> table[4:]
[[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]]
```

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 Section 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.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 $6, 7, \ldots, 10$ 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.

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
```

We recommend to do Exercise 3.29 to get a better understanding of nested `for` loops.

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.
- 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 Section 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`)
`a.remove(e)`	remove an element with value `e`
`a.index('run.py')`	find index corresponding to an element's value
`'run.py' in a`	test if a value is contained in the list
`a.count(v)`	count how many elements that have the value `v`
`len(a)`	number of elements in list `a`
`min(a)`	the smallest element in `a`
`max(a)`	the largest element in `a`
`sum(a)`	add all elements in `a`
`sorted(a)`	return sorted version of list `a`
`reversed(a)`	return reversed sorted version of list `a`
`b[3][0][2]`	nested list indexing
`isinstance(a, list)`	is `True` if `a` is a list
`type(a) is list`	is `True` if `a` is a list

Nested lists. If the list elements are also lists, we have a nested list. The following session summarizes indexing and loop traversal of nested lists:

```
>>> nl = [[0, 0, 1], [-1, -1, 2], [-10, 10, 5]]
>>> nl[0]
[0, 0, 1]
>>> nl[-1]
[-10, 10, 5]
>>> nl[0][2]
1
>>> nl[-1][0]
-10
>>> for p in nl:
...     print p
...
[0, 0, 1]
[-1, -1, 2]
[-10, 10, 5]
>>> for a, b, c in nl:
...     print '%3d %3d %3d' % (a, b, c)
...
  0   0   1
 -1  -1   2
-10  10   5
```

Tuples. A tuple can be viewed as a constant list: no changes in the contents of the tuple is allowed. Tuples employ standard parentheses or no parentheses, and elements are separated with comma as in lists:

```
>>> mytuple = (t, dt, T, 'mynumbers.dat', 100)
>>> mytuple =  t, dt, T, 'mynumbers.dat', 100
```

Many list operations are also valid for tuples, but those that changes the list content cannot be used with tuples (examples are `append`, `del`, `remove`, `index`, and `sort`).

An object `a` containing an ordered collection of other objects such that `a[i]` refers to object with index `i` in the collection, is known as a *sequence* in Python. Lists, tuples, strings, and arrays are examples on sequences. You choose a sequence type when there is a natural ordering of elements. For a collection of unordered objects a *dictionary* (see Section 6.1) is often more convenient.

For loops. A `for` loop is used to run through the elements of a list or a tuple:

```
>>> for elem in [10, 20, 25, 27, 28.5]:
...     print elem,
...
10 20 25 27 28.5
```

The trailing comma after the `print` statement prevents the newline character, which otherwise `print` would automatically add.

The `range` function is frequently used in `for` loops over a sequence of integers. Recall that `range(start, stop, inc)` does not include the upper limit `stop` among the list item.

```
>>> for elem in range(1, 5, 2):
...     print elem,
...
1 3
>>> range(1, 5, 2)
[1, 3]
```

Implementation of a sum $\sum_{j=M}^{N} q(j)$, where $q(j)$ is some mathematical expression involving the integer counter j, is normally implemented using a `for` loop. Choosing, e.g., $q(j) = 1/j^2$, the sum is calculated by

```
s = 0  # accumulation variable
for j in range(M, N+1, 1):
    s += 1./j**2
```

Pretty print. To print a list `a`, `print a` can be used, but the `pprint` and `scitools.pprint2` modules and their `pprint` function give a nicer layout of the output for long and nested lists. The `scitools.pprint2` module has the possibility to control the formatting of floating-point numbers.

Terminology. The important computer science terms in this chapter are

- list
- tuple
- nested list (and nested tuple)
- sublist (subtuple) or slice `a[i:j]`
- `while` loop
- `for` loop
- list comprehension
- boolean expression

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 + `i` and month 1 + `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 $(2009 - 1929 + 1)$.

When looping over years and months it is convenient to have loop variables running over the true years (1929 to 2009) and the standard

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

month number (1 to 12). These variables must be correctly translated to indices in the `data` list such that all indices start at 0. The following code produces the answers to task 1:

```
monthly_mean = [0]*12      # list with 12 elements
for month in range(1, 13):
    m = month - 1   # corresponding list index (starts at 0)
    s = 0           # sum
    n = 2009 - 1929 + 1  # no of years
    for year in range(1929, 2010):
        y = year - 1929  # corresponding list index (starts at 0)
        s += data[y][m]
    monthly_mean[m] = 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 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: 55.9
Feb: 71.8
Mar: 115.1
Apr: 151.3
May: 188.7
Jun: 196.1
Jul: 191.4
Aug: 182.1
Sep: 136.7
Oct: 103.4
Nov: 66.6
Dec: 51.7
```

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 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 Section 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

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

math, which quickly leads you to the official 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 [18] 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 [14].

[5] `http://wiki.python.org/moin/PythonBooks`

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.

2.7 Exercises

Exercise 2.1: Make a Fahrenheit-Celsius conversion table

Write a Python program that prints out a table with Fahrenheit degrees $0, 10, 20, \ldots, 100$ in the first column and the corresponding Celsius degrees in the second column.

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

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.py`.

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.py`.

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.py`.

[6] `http://rgruet.free.fr/PQR27/PQR2.7.html`

Exercise 2.5: Sum of 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.py`.

Exercise 2.6: Generate equally spaced coordinates

We want to generate $n+1$ equally spaced x coordinates in $[a, b]$. Store the coordinates in a list.

a) Start with an empty list, use a `for` loop and append each coordinate to the list.

Hint. With n intervals, corresponding to $n+1$ points, in $[a, b]$, each interval has length $h = (b-a)/n$. The coordinates can then be generated by the formula $x_i = a + ih$, $i = 0, \ldots, n+1$.

b) Use a list comprehension (see Section 2.3.5).
Filename: `coor.py`.

Exercise 2.7: Make a table of values from a formula

The purpose of this exercise is to write code that prints a nicely formatted table of t and $y(t)$ values, where

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

Use $n+1$ uniformly spaced t values throughout the interval $[0, 2v_0/g]$.

a) Use a `for` loop to produce the table.

b) Add code with a `while` loop to produce the table.

Hint. Because of potential round-off errors, you may need to adjust the upper limit of the `while` loop to ensure that the last point ($t = 2v_0/g$, $y = 0$) is included.
Filename: `ball_table1.py`.

Exercise 2.8: Store values from a formula in lists

This exercise aims to produce the same table of numbers as in Exercise 2.7, but with different code. First, store the t and y values in two lists `t` and `y`. Thereafter, write out a nicely formatted table by traversing the two lists with a `for` loop.

Hint. In the `for` loop, use either `zip` to traverse the two lists in parallel, or use an index and the `range` construction.
Filename: `ball_table2.py`.

Exercise 2.9: Simulate operations on lists by hand

You are given the following program:

```
a = [1, 3, 5, 7, 11]
b = [13, 17]
c = a + b
print c
b[0] = -1
d = [e+1 for e in a]
print d
d.append(b[0] + 1)
d.append(b[-1] + 1)
print d[-2:]
for e1 in a:
    for e2 in b:
        print e1 + e2
```

Go through each statement and explain what is printed by the program.

Exercise 2.10: Compute a mathematical sum

The following code is supposed to compute the sum $s = \sum_{k=1}^{M} \frac{1}{k}$:

```
s = 0;  k = 1;  M = 100
while k < M:
    s += 1/k
print s
```

This program does not work correctly. What are the three errors? (If you try to run the program, nothing will happen on the screen. Type `Ctrl+c`, i.e., hold down the Control (`Ctrl`) key and then type the `c` key, to stop the program.) Write a correct program.

Hint. There are two basic ways to find errors in a program:

1. read the program carefully and think about the consequences of each statement,
2. print out intermediate results and compare with hand calculations.

First, try method 1 and find as many errors as you can. Thereafter, try method 2 for $M = 3$ and compare the evolution of `s` with your own hand calculations.
Filename: `sum_while.py`.

Exercise 2.11: Replace a while loop by a for loop

Rewrite the corrected version of the program in Exercise 2.10 using a `for` loop over `k` values instead of a `while` loop. Filename: `sum_for.py`.

Exercise 2.12: 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 2.10, 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.py`.

Exercise 2.13: 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 Section 2.6.3.
Filename: `inverse_sine.py`.

Exercise 2.14: Index a nested lists

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:

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

```
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.py`.

Exercise 2.15: 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.py`.

Exercise 2.16: Store data in a nested list

a) Compute two lists of t and y values as explained in Exercise 2.8. 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 Section 2.4). Loop over the `ty2` list and write out the t and y values with two decimals each.
Filename: `ball_table3.py`.

Exercise 2.17: 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`).

Exercise 2.18: 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.py`.

Exercise 2.19: 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.py`.

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 2.20: Compare two real numbers with a tolerance

Run the following program:

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

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.py`.

Exercise 2.21: 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.py`.

Exercise 2.22: 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 2.10 for how to stop a hanging program.) Filename: `indentation.py`.

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.

Exercise 2.23: 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 2.24: 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.

Functions and branching 3

This chapter introduces two fundamental and extremely useful concepts in programming: user-defined functions and branching of program flow, the latter often referred to as `if` tests. The programs associated with the chapter are found in the folder `src/funcif`[1].

3.1 Functions

In a computer language like Python, the term *function* means more than just a mathematical function. A function is a collection of statements that you can execute wherever and whenever you want in the program. You may send variables to the function to influence what is getting computed by statements in the function, and the function may return new objects back to you.

In particular, functions help avoid duplicating code snippets by putting all similar snippets in a common place. This strategy saves typing and makes it easier to change the program later. Functions are also often used to just split a long program into smaller, more manageable pieces, so the program and your own thinking about it become clearer. Python comes with lots of pre-defined functions (`math.sqrt`, `range`, and `len` are examples we have met so far). This section explains how you can define your own functions.

3.1.1 Mathematical functions as Python functions

Let us start with making a Python function that evaluates a mathematical function, more precisely the function $F(C)$ for converting Celsius degrees C to the corresponding Fahrenheit degrees F:

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

$$F(C) = \frac{9}{5}C + 32\,.$$

The corresponding Python function must take C as argument and return the value $F(C)$. The code for this looks like

```
def F(C):
    return (9.0/5)*C + 32
```

All Python functions begin with `def`, followed by the function name, and then inside parentheses a comma-separated list of *function arguments*. Here we have only one argument `C`. This argument acts as a standard variable inside the function. The statements to be performed inside the function must be indented. At the end of a function it is common to *return* a value, that is, send a value "out of the function". This value is normally associated with the name of the function, as in the present case where the returned value is the result of the mathematical function $F(C)$.

The `def` line with the function name and arguments is often referred to as the *function header*, while the indented statements constitute the *function body*.

To use a function, we must *call* (or *invoke*) it. Because the function returns a value, we need to store this value in a variable or make use of it in other ways. Here are some calls to `F`:

```
temp1 = F(15.5)
a = 10
temp2 = F(a)
print F(a+1)
sum_temp = F(10) + F(20)
```

The returned object from `F(C)` is in our case a `float` object. The call `F(C)` can therefore be placed anywhere in a code where a `float` object would be valid. The `print` statement above is one example.

As another example, say we have a list `Cdegrees` of Celsius degrees and we want to compute a list of the corresponding Fahrenheit degrees using the `F` function above in a list comprehension:

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

Yet another example may involve a slight variation of our `F(C)` function, where a formatted string instead of a real number is returned:

```
>>> def F2(C):
...     F_value = (9.0/5)*C + 32
...     return '%.1f degrees Celsius corresponds to '\
...            '%.1f degrees Fahrenheit' % (C, F_value)
...
>>> s1 = F2(21)
>>> s1
'21.0 degrees Celsius corresponds to 69.8 degrees Fahrenheit'
```

The assignment to `F_value` demonstrates that we can create variables inside a function as needed.

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 Section 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:

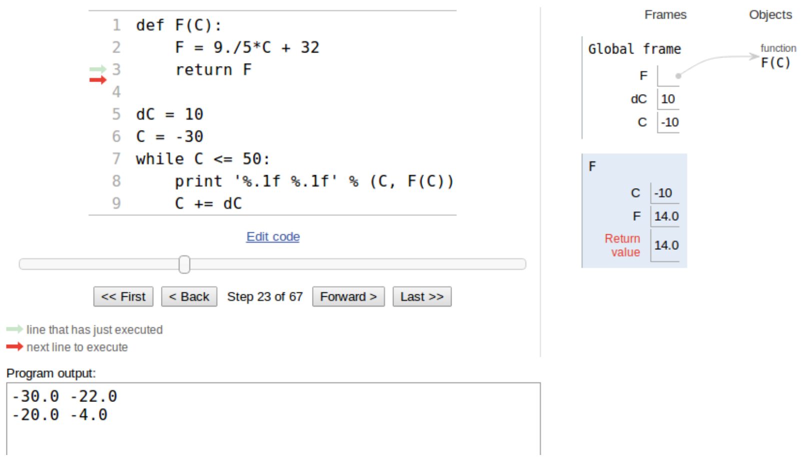
```
def F(C):
    F = 9./5*C + 32
    return F

dC = 10
C = -30
while C <= 50:
    print '%5.1f %.51f' % (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 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 how the flow of statements progresses 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.

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



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

dC = 10
C = -30
while C <= 50:
    print '%.1f %.1f' % (C, F(C))
    C += dC
```

Program output:

```
-30.0 -22.0
-20.0 -4.0
```

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

3.1.3 Local and global variables

Local variables are invisible outside functions. Let us reconsider the `F2(C)` function from Section 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)
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 Section 3.1.2, Figure 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 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 Section 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:

```
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:

```
>>> def yfunc(t):
...     g = 9.81
...     return v0*t - 0.5*g*t**2
...
>>> yfunc(0.6)
...
NameError: global name 'v0' is not defined
```

The remedy is to define `v0` as a global variable prior to calling `yfunc`:

```
>>> v0 = 5
>>> yfunc(0.6)
1.2342
```

The rationale for having `yfunc` as a function of `t` only becomes evident in Section 3.1.12.

3.1.6 Beyond mathematical functions

So far our Python functions have typically computed some mathematical function, but the usefulness of Python functions goes far beyond mathematical functions. Any set of statements that we want to repeatedly execute under potentially slightly different circumstances is a candidate for a Python function. Say we want to make a list of numbers starting from some value and stopping at another value, with increments of a given size. With corresponding variables `start=2`, `stop=8`, and `inc=2`, we should produce the numbers 2, 4, 6, and 8. Let us write a function doing the task, together with a couple of statements that demonstrate how we call the function:

```
def makelist(start, stop, inc):
    value = start
    result = []
    while value <= stop:
        result.append(value)
        value = value + inc
    return result

mylist = makelist(0, 100, 0.2)
print mylist  # will print 0, 0.2, 0.4, 0.6, ... 99.8, 100
```

Remark 1. The `makelist` function has three arguments: `start`, `stop`, and `inc`, which become local variables in the function. Also `value` and `result` are local variables. In the surrounding program we define only one variable, `mylist`, and this is then a global variable.

Remark 2. You might think that `range(start, stop, inc)` makes the `makelist` function redundant, but `range` can only generate integers, while `makelist` can generate real numbers too, and more, as demonstrated in Exercise 3.38.

3.1.7 Multiple return values

Python functions may return more than one value. Suppose we are interested in evaluating both $y(t)$ and $y'(t)$:

$$
\begin{aligned}
y(t) &= v_0 t - \frac{1}{2} g t^2,\\
y'(t) &= v_0 - g t\,.
\end{aligned}
$$

To return both y and y' we simply separate their corresponding variables by a comma in the `return` statement:

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

Calling this latter `yfunc` function makes a need for two values on the left-hand side of the assignment operator because the function returns two values:

```
position, velocity = yfunc(0.6, 3)
```

Here is an application of the `yfunc` function for producing a nicely formatted table of t, $y(t)$, and $y'(t)$ values:

```
t_values = [0.05*i for i in range(10)]
for t in t_values:
    position, velocity = yfunc(t, v0=5)
    print 't=%-10g position=%-10g velocity=%-10g' % \
          (t, position, velocity)
```

The format `%-10g` prints a real number as compactly as possible (decimal or scientific notation) in a field of width 10 characters. The minus sign (-) after the percentage sign implies that the number is *left-adjusted* in this field, a feature that is important for creating nice-looking columns in the output:

```
t=0          position=0          velocity=5
t=0.05       position=0.237737   velocity=4.5095
t=0.1        position=0.45095    velocity=4.019
t=0.15       position=0.639638   velocity=3.5285
t=0.2        position=0.8038     velocity=3.038
t=0.25       position=0.943437   velocity=2.5475
t=0.3        position=1.05855    velocity=2.057
t=0.35       position=1.14914    velocity=1.5665
t=0.4        position=1.2152     velocity=1.076
t=0.45       position=1.25674    velocity=0.5855
```

When a function returns multiple values, separated by a comma in the `return` statement, a tuple (Section 2.5) is actually returned. We can demonstrate that fact by the following session:

```
>>> 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 Section 2.2.1.

3.1.8 Computing sums

Our next example concerns a Python function for calculating the sum

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

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 Section 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
```

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.

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:
   ...
```

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:

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

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 Section 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:

```
def table2(x):
    from math import log
    for k in range(4, 14, 2):
        epsilon = 10**(-k)
        approx, n = L3(x, epsilon=epsilon)
        exact = log(1+x)
        exact_error = exact - approx
```

The output from calling `table2(10)` becomes

```
epsilon: 1e-04, exact error: 8.18e-04, n=55
epsilon: 1e-06, exact error: 9.02e-06, n=97
epsilon: 1e-08, exact error: 8.70e-08, n=142
epsilon: 1e-10, exact error: 9.20e-10, n=187
epsilon: 1e-12, exact error: 9.31e-12, n=233
```

We see that the `epsilon` estimate is almost 10 times smaller than the exact error, regardless of the size of `epsilon`. Since `epsilon` follows the exact error quite well over many orders of magnitude, we may view `epsilon` as a useful indication of the size of the error.

3.1.11 Doc strings

There is a convention in Python to insert a documentation string right after the `def` line of the function definition. The documentation string, known as a *doc string*, should contain a short description of the purpose of the function and explain what the different arguments and return values are. Interactive sessions from a Python shell are also common to illustrate how the code is used. Doc strings are usually enclosed in triple double quotes `"""`, which allow the string to span several lines.

Here are two examples on short and long doc strings:

```
def C2F(C):
    """Convert Celsius degrees (C) to Fahrenheit."""
    return (9.0/5)*C + 32

def line(x0, y0, x1, y1):
    """
    Compute the coefficients a and b in the mathematical
    expression for a straight line y = a*x + b that goes
    through two points (x0, y0) and (x1, y1).

    x0, y0: a point on the line (floats).
    x1, y1: another point on the line (floats).
    return: coefficients a, b (floats) for the line (y=a*x+b).
    """
    a = (y1 - y0)/float(x1 - x0)
    b = y0 - a*x0
    return a, b
```

Note that the doc string must appear before any statement in the function body.

There are several Python tools that can automatically extract doc strings from the source code and produce various types of documentation. The leading tools is Sphinx[3], see also [14, Appendix B.2].

The doc string can be accessed in a code as `funcname.__doc__`, where `funcname` is the name of the function, e.g.,

[3] `http://sphinx-doc.org/invocation.html#invocation-apidoc`

```
print line.__doc__
```

which prints out the documentation of the `line` function above:

```
Compute the coefficients a and b in the mathematical
expression for a straight line y = a*x + b that goes
through two points (x0, y0) and (x1, y1).

x0, y0: a point on the line (float objects).
x1, y1: another point on the line (float objects).
return: coefficients a, b for the line (y=a*x+b).
```

If the function `line` is in a file `funcs.py`, we may also run `pydoc funcs.line` in a terminal window to look the documentation of the `line` function in terms of the function signature and the doc string.

Doc strings often contain interactive sessions, copied from a Python shell, to illustrate how the function is used. We can add such a session to the doc string in the `line` function:

```
def line(x0, y0, x1, y1):
    """
    Compute the coefficients a and b in the mathematical
    expression for a straight line y = a*x + b that goes
    through two points (x0,y0) and (x1,y1).

    x0, y0: a point on the line (float).
    x1, y1: another point on the line (float).
    return: coefficients a, b (floats) for the line (y=a*x+b).

    Example:
    >>> a, b = line(1, -1, 4, 3)
    >>> a
    1.3333333333333333
    >>> b
    -2.333333333333333
    """
    a = (y1 - y0)/float(x1 - x0)
    b = y0 - a*x0
    return a, b
```

A particularly nice feature is that all such interactive sessions in doc strings can be automatically run, and new results are compared to the results found in the doc strings. This makes it possible to use interactive sessions in doc strings both for exemplifying how the code is used and for testing that the code works.

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 (Sections 4.10.2 and A.1.10)
- numerical differentiation: compute $f'(x)$ approximately (Sections B.2 and 7.3.2)
- numerical integration: compute $\int_a^b f(x)dx$ approximately (Sections 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:

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

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

```
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, round-off 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 file `high_precision.py` solves the current problem using arithmetics based on the `decimal` module. With 25 digits in `x` and `h` inside the `diff2nd` function, we get accurate results for $h \leq 10^{-13}$. However, for most practical applications of (3.3), a moderately small h, say $10^{-3} \leq h \leq 10^{-4}$, gives sufficient accuracy and then round-off 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 Section 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 Section 3.1.2.

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 Section 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

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

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

```
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.

With the keyword `elif`, short for *else if*, we can have several mutually exclusive `if` tests, which allows for multiple branching of the program flow:

```
if condition1:
    <block of statements>
elif condition2:
    <block of statements>
elif condition3:
    <block of statements>
else:
    <block of statements>
<next statement>
```

The last `else` part can be skipped if it is not needed. To illustrate multiple branching we will implement a *hat function*, which is widely used in advanced computer simulations in science and industry. One example of a hat function is

$$N(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \leq x < 1 \\ 2 - x, & 1 \leq x < 2 \\ 0, & x \geq 2 \end{cases} \tag{3.5}$$

The solid line in Figure 5.9 in Section 5.4.1 illustrates the shape of this function and why it is called a hat function. The Python implementation associated with (3.5) needs multiple `if` branches:

```
def N(x):
    if x < 0:
        return 0.0
    elif 0 <= x < 1:
        return x
    elif 1 <= x < 2:
        return 2 - x
    elif x >= 2:
        return 0.0
```

This code corresponds directly to the mathematical specification, which is a sound strategy that help reduce the amount of errors in programs. We could mention that there is another way of constructing the `if` test that results in shorter code:

```
def N(x):
    if 0 <= x < 1:
        return x
    elif 1 <= x < 2:
        return 2 - x
    else:
        return 0
```

As a part of learning to program, understanding this latter sample code is important, but we recommend the former solution because of its direct similarity with the mathematical definition of the function.

A popular programming rule is to avoid multiple `return` statements in a function - there should only be one `return` at the end. We can do that in the `N` function by introducing a local variable, assigning values to this variable in the blocks and returning the variable at the end. However, we do not think an extra variable and an extra line make a great improvement in such a short function. Nevertheless, in long and complicated functions the rule can be helpful.

3.2.2 Inline if tests

A variable is often assigned a value that depends on a boolean expression. This can be coded using a common `if-else` test:

```
if condition:
    a = value1
else:
    a = value2
```

Because this construction is often needed, Python provides a one-line syntax for the four lines above:

```
a = (value1 if condition else value2)
```

The parentheses are not required, but recommended style. One example is

```
def f(x):
    return (sin(x) if 0 <= x <= 2*pi else 0)
```

Since the inline `if` test is an expression with a value, it can be used in lambda functions:

```
f = lambda x: sin(x) if 0 <= x <= 2*pi else 0
```

The traditional `if-else` construction with indented blocks cannot be used inside lambda functions because it is not just an expression (lambda functions cannot have statements inside them, only a single expression).

3.3 Mixing loops, branching, and functions in bioinformatics examples

Life is definitely digital. The genetic code of all living organisms are represented by a long sequence of simple molecules called nucleotides, or bases, which makes up the Deoxyribonucleic acid, better known as DNA. There are only four such nucleotides, and the entire genetic code of a human can be seen as a simple, though 3 billion long, string of the letters A, C, G, and T. Analyzing DNA data to gain increased biological understanding is much about searching in long strings for certain string patterns involving the letters A, C, G, and T. This is an integral part of *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 Sections 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.

[4] `http://biopython.org`

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
```

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
```

Correct indentation is here crucial: a typical error is to fail indenting the `j += 1` line correctly.

Summing a boolean list. The idea now is to create a list `m` where `m[i]` is `True` if `dna[i]` equals the letter we search for (`base`). The number of `True` values in `m` is then the number of `base` letters in `dna`. We can use the `sum` function to find this number because doing arithmetics with boolean lists automatically interprets `True` as `1` and `False` as `0`. That is, `sum(m)` returns the number of `True` elements in `m`. A possible function doing this is

```
def count_v5(dna, base):
    m = []   # matches for base in dna: m[i]=True if dna[i]==base
    for c in dna:
        if c == base:
            m.append(True)
        else:
            m.append(False)
    return sum(m)
```

Inline if test. Shorter, more compact code is often a goal if the compactness enhances readability. The four-line `if` test in the previous function can be condensed to one line using the inline `if` construction: `if condition value1 else value2`.

```
def count_v6(dna, base):
    m = []   # matches for base in dna: m[i]=True if dna[i]==base
    for c in dna:
        m.append(True if c == base else False)
    return sum(m)
```

Using boolean values directly. The inline `if` test is in fact redundant in the previous function because the value of the condition `c == base` can be used directly: it has the value `True` or `False`. This saves some typing and adds clarity, at least to Python programmers with some experience:

```
def count_v7(dna, base):
    m = []   # matches for base in dna: m[i]=True if dna[i]==base
    for c in dna:
        m.append(c == base)
    return sum(m)
```

List comprehensions. Building a list with the aid of a `for` loop can often be condensed to a single line by using list comprehensions: `[expr for e in sequence]`, where `expr` is some expression normally involving the iteration variable `e`. In our last example, we can introduce a list comprehension

```
def count_v8(dna, base):
    m = [c == base for c in dna]
    return sum(m)
```

Here it is tempting to get rid of the `m` variable and reduce the function body to a single line:

```
def count_v9(dna, base):
    return sum([c == base for c in dna])
```

Using a sum iterator. The DNA string is usually huge - 3 billion letters for the human species. Making a boolean array with `True` and `False` values therefore increases the memory usage by a factor of two in our sample functions `count_v5` to `count_v9`. Summing without actually storing an extra list is desirable. Fortunately, `sum([x for x in s])` can be replaced by `sum(x for x in s)`, where the latter sums the elements in `s` as `x` visits the elements of `s` one by one. Removing the brackets therefore avoids first making a list before applying `sum` to that list. This is a minor modification of the `count_v9` function:

```
def count_v10(dna, base):
    return sum(c == base for c in dna)
```

Below we shall measure the impact of the various program constructs on the CPU time.

Extracting indices. Instead of making a boolean list with elements expressing whether a letter matches the given `base` or not, we may collect all the indices of the matches. This can be done by adding an `if` test to the list comprehension:

```
def count_v11(dna, base):
    return len([i for i in range(len(dna)) if dna[i] == base])
```

The Online Python Tutor[5] is really helpful to reach an understanding of this compact code. Alternatively, you may play with the constructions in an interactive Python shell:

```
>>> dna = 'AATGCTTA'
>>> base = 'A'
>>> indices = [i for i in range(len(dna)) if dna[i] == base]
>>> indices
[0, 1, 7]
>>> print dna[0], dna[1], dna[7]  # check
A A A
```

Observe that the element `i` in the list comprehension is only made for those `i` where `dna[i] == base`.

Using Python's library. Very often when you set out to do a task in Python, there is already functionality for the task in the object itself, in the Python libraries, or in third-party libraries found on the Internet. Counting how many times a letter (or substring) `base` appears in a string `dna` is obviously a very common task so Python supports it by the syntax `dna.count(base)`:

```
def count_v12(dna, base):
    return dna.count(base)

def compare_efficiency():
```

3.3.2 Efficiency assessment

Now we have 11 different versions of how to count the occurrences of a letter in a string. Which one of these implementations is the fastest? To answer the question we need some test data, which should be a huge string `dna`.

Generating random DNA strings. The simplest way of generating a long string is to repeat a character a large number of times:

```
N = 1000000
dna = 'A'*N
```

The resulting string is just `'AAA...A`, of length `N`, which is fine for testing the efficiency of Python functions. Nevertheless, it is more exciting to work with a DNA string with letters from the whole alphabet A, C, G,

[5] `http://www.pythontutor.com/`

and T. To make a DNA string with a random composition of the letters we can first make a list of random letters and then join all those letters to a string:

```
import random
alphabet = list('ATGC')
dna = [random.choice(alphabet) for i in range(N)]
dna = ''.join(dna)  # join the list elements to a string
```

The `random.choice(x)` function selects an element in the list `x` at random.

Note that `N` is very often a large number. In Python version 2.x, `range(N)` generates a list of `N` integers. We can avoid the list by using `xrange` which generates an integer at a time and not the whole list. In Python version 3.x, the `range` function is actually the `xrange` function in version 2.x. Using `xrange`, combining the statements, and wrapping the construction of a random DNA string in a function, gives

```
import random

def generate_string(N, alphabet='ACGT'):
    return ''.join([random.choice(alphabet) for i in xrange(N)])

dna = generate_string(600000)
```

The call `generate_string(10)` may generate something like `AATGGCAGAA`.

Measuring CPU time. Our next goal is to see how much time the various `count_v*` functions spend on counting letters in a huge string, which is to be generated as shown above. Measuring the time spent in a program can be done by the `time` module:

```
import time
...
t0 = time.clock()
# do stuff
t1 = time.clock()
cpu_time = t1 - t0
```

The `time.clock()` function returns the CPU time spent in the program since its start. If the interest is in the total time, also including reading and writing files, `time.time()` is the appropriate function to call.

Running through all our functions made so far and recording timings can be done by

```
import time
functions = [count_v1, count_v2, count_v3, count_v4,
             count_v5, count_v6, count_v7, count_v8,
             count_v9, count_v10, count_v11, count_v12]
timings = []  # timings[i] holds CPU time for functions[i]

for function in functions:
    t0 = time.clock()
    function(dna, 'A')
    t1 = time.clock()
    cpu_time = t1 - t0
    timings.append(cpu_time)
```

In Python, functions are ordinary objects so making a list of functions is no more special than making a list of strings or numbers.

We can now iterate over `timings` and `functions` simultaneously via `zip` to make a nice printout of the results:

```
for cpu_time, function in zip(timings, functions):
    print '{f:<9s}: {cpu:.2f} s'.format(
        f=function.func_name, cpu=cpu_time)
```

Timings on a MacBook Air 11 running Ubuntu show that the functions using `list.append` require almost the double of the time of the functions that work with list comprehensions. Even faster is the simple iteration over the string. However, the built-in count functionality of strings (`dna.count(base)`) runs over 30 times faster than the best of our handwritten Python functions! The reason is that the `for` loop needed to count in `dna.count(base)` is actually implemented in C and runs very much faster than loops in Python.

A clear lesson learned is: google around before you start out to implement what seems to be a quite common task. Others have probably already done it for you, and most likely is their solution much better than what you can (easily) come up with.

3.3.3 Verifying the implementations

We end this section with showing how to make tests that verify our 12 counting functions. To this end, we make a new function that first computes a certainly correct answer to a counting problem and then calls all the `count_*` functions, stored in the list `functions`, to check that each call has the correct result:

```
def test_count_all():
    dna = 'ATTTGCGGTCCAAA'
    exact = dna.count('A')
    for f in functions:
        if f(dna, 'A') != exact:
            print f.__name__, 'failed'
```

Here, we believe in `dna.count('A')` as the correct answer.

We might take this test function one step further and adopt the conventions in the pytest[6] and nose[7] testing frameworks for Python code. (See Section H.6 for more information about pytest and nose.)

These conventions say that the test function should

[6] `http://pytest.org`
[7] `https://nose.readthedocs.org`

- have a name starting with `test_`;
- have no arguments;
- let a boolean variable, say `success`, be `True` if a test passes and be `False` if the test fails;
- create a message about what failed, stored in some string, say `msg`;
- use the construction `assert success, msg`, which will abort the program and write out the error message `msg` if `success` is `False`.

The pytest and nose test frameworks can search for all Python files in a folder tree, run all `test_*()` functions, and report how many of the tests that failed, if we adopt the conventions above. Our revised test function becomes

```
def test_count_all():
    dna = 'ATTTGCGGTCCAAA'
    exact = dna.count('A')
    for f in functions:
        success = f(dna, 'A') == exact
        msg = '%s failed' % f.__name__
        assert success, msg
```

It is worth notifying that the name of a function `f`, as a string object, is given by `f.__name__`, and we make use of this information to construct an informative message in case a test fails.

It is a good habit to write such test functions since the execution of all tests in all files can be fully automated. Every time you to a change in some file you can with minimum effort rerun all tests.

The entire suite of functions presented above, including the timings and tests, can be found in the file `count.py`.

3.4 Summary

3.4.1 Chapter topics

User-defined functions. Functions are useful (i) when a set of commands are to be executed several times, or (ii) to partition the program into smaller pieces to gain better overview. Function arguments are local variables inside the function whose values are set when calling the function. Remember that when you write the function, the values of the arguments are not known. Here is an example of a function for polynomials of 2nd degree:

```
# function definition:
def quadratic_polynomial(x, a, b, c)
    value = a*x*x + b*x + c
    derivative = 2*a*x + b
    return value, derivative

# function call:
x = 1
```

```
p, dp = quadratic_polynomial(x, 2, 0.5, 1)
p, dp = quadratic_polynomial(x=x, a=-4, b=0.5, c=0)
```

The sequence of the arguments is important, unless all arguments are given as `name=value`.

Functions may have no arguments and/or no return value(s):

```
def print_date():
    """Print the current date in the format 'Jan 07, 2007'."""
    import time
    print time.strftime("%b %d, %Y")

# call:
print_date()
```

A common error is to forget the parentheses: `print_date` is the function object itself, while `print_date()` is a call to the function.

Keyword arguments. Function arguments with default values are called keyword arguments, and they help to document the meaning of arguments in function calls. They also make it possible to specify just a subset of the arguments in function calls.

```
from math import exp, sin, pi

def f(x, A=1, a=1, w=pi):
    return A*exp(-a*x)*sin(w*x)

f1 = f(0)
x2 = 0.1
f2 = f(x2, w=2*pi)
f3 = f(x2, w=4*pi, A=10, a=0.1)
f4 = f(w=4*pi, A=10, a=0.1, x=x2)
```

The sequence of the keyword arguments can be arbitrary, and the keyword arguments that are not listed in the call get their default values according to the function definition. The non-keyword arguments are called positional arguments, which is `x` in this example. Positional arguments must be listed before the keyword arguments. However, also a positional argument can appear as `name=value` in the call (see the last line above), and this syntax allows any positional argument to be listed anywhere in the call.

If tests. The `if-elif-else` tests are used to *branch* the flow of statements. That is, different sets of statements are executed depending on whether some conditions are `True` or `False`.

```
def f(x):
    if x < 0:
        value = -1
    elif x >= 0 and x <= 1:
        value = x
    else:
        value = 1
    return value
```

Inline if tests. Assigning a variable one value if a condition is `True` and another value if `False`, is compactly done with an inline `if` test:

```
sign = -1 if a < 0 else 1
```

Terminology. The important computer science terms in this chapter are

- function
- method
- return statement
- positional arguments
- keyword arguments
- local and global variables
- doc strings
- `if` tests with `if`, `elif`, and `else` (branching)
- the `None` object
- test functions (for verification)

3.4.2 Example: Numerical integration

Problem. An integral

$$\int_a^b f(x)dx$$

can be approximated by the so-called Simpson's rule:

$$\frac{b-a}{3n}\left(f(a)+f(b)+4\sum_{i=1}^{n/2} f(a+(2i-1)h)+2\sum_{i=1}^{n/2-1} f(a+2ih)\right). \tag{3.6}$$

Here, $h=(b-a)/n$ and n must be an even integer. The problem is to make a function `Simpson(f, a, b, n=500)` that returns the right-hand side formula of (3.6). To verify the implementation, one can make use of the fact that Simpson's rule is *exact* for all polynomials $f(x)$ of degree ≤ 2. Apply the `Simpson` function to the integral $\frac{3}{2}\int_0^\pi \sin^3 x dx$, which has exact value 2, and investigate how the approximation error varies with n.

Solution. The evaluation of the formula (3.6) in a program is straightforward if we know how to implement summation ($\sum$) and how to call f. A Python recipe for calculating sums is given in Section 3.1.8. Basically, $\sum_{i=M}^{N} q(i)$, for some expression $q(i)$ involving i, is coded with the aid of a `for` loop over i and an accumulation variable `s` for building up the sum, one term at a time:

```
s = 0
for i in range(M, N):
    s += q(i)
```

The `Simpson` function can then be coded as

```
def Simpson(f, a, b, n=500):
    h = (b - a)/float(n)
    sum1 = 0
    for i in range(1, n/2 + 1):
        sum1 += f(a + (2*i-1)*h)

    sum2 = 0
    for i in range(1, n/2):
        sum2 += f(a + 2*i*h)

    integral = (b-a)/(3*n)*(f(a) + f(b) + 4*sum1 + 2*sum2)
    return integral
```

Note that `Simpson` can integrate any Python function `f` of one variable. Specifically, we can implement

$$h(x) = \frac{3}{2}\sin^3 x dx$$

in a Python function

```
def h(x):
    return (3./2)*sin(x)**3
```

and call `Simpson` to compute $\int_0^\pi h(x)dx$ for various choices of n, as requested:

```
from math import sin, pi

def application():
    print 'Integral of 1.5*sin^3 from 0 to pi:'
    for n in 2, 6, 12, 100, 500:
        approx = Simpson(h, 0, pi, n)
        print 'n=%3d, approx=%18.15f, error=%9.2E' % \
              (n, approx, 2-approx)
```

(We have put the statements inside a function, here called `application`, mainly to group them, and not because `application` will be called several times or with different arguments.)

Verification. Calling `application()` leads to the output

```
Integral of 1.5*sin^3 from 0 to pi:
n=  2, approx= 3.141592653589793, error=-1.14E+00
n=  6, approx= 1.989171700583579, error= 1.08E-02
n= 12, approx= 1.999489233010781, error= 5.11E-04
n=100, approx= 1.999999902476350, error= 9.75E-08
n=500, approx= 1.999999999844138, error= 1.56E-10
```

We clearly see that the approximation improves as n increases. However, every computation will give an answer that deviates from the exact value 2. We cannot from this test alone know if the errors above are those implied by the approximation only, or if there are additional programming mistakes.

A much better way of verifying the implementation is therefore to look for test cases where the numerical approximation formula is *exact*, such that we know exactly what the result of the function should be. Since it is stated that the formula is exact for polynomials up to second degree, we just test the `Simpson` function on an “arbitrary” parabola, say

$$\int_{3/2}^{2} \left(3x^2 - 7x + 2.5\right) dx\,.$$

This integral equals $G(2) - G(3/2)$, where $G(x) = x^3 - 3.5x^2 + 2.5x$. A possible implementation becomes

```
def g(x):
    return 3*x**2 - 7*x + 2.5

def G(x):
    return x**3 - 3.5*x**2 + 2.5*x

def test_Simpson():
    a = 1.5
    b = 2.0
    n = 8
    exact = G(b) - G(a)
    approx = Simpson(g, a, b, n)
    success = abs(exact - approx) < 1E-14
    if not success:
        print 'Error: cannot integrate a quadratic function exactly'
```

Observe that we avoid testing `exact == approx` because there may be small round-off errors in these `float` objects so that the exact `==` test fails. Testing that the two variables are very close (distance less than 10^{-14}) is the rule of thumb for comparing `float` objects.

The `g` and `G` functions are only of interest inside the `test_Simpson` function. Many think the code becomes easier to read and understand if `g` and `G` are moved inside `test_Simpson`, which is indeed possible in Python:

```
def test_Simpson():
    def g(x):
        # test function that Simpson's rule will integrat exactly
        return 3*x**2 - 7*x + 2.5

    def G(x):
        # integral of g(x)
        return x**3 - 3.5*x**2 + 2.5*x

    a = 1.5
    b = 2.0
    n = 8
    exact = G(b) - G(a)
    approx = Simpson(g, a, b, n)
    success = abs(exact - approx) < 1E-14
    if not success:
        print 'Error: cannot integrate a quadratic function exactly'
```

We shall make it a habit to write functions like `test_Simpson` for verifying implementations. As was mentioned in Section 3.3.3, it can be wise to write our test function according to the conventions needed for applying the pytest and nose testing frameworks (Section H.6). Our pytest/nose-compatible test function then looks as follows:

```
def test_Simpson():
    """Check that 2nd-degree polynomials are integrated exactly."""
    a = 1.5
    b = 2.0
    n = 8
    g = lambda x: 3*x**2 - 7*x + 2.5       # test integrand
    G = lambda x: x**3 - 3.5*x**2 + 2.5*x  # integral of g
    exact = G(b) - G(a)
    approx = Simpson(g, a, b, n)
    success = abs(exact - approx) < 1E-14  # never use == for floats!
    msg = 'Cannot integrate a quadratic function exactly'
    assert success, msg
```

Here we have also made the test function more compact by utilizing lambda functions for `g` and `G` (see Section 3.1.14).

Checking the validity of function arguments. Another improvement is to increase the robustness of the function. That is, to check that the input data, i.e., the arguments, are acceptable. Here we may test if $b > a$ and if n is an even integer. For the latter test, we make use of the *mod* function: mod(n, d) gives the remainder when n is divided by d (both n and d are integers). Mathematically, if p is the largest integer such that $pd \leq n$, then mod(n, d) is $n - pd$. For example, mod(3, 2) is 1, mod(3, 1) is 0, mod (3, 3) is 0, and mod(18, 8) is 2. The point is that n divided by d is an integer when mod(n, d) is zero. In Python, the percentage sign is used for the mod function:

```
>>> 18 % 8
2
```

To test if `n` is an odd integer, we see if it can be divided by 2 and yield an integer without any reminder: `n % 2 == 0`.

The improved `Simpson` function with validity tests on the provided arguments, as well as a doc string (Section 3.1.11), can look like this:

```
def Simpson(f, a, b, n=500):
    """
    Return the approximation of the integral of f
    from a to b using Simpson's rule with n intervals.
    """
    if a > b:
        print 'Error: a=%g > b=%g' % (a, b)
        return None

    # check that n is even:
    if n % 2 != 0:
        print 'Error: n=%d is not an even integer!' % n
        n = n+1  # make n even

    h = (b - a)/float(n)
    sum1 = 0
    for i in range(1, n/2 + 1):
        sum1 += f(a + (2*i-1)*h)

    sum2 = 0
    for i in range(1, n/2):
        sum2 += f(a + 2*i*h)

    integral = (b-a)/(3*n)*(f(a) + f(b) + 4*sum1 + 2*sum2)
    return integral
```

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

A very good exercise is to simulate the program flow by hand, starting with the call to the `application` function. The Online Python Tutor or a debugger (see Section F.1) are convenient tools for controlling that your thinking is correct.

3.5 Exercises

Exercise 3.1: Write a Fahrenheit-Celsius conversion function

The formula for converting Fahrenheit degrees to Celsius reads

$$C = \frac{5}{9}(F - 32)\,. \qquad (3.7)$$

Write a function `C(F)` that implements this formula. To verify the implementation, you can use `F(C)` from Section 3.1.1 and test that `C(F(c)` equals `c`.

Hint. Do not test `C(F(c)) == c` exactly, but use a tolerance for the difference.
Filename: `f2c.py`.

Exercise 3.2: Evaluate a sum and write a test function

a) Write a Python function `sum_1k(M)` that returns the sum $s = \sum_{k=1}^{M} \frac{1}{k}$.

b) Compute s for $M = 3$ by hand and write another function `test_sum_1k()` that calls `sum_1k(3)` and checks that the answer is correct.

Hint. We recommend that `test_sum_1k` follows the conventions of the pytest and nose testing frameworks as explained in Sections 3.3.3 and 3.4.2 (see also Section H.6). It means setting a boolean variable `success` to `True` if the test passes, otherwise `False`. The next step is to do `assert success`, which will abort the program with an error message if `success` is `False` and the test failed. To provide an informative error message, you can add your own message string `msg`: `assert success, msg`.
Filename: `sum_func.py`.

Exercise 3.3: Write a function for solving $ax^2 + bx + c = 0$

a) Given a quadratic equation $ax^2+bx+c = 0$, write a function `roots(a, b, c)` that returns the two roots of the equation. The returned roots should be `float` objects when the roots are real, otherwise the function returns `complex` objects.

Hint. Use `sqrt` from the `numpy.lib.scimath` library, see Chapter 1.6.3.

b) Construct two test cases with known solutions, one with real roots and the other with complex roots, Implement the two test cases in two test functions `test_roots_float` and `test_roots_complex`, where you call the `roots` function and check the type and value of the returned objects.
Filename: `roots_quadratic.py`.

Exercise 3.4: Implement the sum function

The standard Python function called `sum` takes a list as argument and computes the sum of the elements in the list:

```
>>> sum([1,3,5,-5])
4
>>> sum([[1,2], [4,3], [8,1]])
[1, 2, 4, 3, 8, 1]
>>> sum(['Hello, ', 'World!'])
'Hello, World!'
```

Implement your own version of `sum`. Filename: `mysum.py`.

Exercise 3.5: Compute a polynomial via a product

Given $n+1$ roots $r_0, r_1, \ldots, r_n$ of a polynomial $p(x)$ of degree $n+1$, $p(x)$ can be computed by

$$p(x) = \prod_{i=0}^{n}(x - r_i) = (x - r_0)(x - r_1)\cdots(x - r_{n-1})(x - r_n)\,. \qquad (3.8)$$

Write a function `poly(x, roots)` that takes x and a list `roots` of the roots as arguments and returns $p(x)$. Construct a test case for verifying the implementation. Filename: `polyprod.py`.

Exercise 3.6: Integrate a function by the Trapezoidal rule

a) An approximation to the integral of a function $f(x)$ over an interval $[a, b]$ can be found by first approximating $f(x)$ by the straight line that goes through the end points $(a, f(a))$ and $(b, f(b))$, and then finding the area under the straight line, which is the area of a trapezoid. The resulting formula becomes

$$\int_a^b f(x)dx \approx \frac{b-a}{2}(f(a)+f(b))\,. \tag{3.9}$$

Write a function `trapezint1(f, a, b)` that returns this approximation to the integral. The argument `f` is a Python implementation `f(x)` of the mathematical function $f(x)$.

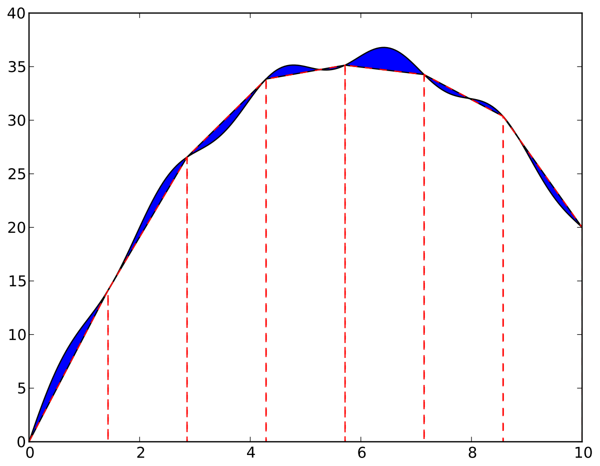
b) Use the approximation (3.9) to compute the following integrals: $\int_0^\pi \cos x\, dx$, $\int_0^\pi \sin x\, dx$, and $\int_0^{\pi/2} \sin x\, dx$, In each case, write out the error, i.e., the difference between the exact integral and the approximation (3.9). Make rough sketches of the trapezoid for each integral in order to understand how the method behaves in the different cases.

c) We can easily improve the formula (3.9) by approximating the area under the function $f(x)$ by two equal-sized trapezoids. Derive a formula for this approximation and implement it in a function `trapezint2(f, a, b)`. Run the examples from b) and see how much better the new formula is. Make sketches of the two trapezoids in each case.

d) A further improvement of the approximate integration method from c) is to divide the area under the $f(x)$ curve into n equal-sized trapezoids. Based on this idea, derive the following formula for approximating the integral:

$$\int_a^b f(x)dx \approx \sum_{0=1}^{n-1} \frac{1}{2}h\left(f(x_i)+f(x_{i+1})\right), \tag{3.10}$$

where h is the width of the trapezoids, $h=(b-a)/n$, and $x_i = a+ih$, $i=0,\ldots,n$, are the coordinates of the sides of the trapezoids. The figure below visualizes the idea of the Trapezoidal rule.



Implement (3.10) in a Python function `trapezint(f, a, b, n)`. Run the examples from b) with $n = 10$.

e) Write a test function `test_trapezint()` for verifying the implementation of the function `trapezint` in d).

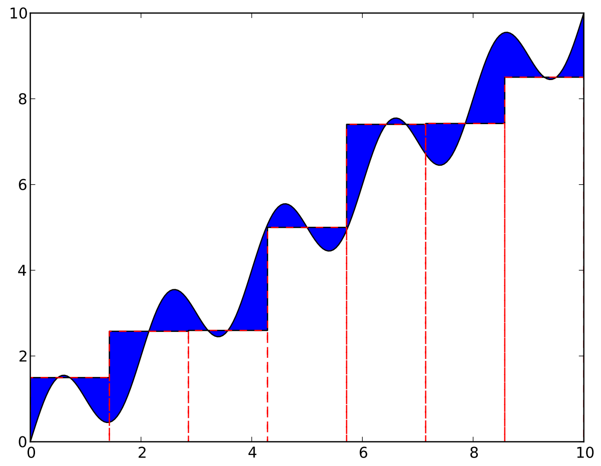
Hint. Obviously, the Trapezoidal method integrates linear functions exactly for any n. Another more surprising result is that the method is also exact for, e.g., $\int_0^{2\pi} \cos x \, dx$ for any n.
Filename: `trapezint.py`.

Remarks. Formula (3.10) is not the most common way of expressing the Trapezoidal integration rule. The reason is that $f(x_{i+1})$ is evaluated twice, first in term i and then as $f(x_i)$ in term $i+1$. The formula can be further developed to avoid unnecessary evaluations of $f(x_{i+1})$, which results in the standard form

$$\int_a^b f(x)dx \approx \frac{1}{2}h(f(a) + f(b)) + h\sum_{i=1}^{n-1} f(x_i)\,. \qquad (3.11)$$

Exercise 3.7: Derive the general Midpoint integration rule

The idea of the Midpoint rule for integration is to divide the area under the curve $f(x)$ into n equal-sized rectangles (instead of trapezoids as in Exercise 3.6). The height of the rectangle is determined by the value of f at the midpoint of the rectangle. The figure below illustrates the idea.



Compute the area of each rectangle, sum them up, and arrive at the formula for the Midpoint rule:

$$\int_a^b f(x)dx \approx h\sum_{i=0}^{n-1} f(a+ih+\frac{1}{2}h), \tag{3.12}$$

where $h = (b-a)/n$ is the width of each rectangle. Implement this formula in a Python function `midpointint(f, a, b, n)` and test the function on the examples listed in Exercise 3.6b. How do the errors in the Midpoint rule compare with those of the Trapezoidal rule for $n = 1$ and $n = 10$? Filename: `midpointint.py`.

Exercise 3.8: Make an adaptive Trapezoidal rule

A problem with the Trapezoidal integration rule (3.10) in Exercise 3.6 is to decide how many trapezoids (n) to use in order to achieve a desired accuracy. Let E be the error in the Trapezoidal method, i.e., the difference between the exact integral and that produced by (3.10). We would like to prescribe a (small) tolerance ϵ and find an n such that $E \leq \epsilon$.

Since the exact value $\int_a^b f(x)dx$ is not available (that is why we use a numerical method!), it is challenging to compute E. Nevertheless, it has been shown by mathematicians that

$$E \leq \frac{1}{12}(b-a)h^2 \max_{x\in[a,b]} |f''(x)| \ . \tag{3.13}$$

The maximum of $|f''(x)|$ can be computed (approximately) by evaluating $f''(x)$ at a large number of points in $[a,b]$, taking the absolute value

$|f''(x)|$, and finding the maximum value of these. The double derivative can be computed by a finite difference formula:

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

With the computed estimate of $\max |f''(x)|$ we can find h from setting the right-hand side in (3.13) equal to the desired tolerance:

$$\frac{1}{12}(b-a)h^2 \max_{x\in[a,b]} |f''(x)| = \epsilon\,.$$

Solving with respect to h gives

$$h = \sqrt{12\epsilon}\left((b-a)\max_{x\in[a,b]} |f''(x)|\right)^{-1/2}\,. \qquad (3.14)$$

With $n = (b-a)/h$ we have the n that corresponds to the desired accuracy ϵ.

a) Make a Python function `adaptive_trapezint(f, a, b, eps=1E-5)` for computing the integral $\int_a^b f(x)dx$ with an error less than or equal to ϵ (`eps`).

Hint. Compute the n corresponding to ϵ as explained above and call `trapezint(f, a, b, n)` from Exercise 3.6.

b) Apply the function to compute the integrals from Exercise 3.6b. Write out the exact error and the estimated n for each case.
Filename: `adaptive_trapezint.py`.

Remarks. A numerical method that applies an expression for the error to adapt the choice of the discretization parameter to a desired error tolerance, is known as an *adaptive* numerical method. The advantage of an adaptive method is that one can control the approximation error, and there is no need for the user to determine an appropriate number of intervals n.

Exercise 3.9: Explain why a program works

Explain how and thereby why the following program works:

```
def add(A, B):
    C = A + B
    return C

A = 3
B = 2
print add(A, B)
```

Exercise 3.10: Simulate a program by hand

Simulate the following program by hand to explain what is printed.

```
def a(x):
    q = 2
    x = 3*x
    return q + x

def b(x):
    global q
    q += x
    return q + x

q = 0
x = 3
print a(x), b(x), a(x), b(x)
```

Hint. If you encounter problems with understanding function calls and local versus global variables, paste the code into the Online Python Tutor[8] and step through the code to get a good explanation of what happens.

Exercise 3.11: Compute the area of an arbitrary triangle

An arbitrary triangle can be described by the coordinates of its three vertices: (x_1, y_1), (x_2, y_2), (x_3, y_3), numbered in a counterclockwise direction. The area of the triangle is given by the formula

$$A = \frac{1}{2}\left|x_2y_3 - x_3y_2 - x_1y_3 + x_3y_1 + x_1y_2 - x_2y_1\right| . \tag{3.15}$$

Write a function `area(vertices)` that returns the area of a triangle whose vertices are specified by the argument `vertices`, which is a nested list of the vertex coordinates. For example, computing the area of the triangle with vertex coordinates $(0, 0)$, $(1, 0)$, and $(0, 2)$ is done by

```
triangle1 = area([[0,0], [1,0], [0,2]])
# or
v1 = (0,0);  v2 = (1,0);  v3 = (0,2)
vertices = [v1, v2, v3]
triangle1 = area(vertices)

print 'Area of triangle is %.2f' % triangle1
```

Test the `area` function on a triangle with known area. Filename: `area_triangle.py`.

Exercise 3.12: Compute the length of a path

Some object is moving along a path in the plane. At $n + 1$ points of time we have recorded the corresponding (x, y) positions of the object:

[8] `http://www.pythontutor.com/visualize.html`

(x_0, y_0), (x_1, y_2), ..., (x_n, y_n). The total length L of the path from (x_0, y_0) to (x_n, y_n) is the sum of all the individual line segments ((x_{i-1}, y_{i-1}) to (x_i, y_i), $i = 1, \ldots, n$):

$$L = \sum_{i=1}^{n} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}\,. \tag{3.16}$$

a) Make a Python function `pathlength(x, y)` for computing L according to the formula. The arguments `x` and `y` hold all the $x_0, \ldots, x_n$ and $y_0, \ldots, y_n$ coordinates, respectively.

b) Write a test function `test_pathlength()` where you check that `pathlength` returns the correct length in a test problem.
Filename: `pathlength.py`.

Exercise 3.13: Approximate π

The value of π equals the circumference of a circle with radius 1/2. Suppose we approximate the circumference by a polygon through $n + 1$ points on the circle. The length of this polygon can be found using the `pathlength` function from Exercise 3.12. Compute $n + 1$ points (x_i, y_i) along a circle with radius 1/2 according to the formulas

$$x_i = \frac{1}{2}\cos(2\pi i/n), \quad y_i = \frac{1}{2}\sin(2\pi i/n), \quad i = 0, \ldots, n\,.$$

Call the `pathlength` function and write out the error in the approximation of π for $n = 2^k$, $k = 2, 3, \ldots, 10$. Filename: `pi_approx.py`.

Exercise 3.14: Write functions

Three functions, `hw1`, `hw2`, and `hw3`, work as follows:

```
>>> print hw1()
Hello, World!
>>> hw2()
Hello, World!
>>> print hw3('Hello, ', 'World!')
Hello, World!
>>> print hw3('Python ', 'function')
Python function
```

Write the three functions. Filename: `hw_func.py`.

Exercise 3.15: Approximate a function by a sum of sines

We consider the piecewise constant function

$$f(t) = \begin{cases} 1, & 0 < t < T/2, \\ 0, & t = T/2, \\ -1, & T/2 < t < T \end{cases} \tag{3.17}$$

Sketch this function on a piece of paper. One can approximate $f(t)$ by the sum

$$S(t;n) = \frac{4}{\pi} \sum_{i=1}^{n} \frac{1}{2i-1} \sin\left(\frac{2(2i-1)\pi t}{T}\right) . \tag{3.18}$$

It can be shown that $S(t;n) \to f(t)$ as $n \to \infty$.

a) Write a Python function `S(t, n, T)` for returning the value of $S(t;n)$.

b) Write a Python function `f(t, T)` for computing $f(t)$.

c) Write out tabular information showing how the error $f(t) - S(t;n)$ varies with n and t for the cases where $n = 1, 3, 5, 10, 30, 100$ and $t = \alpha T$, with $T = 2\pi$, and $\alpha = 0.01, 0.25, 0.49$. Use the table to comment on how the quality of the approximation depends on α and n.
Filename: `sinesum1.py`.

Remarks. A sum of sine and/or cosine functions, as in (3.18), is called a *Fourier series*. Approximating a function by a Fourier series is a very important technique in science and technology. Exercise 5.39 asks for visualization of how well $S(t;n)$ approximates $f(t)$ for some values of n.

Exercise 3.16: Implement a Gaussian function

Make a Python function `gauss(x, m=0, s=1)` for computing the Gaussian function

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

Write out a nicely formatted table of x and $f(x)$ values for n uniformly spaced x values in $[m - 5s, m + 5s]$. (Choose m, s, and n as you like.)
Filename: `gaussian2.py`.

Exercise 3.17: Wrap a formula in a function

Implement the formula (1.9) from Exercise 1.12 in a Python function with three arguments: `egg(M, To=20, Ty=70)`. The parameters ρ, K, c, and T_w can be set as local (constant) variables inside the function. Let t be returned from the function. Compute t for a soft and hard boiled egg, of a small ($M = 47$ g) and large ($M = 67$ g) size, taken from the fridge ($T_o = 4$ C) and from a hot room ($T_o = 25$ C). Filename: `egg_func.py`.

Exercise 3.18: Write a function for numerical differentiation

The formula

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{3.19}$$

can be used to find an approximate derivative of a mathematical function $f(x)$ if h is small.

a) Write a function `diff(f, x, h=1E-5)` that returns the approximation (3.19) of the derivative of a mathematical function represented by a Python function `f(x)`.

b) Write a function `test_diff()` that verifies the implementation of the function `diff`. As test case, one can use the fact that (3.19) is exact for quadratic functions. Follow the conventions of the pytest and nose testing frameworks, as outlined in Exercise 3.2 and Sections 3.3.3, 3.4.2, and H.6.

c) Apply (3.19) to differentiate

- $f(x) = e^x$ at $x = 0$,
- $f(x) = e^{-2x^2}$ at $x = 0$,
- $f(x) = \cos x$ at $x = 2\pi$,
- $f(x) = \ln x$ at $x = 1$.

Use $h = 0.01$. In each case, write out the error, i.e., the difference between the exact derivative and the result of (3.19). Collect these four examples in a function `application()`.
Filename: `centered_diff.py`.

Exercise 3.19: Implement the factorial function

The factorial of n is written as $n!$ and defined as

$$n! = n(n-1)(n-2)\cdots 2\cdot 1, \tag{3.20}$$

with the special cases

$$1! = 1, \quad 0! = 1\,. \tag{3.21}$$

For example, $4! = 4{\cdot}3{\cdot}2{\cdot}1 = 24$, and $2! = 2{\cdot}1 = 2$. Write a Python function `fact(n)` that returns $n!$. (Do not simply call the ready-made function `math.factorial(n)` - that is considered cheating in this context!)

Hint. Return 1 immediately if x is 1 or 0, otherwise use a loop to compute $n!$.
Filename: `fact.py`.

Exercise 3.20: Compute velocity and acceleration from 1D position data

Suppose we have recorded GPS coordinates $x_0, \ldots, x_n$ at times $t_0, \ldots, t_n$ while running or driving along a straight road. We want to compute the velocity v_i and acceleration a_i from these position coordinates. Using finite difference approximations, one can establish the formulas

$$v_i \approx \frac{x_{i+1} - x_{i-1}}{t_{i+1} - t_{i-1})}, \tag{3.22}$$

$$a_i \approx 2(t_{i+1} - t_{i-1})^{-1} \left(\frac{x_{i+1} - x_i}{t_{i+1} - t_i} - \frac{x_i - x_{i-1}}{t_i - t_{i-1}} \right), \tag{3.23}$$

for $i = 1, \ldots, n-1$.

a) Write a Python function `kinematics(x, i, dt=1E-6)` for computing v_i and a_i, given the array `x` of position coordinates $x_0, \ldots, x_n$.

b) Write a Python function `test_kinematics()` for testing the implementation in the case of constant velocity V. Set $t_0 = 0$, $t_1 = 0.5$, $t_2 = 1.5$, and $t_3 = 2.2$, and $x_i = Vt_i$.
Filename: `kinematics1.py`.

Exercise 3.21: Find the max and min values of a function

The maximum and minimum values of a mathematical function $f(x)$ on $[a, b]$ can be found by computing f at a large number (n) of points and selecting the maximum and minimum values at these points. Write a Python function `maxmin(f, a, b, n=1000)` that returns the maximum and minimum value of a function `f(x)`. Also write a test function for verifying the implementation for $f(x) = \cos x$, $x \in [-\pi/2, 2\pi]$.

Hint. The x points where the mathematical function is to be evaluated can be uniformly distributed: $x_i = a + ih$, $i = 0, \ldots, n-1$, $h = (b-a)/(n-1)$. The Python functions `max(y)` and `min(y)` return the maximum and minimum values in the list `y`, respectively.
Filename: `maxmin_f.py`.

Exercise 3.22: Find the max and min elements in a list

Given a list `a`, the `max` function in Python's standard library computes the largest element in `a`: `max(a)`. Similarly, `min(a)` returns the smallest element in `a`. Write your own `max` and `min` functions.

Hint. Initialize a variable `max_elem` by the first element in the list, then visit all the remaining elements (`a[1:]`), compare each element to `max_elem`, and if greater, set `max_elem` equal to that element. Use a similar technique to compute the minimum element.
Filename: `maxmin_list.py`.

Exercise 3.23: Implement the Heaviside function

The following *step function* is known as the *Heaviside function* and is widely used in mathematics:

$$H(x) = \begin{cases} 0, \, x < 0 \\ 1, \, x \geq 0 \end{cases} \tag{3.24}$$

a) Implement $H(x)$ in a Python function `H(x)`.

b) Make a Python function `test_H()` for testing the implementation of `H(x)`. Compute $H(-10)$, $H(-10^{-15})$, $H(0)$, $H(10^{-15})$, $H(10)$ and test that the answers are correct.
Filename: `Heaviside.py`.

Exercise 3.24: Implement a smoothed Heaviside function

The Heaviside function (3.24) listed in Exercise 3.23 is discontinuous. It is in many numerical applications advantageous to work with a smooth version of the Heaviside function where the function itself and its first derivative are continuous. One such smoothed Heaviside function is

$$H_\epsilon(x) = \begin{cases} 0, & x < -\epsilon, \\ \frac{1}{2} + \frac{x}{2\epsilon} + \frac{1}{2\pi}\sin\left(\frac{\pi x}{\epsilon}\right), & -\epsilon \leq x \leq \epsilon \\ 1, & x > \epsilon \end{cases} \tag{3.25}$$

a) Implement $H_\epsilon(x)$ in a Python function `H_eps(x, eps=0.01)`.

b) Make a Python function `test_H_eps()` for testing the implementation of `H_eps`. Check the values of some $x < -\epsilon$, $x = -\epsilon$, $x = 0$, $x = \epsilon$, and some $x > \epsilon$.
Filename: `smoothed_Heaviside.py`.

Exercise 3.25: Implement an indicator function

In many applications there is need for an indicator function, which is 1 over some interval and 0 elsewhere. More precisely, we define

$$I(x; L, R) = \begin{cases} 1, \, x \in [L, R], \\ 0, \, \text{elsewhere} \end{cases} \tag{3.26}$$

a) Make two Python implementations of such an indicator function, one with a direct test if $x \in [L, R]$ and one that expresses the indicator function in terms of Heaviside functions (3.24):

$$I(x; L, R) = H(x - L)H(R - x)\,. \tag{3.27}$$

b) Make a test function for verifying the implementation of the functions in a). Check that correct values are returned for some $x < L$, $x = L$, $x = (L + R)/2$, $x = R$, and some $x > R$.
Filename: `indicator_func.py`.

Exercise 3.26: Implement a piecewise constant function

Piecewise constant functions have a lot of important applications when modeling physical phenomena by mathematics. A piecewise constant function can be defined as

$$f(x) = \begin{cases} v_0,\ x \in [x_0, x_1), \\ v_1,\ x \in [x_1, x_2), \\ \vdots \\ v_i \quad x \in [x_i, x_{i+1}), \\ \vdots \\ v_n \quad x \in [x_n, x_{n+1}] \end{cases} \tag{3.28}$$

That is, we have a union of non-overlapping intervals covering the domain $[x_0, x_{n+1}]$, and $f(x)$ is constant in each interval. One example is the function that is -1 on $[0, 1]$, 0 on $[1, 1.5]$, and 4 on $[1.5, 2]$, where we with the notation in (3.28) have $x_0 = 0, x_1 = 1, x_2 = 1.5, x_3 = 2$ and $v_0 = -1, v_1 = 0, v_3 = 4$.

a) Make a Python function `piecewise(x, data)` for evaluating a piecewise constant mathematical function as in (3.28) at the point `x`. The `data` object is a list of pairs (v_i, x_i) for $i = 0, \ldots, n$. For example, `data` is `[(0, -1), (1, 0), (1.5, 4)]` in the example listed above. Since x_{n+1} is not a part of the `data` object, we have no means for detecting whether `x` is to the right of the last interval $[x_n, x_{n+1}]$, i.e., we must assume that the user of the `piecewise` function sends in an $x \le x_{n+1}$.

b) Design suitable test cases for the function `piecewise` and implement them in a test function `test_piecewise()`.
Filename: `piecewise_constant1.py`.

Exercise 3.27: Apply indicator functions

Implement piecewise constant functions, as defined in Exercise 3.26, by observing that

$$f(x) = \sum_{i=0}^{n} v_i I(x; x_i, x_{i+1}), \tag{3.29}$$

where $I(x; x_i, x_{i+1})$ is the indicator function from Exercise 3.25. Filename: `piecewise_constant2.py`.

Exercise 3.28: Test your understanding of branching

Consider the following code:

```
def where1(x, y):
    if x > 0:
        print 'quadrant I or IV'
    if y > 0:
        print 'quadrant I or II'

def where2(x, y):
    if x > 0:
        print 'quadrant I or IV'
    elif y > 0:
        print 'quadrant II'

for x, y in (-1, 1), (1, 1):
    where1(x,y)
    where2(x,y)
```

What is printed?

Exercise 3.29: Simulate nested loops by hand

Go through the code below by hand, statement by statement, and calculate the numbers that will be printed.

```
n = 3
for i in range(-1, n):
    if i != 0:
        print i

for i in range(1, 13, 2*n):
    for j in range(n):
        print i, j

for i in range(1, n+1):
    for j in range(i):
        if j:
            print i, j

for i in range(1, 13, 2*n):
    for j in range(0, i, 2):
        for k in range(2, j, 1):
            b = i > j > k
            if b:
                print i, j, k
```

You may use a debugger, see Section F.1, or the Online Python Tutor[9], see Section 3.1.2, to control what happens when you step through the code.

Exercise 3.30: Rewrite a mathematical function

We consider the $L(x;n)$ sum as defined in Section 3.1.8 and the corresponding function `L3(x, epsilon)` function from Section 3.1.10. The sum $L(x;n)$ can be written as

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

a) Derive a relation between c_i and c_{i-1},

$$c_i = ac_{i-1},$$

where a is an expression involving i and x.

b) The relation $c_i = ac_{i-1}$ means that we can start with `term` as c_1, and then in each pass of the loop implementing the sum $\sum_i c_i$ we can compute the next term c_i in the sum as

```
term = a*term
```

Write a new version of the `L3` function, called `L3_ci(x, epsilon)`, that makes use of this alternative computation of the terms in the sum.

c) Write a Python function `test_L3_ci()` that verifies the implementation of `L3_ci` by comparing with the original `L3` function.
Filename: `L3_recursive.py`.

Exercise 3.31: Make a table for approximations of $\cos x$

The function $\cos(x)$ can be approximated by the sum

$$C(x;n) = \sum_{j=0}^{n} c_j, \tag{3.30}$$

where

$$c_j = -c_{j-1}\frac{x^2}{2j(2j-1)}, \quad j = 1, 2, \ldots, n,$$

and $c_0 = 1$.

a) Make a Python function for computing $C(x;n)$.

[9] `http://www.pythontutor.com/`

Hint. Represent c_j by a variable `term`, make updates `term = -term*...` inside a `for` loop, and accumulate the `term` variable in a variable for the sum.

b) Make a function for writing out a table of the errors in the approximation $C(x;n)$ of $\cos(x)$ for some x and n values given as arguments to the function. Let the x values run downward in the rows and the n values to the right in the columns. For example, a table for $x = 4\pi, 6\pi, 8\pi, 10\pi$ and $n = 5, 25, 50, 100, 200$ can look like

```
       x          5         25         50        100        200
 12.5664   1.61e+04   1.87e-11   1.74e-12   1.74e-12   1.74e-12
 18.8496   1.22e+06   2.28e-02   7.12e-11   7.12e-11   7.12e-11
 25.1327   2.41e+07   6.58e+04  -4.87e-07  -4.87e-07  -4.87e-07
 31.4159   2.36e+08   6.52e+09   1.65e-04   1.65e-04   1.65e-04
```

Observe how the error increases with x and decreases with n.
Filename: `cos_sum.py`.

Exercise 3.32: Use None in keyword arguments

Consider the functions `L2(x, n)` and `L3(x, epsilon)` from Sections 3.1.8 and 3.1.10, whose program code is found in the file `lnsum.py`.

Make a more flexible function `L4` where we can either specify a tolerance `epsilon` *or* a number of terms `n` in the sum. Moreover, we can also choose whether we want the sum to be returned or the sum and the number of terms:

```
value, n = L4(x, epsilon=1E-8, return_n=True)
value = L4(x, n=100)
```

Hint. The starting point for all this flexibility is to have some keyword arguments initialized to an "undefined" value, called `None`, which can be recognized inside the function:

```
def L3(x, n=None, epsilon=None, return_n=False):
    if n is not None:
        ...
    if epsilon is not None:
        ...
```

One can also apply `if n != None`, but the `is` operator is most common.

Print error messages for incompatible values when `n` *and* `epsilon` are `None` or both are given by the user.
Filename: `L4.py`.

Exercise 3.33: Write a sort function for a list of 4-tuples

Below is a list of the nearest stars and some of their properties. The list elements are 4-tuples containing the name of the star, the distance from the sun in light years, the apparent brightness, and the luminosity. The apparent brightness is how bright the stars look in our sky compared to the brightness of Sirius A. The luminosity, or the true brightness, is how bright the stars would look if all were at the same distance compared to the Sun. The list data are found in the file `stars.txt`[10], which looks as follows:

```
data = [
('Alpha Centauri A',    4.3,  0.26,      1.56),
('Alpha Centauri B',    4.3,  0.077,     0.45),
('Alpha Centauri C',    4.2,  0.00001,   0.00006),
("Barnard's Star",      6.0,  0.00004,   0.0005),
('Wolf 359',            7.7,  0.000001,  0.00002),
('BD +36 degrees 2147', 8.2,  0.0003,    0.006),
('Luyten 726-8 A',      8.4,  0.000003,  0.00006),
('Luyten 726-8 B',      8.4,  0.000002,  0.00004),
('Sirius A',            8.6,  1.00,      23.6),
('Sirius B',            8.6,  0.001,     0.003),
('Ross 154',            9.4,  0.00002,   0.0005),
]
```

The purpose of this exercise is to sort this list with respect to distance, apparent brightness, and luminosity. Write a program that initializes the `data` list as above and writes out three sorted tables: star name versus distance, star name versus apparent brightness, and star name versus luminosity.

Hint. To sort a list `data`, one can call `sorted(data)`, as in

```
for item in sorted(data):
    ...
```

However, in the present case each element is a 4-tuple, and the default sorting of such 4-tuples results in a list with the stars appearing in alphabetic order. This is not what you want. Instead, we need to sort with respect to the 2nd, 3rd, or 4th element of each 4-tuple. If such a tailored sort mechanism is necessary, we can provide our own sort function as a second argument to `sorted`: `sorted(data, mysort)`. A sort function `mysort` must take two arguments, say `a` and `b`, and return -1 if `a` should become before `b` in the sorted sequence, 1 if `b` should become before `a`, and 0 if they are equal. In the present case, `a` and `b` are 4-tuples, so we need to make the comparison between the right elements in `a` and `b`. For example, to sort with respect to luminosity we can write

```
def mysort(a, b):
    if a[3] < b[3]:
        return -1
    elif a[3] > b[3]:
        return 1
    else:
        return 0
```

Filename: `sorted_stars_data.py`.

[10] `http://tinyurl.com/pwyasaa/funcif/stars.txt`

Exercise 3.34: Find prime numbers

The *Sieve of Eratosthenes* is an algorithm for finding all prime numbers less than or equal to a number N. Read about this algorithm on Wikipedia and implement it in a Python program. Filename: `find_primes.py`.

Exercise 3.35: Find pairs of characters

Write a function `count_pairs(dna, pair)` that returns the number of occurrences of a pair of characters (`pair`) in a DNA string (`dna`). For example, calling the function with `dna` as `'ACTGCTATCCATT'` and `pair` as `'AT'` will return 2. Filename: `count_pairs.py`.

Exercise 3.36: Count substrings

This is an extension of Exercise 3.35: count how many times a certain string appears in another string. For example, the function returns 3 when called with the DNA string `'ACGTTACGGAACG'` and the substring `'ACG'`.

Hint. For each match of the first character of the substring in the main string, check if the next `n` characters in the main string matches the substring, where `n` is the length of the substring. Use slices like `s[3:9]` to pick out a substring of `s`.
Filename: `count_substr.py`.

Exercise 3.37: Resolve a problem with a function

Consider the following interactive session:

```
>>> def f(x):
...     if 0 <= x <= 2:
...         return x**2
...     elif 2 < x <= 4:
...         return 4
...     elif x < 0:
...         return 0
...
>>> f(2)
4
>>> f(5)
>>> f(10)
```

Why do we not get any output when calling `f(5)` and `f(10)`?

Hint. Save the `f` value in a variable `r` and do `print r`.

Exercise 3.38: Determine the types of some objects

Consider the following calls to the `makelist` function from Section 3.1.6:

```
l1 = makelist(0, 100, 1)
l2 = makelist(0, 100, 1.0)
l3 = makelist(-1, 1, 0.1)
l4 = makelist(10, 20, 20)
l5 = makelist([1,2], [3,4], [5])
l6 = makelist((1,-1,1), ('myfile.dat', 'yourfile.dat'))
l7 = makelist('myfile.dat', 'yourfile.dat', 'herfile.dat')
```

Simulate each call by hand to determine what type of objects that become elements in the returned list and what the contents of `value` is after one pass in the loop.

Hint. Note that some of the calls will lead to infinite loops if you really perform the above `makelist` calls on a computer.

You can go to the Online Python Tutor[11], paste in the `makelist` function and the session above, and step through the program to see what actually happens.

Remarks. This exercise demonstrates that we can write a function and have in mind certain types of arguments, here typically `int` and `float` objects. However, the function can be used with other (originally unintended) arguments, such as lists and strings in the present case, leading to strange and irrelevant behavior (the problem here lies in the boolean expression `value <= stop` which is meaningless for some of the arguments).

Exercise 3.39: Find an error in a program

Consider the following program for computing

$$f(x) = e^{rx}\sin(mx) + e^{sx}\sin(nx)\,.$$

```
def f(x, m, n, r, s):
    return expsin(x, r, m) + expsin(x, s, n)

x = 2.5
print f(x, 0.1, 0.2, 1, 1)

from math import exp, sin

def expsin(x, p, q):
    return exp(p*x)*sin(q*x)
```

Running this code results in

```
NameError: global name 'expsin' is not defined
```

What is the problem? Simulate the program flow by hand, use the debugger to step from line to line, or use the Online Python Tutor. Correct the program.

[11] `http://www.pythontutor.com/`

User input and error handling 4

Consider a program for evaluating the formula $x = A\sin(wt)$:

```
from math import sin
A = 0.1
w = 1
t = 0.6
x = A*sin(w*t)
print x
```

In this program, `A`, `w`, and `t` are input data in the sense that these parameters must be known before the program can perform the calculation of `x`. The results produced by the program, here `x`, constitute the output data.

Input data can be hardcoded in the program as we do above. That is, we explicitly set variables to specific values: `A=0.1`, `w=1`, `t=0.6`. This programming style may be suitable for small programs. In general, however, it is considered good practice to let a user of the program provide input data when the program is running. There is then no need to modify the program itself when a new set of input data is to be explored. This is an important feature, because a golden rule of programming is that modification of the source code always represents a danger of introducing new errors by accident.

This chapter starts with describing three different ways of reading data into a program:

1. let the user answer questions in a dialog in the terminal window (Section 4.1),
2. let the user provide input on the command line (Section 4.2),
3. let the user provide input data in a file (Section 4.5),
4. let the user write input data in a graphical interface (Section 4.8).

Even if your program works perfectly, wrong input data from the user may cause the program to produce wrong answers or even crash. Checking that the input data are correct is important, and Section 4.7 tells you how to do this with so-called exceptions.

The Python programming environment is organized as a big collection of modules. Organizing your own Python software in terms of modules is therefore a natural and wise thing to do. Section 4.9 tells you how easy it is to make your own modules.

All the program examples from the present chapter are available in files in the `src/input`[1] folder.

4.1 Asking questions and reading answers

One of the simplest ways of getting data into a program is to ask the user a question, let the user type in an answer, and then read the text in that answer into a variable in the program. These tasks are done by calling a function with name `raw_input` in Python 2 - the name is just `input` in Python 3.

4.1.1 Reading keyboard input

A simple problem involving the temperature conversion from Celsius to Fahrenheit constitutes our main example: $F = \frac{9}{5}C + 32$. The associated program with setting `C` explicitly in the program reads

```
C = 22
F = 9./5*C + 32
print F
```

We may ask the user a question `C=?` and wait for the user to enter a number. The program can then read this number and store it in a variable `C`. These actions are performed by the statement

```
C = raw_input('C=? ')
```

The `raw_input` function always returns the user input as a string object. That is, the variable `C` above refers to a string object. If we want to compute with this `C`, we must convert the string to a floating-point number: `C = float(C)`. A complete program for reading `C` and computing the corresponding degrees in Fahrenheit now becomes

```
    C = raw_input('C=? ')
    C = float(C)
    F = (9./5)*C + 32
    print F
```

In general, the `raw_input` function takes a string as argument, displays this string in the terminal window, waits until the user presses the Return

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

key, and then returns a string object containing the sequence of characters that the user typed in.

The program above is stored in a file called `c2f_qa.py` (the `qa` part of the name reflects *question and answer*). We can run this program in several ways. The convention in this book is to indicate the execution by writing the program name only, but for a real execution you need to do more: write `run` before the program name in an interactive IPython session, or write `python` before the program name in a terminal session. Here is the execution of our sample program and the resulting dialog with the user:

Terminal

```
c2f_qa.py
C=? 21
69.8
```

In this particular example, the `raw_input` function reads the characters `21` from the keyboard and returns the string `'21'`, which we refer to by the variable `C`. Then we create a new `float` object by `float(C)` and let the name `C` refer to this `float` object, with value 21.

You should now try out Exercises 4.1, 4.6, and 4.9 to make sure you understand how `raw_input` behaves.

4.2 Reading from the command line

Programs running on Unix computers usually avoid asking the user questions. Instead, input data are very often fetched from the *command line*. This section explains how we can access information on the command line in Python programs.

4.2.1 Providing input on the command line

We look at the Celsius-Fahrenheit conversion program again. The idea now is to provide the Celsius input temperature as a *command-line argument* right after the program name. This means that we write the program name, here `c2f_cml.py` (`cml` for *command line*), followed the Celsius temperature:

Terminal

```
c2f_cml.py 21
69.8
```

Inside the program we can fetch the text `21` as `sys.argv[1]`. The `sys` module has a list `argv` containing all the command-line arguments to the program, i.e., all the “words” appearing after the program name when we run the program. In the present case there is only one argument and it is stored in `sys.argv[1]`. The first element in the `sys.argv` list, `sys.argv[0]`, is always the name of the program.

A command-line argument is treated as a text, so `sys.argv[1]` refers to a string object, in this case `'21'`. Since we interpret the command-line argument as a number and want to compute with it, it is necessary to explicitly convert the string to a `float` object. In the program we therefore write

```
import sys
C = float(sys.argv[1])
F = 9.0*C/5 + 32
print F
```

As another example, consider the program

```
v0 = 5
g = 9.81
t = 0.6
y = v0*t - 0.5*g*t**2
print y
```

for computing the formula $y(t) = v_0t - \frac{1}{2}gt^2$. Instead of hardcoding the values of `v0` and `t` in the program we can read the two values from the command line:

Terminal

```
ball2_cml.py 0.6 5
1.2342
```

The two command-line arguments are now available as `sys.argv[1]` and `sys.argv[2]`. The complete `ball2_cml.py` program thus takes the form

```
import sys
t  = float(sys.argv[1])
v0 = float(sys.argv[2])
g = 9.81
y = v0*t - 0.5*g*t**2
print y
```

4.2.2 A variable number of command-line arguments

Let us make a program `addall.py` that adds all its command-line arguments. That is, we may run something like

Terminal

```
addall.py 1 3 5 -9.9
The sum of 1 3 5 -9.9 is -0.9
```

The command-line arguments are stored in the sublist `sys.argv[1:]`. Each element is a string so we must perform a conversion to `float` before performing the addition. There are many ways to write this program. Let us start with version 1, `addall_v1.py`:

```
import sys
s = 0
for arg in sys.argv[1:]:
    number = float(arg)
    s += number
print 'The sum of ',
for arg in sys.argv[1:]:
    print arg,
print 'is ', s
```

The output is on one line, but built of several `print` statements with a comma at the end to prevent the usual newline character that `print` otherwise adds to the text. The command-line arguments must be converted to numbers in the first `for` loop because we need to compute with them, but in the second loop we only need to print them and then the string representation is appropriate.

The program above can be written more compactly if desired:

```
import sys
s = sum([float(x) for x in sys.argv[1:]])
print 'The sum of %s is %s' % (' '.join(sys.argv[1:]), s)
```

Here, we convert the list `sys.argv[1:]` to a list of `float` objects and then pass this list to Python's `sum` function for adding the numbers. The construction `S.join(L)` places all the elements in the list `L` after each other with the string `S` in between. The result here is a string with all the elements in `sys.argv[1:]` and a space in between, which is the text that originally appeared on the command line.

4.2.3 More on command-line arguments

Unix commands make heavy use of command-line arguments. For example, when you write `ls -s -t` to list the files in the current folder, you run the program `ls` with two command-line arguments: `-s` and `-t`. The former specifies that `ls` is to print the file name together with the size of the file, and the latter sorts the list of files according to their dates of last modification. Similarly, `cp -r my new` for copying a folder tree `my` to a new folder tree `new` invokes the `cp` program with three command line arguments: `-r` (for recursive copying of files), `my`, and `new`. Most programming languages have support for extracting the command-line arguments given to a program.

An important rule is that *command-line arguments are separated by blanks*. What if we want to provide a text containing blanks as command-line argument? The text containing blanks must then appear inside single or double quotes. Let us demonstrate this with a program that simply prints the command-line arguments:

```
import sys, pprint
pprint.pprint(sys.argv[1:])
```

Say this program is named `print_cml.py`. The execution

Terminal

```
print_cml.py 21 a string with blanks 31.3
['21', 'a', 'string', 'with', 'blanks', '31.3']
```

demonstrates that each word on the command line becomes an element in `sys.argv`. Enclosing strings in quotes, as in

Terminal

```
print_cml.py 21 "a string with blanks" 31.3
['21', 'a string with blanks', '31.3']
```

shows that the text inside the quotes becomes a single command line argument.

4.3 Turning user text into live objects

It is possible to provide text with valid Python code as input to a program and then turn the text into live objects as if the text were written directly into the program beforehand. This is a very powerful tool for letting users specify function formulas, for instance, as input to a program. The program code itself has no knowledge about the kind of function the user wants to work with, yet at run time the user's desired formula enters the computations.

4.3.1 The magic eval function

The `eval` functions takes a string as argument and evaluates this string as a Python *expression*. The result of an expression is an object. Consider

```
>>> r = eval('1+2')
>>> r
3
>>> type(r)
<type 'int'>
```

The result of `r = eval('1+2')` is the same as if we had written `r = 1+2` directly:

```
>>> r = 1+2
>>> r
3
>>> type(r)
<type 'int'>
```

In general, any valid Python expression stored as text in a string `s` can be turned into live Python code by `eval(s)`.

Here is an example where the string to be evaluated is `'2.5'`, which causes Python to see `r = 2.5` and make a `float` object:

```
>>> r = eval('2.5')
>>> r
2.5
>>> type(r)
<type 'float'>
```

Let us proceed with some more examples. We can put the initialization of a list inside quotes and use `eval` to make a `list` object:

```
>>> r = eval('[1, 6, 7.5]')
>>> r
[1, 6, 7.5]
>>> type(r)
<type 'list'>
```

Again, the assignment to `r` is equivalent to writing

```
>>> r = [1, 6, 7.5]
```

We can also make a `tuple` object by using tuple syntax (standard parentheses instead of brackets):

```
>>> r = eval('(-1, 1)')
>>> r
(-1, 1)
>>> type(r)
<type 'tuple'>
```

Another example reads

```
>>> from math import sqrt
>>> r = eval('sqrt(2)')
>>> r
1.4142135623730951
>>> type(r)
<type 'float'>
```

At the time we run `eval('sqrt(2)')`, this is the same as if we had written

```
>>> r = sqrt(2)
```

directly, and this is valid syntax only if the `sqrt` function is defined. Therefore, the import of `sqrt` prior to running `eval` is important in this example.

Applying eval to strings. If we put a string, enclosed in quotes, inside the expression string, the result is a string object:

```
>>>
>>> r = eval('"math programming"')
>>> r
'math programming'
>>> type(r)
<type 'str'>
```

Note that we must use two types of quotes: first double quotes to mark `math programming` as a string object and then another set of quotes, here single quotes (but we could also have used triple single quotes), to embed the text `"math programming"` inside a string. It does not matter if we have single or double quotes as inner or outer quotes, i.e., `'"..."'` is the same as `"'...'"`, because `'` and `"` are interchangeable as long as a pair of either type is used consistently.

Writing just

```
>>> r = eval('math programming')
```

is the same as writing

```
>>> r = math programming
```

which is an invalid expression. Python will in this case think that `math` and `programming` are two (undefined) variables, and setting two variables next to each other with a space in between is invalid Python syntax. However,

```
>>> r = 'math programming'
```

is valid syntax, as this is how we initialize a string `r` in Python. To repeat, if we put the valid syntax `'math programming'` inside a string,

```
s = "'math programming'"
```

`eval(s)` will evaluate the text inside the double quotes as `'math programming'`, which yields a string.

Applying eval to user input. So, why is the `eval` function so useful? When we get input via `raw_input` or `sys.argv`, it is always in the form of a string object, which often must be converted to another type of object, usually an `int` or `float`. Sometimes we want to avoid specifying one particular type. The `eval` function can then be of help: we feed the string object from the input to the `eval` function and let the it interpret the string and convert it to the right object.

An example may clarify the point. Consider a small program where we read in two values and add them. The values could be strings, floats, integers, lists, and so forth, as long as we can apply a `+` operator to the values. Since we do not know if the user supplies a string, float, integer, or something else, we just convert the input by `eval`, which means that the user's syntax will determine the type. The program goes as follows (`add_input.py`):

```
i1 = eval(raw_input('Give input: '))
i2 = eval(raw_input('Give input: '))
r = i1 + i2
print '%s + %s becomes %s\nwith value %s' % \
      (type(i1), type(i2), type(r), r)
```

Observe that we write out the two supplied values, together with the types of the values (obtained by `eval`), and the sum. Let us run the program with an integer and a real number as input:

Terminal

```
add_input.py
Give input: 4
Give input: 3.1
<type 'int'> + <type 'float'> becomes <type 'float'>
with value 7.1
```

The string `'4'`, returned by the first call to `raw_input`, is interpreted as an `int` by `eval`, while `'3.1'` gives rise to a `float` object.

Supplying two lists also works fine:

Terminal

```
add_input.py
Give input: [-1, 3.2]
Give input: [9,-2,0,0]
<type 'list'> + <type 'list'> becomes <type 'list'>
with value [-1, 3.2000000000000002, 9, -2, 0, 0]
```

If we want to use the program to add two strings, the strings must be enclosed in quotes for `eval` to recognize the texts as string objects (without the quotes, `eval` aborts with an error):

Terminal

```
add_input.py
Give input: 'one string'
Give input: " and another string"
<type 'str'> + <type 'str'> becomes <type 'str'>
with value one string and another string
```

Not all objects are meaningful to add:

Terminal

```
add_input.py
Give input: 3.2
Give input: [-1,10]
Traceback (most recent call last):
  File "add_input.py", line 3, in <module>
    r = i1 + i2
TypeError: unsupported operand type(s) for +: 'float' and 'list'
```

A similar program adding two arbitrary command-line arguments reads ((`add_input.py`):

```
import sys
i1 = eval(sys.argv[1])
i2 = eval(sys.argv[2])
r = i1 + i2
print '%s + %s becomes %s\nwith value %s' % \
      (type(i1), type(i2), type(r), r)
```

Another important example on the usefulness of `eval` is to turn formulas, given as input, into mathematics in the program. Consider the program

```
from math import *   # make all math functions available
import sys
formula = sys.argv[1]
x = eval(sys.argv[2])
result = eval(formula)
print '%s for x=%g yields %g' % (formula, x, result)
```

Two command-line arguments are expected: a formula and a number. Say the formula given is `2*sin(x)+1` and the number is 3.14. This information is read from the command line as strings. Doing `x = eval(sys.argv[2])` means `x = eval('3.14')`, which is equivalent to `x = 3.14`, and `x` refers to a `float` object. The `eval(formula)` expression means `eval('2*sin(x)+1')`, and the corresponding statement `result = eval(formula` is therefore effectively `result = 2*sin(x)+1`, which requires `sin` and `x` to be defined objects. The result is a `float` (approximately 1.003). Providing `cos(x)` as the first command-line argument creates a need to have `cos` defined, so that is why we import all functions from the `math` module. Let us try to run the program:

```
Terminal
eval_formula.py "2*sin(x)+1" 3.14
2*sin(x)+1 for x=3.14 yields 1.00319
```

The very nice thing with using `eval` in `x = eval(sys.argv[2])` is that we can provide mathematical expressions like `pi/2` or even `tanh(2*pi)`, as the latter just effectively results in the statement `x = tanh(2*pi)`, and this works fine as long has we have imported `tanh` and `pi`.

4.3.2 The magic exec function

Having presented `eval` for turning strings into Python code, we take the opportunity to also describe the related `exec` function to execute a string containing arbitrary Python code, not only an expression.

Suppose the user can write a formula as input to the program, available to us in the form of a string object. We would then like to turn this formula into a callable Python function. For example, writing `sin(x)*cos(3*x) + x**2` as the formula, we would make the function

```
def f(x):
    return sin(x)*cos(3*x) + x**2
```

This is easy with `exec`: just construct the right Python syntax for defining `f(x)` in a string and apply `exec` to the string,

```
formula = sys.argv[1]
code = """
def f(x):
    return %s
""" % formula
exec(code)
```

As an example, think of `"sin(x)*cos(3*x) + x**2"` as the first command-line argument. Then `formula` will hold this text, which is inserted into the `code` string such that it becomes

```
"""
def f(x):
    return sin(x)*cos(3*x) + x**2
"""
```

Thereafter, `exec(code)` executes the code as if we had written the contents of the `code` string directly into the program by hand. With this technique, we can turn any user-given formula into a Python function!

Let us now use this technique in a useful application. Suppose we have made a function for computing the integral $\int_a^b f(x)dx$ by the Midpoint rule with n intervals:

```
def midpoint_integration(f, a, b, n=100):
    h = (b - a)/float(n)
    I = 0
    for i in range(n):
        I += f(a + i*h + 0.5*h)
    return h*I
```

We now want to read a, b, and n from the command line as well as the formula that makes up the $f(x)$ function:

```
from math import *
import sys
f_formula = sys.argv[1]
a = eval(sys.argv[2])
b = eval(sys.argv[3])
if len(sys.argv) >= 5:
    n = int(sys.argv[4])
else:
    n = 200
```

Note that we import everything from `math` and use `eval` when reading the input for `a` and `b` as this will allow the user to provide values like `2*cos(pi/3)`.

The next step is to convert the `f_formula` for $f(x)$ into a Python function `g(x)`:

```
code = """
def g(x):
    return %s
""" % f_formula
exec(code)
```

Now we have an ordinary Python function `g(x)` that we can ask the integration function to integrate:

```
I = midpoint_integration(g, a, b, n)
print 'Integral of %s on [%g, %g] with n=%d: %g' % \
      (f_formula, a, b, n, I)
```

The complete code is found in `integrate.py`. A sample run for $\int_0^{\pi/2} \sin(x)dx$ goes like

```
Terminal
integrate.py "sin(x)" 0 pi/2
integral of sin(x) on [0, 1.5708] with n=200: 0.583009
```

(The quotes in `"sin(x)"` are needed because of the parenthesis will otherwise be interpreted by the shell.)

4.3.3 Turning string expressions into functions

The examples in the previous section indicate that it can be handy to ask the user for a formula and turn that formula into a Python function. Since this operation is so useful, we have made a special tool that hides the technicalities. The tool is named `StringFunction` and works as follows:

```
>>> from scitools.StringFunction import StringFunction
>>> formula = 'exp(x)*sin(x)'
>>> f = StringFunction(formula)   # turn formula into function f(x)
```

The `f` object now behaves as an ordinary Python function of `x`:

```
>>> f(0)
0.0
>>> f(pi)
2.8338239229952166e-15
>>> f(log(1))
0.0
```

Expressions involving other independent variables than `x` are also possible. Here is an example with the function $g(t) = Ae^{-at}\sin(\omega x)$:

```
g = StringFunction('A*exp(-a*t)*sin(omega*x)',
                   independent_variable='t',
                   A=1, a=0.1, omega=pi, x=0.5)
```

The first argument is the function formula, as before, but now we need to specify the name of the independent variable (`'x'` is default). The other parameters in the function (A, a, ω, and x) must be specified with values, and we use keyword arguments, consistent with the names in the function formula, for this purpose. Any of the parameters `A`, `a`, `omega`, and `x` can be changed later by calls like

```
g.set_parameters(omega=0.1)
g.set_parameters(omega=0.1, A=5, x=0)
```

Calling `g(t)` works as if `g` were a plain Python function of `t`, which also stores all the parameters `A`, `a`, `omega`, and `x`, and their values. You can use `pydoc` to bring up more documentation on the possibilities with `StringFunction`. Just run

```
    pydoc scitools.StringFunction.StringFunction
```

A final important point is that `StringFunction` objects are as computationally efficient as hand-written Python functions. (This property is quite remarkable, as a string formula will in most other programming languages be much slower to evaluate than if the formula were hardcoded inside a plain function.)

4.4 Option-value pairs on the command line

The examples on using command-line arguments so far require the user of the program to type all arguments in their right sequence, just as when calling a function with positional arguments in the right order. It would be very convenient to assign command-line arguments in the same way as we use keyword arguments. That is, arguments are associated with a name, their sequence can be arbitrary, and only the arguments where the default value is not appropriate need to be given. Such type of command-line arguments may have `-option value` pairs, where `option` is some name of the argument.

As usual, we shall use an example to illustrate how to work with `-option value` pairs. Consider the physics formula for the location $s(t)$ of an object at time t, given that the object started at $s = s_0$ at $t = 0$ with a velocity v_0, and thereafter was subject to a constant acceleration a:

$$s(t) = s_0 + v_0 t + \frac{1}{2}at^2 \,. \tag{4.1}$$

This formula requires four input variables: s_0, v_0, a, and t. We can make a program `location.py` that takes four options, `-s0`, `-v0`, `-a`, and `-t`, on the command line. The program is typically run like this:

Terminal

```
location.py --t 3 --s0 1 --v0 1 --a 0.5
```

The sequence of `-option value` pairs is arbitrary. All options have a default value such that one does not have to specify all options on the command line.

All input variables should have sensible default values such that we can leave out the options for which the default value is suitable. For example, if $s_0 = 0$, $v_0 = 0$, $a = 1$, and $t = 1$ by default, and we only want to change t, we can run

Terminal

```
location.py --t 3
```

4.4.1 Basic usage of the argparse module

Python has a flexible and powerful module `argparse` for reading (parsing) `-option value` pairs on the command line. Using `argparse` consists of three steps. First, a parser object must be created:

```
import argparse
parser = argparse.ArgumentParser()
```

Second, we need to define the various command-line options,

```
parser.add_argument('--v0', '--initial_velocity', type=float,
                    default=0.0, help='initial velocity',
                    metavar='v')
parser.add_argument('--s0', '--initial_position', type=float,
                    default=0.0, help='initial position',
                    metavar='s')
parser.add_argument('--a', '--acceleration', type=float,
                    default=1.0, help='acceleration', metavar='a')
parser.add_argument('--t', '--time', type=float,
                    default=1.0, help='time', metavar='t')
```

The first arguments to `parser.add_argument` is the set of options that we want to associate with an input parameter. Optional arguments are the type, a default value, a help string, and a name for the value of the argument (`metavar`) in a usage string. The `argparse` module will automatically allow an option `-h` or `-help` that prints a usage string for all the registered options. By default, the type is `str`, the default value is `None`, the help string is empty, and `metavar` is the option in upper case without initial dashes.

Third, we must read the command line arguments and interpret them:

```
args = parser.parse_args()
```

Through the `args` object we can extract the values of the various registered parameters: `args.v0`, `args.s0`, `args.a`, and `args.t`. The name of the parameter is determined by the first option to `parser.add_argument`, so writing

```
parser.add_argument('--initial_velocity', '--v0', type=float,
                    default=0.0, help='initial velocity')
```

will make the initial velocity value appear as `args.initial_velocity`. We can add the `dest` keyword to explicitly specify the name where the value is stored:

```
parser.add_argument('--initial_velocity', '--v0', dest='V0',
                    type=float, default=0, help='initial velocity')
```

Now, `args.V0` will retrieve the value of the initial velocity. In case we do not provide any default value, the value will be `None`.

Our example is completed either by evaluating `s` as

```
s = args.s0 + args.v0*t + 0.5*args.a*args.t**2
```

or by introducing new variables so that the formula aligns better with the mathematical notation:

```
s0 = args.s0; v0 = args.v0; a = args.a; t = args.t
s = s0 + v0*t + 0.5*a*t**2
```

A complete program for the example above is found in the file `location.py`. Try to run it with the `-h` option to see an automatically generated explanation of legal command-line options.

4.4.2 Mathematical expressions as values

Values on the command line involving mathematical symbols and functions, say `-v0 'pi/2'`, pose a problem with the code example above. The `argparse` module will in that case try to do `float('pi/2')` which does not work well since `pi` is an undefined name. Changing `type=float` to `type=eval` is required to interpret the expression `pi/2`, but even `eval('pi/2')` fails since `pi` is not defined inside the `argparse` module. There are various remedies for this problem.

One can write a tailored function for converting a string value given on the command line to the desired object. For example,

```
def evalcmlarg(text):
    return eval(text)

parser.add_argument('--s0', '--initial_position', type=evalcmlarg,
                    default=0.0, help='initial position')
```

The file `location_v2.py` demonstrates such explicit type conversion through a user-provided conversion function. Note that `eval` is now taken in the programmer's namespace where (hopefully) `pi` or other symbols are imported.

More sophisticated conversions are possible. Say s_0 is specified in terms of a function of some parameter p, like $s_0 = (1 - p^2)$. We could then use a string for `-s0` and the `StringFunction` tool from Section 4.3.3 to turn the string into a function:

```
def toStringFunction4s0(text):
    from scitools.std import StringFunction
    return StringFunction(text, independent_variable='p')

parser.add_argument('--s0', '--initial_position',
                    type=toStringFunction4s0,
                    default='0.0', help='initial position')
```

Giving a command-line argument `-s0 'exp(-1.5) + 10(1-p**2)` results in `args.s0` being a `StringFunction` object, which we must evaluate for a `p` value:

```
s0 = args.s0
p = 0.05
...
s = s0(p) + v0*t + 0.5*a*t**2
```

The file `location_v3.py` contains the complete code for this example.

Another alternative is to perform the correct conversion of values in our own code *after* the `parser` object has read the values. To this end, we treat argument types as strings in the `parser.add_argument` calls, meaning that we replace `type=float` by set `type=str` (which is also the default choice of `type`). Recall that this approach requires specification of default values as strings too, say `'0'`:

```
parser.add_argument('--s0', '--initial_position', type=str,
                    default='0', help='initial position')
...
from math import *
args.v0 = eval(args.v0)
# or
v0 = eval(args.v0)

s0 = StringFunction(args.s0, independent_variable='p')
p = 0.5
...
s = s0(p) + v0*t + 0.5*a*t**2
```

Such code is found in the file `location_v4.py`. You can try out that program with the command-line arguments `-s0 'pi/2 + sqrt(p)' -v0 pi/4'`.

The final alternative is to write an `Action` class to handle the conversion from string to the right type. This is the preferred way to perform conversions and well described in the `argparse` documentation. We shall exemplify it here, but the technicalities involved require understanding of classes (Chapter 7) and inheritance (Chapter 9). For the conversion from string to any object via `eval` we write

```
import argparse
from math import *

class ActionEval(argparse.Action):
    def __call__(self, parser, namespace, values,
                 option_string=None):
        setattr(namespace, self.dest, eval(values))
```

The command-line arguments supposed to be run through `eval` must then have an `action` parameter:

```
parser.add_argument('--v0', '--initial_velocity',
                    default=0.0, help='initial velocity',
                    action=ActionEval)
```

From string to function via `StringFunction` for the `-s0` argument we write

```
from scitools.std import StringFunction

class ActionStringFunction4s0(argparse.Action):
    def __call__(self, parser, namespace, values,
                 option_string=None):
        setattr(namespace, self.dest,
                StringFunction(values, independent_variable='p'))
```

A complete code appears in the file `location_v5.py`.

4.5 Reading data from file

Getting input data into a program from the command line, or from questions and answers in the terminal window, works for small amounts of data. Otherwise, input data must be available in files. Anyone with some computer experience is used to save and load data files in programs. The task now is to understand how Python programs can read and write files. The basic recipes are quite simple and illustrated through examples.

Suppose we have recorded some measurement data in the file `src/input/data.txt`[2]. The goal of our first example of reading files is to read the measurement values in `data.txt`, find the average value, and print it out in the terminal window.

Before trying to let a program read a file, we must know the *file format*, i.e., what the contents of the file looks like, because the structure of the text in the file greatly influences the set of statements needed to read the file. We therefore start with viewing the contents of the file `data.txt`. To this end, load the file into a text editor or viewer (one can use `emacs`, `vim`, `more`, or `less` on Unix and Mac, while on Windows, `WordPad` is appropriate, or the `type` command in a DOS or PowerShell window, and even Word processors such as LibreOffice or Microsoft Word can also be used on Windows). What we see is a column with numbers:

```
21.8
18.1
19
23
26
17.8
```

Our task is to read this column of numbers into a list in the program and compute the average of the list items.

4.5.1 Reading a file line by line

To read a file, we first need to *open* the file. This action creates a file object, here stored in the variable `infile`:

[2] `http://tinyurl.com/pwyasaa/input/data.txt`

```
infile = open('data.txt', 'r')
```

The second argument to the `open` function, the string `'r'`, tells that we want to open the file for reading. We shall later see that a file can be opened for writing instead, by providing `'w'` as the second argument. After the file is read, one should close the file object with `infile.close()`.

The basic technique for reading the file line by line applies a `for` loop like this:

```
for line in infile:
    # do something with line
```

The `line` variable is a string holding the current line in the file. The `for` loop over lines in a file has the same syntax as when we go through a list. Just think of the file object `infile` as a collection of elements, here lines in a file, and the `for` loop visits these elements in sequence such that the `line` variable refers to one line at a time. If something seemingly goes wrong in such a loop over lines in a file, it is useful to do a `print line` inside the loop.

Instead of reading one line at a time, we can load all lines into a list of strings (lines) by

```
lines = infile.readlines()
```

This statement is equivalent to

```
lines = []
for line in infile:
    lines.append(line)
```

or the list comprehension:

```
lines = [line for line in infile]
```

In the present example, we load the file into the list `lines`. The next task is to compute the average of the numbers in the file. Trying a straightforward sum of all numbers on all lines,

```
mean = 0
for number in lines:
    mean = mean + number
mean = mean/len(lines)
```

gives an error message:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The reason is that `lines` holds each line (`number`) as a string, not a `float` or `int` that we can add to other numbers. A fix is to convert each line to a `float`:

```
mean = 0
for line in lines:
    number = float(line)
    mean = mean + number
mean = mean/len(lines)
```

This code snippet works fine. The complete code can be found in the file `mean1.py`.

Summing up a list of numbers is often done in numerical programs, so Python has a special function `sum` for performing this task. However, `sum` must in the present case operate on a list of floats, not strings. We can use a list comprehension to turn all elements in `lines` into corresponding `float` objects:

```
mean = sum([float(line) for line in lines])/len(lines)
```

An alternative implementation is to load the lines into a list of `float` objects directly. Using this strategy, the complete program (found in file `mean2.py`) takes the form

```
infile = open('data.txt', 'r')
numbers = [float(line) for line in infile.readlines()]
infile.close()
mean = sum(numbers)/len(numbers)
print mean
```

4.5.2 Alternative ways of reading a file

A newcomer to programming might find it confusing to see that one problem is solved by many alternative sets of statements, but this is the very nature of programming. A clever programmer will judge several alternative solutions to a programming task and choose one that is either particularly compact, easy to understand, and/or easy to extend later. We therefore present more examples on how to read the `data.txt` file and compute with the data.

The modern with statement. Modern Python code applies the `with` statement to deal with files:

```
with open('data.txt', 'r') as infile:
    for line in infile:
        # process line
```

This snippet is equivalent to

```
infile = open('data.txt', 'r')
for line in infile:
    # process line
infile.close()
```

Note that there is no need to close the file when using the `with` statement. The advantage of the `with` construction is shorter code and better handling of errors if something goes wrong with opening or working with the file. A downside is that the syntax differs from the very classical open-close pattern that one finds in most other programming languages. Remembering to close a file is key in programming, and to train that task, we mostly apply the open-close construction in this book.

The old while construction. The call `infile.readline()` returns a string containing the text at the current line. A new `infile.readline()` will read the next line. When `infile.readline()` returns an empty string, the end of the file is reached and we must stop further reading. The following `while` loop reads the file line by line using `infile.readline()`:

```
while True:
    line = infile.readline()
    if not line:
        break
    # process line
```

This is perhaps a somewhat strange loop, but it is a well-established way of reading a file in Python, especially in older code. The shown `while` loop runs forever since the condition is always `True`. However, inside the loop we test if `line` is `False`, and it is `False` when we reach the end of the file, because `line` then becomes an empty string, which in Python evaluates to `False`. When `line` is `False`, the `break` statement breaks the loop and makes the program flow jump to the first statement after the `while` block.

Computing the average of the numbers in the `data.txt` file can now be done in yet another way:

```
infile = open('data.txt', 'r')
mean = 0
n = 0
while True:
    line = infile.readline()
    if not line:
        break
    mean += float(line)
    n += 1
mean = mean/float(n)
```

Reading a file into a string. The call `infile.read()` reads the whole file and returns the text as a string object. The following interactive session illustrates the use and result of `infile.read()`:

```
>>> infile = open('data.txt', 'r')
>>> filestr = infile.read()
>>> filestr
'21.8\n18.1\n19\n23\n26\n17.8\n'
>>> print filestr
21.8
18.1
19
23
26
17.8
```

Note the difference between just writing `filestr` and writing `print filestr`. The former dumps the string with newlines as *backslash n* characters, while the latter is a *pretty print* where the string is written out without quotes and with the newline characters as visible line shifts.

Having the numbers inside a string instead of inside a file does not look like a major step forward. However, string objects have many useful functions for extracting information. A very useful feature is *split*: `filestr.split()` will split the string into words (separated by blanks or any other sequence of characters you have defined). The "words" in this file are the numbers:

```
>>> words = filestr.split()
>>> words
['21.8', '18.1', '19', '23', '26', '17.8']
>>> numbers = [float(w) for w in words]
>>> mean = sum(numbers)/len(numbers)
>>> print mean
20.95
```

A more compact program looks as follows (`mean3.py`):

```
infile = open('data.txt', 'r')
numbers = [float(w) for w in infile.read().split()]
mean = sum(numbers)/len(numbers)
```

The next section tells you more about splitting strings.

4.5.3 Reading a mixture of text and numbers

The `data.txt` file has a very simple structure since it contains numbers only. Many data files contain a mix of text and numbers. The file `rainfall.dat` from `www.worldclimate.com`[3] provides an example:

```
Average rainfall (in mm) in Rome: 1188 months between 1782 and 1970
Jan  81.2
Feb  63.2
Mar  70.3
Apr  55.7
May  53.0
Jun  36.4
Jul  17.5
Aug  27.5
Sep  60.9
Oct  117.7
Nov  111.0
Dec  97.9
Year 792.9
```

How can we read the rainfall data in this file and store the information in lists suitable for further analysis? The most straightforward solution is to read the file line by line, and for each line split the line into words, store the first word (the month) in one list and the second word (the average rainfall) in another list. The elements in this latter list needs to be `float` objects if we want to compute with them.

[3] `http://www.worldclimate.com/cgi-bin/data.pl?ref=N41E012+2100+1623501G1`

The complete code, wrapped in a function, may look like this (file `rainfall1.py`):

```
def extract_data(filename):
    infile = open(filename, 'r')
    infile.readline() # skip the first line
    months = []
    rainfall = []
    for line in infile:
        words = line.split()
        # words[0]: month, words[1]: rainfall
        months.append(words[0])
        rainfall.append(float(words[1]))
    infile.close()
    months = months[:-1]      # Drop the "Year" entry
    annual_avg = rainfall[-1] # Store the annual average
    rainfall = rainfall[:-1]  # Redefine to contain monthly data
    return months, rainfall, annual_avg

months, values, avg = extract_data('rainfall.dat')
print 'The average rainfall for the months:'
for month, value in zip(months, values):
    print month, value
print 'The average rainfall for the year:', avg
```

Note that the first line in the file is just a comment line and of no interest to us. We therefore read this line by `infile.readline()` and do not store the content in any object. The `for` loop over the lines in the file will then start from the next (second) line.

We store all the data into 13 elements in the `months` and `rainfall` lists. Thereafter, we manipulate these lists a bit since we want `months` to contain the name of the 12 months only. The `rainfall` list should correspond to this `month` list. The annual average is taken out of `rainfall` and stored in a separate variable. Recall that the `-1` index corresponds to the last element of a list, and the slice `:-1` picks out all elements from the start up to, but not including, the last element.

We could, alternatively, have written a shorter code where the name of the months and the rainfall numbers are stored in a nested list:

```
def extract_data(filename):
    infile = open(filename, 'r')
    infile.readline()  # skip the first line
    data = [line.split() for line in infile]
    annual_avg = data[-1][1]
    data = [(m, float(r)) for m, r in data[:-1]]
    infile.close()
    return data, annual_avg
```

This is more advanced code, but understanding what is going on is a good test on the understanding of nested lists indexing and list comprehensions. An executable program is found in the file `rainfall2.py`.

Is it more to file reading? With the example code in this section, you have the very basic tools for reading files with a simple structure: columns of text or numbers. Many files used in scientific computations have such a format, but many files are more complicated too. Then you need the techniques of string processing. This is explained in detail in Chapter 6.

4.6 Writing data to file

Writing data to file is easy. There is basically one function to pay attention to: `outfile.write(s)`, which writes a string `s` to a file handled by the file object `outfile`. Unlike `print`, `outfile.write(s)` does not append a newline character to the written string. It will therefore often be necessary to add a newline character,

```
outfile.write(s + '\n')
```

if the string `s` is meant to appear on a single line in the file and `s` does not already contain a trailing newline character. File writing is then a matter of constructing strings containing the text we want to have in the file and for each such string call `outfile.write`.

Writing to a file demands the file object `f` to be opened for writing:

```
# write to new file, or overwrite file:
outfile = open(filename, 'w')

# append to the end of an existing file:
outfile = open(filename, 'a')
```

4.6.1 Example: Writing a table to file

Problem. As a worked example of file writing, we shall write out a nested list with tabular data to file. A sample list may take look as

```
[[ 0.75,        0.29619813, -0.29619813, -0.75      ],
 [ 0.29619813,  0.11697778, -0.11697778, -0.29619813],
 [-0.29619813, -0.11697778,  0.11697778,  0.29619813],
 [-0.75,       -0.29619813,  0.29619813,  0.75      ]]
```

Solution. We iterate through the rows (first index) in the list, and for each row, we iterate through the column values (second index) and write each value to the file. At the end of each row, we must insert a newline character in the file to get a linebreak. The code resides in the file `write1.py`:

```
data = [[ 0.75,        0.29619813, -0.29619813, -0.75      ],
        [ 0.29619813,  0.11697778, -0.11697778, -0.29619813],
        [-0.29619813, -0.11697778,  0.11697778,  0.29619813],
        [-0.75,       -0.29619813,  0.29619813,  0.75      ]]

outfile = open('tmp_table.dat', 'w')
for row in data:
    for column in row:
        outfile.write('%14.8f' % column)
    outfile.write('\n')
outfile.close()
```

The resulting data file becomes

```
    0.75000000    0.29619813   -0.29619813   -0.75000000
    0.29619813    0.11697778   -0.11697778   -0.29619813
   -0.29619813   -0.11697778    0.11697778    0.29619813
   -0.75000000   -0.29619813    0.29619813    0.75000000
```

An extension of this program consists in adding column and row headings:

```
              column  1     column  2     column  3     column  4
row  1    0.75000000    0.29619813   -0.29619813   -0.75000000
row  2    0.29619813    0.11697778   -0.11697778   -0.29619813
row  3   -0.29619813   -0.11697778    0.11697778    0.29619813
row  4   -0.75000000   -0.29619813    0.29619813    0.75000000
```

To obtain this end result, we need to the add some statements to the program `write1.py`. For the column headings we must know the number of columns, i.e., the length of the rows, and loop from 1 to this length:

```
ncolumns = len(data[0])
outfile.write('         ')
for i in range(1, ncolumns+1):
    outfile.write('%10s    ' % ('column %2d' % i))
outfile.write('\n')
```

Note the use of a nested printf construction: the text we want to insert is itself a printf string. We could also have written the text as `'column ' + str(i)`, but then the length of the resulting string would depend on the number of digits in `i`. It is recommended to always use printf constructions for a tabular output format, because this gives automatic padding of blanks so that the width of the output strings remains the same. The tuning of the widths is commonly done in a trial-and-error process.

To add the row headings, we need a counter over the row numbers:

```
row_counter = 1
for row in data:
    outfile.write('row %2d' % row_counter)
    for column in row:
        outfile.write('%14.8f' % column)
    outfile.write('\n')
    row_counter += 1
```

The complete code is found in the file `write2.py`. We could, alternatively, iterate over the indices in the list:

```
for i in range(len(data)):
    outfile.write('row %2d' % (i+1))
    for j in range(len(data[i])):
        outfile.write('%14.8f' % data[i][j])
    outfile.write('\n')
```

4.6.2 Standard input and output as file objects

Reading user input from the keyboard applies the function `raw_input` as explained in Section 4.1. The keyboard is a medium that the computer in fact treats as a file, referred to as *standard input*.

The `print` command prints text in the terminal window. This medium is also viewed as a file from the computer's point of view and called *standard output*. All general-purpose programming languages allow reading from standard input and writing to standard output. This reading and writing can be done with two types of tools, either file-like objects or special tools like `raw_input` and `print` in Python. We will here describe the file-line objects: `sys.stdin` for standard input and `sys.stdout` for standard output. These objects behave as file objects, except that they do not need to be opened or closed. The statement

```
s = raw_input('Give s:')
```

is equivalent to

```
print 'Give s: ',
s = sys.stdin.readline()
```

Recall that the trailing comma in the `print` statement avoids the newline that `print` by default adds to the output string. Similarly,

```
s = eval(raw_input('Give s:'))
```

is equivalent to

```
print 'Give s: ',
s = eval(sys.stdin.readline())
```

For output to the terminal window, the statement

```
print s
```

is equivalent to

```
sys.stdout.write(s + '\n')
```

Why it is handy to have access to standard input and output as file objects can be illustrated by an example. Suppose you have a function that reads data from a file object `infile` and writes data to a file object `outfile`. A sample function may take the form

```
def x2f(infile, outfile, f):
    for line in infile:
        x = float(line)
        y = f(x)
        outfile.write('%g\n' % y)
```

This function works with all types of files, including web pages as `infile` (see Section 6.3). With `sys.stdin` as `infile` and/or `sys.stdout` as `outfile`, the `x2f` function also works with standard input and/or standard output. Without `sys.stdin` and `sys.stdout`, we would need different code, employing `raw_input` and `print`, to deal with standard input and output. Now we can write a single function that deals with all file media in a unified way.

There is also something called *standard error*. Usually this is the terminal window, just as standard output, but programs can distinguish between writing ordinary output to standard output and error messages to standard error, and these output media can be redirected to, e.g., files such that one can separate error messages from ordinary output. In Python, standard error is the file-like object `sys.stderr`. A typical application of `sys.stderr` is to report errors:

```
if x < 0:
    sys.stderr.write('Illegal value of x'); sys.exit(1)
```

This message to `sys.stderr` is an alternative to `print` or raising an exception.

Redirecting standard input, output, and error. Standard output from a program `prog` can be redirected to a file `output` instead of the screen, by using the greater than sign:

Terminal

```
Terminal> prog > output
```

Here, `prog` can be any program, including a Python program run as `python myprog.py`. Similarly, output to the medium called *standard error* can be redirected by

Terminal

```
Terminal> prog &> output
```

For example, error messages are normally written to standard error, which is exemplified in this little terminal session on a Unix machine:

Terminal

```
Terminal> ls bla-bla1 bla-bla2
ls: cannot access bla-bla1: No such file or directory
ls: cannot access bla-bla2: No such file or directory
Terminal> ls bla-bla1 bla-bla2 &> errors
Terminal> cat errors  # print the file errors
ls: cannot access bla-bla1: No such file or directory
ls: cannot access bla-bla2: No such file or directory
```

When the program reads from standard input (the keyboard), we can equally well redirect standard input from a file, say with name `input`, such that the program reads from this file rather than from the keyboard:

Terminal

```
Terminal> prog < input
```

Combinations are also possible:

Terminal

```
Terminal> prog < input > output
```

Note. The redirection of standard output, input, and error does not work for Python programs executed with the `run` command inside IPython, only when executed directly in the operating system in a terminal window, or with the same command prefixed with an exclamation mark in IPython.

Inside a Python program we can also let standard input, output, and error work with ordinary files instead. Here is the technique:

```
sys_stdout_orig = sys.stdout
sys.stdout = open('output', 'w')
sys_stdin_orig = sys.stdin
sys.stdin = open('input', 'r')
```

Now, any `print` statement will write to the `output` file, and any `raw_input` call will read from the `input` file. (Without storing the original `sys.stdout` and `sys.stdin` objects in new variables, these objects would get lost in the redefinition above and we would never be able to reach the common standard input and output in the program.)

4.6.3 What is a file, really?

This section is not mandatory for understanding the rest of the book. Nevertheless, the information here is fundamental for understanding what files are about.

A file is simply a sequence of characters. In addition to the sequence of characters, a file has some data associated with it, typically the name of the file, its location on the disk, and the file size. These data are stored somewhere by the operating system. Without this extra information beyond the pure file contents as a sequence of characters, the operating system cannot find a file with a given name on the disk.

Each character in the file is represented as a *byte*, consisting of eight *bits*. Each bit is either 0 or 1. The zeros and ones in a byte can be combined in $2^8 = 256$ ways. This means that there are 256 different types of characters. Some of these characters can be recognized from the keyboard, but there are also characters that do not have a familiar symbol. Such characters looks cryptic when printed.

Pure text files. To see that a file is really just a sequence of characters, invoke an editor for plain text, typically the editor you use to write Python programs. Write the four characters `ABCD` into the editor, do not press the Return key, and save the text to a file `test1.txt`. Use your favorite tool for file and folder overview and move to the folder containing the `test1.txt` file. This tool may be Windows Explorer, My Computer, or a DOS window on Windows; a terminal window, Konqueror, or Nautilus on Linux; or a terminal window or Finder on Mac. If you choose a terminal window, use the `cd` (change directory) command to move to the proper folder and write `dir` (Windows) or `ls -l` (Linux/Mac) to list the files and their sizes. In a graphical program like Windows Explorer, Konqueror, Nautilus, or Finder, select a view that shows the *size* of each file (choose *view as details* in Windows Explorer, *View as List* in Nautilus, the list view icon in Finder, or you just point at a file icon in Konqueror and watch the pop-up text). You will see that the `test1.txt` file has a size of 4 bytes (if you use `ls -l`, the size measured in bytes is found in column 5, right before the date). The 4 bytes are exactly the 4 characters `ABCD` in the file. Physically, the file is just a sequence of 4 bytes on your hard disk.

Go back to the editor again and add a newline by pressing the Return key. Save this new version of the file as `test2.txt`. When you now check the size of the file it has grown to five bytes. The reason is that we added a newline character (symbolically known as *backslash n*: `\n`).

Instead of examining files via editors and folder viewers we may use Python interactively:

```
>>> file1 = open('test1.txt', 'r').read()  # read file into string
>>> file1
'ABCD'
>>> len(file1)        # length of string in bytes/characters
4
>>> file2 = open('test2.txt', 'r').read()
>>> file2
'ABCD\n'
>>> len(file2)
5
```

Python has in fact a function that returns the size of a file directly:

```
>>> import os
>>> size = os.path.getsize('test1.txt')
>>> size
4
```

Word processor files. Most computer users write text in a word processing program, such as Microsoft Word or LibreOffice. Let us investigate what happens with our four characters `ABCD` in such a program. Start the word processor, open a new document, and type in the four characters `ABCD` only. Save the document as a `.docx` file (Microsoft Word) or an `.odt` file (LibreOffice). Load this file into an editor for pure text and look at the contents. You will see that there are numerous strange characters that you did not write (!). This additional "text" contains information on what type of document this is, the font you used, etc. The LibreOffice version of this file has 8858 bytes and the Microsoft Word version contains over 26 Kb! However, if you save the file as a pure text file, with extension `.txt`, the size is down to 8 bytes in LibreOffice and five in Microsoft Word.

Instead of loading the LibreOffice file into an editor we can again read the file contents into a string in Python and examine this string:

```
>>> infile = open('test3.odt', 'r')  # open LibreOffice file
>>> s = infile.read()
>>> len(s)   # file size
8858
>>> s
'PK\x03\x04\x14\x00\x00\x08\x00\x00sKWD^\xc62\x0c\'\x00...
\x00meta.xml<?xml version="1.0" encoding="UTF-8"?>\n<office:...
" xmlns:meta="urn:oasis:names:tc:opendocument:xmlns:meta:1.0"
```

Each backslash followed by `x` and a number is a code for a special character not found on the keyboard (recall that there are 256 characters and only a subset is associated with keyboard symbols). Although we show just a small portion of all the characters in this file in the above output (otherwise, the output would have occupied several pages in this book with thousands symbols like `\x04`...), we can guarantee that you cannot find the pure sequence of characters `ABCD`. However, the computer program that generated the file, LibreOffice in this example, can easily interpret the meaning of all the characters in the file and translate the information into nice, readable text on the screen where you can recognize the text `ABCD`.

Your are now in a position to look into Exercise 4.8 to see what happens if one attempts to use LibreOffice to write Python programs.

Image files. A digital image - captured by a digital camera or a mobile phone - is a file. And since it is a file, the image is just a sequence of characters. Loading some JPEG file into a pure text editor, reveals all the strange characters in there. On the first line you will (normally) find some recognizable text in between the strange characters. This text reflects the type of camera used to capture the image and the date and time when the picture was taken. The next lines contain more information about the image. Thereafter, the file contains a set of numbers representing the image. The basic representation of an image is a set of $m \times n$ pixels, where each pixel has a color represented as a combination of 256 values of red, green, and blue, which can be stored as three bytes (resulting in 256^3 color values). A 6-megapixel camera will then need to store $3 \times 6 \cdot 10^6 = 18$ megabytes for one picture. The JPEG file contains only a couple of megabytes. The reason is that JPEG is a *compressed* file format, produced by applying a smart technique that can throw away pixel information in the original picture such that the human eye hardly can detect the inferior quality.

A video is just a sequence of images, and therefore a video is also a stream of bytes. If the change from one video frame (image) to the next is small, one can use smart methods to compress the image information in time. Such compression is particularly important for videos since the file sizes soon get too large for being transferred over the Internet. A small video file occasionally has bad visual quality, caused by too much compression.

Music files. An MP3 file is much like a JPEG file: first, there is some information about the music (artist, title, album, etc.), and then comes the music itself as a stream of bytes. A typical MP3 file has a size of something like five million bytes or five megabytes (5 Mb). The exact size depends on the complexity of the music, the length of the track, and the MP3 resolution. On a 16 Gb MP3 player you can then store roughly $16,000,000,000/5,000,000 = 3200$ MP3 files. MP3 is, like JPEG, a compressed format. The complete data of a song on a CD (the WAV file) contains about ten times as many bytes. As for pictures, the idea is that one can throw away a lot of bytes in an intelligent way, such that the human ear hardly detects the difference between a compressed and uncompressed version of the music file.

PDF files. Looking at a PDF file in a pure text editor shows that the file contains some readable text mixed with some unreadable characters. It is not possible for a human to look at the stream of bytes and deduce the text in the document (well, from the assumption that there are always some strange people doing strange things, there might be somebody out there who, with a lot of training, can interpret the pure PDF code with the eyes). A PDF file reader can easily interpret the contents of the file and display the text in a human-readable form on the screen.

Remarks. We have repeated many times that a file is just a stream of bytes. A human can interpret (read) the stream of bytes if it makes sense in a human language - or a computer language (provided the human is a programmer). When the series of bytes does not make sense to any human, a computer program must be used to interpret the sequence of characters.

Think of a report. When you write the report as pure text in a text editor, the resulting file contains just the characters you typed in from the keyboard. On the other hand, if you applied a word processor like Microsoft Word or LibreOffice, the report file contains a large number of extra bytes describing properties of the formatting of the text. This stream of extra bytes does not make sense to a human, and a computer program is required to interpret the file content and display it in a form that a human can understand. Behind the sequence of bytes in the file there are strict rules telling what the series of bytes means. These rules reflect the *file format*. When the rules or file format is publicly documented, a programmer can use this documentation to make her own program for interpreting the file contents (however, interpreting such files is much more complicated than our examples on reading human-readable files in this book). It happens, though, that secret file formats are used, which require certain programs from certain companies to interpret the files.

4.7 Handling errors

Suppose we forget to provide a command-line argument to the `c2f_cml.py` program from Section 4.2.1:

Terminal
```
c2f_cml.py
Traceback (most recent call last):
  File "c2f_cml.py", line 2, in ?
    C = float(sys.argv[1])
IndexError: list index out of range
```

Python aborts the program and shows an error message containing the line where the error occurred, the type of the error (`IndexError`), and a quick explanation of what the error is. From this information we deduce that the index 1 is out of range. Because there are no command-line arguments in this case, `sys.argv` has only one element, namely the program name. The only valid index is then 0.

For an experienced Python programmer this error message will normally be clear enough to indicate what is wrong. For others it would be very helpful if wrong usage could be detected by our program and a description of correct operation could be printed. The question is how to detect the error inside the program.

The problem in our sample execution is that `sys.argv` does not contain two elements (the program name, as always, plus one command-line argument). We can therefore test on the length of `sys.argv` to detect wrong usage: if `len(sys.argv)` is less than 2, the user failed to provide information on the `C` value. The new version of the program, `c2f_cml_if.py`, starts with this `if` test:

```
if len(sys.argv) < 2:
    print 'You failed to provide Celsius degrees as input '\
          'on the command line!'
    sys.exit(1)  # abort because of error
F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

We use the `sys.exit` function to abort the program. Any argument different from zero signifies that the program was aborted due to an error, but the precise value of the argument does not matter so here we simply choose it to be `1`. If no errors are found, but we still want to abort the program, `sys.exit(0)` is used.

A more modern and flexible way of handling potential errors in a program is to *try* to execute some statements, and if something goes wrong, the program can detect this and jump to a set of statements that handle the erroneous situation as desired. The relevant program construction reads

```
try:
    <statements>
except:
    <statements>
```

If something goes wrong when executing the statements in the `try` block, Python raises what is known as an *exception*. The execution jumps directly to the `except` block whose statements can provide a remedy for the error. The next section explains the `try-except` construction in more detail through examples.

4.7.1 Exception handling

To clarify the idea of exception handling, let us use a `try-except` block to handle the potential problem arising when our Celsius-Fahrenheit conversion program lacks a command-line argument:

```
import sys
try:
    C = float(sys.argv[1])
except:
    print 'You failed to provide Celsius degrees as input '\
          'on the command line!'
    sys.exit(1)  # abort
F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

The program is stored in the file `c2f_cml_except1.py`. If the command-line argument is missing, the indexing `sys.argv[1]`, which has an invalid index `1`, *raises an exception*. This means that the program jumps directly to the `except` block, implying that `float` is not called, and `C` is not initialized with a value. In the except block, the programmer can retrieve information about the exception and perform statements to recover from the error. In our example, we know what the error can be, and therefore we just print a message and abort the program.

Suppose the user provides a command-line argument. Now, the `try` block is executed successfully, and the program neglects the `except` block and continues with the Fahrenheit conversion. We can try out the last program in two cases:

Terminal

```
c2f_cml_except1.py
You failed to provide Celsius degrees as input on the command line!

c2f_cml_except1.py 21
21C is 69.8F
```

In the first case, the illegal index in `sys.argv[1]` causes an exception to be raised, and we perform the steps in the `except` block. In the second case, the `try` block executes successfully, so we jump over the `except` block and continue with the computations and the printout of results.

For a user of the program, it does not matter if the programmer applies an `if` test or exception handling to recover from a missing command-line argument. Nevertheless, exception handling is considered a better programming solution because it allows more advanced ways to abort or continue the execution. Therefore, we adopt exception handling as our standard way of dealing with errors in the rest of this book.

Testing for a specific exception. Consider the assignment

```
C = float(sys.argv[1])
```

There are two typical errors associated with this statement: i) `sys.argv[1]` is illegal indexing because no command-line arguments are provided, and ii) the content in the string `sys.argv[1]` is not a pure number that can be converted to a `float` object. Python detects both these errors and raises an `IndexError` exception in the first case and a `ValueError` in the second. In the program above, we jump to the `except` block and issue the same message regardless of what went wrong in the `try` block. For example, when we indeed provide a command-line argument, but write it on an illegal form (`21C`), the program jumps to the `except` block and prints a misleading message:

Terminal

```
c2f_cml_except1.py 21C
You failed to provide Celsius degrees as input on the command line!
```

The solution to this problem is to branch into different `except` blocks depending on what type of exception that was raised in the `try` block (program `c2f_cml_except2.py`):

```
import sys
try:
    C = float(sys.argv[1])
except IndexError:
    print 'Celsius degrees must be supplied on the command line'
    sys.exit(1)  # abort execution
except ValueError:
    print 'Celsius degrees must be a pure number, '\
          'not "%s"' % sys.argv[1]
    sys.exit(1)

F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

Now, if we fail to provide a command-line argument, an `IndexError` occurs and we tell the user to write the `C` value on the command line. On the other hand, if the `float` conversion fails, because the command-line argument has wrong syntax, a `ValueError` exception is raised and we branch into the second `except` block and explain that the form of the given number is wrong:

Terminal

```
c2f_cml_except1.py 21C
Celsius degrees must be a pure number, not "21C"
```

Examples on exception types. List indices out of range lead to `IndexError` exceptions:

```
>>> data = [1.0/i for i in range(1,10)]
>>> data[9]
...
IndexError: list index out of range
```

Some programming languages (Fortran, C, C++, and Perl are examples) allow list indices outside the legal index values, and such unnoticed errors can be hard to find. Python always stops a program when an invalid index is encountered, unless you handle the exception explicitly as a programmer.

Converting a string to `float` is unsuccessful and gives a `ValueError` if the string is not a pure integer or real number:

```
>>> C = float('21 C')
...
ValueError: invalid literal for float(): 21 C
```

Trying to use a variable that is not initialized gives a `NameError` exception:

```
>>> print a
...
NameError: name 'a' is not defined
```

Division by zero raises a `ZeroDivisionError` exception:

```
>>> 3.0/0
...
ZeroDivisionError: float division
```

Writing a Python keyword illegally or performing a Python grammar error leads to a `SyntaxError` exception:

```
>>> forr d in data:
...
    forr d in data:
         ^
SyntaxError: invalid syntax
```

What if we try to multiply a string by a number?

```
>>> 'a string'*3.14
...
TypeError: can't multiply sequence by non-int of type 'float'
```

The `TypeError` exception is raised because the object types involved in the multiplication are wrong (`str` and `float`).

Digression. It might come as a surprise, but multiplication of a string and a number is legal if the number is an integer. The multiplication means that the string should be repeated the specified number of times. The same rule also applies to lists:

```
>>> '--'*10   # ten double dashes = 20 dashes
'--------------------'
>>> n = 4
>>> [1, 2, 3]*n
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> [0]*n
[0, 0, 0, 0]
```

The latter construction is handy when we want to create a list of `n` elements and later assign specific values to each element in a `for` loop.

4.7.2 Raising exceptions

When an error occurs in your program, you may either print a message and use `sys.exit(1)` to abort the program, or you may raise an exception. The latter task is easy. You just write `raise E(message)`, where `E` can be a known exception type in Python and `message` is a string explaining what is wrong. Most often `E` means `ValueError` if the value of some variable is illegal, or `TypeError` if the type of a variable is wrong. You can also define your own exception types. An exception can be raised from any location in a program.

Example. In the program `c2f_cml_except2.py` from Section 4.7.1 we show how we can test for different exceptions and abort the program. Sometimes we see that an exception may happen, but if it happens, we want a more precise error message to help the user. This can be done by raising a new exception in an `except` block and provide the desired exception type and message.

Another application of raising exceptions with tailored error messages arises when input data are invalid. The code below illustrates how to raise exceptions in various cases.

We collect the reading of `C` and handling of errors a separate function:

```
def read_C():
    try:
        C = float(sys.argv[1])
    except IndexError:
        raise IndexError\
        ('Celsius degrees must be supplied on the command line')
    except ValueError:
        raise ValueError\
        ('Celsius degrees must be a pure number, '\
        'not "%s"' % sys.argv[1])
    # C is read correctly as a number, but can have wrong value:
    if C < -273.15:
        raise ValueError('C=%g is a non-physical value!' % C)
    return C
```

There are two ways of using the `read_C` function. The simplest is to call the function,

```
C = read_C()
```

Wrong input will now lead to a raw dump of exceptions, e.g.,

Terminal

```
c2f_cml_v5.py
Traceback (most recent call last):
  File "c2f_cml4.py", line 5, in ?
    raise IndexError\
IndexError: Celsius degrees must be supplied on the command line
```

New users of this program may become uncertain when getting raw output from exceptions, because words like `Traceback`, `raise`, and `IndexError` do not make much sense unless you have some experience with Python. A more user-friendly output can be obtained by calling the `read_C` function inside a `try-except` block, check for any exception (or better: check for `IndexError` *or* `ValueError`), and write out the exception message in a more nicely formatted form. In this way, the programmer takes complete control of how the program behaves when errors are encountered:

```
try:
    C = read_C()
except Exception as e:
    print e             # exception message
    sys.exit(1)         # terminate execution
```

`Exception` is the parent name of all exceptions, and `e` is an exception object. Nice printout of the exception message follows from a straight `print e`. Instead of `Exception` we can write `(ValueError, IndexError)` to test more specifically for two exception types we can expect from the `read_C` function:

```
try:
    C = read_C()
except (ValueError, IndexError) as e:
    print e             # exception message
    sys.exit(1)         # terminate execution
```

After the `try-except` block above, we can continue with computing `F = 9*C/5 + 32` and print out `F`. The complete program is found in the file `c2f_cml.py`. We may now test the program's behavior when the input is wrong and right:

Terminal

```
c2f_cml.py
Celsius degrees must be supplied on the command line

c2f_cml.py 21C
Celsius degrees must be a pure number, not "21C"

c2f_cml.py -500
C=-500 is a non-physical value!

c2f_cml.py 21
21C is 69.8F
```

This program deals with wrong input, writes an informative message, and terminates the execution without annoying behavior.

Scattered `if` tests with `sys.exit` calls are considered a bad programming style compared to the use of nested exception handling as illustrated above. You should abort execution in the main program only, not inside functions. The reason is that the functions can be re-used in other occasions where the error can be dealt with differently. For instance, one may avoid abortion by using some suitable default data.

The programming style illustrated above is considered the best way of dealing with errors, so we suggest that you hereafter apply exceptions for handling potential errors in the programs you make, simply because this is what experienced programmers expect from your codes.

4.8 A glimpse of graphical user interfaces

Maybe you find it somewhat strange that the usage of the programs we have made so far in this book - and the programs we will make in the rest of the book - are less graphical and intuitive than the computer programs you are used to from school or entertainment. Those programs are operated through some self-explaining graphics, and most of the things you want to do involve pointing with the mouse, clicking on graphical elements on the screen, and maybe filling in some text fields. The programs in this book, on the other hand, are run from the command line in a terminal window or inside IPython, and input is also given here in form of plain text.

The reason why we do not equip the programs in this book with graphical interfaces for providing input, is that such graphics is both complicated and tedious to write. If the aim is to solve problems from mathematics and science, we think it is better to focus on this part rather than large amounts of code that merely offers some "expected" graphical cosmetics for putting data into the program. Textual input from the command line is also quicker to provide. Also remember that the computational functionality of a program is obviously independent from the type of user interface, textual or graphic.

As an illustration, we shall now show a Celsius to Fahrenheit conversion program with a graphical user interface (often called a GUI). The GUI is shown in Figure 4.1. We encourage you to try out the graphical interface - the name of the program is `c2f_gui.py`. The complete program text is listed below.

```
from Tkinter import *
root = Tk()
C_entry = Entry(root, width=4)
C_entry.pack(side='left')
Cunit_label = Label(root, text='Celsius')
Cunit_label.pack(side='left')
```

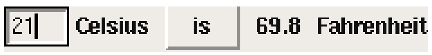


Fig. 4.1 Screen dump of the graphical interface for a Celsius to Fahrenheit conversion program. The user can type in the temperature in Celsius degrees, and when clicking on the *is* button, the corresponding Fahrenheit value is displayed.

```
def compute():
    C = float(C_entry.get())
    F = (9./5)*C + 32
    F_label.configure(text='%g' % F)

compute = Button(root, text=' is ', command=compute)
compute.pack(side='left', padx=4)

F_label = Label(root, width=4)
F_label.pack(side='left')
Funit_label = Label(root, text='Fahrenheit')
Funit_label.pack(side='left')

root.mainloop()
```

The goal of the forthcoming dissection of this program is to give a taste of how graphical user interfaces are coded. The aim is not to equip you with knowledge on how you can make such programs on your own.

A GUI is built of many small graphical elements, called *widgets*. The graphical window generated by the program above and shown in Figure 4.1 has five such widgets. To the left there is an *entry* widget where the user can write in text. To the right of this entry widget is a *label* widget, which just displays some text, here "Celsius". Then we have a *button* widget, which when being clicked leads to computations in the program. The result of these computations is displayed as text in a *label* widget to the right of the button widget. Finally, to the right of this result text we have another *label* widget displaying the text "Fahrenheit". The program must construct each widget and pack it correctly into the complete window. In the present case, all widgets are packed from left to right.

The first statement in the program imports functionality from the GUI toolkit `Tkinter` to construct widgets. First, we need to make a root widget that holds the complete window with all the other widgets. This root widget is of type `Tk`. The first entry widget is then made and referred to by a variable `C_entry`. This widget is an object of type `Entry`, provided by the `Tkinter` module. Widgets constructions follow the syntax

```
variable_name = Widget_type(parent_widget, option1, option2, ...)
variable_name.pack(side='left')
```

When creating a widget, we must bind it to a *parent widget*, which is the graphical element in which this new widget is to be packed. Our widgets in the present program have the `root` widget as parent widget. Various widgets have different types of options that we can set. For example, the `Entry` widget has a possibility for setting the width of the text field, here `width=4` means that the text field is 4 characters wide. The pack statement is important to remember - without it, the widget remains invisible.

The other widgets are constructed in similar ways. The next fundamental feature of our program is how computations are tied to the event of clicking the button *is*. The `Button` widget has naturally a text, but more important, it binds the button to a function `compute` through the `command=compute` option. This means that when the user clicks the button *is*, the function `compute` is called. Inside the `compute` function we first fetch the Celsius value from the `C_entry` widget, using this widget's `get` function, then we transform this string (everything typed in by the user is interpreted as text and stored in strings) to a `float` before we compute the corresponding Fahrenheit value. Finally, we can update (`configure`) the text in the `Label` widget `F_label` with a new text, namely the computed degrees in Fahrenheit.

A program with a GUI behaves differently from the programs we construct in this book. First, all the statements are executed from top to bottom, as in all our other programs, but these statements just construct the GUI and define functions. No computations are performed. Then the program enters a so-called *event loop*: `root.mainloop()`. This is an infinite loop that "listens" to user events, such as moving the mouse, clicking the mouse, typing characters on the keyboard, etc. When an event is recorded, the program starts performing associated actions. In the present case, the program waits for only one event: clicking the button *is*. As soon as we click on the button, the `compute` function is called and the program starts doing mathematical work. The GUI will appear on the screen until we destroy the window by click on the X up in the corner of the window decoration. More complicated GUIs will normally have a special *Quit* button to terminate the event loop.

In all GUI programs, we must first create a hierarchy of widgets to build up all elements of the user interface. Then the program enters an event loop and waits for user events. Lots of such events are registered as actions in the program when creating the widgets, so when the user clicks on buttons, move the mouse into certain areas, etc., functions in the program are called and "things happen".

Many books explain how to make GUIs in Python programs, see for instance [5, 7, 14, 17].

4.9 Making modules

Sometimes you want to reuse a function from an old program in a new program. The simplest way to do this is to copy and paste the old source code into the new program. However, this is not good programming practice, because you then over time end up with multiple identical versions of the same function. When you want to improve the function or correct a bug, you need to remember to do the same update in all files with a copy of the function, and in real life most programmers fail to do so. You easily end up with a mess of different versions with different quality of basically the same code. Therefore, a golden rule of programming is to have one and only one version of a piece of code. All programs that want to use this piece of code must access one and only one place where the source code is kept. This principle is easy to implement if we create a module containing the code we want to reuse later in different programs.

When reading this, you probably know how to use a ready-made module. For example, if you want to compute the factorial $k! = k(k-1)(k-2)\cdots 1$, there is a function `factorial` in Python's `math` module that can be help us out. The usage goes with the `math` prefix,

```
import math
value = math.factorial(5)
```

or without,

```
from math import factorial
# or: from math import *
value = factorial(5)
```

Now you shall learn how to make your own Python modules. There is hardly anything to learn, because you just collect all the functions that constitute the module in one file, say with name `mymodule.py`. This file is automatically a module, with name `mymodule`, and you can import functions from this module in the standard way. Let us make everything clear in detail by looking at an example.

4.9.1 Example: Interest on bank deposits

The classical formula for the growth of money in a bank reads

$$A = A_0\left(1 + \frac{p}{360\cdot 100}\right)^n, \tag{4.2}$$

where A_0 is the initial amount of money, and A is the present amount after n days with p percent annual interest rate. (The formula applies the convention that the rate per day is computed as $p/360$, while n counts the actual number of days the money is in the bank, see the Wikipedia entry Day count convention[4] for explanation. There is a handy Python module `datetime` for computing the number of days between two dates.)

Equation (4.2) involves four parameters: A, A_0, p, and n. We may solve for any of these, given the other three:

[4] `http://en.wikipedia.org/wiki/Day_count_convention`

$$A_0 = A\left(1+\frac{p}{360\cdot 100}\right)^{-n}, \tag{4.3}$$

$$n = \frac{\ln \frac{A}{A_0}}{\ln\left(1+\frac{p}{360\cdot 100}\right)}, \tag{4.4}$$

$$p = 360\cdot 100\left(\left(\frac{A}{A_0}\right)^{1/n} - 1\right). \tag{4.5}$$

Suppose we have implemented (4.2)-(4.5) in four functions:

```
from math import log as ln

def present_amount(A0, p, n):
    return A0*(1 + p/(360.0*100))**n

def initial_amount(A, p, n):
    return A*(1 + p/(360.0*100))**(-n)

def days(A0, A, p):
    return ln(A/A0)/ln(1 + p/(360.0*100))

def annual_rate(A0, A, n):
    return 360*100*((A/A0)**(1.0/n) - 1)
```

We want to make these functions available in a module, say with name `interest`, so that we can import functions and compute with them in a program. For example,

```
from interest import days
A0 = 1; A = 2; p = 5
n = days(A0, 2, p)
years = n/365.0
print 'Money has doubled after %.1f years' % years
```

How to make the `interest` module is described next.

4.9.2 Collecting functions in a module file

To make a module of the four functions `present_amount`, `initial_amount`, `days`, and `annual_rate`, we simply open an empty file in a text editor and copy the program code for all the four functions over to this file. This file is then automatically a Python module provided we save the file under any valid filename. The extension must be `.py`, but the module name is only the base part of the filename. In our case, the filename `interest.py` implies a module name `interest`. To use the `annual_rate` function in another program we simply write, in that program file,

```
from interest import annual_rate
```

or we can write

```
from interest import *
```

to import all four functions, or we can write

```
import interest
```

and access individual functions as `interest.annual_rate` and so forth.

4.9.3 Test block

It is recommended to only have functions and not any statements outside functions in a module. The reason is that the module file is executed from top to bottom during the import. With function definitions only in the module file, and no main program, there will be no calculations or output from the import, just definitions of functions. This is the desirable behavior. However, it is often convenient to have test or demonstrations in the module file, and then there is need for a main program. Python allows a very fortunate construction to let the file act both as a module with function definitions only (and no main program) *and* as an ordinary program we can run, with functions and a main program.

This two-fold “magic” is realized by putting the main program after an `if` test of the form

```
if __name__ == '__main__':
    <block of statements>
```

The `__name__` variable is automatically defined in any module and equals the module name if the module file is imported in another program, or `__name__` equals the string `'__main__'` if the module file is run as a program. This implies that the `<block of statements>` part is executed if and only if we run the module file as a program. We shall refer to `<block of statements>` as the *test block* of a module.

Example on a test block in a minimalistic module. A very simple example will illustrate how this works. Consider a file `mymod.py` with the content

```
def add1(x):
    return x + 1

if __name__ == '__main__':
    print 'run as program'
    print add1(float(sys.argv[1]))
```

We can import `mymod` as a module and make use of the `add1` function:

```
>>> import mymod
>>> print mymod.add1(4)
5
```

During the import, the `if` test is false, and the only the function definition is executed. However, if we run `mymod.py` as a program,

Terminal
```
mymod.py 5
run as program
6
```

the `if` test becomes true, and the `print` statements are executed.

Tip on easy creation of a module

If you have some functions and a main program in some program file, just move the main program to the test block. Then the file can act as a module, giving access to all the functions in other files, or the file can be executed from the command line, in the same way as the original program.

A test block in the `interest` module. Let us write a little main program for demonstrating the `interest` module in a test block. We read *p* from the command line and write out how many years it takes to double an amount with that interest rate:

```
if __name__ == '__main__':
    import sys
    p = float(sys.argv[1])
    years = days(1, 2, p)
    print 'With p=%.2f it takes %.1 years to double' % (p, years)
```

Running the module file as a program gives this output:

Terminal
```
interest.py 2.45
With p=2.45 it takes 27.9 years to double
```

To test that the `interest.py` file also works as a module, invoke a Python shell and try to import a function and compute with it:

```
>>> from interest import present_amount
>>> present_amount(2, 5, 730)
2.2133983053266699
```

We have hence demonstrated that the file `interest.py` works both as a program and as a module.

Recommended practice in a test block

It is a good programming habit to let the test block do one or more of three things:

- provide information on how the module or program is used,
- test if the module functions work properly,
- offer interaction with users such that the module file can be applied as a useful program.

Instead of having a lot of statements in the test block, it is better to collect the statements in separate functions, which then are called from the test block.

4.9.4 Verification of the module code

Functions that verify the implementation in a module should

- have names starting with `test_`,
- express the success or failure of a test through a boolean variable, say `success`,
- run `assert success, msg` to raise an `AssertionError` with an optional message `msg` in case the test fails.

Adopting this style makes it trivial to let the tools *pytest* or *nose* automatically run through all our `test_*()` functions in all files in a folder tree. A very brief introduction to test functions compatible with pytest and nose is provided in Section 3.4.2, while Section H.6 contains a more thorough introduction to the pytest and nose testing frameworks for beginners.

A proper test function for verifying the functionality of the `interest` module, written in a way that is compatible with the pytest and nose testing frameworks, looks as follows:

```
def test_all_functions():
    # Compatible values
    A = 2.2133983053266699; A0 = 2.0; p = 5; n = 730
    # Given three of these, compute the remaining one
    # and compare with the correct value (in parenthesis)
    A_computed  = present_amount(A0, p, n)
    A0_computed = initial_amount(A, p, n)
    n_computed  = days(A0, A, p)
    p_computed  = annual_rate(A0, A, n)

    def float_eq(a, b, tolerance=1E-14):
        """Return True if a == b within the tolerance."""
        return abs(a - b) < tolerance

    success = float_eq(A_computed,  A)  and \
              float_eq(A0_computed, A0) and \
              float_eq(p_computed,  p)  and \
              float_eq(n_computed,  n)
    msg = """Computations failed (correct answers in parenthesis):
A=%g (%g)
A0=%g (%.1f)
n=%d (%d)
p=%g (%.1f)""" % (A_computed, A, A0_computed, A0,
                  n_computed, n, p_computed, p)
    assert success, msg
```

We may require a single command-line argument `test` to run the verification. The test block can then be expressed as

```
if __name__ == '__main__':
    if len(sys.argv) == 2 and sys.argv[1] == 'test':
        test_all_functions()
```

4.9.5 Getting input data

To make a useful program, we should allow setting three parameters on the command line and let the program compute the remaining parameter. For example, running the program as

```
Terminal
interest.py A0=2 A=1 n=1095
```

will lead to a computation of p, in this case for seeing the size of the annual interest rate if the amount is to be doubled after three years.

How can we achieve the desired functionality? Since variables are already introduced and "initialized" on the command line, we could grab this text and execute it as Python code, either as three different lines or with semicolon between each assignment. This is easy:

```
init_code = ''
for statement in sys.argv[1:]:
    init_code += statement + '\n'
exec(init_code)
```

(We remark that an experienced Python programmer would have created `init_code` by `'\n'.join(sys.argv[1:])`.) For the sample run above with `A0=2 A=1 n=1095` on the command line, `init_code` becomes the string

```
A0=2
A=1
n=1095
```

Note that one cannot have spaces around the equal signs on the command line as this will break an assignment like `A0 = 2` into three command-line arguments, which will give rise to a `SyntaxError` in `exec(init_code)`. To tell the user about such errors, we execute `init_code` inside a `try-except` block:

```
try:
    exec(init_code)
except SyntaxError as e:
    print e
    print init_code
    sys.exit(1)
```

At this stage, our program has hopefully initialized three parameters in a successful way, and it remains to detect the remaining parameter to be computed. The following code does the work:

```
if 'A=' not in init_code:
    print 'A =', present_amount(A0, p, n)
elif 'A0=' not in init_code:
    print 'A0 =', initial_amount(A, p, n)
elif 'n=' not in init_code:
    print 'n =', days(A0, A , p)
elif 'p=' not in init_code:
    print 'p =', annual_rate(A0, A, n)
```

It may happen that the user of the program assigns value to a parameter with wrong name or forget a parameter. In those cases we call one of our four functions with uninitialized arguments, and Python raises an exception. Therefore, we should embed the code above in a `try-except` block. An uninitialized variable will lead to a `NameError` exception, while another frequent error is illegal values in the computations, leading to a `ValueError` exception. It is also a good habit to collect all the code related to computing the remaining, fourth parameter in a function for separating this piece of code from other parts of the module file:

```
def compute_missing_parameter(init_code):
    try:
        exec(init_code)
    except SyntaxError as e:
        print e
        print init_code
        sys.exit(1)
    # Find missing parameter
    try:
        if 'A=' not in init_code:
            print 'A =', present_amount(A0, p, n)
        elif 'A0=' not in init_code:
            print 'A0 =', initial_amount(A, p, n)
        elif 'n=' not in init_code:
            print 'n =', days(A0, A , p)
        elif 'p=' not in init_code:
            print 'p =', annual_rate(A0, A, n)
    except NameError as e:
        print e
        sys.exit(1)
    except ValueError:
        print 'Illegal values in input:', init_code
        sys.exit(1)
```

If the user of the program fails to give any command-line arguments, we print a usage statement. Otherwise, we run a verification if the first command-line argument is `test`, and else we run the missing parameter computation (i.e., the useful main program):

```
_filename = sys.argv[0]
_usage = """
Usage: %s A=10 p=5 n=730
Program computes and prints the 4th parameter'
(A, A0, p, or n)""" % _filename

if __name__ == '__main__':
    if len(sys.argv) == 1:
        print _usage
    elif len(sys.argv) == 2 and sys.argv[1] == 'test':
        test_all_functions()
    else:
        init_code = ''
        for statement in sys.argv[1:]:
            init_code += statement + '\n'
        compute_missing_parameter(init_code)
```

Executing user input can be dangerous

Some purists would never demonstrate `exec` the way we do above. The reason is that our program tries to execute whatever the user writes. Consider

Terminal

```
input.py 'import shutil; shutil.rmtree("/")'
```

This evil use of the program leads to an attempt to remove all files on the computer system (the same as writing `rm -rf /` in the terminal window!). However, for small private programs helping the program writer out with mathematical calculations, this potential dangerous misuse is not so much of a concern (the user just does harm to his own computer anyway).

4.9.6 Doc strings in modules

It is also a good habit to include a doc string in the beginning of the module file. This doc string explains the purpose and use of the module:

```
"""
Module for computing with interest rates.
Symbols: A is present amount, A0 is initial amount,
n counts days, and p is the interest rate per year.

Given three of these parameters, the fourth can be
computed as follows:

    A  = present_amount(A0, p, n)
    A0 = initial_amount(A, p, n)
    n  = days(A0, A, p)
    p  = annual_rate(A0, A, n)
"""
```

You can run the `pydoc` program to see a documentation of the new module, containing the doc string above and a list of the functions in the module: just write `pydoc interest` in a terminal window.

Now the reader is recommended to take a look at the actual file `interest.py` to see all elements of a good module file at once: doc strings, a set of functions, a test function, a function with the main program, a usage string, and a test block.

4.9.7 Using modules

Let us further demonstrate how to use the `interest.py` module in programs. For illustration purposes, we make a separate program file, say with name `doubling.py`, containing some computations:

```
from interest import days

# How many days does it take to double an amount when the
# interest rate is p=1,2,3,...14?
for p in range(1, 15):
    years = days(1, 2, p)/365.0
    print 'With p=%d%% it takes %.1f years to double the amount' %\
    (p, years)
```

What gets imported by various import statements? There are different ways to import functions in a module, and let us explore these in an interactive session. The function call `dir()` will list all names we have defined, including imported names of variables and functions. Calling `dir(m)` will print the names defined inside a module with name `m`. First we start an interactive shell and call `dir()`

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

These variables are always defined. Running the IPython shell will introduce several other standard variables too. Doing

```
>>> from interest import *
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__',
 'annual_rate', 'compute_missing_parameter', 'days',
 'initial_amount', 'ln', 'present_amount', 'sys',
 'test_all_functions']
```

shows that we get our four functions imported, along with `ln` and `sys`. The latter two are needed in the `interest` module, but not necessarily in our new program `doubling.py`.

The alternative `import interest` actually gives us access to more names in the module, namely also all variables and functions that start with an underscore:

```
>>> import interest
>>> dir(interest)
['__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '_filename', '_usage', 'annual_rate',
 'compute_missing_parameter', 'days', 'initial_amount',
 'ln', 'present_amount', 'sys', 'test_all_functions']
```

It is a habit to use an underscore for all variables that are not to be included in a `from interest import *` statement. These variables can, however, be reached through `interest._filename` and `interest._usage` in the present example.

It would be best that a statement `from interest import *` just imported the four functions doing the computations of general interest in other programs. This can be archived by deleting all unwanted names (among those without an initial underscore) at the very end of the module:

```
del sys, ln, compute_missing_parameter, test_all_functions
```

Instead of deleting variables and using initial underscores in names, it is in general better to specify the special variable `__all__`, which is used by Python to select functions to be imported in `from interest import *` statements. Here we can define `__all__` to contain the four function of main interest:

```
__all__ = ['annual_rate', 'days', 'initial_amount', 'present_amount']
```

Now we get

```
>>> from interest import *
['__builtins__', '__doc__', '__name__', '__package__',
 'annual_rate', 'days', 'initial_amount', 'present_amount']
```

How to make Python find a module file. The `doubling.py` program works well as long as it is located in the same folder as the `interest.py` module. However, if we move `doubling.py` to another folder and run it, we get an error:

Terminal
```
doubling.py
Traceback (most recent call last):
  File "doubling.py", line 1, in <module>
    from interest import days
ImportError: No module named interest
```

Unless the module file resides in the same folder, we need to tell Python where to find our module. Python looks for modules in the folders contained in the list `sys.path`. A little program

```
import sys, pprint
pprint.pprint(sys.path)
```

prints out all these predefined module folders. You can now do one of two things:

1. Place the module file in one of the folders in `sys.path`.
2. Include the folder containing the module file in `sys.path`.

There are two ways of doing the latter task. Alternative 1 is to explicitly insert a new folder name in `sys.path` in the program that uses the module:

```
modulefolder = '../../pymodules'
sys.path.insert(0, modulefolder)
```

(In this sample path, the slashes are Unix specific. On Windows you must use backslashes and a raw string. A better solution is to express the path as `os.path.join(os.pardir, os.pardir, 'mymodules')`. This will work on all platforms.)

Python searches the folders in the sequence they appear in the `sys.path` list so by inserting the folder name as the first list element we ensure that our module is found quickly, and in case there are other modules with the same name in other folders in `sys.path`, the one in `modulefolder` gets imported.

Alternative 2 is to specify the folder name in the `PYTHONPATH` environment variable. All folder names listed in `PYTHONPATH` are automatically included in `sys.path` when a Python program starts. On Mac and Linux systems, environment variables like `PYTHONPATH` are set in the `.bashrc` file in the home folder, typically as

```
export PYTHONPATH=$HOME/software/lib/pymodules:$PYTHONPATH
```

if `$HOME/software/lib/pymodules` is the folder containing Python modules. On Windows, you launch *Computer - Properties - Advanced System Settings - Environment Variables*, click under *System Variable*, write in `PYTHONPATH` as variable name and the relevant folder(s) as value.

How to make Python run the module file. The description above concerns importing the module in a program located anywhere on the system. If we want to run the module file as a program, anywhere on the system, the operating system searches the `PATH` environment variable for the program name `interst.py`. It is therefore necessary to update `PATH` with the folder where `interest.py` resides.

On Mac and Linux system this is done in `.bashrc` in the same way as for `PYTHONPATH`:

```
export PYTH=$HOME/software/lib/pymodules:$PATH
```

On Windows, launch the dialog for setting environment variables as described above and find the `PATH` variable. It already has much content, so you add your new folder value either at the beginning or end, using a semicolon to separate the new value from the existing ones.

4.9.8 Distributing modules

Modules are usually useful pieces of software that others can take advantage of. Even though our simple `interest` module is of less interest to the world, we can illustrate how such a module is most effectively distributed to other users. The standard in Python is to distribute the module file together with a program called `setup.py` such that any user can just do

```
Terminal> sudo python setup.py install
```

to install the module in one of the directories in `sys.path` so that the module is immediately accessible anywhere, both for import in a Python program and for execution as a stand-alone program.

The `setup.py` file is in the case of one module file very short:

```
from distutils.core import setup
setup(name='interest',
      version='1.0',
      py_modules=['interest'],
      scripts=['interest.py'],
      )
```

The `scripts=` keyword argument can be dropped if the module is just to be imported and not run as a program as well. More module files can trivially be added to the list.

A user who runs `setup.py install` on an Ubuntu machine will see from the output that `interest.py` is copied to the system folders `/usr/local/lib/python2.7/dist-packages` and `/usr/local/bin`. The former folder is for module files, the latter for executable programs.

Remark

Distributing a single module file can be done as shown, but if you have two or more module files that belong together, you should definitely create a *package* [26].

4.9.9 Making software available on the Internet

Distributing software today means making it available on one of the major project hosting sites such as Googlecode, GitHub, or Bitbucket. You will develop and maintain the project files on your own computer(s), but frequently push the software out in the cloud such that others also get your updates. The mentioned sites have is very strong support for collaborative software development.

Since many already have a Gmail or Google account, we briefly describe how you can make your software available at Googlecode.

1. Go to `http://googlecode.com`.
2. Sign in with your Gmail/Google username and password.
3. Click on *Create a new project*.
4. Fill in project name, summary, and description.

5. Choose a *version control system*. Git is recommended.
6. Select a license for the software. Notice that on Googlecode a software project must be available to the whole world as open source code. Other sites (Bitbucket and GitHub) allows private projects.
7. Press *Create project*.
8. Click on *Source* and then on *Checkout*.
9. Go to some appropriate place in your home folder tree where you want to store this Googlecode project.
10. Copy and paste the command under *Option 1* and run it in a terminal window (this requires that Git is installed on your system).

You now have a folder with the same name as the project name. Copy `setup.py` and `interst.py` to the folder and move to the folder. It is good to also write a short `README` file explaining what the project is about. Alternatively, you can later extend the description on the Googlecode web page for the project. Run

Terminal

```
Terminal> git add .
Terminal> git commit -am 'First registration of project files'
Terminal> git push origin master
```

You are now prompted for the `googlecode.com` password, which you can click on right under the *Option 1* command on the web page.

The above `git` commands look cryptic, but these commands plus 2-3 more are the essence of how programmers today work on software projects, small or big. I strongly encourage you to learn more about version control systems and project hosting sites [13]. The tools are in nature like Dropbox and Google Drive, just much more powerful when you collaborate with others.

Your project files are now stored in the cloud at `http://code.google.com/p/project-name`. Anyone can get the software by the listed `git clone` command you used above, or by clicking on the `zip` link under *Source* and *Browse* to download a zip file.

Every time you update the project files, you need to register the update at the Googlecode project site by

Terminal

```
Terminal> git commit -am 'Description of the changes you made...'
Terminal> git push origin master
```

The files at Googlecode are now synchronized with your local ones.

There is a bit more to be said here to make you up and going with this style of professional work [13], but the information above gives you at least a glimpse of how to put your software project in the cloud and opening it up for others. The Googlecode address for the particular `interest` module described above is `http://code.google.com/p/interest-primer`.

4.10 Summary

4.10.1 Chapter topics

Question and answer input. Prompting the user and reading the answer back into a variable is done by

```
var = raw_input('Give value: ')
```

The `raw_input` function returns a string containing the characters that the user wrote on the keyboard before pressing the Return key. It is necessary to convert `var` to an appropriate object (`int` or `float`, for instance) if we want to perform mathematical operations with `var`. Sometimes

```
var = eval(raw_input('Give value: '))
```

is a flexible and easy way of transforming the string to the right type of object (integer, real number, list, tuple, and so on). This last statement will not work, however, for strings unless the text is surrounded by quotes when written on the keyboard. A general conversion function that turns any text without quotes into the right object is `scitools.misc.str2obj`:

```
from scitools.misc import str2obj
var = str2obj(raw_input('Give value: '))
```

Typing, for example, `3` makes `var` refer to an `int` object, `3.14` results in a `float` object, `[-1,1]` results in a `list`, `(1,3,5,7)` in a `tuple`, and `some text` in the string (`str`) object `'some text'` (run the program `str2obj_demo.py` to see this functionality demonstrated).

Getting command-line arguments. The `sys.argv[1:]` list contains all the command-line arguments given to a program (`sys.argv[0]` contains the program name). All elements in `sys.argv` are strings. A typical usage is

```
parameter1 = float(sys.argv[1])
parameter2 = int(sys.argv[2])
parameter3 = sys.argv[3]          # parameter3 can be string
```

Using option-value pairs. The `argparse` module is recommended for interpreting command-line arguments of the form `-option value`. A simple recipe with `argparse` reads

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--p1', '--parameter_1', type=float,
                    default=0.0, help='1st parameter')
parser.add_argument('--p2', type=float,
                    default=0.0, help='2nd parameter')

args = parser.parse_args()
p1 = args.p1
p2 = args.p2
```

On the command line we can provide any or all of these options:

```
--parameter_1 --p1 --p2
```

where each option must be succeeded by a suitable value. However, `argparse` is very flexible can easily handle options without values or command-line arguments without any option specifications.

Generating code on the fly. Calling `eval(s)` turns a string `s`, containing a Python expression, into code as if the contents of the string were written directly into the program code. The result of the following `eval` call is a `float` object holding the number 21.1:

```
>>> x = 20
>>> r = eval('x + 1.1')
>>> r
21.1
>>> type(r)
<type 'float'>
```

The `exec` function takes a string with arbitrary Python code as argument and executes the code. For example, writing

```
exec("""
def f(x):
    return %s
""" % sys.argv[1])
```

is the same as if we had hardcoded the (for the programmer unknown) contents of `sys.argv[1]` into a function definition in the program.

Turning string formulas into Python functions. Given a mathematical formula as a string, `s`, we can turn this formula into a callable Python function `f(x)` by

```
from scitools.std import StringFunction

f = StringFunction(s)
```

The string formula can contain parameters and an independent variable with another name than `x`:

```
Q_formula = 'amplitude*sin(w*t-phaseshift)'
Q = StringFunction(Q_formula, independent_variable='t',
                   amplitude=1.5, w=pi, phaseshift=0)
values1 = [Q(i*0.1) for t in range(10)]
Q.set_parameters(phaseshift=pi/4, amplitude=1)
values2 = [Q(i*0.1) for t in range(10)]
```

Functions of several independent variables are also supported:

```
f = StringFunction('x+y**2+A', independent_variables=('x', 'y'),
                   A=0.2)
x = 1; y = 0.5
print f(x, y)
```

File operations. Reading from or writing to a file first requires that the file is opened, either for reading, writing, or appending:

```
infile  = open(filename, 'r')   # read
outfile = open(filename, 'w')   # write
outfile = open(filename, 'a')   # append
```

or using `with`:

```
with open(filename, 'r') as infile:   # read
with open(filename, 'w') as outfile:  # write
with open(filename, 'a') as outfile:  # append
```

There are four basic reading commands:

```
line    = infile.readline()   # read the next line
filestr = infile.read()       # read rest of file into string
lines   = infile.readlines()  # read rest of file into list
for line in infile:           # read rest of file line by line
```

File writing is usually about repeatedly using the command

```
outfile.write(s)
```

where `s` is a string. Contrary to `print s`, no newline is added to `s` in `outfile.write(s)`.

After reading or writing is finished, the file must be closed:

```
somefile.close()
```

However, closing the file is not necessary if we employ the `with` statement for reading or writing files:

```
with open(filename, 'w') as outfile:
    for var1, var2 in data:
        outfile.write('%5.2f %g\n' % (var1, var2))
# outfile is closed
```

Handling exceptions. Testing for potential errors is done with `try-except` blocks:

```
try:
    <statements>
except ExceptionType1:
    <provide a remedy for ExceptionType1 errors>
except ExceptionType2, ExceptionType3, ExceptionType4:
    <provide a remedy for three other types of errors>
except:
    <provide a remedy for any other errors>
...
```

The most common exception types are `NameError` for an undefined variable, `TypeError` for an illegal value in an operation, and `IndexError` for a list index out of bounds.

Raising exceptions. When some error is encountered in a program, the programmer can raise an exception:

```
if z < 0:
    raise ValueError('z=%s is negative - cannot do log(z)' % z)
r = log(z)
```

Modules. A module is created by putting a set of functions in a file. The filename (minus the required extension `.py`) is the name of the module. Other programs can import the module only if it resides in the same folder or in a folder contained in the `sys.path` list (see Section 4.9.7 for how to deal with this potential problem). Optionally, the module file can have a special `if` construct at the end, called test block, which tests the module or demonstrates its usage. The test block does not get executed when the module is imported in another program, only when the module file is run as a program.

Terminology. The important computer science topics and Python tools in this chapter are

- command line
- `sys.argv`
- `raw_input`
- `eval` and `exec`
- file reading and writing
- handling and raising exceptions
- module
- test block

4.10.2 Example: Bisection root finding

Problem. The summarizing example of this chapter concerns the implementation of the Bisection method for solving nonlinear equations of the form $f(x) = 0$ with respect to x. For example, the equation

$$x = 1 + \sin x$$

can be cast in the form $f(x) = 0$ if we move all terms to the left-hand side and define $f(x) = x - 1 - \sin x$. We say that x is a *root* of the equation $f(x) = 0$ if x is a solution of this equation. Nonlinear equations $f(x) = 0$ can have zero, one, several, or infinitely many roots.

Numerical methods for computing roots normally lead to approximate results only, i.e., $f(x)$ is not made exactly zero, but very close to zero. More precisely, an approximate root x fulfills $|f(x)| \leq \epsilon$, where ϵ is a small number. Methods for finding roots are of an iterative nature: we start with a rough approximation to a root and perform a repetitive set of steps that aim to improve the approximation. Our particular method for computing roots, the Bisection method, guarantees to find an approximate root, while other methods, such as the widely used Newton's method (see Section A.1.10), can fail to find roots.

The idea of the Bisection method is to start with an interval $[a, b]$ that contains a root of $f(x)$. The interval is halved at $m = (a + b)/2$, and if $f(x)$ changes sign in the left half interval $[a, m]$, one continues with that interval, otherwise one continues with the right half interval $[m, b]$. This procedure is repeated, say n times, and the root is then guaranteed to be inside an interval of length $2^{-n}(b - a)$. The task is to write a program that implements the Bisection method and verify the implementation.

Solution. To implement the Bisection method, we need to translate the description in the previous paragraph to a precise algorithm that can be almost directly translated to computer code. Since the halving of the interval is repeated many times, it is natural to do this inside a loop. We start with the interval $[a, b]$, and adjust a to m if the root must be in the right half of the interval, or we adjust b to m if the root must be in the left half. In a language close to computer code we can express the algorithm precisely as follows:

```
for i = 0,1,2, ..., n:
    m = (a + b)/2
    if f(a)*f(m) <= 0:
        b = m  # root is in left half
    else:
        a = m  # root is in right half

# f(x) has a root in [a,b]
```

Figure 4.2 displays graphically the first four steps of this algorithm for solving the equation $\cos(\pi x) = 0$, starting with the interval $[0, 0.82]$. The graphs are automatically produced by the program `bisection_movie.py`, which was run as follows for this particular example:

Terminal

```
bisection_movie.py 'cos(pi*x)' 0 0.82
```

The first command-line argument is the formula for $f(x)$, the next is a, and the final is b.

In the algorithm listed above, we recompute $f(a)$ in each `if`-test, but this is not necessary if a has not changed since the last $f(a)$ computations. It is a good habit in numerical programming to avoid redundant work. On modern computers the Bisection algorithm normally runs so fast that we can afford to do more work than necessary. However, if $f(x)$ is not a simple formula, but computed by comprehensive calculations in a program, the evaluation of f might take minutes or even hours, and reducing the number of evaluations in the Bisection algorithm is then very important. We will therefore introduce extra variables in the

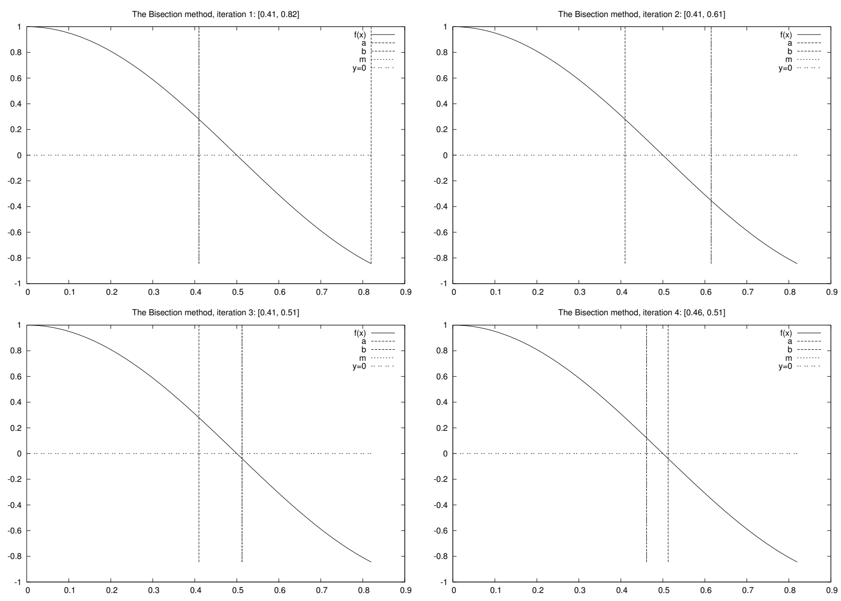


Fig. 4.2 Illustration of the first four iterations of the Bisection algorithm for solving $\cos(\pi x) = 0$. The vertical lines correspond to the current value of a and b.

algorithm above to save an $f(m)$ evaluation in each iteration in the `for` loop:

```
f_a = f(a)
for i = 0,1,2, ..., n:
    m = (a + b)/2
    f_m = f(m)
    if f_a*f_m <= 0:
        b = m   # root is in left half
    else:
        a = m   # root is in right half
        f_a = f_m

# f(x) has a root in [a,b]
```

To execute the algorithm above, we need to specify n. Say we want to be sure that the root lies in an interval of maximum extent ϵ. After n iterations the length of our current interval is $2^{-n}(b-a)$, if $[a,b]$ is the initial interval. The current interval is sufficiently small if

$$2^{-n}(b-a) = \epsilon,$$

which implies

$$n = -\frac{\ln \epsilon - \ln(b-a)}{\ln 2}\,. \tag{4.6}$$

Instead of calculating this n, we may simply stop the iterations when the length of the current interval is less than ϵ. The loop is then naturally implemented as a `while` loop testing on whether $b - a \leq \epsilon$. To make the algorithm more foolproof, we also insert a test to ensure that $f(x)$

really changes sign in the initial interval. This guarantees a root in $[a, b]$. (However, $f(a)f(b) < 0$ is not a necessary condition if there is an even number of roots in the initial interval.)

Our final version of the Bisection algorithm now becomes

```
f_a=f(a)
if f_a*f(b) > 0:
    # error: f does not change sign in [a,b]

i = 0
while b-a > epsilon:
    i = i + 1
    m = (a + b)/2
    f_m = f(m)
    if f_a*f_m <= 0:
        b = m  # root is in left half
    else:
        a = m  # root is in right half
        f_a = f_m

# if x is the real root, |x-m| < epsilon
```

This is the algorithm we aim to implement in a Python program.

A direct translation of the previous algorithm to a valid Python program is a matter of some minor edits:

```
eps = 1E-5
a, b = 0, 10

fa = f(a)
if fa*f(b) > 0:
    print 'f(x) does not change sign in [%g,%g].' % (a, b)
    sys.exit(1)

i = 0   # iteration counter
while b-a > eps:
    i += 1
    m = (a + b)/2.0
    fm = f(m)
    if fa*fm <= 0:
        b = m  # root is in left half of [a,b]
    else:
        a = m  # root is in right half of [a,b]
        fa = fm
    print 'Iteration %d: interval=[%g, %g]' % (i, a, b)

x = m          # this is the approximate root
print 'The root is', x, 'found in', i, 'iterations'
print 'f(%g)=%g' % (x, f(x))
```

This program is found in the file `bisection_v1.py`.

Verification. To verify the implementation in `bisection_v1.py` we choose a very simple $f(x)$ where we know the exact root. One suitable example is a linear function, $f(x) = 2x - 3$ such that $x = 3/2$ is the root of f. As can be seen from the source code above, we have inserted a `print` statement inside the `while` loop to control that the program really does the right things. Running the program yields the output

```
Iteration 1: interval=[0, 5]
Iteration 2: interval=[0, 2.5]
Iteration 3: interval=[1.25, 2.5]
Iteration 4: interval=[1.25, 1.875]
...
Iteration 19: interval=[1.5, 1.50002]
Iteration 20: interval=[1.5, 1.50001]
The root is 1.50000572205 found in 20 iterations
f(1.50001)=1.14441e-05
```

It seems that the implementation works. Further checks should include hand calculations for the first (say) three iterations and comparison of the results with the program.

Making a function. The previous implementation of the bisection algorithm is fine for many purposes. To solve a new problem $f(x) = 0$ it is just necessary to change the `f(x)` function in the program. However, if we encounter solving $f(x) = 0$ in another program in another context, we must put the bisection algorithm into that program in the right place. This is simple in practice, but it requires some careful work, and it is easy to make errors. The task of solving $f(x) = 0$ by the bisection algorithm is much simpler and safer if we have that algorithm available as a function in a module. Then we can just import the function and call it. This requires a minimum of writing in later programs.

When you have a "flat" program as shown above, without basic steps in the program collected in functions, you should always consider dividing the code into functions. The reason is that parts of the program will be much easier to reuse in other programs. You save coding, and that is a good rule! A program with functions is also easier to understand, because statements are collected into logical, separate units, which is another good rule! In a mathematical context, functions are particularly important since they naturally split the code into general algorithms (like the bisection algorithm) and a problem-specific part (like a special choice of $f(x)$).

Shuffling statements in a program around to form a new and better designed version of the program is called *refactoring*. We shall now refactor the `bisection_v1.py` program by putting the statements in the bisection algorithm in a function `bisection`. This function naturally takes $f(x)$, a, b, and ϵ as parameters and returns the found root, perhaps together with the number of iterations required:

```
def bisection(f, a, b, eps):
    fa = f(a)
    if fa*f(b) > 0:
        return None, 0

    i = 0   # iteration counter
    while b-a > eps:
        i += 1
        m = (a + b)/2.0
        fm = f(m)
        if fa*fm <= 0:
            b = m  # root is in left half of [a,b]
        else:
            a = m  # root is in right half of [a,b]
            fa = fm
    return m, i
```

After this function we can have a test program:

```
def f(x):
    return 2*x - 3   # one root x=1.5

x, iter = bisection(f, a=0, b=10, eps=1E-5)
if x is None:
    print 'f(x) does not change sign in [%g,%g].' % (a, b)
else:
    print 'The root is', x, 'found in', iter, 'iterations'
    print 'f(%g)=%g' % (x, f(x))
```

The complete code is found in file `bisection_v2.py`.

Making a test function. Rather than having a main program as above for verifying the implementation, we should make a test function `test_bisection` as described in Section 4.9.4. To this end, we move the statements above inside a function, drop the output, but instead make a boolean variable `success` that is `True` if the test is passed and `False` otherwise. Then we do `assert success, msg`, which will abort the program if the test fails. The `msg` variable is a string with more explanation of what went wrong the test fails. A test function with this structure is easy to integrate into the widely used testing frameworks nose and pytest, and there are no good reasons for not adopting this structure. The code checking that the root is within a distance ϵ to the exact root becomes

```
def test_bisection():
    def f(x):
        return 2*x - 3   # one root x=1.5

    x, iter = bisection(f, a=0, b=10, eps=1E-5)
    success = abs(x - 1.5) < 1E-5  # test within eps tolerance
    assert success, 'found x=%g != 1.5' % x
```

Making a module. A motivating factor for implementing the bisection algorithm as a function `bisection` was that we could import this function in other programs to solve $f(x) = 0$ equations. We therefore need to make a module file `bisection.py` such that we can do, e.g.,

```
from bisection import bisection
x, iter = bisection(lambda x: x**3 + 2*x -1, -10, 10, 1E-5)
```

A module file should not execute a main program, but just define functions, import modules, and define global variables. Any execution of a main program must take place in the test block, otherwise the `import` statement will start executing the main program, resulting in very disturbing statements for another program that wants to solve a different $f(x) = 0$ equation.

The `bisection_v2.py` file had a main program that was just a simple test for checking that the `bisection` algorithm works for a linear function. We took this main program and wrapped in a test function `test_bisection` above. To run the test, we make the call to this function from the test block:

```
if __name__ == '__main__':
    test_bisection()
```

This is all that is demanded to turn the file `bisection_v2.py` into a proper module file `bisection.py`.

Defining a user interface. It is nice to have our `bisection` module do more than just test itself: there should be a user interface such that we can solve real problems $f(x) = 0$, where $f(x)$, a, b, and ϵ are defined on the command line by the user. A dedicated function can read from the command line and return the data as Python object. For reading the function $f(x)$ we can either apply `eval` on the command-line argument, or use the more sophisticated `StringFunction` tool from Section 4.3.3. With `eval` we need to import functions from the `math` module in case the user have such functions in the expression for $f(x)$. With `StringFunction` this is not necessary.

A `get_input()` for getting input from the command line can be implemented as

```
def get_input():
    """Get f, a, b, eps from the command line."""
    from scitools.std import StringFunction
    try:
        f = StringFunction(sys.argv[1])
        a = float(sys.argv[2])
        b = float(sys.argv[3])
        eps = float(sys.argv[4])
    except IndexError:
        print 'Usage %s: f a b eps' % sys.argv[0]
        sys.exit(1)
    return f, a, b, eps
```

To solve the corresponding $f(x) = 0$ problem, we simply add a branch in the `if` test in the test block:

```
if __name__ == '__main__':
    import sys
    if len(sys.argv) >= 2 and sys.argv[1] == 'test':
        test_bisection()
    else:
        f, a, b, eps = get_input()
        x, iter = bisection(f, a, b, eps)
        print 'Found root x=%g in %d iterations' % (x, iter)
```

Desired properties of a module

Our `bisection.py` code is a complete module file with the following generally desired features of Python modules:

- other programs can import the `bisection` function,
- the module can test itself (with a pytest/nose-compatible test function),
- the module file can be run as a program with a user interface where a general rooting finding problem can be specified in terms of a formula for $f(x)$ along with the parameters a, b, and ϵ.

Using the module. Suppose you want to solve $x/(x-1) = \sin x$ using the `bisection` module. What do you have to do? First, you must reformulate the equation as $f(x) = 0$, i.e., $x/(x-1) - \sin x = 0$, or maybe multiply by $x-1$ to get $f(x) = x - (x-1)\sin x$.

It is required to identify an interval for the root. By evaluating $f(x)$ for some points x one can be trial and error locate an interval. A more convenient approach is to plot the function $f(x)$ and visually inspect where a root is. Chapter 5 describes the techniques, but here we simply state the recipe. We start `ipython -pylab` and write

```
In [1]: x = linspace(-3, 3, 50)  # generate 50 coordinates in [-3,3]

In [2]: y = x - (x-1)*sin(x)

In [3]: plot(x, y)
```

Figure 4.3 shows $f(x)$ and we clearly see that, e.g., $[-2, 1]$ is an appropriate interval.

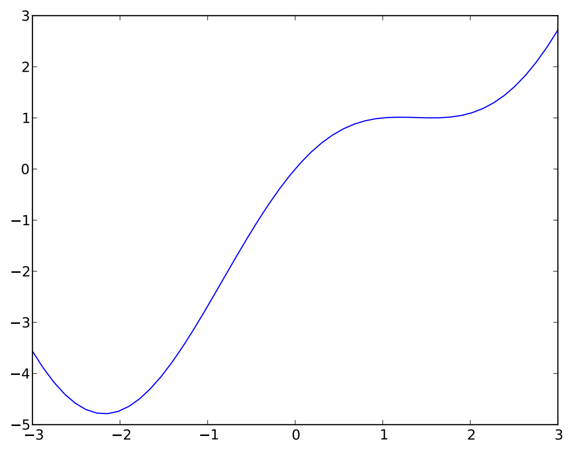


Fig. 4.3 Plot of $f(x) = x - \sin(x)$.

The next step is to run the Bisection algorithm. There are two possibilities:

- make a program where you code $f(x)$ and run the `bisection` function, or
- run the `bisection.py` program directly.

The latter approach is the simplest:

Terminal

```
bisection.py "x - (x-1)*sin(x)" -2 1 1E-5
Found root x=-1.90735e-06 in 19 iterations
```

The alternative approach is to make a program:

```python
from bisection import bisection
from math import sin

def f(x):
    return x - (x-1)*sin(x)

x, iter = bisection(f, a=-2, b=1, eps=1E-5)
print x, iter
```

Potential problems with the software. Let us solve

- $x = \tanh x$ with start interval $[-10, 10]$ and $\epsilon = 10^{-6}$,
- $x^5 = \tanh(x^5)$ with start interval $[-10, 10]$ and $\epsilon = 10^{-6}$.

Both equations have one root $x = 0$.

```
Terminal
bisection.py "x-tanh(x)" -10 10
Found root x=-5.96046e-07 in 25 iterations

bisection.py "x**5-tanh(x**5)" -10 10
Found root x=-0.0266892 in 25 iterations
```

These results look strange. In both cases we halve the start interval $[-10, 10]$ 25 times, but in the second case we end up with a much less accurate root although the value of ϵ is the same. A closer inspection of what goes on in the bisection algorithm reveals that the inaccuracy is caused by round-off errors. As $a, b, m \to 0$, raising a small number to the fifth power in the expression for $f(x)$ yields a much smaller result. Subtracting a very small number $\tanh x^5$ from another very small number x^5 may result in a small number with wrong sign, and the sign of f is essential in the bisection algorithm. We encourage the reader to graphically inspect this behavior by running these two examples with the `bisection_plot.py` program using a smaller interval $[-1, 1]$ to better see what is going on. The command-line arguments for the `bisection_plot.py` program are `'x-tanh(x)' -1 1` and `'x**5-tanh(x**5)' -1 1`. The very flat area, in the latter case, where $f(x) \approx 0$ for $x \in [-1/2, 1/2]$ illustrates well that it is difficult to locate an exact root.

Distributing the bisection module to others. The Python standard for installing software is to run a `setup.py` program,

```
Terminal
Terminal> sudo python setup.py install
```

to install the system. The relevant `setup.py` for the `bisection` module arises from substituting the name `interest` by `bisection` in the `setup.py` file listed in Section 4.9.8. You can then distribute `bisection.py` and `setup.py` together.

4.11 Exercises

Exercise 4.1: Make an interactive program

Make a program that asks the user for a temperature in Fahrenheit degrees and reads the number; computes the corresponding temperature in Celsius degrees; and prints out the temperature in the Celsius scale.
Filename: `f2c_qa.py`.

Exercise 4.2: Read a number from the command line

Modify the program from Exercise 4.1 such that the Fahrenheit temperature is read from the command line. Filename: `f2c_cml.py`.

Exercise 4.3: Read a number from a file

Modify the program from Exercise 4.1 such that the Fahrenheit temperature is read from a file with the following content:

```
Temperature data
----------------

Fahrenheit degrees: 67.2
```

Hint. Create a sample file manually. In the program, skip the first three lines, split the fourth line into words and grab the third word.
Filename: `f2c_file_read.py`.

Exercise 4.4: Read and write several numbers from and to file

This is a variant of Exercise 4.3 where we have several Fahrenheit degrees in a file and want to read all of them into a list and convert the numbers to Celsius degrees. Thereafter, we want to write out a file with two columns, the left with the Fahrenheit degrees and the right with the Celsius degrees.

An example on the input file format looks like

```
Temperature data
----------------

Fahrenheit degrees: 67.2
Fahrenheit degrees: 66.0
Fahrenheit degrees: 78.9
Fahrenheit degrees: 102.1
Fahrenheit degrees: 32.0
Fahrenheit degrees: 87.8
```

A sample file is `Fdeg.dat`[5]. Filename: `f2c_file_read_write.py`.

Exercise 4.5: Use exceptions to handle wrong input

Extend the program from Exercise 4.2 with a `try-except` block to handle the potential error that the Fahrenheit temperature is missing on the command line. Filename: `f2c_cml_exc.py`.

Exercise 4.6: Read input from the keyboard

Make a program that asks for input from the user, applies `eval` to this input, and prints out the type of the resulting object and its value. Test the program by providing five types of input: an integer, a real number, a complex number, a list, and a tuple. Filename: `objects_qa.py`.

Exercise 4.7: Read input from the command line

a) Let a program store the result of applying the `eval` function to the first command-line argument. Print out the resulting object and its type.

b) Run the program with different input: an integer, a real number, a list, and a tuple.

Hint. On Unix systems you need to surround the tuple expressions in quotes on the command line to avoid error message from the Unix shell.

c) Try the string `"this is a string"` as a command-line argument. Why does this string cause problems and what is the remedy?
Filename: `objects_cml.py`.

Exercise 4.8: Try MSWord or LibreOffice to write a program

The purpose of this exercise is to tell you how hard it may be to write Python programs in the standard programs that most people use for writing text.

a) Type the following one-line program in either MSWord or LibreOffice:

```
print "Hello, World!"
```

Both Word and LibreOffice are so “smart” that they automatically edit “print” to “Print” since a sentence should always start with a capital. This is just an example that word processors are made for writing documents, not computer programs.

[5] `http://tinyurl.com/pwyasaa/input/Fdeg.dat`

b) Save the program as a `.docx` (Word) or `.odt` (LibreOffice) file. Now try to run this file as a Python program. What kind of error message do you get? Can you explain why?

c) Save the program as a `.txt` file in Word or LibreOffice and run the file as a Python program. What happened now? Try to find out what the problem is.

Exercise 4.9: Prompt the user for input to a formula

Consider the simplest program for evaluating the formula $y(t) = v_0 t - \frac{1}{2}gt^2$:

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

Modify this code so that the program asks the user questions `t=?` and `v0=?`, and then gets `t` and `v0` from the user's input through the keyboard. Filename: `ball_qa.py`.

Exercise 4.10: Read parameters in a formula from the command line

Modify the program listed in Exercise 4.9 such that `v0` and `t` are read from the command line. Filename: `ball_cml.py`.

Exercise 4.11: Use exceptions to handle wrong input

The program from Exercise 4.10 reads input from the command line. Extend that program with exception handling such that missing command-line arguments are detected. In the `except IndexError` block, use the `raw_input` function to ask the user for missing input data. Filename: `ball_cml_qa.py`.

Exercise 4.12: Test validity of input data

Test if the `t` value read in the program from Exercise 4.10 lies between 0 and $2v_0/g$. If not, print a message and abort the execution. Filename: `ball_cml_tcheck.py`.

Exercise 4.13: Raise an exception in case of wrong input

Instead of printing an error message and aborting the program explicitly, raise a `ValueError` exception in the `if` test on legal `t` values in the program from Exercise 4.12. Notify the user about the legal interval for t in the exception message. Filename: `ball_cml_ValueError.py`.

Exercise 4.14: Evaluate a formula for data in a file

We consider the formula $y(t) = v_0 t - 0.5gt^2$ and want to evaluate y for a range of t values found in a file with format

```
v0: 3.00
t:
0.15592  0.28075   0.36807889 0.35 0.57681501876
0.21342619  0.0519085  0.042  0.27  0.50620017 0.528
0.2094294  0.1117  0.53012  0.3729850  0.39325246
0.21385894  0.3464815 0.57982969 0.10262264
0.29584013  0.17383923
```

More precisely, the first two lines are always present, while the next lines contain an arbitrary number of t values on each line, separated by one or more spaces.

a) Write a function that reads the input file and returns v_0 and a list with the t values.

b) Write a function that creates a file with two nicely formatted columns containing the t values to the left and the corresponding y values to the right. Let the t values appear in increasing order (note that the input file does not necessarily have the t values sorted).

c) Make a test function that generates an input file, calls the function for reading the file, and checks that the returned data objects are correct. Filename: `ball_file_read_write.py`.

Exercise 4.15: Compute the distance it takes to stop a car

A car driver, driving at velocity v_0, suddenly puts on the brake. What braking distance d is needed to stop the car? One can derive, using Newton's second law of motion or a corresponding energy equation, that

$$d = \frac{1}{2}\frac{v_0^2}{\mu g}\,. \tag{4.7}$$

Make a program for computing d in (4.7) when the initial car velocity v_0 and the friction coefficient μ are given on the command line. Run the program for two cases: $v_0 = 120$ and $v_0 = 50$ km/h, both with $\mu = 0.3$ (μ is dimensionless).

Hint. Remember to convert the velocity from km/h to m/s before inserting the value in the formula.
Filename: `stopping_length.py`.

Exercise 4.16: Look up calendar functionality

The purpose of this exercise is to make a program that takes a date, consisting of year (4 digits), month (2 digits), and day (1-31) on the command line and prints the corresponding name of the weekday (Monday, Tuesday, etc.). Python has a module `calendar`, which makes it easy to solve the exercise, but the task is to find out how to use this module. Filename: `weekday.py`.

Exercise 4.17: Use the StringFunction tool

Make the program `user_formula.py` from Section 4.3.2 shorter by using the convenient `StringFunction` tool from Section 4.3.3. Filename: `user_formula2.py`.

Exercise 4.18: Why we test for specific exception types

The simplest way of writing a `try-except` block is to test for *any* exception, for example,

```
try:
    C = float(sys.arg[1])
except:
    print 'C must be provided as command-line argument'
    sys.exit(1)
```

Write the above statements in a program and test the program. What is the problem?

The fact that a user can forget to supply a command-line argument when running the program was the original reason for using a `try` block. Find out what kind of exception that is relevant for this error and test for this specific exception and re-run the program. What is the problem now? Correct the program. Filename: `unnamed_exception.py`.

Exercise 4.19: Make a complete module

a) Make six conversion functions between temperatures in Celsius, Kelvin, and Fahrenheit: `C2F`, `F2C`, `C2K`, `K2C`, `F2K`, and `K2F`.

b) Collect these functions in a module `convert_temp`.

c) Import the module in an interactive Python shell and demonstrate some sample calls on temperature conversions.

d) Insert the session from c) in a triple quoted string at the top of the module file as a doc string for demonstrating the usage.

e) Write a function `test_conversion()` that verifies the implementation. Call this function from the test block if the first command-line argument is `verify`.

Hint. Check that `C2F(F2C(f))` is `f`, `K2C(C2K(c))` is `c`, and `K2F(F2K(f))` is `f` - with tolerance. Follow the conventions for test functions outlined in Sections 4.9.4 and 4.10.2 with a boolean variable that is `False` if a test failed, and `True` if all test are passed, and then an `assert` statement to abort the program when any test fails.

f) Add a user interface to the module such that the user can write a temperature as the first command-line argument and the corresponding temperature scale as the second command-line argument, and then get the temperature in the two other scales as output. For example, `21.3 C` on the command line results in the output `70.3 F 294.4 K`. Encapsulate the user interface in a function, which is called from the test block.
Filename: `convert_temp.py`.

Exercise 4.20: Make a module

Collect the `f` and `S` functions in the program from Exercise 3.15 in a separate file such that this file becomes a module. Put the statements making the table (i.e., the main program from Exercise 3.15) in a separate function `table(n_values, alpha_values, T)`. Make a test block in the module to read T and a series of n and α values from the command line and make a corresponding call to `table`. Filename: `sinesum2.py`.

Exercise 4.21: Read options and values from the command line

Let the input to the program in Exercise 4.20 be option-value pairs with the options `-n`, `-alpha`, and `-T`. Provide sensible default values in the module file.

Hint. Apply the `argparse` module to read the command-line arguments. Do not copy code from the `sinesum2` module, but make a new file for reading option-value pairs from the command and import the `table` function from the `sinesum2` module.
Filename: `sinesum3.py`.

Exercise 4.22: Check if mathematical identities hold

Because of round-off errors, it could happen that a mathematical rule like $(ab)^3 = a^3b^3$ does not hold exactly on a computer. The idea of testing this potential problem is to check such identities for a large number of random numbers. We can make random numbers using the `random` module in Python:

```
import random
a = random.uniform(A, B)
b = random.uniform(A, B)
```

Here, `a` and `b` will be random numbers, which are always larger than or equal to `A` and smaller than `B`.

a) Make a function `power3_identity(A=-100, B=100, n=1000)` that tests the identity `(a*b)**3 == a**3*b**3` a large number of times, `n`. Return the fraction of failures.

Hint. Inside the loop over `n`, draw random numbers `a` and `b` as described above and count the number of times the test is `True`.

b) We shall now parameterize the expressions to be tested. Make a function

```
equal(expr1, expr2, A=-100, B=100, n=500)
```

where `expr1` and `expr2` are strings containing the two mathematical expressions to be tested. More precisely, the function draws random numbers `a` and `b` between `A` and `B` and tests if `eval(expr1) == eval(expr2)`. Return the fraction of failures.

Test the function on the identities $(ab)^3 = a^3b^3$, $e^{a+b} = e^a e^b$, and $\ln a^b = b \ln a$.

Hint. Make the `equal` function robust enough to handle illegal a and b values in the mathematical expressions (e.g., $a \leq 0$ in $\ln a$).

c) We want to test the validity of the following set of identities on a computer:

- $a - b$ and $-(b - a)$
- a/b and $1/(b/a)$
- $(ab)^4$ and a^4b^4
- $(a + b)^2$ and $a^2 + 2ab + b^2$
- $(a + b)(a - b)$ and $a^2 - b^2$
- e^{a+b} and $e^a e^b$
- $\ln a^b$ and $b \ln a$
- $\ln ab$ and $\ln a + \ln b$
- ab and $e^{\ln a + \ln b}$
- $1/(1/a + 1/b)$ and $ab/(a + b)$
- $a(\sin^2 b + \cos^2 b)$ and a
- $\sinh(a + b)$ and $(e^a e^b - e^{-a} e^{-b})/2$
- $\tan(a + b)$ and $\sin(a + b)/\cos(a + b)$
- $\sin(a + b)$ and $\sin a \cos b + \sin b \cos a$

Store all the expressions in a list of 2-tuples, where each 2-tuple contains two mathematically equivalent expressions as strings, which can be sent to the `equal` function. Make a nicely formatted table with a pair of equivalent expressions at each line followed by the failure rate. Write this table to a file. Try out `A=1` and `B=2` as well as `A=1` and `B=100`. Does the failure rate seem to depend on the magnitude of the numbers a and b? Filename: `math_identities_failures.py`.

Exercise 4.23: Compute probabilities with the binomial distribution

Consider an uncertain event where there are two outcomes only, typically success or failure. Flipping a coin is an example: the outcome is uncertain and of two types, either head (can be considered as success) or tail (failure). Throwing a die can be another example, if (e.g.) getting a six is considered success and all other outcomes represent failure. Such experiments are called *Bernoulli trials*.

Let the probability of success be p and that of failure $1-p$. If we perform n experiments, where the outcome of each experiment does not depend on the outcome of previous experiments, the probability of getting success x times, and consequently failure $n-x$ times, is given by

$$B(x,n,p) = \frac{n!}{x!(n-x)!}p^x(1-p)^{n-x}\,. \tag{4.8}$$

This formula (4.8) is called the binomial distribution. The expression $x!$ is the factorial of x: $x! = x(x-1)(x-2)\cdots 1$ and `math.factorial` can do this computation.

a) Implement (4.8) in a function `binomial(x, n, p)`.

b) What is the probability of getting two heads when flipping a coin five times? This probability corresponds to $n=5$ events, where the success of an event means getting head, which has probability $p=1/2$, and we look for $x=2$ successes.

c) What is the probability of getting four ones in a row when throwing a die? This probability corresponds to $n=4$ events, success is getting one and has probability $p=1/6$, and we look for $x=4$ successful events.

d) Suppose cross country skiers typically experience one ski break in one out of 120 competitions. Hence, the probability of breaking a ski can be set to $p=1/120$. What is the probability b that a skier will experience a ski break during five competitions in a world championship?

Hint. This question is a bit more demanding than the other two. We are looking for the probability of 1, 2, 3, 4 or 5 ski breaks, so it is simpler to ask for the probability c of *not* breaking a ski, and then compute $b=1-c$. Define *success* as breaking a ski. We then look for $x=0$ successes out of $n=5$ trials, with $p=1/120$ for each trial. Compute b. Filename: `Bernoulli_trials.py`.

Exercise 4.24: Compute probabilities with the Poisson distribution

Suppose that over a period of t_m time units, a particular uncertain event happens (on average) νt_m times. The probability that there will be x such events in a time period t is approximately given by the formula

$$P(x,t,\nu) = \frac{(\nu t)^x}{x!}e^{-\nu t}\,. \tag{4.9}$$

This formula is known as the Poisson distribution. (It can be shown that (4.9) arises from (4.8) when the probability p of experiencing the event in a small time interval t/n is $p = \nu t/n$ and we let $n \to \infty$.) An important assumption is that all events are independent of each other and that the probability of experiencing an event does not change significantly over time. This is known as a *Poisson process* in probability theory.

a) Implement (4.9) in a function `Poisson(x, t, nu)`, and make a program that reads x, t, and ν from the command line and writes out the probability $P(x,t,\nu)$. Use this program to solve the problems below.

b) Suppose you are waiting for a taxi in a certain street at night. On average, 5 taxis pass this street every hour at this time of the night. What is the probability of not getting a taxi after having waited 30 minutes? Since we have 5 events in a time period of $t_m = 1$ hour, $\nu t_m = \nu = 5$. The sought probability is then $P(0, 1/2, 5)$. Compute this number. What is the probability of having to wait two hours for a taxi? If 8 people need two taxis, that is the probability that two taxis arrive in a period of 20 minutes?

c) In a certain location, 10 earthquakes have been recorded during the last 50 years. What is the probability of experiencing exactly three earthquakes over a period of 10 years in this area? What is the probability that a visitor for one week does not experience any earthquake? With 10 events over 50 years we have $\nu t_m = \nu \cdot 50$ years $= 10$ events, which implies $\nu = 1/5$ event per year. The answer to the first question of having $x = 3$ events in a period of $t = 10$ years is given directly by (4.9). The second question asks for $x = 0$ events in a time period of 1 week, i.e., $t = 1/52$ years, so the answer is $P(0, 1/52, 1/5)$.

d) Suppose that you count the number of misprints in the first versions of the reports you write and that this number shows an average of six misprints per page. What is the probability that a reader of a first draft of one of your reports reads six pages without hitting a misprint? Assuming that the Poisson distribution can be applied to this problem, we have "time" t_m as 1 page and $\nu \cdot 1 = 6$, i.e., $\nu = 6$ events (misprints) per page. The probability of no events in a "period" of six pages is $P(0, 6, 6)$.
Filename: `Poisson_processes.py`.

Array computing and curve plotting 5

A list object is handy for storing tabular data, such as a sequence of objects or a table of objects. An array is very similar to a list, but less flexible and computationally much more efficient. When using the computer to perform mathematical calculations, we often end up with a huge amount of numbers and associated arithmetic operations. Storing numbers in lists may in such contexts lead to slow programs, while arrays can make the programs run much faster. This is crucial for many advanced applications of mathematics in industry and science, where computer programs may run for hours and days, or even weeks. Any clever idea that reduces the execution time by some factor is therefore paramount.

However, one can argue that programmers of mathematical software have traditionally paid too much attention to efficiency and “clever” program constructs. The resulting software often becomes very hard to maintain and extend. In this book we advocate a focus on clear, well-designed, and easy-to-understand programs that work correctly. Thereafter, one can start thinking about optimization for speed. Fortunately, arrays contribute to clear code, correctness and speed - all at once.

This chapter gives an introduction to arrays: how they are created and what they can be used for. Array computing usually ends up with a lot of numbers. It may be very hard to understand what these numbers mean by just looking at them. Since the human is a visual animal, a good way to understand numbers is to visualize them. In this chapter we concentrate on visualizing curves that reflect functions of one variable; i.e., curves of the form $y = f(x)$. A synonym for curve is graph, and the image of curves on the screen is often called a plot. We will use arrays to store the information about points along the curve. In a nutshell, array computing demands visualization and visualization demands arrays.

All program examples in this chapter can be found as files in the folder `src/plot`[1].

5.1 Vectors

This section gives a brief introduction to the vector concept, assuming that you have heard about vectors in the plane and maybe vectors in space before. This background will be valuable when we start to work with arrays and curve plotting.

5.1.1 The vector concept

Some mathematical quantities are associated with a set of numbers. One example is a point in the plane, where we need two coordinates (real numbers) to describe the point mathematically. Naming the two coordinates of a particular point as x and y, it is common to use the notation (x, y) for the point. That is, we group the numbers inside parentheses. Instead of symbols we might use the numbers directly: $(0, 0)$ and $(1.5, -2.35)$ are also examples of coordinates in the plane.

A point in three-dimensional space has three coordinates, which we may name x_1, x_2, and x_3. The common notation groups the numbers inside parentheses: (x_1, x_2, x_3). Alternatively, we may use the symbols x, y, and z, and write the point as (x, y, z), or numbers can be used instead of symbols.

From high school you may have a memory of solving two equations with two unknowns. At the university you will soon meet problems that are formulated as n equations with n unknowns. The solution of such problems contains n numbers that we can collect inside parentheses and number from 1 to n: $(x_1, x_2, x_3, \ldots, x_{n-1}, x_n)$.

Quantities such as (x, y), (x, y, z), or $(x_1, \ldots, x_n)$ are known as *vectors* in mathematics. A visual representation of a vector is an arrow that goes from the origin to a point. For example, the vector (x, y) is an arrow that goes from $(0, 0)$ to the point with coordinates (x, y) in the plane. Similarly, (x, y, z) is an arrow from $(0, 0, 0)$ to the point (x, y, z) in three-dimensional space.

Mathematicians found it convenient to introduce spaces with higher dimension than three, because when we have a solution of n equations collected in a vector $(x_1, \ldots, x_n)$, we may think of this vector as a point in a space with dimension n, or equivalently, an arrow that goes from the origin $(0, \ldots, 0)$ in n-dimensional space to the point $(x_1, \ldots, x_n)$. Figure 5.1 illustrates a vector as an arrow, either starting at the origin,

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

or at any other point. Two arrows/vectors that have the same direction and the same length are mathematically equivalent.

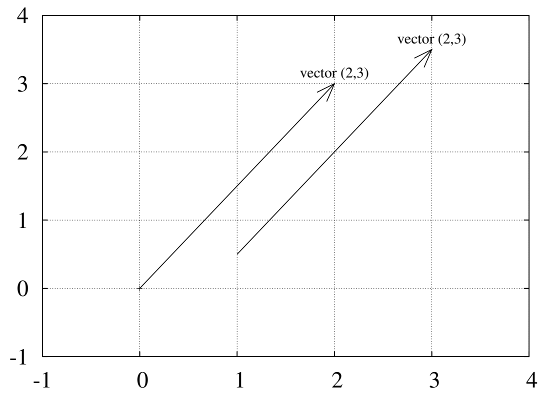


Fig. 5.1 A vector $(2,3)$ visualized in the standard way as an arrow from the origin to the point $(2,3)$, and mathematically equivalently, as an arrow from $(1,\frac{1}{2})$ (or any point (a,b)) to $(3,3\frac{1}{2})$ (or $(a+2,b+3)$).

We say that $(x_1,\ldots,x_n)$ is an n-vector or a vector with n components. Each of the numbers x_1, x_2, ... is a component or an element. We refer to the first component (or element), the second component (or element), and so forth.

A Python program may use a list or tuple to represent a vector:

```
v1 = [x, y]          # list of variables
v2 = (-1, 2)         # tuple of numbers
v3 = (x1, x2, x3)    # tuple of variables
from math import exp
v4 = [exp(-i*0.1) for i in range(150)]
```

While `v1` and `v2` are vectors in the plane and `v3` is a vector in three-dimensional space, `v4` is a vector in a 150-dimensional space, consisting of 150 values of the exponential function. Since Python lists and tuples have 0 as the first index, we may also in mathematics write the vector (x_1,x_2) as (x_0,x_1). This is not at all common in mathematics, but makes the distance from a mathematical description of a problem to its solution in Python shorter.

It is impossible to visually demonstrate how a space with 150 dimensions looks like. Going from the plane to three-dimensional space gives a rough feeling of what it means to add a dimension, but if we forget about the idea of a visual perception of space, the mathematics is very simple: going from a 4-dimensional vector to a 5-dimensional vector is just as easy as adding an element to a list of symbols or numbers.

5.1.2 Mathematical operations on vectors

Since vectors can be viewed as arrows having a length and a direction, vectors are extremely useful in geometry and physics. The velocity of a car has a magnitude and a direction, so has the acceleration, and the position of a point in the car is also a vector. An edge of a triangle can be viewed as a line (arrow) with a direction and length.

In geometric and physical applications of vectors, mathematical operations on vectors are important. We shall exemplify some of the most important operations on vectors below. The goal is not to teach computations with vectors, but more to illustrate that such computations are defined by mathematical rules. Given two vectors, (u_1, u_2) and (v_1, v_2), we can add these vectors according to the rule:

$$(u_1, u_2) + (v_1, v_2) = (u_1 + v_1, u_2 + v_2)\,. \tag{5.1}$$

We can also subtract two vectors using a similar rule:

$$(u_1, u_2) - (v_1, v_2) = (u_1 - v_1, u_2 - v_2)\,. \tag{5.2}$$

A vector can be multiplied by a number. This number, called a below, is usually denoted as a *scalar*:

$$a \cdot (v_1, v_2) = (av_1, av_2)\,. \tag{5.3}$$

The inner product, also called dot product, or scalar product, of two vectors is a number:

$$(u_1, u_2) \cdot (v_1, v_2) = u_1v_1 + u_2v_2\,. \tag{5.4}$$

(From high school mathematics and physics you might recall that the inner or dot product also can be expressed as the product of the lengths of the two vectors multiplied by the cosine of the angle between them, but we will not make use of that formula. There is also a *cross product* defined for 2-vectors or 3-vectors, but we do not list the cross product formula here.)

The length of a vector is defined by

$$||(v_1, v_2)|| = \sqrt{(v_1, v_2) \cdot (v_1, v_2)} = \sqrt{v_1^2 + v_2^2}\,. \tag{5.5}$$

The same mathematical operations apply to n-dimensional vectors as well. Instead of counting indices from 1, as we usually do in mathematics, we now count from 0, as in Python. The addition and subtraction of two vectors with n components (or elements) read

$$(u_0, \ldots, u_{n-1}) + (v_0, \ldots, v_{n-1}) = (u_0 + v_0, \ldots, u_{n-1} + v_{n-1}), \tag{5.6}$$

$$(u_0, \ldots, u_{n-1}) - (v_0, \ldots, v_{n-1}) = (u_0 - v_0, \ldots, u_{n-1} - v_{n-1})\,. \tag{5.7}$$

Multiplication of a scalar a and a vector $(v_0, \ldots, v_{n-1})$ equals

$$(av_0, \ldots, av_{n-1}) \,. \tag{5.8}$$

The inner or dot product of two n-vectors is defined as

$$(u_0, \ldots, u_{n-1}) \cdot (v_0, \ldots, v_{n-1}) = u_0v_0 + \cdots + u_{n-1}v_{n-1} = \sum_{j=0}^{n-1} u_jv_j \,. \tag{5.9}$$

Finally, the length $||v||$ of an n-vector $v = (v_0, \ldots, v_{n-1})$ is

$$\begin{aligned}\sqrt{(v_0, \ldots, v_{n-1}) \cdot (v_0, \ldots, v_{n-1})} &= \left(v_0^2 + v_1^2 + \cdots + v_{n-1}^2\right)^{\frac{1}{2}} \\ &= \left(\sum_{j=0}^{n-1} v_j^2\right)^{\frac{1}{2}} \,. \end{aligned} \tag{5.10}$$

5.1.3 Vector arithmetics and vector functions

In addition to the operations on vectors in Section 5.1.2, which you might recall from high school mathematics, we can define other operations on vectors. This is very useful for speeding up programs. Unfortunately, the forthcoming vector operations are hardly treated in textbooks on mathematics, yet these operations play a significant role in mathematical software, especially in computing environment such as MATLAB, Octave, Python, and R.

Applying a mathematical function of one variable, $f(x)$, to a vector is defined as a vector where f is applied to each element. Let $v = (v_0, \ldots, v_{n-1})$ be a vector. Then

$$f(v) = (f(v_0), \ldots, f(v_{n-1})) \,.$$

For example, the sine of v is

$$\sin(v) = (\sin(v_0), \ldots, \sin(v_{n-1})) \,.$$

It follows that squaring a vector, or the more general operation of raising the vector to a power, can be defined as applying the operation to each element:

$$v^b = (v_0^b, \ldots, v_{n-1}^b) \,.$$

Another operation between two vectors that arises in computer programming of mathematics is the "asterisk" multiplication, defined as

$$u * v = (u_0v_0, u_1v_1, \ldots, u_{n-1}v_{n-1}) \,. \tag{5.11}$$

Adding a scalar to a vector or array can be defined as adding the scalar to each component. If a is a scalar and v a vector, we have

$$a + v = (a + v_0, \ldots, a + v_{n-1})\,.$$

A compound vector expression may look like

$$v^2 * \cos(v) * e^v + 2\,. \tag{5.12}$$

How do we calculate this expression? We use the normal rules of mathematics, working our way, term by term, from left to right, paying attention to the fact that powers are evaluated before multiplications and divisions, which are evaluated prior to addition and subtraction. First we calculate v^2, which results in a vector we may call u. Then we calculate $\cos(v)$ and call the result p. Then we multiply $u * p$ to get a vector which we may call w. The next step is to evaluate e^v, call the result q, followed by the multiplication $w * q$, whose result is stored as r. Then we add $r + 2$ to get the final result. It might be more convenient to list these operations after each other:

- $u = v^2$
- $p = \cos(v)$
- $w = u * p$
- $q = e^v$
- $r = w * q$
- $s = r + 2$

Writing out the vectors u, w, p, q, and r in terms of a general vector $v = (v_0, \ldots, v_{n-1})$ (do it!) shows that the result of the expression (5.12) is the vector

$$(v_0^2 \cos(v_0) e^{v_0} + 2, \ldots, v_{n-1}^2 \cos(v_{n-1}) e^{v_{n-1}} + 2)\,.$$

That is, component no. i in the result vector equals the number arising from applying the formula (5.12) to v_i, where the * multiplication is ordinary multiplication between two numbers.

We can, alternatively, introduce the function

$$f(x) = x^2 \cos(x) e^x + 2$$

and use the result that $f(v)$ means applying f to each element in v. The result is the same as in the vector expression (5.12).

In Python programming it is important for speed (and convenience too) that we can apply functions of one variable, like $f(x)$, to vectors. What this means mathematically is something we have tried to explain in this subsection. Doing Exercises 5.5 and 5.6 may help to grasp the ideas of vector computing, and with more programming experience you will hopefully discover that vector computing is very useful. It is not necessary to have a thorough understanding of vector computing in order to proceed with the next sections.

Arrays are used to represent vectors in a program, but one can do more with arrays than with vectors. Until Section 5.7 it suffices to think of arrays as the same as vectors in a program.

5.2 Arrays in Python programs

This section introduces array programming in Python, but first we create some lists and show how arrays differ from lists.

5.2.1 Using lists for collecting function data

Suppose we have a function $f(x)$ and want to evaluate this function at a number of x points $x_0, x_1, \ldots, x_{n-1}$. We could collect the n pairs $(x_i, f(x_i))$ in a list, or we could collect all the x_i values, for $i = 0, \ldots, n-1$, in a list and all the associated $f(x_i)$ values in another list. The following interactive session demonstrates how to create these three types of lists:

```
>>> def f(x):
...     return x**3       # sample function
...
>>> n = 5                 # no of points along the x axis
>>> dx = 1.0/(n-1)        # spacing between x points in [0,1]
>>> xlist = [i*dx for i in range(n)]
>>> ylist = [f(x) for x in xlist]
>>> pairs = [[x, y] for x, y in zip(xlist, ylist)]
```

Here we have used list comprehensions for achieving compact code. Make sure that you understand what is going on in these list comprehensions (if not, try to write the same code using standard `for` loops and appending new list elements in each pass of the loops).

The list elements consist of objects of the same type: any element in `pairs` is a list of two `float` objects, while any element in `xlist` or `ylist` is a `float`. Lists are more flexible than that, because an element can be an object of any type, e.g.,

```
mylist = [2, 6.0, 'tmp.pdf', [0,1]]
```

Here `mylist` holds an `int`, a `float`, a string, and a list. This combination of diverse object types makes up what is known as *heterogeneous* lists. We can also easily remove elements from a list or add new elements anywhere in the list. This flexibility of lists is in general convenient to have as a programmer, but in cases where the elements are of the same type and the number of elements is fixed, arrays can be used instead. The benefits of arrays are faster computations, less memory demands, and extensive support for mathematical operations on the data. Because of greater efficiency and mathematical convenience, arrays will be used to a large extent in this book. The great use of arrays is also prominent in other programming environments such as MATLAB, Octave, and R, for instance. Lists will be our choice instead of arrays when we need the flexibility of adding or removing elements or when the elements may be of different object types.

5.2.2 Basics of numerical Python arrays

An *array* object can be viewed as a variant of a list, but with the following assumptions and features:

- All elements must be of the same type, preferably integer, real, or complex numbers, for efficient numerical computing and storage.
- The number of elements must be known when the array is created.
- Arrays are not part of standard Python - one needs an additional package called *Numerical Python*, often abbreviated as NumPy. The Python name of the package, to be used in `import` statements, is `numpy`.
- With `numpy`, a wide range of mathematical operations can be done directly on complete arrays, thereby removing the need for loops over array elements. This is commonly called *vectorization*
- Arrays with one index are often called vectors. Arrays with two indices are used as an efficient data structure for tables, instead of lists of lists. Arrays can also have three or more indices.

We have two remarks to the above list. First, there is actually an object type called `array` in standard Python, but this data type is not so efficient for mathematical computations, and we will not use it in this book. Second, the number of elements in an array *can* be changed, but at a substantial computational cost.

The following text lists some important functionality of NumPy arrays. A more comprehensive treatment is found in the excellent *NumPy Tutorial*, *NumPy User Guide*, *NumPy Reference*, *Guide to NumPy*, and *NumPy for MATLAB Users*, all accessible at scipy.org[2].

The standard import statement for Numerical Python reads

```
import numpy as np
```

To convert a list `r` to an array, we use the `array` function from `numpy`:

```
a = np.array(r)
```

To create a new array of length `n`, filled with zeros, we write

[2] `http://scipy.org`

```
a = np.zeros(n)
```

The array elements are of a type that corresponds to Python's `float` type. A second argument to `np.zeros` can be used to specify other element types, e.g., `int`. A similar function,

```
a = np.zeros_like(c)
```

generates an array of zeros where the length is that of the array `c` and the element type is the same as those in `c`. Arrays with more than one index are treated in Section 5.7.

Often one wants an array to have n elements with uniformly distributed values in an interval $[p, q]$. The `numpy` function `linspace` creates such arrays:

```
a = np.linspace(p, q, n)
```

Array elements are accessed by square brackets as for lists: `a[i]`. Slices also work as for lists, for example, `a[1:-1]` picks out all elements except the first and the last, but contrary to lists, `a[1:-1]` is not a copy of the data in `a`. Hence,

```
b = a[1:-1]
b[2] = 0.1
```

will also change `a[3]` to `0.1`. A slice `a[i:j:s]` picks out the elements starting with index `i` and stepping `s` indices at the time up to, but not including, `j`. Omitting `i` implies `i=0`, and omitting `j` implies `j=n` if `n` is the number of elements in the array. For example, `a[0:-1:2]` picks out every two elements up to, but not including, the last element, while `a[::4]` picks out every four elements in the whole array.

Remarks on importing NumPy

The statement

```
import numpy as np
```

with subsequent prefixing of all NumPy functions and variables by `np`, has evolved as a standard syntax in the Python scientific computing community. However, to make Python programs look closer to MATLAB and ease the transition to and from that language, one can do

```
from numpy import *
```

to get rid of the prefix (this is evolved as the standard in *interactive* Python shells). This author prefers mathematical functions from numpy to be written without the prefix to make the formulas as close as possible to the mathematics. So, $f(x) = \sinh(x-1)\sin(wt)$ would be coded as

```
from numpy import sinh, sin

def f(x):
    return sinh(x-1)*sin(w*t)
```

or one may take the less recommended lazy approach `from numpy import *` and fill up the program with *a lot* of functions and variables from `numpy`.

5.2.3 Computing coordinates and function values

With these basic operations at hand, we can continue the session from the previous section and make arrays out of the lists `xlist` and `ylist`:

```
>>> import numpy as np
>>> x2 = np.array(xlist)      # turn list xlist into array x2
>>> y2 = np.array(ylist)
>>> x2
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
>>> y2
array([ 0.      ,  0.015625,  0.125   ,  0.421875,  1.      ])
```

Instead of first making a list and then converting the list to an array, we can compute the arrays directly. The equally spaced coordinates in `x2` are naturally computed by the `np.linspace` function. The `y2` array can be created by `np.zeros`, to ensure that `y2` has the right length `len(x2)`, and then we can run a `for` loop to fill in all elements in `y2` with `f` values:

```
>>> n = len(xlist)
>>> x2 = np.linspace(0, 1, n)
>>> y2 = np.zeros(n)
>>> for i in xrange(n):
...     y2[i] = f(x2[i])
...
>>> y2
array([ 0.      ,  0.015625,  0.125   ,  0.421875,  1.      ])
```

Note that we here in the `for` loop have used `xrange` instead of `range`. The former is faster for long loops because it avoids generating *and storing* a list of integers, it just generates the values one by one. Hence, we prefer `xrange` over `range` for loops over long arrays. In Python version 3.x, `range` is the same as `xrange`.

Creating an array of a given length is frequently referred to as *allocating* the array. It simply means that a part of the computer's memory is marked for being occupied by this array. Mathematical computations will often fill up most of the computer's memory by allocating long arrays.

We can shorten the previous code by creating the `y2` data in a list comprehension, but list comprehensions produce lists, not arrays, so we need to transform the list object to an array object:

```
>>> x2 = np.linspace(0, 1, n)
>>> y2 = np.array([f(xi) for xi in x2])
```

Nevertheless, there is a much faster way of computing `y2` as the next paragraph explains.

5.2.4 Vectorization

Loops over very long arrays may run slowly. A great advantage with arrays is that we can get rid of the loops and apply `f` directly to the whole array:

```
>>> y2 = f(x2)
>>> y2
array([ 0.      ,  0.015625,  0.125   ,  0.421875,  1.      ])
```

The magic that makes `f(x2)` work builds on the vector computing concepts from Section 5.1.3. Instead of calling `f(x2)` we can equivalently write the function formula `x2**3` directly.

The point is that `numpy` implements vector arithmetics *for arrays* of any dimension. Moreover, `numpy` provides its own versions of mathematical functions like `cos`, `sin`, `exp`, `log`, etc., which work for array arguments and apply the mathematical function to each element. The following code, computes each array element separately:

```
from math import sin, cos, exp
import numpy as np
x = np.linspace(0, 2, 201)
r = np.zeros(len(x))
for i in xrange(len(x)):
    r[i] = sin(np.pi*x[i])*cos(x[i])*exp(-x[i]**2) + 2 + x[i]**2
```

while here is a corresponding code that operates on arrays directly:

```
r = np.sin(np.pi*x)*np.cos(x)*np.exp(-x**2) + 2 + x**2
```

Many will prefer to see such formulas without the `np` prefix:

```
from numpy import sin, cos, exp, pi
r = sin(pi*x)*cos(x)*exp(-x**2) + 2 + x**2
```

An important thing to understand is that `sin` from the `math` module is different from the `sin` function provided by `numpy`. The former does not allow array arguments, while the latter accepts both real numbers and arrays.

Replacing a loop like the one above, for computing `r[i]`, by a vector/array expression like `sin(x)*cos(x)*exp(-x**2) + 2 + x**2`, is called *vectorization*. The loop version is often referred to as *scalar code*. For example,

```
import numpy as np
import math
x = np.zeros(N);  y = np.zeros(N)
dx = 2.0/(N-1) # spacing of x coordinates
for i in range(N):
    x[i] = -1 + dx*i
    y[i] = math.exp(-x[i])*x[i]
```

is scalar code, while the corresponding vectorized version reads

```
x = np.linspace(-1, 1, N)
y = np.exp(-x)*x
```

We remark that list comprehensions,

```
x = array([-1 + dx*i for i in range(N)])
y = array([np.exp(-xi)*xi for xi in x])
```

result in scalar code because we still have explicit, slow Python `for` loops operating on scalar quantities. The requirement of vectorized code is that there are no explicit Python `for` loops. The loops required to compute each array element are performed in fast C or Fortran code in the `numpy` package.

Most Python functions intended for a scalar argument `x`, like

```
def f(x):
    return x**4*exp(-x)
```

automatically work for an array argument `x`:

```
x = np.linspace(-3, 3, 101)
y = f(x)
```

provided that the `exp` function in the definition of `f` accepts an array argument. This means that `exp` must have been imported as `from numpy import *` or explicitly as `from numpy import exp`. One can, of course, prefix `exp` as in `np.exp`, at the loss of a less attractive mathematical syntax in the formula.

When a Python function `f(x)` works for an array argument `x`, we say that the function `f` is vectorized. Provided that the mathematical expressions in `f` involve arithmetic operations and basic mathematical functions from the `math` module, `f` will be automatically vectorized by just importing the functions from `numpy` instead of `math`. However, if the expressions inside `f` involve `if` tests, the code needs a rewrite to work with arrays. Section 5.4.1 presents examples where we have to do special actions in order to vectorize functions.

Vectorization is very important for speeding up Python programs that perform heavy computations with arrays. Moreover, vectorization gives more compact code that is easier to read. Vectorization is particularly important for statistical simulations, as demonstrated in Chapter 8.

5.3 Curve plotting

Visualizing a function $f(x)$ is done by drawing the curve $y = f(x)$ in an xy coordinate system. When we use a computer to do this task, we say that we *plot* the curve. Technically, we plot a curve by drawing straight lines between n points on the curve. The more points we use, the smoother the curve appears.

Suppose we want to plot the function $f(x)$ for $a \leq x \leq b$. First we pick out n x coordinates in the interval $[a, b]$, say we name these $x_0, x_1, \ldots, x_{n-1}$. Then we evaluate $y_i = f(x_i)$ for $i = 0, 1, \ldots, n-1$. The points (x_i, y_i), $i = 0, 1, \ldots, n-1$, now lie on the curve $y = f(x)$. Normally, we choose the x_i coordinates to be equally spaced, i.e.,

$$x_i = a + ih, \quad h = \frac{b-a}{n-1}\,.$$

If we store the x_i and y_i values in two arrays `x` and `y`, we can plot the curve by a command like `plot(x,y)`.

Sometimes the names of the independent variable and the function differ from x and f, but the plotting procedure is the same. Our first example of curve plotting demonstrates this fact by involving a function of t.

5.3.1 Matplotlib; pylab

The standard package for curve plotting in Python is Matplotlib. First we exemplify Matplotlib using `matplotlib.pylab`, which enables a syntax very close to that of MATLAB. This is a great advantage since many readers may have experience with plotting in MATLAB, or they will certainly meet MATLAB sometime in their scientific work.

A basic plot. Let us plot the curve $y = t^2 \exp(-t^2)$ for t values between 0 and 3. First we generate equally spaced coordinates for t, say 51 values (50 intervals). Then we compute the corresponding y values at these points, before we call the `plot(t,y)` command to make the curve plot. Here is the complete program:

```
from matplotlib.pylab import *

def f(t):
    return t**2*exp(-t**2)

t = linspace(0, 3, 31)    # 31 points between 0 and 3
y = zeros(len(t))         # allocate y with float elements
for i in xrange(len(t)):
    y[i] = f(t[i])

plot(t, y)
show()
```

The `from matplotlib.pylab import *` performs a `from numpy import *` import as well as an import of all Matplotlib commands that resemble MATLAB-style syntax. In this program we pre-allocate the `y` array and fill it with values, element by element, in a Python loop. Alternatively, we may operate on the whole `t` array at once, which yields faster and shorter code:

```
y = f(t)
```

To include the plot in electronic documents, we need a hardcopy of the figure in PDF, PNG, or another image format. The `savefig` function saves the plot to files in various image formats:

```
savefig('tmp1.pdf') # produce PDF
savefig('tmp1.png') # produce PNG
```

The filename extension determines the format: `.pdf` for PDF and `.png` for PNG. Figure 5.2 displays the resulting plot.

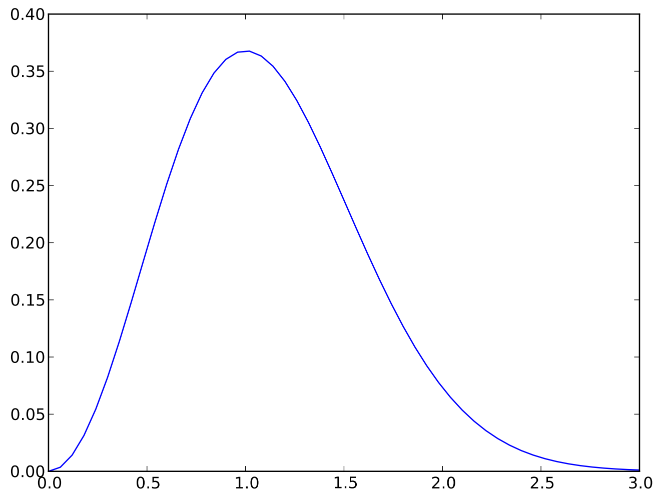


Fig. 5.2 A simple plot in PDF format (Matplotlib).

Decorating the plot. The x and y axes in curve plots should have labels, here t and y, respectively. Also, the curve should be identified with a label, or legend as it is often called. A title above the plot is also common. In addition, we may want to control the extent of the axes (although most plotting programs will automatically adjust the axes to the range of the data). All such things are easily added after the `plot` command:

```
plot(t, y)
xlabel('t')
ylabel('y')
legend(['t^2*exp(-t^2)'])
axis([0, 3, -0.05, 0.6])   # [tmin, tmax, ymin, ymax]
title('My First Matplotlib Demo')
savefig('tmp2.pdf')
show()
```

Removing the `show()` call prevents the plot from being shown on the screen, which is advantageous if the program's purpose is to make a large number of plots in PDF or PNG format (you do not want all the plot windows to appear on the screen and then kill all of them manually). This decorated plot is displayed in Figure 5.3.

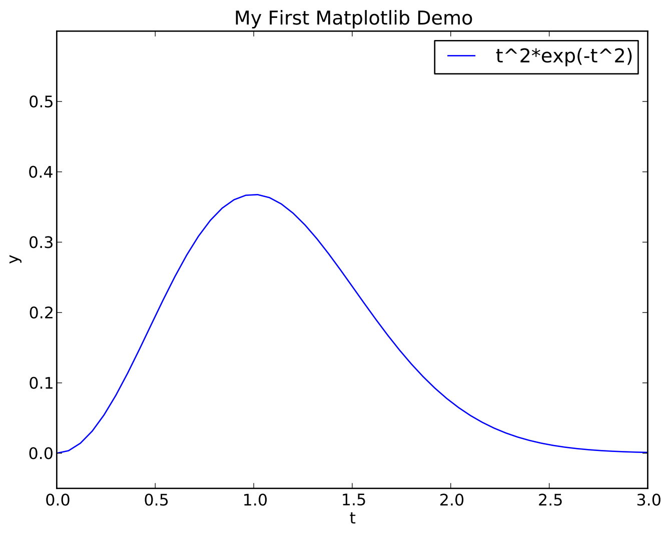


Fig. 5.3 A single curve with label, title, and axis adjusted (Matplotlib).

Plotting multiple curves. A common plotting task is to compare two or more curves, which requires multiple curves to be drawn in the same plot. Suppose we want to plot the two functions $f_1(t) = t^2 \exp(-t^2)$ and $f_2(t) = t^4 \exp(-t^2)$. We can then just issue two `plot` commands, one for each function. To make the syntax resemble MATLAB, we call `hold('on')` after the first `plot` command to indicate that subsequent `plot` commands are to draw the curves in the first plot.

```
def f1(t):
    return t**2*exp(-t**2)

def f2(t):
    return t**2*f1(t)

t = linspace(0, 3, 51)
y1 = f1(t)
```

```
y2 = f2(t)

plot(t, y1, 'r-')
hold('on')
plot(t, y2, 'bo')
xlabel('t')
ylabel('y')
legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
title('Plotting two curves in the same plot')
show()
```

In these `plot` commands, we have also specified the line type: `r-` means red (`r`) line (`-`), while `bo` means a blue (`b`) circle (`o`) at each data point. Figure 5.4 shows the result. The legends for each curve is specified in a list where the sequence of strings correspond to the sequence of `plot` commands. Doing a `hold('off')` makes the next `plot` command create a new plot.

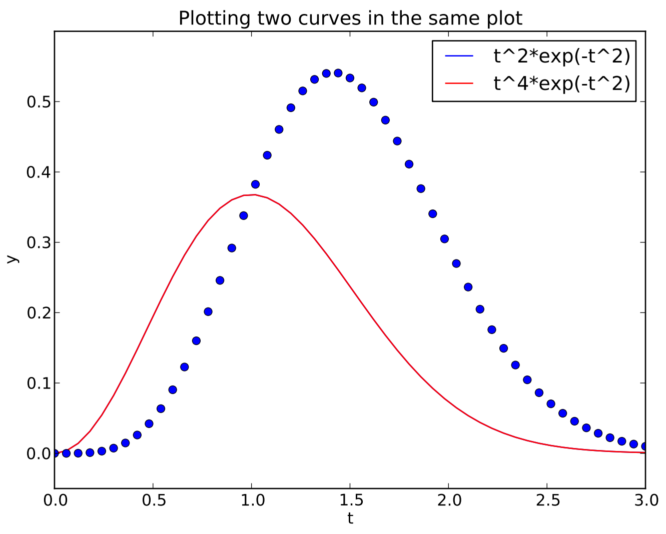


Fig. 5.4 Two curves in the same plot (Matplotlib).

Placing several plots in one figure. We may also put plots together in a figure with `r` rows and `c` columns of plots. The `subplot(r,c,a)` does this, where `a` is a row-wise counter for the individual plots. Here is an example with two rows of plots, and one plot in each row, (see Figure 5.5):

```
figure()  # make separate figure
subplot(2, 1, 1)
t = linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)

plot(t, y1, 'r-', t, y2, 'bo')
xlabel('t')
```

```
ylabel('y')
axis([t[0], t[-1], min(y2)-0.05, max(y2)+0.5])
legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
title('Top figure')

subplot(2, 1, 2)
t3 = t[::4]
y3 = f2(t3)

plot(t, y1, 'b-', t3, y3, 'ys')
xlabel('t')
ylabel('y')
axis([0, 4, -0.2, 0.6])
legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
savefig('tmp4.pdf')
show()
```

The `figure()` call creates a new plot window on the screen.

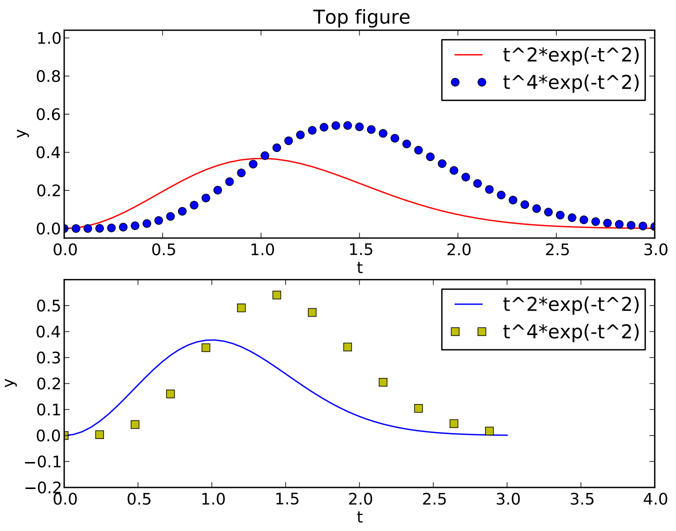


Fig. 5.5 Example on two plots in one figure (Matplotlib).

All of the examples above on plotting with Matplotlib are collected in the file `mpl_pylab_examples.py`.

5.3.2 Matplotlib; pyplot

The Matplotlib developers do not promote the `matplotlib.pylab` interface. Instead, they recommend the `matplotlib.pyplot` module and prefix Numerical Python and Matplotlib functionality by short forms of their package names:

```
import numpy as np
import matplotlib.pyplot as plt
```

The commands in `matplotlib.pyplot` are similar to those in `matplotlib.pylab`. The plot in Figure 5.3 can typically be obtained by prefixing the `pylab` commands with `plt`:

```
plt.plot(t, y)
plt.legend(['t^2*exp(-t^2)'])
plt.xlabel('t')
plt.ylabel('y')
plt.axis([0, 3, -0.05, 0.6])   # [tmin, tmax, ymin, ymax]
plt.title('My First Matplotlib Demo')
plt.show()
plt.savefig('tmp2.pdf') # produce PDF
```

Instead of giving plot data and legends separately, it is more common to write

```
plt.plot(t, y, label='t^2*exp(-t^2)')
```

However, in this book we shall stick to the `legend` command since this makes the transition to/from MATLAB easier.

Figure 5.4 can be produced by

```
def f1(t):
    return t**2*np.exp(-t**2)

def f2(t):
    return t**2*f1(t)

t = np.linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)

plt.plot(t, y1, 'r-')
plt.plot(t, y2, 'bo')
plt.xlabel('t')
plt.ylabel('y')
plt.legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
plt.title('Plotting two curves in the same plot')
plt.savefig('tmp3.pdf')
plt.show()
```

Putting multiple plots in a figure follows the same set-up with `subplot` as shown for `pylab`, except that commands are prefixed by `plt`. The complete example, along with the codes listed above, are found in the file `mpl_pyplot_examples.py`.

The `plt` prefix is not a requirement for using `matplotlib.pyplot`, as one can equally well do `from matplotlib.pyplot import *` and get access to the same functions as shown in Section 5.3.1 for the `pylab` functions.

Once you have created a basic plot, there are numerous possibilities for fine-tuning the figure, i.e., adjusting tick marks on the axis, inserting text, etc. The Matplotlib website is full of instructive examples on what you can do with this excellent package.

5.3.3 SciTools and Easyviz

Matplotlib has become the *de facto* standard for curve plotting in Python, but there are several other alternative packages, especially if we also consider plotting of 2D/3D scalar and vector fields. Python has interfaces to many leading visualization packages: MATLAB, Gnuplot, Grace, OpenDX, and VTK. Even basic plotting with these packages has very different syntax, and deciding what package and syntax to go with was and still is a challenge. As a response to this challenge, Easyviz was created to provide a common uniform interface to all the mentioned visualization packages (including Matplotlib). The syntax of this interface was made very close to that of MATLAB, since most scientists and engineers have experience with MATLAB or most probably will be using it in some context. (In general, the Python syntax used in the examples in this book is constructed to ease the transition to and from MATLAB.)

Easyviz is part of the SciTools package, which consists of a set of Python tools building on Numerical Python, ScientificPython, the comprehensive SciPy environment, and other packages for scientific computing with Python. SciTools contains in particular software related to the book [14] and the present text. Installation is straightforward as described on the web page `http://code.google.com/p/scitools`.

Importing SciTools and Easyviz. A standard import of SciTools is

```
from scitools.std import *
```

The advantage of this statement is that it, with a minimum of typing, imports a lot of useful modules for numerical Python programming: Easyviz for MATLAB-style plotting, all of `numpy` (`from numpy import *`), all of `scipy` (`from scipy import *`) if available, the `StringFunction` tool (see Section 4.3.3), many mathematical functions and tools in SciTools, plus commonly applied modules such as `sys`, `os`, and `math`. The imported standard mathematical functions (`sqrt`, `sin`, `asin`, `exp`, etc.) are from `numpy.lib.scimath` and deal transparently with real and complex input/output (as the corresponding MATLAB functions):

```
>>> from scitools.std import *
>>> a = array([-4., 4])
>>> sqrt(a)                # need complex output
array([ 0.+2.j,  2.+0.j])
>>> a = array([16., 4])
>>> sqrt(a)                # can reduce to real output
array([ 4.,  2.])
```

The inverse trigonometric functions have different names in `math` and `numpy`, a fact that prevents an expression written for scalars, using `math` names, to be immediately valid for arrays. Therefore, the `from scitools.std import *` action also imports the names `asin`, `acos`, and `atan` for the `numpy` or `scipy` names `arcsin`, `arccos`, and `arctan` functions, to ease vectorization of mathematical expressions involving inverse trigonometric functions.

The downside of the "star import" from `scitools.std` is twofold. First, it fills up your program or interactive session with the names of several hundred functions. Second, when using a particular function, you do not know the package it comes from. Both problems are solved by doing an import of the type used in Section 5.3.2:

```
import scitools.std as st
import numpy as np
```

All of the SciTools and Easyviz functions must then be prefixed by `st`. Although the `numpy` functions are available through the `st` prefix, we recommend using the `np` prefix to clearly see where functionality comes from.

Since the Easyviz syntax for plotting is very close to that of MATLAB, it is also very close to the syntax of Matplotlib shown earlier. This will be demonstrated in the forthcoming examples. The advantage of using Easyviz is that the underlying plotting package, used to create the graphics and known as a *backend*, can trivially be replaced by another package. If users of your Python software have not installed a particular visualization package, the software can still be used with another alternative (which might be considerably easier to install). By default, Easyviz now employs Matplotlib for plotting. Other popular alternatives are Gnuplot and MATLAB. For 2D/3D scalar and vector fields, VTK is a popular backend for Easyviz.

We shall next redo the curve plotting examples from Section 5.3.1 using Easyviz syntax.

A basic plot. Plotting the curve $y = t^2 \exp(-t^2)$ for $t \in [0, 3]$, using 31 equally spaced points (30 intervals) is performed by like this:

```
from scitools.std import *

def f(t):
    return t**2*exp(-t**2)

t = linspace(0, 3, 31)
y = f(t)
plot(t, y, '-')
```

To save the plot in a file, we use the `savefig` function, which takes the filename as argument:

```
savefig('tmp1.pdf') # produce PDF
savefig('tmp1.eps') # produce PostScript
savefig('tmp1.png') # produce PNG
```

The filename extension determines the format: `.pdf` for PDF, `.ps` or `.eps` for PostScript, and `.png` for PNG. A synonym for the `savefig` function is `hardcopy`.

What if the plot window quickly disappears?

On some platforms, some backends may result in a plot that is shown in just a fraction of a second on the screen before the plot window disappears (the Gnuplot backend on Windows machines and the Matplotlib backend constitute two examples). To make the window stay on the screen, add

```
raw_input('Press the Return key to quit: ')
```

at the end of the program. The plot window is killed when the program terminates, and this statement postpones the termination until the user hits the Return key.

Decorating the plot. Let us plot the same curve, but now with a legend, a plot title, labels on the axes, and specified ranges of the axes:

```
from scitools.std import *

def f(t):
    return t**2*exp(-t**2)

t = linspace(0, 3, 31)
y = f(t)
plot(t, y, '-')
xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)')
axis([0, 3, -0.05, 0.6])   # [tmin, tmax, ymin, ymax]
title('My First Easyviz Demo')
```

Easyviz has also introduced a more Pythonic `plot` command where all the plot properties can be set at once through keyword arguments:

```
plot(t, y, '-',
     xlabel='t',
     ylabel='y',
     legend='t^2*exp(-t^2)',
     axis=[0, 3, -0.05, 0.6],
     title='My First Easyviz Demo',
     savefig='tmp1.pdf',
     show=True)
```

With `show=False` one can avoid the plot window on the screen and just make the plot file.

Note that we in the curve legend write `t` square as `t^2` (LaTeX style) rather than `t**2` (program style). Whichever form you choose is up to you, but the LaTeX form sometimes looks better in some plotting programs (Matplotlib and Gnuplot are two examples).

Plotting multiple curves. Next we want to compare the two functions $f_1(t) = t^2 \exp(-t^2)$ and $f_2(t) = t^4 \exp(-t^2)$. Writing two `plot` commands after each other makes two separate plots. To make the second curve appear together with the first one, we need to issue a `hold('on')` call after the first `plot` command. All subsequent `plot` commands will then draw curves in the same plot, until `hold('off')` is called.

```
from scitools.std import *

def f1(t):
    return t**2*exp(-t**2)

def f2(t):
    return t**2*f1(t)

t = linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)

plot(t, y1, 'r-')
hold('on')
plot(t, y2, 'b-')

xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)', 't^4*exp(-t^2)')
title('Plotting two curves in the same plot')
savefig('tmp3.pdf')
```

The sequence of the multiple legends is such that the first legend corresponds to the first curve, the second legend to the second curve, and so forth.

Instead of separate calls to `plot` and the use of `hold('on')`, we can do everything at once and just send several curves to `plot`:

```
plot(t, y1, 'r-', t, y2, 'b-', xlabel='t', ylabel='y',
     legend=('t^2*exp(-t^2)', 't^4*exp(-t^2)'),
     title='Plotting two curves in the same plot',
     savefig='tmp3.pdf')
```

Throughout this book, we very often make use of this type of compact `plot` command, which also only requires an import of the form `from scitools.std import plot`.

Changing backend. Easyviz applies Matplotlib for plotting by default, so the resulting figures so far will be similar to those of Figure 5.2-5.4.

However, we can use other backends (plotting packages) for creating the graphics. The specification of what package to use is defined in a configuration file (see the heading *Setting Parameters in the Configuration File* in the Easyviz documentation), or on the command line:

Terminal

```
Terminal> python myprog.py --SCITOOLS_easyviz_backend gnuplot
```

Now, the plotting commands in `myprog.py` will make use of Gnuplot to create the graphics, with a slightly different result than that created by Matplotlib (compare Figures 5.4 and 5.6). A nice feature of Gnuplot is that the line types are automatically changed if we save a figure to file, such that the lines are easily distinguishable in a black-and-white plot. With Matplotlib one has to carefully set the line types to make them effective on a grayscale.

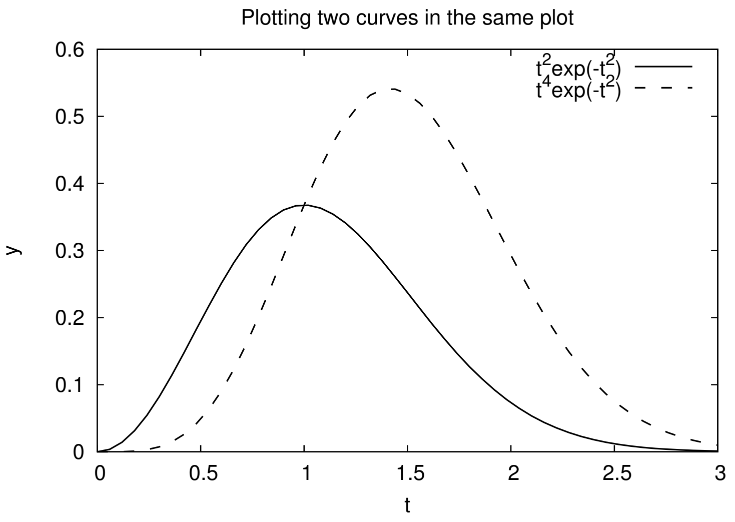


Fig. 5.6 Two curves in the same plot (Gnuplot).

Placing several plots in one figure. Finally, we redo the example from Section 5.3.1 where two plots are combined into one figure, using the `subplot` command:

```
figure()
subplot(2, 1, 1)
t = linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)
plot(t, y1, 'r-', t, y2, 'bo', xlabel='t', ylabel='y',
     legend=('t^2*exp(-t^2)', 't^4*exp(-t^2)'),
     axis=[t[0], t[-1], min(y2)-0.05, max(y2)+0.5],
     title='Top figure')

subplot(2, 1, 2)
t3 = t[::4]
y3 = f2(t3)

plot(t, y1, 'b-', t3, y3, 'ys',
     xlabel='t', ylabel='y',
     axis=[0, 4, -0.2, 0.6],
     legend=('t^2*exp(-t^2)', 't^4*exp(-t^2)'))
savefig('tmp4.pdf')
```

Note that `figure()` must be used if you want a program to make different plot windows on the screen: each `figure()` call creates a new, separate plot.

All of the Easyviz examples above are found in the file `easyviz_examples.py`. We remark that Easyviz is just a thin layer of code providing access to the most common plotting functionality for curves as well as 2D/3D scalar and vector fields. Fine-tuning of plots, e.g., specifying tick marks on the axes, is not supported, simply because most of the curve plots in the daily work can be made without such functionality. For fine-tuning the plot with special commands, you need to grab an object in Easyviz that communicates directly with the underlying plotting package used to create the graphics. With this object you can issue package-specific commands and do whatever the underlying package allows you do. This is explained in the Easyviz manual[3], which also comes up by running `pydoc scitools.easyviz`. As soon as you have digested the very basics of plotting, you are strongly recommend to read through the curve plotting part of the Easyviz manual.

5.3.4 Making animations

A sequence of plots can be combined into an animation on the screen and stored in a video file. The standard procedure is to generate a series of individual plots and to show them in sequence to obtain an animation effect. Plots store in files can be combined to a video file.

Example. The function

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

is known as the Gaussian function or the probability density function of the normal (or Gaussian) distribution. This bell-shaped function is wide for large s and peak-formed for small s, see Figure 5.7. The function is symmetric around $x = m$ ($m = 0$ in the figure). Our goal is to make an animation where we see how this function evolves as s is decreased. In Python we implement the formula above as a function `f(x, m, s)`.

The animation is created by varying s in a loop and for each s issue a `plot` command. A moving curve is then visible on the screen. One can also make a video that can be played as any other computer movie using a standard movie player. To this end, each plot is saved to a file, and all the files are combined together using some suitable tool to be explained later. Before going into programming detail there is one key point to emphasize.

Keep the extent of axes fixed during animations!

The underlying plotting program will normally adjust the axis to the maximum and minimum values of the curve if we do not specify the axis ranges explicitly. For an animation such automatic axis

[3] `https://scitools.googlecode.com/hg/doc/easyviz/easyviz.html`

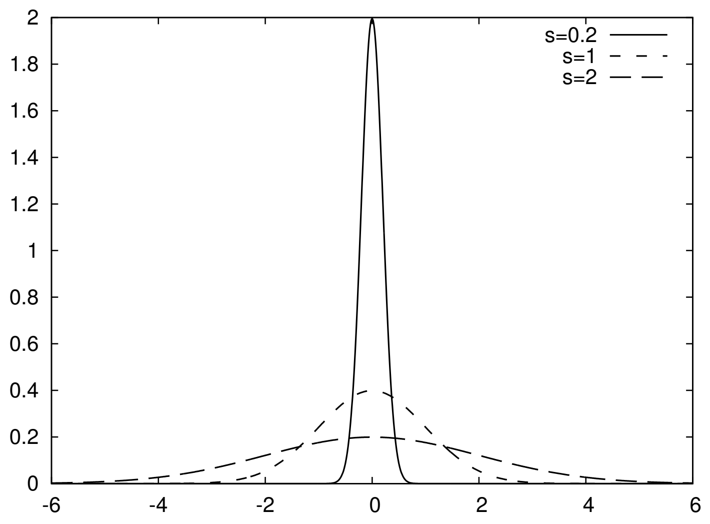


Fig. 5.7 Different shapes of a Gaussian function.

adjustment is misleading - any axis range must be fixed to avoid a jumping axis.

The relevant values for the y axis range in the present example is the minimum and maximum value of f. The minimum value is zero, while the maximum value appears for $x = m$ and increases with decreasing s. The range of the y axis must therefore be $[0, f(m; m, \min s)]$.

The function f is defined for all $-\infty < x < \infty$, but the function value is very small already $3s$ away from $x = m$. We may therefore limit the x coordinates to $[m - 3s, m + 3s]$.

Animation in Easyviz. We start with using Easyviz for animation since this is almost identical to making standard static plots, and you can choose the plotting engine you want to use, say Gunplot or Matplotlib. The Easyviz recipe for animating the Gaussian function as s goes from 2 to 0.2 looks as follows.

```
from scitools.std import sqrt, pi, exp, linspace, plot, movie
import time

def f(x, m, s):
    return (1.0/(sqrt(2*pi)*s))*exp(-0.5*((x-m)/s)**2)

m = 0
s_min = 0.2
s_max = 2
x = linspace(m -3*s_max, m + 3*s_max, 1000)
s_values = linspace(s_max, s_min, 30)
# f is max for x=m; smaller s gives larger max value
max_f = f(m, m, s_min)

# Show the movie on the screen
# and make hardcopies of frames simultaneously.
```

```
counter = 0
for s in s_values:
    y = f(x, m, s)
    plot(x, y, '-', axis=[x[0], x[-1], -0.1, max_f],
         xlabel='x', ylabel='f', legend='s=%4.2f' % s,
         savefig='tmp%04d.png' % counter)
    counter += 1
    #time.sleep(0.2)  # can insert a pause to control movie speed
```

Note that the s values are decreasing (`linspace` handles this automatically if the start value is greater than the stop value). Also note that we, simply because we think it is visually more attractive, let the y axis go from -0.1 although the f function is always greater than zero. The complete code is found in the file `movie1.py`.

Notice

It is crucial to use the single, compound `plot` command shown above, where axis, labels, legends, etc., are set in the same call. Splitting up in individual calls to `plot`, `axis`, and so forth, results in jumping curves and axis. Also, when visualizing more than one animated curve at a time, make sure you send all data to a single `plot` command.

Remark on naming plot files

For each frame (plot) in the movie we store the plot in a file, with the purpose of combining all the files to an ordinary video file. The different files need different names such that various methods for listing the files will list them in the correct order. To this end, we recommend using filenames of the form `tmp0001.png`, `tmp0002.png`, `tmp0003.png`, etc. The printf format `04d` pads the integers with zeros such that `1` becomes `0001`, `13` becomes `0013` and so on. The expression `tmp*.png` will now expand (by an alphabetic sort) to a list of all files in proper order.

Without the padding with zeros, i.e., names of the form `tmp1.png`, `tmp2.png`, ..., `tmp12.png`, etc., the alphabetic order will give a wrong sequence of frames in the movie. For instance, `tmp12.png` will appear before `tmp2.png`.

Basic animation in Matplotlib. Animation is Matplotib requires more than a loop over a parameter and making a plot inside the loop. The set-up that is closest to standard static plots is shown first, while the newer and more widely used tool `FuncAnimation` is explained afterwards.

The first part of the program, where we define `f`, `x`, `s_values`, and so forth, is the same regardless of the animation technique. Therefore, we concentrate on the graphics part here:

```
import matplotlib.pyplot as plt
...
# Make a first plot
plt.ion()
y = f(x, m, s_max)
lines = plt.plot(x, y)
plt.axis([x[0], x[-1], -0.1, max_f])
plt.xlabel('x')
plt.ylabel('f')

# Show the movie, and make hardcopies of frames simulatenously
counter = 0
for s in s_values:
    y = f(x, m, s)
    lines[0].set_ydata(y)
    plt.legend(['s=%4.2f' % s])
    plt.draw()
    plt.savefig('tmp_%04d.png' % counter)
    counter += 1
```

The `plt.ion()` call is important, so is the first plot, where we grab the result of the `plot` command, which is a list of Matplotlib's `Line2D` objects. The idea is then to update the data via `lines[0].set_ydata` and show the plot via `plt.draw()` for each frame. For multiple curves we must update the y data for each curve, e.g.,

```
lines = plot(x, y1, x, y2, x, y3)

for parameter in parameters:
    y1 = ...
    y2 = ...
    y3 = ...
    for line, y in zip(lines, [y1, y2, y3]):
        line.set_ydata(y)
    plt.draw()
```

The file `movie1_mpl1.py` contains the complete program for doing animation with native Matplotlib syntax.

Using FuncAnimation in Matplotlib. The recommended approach to animation in Matplotlib is to use the `FuncAnimation` tool:

```
import matplotlib.pyplot as plt
from matplotlib.animation import animation

anim = animation.FuncAnimation(
    fig, frame, all_args, interval=150, init_func=init, blit=True)
```

Here, `fig` is the `plt.figure()` object for the current figure, `frame` is a user-defined function for plotting each frame, `all_args` is a list of arguments for `frame`, `interval` is the delay in ms between each frame, `init_func` is a function called for defining the background plot in the animation, and `blit=True` speeds up the animation. For frame number `i`, `FuncAnimation` will call `frame(all_args[i])`. Hence, the user's task is mostly to write the `frame` function and construct the `all_args` arguments.

After having defined `m`, `s_max`, `s_min`, `s_values`, and `max_f` as shown earlier, we have to make a first plot:

```
fig = plt.figure()
plt.axis([x[0], x[-1], -0.1, max_f])
lines = plt.plot([], [])
plt.xlabel('x')
plt.ylabel('f')
```

Notice that we save the return value of `plt.plot` in `lines` such that we can conveniently update the data for the curve(s) in each frame.

The function for defining a background plot draws an empty plot in this example:

```
def init():
    lines[0].set_data([], [])  # empty plot
    return lines
```

The function that defines the individual plots in the animation basically computes `y` from `f` and updates the data of the curve:

```
def frame(args):
    frame_no, s, x, lines = args
    y = f(x, m, s)
    lines[0].set_data(x, y)
    return lines
```

Multiple curves can be updated as shown earlier.

We are now ready to call `animation.FuncAnimation`:

```
anim = animation.FuncAnimation(
    fig, frame, all_args, interval=150, init_func=init, blit=True)
```

A common next action is to make a video file, here in the MP4 format with 5 frames per second:

```
anim.save('movie1.mp4', fps=5)   # movie in MP4 format
```

Finally, we must `plt.show()` as always to watch any plots on the screen.

The video making requires additional software on the computer, such as `ffmpeg`, and can fail. One gets more control over the potentially fragile movie making process by explicitly saving plots to file and explicitly running movie making programs like `ffmeg` later. Such programs are explained in Section 5.3.5.

The complete code showing the basic use of `FuncAnimation` is available in `movie1_FuncAnimation.py`. There is also a Matlab Animation Tutorial with more basic information, plus a set of animation examples on `http://matplotlib.org/examples`.

Remove old plot files!

We strongly recommend removing previously generated plot files before a new set of files is made. Otherwise, the movie may get old and new files mixed up. The following Python code removes all files of the form `tmp*.png`:

```
import glob, os
for filename in glob.glob('tmp*.png'):
    os.remove(filename)
```

These code lines should be inserted at the beginning of programs or functions performing animations.

Instead of deleting the individual plotfiles, one may store all plot files in a subfolder and later delete the subfolder. Here is a suitable code segment:

```
import shutil, os
subdir = 'temp'             # subfolder name for plot files
if os.path.isdir(subdir):   # does the subfolder already exist?
    shutil.rmtree(subdir)   # delete the whole folder
os.mkdir(subdir)            # make new subfolder
os.chdir(subdir)            # move to subfolder
# ... perform all the plotting, make movie ...
os.chdir(os.pardir)         # optional: move up to parent folder
```

Note that Python and many other languages use the word directory instead of folder. Consequently, the name of functions dealing with folders have a name containing `dir` for directory.

5.3.5 Making videos

Suppose we have a set of frames in an animation, saved as plot files `tmp_*.png`. The filenames are generated by the printf syntax `'tmp_%04d.png' % i`, using a frame counter `i` that goes from 0 to some value. The corresponding files are then `tmp_0000.png`, `tmp_0001.png`, `tmp_0002.png`, and so on. Several tools can be used to create videos in common formats from the individual frames in the plot files.

Animated GIF file. The ImageMagick[4] software suite contains a program `convert` for making animated GIF files:

Terminal

```
Terminal> convert -delay 50 tmp_*.png movie.gif
```

The delay between frames, here 50, is measured in units of 1/100 s. The resulting animated GIF file `movie.gif` can be viewed by another

[4] `http://www.imagemagick.org/`

program in the ImageMagick suite: `animate movie.gif`, but the most common way of displaying animated GIF files is to include them in web pages. Writing the HTML code

```
<img src="movie.gif">
```

in some file with extension `.html` and loading this file into a web browser will play the movie repeatedly. You may try this out online[5].

MP4, Ogg, WebM, and Flash videos. The modern video formats that are best suited for being displayed in web browsers are MP4, Ogg, WebM, and Flash. The program `ffmpeg`, or the almost equivalent `avconv`, is a common tool to create such movies. Creating a flash video is done by

```
Terminal

Terminal> ffmpeg -r 5 -i tmp_%04d.png -vcodec flv movie.flv
```

The `-r` option specifies the number of frames per second, here 5, the `-i` option specifies the printf string that was used to make the names of the individual plot files, `-vcodec` is the video codec for Flash, which is called `flv`, and the final argument is the name of the video file. On Debian Linux systems, such as Ubuntu, you use the `avconv` program instead of `ffmpeg`.

Other formats are created in the same way, but we need to specify the codec and use the right extension in the video file:

Format	Codec and filename
Flash	`-vcodec flv movie.flv`
MP4	`-vcodec libx264 movie.mp4`
Webm	`-vcodec libvpx movie.webm`
Ogg	`-vcodec libtheora movie.ogg`

Video files are normally trivial to play in graphical file browser: double lick the filename or right-click and choose a player. On Linux systems there are several players that can be run from the command line, e.g., `vlc`, `mplayer`, `gxine`, and `totem`.

It is easy to create the video file from a Python program since we can run any operating system command in (e.g.) `os.system`:

```
cmd = 'convert -delay 50 tmp_*.png movie.gif'
os.system(cmd)
```

It might happen that your downloaded and installed version of `ffmpeg` fails to generate videos in some of the mentioned formats. The reason is that `ffmpeg` depends on many other packages that may be missing on your system. Getting `ffmpeg` to work with the `libx264` codec for making MP4 files is often challenging. On Debian-based Linux systems, such as Ubuntu, the installation procedure at the time of this writing goes like

[5] `http://hplgit.github.io/scipro-primer/video/gaussian.html`

```
Terminal> sudo apt-get install lib-avtools libavcodec-extra-53 \
          libx264-dev
```

5.3.6 Curve plots in pure text

Sometimes it can be desirable to show a graph in pure ASCII text, e.g., as part of a trial run of a program included in the program itself, or a graph that can be illustrative in a doc string. For such purposes we have slightly extended a module by Imri Goldberg (`aplotter.py`) and included it as a module in SciTools. Running `pydoc` on `scitools.aplotter` describes the capabilities of this type of primitive plotting. Here we just give an example of what it can do:

```
>>> import numpy as np
>>> x = np.linspace(-2, 2, 81)
>>> y = np.exp(-0.5*x**2)*np.cos(np.pi*x)
>>> from scitools.aplotter import plot
>>> plot(x, y)
                                        |
                                       -+1
                                     // |\\
                                    /   |  \
                                   /    |   \
                                  /     |    \
                                 /      |     \
                                /       |      \
   -------\                    /        |       \
---+-------\\-----------------/---------+--------\-----------------/
   -2        \                /         |        \                /
              \\              /         |         \             //
                \            /          |         \            /
                 \\         /           |          \         //
                   \       /            |           \       /
                    \    //             |            \-    //
                     ----               -0.63          ---/
                                        |
>>> # plot circles at data points only:
>>> plot(x, y, dot=’o’, plot_slope=False)
                                        |
                                       o+1
                                     oo |oo
                                    o   |  o
                                   o    |   o
                                        |
                                  o     |    o
                                 o      |     o
                                o       |      o
                                        |
   oooooooo                    o        |       o
---+-------oo-----------------o---------+--------o-----------------o
   -2        o                          |                         o
              oo              o         |         o             oo
                o            o          |         o            o
                 oo         o           |          o         oo
                   o       o            |           o       o
                    o    oo             |            oo    oo
                     oooo               -0.63          oooo
```

```
                                          |
>>> p = plot(x, y, output=str)   # store plot in a string p
>>> print p
```

(The last 13 characters of the output lines are here removed to make the lines fit the maximum textwidth of this book.)

5.4 Plotting difficulties

The previous examples on plotting functions demonstrate how easy it is to make graphs. Nevertheless, the shown techniques might easily fail to plot some functions correctly unless we are careful. Next we address two types of difficult functions: piecewisely defined functions and rapidly varying functions.

5.4.1 Piecewisely defined functions

A piecewisely defined function has different function definitions in different intervals along the x axis. The resulting function, made up of pieces, may have discontinuities in the function value or in derivatives. We have to be very careful when plotting such functions, as the next two examples will show. The problem is that the plotting mechanism draws straight lines between coordinates on the function's curve, and these straight lines may not yield a satisfactory visualization of the function. The first example has a discontinuity in the function itself at one point, while the other example has a discontinuity in the derivative at three points.

Example: The Heaviside function. Let us plot the Heaviside function

$$H(x) = \begin{cases} 0, x < 0 \\ 1, x \geq 0 \end{cases}$$

The most natural way to proceed is first to define the function as

```
def H(x):
    return (0 if x < 0 else 1)
```

The standard plotting procedure where we define a coordinate array `x` and call `y = H(x)` will not work for array arguments `x`, of reasons to be explained in Section 5.5.2. However, we may use techniques from that chapter to create a function `Hv(x)` that works for array arguments. Even with such a function we face difficulties with plotting it.

Since the Heaviside function consists of two flat lines, one may think that we do not need many points along the x axis to describe the curve. Let us try with nine points:

```
x = np.linspace(-10, 10, 9)
from scitools.std import plot
plot(x, Hv(x), axis=[x[0], x[-1], -0.1, 1.1])
```

However, so few x points are not able to describe the jump from 0 to 1 at $x = 0$, as shown by the solid line in Figure 5.8a. Using more points, say 50 between -10 and 10,

```
x2 = np.linspace(-10, 10, 50)
plot(x, Hv(x), 'r', x2, Hv(x2), 'b',
     legend=('5 points', '50 points'),
     axis=[x[0], x[-1], -0.1, 1.1])
```

makes the curve look better. However, the step is still not strictly vertical. More points will improve the situation. Nevertheless, the best is to draw two flat lines directly: from $(-10, 0)$ to $(0, 0)$, then to $(0, 1)$ and then to $(10, 1)$:

```
plot([-10, 0, 0, 10], [0, 0, 1, 1],
     axis=[x[0], x[-1], -0.1, 1.1])
```

The result is shown in Figure 5.8b.

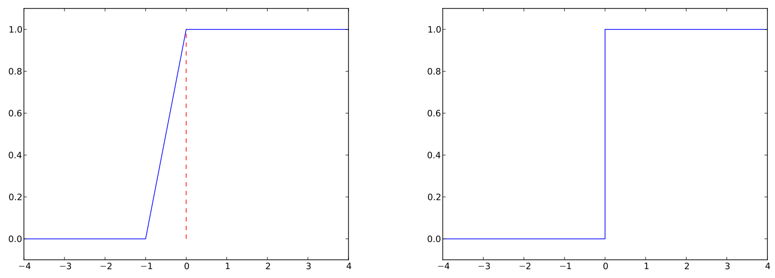


Fig. 5.8 Plot of the Heaviside function using 9 equally spaced x points (left) and with a double point at $x = 0$ (right).

Some will argue that the plot of $H(x)$ should not contain the vertical line from $(0, 0)$ to $(0, 1)$, but only two horizontal lines. To make such a plot, we must draw two distinct curves, one for each horizontal line:

```
plot([-10,0], [0,0], 'r-', [0,10], [1,1], 'r-',
     axis=[x[0], x[-1], -0.1, 1.1])
```

Observe that we must specify the same line style for both lines (curves), otherwise they would by default get different colors on the screen or different line types in a hardcopy of the plot. We remark, however, that discontinuous functions like $H(x)$ are often visualized with vertical lines at the jumps, as we do in Figure 5.8b.

Example: A hat function. Let us plot the hat function $N(x)$, shown as the solid line in Figure 5.9. This function is a piecewise linear function. The implementation of $N(x)$ must use `if` tests to locate where we are

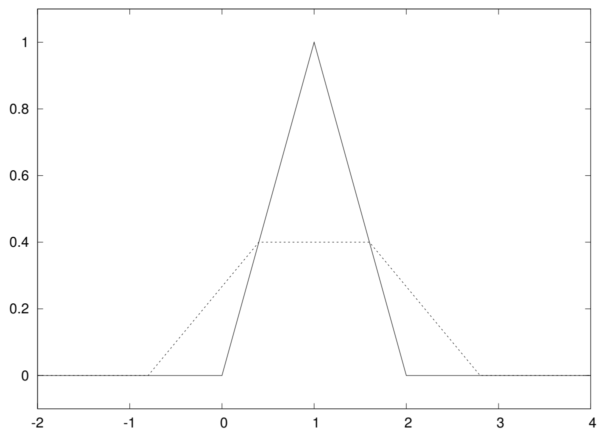


Fig. 5.9 Plot of a hat function. The solid line shows the exact function, while the dashed line arises from using inappropriate points along the x axis.

along the x axis and then evaluate the right linear piece of $N(x)$. A straightforward implementation with plain `if` tests does not work with array arguments `x`, but Section 5.5.3 explains how to make a vectorized version `Nv(x)` that works for array arguments as well. Anyway, both the scalar and the vectorized versions face challenges when it comes to plotting.

A first approach to plotting could be

```
x = np.linspace(-2, 4, 6)
plot(x, Nv(x), 'r', axis=[x[0], x[-1], -0.1, 1.1])
```

This results in the dashed line in Figure 5.9. What is the problem? The problem lies in the computation of the `x` vector, which does not contain the points $x = 1$ and $x = 2$ where the function makes significant changes. The result is that the hat is flattened. Making an `x` vector with all critical points in the function definitions, $x = 0, 1, 2$, provides the necessary remedy, either

```
x = np.linspace(-2, 4, 7)
```

or the simple

```
x = [-2, 0, 1, 2, 4]
```

Any of these `x` alternatives and a `plot(x, Nv(x))` will result in the solid line in Figure 5.9, which is the correct visualization of the $N(x)$ function.

As in the case of the Heaviside function, it is perhaps best to drop using vectorized evaluations and just draw straight lines between the critical points of the function (since the function is linear):

```
x = [-2, 0, 1, 2, 4]
y = [N(xi) for xi in x]
plot(x, y, 'r', axis=[x[0], x[-1], -0.1, 1.1])
```

5.4.2 Rapidly varying functions

Let us now visualize the function $f(x) = \sin(1/x)$, using 10 and 1000 points:

```
def f(x):
    return sin(1.0/x)

from scitools.std import linspace, plot
x1 = linspace(-1, 1, 10)
x2 = linspace(-1, 1, 1000)
plot(x1, f(x1), label='%d points' % len(x))
plot(x2, f(x2), label='%d points' % len(x))
```

The two plots are shown in Figure 5.10. Using only 10 points gives a completely wrong picture of this function, because the function oscillates faster and faster as we approach the origin. With 1000 points we get an impression of these oscillations, but the accuracy of the plot in the vicinity of the origin is still poor. A plot with 100000 points has better accuracy, in principle, but the extremely fast oscillations near the origin just drowns in black ink (you can try it out yourself).

Another problem with the $f(x) = \sin(1/x)$ function is that it is easy to define an `x` vector containing $x = 0$, such that we get division by zero. Mathematically, the $f(x)$ function has a singularity at $x = 0$: it is difficult to define $f(0)$, so one should exclude this point from the function definition and work with a domain $x \in [-1, -\epsilon] \cup [\epsilon, 1]$, with ϵ chosen small.

The lesson learned from these examples is clear. You must investigate the function to be visualized and make sure that you use an appropriate set of x coordinates along the curve. A relevant first step is to double the number of x coordinates and check if this changes the plot. If not, you probably have an adequate set of x coordinates.

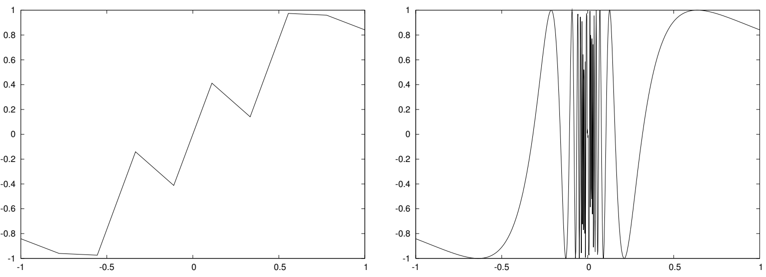


Fig. 5.10 Plot of the function $\sin(1/x)$ with 10 points (left) and 1000 points (right).

5.5 More advanced vectorization of functions

So far we have seen that vectorization of a Python function `f(x)` implementing some mathematical function $f(x)$ seems trivial: `f(x)` works right away with an array argument `x` and, in that case, returns an array where f is applied to each element in `x`. When the expression for $f(x)$ is given in terms of a string and the `StringFunction` tool is used to generate the corresponding Python function `f(x)`, one extra step must be performed to vectorize the Python function. This step is explained in Section 5.5.1.

The described vectorization works well as long as the expression $f(x)$ is a mathematical formula without any `if` test. As soon as we have `if` tests (conditional mathematical expressions) the vectorization becomes more challenging. Some useful techniques are explained through two examples in Sections 5.5.2 and 5.5.3. The described techniques are considered advanced material and only necessary when the time spent on evaluating a function at a very large set of points needs to be significantly decreased.

5.5.1 Vectorization of StringFunction objects

The `StringFunction` object from `scitools.std` can convert a formula, available as a string, to a callable Python function (see Section 4.3.3). However, the function cannot work with array arguments unless we explicitly tell the `StringFunction` object to do so. The recipe is very simple. Say `f` is some `StringFunction` object. To allow array arguments we are required to call `f.vectorize(globals())` once:

```
from numpy import *
x = linspace(0, 1, 30)
# f(x) will in general not work

f.vectorize(globals())
values = f(x)            # f works with array arguments
```

It is important that you import everything from `numpy` (or `scitools.std`) *before* calling `f.vectorize`, exactly as shown.

You may take the `f.vectorize` call as a magic recipe. Still, some readers want to know what problem `f.vectorize` solves. Inside the `StringFunction` module we need to have access to mathematical functions for expressions like `sin(x)*exp(x)` to be evaluated. These mathematical functions are by default taken from the `math` module and hence they do not work with array arguments. If the user, in the main program, has imported mathematical functions that work with array arguments, these functions are registered in a dictionary returned from `globals()`. By the `f.vectorize` call we supply the `StringFunction` module with the user's global namespace so that the evaluation of the string expression can make use of the mathematical functions for arrays from the user's program. Unless you use `np.sin(x)*np.cos(x)` etc. in the string formulas, make sure you do a `from numpy import *` so that the function names are defined without any prefix.

Even after calling `f.vectorize(globals())`, a `StringFunction` object may face problems with vectorization. One example is a piecewise constant function as specified by a string expression `'1 if x > 2 else 0'`. Section 5.5.2 explains why `if` tests fail for arrays and what the remedies are.

5.5.2 Vectorization of the Heaviside function

We consider the widely used Heaviside function defined by

$$H(x) = \begin{cases} 0, x < 0 \\ 1, x \geq 0 \end{cases}$$

The most compact way if implementing this function is

```
def H(x):
    return (0 if x < 0 else 1)
```

Trying to call `H(x)` with an array argument `x` fails:

```
>>> def H(x): return (0 if x < 0 else 1)
...
>>> import numpy as np
>>> x = np.linspace(-10, 10, 5)
>>> x
array([-10.,  -5.,   0.,   5.,  10.])
>>> H(x)
...
ValueError: The truth value of an array with more than
one element is ambiguous. Use a.any() or a.all()
```

The problem is related to the test `x < 0`, which results in an array of boolean values, while the `if` test needs a single boolean value (essentially taking `bool(x < 0)`):

```
>>> b = x < 0
>>> b
array([ True,  True, False, False, False], dtype=bool)
>>> bool(b)  # evaluate b in a boolean context
...
ValueError: The truth value of an array with more than
one element is ambiguous. Use a.any() or a.all()
>>> b.any()  # True if any element in b is True
True
>>> b.all()  # True if all elements in b are True
False
```

The `any` and `all` calls do not help us since we want to take actions element by element depending on whether `x[i] < 0` or not.

There are four ways to find a remedy to our problems with the `if x < 0` test: (i) we can write an explicit loop for computing the elements,

(ii) we can use a tool for automatically vectorize `H(x)`, (iii) we can mix boolean and floating-point calculations, or (iv) we can manually vectorize the `H(x)` function. All four methods will be illustrated next.

Loop. The following function works well for arrays if we insert a simple loop over the array elements (such that `H(x)` operates on scalars only):

```
def H_loop(x):
    r = np.zeros(len(x))
    for i in xrange(len(x)):
        r[i] = H(x[i])
    return r

# Example:
x = np.linspace(-5, 5, 6)
y = H_loop(x)
```

Automatic vectorization. Numerical Python contains a method for automatically vectorizing a Python function `H(x)` that works with scalars (pure numbers) as `x` argument:

```
import numpy as np
H_vec = np.vectorize(H)
```

The `H_vec(x)` function will now work with vector/array arguments `x`. Unfortunately, such automatically vectorized functions runs at a fairly slow speed compared to the implementations below (see the end of Section 5.5.3 for specific timings).

Mixing boolean and floating-point calculations. It appears that a very simple solution to vectorizing the `H(x)` function is to implement it as

```
def H(x):
    return x >= 0
```

The return value is now a `bool` object, not an `int` or `float` as we would mathematically expect to be the proper type of the result. However, the `bool` object works well in both scalar and vectorized operations as long as we involve the returned `H(x)` in some arithmetic expression. The `True` and `False` values are then interpreted as 1 and 0. Here is a demonstration:

```
>>> x = np.linspace(-1, 1, 5)
>>> H(x)
array([False, False,  True,  True,  True], dtype=bool)
>>> 1*H(x)
array([0, 0, 1, 1, 1])
>>> H(x) - 2
array([-2, -2, -1, -1, -1])
>>>
>>> x = 0.2   # test scalar argument
>>> H(x)
True
>>> 1*H(x)
1
```

```
>>> H(x) - 2
-1
```

If returning a boolean value is considered undesirable, we can turn the `bool` object into the proper type by

```
def H(x):
    r = x >= 0
    if isinstance(x, (int,float)):
        return int(r)
    elif isinstance(x, np.ndarray):
        return np.asarray(r, dtype=np.int)
```

Manual vectorization. By manual vectorization we normally mean translating the algorithm into a set of calls to functions in the `numpy` package such that no loops are visible in the Python code. The last version of the `H(x)` is a manual vectorization, but now we shall look at a more general technique when the result is not necessarily 0 or 1. In general, manual vectorization is non-trivial and requires knowledge of and experience with the underlying library for array computations. Fortunately, there is a simple `numpy` recipe for turning functions of the form

```
def f(x):
    if condition:
        r = <expression1>
    else:
        r = <expression2>
    return r
```

into vectorized form:

```
def f_vectorized(x):
    x1 = <expression1>
    x2 = <expression2>
    r = np.where(condition, x1, x2)
    return r
```

The `np.where` function returns an array of the same length as `condition`, whose element no. `i` equals `x1[i]` if `condition[i]` is `True`, and `x2[i]` otherwise. With Python loops we can express this principle as

```
def my_where(condition, x1, x2):
    r = np.zeros(len(condition))   # result
    for i in xrange(condition):
        r[i] = x1[i] if condition[i] else x2[i]
    return r
```

The `x1` and `x2` variables can be pure numbers too in the call to `np.where`.

In our case we can use the `np.where` function as follows:

```
def Hv(x):
    return np.where(x < 0, 0.0, 1.0)
```

Instead of using `np.where` we can apply *boolean indexing*. The idea is that an array `a` allows to be indexed by an array `b` of boolean values:

a[b]. The result a[b] is a new array with all the elements a[i] where b[i] is True:

```
>>> a
array([  0. ,   2.5,   5. ,   7.5,  10. ])
>>> b = a > 5
>>> b
array([False, False, False,  True,  True], dtype=bool)
>>> a[b]
array([  7.5,  10. ])
```

We can assign new values to the elements in a where b is True:

```
>>> a[b]
array([  7.5,  10. ])
>>> a[b] = np.array([-10, -20], dtype=np.float)
>>> a
array([  0. ,   2.5,   5. , -10. , -20. ])
>>> a[b] = -4
>>> a
array([ 0. ,  2.5,  5. , -4. , -4. ])
```

To implement the Heaviside function, we start with an array of zeros and then assign 1 to the elements where x >= 0:

```
def Hv(x):
    r = np.zeros(len(x), dtype=np.int)
    r[x >= 0] = 1
    return r
```

5.5.3 Vectorization of a hat function

We now turn the attention to the hat function $N(x)$ defined by

$$N(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \le x < 1 \\ 2 - x, & 1 \le x < 2 \\ 0, & x \ge 2 \end{cases}$$

The corresponding Python implementation N(x) is

```
def N(x):
    if x < 0:
        return 0.0
    elif 0 <= x < 1:
        return x
    elif 1 <= x < 2:
        return 2 - x
    elif x >= 2:
        return 0.0
```

Unfortunately, this N(x) function does not work with array arguments x, because the boolean expressions, like x < 0, are arrays and they cannot yield a single True or False value for the if tests, as explained in Section 5.5.2.

The simplest remedy is to use np.vectorize from Section 5.5.2:

```
N_vec = np.vectorize(N)
```

It is then important that `N(x)` returns `float` and not `int` values, otherwise the vectorized version will produce `int` values and hence be incorrect.

A manual rewrite, yielding a faster vectorized function, is more demanding than for the Heaviside function because we now have multiple branches in the `if` test. One sketch is to replace

```
if condition1:
   r = <expression1>
elif condition2:
   r = <expression2>
elif condition3:
   r = <expression3>
else:
   r = <expression4>
```

by

```
x1 = <expression1>
x2 = <expression2>
x3 = <expression3>
x4 = <expression4>
r = np.where(condition1, x1, x4)  # initialize with "else" expr.
r = np.where(condition2, x2, r)
r = np.where(condition3, x3, r)
```

Alternatively, we can use boolean indexing. Assuming that `<expressionX>` is some expression involving an array `x` and coded as a Python function `fX(x)` (`X` is 1, 2, 3, or 4), we can write

```
r = f4(x)
r[condition1] = f1(x[condition1])
r[condition2] = f2(x[condition2])
r[condition3] = f2(x[condition3])
```

Specifically, when the function for scalar arguments `x` reads

```
def N(x):
    if x < 0:
        return 0.0
    elif 0 <= x < 1:
        return x
    elif 1 <= x < 2:
        return 2 - x
    elif x >= 2:
        return 0.0
```

a vectorized attempt would be

```
def Nv(x):
    r = np.where(x < 0,      0.0,  0.0)
    r = np.where(0 <= x < 1, x,    r)
    r = np.where(1 <= x < 2, 2-x,  r)
    r = np.where(x >= 2,     0.0,  r)
    return r
```

The first and last line are not strictly necessary as we could just start with a zero vector (making the insertion of zeros for the first and last condition a redundant operation).

However, any condition like `0 <= x < 1`, which is equivalent to `0 <= x and x < 1`, does not work because the `and` operator does not work with array arguments. Fortunately, there is a simple solution to this problem: the function `logical_and` in `numpy`. A working `Nv` function must apply `logical_and` instead in each condition:

```
def Nv1(x):
    condition1 = x < 0
    condition2 = np.logical_and(0 <= x, x < 1)
    condition3 = np.logical_and(1 <= x, x < 2)
    condition4 = x >= 2

    r = np.where(condition1, 0.0, 0.0)
    r = np.where(condition2, x,   r)
    r = np.where(condition3, 2-x, r)
    r = np.where(condition4, 0.0, r)
    return r
```

With boolean indexing we get the alternative form

```
def Nv2(x):
    condition1 = x < 0
    condition2 = np.logical_and(0 <= x, x < 1)
    condition3 = np.logical_and(1 <= x, x < 2)
    condition4 = x >= 2

    r = np.zeros(len(x))
    r[condition1] = 0.0
    r[condition2] = x[condition2]
    r[condition3] = 2-x[condition3]
    r[condition4] = 0.0
    return r
```

Again, the first and last assignment to `r` can be omitted in this special case where we start with a zero vector.

The file `hat.py` implements four vectorized versions of the `N(x)` function: `N_loop`, which is a plain loop calling up `N(x)` for each `x[i]` element in the array `x`; `N_vec`, which is the result of automatic vectorization via `np.vectorize`; the `Nv1` function shown above, which uses the `np.where` constructions; and the `Nv2` function, which uses boolean indexing. With a length of `x` of 1,000,000, the results on my computer (MacBook Air 11", 2 1.6GHz Intel CPU, running Ubuntu 12.04 in a VMWare virtual machine) became 4.8 s for `N_loop`, 1 s `N_vec`, 0.3 s for `Nv1`, and 0.08 s for `Nv2`. Boolean indexing is clearly the fastest method.

5.6 More on numerical Python arrays

This section lists some more advanced but useful operations with Numerical Python arrays.

5.6.1 Copying arrays

Let `x` be an array. The statement `a = x` makes `a` refer to the same array as `x`. Changing `a` will then also affect `x`:

```
>>> import numpy as np
>>> x = np.array([1, 2, 3.5])
>>> a = x
>>> a[-1] = 3  # this changes x[-1] too!
>>> x
array([ 1.,  2.,  3.])
```

Changing `a` without changing `x` requires `a` to be a copy of `x`:

```
>>> a = x.copy()
>>> a[-1] = 9
>>> a
array([ 1.,  2.,  9.])
>>> x
array([ 1.,  2.,  3.])
```

5.6.2 In-place arithmetics

Let `a` and `b` be two arrays of the same shape. The expression `a += b` means `a = a + b`, but this is not the complete story. In the statement `a = a + b`, the sum `a + b` is first computed, yielding a new array, and then the name `a` is bound to this new array. The old array `a` is lost unless there are other names assigned to this array. In the statement `a += b`, elements of `b` are added directly into the elements of `a` (in memory). There is no hidden intermediate array as in `a = a + b`. This implies that `a += b` is more efficient than `a = a + b` since Python avoids making an extra array. We say that the operators `+=`, `*=`, and so on, perform *in-place* arithmetics in arrays.

Consider the compound array expression

```
a = (3*x**4 + 2*x + 4)/(x + 1)
```

The computation actually goes as follows with seven hidden arrays for storing intermediate results:

- `r1 = x**4`
- `r2 = 3*r1`
- `r3 = 2*x`
- `r4 = r2 + r3`
- `r5 = r4 + 4`
- `r6 = x + 1`
- `r7 = r5/r6`
- `a = r7`

With in-place arithmetics we can get away with creating three new arrays, at a cost of a significantly less readable code:

```
a = x.copy()
a **= 4
a *= 3
a += 2*x
a += 4
a /= x + 1
```

The three extra arrays in this series of statement arise from copying `x`, and computing the right-hand sides `2*x` and `x+1`.

Quite often in computational science and engineering, a huge number of arithmetics is performed on very large arrays, and then saving memory and array allocation time by doing in-place arithmetics is important.

The mix of assignment and in-place arithmetics makes it easy to make unintended changes of more than one array. For example, this code changes `x`:

```
a = x
a += y
```

since `a` refers to the same array as `x` and the change of `a` is done in-place.

5.6.3 Allocating arrays

We have already seen that the `np.zeros` function is handy for making a new array `a` of a given size. Very often the size and the type of array elements have to match another existing array `x`. We can then either copy the original array, e.g.,

```
a = x.copy()
```

and fill elements in `a` with the right new values, or we can say

```
a = np.zeros(x.shape, x.dtype)
```

The attribute `x.dtype` holds the array element type (`dtype` for data type), and as mentioned before, `x.shape` is a tuple with the array dimensions.

Sometimes we may want to ensure that an object is an array, and if not, turn it into an array. The `np.asarray` function is useful in such cases:

```
a = np.asarray(a)
```

Nothing is copied if `a` already is an array, but if `a` is a list or tuple, a new array with a copy of the data is created.

5.6.4 Generalized indexing

Section 5.2.2 shows how slices can be used to extract and manipulate subarrays. The slice `f:t:i` corresponds to the index set `f, f+i, f+2*i,`

. . . up to, but not including, `t`. Such an index set can be given explicitly too: `a[range(f,t,i)]`. That is, the integer list from `range` can be used as a set of indices. In fact, any integer list or integer array can be used as index:

```
>>> a = np.linspace(1, 8, 8)
>>> a
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.])
>>> a[[1,6,7]] = 10
>>> a
array([  1.,  10.,   3.,   4.,   5.,   6.,  10.,  10.])
>>> a[range(2,8,3)] = -2   # same as a[2:8:3] = -2
>>> a
array([  1.,  10.,  -2.,   4.,   5.,  -2.,  10.,  10.])
```

We can also use boolean arrays to generate an index set. The indices in the set will correspond to the indices for which the boolean array has `True` values. This functionality allows expressions like `a[x<m]`. Here are two examples, continuing the previous interactive session:

```
>>> a[a < 0]            # pick out the negative elements of a
array([-2., -2.])
>>> a[a < 0] = a.max()
>>> a
array([  1.,  10.,  10.,   4.,   5.,  10.,  10.,  10.])
>>> # Replace elements where a is 10 by the first
>>> # elements from another array/list:
>>> a[a == 10] = [10, 20, 30, 40, 50, 60, 70]
>>> a
array([  1.,  10.,  20.,   4.,   5.,  30.,  40.,  50.])
```

Generalized indexing using integer arrays or lists is important for vectorized initialization of array elements. The syntax for generalized indexing of higher-dimensional arrays is slightly different, see Section 5.7.2.

5.6.5 Testing for the array type

Inside an interactive Python shell you can easily check an object's type using the `type` function (see Section 1.5.2). In case of a Numerical Python array, the type name is `ndarray`:

```
>>> a = np.linspace(-1, 1, 3)
>>> a
array([-1.,  0.,  1.])
>>> type(a)
<type 'numpy.ndarray'>
```

Sometimes you need to test if a variable is an `ndarray` or a `float` or `int`. The `isinstance` function can be used this purpose:

```
>>> isinstance(a, np.ndarray)
True
>>> isinstance(a, (float,int))  # float or int?
False
```

A typical use of `isinstance` and `type` to check on object's type is shown next.

Example: Vectorizing a constant function. Suppose we have a constant function,

```
def f(x):
    return 2
```

This function accepts an array argument `x`, but will return a `float` while a vectorized version of the function should return an array of the same shape as `x` where each element has the value 2. The vectorized version can be realized as

```
def fv(x):
    return np.zeros(x.shape, x.dtype) + 2
```

The optimal vectorized function would be one that works for both a scalar and an array argument. We must then test on the argument type:

```
def f(x):
    if isinstance(x, (float, int)):
        return 2
    elif isinstance(x, np.ndarray):
        return np.zeros(x.shape, x.dtype) + 2
    else:
        raise TypeError\
        ('x must be int, float or ndarray, not %s' % type(x))
```

5.6.6 Compact syntax for array generation

There is a special compact syntax `r_[f:t:s]` for the `linspace` function:

```
>>> a = r_[-5:5:11j]  # same as linspace(-5, 5, 11)
>>> print a
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.]
```

Here, `11j` means 11 coordinates (between -5 and 5, including the upper limit 5). That is, the number of elements in the array is given with the imaginary number syntax.

5.6.7 Shape manipulation

The `shape` attribute in array objects holds the shape, i.e., the size of each dimension. A function `size` returns the total number of elements in an array. Here are a few equivalent ways of changing the shape of an array:

```
>>> a = np.linspace(-1, 1, 6)
>>> print a
[-1.  -0.6 -0.2  0.2  0.6  1. ]
>>> a.shape
(6,)
>>> a.size
6
>>> a.shape = (2, 3)
>>> a = a.reshape(2, 3)    # alternative
>>> a.shape
(2, 3)
>>> print a
[[-1.  -0.6 -0.2]
 [ 0.2  0.6  1. ]]
>>> a.size                 # total no of elements
6
>>> len(a)                 # no of rows
2
>>> a.shape = (a.size,)    # reset shape
```

Note that `len(a)` always returns the length of the first dimension of an array.

5.7 Higher-dimensional arrays

5.7.1 Matrices and arrays

Vectors appeared when mathematicians needed to calculate with a list of numbers. When they needed a table (or a list of lists in Python terminology), they invented the concept of *matrix* (singular) and *matrices* (plural). A table of numbers has the numbers ordered into rows and columns. One example is

$$\begin{bmatrix} 0 & 12 & -1 & 5 \\ -1 & -1 & -1 & 0 \\ 11 & 5 & 5 & -2 \end{bmatrix}$$

This table with three rows and four columns is called a 3×4 matrix (mathematicians don't like this sentence, but it suffices for our purposes). If the symbol A is associated with this matrix, $A_{i,j}$ denotes the number in row number i and column number j. Counting rows and columns from 0, we have, for instance, $A_{0,0} = 0$ and $A_{2,3} = -2$. We can write a general $m \times n$ matrix A as

$$\begin{bmatrix} A_{0,0} & \cdots & A_{0,n-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,n-1} \end{bmatrix}$$

Matrices can be added and subtracted. They can also be multiplied by a scalar (a number), and there is a concept of length or size. The formulas are quite similar to those presented for vectors, but the exact form is not important here.

We can generalize the concept of table and matrix to *array*, which holds quantities with in general d indices. Equivalently we say that the array has rank d. For $d = 3$, an array A has elements with three indices: $A_{p,q,r}$. If p goes from 0 to $n_p - 1$, q from 0 to $n_q - 1$, and r from 0 to $n_r - 1$, the A array has $n_p \times n_q \times n_r$ elements in total. We may speak about the *shape* of the array, which is a d-vector holding the number of elements in each "array direction", i.e., the number of elements for each index. For the mentioned A array, the shape is (n_p, n_q, n_r).

The special case of $d = 1$ is a vector, and $d = 2$ corresponds to a matrix. When we program we may skip thinking about vectors and matrices (if you are not so familiar with these concepts from a mathematical point of view) and instead just work with arrays. The number of indices corresponds to what is convenient in the programming problem we try to solve.

5.7.2 Two-dimensional numerical Python arrays

Consider a nested list `table` of two-pairs `[C, F]` (see Section 2.4) constructed by

```
>>> Cdegrees = [-30 + i*10 for i in range(3)]
>>> Fdegrees = [9./5*C + 32 for C in Cdegrees]
>>> table = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]
>>> print table
[[-30, -22.0], [-20, -4.0], [-10, 14.0]]
```

This nested list can be turned into an array,

```
>>> table2 = np.array(table)
>>> print table2
[[-30. -22.]
 [-20.  -4.]
 [-10.  14.]]
>>> type(table2)
<type 'numpy.ndarray'>
```

We say that `table2` is a *two-dimensional* array, or an array of rank 2.

The `table` list and the `table2` array are stored very differently in memory. The `table` variable refers to a list object containing three elements. Each of these elements is a reference to a separate list object with two elements, where each element refers to a separate `float` object. The `table2` variable is a reference to a single array object that again refers to a consecutive sequence of bytes in memory where the six floating-point numbers are stored. The data associated with `table2` are found in one chunk in the computer's memory, while the data associated with `table` are scattered around in memory. On today's machines, it is much more expensive to find data in memory than to compute with the data. Arrays make the data fetching more efficient, and this is major reason for using arrays. However, this efficiency gain is only present for very large arrays, not for a 3×2 array.

Indexing a nested list is done in two steps, first the outer list is indexed, giving access to an element that is another list, and then this latter list is indexed:

```
>>> table[1][0]     # table[1] is [-20,4], whose index 0 holds -20
-20
```

This syntax works for two-dimensional arrays too:

```
>>> table2[1][0]
-20.0
```

but there is another syntax that is more common for arrays:

```
>>> table2[1,0]
-20.0
```

A two-dimensional array reflects a table and has a certain number of rows and columns. We refer to rows as the *first dimension* of the array and columns as the *second dimension*. These two dimensions are available as `table2.shape`:

```
>>> table2.shape
(3, 2)
```

Here, 3 is the number of rows and 2 is the number of columns.

A loop over all the elements in a two-dimensional array is usually expressed as two *nested* `for` loops, one for each index:

```
>>> for i in range(table2.shape[0]):
...     for j in range(table2.shape[1]):
...         print 'table2[%d,%d] = %g' % (i, j, table2[i,j])
...
table2[0,0] = -30
table2[0,1] = -22
table2[1,0] = -20
table2[1,1] = -4
table2[2,0] = -10
table2[2,1] = 14
```

An alternative (but less efficient) way of visiting each element in an array with any number of dimensions makes use of a single `for` loop:

```
>>> for index_tuple, value in np.ndenumerate(table2):
...     print 'index %s has value %g' % \
...           (index_tuple, table2[index_tuple])
...
index (0,0) has value -30
index (0,1) has value -22
index (1,0) has value -20
index (1,1) has value -4
index (2,0) has value -10
index (2,1) has value 14
```

In the same way as we can extract sublists of lists, we can extract subarrays of arrays using slices.

```
table2[0:table2.shape[0], 1]  # 2nd column (index 1)
array([-22.,  -4.,  14.])

>>> table2[0:, 1]             # same
array([-22.,  -4.,  14.])

>>> table2[:, 1]              # same
array([-22.,  -4.,  14.])
```

To illustrate array slicing further, we create a bigger array:

```
>>> t = np.linspace(1, 30, 30).reshape(5, 6)
>>> t
array([[  1.,   2.,   3.,   4.,   5.,   6.],
       [  7.,   8.,   9.,  10.,  11.,  12.],
       [ 13.,  14.,  15.,  16.,  17.,  18.],
       [ 19.,  20.,  21.,  22.,  23.,  24.],
       [ 25.,  26.,  27.,  28.,  29.,  30.]])

>>> t[1:-1:2, 2:]
array([[  9.,  10.,  11.,  12.],
       [ 21.,  22.,  23.,  24.]])
```

To understand the slice, look at the original `t` array and pick out the two rows corresponding to the first slice `1:-1:2`,

```
[  7.,   8.,   9.,  10.,  11.,  12.]
[ 19.,  20.,  21.,  22.,  23.,  24.]
```

Among the rows, pick the columns corresponding to the second slice `2:`,

```
[  9.,  10.,  11.,  12.]
[ 21.,  22.,  23.,  24.]
```

Another example is

```
>>> t[:-2, :-1:2]
array([[  1.,   3.,   5.],
       [  7.,   9.,  11.],
       [ 13.,  15.,  17.]])
```

Generalized indexing as described for one-dimensional arrays in Section 5.6.4 requires a more comprehensive syntax for higher-dimensional arrays. Say we want to extract a subarray of `t` that consists of the rows with indices 0 and 3 and the columns with indices 1 and 2:

```
>>> t[np.ix_([0,3], [1,2])]
array([[  2.,   3.],
       [ 20.,  21.]])
>>> t[np.ix_([0,3], [1,2])] = 0
>>> t
array([[  1.,   0.,   0.,   4.,   5.,   6.],
       [  7.,   8.,   9.,  10.,  11.,  12.],
       [ 13.,  14.,  15.,  16.,  17.,  18.],
       [ 19.,   0.,   0.,  22.,  23.,  24.],
       [ 25.,  26.,  27.,  28.,  29.,  30.]])
```

Recall that slices only gives a view to the array, not a copy of the values:

```
>>> a = t[1:-1:2, 1:-1]
>>> a
array([[  8.,   9.,  10.,  11.],
       [  0.,   0.,  22.,  23.]])
>>> a[:,:] = -99
>>> a
array([[-99., -99., -99., -99.],
       [-99., -99., -99., -99.]])
>>> t  # is t changed to? yes!
array([[  1.,   0.,   0.,   4.,   5.,   6.],
       [  7., -99., -99., -99., -99.,  12.],
       [ 13.,  14.,  15.,  16.,  17.,  18.],
       [ 19., -99., -99., -99., -99.,  24.],
       [ 25.,  26.,  27.,  28.,  29.,  30.]])
```

5.7.3 Array computing

The operations on vectors in Section 5.1.3 can quite straightforwardly be extended to arrays of any dimension. Consider the definition of applying a function $f(v)$ to a vector v: we apply the function to each element v_i in v. For a two-dimensional array A with elements $A_{i,j}$, $i = 0,\ldots,m$, $j = 0,\ldots,n$, the same definition yields

$$f(A) = (f(A_{0,0}),\ldots,f(A_{m-1,0}),f(A_{1,0}),\ldots,f(A_{m-1,n-1}))\,.$$

For an array B with any rank, $f(B)$ means applying f to each array entry.

The asterisk operation from Section 5.1.3 is also naturally extended to arrays: $A * B$ means multiplying an element in A by the corresponding element in B, i.e., element (i,j) in $A * B$ is $A_{i,j}B_{i,j}$. This definition naturally extends to arrays of any rank, provided the two arrays have the same shape.

Adding a scalar to an array implies adding the scalar to each element in the array. Compound expressions involving arrays, e.g., $\exp(-A^2) * A + 1$, work as for vectors. One can in fact just imagine that all the array elements are stored after each other in a long vector (this is actually the way the array elements are stored in the computer's memory), and the array operations can then easily be defined in terms of the vector operations from Section 5.1.3.

Remark. Readers with knowledge of matrix computations may get confused by the meaning of A^2 in matrix computing and A^2 in array computing. The former is a matrix-matrix product, while the latter means squaring all elements of A. Which rule to apply, depends on the context, i.e., whether we are doing linear algebra or vectorized arithmetics. In mathematical typesetting, A^2 can be written as AA, while the array computing expression A^2 can be alternatively written as $A * A$. In a program, `A*A` and `A**2` are identical computations, meaning squaring all elements (array arithmetics). With NumPy arrays the matrix-matrix product is obtained by `dot(A, A)`. The matrix-vector product Ax, where x is a vector, is computed by `dot(A, x)`. However, with matrix objects (see Section 5.7.5) `A*A` implies the mathematical matrix multiplication AA.

We shall leave this subject of notational confusion between array computing and linear algebra here since this book will not further understanding and the confusion is seldom serious in program code if one has a good overview of the mathematics that is to be carried out.

5.7.4 Two-dimensional arrays and functions of two variables

Given a function of two variables, say

```
def f(x, y):
    return sin(sqrt(x**2 + y**2))
```

we can plot this function by writing

```
from scitools.std import sin, sqrt, linspace, ndgrid, mesh
x = y = linspace(-5, 5, 21)  # coordinates in x and y direction
xv, yv = ndgrid(x, y)
z = f(xv, yv)
mesh(xv, yv, z)
```

There are two new things here: (i) the call to `ndgrid`, which is necessary to transform one-dimensional coordinate arrays in the x and y direction into arrays valid for evaluating `f` over a two-dimensional grid; and (ii) the plot function whose name now is `mesh`, which is one out of many plot functions for two-dimensional functions. Another plot type you can try out is

```
surf(xv, yv, z)
```

More material on visualizing $f(x, y)$ functions is found in the section *Visualizing Scalar Fields* in the Easyviz tutorial. This tutorial can be reached through the command `pydoc scitools.easyviz` in a terminal window or from `code.google.com/p/scitools`[6].

5.7.5 Matrix objects

This section only makes sense if you are familiar with basic linear algebra and the matrix concept. The arrays created so far have been of type `ndarray`. NumPy also has a matrix type called `matrix` or `mat` for one- and two-dimensional arrays. One-dimensional arrays are then extended with one extra dimension such that they become matrices, i.e., either a row vector or a column vector:

[6] `http://code.google.com/p/scitools`

```
>>> import numpy as np
>>> x1 = np.array([1, 2, 3], float)
>>> x2 = np.matrix(x1)              # or mat(x1)
>>> x2                              # row vector
matrix([[ 1.,  2.,  3.]])
>>> x3 = mat(x).transpose()         # column vector
>>> x3
matrix([[ 1.],
        [ 2.],
        [ 3.]])

>>> type(x3)
<class 'numpy.matrixlib.defmatrix.matrix'>
>>> isinstance(x3, np.matrix)
True
```

A special feature of `matrix` objects is that the multiplication operator represents the matrix-matrix, vector-matrix, or matrix-vector product as we know from linear algebra:

```
>>> A = eye(3)                   # identity matrix
>>> A
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> A = mat(A)
>>> A
matrix([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
>>> y2 = x2*A                    # vector-matrix product
>>> y2
matrix([[ 1.,  2.,  3.]])
>>> y3 = A*x3                    # matrix-vector product
>>> y3
matrix([[ 1.],
        [ 2.],
        [ 3.]])
```

One should note here that the multiplication operator between standard `ndarray` objects is quite different, as the next interactive session demonstrates.

```
>>> A*x1                         # no matrix-array product!
Traceback (most recent call last):
...
ValueError: matrices are not aligned

>>> # try array*array product:
>>> A = (zeros(9) + 1).reshape(3,3)
>>> A
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> A*x1                     # [A[0,:]*x1, A[1,:]*x1, A[2,:]*x1]
array([[ 1.,  2.,  3.],
       [ 1.,  2.,  3.],
       [ 1.,  2.,  3.]])
>>> B = A + 1
>>> A*B                      # element-wise product
array([[ 2.,  2.,  2.],
       [ 2.,  2.,  2.],
       [ 2.,  2.,  2.]])
>>> A = mat(A);  B = mat(B)
>>> A*B                          # matrix-matrix product
matrix([[ 6.,  6.,  6.],
        [ 6.,  6.,  6.],
        [ 6.,  6.,  6.]])
```

Readers who are familiar with MATLAB, or intend to use Python and MATLAB together, should seriously think about programming with `matrix` objects instead of `ndarray` objects, because the `matrix` type behaves quite similar to matrices and vectors in MATLAB. Nevertheless, `matrix` cannot be used for arrays of larger dimension than two.

5.8 Summary

5.8.1 Chapter topics

This chapter has introduced computing with arrays and plotting curve data stored in arrays. The Numerical Python package contains lots of functions for array computing, including the ones listed in the table below. Plotting has been done with tools that closely resemble the syntax of MATLAB.

Construction	Meaning
`array(ld)`	copy list data `ld` to a `numpy` array
`asarray(d)`	make array of data `d` (no data copy if already array)
`zeros(n)`	make a `float` vector/array of length `n`, with zeros
`zeros(n, int)`	make an `int` vector/array of length `n` with zeros
`zeros((m,n))`	make a two-dimensional `float` array with shape (`m`,‘n‘)
`zeros_like(x)`	make array of same shape and element type as `x`
`linspace(a,b,m)`	uniform sequence of `m` numbers in $[a, b]$
`a.shape`	tuple containing `a`’s shape
`a.size`	total no of elements in `a`
`len(a)`	length of a one-dim. array `a` (same as `a.shape[0]`)
`a.dtype`	the type of elements in `a`
`a.reshape(3,2)`	return `a` reshaped as 3×2 array
`a[i]`	vector indexing
`a[i,j]`	two-dim. array indexing
`a[1:k]`	slice: reference data with indices `1`,. . . ,‘k-1‘
`a[1:8:3]`	slice: reference data with indices `1`, `4`,. . . ,‘7‘
`b = a.copy()`	copy an array
`sin(a), exp(a), ...`	`numpy` functions applicable to arrays
`c = concatenate((a, b))`	`c` contains `a` with `b` appended
`c = where(cond, a1, a2)`	`c[i] = a1[i]` if `cond[i]`, else `c[i] = a2[i]`
`isinstance(a, ndarray)`	is `True` if `a` is an array

Array computing. When we apply a Python function `f(x)` to a Numerical Python array `x`, the result is the same as if we apply `f` to each element in `x` separately. However, when `f` contains `if` statements, these are in general invalid if an array `x` enters the boolean expression. We then have to rewrite the function, often by applying the `where` function from Numerical Python.

Plotting curves. Sections 5.3.1 and 5.3.2 provide a quick overview of how to plot curves with the aid of Matplotlib. The same examples coded with the Easyviz plotting interface appear in Section 5.3.3.

Making movies. Each frame in a movie must be a hardcopy of a plot in PNG format. These plot files should have names containing a counter padded with leading zeros. One example may be `tmp_0000.png`, `tmp_0001.png`, `tmp_0002.png`. Having the plot files with names on this form, we can make an animated GIF movie in the file `movie.gif`, with two frames per second, by

```
os.system('convert -delay 50 tmp_*.png movie.gif')
```

Alternatively, we may combine the plot files to a Flash video:

```
os.system('ffmpeg -r 5 -i tmp_%04d.png -vcodec flv movie.flv')
```

Other formats can be made using other codecs, see Section 5.3.5.

Terminology. The important topics in this chapter are

- array computing
- vectorization
- plotting
- animations

5.8.2 Example: Animating a function

Problem. In this chapter's summarizing example we shall visualize how the temperature varies downward in the earth as the surface temperature oscillates between high day and low night values. One question may be: What is the temperature change 10 m down in the ground if the surface temperature varies between 2 C in the night and 15 C in the day?

Let the z axis point downwards, towards the center of the earth, and let $z = 0$ correspond to the earth's surface. The temperature at some depth z in the ground at time t is denoted by $T(z, t)$. If the surface temperature has a periodic variation around some mean value T_0, according to

$$T(0, t) = T_0 + A\cos(\omega t),$$

one can find, from a mathematical model for heat conduction, that the temperature at an arbitrary depth is

$$T(z, t) = T_0 + Ae^{-az}\cos(\omega t - az), \quad a = \sqrt{\frac{\omega}{2k}}\,. \tag{5.13}$$

The parameter k reflects the ground's ability to conduct heat (k is called the *thermal diffusivity* or the *heat conduction coefficient*).

The task is to make an animation of how the temperature profile in the ground, i.e., T as a function of z, varies in time. Let ω correspond to a time period of 24 hours. The mean temperature T_0 is taken as 10 C, and the maximum variation A is assumed to be 10 C. The heat conduction coefficient k may be set as 1 mm^2/s (which is 10^{-6} m^2/s in proper SI units).

Solution. To animate $T(z,t)$ in time, we need to make a loop over points in time, and in each pass in the loop we must save a plot of T, as a function of z, to file. The plot files can then be combined to a movie. The algorithm becomes

- for $t_i = i\Delta t$, $i = 0, 1, 2 \ldots, n$:
  - plot the curve $y(z) = T(z, t_i)$
  - store the plot in a file
- combine all the plot files into a movie

It can be wise to make a general `animate` function where we just feed in some $f(x,t)$ function and make all the plot files. If `animate` has arguments for setting the labels on the axis and the extent of the y axis, we can easily use `animate` also for a function $T(z,t)$ (we just use z as the name for the x axis and T as the name for the y axis in the plot). Recall that it is important to fix the extent of the y axis in a plot when we make animations, otherwise most plotting programs will automatically fit the extent of the axis to the current data, and the tick marks on the y axis will jump up and down during the movie. The result is a wrong visual impression of the function.

The names of the plot files must have a common stem appended with some frame number, and the frame number should have a fixed number of digits, such as 0001, 0002, etc. (if not, the sequence of the plot files will not be correct when we specify the collection of files with an asterisk for the frame numbers, e.g., as in `tmp*.png`). We therefore include an argument to `animate` for setting the name stem of the plot files. By default, the stem is `tmp_`, resulting in the filenames `tmp_0000.png`, `tmp_0001.png`, `tmp_0002.png`, and so forth. Other convenient arguments for the `animate` function are the initial time in the plot, the time lag Δt between the plot frames, and the coordinates along the x axis. The `animate` function then takes the form

```
def animate(tmax, dt, x, function, ymin, ymax, t0=0,
            xlabel='x', ylabel='y', filename='tmp_'):
    t = t0
    counter = 0
    while t <= tmax:
        y = function(x, t)
        plot(x, y, '-',
             axis=[x[0], x[-1], ymin, ymax],
             title='time=%2d h' % (t/3600.0),
             xlabel=xlabel, ylabel=ylabel,
             savefig=filename + '%04d.png' % counter)
        savefig('tmp_%04d.pdf' % counter)
        t += dt
        counter += 1
```

The $T(z,t)$ function is easy to implement, but we need to decide whether the parameters A, ω, T_0, and k shall be arguments to the Python implementation of $T(z,t)$ or if they shall be global variables. Since the `animate` function expects that the function to be plotted has only two arguments, we must implement $T(z,t)$ as `T(z,t)` in Python and let the other parameters be global variables (Sections 7.1.1 and 7.1.2 explain this problem in more detail and present a better implementation). The `T(z,t)` implementation then reads

```
def T(z, t):
    # T0, A, k, and omega are global variables
    a = sqrt(omega/(2*k))
    return T0 + A*exp(-a*z)*cos(omega*t - a*z)
```

Suppose we plot $T(z,t)$ at n points for $z \in [0, D]$. We make such plots for $t \in [0, t_{\max}]$ with a time lag Δt between the them. The frames in the movie are now made by

```
# set T0, A, k, omega, D, n, tmax, dt
z = linspace(0, D, n)
animate(tmax, dt, z, T, T0-A, T0+A, 0, 'z', 'T')
```

We have here set the extent of the y axis in the plot as $[T_0 - A, T_0 + A]$, which is in accordance with the $T(z,t)$ function.

The call to `animate` above creates a set of files with names of the form `tmp_*.png`. Out of these files we can create an animated GIF movie or a video in, e.g., Flash format by running operating systems commands with `convert` and `avconv` (or `ffmpeg`):

```
os.system('convert -delay 50 tmp_*.png movie.gif')
os.system('avconv -r 5 -i tmp_%04d.png -vcodec flv movie.flv')
```

See Section 5.3.5 for how to create videos in other formats.

It now remains to assign proper values to all the global variables in the program: `n`, `D`, `T0`, `A`, `omega`, `dt`, `tmax`, and `k`. The oscillation period is 24 hours, and ω is related to the period P of the cosine function by $\omega = 2\pi/P$ (realize that $\cos(t2\pi/P)$ has period P). We then express $P = 24$ h as $24\cdot 60\cdot 60$ s and compute ω as $2\pi/P \approx 7\cdot 10^{-5}$ s^{-1}. The total simulation time can be 3 periods, i.e., $t_{\max} = 3P$. The $T(z,t)$ function decreases exponentially with the depth z so there is no point in having the maximum depth D larger than the depth where T is visually zero, say 0.001. We have that $e^{-aD} = 0.001$ when $D = -a^{-1}\ln 0.001$, so we can use this estimate in the program. The proper initialization of all parameters can then be expressed as follows:

```
k = 1E-6          # thermal diffusivity (in m*m/s)
P = 24*60*60.     # oscillation period of 24 h (in seconds)
omega = 2*pi/P
dt = P/24         # time lag: 1 h
```

```
tmax = 3*P       # 3 day/night simulation
T0 = 10          # mean surface temperature in Celsius
A = 10           # amplitude of the temperature variations in Celsius
a = sqrt(omega/(2*k))
D = -(1/a)*log(0.001) # max depth
n = 501          # no of points in the z direction
```

Note that it is very important to use consistent units. Here we express all units in terms of meter, second, and Kelvin or Celsius.

We encourage you to run the program `heatwave.py` to see the movie. The hardcopy of the movie is in the file `movie.gif`. Figure 5.11 displays two snapshots in time of the $T(z,t)$ function.

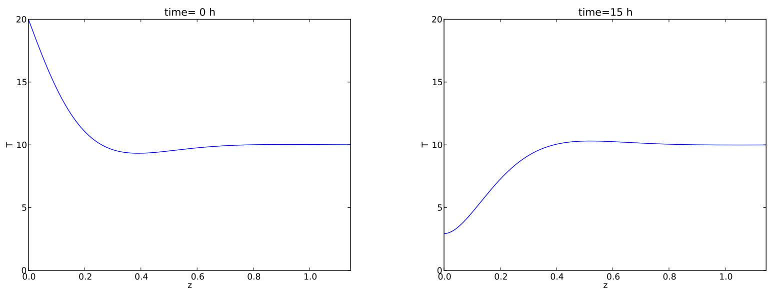


Fig. 5.11 Plot of the temperature $T(z,t)$ in the ground for two different t values.

Scaling. In this example, as in many other scientific problems, it was easier to write the code than to assign proper physical values to the input parameters in the program. To learn about the physical process, here how heat propagates from the surface and down in the ground, it is often advantageous to scale the variables in the problem so that we work with dimensionless variables. Through the scaling procedure we normally end up with much fewer physical parameters that must be assigned values. Let us show how we can take advantage of scaling the present problem.

Consider a variable x in a problem with some dimension. The idea of scaling is to introduce a new variable $\bar{x} = x/x_c$, where x_c is a *characteristic size* of x. Since x and x_c have the same dimension, the dimension cancels in $\bar{x}$ such that $\bar{x}$ is dimensionless. Choosing x_c to be the expected maximum value of x, ensures that $\bar{x} \leq 1$, which is usually considered a good idea. That is, we try to have all dimensionless variables varying between zero and one. For example, we can introduce a dimensionless z coordinate: $\bar{z} = z/D$, and now $\bar{z} \in [0,1]$. Doing a proper scaling of a problem is challenging so for now it is sufficient to just follow the steps below - and not worry why we choose a certain scaling.

In the present problem we introduce these dimensionless variables:

$$\begin{aligned}\bar{z} &= z/D\\ \bar{T} &= \frac{T-T_0}{A}\\ \bar{t} &= \omega t\end{aligned}$$

We now insert $z = \bar{z}D$ and $t = \bar{t}/\omega$ in the expression for $T(z,t)$ and get

$$T = T_0 + Ae^{-b\bar{z}}\cos(\bar{t} - b\bar{z}), \quad b = aD$$

or

$$\bar{T}(\bar{z}, \bar{t}) = \frac{T-T_0}{A} = e^{-b\bar{z}}\cos(\bar{t} - b\bar{z})\,.$$

We see that $\bar{T}$ depends on only *one* dimensionless parameter b in addition to the independent dimensionless variables $\bar{z}$ and $\bar{t}$. It is common practice at this stage of the scaling to just drop the bars and write

$$T(z,t) = e^{-bz}\cos(t - bz)\,. \tag{5.14}$$

This function is much simpler to plot than the one with lots of physical parameters, because now we know that T varies between -1 and 1, t varies between 0 and 2π for one period, and z varies between 0 and 1. The scaled temperature has only one parameter b in addition to the independent variable. That is, the shape of the graph is completely determined by b.

In our previous movie example, we used specific values for D, ω, and k, which then implies a certain $b = D\sqrt{\omega/(2k)}$ (≈ 6.9). However, we can now run different b values and see the effect on the heat propagation. Different b values will in our problems imply different periods of the surface temperature variation and/or different heat conduction values in the ground's composition of rocks. Note that doubling ω and k leaves the same b - it is only the fraction ω/k that influences the value of b.

We can reuse the `animate` function also in the scaled case, but we need to make a new $T(z,t)$ function and, e.g., a main program where b can be read from the command line:

```
def T(z, t):
    return exp(-b*z)*cos(t - b*z)  # b is global

b = float(sys.argv[1])
n = 401
z = linspace(0, 1, n)
animate(3*2*pi, 0.05*2*pi, z, T, -1.2, 1.2, 0, 'z', 'T')
movie('tmp_*.png', encoder='convert', fps=2,
      output_file='tmp_heatwave.gif')
os.system('convert -delay 50 tmp_*.png movie.gif')
```

Running the program, found as the file `heatwave_scaled.py`, for different b values shows that b governs how deep the temperature variations on the surface $z = 0$ penetrate. A large b makes the temperature changes confined to a thin layer close to the surface, while a small b leads to temperature variations also deep down in the ground. You are encouraged to run the program with $b = 2$ and $b = 20$ to experience the major difference, or just view the ready-made animations[7].

We can understand the results from a physical perspective. Think of increasing ω, which means reducing the oscillation period so we get a more rapid temperature variation. To preserve the value of b we must increase k by the same factor. Since a large k means that heat quickly spreads down in the ground, and a small k implies the opposite, we see that more rapid variations at the surface requires a larger k to more quickly conduct the variations down in the ground. Similarly, slow temperature variations on the surface can penetrate deep in the ground even if the ground's ability to conduct (k) is low.

5.9 Exercises

Exercise 5.1: Fill lists with function values

Define

$$h(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \, . \tag{5.15}$$

Fill lists `xlist` and `hlist` with x and $h(x)$ values for 41 uniformly spaced x coordinates in $[-4, 4]$.

Hint. You may adapt the example in Section 5.2.1.
Filename: `fill_lists.py`.

Exercise 5.2: Fill arrays; loop version

The aim is to fill two arrays `x` and `y` with x and $h(x)$ values, respectively, where $h(x)$ is defined in (5.15). Let the x values be as in Exercise 5.1. Create empty `x` and `y` arrays and compute each element in `x` and `y` with a `for` loop. Filename: `fill_arrays_loop.py`.

Exercise 5.3: Fill arrays; vectorized version

Vectorize the code in Exercise 5.2 by creating the x values using the `linspace` function from the `numpy` package and by evaluating $h(x)$ for an array argument. Filename: `fill_arrays_vectorized.py`.

[7] `http://hplgit.github.io/scipro-primer/video/heatwave.html`

Exercise 5.4: Plot a function

Make a plot of the function in Exercise 5.1 for $x \in [-4, 4]$. Filename: `plot_Gaussian.py`.

Exercise 5.5: Apply a function to a vector

Given a vector $v = (2, 3, -1)$ and a function $f(x) = x^3+xe^x+1$, apply f to each element in v. Then calculate by hand $f(v)$ as the NumPy expression `v**3 + v*exp(v) + 1` using vector computing rules. Demonstrate that the two results are equal.

Exercise 5.6: Simulate by hand a vectorized expression

Suppose `x` and `t` are two arrays of the same length, entering a vectorized expression

```
y = cos(sin(x)) + exp(1/t)
```

If `x` holds two elements, 0 and 2, and `t` holds the elements 1 and 1.5, calculate by hand (using a calculator) the `y` array. Thereafter, write a program that mimics the series of computations you did by hand (typically a sequence of operations of the kind we listed in Section 5.1.3 - use explicit loops, but at the end you can use Numerical Python functionality to check the results). Filename: `simulate_vector_computing.py`.

Exercise 5.7: Demonstrate array slicing

Create an array `w` with values $0, 0.1, 0.2, \ldots, 3$. Write out `w[:]`, `w[:-2]`, `w[::5]`, `w[2:-2:6]`. Convince yourself in each case that you understand which elements of the array that are printed. Filename: `slicing.py`.

Exercise 5.8: Replace list operations by array computing

The data analysis problem in Section 2.6.2 is solved by list operations. Convert the list to a two-dimensional array and perform the computations using array operations (i.e., no explicit loops, but you need a loop to make the printout). Filename: `sun_data_vec.py`.

Exercise 5.9: Plot a formula

Make a plot of the function $y(t) = v_0t - \frac{1}{2}gt^2$ for $v_0 = 10$, $g = 9.81$, and $t \in [0, 2v_0/g]$. Set the axes labels as `time (s)` and `height (m)`. Filename: `plot_ball1.py`.

Exercise 5.10: Plot a formula for several parameters

Make a program that reads a set of v_0 values from the command line and plots the corresponding curves $y(t) = v_0 t - \frac{1}{2}gt^2$ in the same figure, with $t \in [0, 2v_0/g]$ for each curve. Set $g = 9.81$.

Hint. You need a different vector of t coordinates for each curve. Filename: `plot_ball2.py`.

Exercise 5.11: Specify the extent of the axes in a plot

Extend the program from Exercises 5.10 such that the minimum and maximum t and y values are computed, and use the extreme values to specify the extent of the axes. Add some space above the highest curve to make the plot look better. Filename: `plot_ball3.py`.

Exercise 5.12: Plot exact and inexact Fahrenheit-Celsius conversion formulas

A simple rule to quickly compute the Celsius temperature from the Fahrenheit degrees is to subtract 30 and then divide by 2: $C = (F-30)/2$. Compare this curve against the exact curve $C = (F - 32)5/9$ in a plot. Let F vary between -20 and 120. Filename: `f2c_shortcut_plot.py`.

Exercise 5.13: Plot the trajectory of a ball

The formula for the trajectory of a ball is given by

$$f(x) = x \tan\theta - \frac{1}{2v_0^2}\frac{gx^2}{\cos^2\theta} + y_0, \tag{5.16}$$

where x is a coordinate along the ground, g is the acceleration of gravity, v_0 is the size of the initial velocity, which makes an angle θ with the x axis, and $(0, y_0)$ is the initial position of the ball.

In a program, first read the input data y_0, θ, and v_0 from the command line. Then plot the trajectory $y = f(x)$ for $y \geq 0$. Filename: `plot_trajectory.py`.

Exercise 5.14: Plot data in a two-column file

The file `src/plot/xy.dat`[8] contains two columns of numbers, corresponding to x and y coordinates on a curve. The start of the file looks as this:

[8] `http://tinyurl.com/pwyasaa/plot/xy.dat`

```
        -1.0000       -0.0000
        -0.9933       -0.0087
        -0.9867       -0.0179
        -0.9800       -0.0274
        -0.9733       -0.0374
```

Make a program that reads the first column into a list `x` and the second column into a list `y`. Plot the curve. Print out the mean y value as well as the maximum and minimum y values.

Hint. Read the file line by line, split each line into words, convert to `float`, and append to `x` and `y`. The computations with `y` are simpler if the list is converted to an array.
Filename: `read_2columns.py`.

Remarks. The function `loadtxt` in `numpy` can read files with tabular data (any number of columns) and return the data in a two-dimensional array:

```
import numpy as np
data = np.loadtxt('xy.dat', dtype=np.float)  # read table of floats
x = data[:,0]  # column with index 0
y = data[:,1]  # column with index 1
```

The present exercise asks you to implement a simplified version of `loadtxt`, but for later loading of a file with tabular data into an array you will certainly use `loadtxt`.

Exercise 5.15: Write function data to file

We want to dump x and $f(x)$ values to a file, where the x values appear in the first column and the $f(x)$ values appear in the second. Choose n equally spaced x values in the interval $[a, b]$. Provide f, a, b, n, and the filename as input data on the command line.

Hint. Use the `StringFunction` tool (see Sections 4.3.3 and 5.5.1) to turn the textual expression for f into a Python function. (Note that the program from Exercise 5.14 can be used to read the file generated in the present exercise into arrays again for visualization of the curve $y = f(x)$.)
Filename: `write_cml_function.py`.

Exercise 5.16: Plot data from a file

The files `density_water.dat` and `density_air.dat` files in the folder `src/plot`[9] contain data about the density of water and air (respectively) for different temperatures. The data files have some comment lines starting with `#` and some lines are blank. The rest of the lines contain density data: the temperature in the first column and the corresponding

[9] `http://tinyurl.com/pwyasaa/plot`

density in the second column. The goal of this exercise is to read the data in such a file and plot the density versus the temperature as distinct (small) circles for each data point. Let the program take the name of the data file as command-line argument. Apply the program to both files. Filename: `read_density_data.py`.

Exercise 5.17: Fit a polynomial to data points

The purpose of this exercise is to find a simple mathematical formula for how the density of water or air depends on the temperature. The idea is to load density and temperature data from file as explained in Exercise 5.16 and then apply some NumPy utilities that can find a polynomial that approximates the density as a function of the temperature.

NumPy has a function `polyfit(x, y, deg)` for finding a "best fit" of a polynomial of degree `deg` to a set of data points given by the array arguments `x` and `y`. The `polyfit` function returns a list of the coefficients in the fitted polynomial, where the first element is the coefficient for the term with the highest degree, and the last element corresponds to the constant term. For example, given points in `x` and `y`, `polyfit(x, y, 1)` returns the coefficients `a, b` in a polynomial `a*x + b` that fits the data in the best way. (More precisely, a line $y = ax + b$ is a "best fit" to the data points (x_i, y_i), $i = 0, \ldots, n-1$ if a and b are chosen to make the sum of squared errors $R = \sum_{j=0}^{n-1}(y_j - (ax_j + b))^2$ as small as possible. This approach is known as *least squares approximation* to data and proves to be extremely useful throughout science and technology.)

NumPy also has a utility `poly1d`, which can take the tuple or list of coefficients calculated by, e.g., `polyfit` and return the polynomial as a Python function that can be evaluated. The following code snippet demonstrates the use of `polyfit` and `poly1d`:

```
coeff = polyfit(x, y, deg)
p = poly1d(coeff)
print p             # prints the polynomial expression
y_fitted = p(x)     # computes the polynomial at the x points
# use red circles for data points and a blue line for the polynomial
plot(x, y, 'ro', x, y_fitted, 'b-',
     legend=('data', 'fitted polynomial of degree %d' % deg))
```

a) Write a function `fit(x, y, deg)` that creates a plot of data in `x` and `y` arrays along with polynomial approximations of degrees collected in the list `deg` as explained above.

b) We want to call `fit` to make a plot of the density of water versus temperature and another plot of the density of air versus temperature. In both calls, use `deg=[1,2]` such that we can compare linear and quadratic approximations to the data.

c) From a visual inspection of the plots, can you suggest simple mathematical formulas that relate the density of air to temperature and the density of water to temperature?
Filename: `fit_density_data.py`.

Exercise 5.18: Fit a polynomial to experimental data

Suppose we have measured the oscillation period T of a simple pendulum with a mass m at the end of a massless rod of length L. We have varied L and recorded the corresponding T value. The measurements are found in a file `src/plot/pendulum.dat`[10]. The first column in the file contains L values and the second column has the corresponding T values.

a) Plot L versus T using circles for the data points.

b) We shall assume that L as a function of T is a polynomial. Use the NumPy utilities `polyfit` and `poly1d`, as explained in Exercise 5.17, to fit polynomials of degree 1, 2, and 3 to the L and T data. Visualize the polynomial curves together with the experimental data. Which polynomial fits the measured data best?
Filename: `fit_pendulum_data.py`.

Exercise 5.19: Read acceleration data and find velocities

A file `src/plot/acc.dat`[11] contains measurements $a_0, a_1, \ldots, a_{n-1}$ of the acceleration of an object moving along a straight line. The measurement a_k is taken at time point $t_k = k\Delta t$, where Δt is the time spacing between the measurements. The purpose of the exercise is to load the acceleration data into a program and compute the velocity $v(t)$ of the object at some time t.

In general, the acceleration $a(t)$ is related to the velocity $v(t)$ through $v'(t) = a(t)$. This means that

$$v(t) = v(0) + \int_0^t a(\tau) d\tau\,. \tag{5.17}$$

If $a(t)$ is only known at some discrete, equally spaced points in time, $a_0, \ldots, a_{n-1}$ (which is the case in this exercise), we must compute the integral in (5.17) numerically, for example by the Trapezoidal rule:

$$v(t_k) \approx \Delta t \left(\frac{1}{2}a_0 + \frac{1}{2}a_k + \sum_{i=1}^{k-1} a_i\right), \quad 1 \leq k \leq n-1\,. \tag{5.18}$$

[10] `http://tinyurl.com/pwyasaa/plot/pendulum.dat`
[11] `http://tinyurl.com/pwyasaa/plot/acc.dat`

We assume $v(0) = 0$ so that also $v_0 = 0$.

Read the values $a_0, \ldots, a_{n-1}$ from file into an array, plot the acceleration versus time, and use (5.18) to compute one $v(t_k)$ value, where Δt and $k \geq 1$ are specified on the command line. Filename: `acc2vel_v1.py`.

Exercise 5.20: Read acceleration data and plot velocities

The task in this exercise is the same as in Exercise 5.19, except that we now want to compute $v(t_k)$ for all time points $t_k = k\Delta t$ and plot the velocity versus time. Now only Δt is given on the command line, and the $a_0, \ldots, a_{n-1}$ values must be read from file as in Exercise 5.19.

Hint. Repeated use of (5.18) for all k values is very inefficient. A more efficient formula arises if we add the area of a new trapezoid to the previous integral (see also Section A.1.7):

$$v(t_k) = v(t_{k-1}) + \int\limits_{t_{k-1}}^{t_k} a(\tau)d\tau \approx v(t_{k-1}) + \Delta t\frac{1}{2}(a_{k-1} + a_k), \qquad (5.19)$$

for $k = 1, 2, \ldots, n-1$, while $v_0 = 0$. Use this formula to fill an array `v` with velocity values.
Filename: `acc2vel.py`.

Exercise 5.21: Plot a trip's path and velocity from GPS coordinates

A GPS device measures your position at every s seconds. Imagine that the positions corresponding to a specific trip are stored as (x, y) coordinates in a file `src/plot/pos.dat`[12] with an x and y number on each line, except for the first line, which contains the value of s.

a) Plot the two-dimensional curve of corresponding to the data in the file.

Hint. Load s into a `float` variable and then the x and y numbers into two arrays. Draw a straight line between the points, i.e., plot the y coordinates versus the x coordinates.

b) Plot the velocity in x direction versus time in one plot and the velocity in y direction versus time in another plot.

[12] `http://tinyurl.com/pwyasaa/plot/pos.dat`

Hint. If $x(t)$ and $y(t)$ are the coordinates of the positions as a function of time, we have that the velocity in x direction is $v_x(t) = dx/dt$, and the velocity in y direction is $v_y = dy/dt$. Since x and y are only known for some discrete times, $t_k = ks$, $k = 0, \ldots, n-1$, we must use numerical differentiation. A simple (forward) formula is

$$v_x(t_k) \approx \frac{x(t_{k+1}) - x(t_k)}{s}, \quad v_y(t_k) \approx \frac{y(t_{k+1}) - y(t_k)}{s}, \quad k = 0, \ldots, n-2\,.$$

Compute arrays `vx` and `vy` with velocities based on the formulas above for $v_x(t_k)$ and $v_y(t_k)$, $k = 0, \ldots, n-2$.
Filename: `position2velocity.py`.

Exercise 5.22: Vectorize the Midpoint rule for integration

The Midpoint rule for approximating an integral can be expressed as

$$\int_a^b f(x)dx \approx h\sum_{i=1}^{n} f(a - \frac{1}{2}h + ih), \tag{5.20}$$

where $h = (b - a)/n$.

a) Write a function `midpointint(f, a, b, n)` to compute Midpoint rule. Use a plain Python `for` loop to implement the sum.

b) Make a vectorized implementation of the Midpoint rule where you compute the sum by Python's built-in function `sum`.

c) Make another vectorized implementation of the Midpoint rule where you compute the sum by the `sum` function in the `numpy` package.

d) Organize the three implementations above in a module file `midpoint_vec.py`.

e) Start IPython, import the functions from `midpoint_vec.py`, define some Python implementation of a mathematical function $f(x)$ to integrate, and use the `%timeit` feature of IPython to measure the efficiency of the three alternative implementations.

Hint. The `%timeit` feature is described in Section H.5.1.
Filename: `midpoint_vec.py`.

Remarks. The lesson learned from the experiments in e) is that `numpy.sum` is much more efficient than Python's built-in function `sum`. Vectorized implementations must always make use of `numpy.sum` to compute sums.

Exercise 5.23: Implement Lagrange's interpolation formula

Imagine we have $n+1$ measurements of some quantity y that depends on x: $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$. We may think of y as a function of x and ask what y is at some arbitrary point x not coinciding with any of the points $x_0, \ldots, x_n$. It is not clear how y varies between the measurement points, but we can make assumptions or models for this behavior. Such a problem is known as *interpolation*.

One way to solve the interpolation problem is to fit a continuous function that goes through all the $n+1$ points and then evaluate this function for any desired x. A candidate for such a function is the polynomial of degree n that goes through all the points. It turns out that this polynomial can be written

$$p_L(x) = \sum_{k=0}^{n} y_k L_k(x), \tag{5.21}$$

where

$$L_k(x) = \prod_{i=0, i\neq k}^{n} \frac{x - x_i}{x_k - x_i}\,. \tag{5.22}$$

The $\prod$ notation corresponds to $\sum$, but the terms are multiplied. For example,

$$\prod_{i=0, i\neq k}^{n} x_i = x_0 x_1 \cdots x_{k-1} x_{k+1} \cdots x_n\,.$$

The polynomial $p_L(x)$ is known as Lagrange's interpolation formula, and the points $(x_0, y_0), \ldots, (x_n, y_n)$ are called interpolation points.

a) Make functions `p_L(x, xp, yp)` and `L_k(x, k, xp, yp)` that evaluate $p_L(x)$ and $L_k(x)$ by (5.21) and (5.22), respectively, at the point `x`. The arrays `xp` and `yp` contain the x and y coordinates of the $n+1$ interpolation points, respectively. That is, `xp` holds $x_0, \ldots, x_n$, and `yp` holds $y_0, \ldots, y_n$.

b) To verify the program, we observe that $L_k(x_k) = 1$ and that $L_k(x_i) = 0$ for $i \neq k$, implying that $p_L(x_k) = y_k$. That is, the polynomial p_L goes through all the points $(x_0, y_0), \ldots, (x_n, y_n)$. Write a function `test_p_L(xp, yp)` that computes $|p_L(x_k) - y_k|$ at all the interpolation points (x_k, y_k) and checks that the value is approximately zero. Call `test_p_L` with `xp` and `yp` corresponding to 5 equally spaced points along the curve $y = \sin(x)$ for $x \in [0, \pi]$. Thereafter, evaluate $p_L(x)$ for an x in the middle of two interpolation points and compare the value of $p_L(x)$ with the exact one.
Filename: `Lagrange_poly1.py`.

Exercise 5.24: Plot Lagrange's interpolating polynomial

a) Write a function `graph(f, n, xmin, xmax, resolution=1001)` for plotting $p_L(x)$ in Exercise 5.23, based on interpolation points taken from some mathematical function $f(x)$ represented by the argument `f`. The argument `n` denotes the number of interpolation points sampled from the $f(x)$ function, and `resolution` is the number of points between `xmin` and `xmax` used to plot $p_L(x)$. The x coordinates of the `n` interpolation points can be uniformly distributed between `xmin` and `xmax`. In the graph, the interpolation points $(x_0, y_0), \ldots, (x_n, y_n)$ should be marked by small circles. Test the `graph` function by choosing 5 points in $[0, \pi]$ and `f` as $\sin x$.

b) Make a module `Lagrange_poly2` containing the `p_L`, `L_k`, `test_p_L`, and `graph` functions. The call to `test_p_L` described in Exercise 5.23 and the call to `graph` described above should appear in the module's test block.

Hint. Section 4.9 describes how to make a module. In particular, a test block is explained in Section 4.9.3, test functions like `test_p_L` are demonstrated in Section 4.9.4 and also in Section 3.4.2, and how to combine `test_p_L` and `graph` calls in the test block is exemplified in Section 4.9.5.
Filename: `Lagrange_poly2.py`.

Exercise 5.25: Investigate the behavior of Lagrange's interpolating polynomials

Unfortunately, the polynomial $p_L(x)$ defined and implemented in Exercise 5.23 can exhibit some undesired oscillatory behavior that we shall explore graphically in this exercise. Call the `graph` function from Exercise 5.24 with $f(x) = |x|$, $x \in [-2, 2]$, for $n = 2, 4, 6, 10$. All the graphs of $p_L(x)$ should appear in the same plot for comparison. In addition, make a new figure with calls to `graph` for $n = 13$ and $n = 20$. All the code necessary for solving this exercise should appear in some separate program file, which imports the `Lagrange_poly2` module made in Exercise 5.24.
Filename: `Lagrange_poly2b.py`.

Remarks. The purpose of the $p_L(x)$ function is to compute (x, y) between some given (often measured) data points $(x_0, y_0), \ldots, (x_n, y_n)$. We see from the graphs that for a small number of interpolation points, $p_L(x)$ is quite close to the curve $y = |x|$ we used to generate the data points, but as n increases, $p_L(x)$ starts to oscillate, especially toward the end points (x_0, y_0) and (x_n, y_n). Much research has historically been focused on methods that do not result in such strange oscillations when fitting a polynomial to a set of points.

Exercise 5.26: Plot a wave packet

The function

$$f(x,t) = e^{-(x-3t)^2}\sin\left(3\pi(x-t)\right) \tag{5.23}$$

describes for a fixed value of t a wave localized in space. Make a program that visualizes this function as a function of x on the interval $[-4,4]$ when $t=0$. Filename: `plot_wavepacket.py`.

Exercise 5.27: Judge a plot

Assume you have the following program for plotting a parabola:

```
import numpy as np
x = np.linspace(0, 2, 20)
y = x*(2 - x)
import matplotlib.pyplot as plt
plt.plot(x, y)
plt.show()
```

Then you switch to the function $\cos(18\pi x)$ by altering the computation of `y` to `y = cos(18*pi*x)`. Judge the resulting plot. Is it correct? Display the $\cos(18\pi x)$ function with 1000 points in the same plot. Filename: `judge_plot.py`.

Exercise 5.28: Plot the viscosity of water

The viscosity of water, μ, varies with the temperature T (in Kelvin) according to

$$\mu(T) = A\cdot 10^{B/(T-C)}, \tag{5.24}$$

where $A = 2.414\cdot 10^{-5}$ Pa s, $B = 247.8$ K, and $C = 140$ K. Plot $\mu(T)$ for T between 0 and 100 degrees Celsius. Label the x axis with 'temperature (C)' and the y axis with 'viscosity (Pa s)'. Note that T in the formula for μ must be in Kelvin. Filename: `water_viscosity.py`.

Exercise 5.29: Explore a complicated function graphically

The wave speed c of water surface waves depends on the length λ of the waves. The following formula relates c to λ:

$$c(\lambda) = \sqrt{\frac{g\lambda}{2\pi}\left(1 + s\frac{4\pi^2}{\rho g\lambda^2}\right)\tanh\left(\frac{2\pi h}{\lambda}\right)}\,. \tag{5.25}$$

Here, g is the acceleration of gravity (9.81 m/s^2), s is the air-water surface tension ($7.9\cdot 10^{-2}$ N/m) , ρ is the density of water (can be taken as

1000 kg/m^3), and h is the water depth. Let us fix h at 50 m. First make a plot of $c(\lambda)$ (in m/s) for small λ (0.001 m to 0.1 m). Then make a plot $c(\lambda)$ for larger λ (1 m to 2 km. Filename: `water_wave_velocity.py`.

Exercise 5.30: Plot Taylor polynomial approximations to $\sin x$

The sine function can be approximated by a polynomial according to the following formula:

$$\sin x \approx S(x;n) = \sum_{j=0}^{n} (-1)^j \frac{x^{2j+1}}{(2j+1)!}\,. \tag{5.26}$$

The expression $(2j+1)!$ is the factorial (`math.factorial` can compute this quantity). The error in the approximation $S(x;n)$ decreases as n increases and in the limit we have that $\lim_{n\to\infty} S(x;n) = \sin x$. The purpose of this exercise is to visualize the quality of various approximations $S(x;n)$ as n increases.

a) Write a Python function `S(x, n)` that computes $S(x;n)$. Use a straightforward approach where you compute each term as it stands in the formula, i.e., $(-1)^j x^{2j+1}$ divided by the factorial $(2j+1)!$. (We remark that Exercise A.14 outlines a much more efficient computation of the terms in the series.)

b) Plot $\sin x$ on $[0, 4\pi]$ together with the approximations $S(x;1)$, $S(x;2)$, $S(x;3)$, $S(x;6)$, and $S(x;12)$.
Filename: `plot_Taylor_sin.py`.

Exercise 5.31: Animate a wave packet

Display an animation of the function $f(x,t)$ in Exercise 5.26 by plotting f as a function of x on $[-6, 6]$ for a set of t values in $[-1, 1]$. Also make an animated GIF file.

Hint. A suitable resolution can be 1000 intervals (1001 points) along the x axis, 60 intervals (61 points) in time, and 6 frames per second in the animated GIF file. Use the recipe in Section 5.3.4 and remember to remove the family of old plot files in the beginning of the program.
Filename: `plot_wavepacket_movie.py`.

Exercise 5.32: Animate a smoothed Heaviside function

Visualize the smoothed Heaviside function $H_\epsilon(x)$, defined in (3.25), as an animation where ϵ starts at 2 and then goes to zero. Filename: `smoothed_Heaviside_movie.py`.

Exercise 5.33: Animate two-scale temperature variations

We consider temperature oscillations in the ground as addressed in Section 5.8.2. Now we want to visualize daily and annual variations. Let A_1 be the amplitude of annual variations and A_2 the amplitude of the day/night variations. Let also $P_1 = 365$ days and $P_2 = 24$ h be the periods of the annual and the daily oscillations. The temperature at time t and depth z is then given by

$$T(z,t) = T_0 + A_1 e^{-a_1 z}\sin(\omega_1 t - a_1 z) + A_2 e^{-a_2 z}\sin(\omega_2 t - a_2 z), \quad (5.27)$$

where

$$\begin{aligned}\omega_1 &= 2\pi P_1,\\ \omega_2 &= 2\pi P_2,\\ a_1 &= \sqrt{\frac{\omega_1}{2k}},\\ a_2 &= \sqrt{\frac{\omega_2}{2k}}\,.\end{aligned}$$

Choose $k = 10^{-6}$ m^2/s, $A_1 = 15$ C, $A_2 = 7$ C, and the resolution Δt as $P_2/10$. Modify the `heatwave.py` program in order to animate this new temperature function. Filename: `heatwave2.py`.

Remarks. We assume in this problem that the temperature T equals the reference temperature T_0 at $t = 0$, resulting in a sine variation rather than the cosine variation in (5.13).

Exercise 5.34: Use non-uniformly distributed coordinates for visualization

Watching the animation in Exercise 5.33 reveals that there are rapid oscillations in a small layer close to $z = 0$. The variations away from $z = 0$ are much smaller in time and space. It would therefore be wise to use more z coordinates close to $z = 0$ than for larger z values. Given a set $x_0 < x_1 < \cdots < x_n$ of uniformly spaced coordinates in $[a, b]$, we can compute new coordinates $\bar{x}_i$, stretched toward $x = a$, by the formula

$$\bar{x}_i = a + (b-a)\left(\frac{x_i - a}{b-a}\right)^s,$$

for some $s > 1$. In the present example, we can use this formula to stretch the z coordinates to the left.

a) Experiment with $s \in [1.2, 3]$ and few points (say 15) and visualize the curve as a line with circles at the points so that you can easily see the distribution of points toward the left end. Identify a suitable value of s.

b) Run the animation with no circles and (say) 501 points with the found s value.
Filename: `heatwave2a.py`.

Exercise 5.35: Animate a sequence of approximations to π

Exercise 3.13 outlines an idea for approximating π as the length of a polygon inside the circle. Wrap the code from Exercise 3.13 in a function `pi_approx(N)`, which returns the approximation to π using a polygon with $N+1$ equally distributed points. The task of the present exercise is to visually display the polygons as a movie, where each frame shows the polygon with $N+1$ points together with the circle and a title reflecting the corresponding error in the approximate value of π. The whole movie arises from letting N run through $4, 5, 6, \ldots, K$, where K is some (large) prescribed value. Let there be a pause of 0.3 s between each frame in the movie. By playing the movie you will see how the polygons move closer and closer to the circle and how the approximation to π improves.
Filename: `pi_polygon_movie.py`.

Exercise 5.36: Animate a planet's orbit

A planet's orbit around a star has the shape of an ellipse. The purpose of this exercise is to make an animation of the movement along the orbit. One should see a small disk, representing the planet, moving along an elliptic curve. An evolving solid line shows the development of the planet's orbit as the planet moves and the title displays the planet's instantaneous velocity magnitude. As a test, run the special case of a circle and verify that the magnitude of the velocity remains constant as the planet moves.

Hint 1. The points (x, y) along the ellipse are given by the expressions

$$x = a\cos(\omega t), \quad y = b\sin(\omega t),$$

where a is the semi-major axis of the ellipse, b is the semi-minor axis, ω is an angular velocity of the planet around the star, and t denotes time. One complete orbit corresponds to $t \in [0, 2\pi/\omega]$. Let us discretize time into time points $t_k = k\Delta t$, where $\Delta t = 2\pi/(\omega n)$. Each frame in the movie corresponds to (x, y) points along the curve with t values $t_0, t_1, \ldots, t_i$, i representing the frame number ($i = 1, \ldots, n$).

Hint 2. The velocity vector is

$$(\frac{dx}{dt}, \frac{dy}{dt}) = (-\omega a \sin(\omega t), \omega b \cos(\omega t)),$$

and the magnitude of this vector becomes $\omega\sqrt{a^2 \sin^2(\omega t) + b^2 \cos^2(\omega t)}$. Filename: `planet_orbit.py`.

Exercise 5.37: Animate the evolution of Taylor polynomials

A general series approximation (to a function) can be written as

$$S(x; M, N) = \sum_{k=M}^{N} f_k(x)\,.$$

For example, the Taylor polynomial of degree N for e^x equals $S(x; 0, N)$ with $f_k(x) = x^k/k!$. The purpose of the exercise is to make a movie of how $S(x; M, N)$ develops and improves as an approximation as we add terms in the sum. That is, the frames in the movie correspond to plots of $S(x; M, M)$, $S(x; M, M + 1)$, $S(x; M, M + 2)$, …, $S(x; M, N)$.

a) Make a function

```
animate_series(fk, M, N, xmin, xmax, ymin, ymax, n, exact)
```

for creating such animations. The argument `fk` holds a Python function implementing the term $f_k(x)$ in the sum, `M` and `N` are the summation limits, the next arguments are the minimum and maximum x and y values in the plot, `n` is the number of x points in the curves to be plotted, and `exact` holds the function that $S(x)$ aims at approximating.

Hint. Here is some more information on how to write the `animate_series` function. The function must accumulate the $f_k(x)$ terms in a variable s, and for each k value, s is plotted against x together with a curve reflecting the exact function. Each plot must be saved in a file, say with names `tmp_0000.png`, `tmp_0001.png`, and so on (these filenames can be generated by `tmp_%04d.png`, using an appropriate counter). Use the `movie` function to combine all the plot files into a movie in a desired movie format.

In the beginning of the `animate_series` function, it is necessary to remove all old plot files of the form `tmp_*.png`. This can be done by the `glob` module and the `os.remove` function as exemplified in Section 5.3.4.

b) Call the `animate_series` function for the Taylor series for $\sin x$, where $f_k(x) = (-1)^k x^{2k+1}/(2k + 1)!$, and $x \in [0, 13\pi]$, $M = 0$, $N = 40$, $y \in [-2, 2]$.

c) Call the `animate_series` function for the Taylor series for e^{-x}, where $f_k(x) = (-x)^k/k!$, and $x \in [0, 15]$, $M = 0$, $N = 30$, $y \in [-0.5, 1.4]$. Filename: `animate_Taylor_series.py`.

Exercise 5.38: Plot the velocity profile for pipeflow

A fluid that flows through a (very long) pipe has zero velocity on the pipe wall and a maximum velocity along the centerline of the pipe. The velocity v varies through the pipe cross section according to the following formula:

$$v(r) = \left(\frac{\beta}{2\mu_0}\right)^{1/n} \frac{n}{n+1}\left(R^{1+1/n} - r^{1+1/n}\right), \tag{5.28}$$

where R is the radius of the pipe, β is the pressure gradient (the force that drives the flow through the pipe), μ_0 is a viscosity coefficient (small for air, larger for water and even larger for toothpaste), n is a real number reflecting the viscous properties of the fluid ($n = 1$ for water and air, $n < 1$ for many modern plastic materials), and r is a radial coordinate that measures the distance from the centerline ($r = 0$ is the centerline, $r = R$ is the pipe wall).

a) Make a Python function that evaluates $v(r)$.

b) Plot $v(r)$ as a function of $r \in [0, R]$, with $R = 1$, $\beta = 0.02$, $\mu_0 = 0.02$, and $n = 0.1$.

c) Make an animation of how the $v(r)$ curves varies as n goes from 1 and down to 0.01. Because the maximum value of $v(r)$ decreases rapidly as n decreases, each curve can be normalized by its $v(0)$ value such that the maximum value is always unity.
Filename: `plot_velocity_pipeflow.py`.

Exercise 5.39: Plot sum-of-sines approximations to a function

Exercise 3.15 defines the approximation $S(t; n)$ to a function $f(t)$. Plot $S(t; 1)$, $S(t; 3)$, $S(t; 20)$, $S(t; 200)$, and the exact $f(t)$ function in the same plot. Use $T = 2\pi$. Filename: `sinesum1_plot.py`.

Exercise 5.40: Animate the evolution of a sum-of-sine approximation to a function

First perform Exercise 5.39. A natural next step is to animate the evolution of $S(t; n)$ as n increases. Create such an animation and observe how the discontinuity in $f(t)$ is poorly approximated by $S(t; n)$, even when n grows large (plot $f(t)$ in each frame). This is a well-known deficiency, called Gibb's phenomenon, when approximating discontinuous functions by sine or cosine (Fourier) series. Filename: `sinesum1_movie.py`.

Exercise 5.41: Plot functions from the command line

For quickly getting a plot a function $f(x)$ for $x \in [x_{\min}, x_{\max}]$ it could be nice to a have a program that takes the minimum amount of information from the command line and produces a plot on the screen and saves the plot to a file `tmp.png`. The usage of the program goes as follows:

Terminal
```
plotf.py "f(x)" xmin xmax
```

A specific example is

Terminal
```
plotf.py "exp(-0.2*x)*sin(2*pi*x)" 0 4*pi
```

Write the `plotf.py` program with as short code as possible (we leave it to Exercise 5.42 to test for valid input).

Hint. Make x coordinates from the second and third command-line arguments and then use `eval` (or `StringFunction` from `scitools.std`, see Sections 4.3.3 and 5.5.1) on the first argument.
Filename: `plotf.py`.

Exercise 5.42: Improve command-line input

Equip the program from Exercise 5.41 with tests on valid input on the command line. Also allow an optional fourth command-line argument for the number of points along the function curve. Set this number to 501 if it is not given. Filename: `plotf2.py`.

Exercise 5.43: Demonstrate energy concepts from physics

The vertical position $y(t)$ of a ball thrown upward is given by $y(t) = v_0 t - \frac{1}{2}gt^2$, where g is the acceleration of gravity and v_0 is the velocity at $t = 0$. Two important physical quantities in this context are the potential energy, obtained by doing work against gravity, and the kinetic energy, arising from motion. The potential energy is defined as $P = mgy$, where m is the mass of the ball. The kinetic energy is defined as $K = \frac{1}{2}mv^2$, where v is the velocity of the ball, related to y by $v(t) = y'(t)$.

Make a program that can plot $P(t)$ and $K(t)$ in the same plot, along with their sum $P + K$. Let $t \in [0, 2v_0/g]$. Read m and v_0 from the command line. Run the program with various choices of m and v_0 and observe that $P + K$ is always constant in this motion. (In fact, it turns out that $P + K$ is constant for a large class of motions, and this is a very important result in physics.) Filename: `energy_physics.py`.

Exercise 5.44: Plot a w-like function

Define mathematically a function that looks like the “w” character. Plot this function. Filename: `plot_w.py`.

Exercise 5.45: Plot a piecewise constant function

Consider the piecewise constant function defined in Exercise 3.26. Make a Python function `plot_piecewise(data, xmax)` that draws a graph of the function, where `data` is the nested list explained in Exercise 3.26 and `xmax` is the maximum x coordinate. Use ideas from Section 5.4.1. Filename: `plot_piecewise_constant.py`.

Exercise 5.46: Vectorize a piecewise constant function

Consider the piecewise constant function defined in Exercise 3.26. Make a vectorized implementation `piecewise_constant_vec(x, data, xmax)` of such a function, where `x` is an array.

Hint. You can use ideas from the `Nv1` function in Section 5.5.3. However, since the number of intervals is not known, it is necessary to store the various intervals and conditions in lists.
Filename: `piecewise_constant_vec.py`.

Remarks. Plotting the array returned from `piecewise_constant_vec` faces the same problems as encountered in Section 5.4.1. It is better to make a custom plotting function that simply draws straight horizontal lines in each interval (Exercise 5.45).

Exercise 5.47: Visualize approximations in the Midpoint integration rule

Consider the midpoint rule for integration from Exercise 3.7. Use Matplotlib to make an illustration of the midpoint rule as shown to the left in Figure 5.12.

The $f(x)$ function used in Figure 5.12 is

$$f(x) = x(12 - x) + \sin(\pi x), \quad x \in [0, 10]\,.$$

Hint. Look up the documentation of the Matplotlib function `fill_between` and use this function to create the filled areas between $f(x)$ and the approximating rectangles.

Note that the `fill_between` requires the two curves to have the same number of points. For accurate visualization of $f(x)$ you need quite many x coordinates, and the rectangular approximation to $f(x)$ must be drawn using the same set of x coordinates.
Filename: `viz_midpoint.py`.

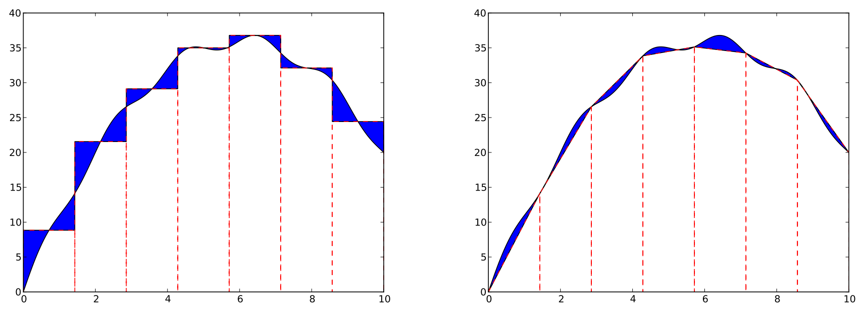


Fig. 5.12 Visualization of numerical integration rules, with the Midpoint rule to the left and the Trapezoidal rule to the right. The filled areas illustrate the deviations in the approximation of the area under the curve.

Exercise 5.48: Visualize approximations in the Trapezoidal integration rule

Redo Exercise 5.47 for the Trapezoidal rule from Exercise 3.6 to produce the graph shown to the right in Figure 5.12. Filename: `viz_trapezoidal.py`.

Exercise 5.49: Experience overflow in a function

We are give the mathematical function

$$v(x) = \frac{1 - e^{x/\mu}}{1 - e^{1/\mu}},$$

where μ is a parameter.

a) Make a Python function `v(x, mu=1E-6, exp=math.exp)` for calculating the formula for $v(x)$ using `exp` as a possibly user-given exponential function. Let the `v` function return the nominator and denominator in the formula as well as the fraction.

b) Call the `v` function for various `x` values between 0 and 1 in a `for` loop, let `mu` be `1E-3`, and have an inner `for` loop over two different `exp` functions: `math.exp` and `numpy.exp`. The output will demonstrate how the denominator is subject to overflow and how difficult it is to calculate this function on a computer.

c) Plot $v(x)$ for $\mu = 1, 0.01, 0.001$ on $[0, 1]$ using 10,000 points to see what the function looks like.

d) Convert `x` and `eps` to a higher precision representation of real numbers, with the aid of the NumPy type `float96`, before calling `v`:

```
import numpy
x = numpy.float96(x); mu = numpy.float96(e)
```

Repeat point b) with these type of variables and observe how much better results we get with `float96` compared with the standard `float` value, which is `float64` (the number reflects the number of bits in the machine's representation of a real number).

e) Call the `v` function with `x` and `mu` as `float32` variables and report how the function now behaves.
Filename: `boundary_layer_func1.py`.

Remarks. When an object (ball, car, airplane) moves through the air, there is a very, very thin layer of air close to the object's surface where the air velocity varies dramatically, from the same value as the velocity of the object at the object's surface to zero a few centimeters away. This layer is called a *boundary layer*. The physics in the boundary layer is important for air resistance and cooling/heating of objects. The change in velocity in the boundary layer is quite abrupt and can be modeled by the functiion $v(x)$, where $x = 1$ is the object's surface, and $x = 0$ is some distance away where one cannot notice any wind velocity v because of the passing object ($v = 0$). The wind velocity coincides with the velocity of the object at $x = 1$, here set to $v = 1$. The parameter μ is very small and related to the viscosity of air. With a small value of μ, it becomes difficult to calculate $v(x)$ on a computer. The exercise demonstrates the difficulties and provides a remedy.

Exercise 5.50: Apply a function to a rank 2 array

Let A be the two-dimensional array

$$\begin{bmatrix} 0 & 2 & -1 \\ -1 & -1 & 0 \\ 0 & 5 & 0 \end{bmatrix}$$

Apply the function f from Exercise 5.5 to each element in A. Then calculate the result of the array expression `A**3 + A*exp(A) + 1`, and demonstrate that the end result of the two methods are the same.

Exercise 5.51: Explain why array computations fail

The following loop computes the array `y` from `x`:

```
>>> import numpy as np
>>> x = np.linspace(0, 1, 3)
>>> y = np.zeros(len(x))
>>> for i in range(len(x)):
...     y[i] = x[i] + 4
```

However, the alternative loop

```
>>> for xi, yi in zip(x, y):
...     yi = xi + 5
```

leaves `y` unchanged. Why? Explain in detail what happens in each pass of this loop and write down the contents of `xi`, `yi`, `x`, and `y` as the loop progresses.

A Sequences and difference equations

From mathematics you probably know the concept of a *sequence*, which is nothing but a collection of numbers with a specific order. A general sequence is written as

$$x_0,\ x_1,\ x_2,\ \ldots,\ x_n, \ldots,$$

One example is the sequence of all odd numbers:

$$1, 3, 5, 7, \ldots, 2n+1, \ldots$$

For this sequence we have an explicit formula for the n-th term: $2n+1$, and n takes on the values 0, 1, 2, …. We can write this sequence more compactly as $(x_n)_{n=0}^{\infty}$ with $x_n = 2n+1$. Other examples of infinite sequences from mathematics are

$$1,\ 4,\ 9,\ 16,\ 25,\ \ldots \quad (x_n)_{n=0}^{\infty},\ x_n = (n+1)^2, \tag{A.1}$$

$$1,\ \frac{1}{2},\ \frac{1}{3},\ \frac{1}{4},\ \ldots \quad (x_n)_{n=0}^{\infty},\ x_n = \frac{1}{n+1}\,. \tag{A.2}$$

The former sequences are infinite, because they are generated from all integers ≥ 0 and there are infinitely many such integers. Nevertheless, most sequences from real life applications are finite. If you put an amount x_0 of money in a bank, you will get an interest rate and therefore have an amount x_1 after one year, x_2 after two years, and x_N after N years. This process results in a finite sequence of amounts

$$x_0, x_1, x_2, \ldots, x_N, \quad (x_n)_{n=0}^{N}\,.$$

Usually we are interested in quite small N values (typically $N \leq 20-30$). Anyway, the life of the bank is finite, so the sequence definitely has an end.

For some sequences it is not so easy to set up a general formula for the n-th term. Instead, it is easier to express a relation between two or more consecutive elements. One example where we can do both things is the sequence of odd numbers. This sequence can alternatively be generated by the formula

$$x_{n+1} = x_n + 2\,. \tag{A.3}$$

To start the sequence, we need an *initial condition* where the value of the first element is specified:

$$x_0 = 1\,.$$

Relations like (A.3) between consecutive elements in a sequence is called recurrence relations or *difference equations*. Solving a difference equation can be quite challenging in mathematics, but it is almost trivial to solve it on a computer. That is why difference equations are so well suited for computer programming, and the present appendix is devoted to this topic. Necessary background knowledge is programming with loops, arrays, and command-line arguments and visualization of a function of one variable.

The program examples regarding difference equations are found in the folder `src/diffeq`[1].

A.1 Mathematical models based on difference equations

The objective of science is to understand complex phenomena. The phenomenon under consideration may be a part of nature, a group of social individuals, the traffic situation in Los Angeles, and so forth. The reason for addressing something in a scientific manner is that it appears to be complex and hard to comprehend. A common scientific approach to gain understanding is to create a model of the phenomenon, and discuss the properties of the model instead of the phenomenon. The basic idea is that the model is easier to understand, but still complex enough to preserve the basic features of the problem at hand.

Essentially, all models are wrong, but some are useful. George E. P. Box, statistician, 1919-2013.

Modeling is, indeed, a general idea with applications far beyond science. Suppose, for instance, that you want to invite a friend to your home for the first time. To assist your friend, you may send a map of your neighborhood. Such a map is a model: it exposes the most important landmarks and leaves out billions of details that your friend can do very well without. This is the essence of modeling: a good model should be

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

as simple as possible, but still rich enough to include the important structures you are looking for.

Everything should be made as simple as possible, but not simpler.
Paraphrased quote attributed to Albert Einstein, physicist, 1879-1955.

Certainly, the tools we apply to model a certain phenomenon differ a lot in various scientific disciplines. In the natural sciences, mathematics has gained a unique position as the key tool for formulating models. To establish a model, you need to understand the problem at hand and describe it with mathematics. Usually, this process results in a set of equations, i.e., the model consists of equations that must be solved in order to see how realistically the model describes a phenomenon. Difference equations represent one of the simplest yet most effective type of equations arising in mathematical models. The mathematics is simple and the programming is simple, thereby allowing us to focus more on the modeling part. Below we will derive and solve difference equations for diverse applications.

A.1.1 Interest rates

Our first difference equation model concerns how much money an initial amount x_0 will grow to after n years in a bank with annual interest rate p. You learned in school the formula

$$x_n = x_0\left(1+\frac{p}{100}\right)^n . \tag{A.4}$$

Unfortunately, this formula arises after some limiting assumptions, like that of a constant interest rate over all the n years. Moreover, the formula only gives us the amount after each year, not after some months or days. It is much easier to compute with interest rates if we set up a more fundamental model in terms of a difference equation and then solve this equation on a computer.

The fundamental model for interest rates is that an amount x_{n-1} at some point of time t_{n-1} increases its value with p percent to an amount x_n at a new point of time t_n:

$$x_n = x_{n-1} + \frac{p}{100}x_{n-1} . \tag{A.5}$$

If n counts years, p is the annual interest rate, and if p is constant, we can with some arithmetics derive the following solution to (A.5):

$$x_n = \left(1+\frac{p}{100}\right)x_{n-1} = \left(1+\frac{p}{100}\right)^2 x_{n-2} = \ldots = \left(1+\frac{p}{100}\right)^n x_0 .$$

Instead of first deriving a formula for x_n and then program this formula, we may attack the fundamental model (A.5) in a program (`growth_years.py`) and compute x_1, x_2, and so on in a loop:

```
from scitools.std import *
x0 = 100                      # initial amount
p = 5                         # interest rate
N = 4                         # number of years
index_set = range(N+1)
x = zeros(len(index_set))

# Compute solution
x[0] = x0
for n in index_set[1:]:
    x[n] = x[n-1] + (p/100.0)*x[n-1]
print x
plot(index_set, x, 'ro', xlabel='years', ylabel='amount')
```

The output of `x` is

```
[ 100.       105.       110.25      115.7625    121.550625]
```

Programmers of mathematical software who are trained in making programs more efficient, will notice that it is not necessary to store all the x_n values in an array or use a list with all the indices $0, 1, \ldots, N$. Just one integer for the index and two floats for x_n and x_{n-1} are strictly necessary. This can save quite some memory for large values of N. Exercise A.3 asks you to develop such a memory-efficient program.

Suppose now that we are interested in computing the growth of money after N days instead. The interest rate per day is taken as $r = p/D$ if p is the annual interest rate and D is the number of days in a year. The fundamental model is the same, but now n counts days and p is replaced by r:

$$x_n = x_{n-1} + \frac{r}{100}x_{n-1}\,. \tag{A.6}$$

A common method in international business is to choose $D = 360$, yet let n count the exact number of days between two dates (see the Wikipedia entry Day count convention[2] for an explanation). Python has a module `datetime` for convenient calculations with dates and times. To find the number of days between two dates, we perform the following operations:

```
>>> import datetime
>>> date1 = datetime.date(2007, 8, 3)  # Aug 3, 2007
>>> date2 = datetime.date(2008, 8, 4)  # Aug 4, 2008
>>> diff = date2 - date1
>>> print diff.days
367
```

We can modify the previous program to compute with days instead of years:

```
from scitools.std import *
x0 = 100                          # initial amount
p = 5                             # annual interest rate
r = p/360.0                       # daily interest rate
```

[2] `http://en.wikipedia.org/wiki/Day_count_convention`

```
    import datetime
    date1 = datetime.date(2007, 8, 3)
    date2 = datetime.date(2011, 8, 3)
    diff = date2 - date1
    N = diff.days
    index_set = range(N+1)
    x = zeros(len(index_set))

    # Compute solution
    x[0] = x0
    for n in index_set[1:]:
        x[n] = x[n-1] + (r/100.0)*x[n-1]
    print x
    plot(index_set, x, 'ro', xlabel='days', ylabel='amount')
```

Running this program, called `growth_days.py`, prints out 122.5 as the final amount.

It is quite easy to adjust the formula (A.4) to the case where the interest is added every day instead of every year. However, the strength of the model (A.6) and the associated program `growth_days.py` becomes apparent when r varies in time - and this is what happens in real life. In the model we can just write $r(n)$ to explicitly indicate the dependence upon time. The corresponding time-dependent annual interest rate is what is normally specified, and $p(n)$ is usually a piecewise constant function (the interest rate is changed at some specific dates and remains constant between these days). The construction of a corresponding array `p` in a program, given the dates when p changes, can be a bit tricky since we need to compute the number of days between the dates of changes and index `p` properly. We do not dive into these details now, but readers who want to compute `p` and who is ready for some extra brain training and index puzzling can attack Exercise A.8. For now we assume that an array `p` holds the time-dependent annual interest rates for each day in the total time period of interest. The `growth_days.py` program then needs a slight modification, typically,

```
p = zeros(len(index_set))
# set up p (might be challenging!)
r = p/360.0                          # daily interest rate
...
for n in index_set[1:]:
    x[n] = x[n-1] + (r[n-1]/100.0)*x[n-1]
```

For the very simple (and not-so-relevant) case where p grows linearly (i.e., daily changes) from 4 to 6 percent over the period of interest, we have made a complete program in the file `growth_days_timedep.py`. You can compare a simulation with linearly varying p between 4 and 6 and a simulation using the average p value 5 throughout the whole time interval.

A difference equation with $r(n)$ is quite difficult to solve mathematically, but the n-dependence in r is easy to deal with in the computerized solution approach.

A.1.2 The factorial as a difference equation

The difference equation

$$x_n = nx_{n-1}, \quad x_0 = 1 \tag{A.7}$$

can quickly be solved recursively:

$$\begin{aligned} x_n &= nx_{n-1} \\ &= n(n-1)x_{n-2} \\ &= n(n-1)(n-2)x_{n-3} \\ &= n(n-1)(n-2)\cdots 1\,. \end{aligned}$$

The result x_n is nothing but the factorial of n, denoted as $n!$. Equation (A.7) then gives a standard recipe to compute $n!$.

A.1.3 Fibonacci numbers

Every textbook with some material on sequences usually presents a difference equation for generating the famous Fibonacci numbers[3]:

$$x_n = x_{n-1} + x_{n-2}, \quad x_0 = 1,\ x_1 = 1,\ n = 2, 3, \ldots \tag{A.8}$$

This equation has a relation between three elements in the sequence, not only two as in the other examples we have seen. We say that this is a difference equation of second order, while the previous examples involving two n levels are said to be difference equations of first order. The precise characterization of (A.8) is a homogeneous difference equation of second order. Such classification is not important when computing the solution in a program, but for mathematical solution methods by pen and paper, the classification helps determine the most suitable mathematical technique for solving the problem.

A straightforward program for generating Fibonacci numbers takes the form (`fibonacci1.py`):

```
import sys
import numpy as np
N = int(sys.argv[1])
x = np.zeros(N+1, int)
x[0] = 1
x[1] = 1
for n in range(2, N+1):
    x[n] = x[n-1] + x[n-2]
    print n, x[n]
```

Since x_n is an infinite sequence we could try to run the program for very large N. This causes two problems: the storage requirements of the `x` array may become too large for the computer, but long before this

[3] `http://en.wikipedia.org/wiki/Fibonacci_number`

happens, x_n grows in size far beyond the largest integer that can be represented by `int` elements in arrays (the problem appears already for $N = 50$). A possibility is to use array elements of type `int64`, which allows computation of twice as many numbers as with standard `int` elements (see the program `fibonacci1_int64.py`). A better solution is to use `float` elements in the `x` array, despite the fact that the numbers x_n are integers. With `float96` elements we can compute up to $N = 23600$ (see the program `fibinacci1_float.py`).

The best solution goes as follows. We observe, as mentioned after the `growth_years.py` program and also explained in Exercise A.3, that we need only three variables to generate the sequence. We can therefore work with just three standard `int` variables in Python:

```
import sys
N = int(sys.argv[1])
xnm1 = 1
xnm2 = 1
n = 2
while n <= N:
    xn = xnm1 + xnm2
    print 'x_%d = %d' % (n, xn)
    xnm2 = xnm1
    xnm1 = xn
    n += 1
```

Here `xnm1` denotes x_{n-1} and `xnm2` denotes x_{n-2}. To prepare for the next pass in the loop, we must shuffle the `xnm1` down to `xnm2` and store the new x_n value in `xnm1`. The nice thing with integers in Python (contrary to `int` elements in NumPy arrays) is that they can hold integers of arbitrary size. More precisely, when the integer is too large for the ordinary `int` object, `xn` becomes a `long` object that can hold integers as big as the computer's memory allows. We may try a run with `N` set to 250:

```
x_2 = 2
x_3 = 3
x_4 = 5
x_5 = 8
x_6 = 13
x_7 = 21
x_8 = 34
x_9 = 55
x_10 = 89
x_11 = 144
x_12 = 233
x_13 = 377
x_14 = 610
x_15 = 987
x_16 = 1597
...
x_249 = 7896325826131730509282738943634332893686268675876375
x_250 = 12776523572924732586037033894655031898659556447352249
```

In mathematics courses you learn how to derive a formula for the n-th term in a Fibonacci sequence. This derivation is much more complicated than writing a simple program to generate the sequence, but there is a lot of interesting mathematics both in the derivation and the resulting formula!

A.1.4 Growth of a population

Let x_{n-1} be the number of individuals in a population at time t_{n-1}. The population can consists of humans, animals, cells, or whatever objects where the number of births and deaths is proportional to the number of individuals. Between time levels t_{n-1} and t_n, bx_{n-1} individuals are born, and dx_{n-1} individuals die, where b and d are constants. The net growth of the population is then $(b-d)x_n$. Introducing $r=(b-d)100$ for the net growth factor measured in percent, the new number of individuals become

$$x_n = x_{n-1} + \frac{r}{100}x_{n-1}\,. \tag{A.9}$$

This is the same difference equation as (A.5). It models growth of populations quite well as long as there are optimal growing conditions for each individual. If not, one can adjust the model as explained in Section A.1.5.

To solve (A.9) we need to start out with a known size x_0 of the population. The b and d parameters depend on the time difference $t_n - t_{n-1}$, i.e., the values of b and d are smaller if n counts years than if n counts generations.

A.1.5 Logistic growth

The model (A.9) for the growth of a population leads to exponential increase in the number of individuals as implied by the solution (A.4). The size of the population increases faster and faster as time n increases, and $x_n \to \infty$ when $n \to \infty$. In real life, however, there is an upper limit M of the number of individuals that can exist in the environment at the same time. Lack of space and food, competition between individuals, predators, and spreading of contagious diseases are examples on factors that limit the growth. The number M is usually called the *carrying capacity* of the environment, the maximum population which is sustainable over time. With limited growth, the growth factor r must depend on time:

$$x_n = x_{n-1} + \frac{r(n-1)}{100}x_{n-1}\,. \tag{A.10}$$

In the beginning of the growth process, there is enough resources and the growth is exponential, but as x_n approaches M, the growth stops and r must tend to zero. A simple function $r(n)$ with these properties is

$$r(n) = \varrho\left(1 - \frac{x_n}{M}\right)\,. \tag{A.11}$$

For small n, $x_n \ll M$ and $r(n) \approx \varrho$, which is the growth rate with unlimited resources. As $n \to M$, $r(n) \to 0$ as we want. The model (A.11) is used for *logistic growth*. The corresponding *logistic difference equation* becomes

$$x_n = x_{n-1} + \frac{\varrho}{100}x_{n-1}\left(1 - \frac{x_{n-1}}{M}\right) . \tag{A.12}$$

Below is a program (growth_logistic.py) for simulating $N = 200$ time intervals in a case where we start with $x_0 = 100$ individuals, a carrying capacity of $M = 500$, and initial growth of $\varrho = 4$ percent in each time interval:

```
from scitools.std import *
x0 = 100              # initial amount of individuals
M = 500               # carrying capacity
rho = 4               # initial growth rate in percent
N = 200               # number of time intervals
index_set = range(N+1)
x = zeros(len(index_set))

# Compute solution
x[0] = x0
for n in index_set[1:]:
    x[n] = x[n-1] + (rho/100.0)*x[n-1]*(1 - x[n-1]/float(M))
print x
plot(index_set, x, 'r', xlabel='time units',
     ylabel='no of individuals', hardcopy='tmp.pdf')
```

Figure A.1 shows how the population stabilizes, i.e., that x_n approaches M as N becomes large (of the same magnitude as M).

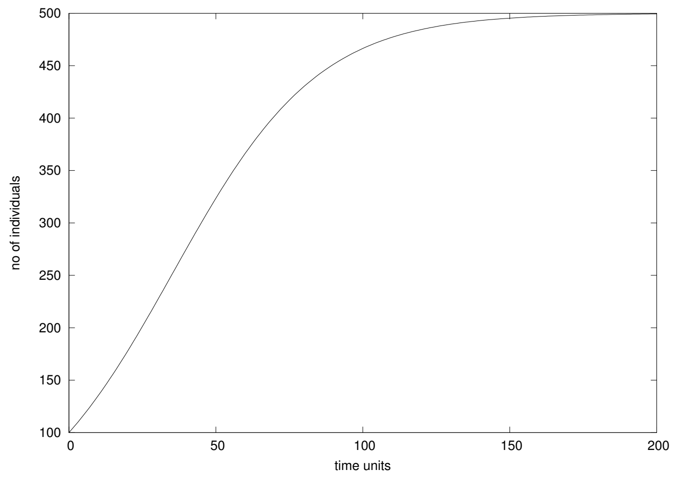


Fig. A.1 Logistic growth of a population ($\varrho = 4$, $M = 500$, $x_0 = 100$, $N = 200$).

If the equation stabilizes as $n \to \infty$, it means that $x_n = x_{n-1}$ in this limit. The equation then reduces to

$$x_n = x_n + \frac{\varrho}{100}x_n\left(1 - \frac{x_n}{M}\right) .$$

By inserting $x_n = M$ we see that this solution fulfills the equation. The same solution technique (i.e., setting $x_n = x_{n-1}$) can be used to check if x_n in a difference equation approaches a limit or not.

Mathematical models like (A.12) are often easier to work with if we *scale* the variables. Basically, this means that we divide each variable by a characteristic size of that variable such that the value of the new variable is typically 1. In the present case we can scale x_n by M and introduce a new variable,

$$y_n = \frac{x_n}{M}\,.$$

Similarly, x_0 is replaced by $y_0 = x_0/M$. Inserting $x_n = My_n$ in (A.12) and dividing by M gives

$$y_n = y_{n-1} + qy_{n-1}\left(1 - y_{n-1}\right), \tag{A.13}$$

where $q = \varrho/100$ is introduced to save typing. Equation (A.13) is simpler than (A.12) in that the solution lies approximately between y_0 and 1 (values larger than 1 can occur, see Exercise A.19), and there are only two dimensionless input parameters to care about: q and y_0. To solve (A.12) we need knowledge of three parameters: x_0, ϱ, and M.

A.1.6 Payback of a loan

A loan L is to be paid back over N months. The payback in a month consists of the fraction L/N plus the interest increase of the loan. Let the annual interest rate for the loan be p percent. The monthly interest rate is then $\frac{p}{12}$. The value of the loan after month n is x_n, and the change from x_{n-1} can be modeled as

$$x_n = x_{n-1} + \frac{p}{12 \cdot 100}x_{n-1} - \left(\frac{p}{12 \cdot 100}x_{n-1} + \frac{L}{N}\right), \tag{A.14}$$

$$= x_{n-1} - \frac{L}{N}, \tag{A.15}$$

for $n = 1, \ldots, N$. The initial condition is $x_0 = L$. A major difference between (A.15) and (A.6) is that all terms in the latter are proportional to x_n or x_{n-1}, while (A.15) also contains a constant term (L/N). We say that (A.6) is homogeneous and linear, while (A.15) is inhomogeneous (because of the constant term) and linear. The mathematical solution of inhomogeneous equations are more difficult to find than the solution of homogeneous equations, but in a program there is no big difference: we just add the extra term $-L/N$ in the formula for the difference equation.

The solution of (A.15) is not particularly exciting (just use (A.15) repeatedly to derive the solution $x_n = L - nL/N$). What is more interesting, is what we pay each month, y_n. We can keep track of both y_n and x_n in a variant of the previous model:

$$y_n = \frac{p}{12\cdot 100}x_{n-1} + \frac{L}{N}, \qquad (A.16)$$

$$x_n = x_{n-1} + \frac{p}{12\cdot 100}x_{n-1} - y_n\,. \qquad (A.17)$$

Equations (A.16)-(A.17) is a system of difference equations. In a computer code, we simply update y_n first, and then we update x_n, inside a loop over n. Exercise A.4 asks you to do this.

A.1.7 The integral as a difference equation

Suppose a function $f(x)$ is defined as the integral

$$f(x) = \int_a^x g(t)dt\,. \qquad (A.18)$$

Our aim is to evaluate $f(x)$ at a set of points $x_0 = a < x_1 < \cdots < x_N$. The value $f(x_n)$ for any $0 \le n \le N$ can be obtained by using the Trapezoidal rule for integration:

$$f(x_n) = \sum_{k=0}^{n-1}\frac{1}{2}(x_{k+1} - x_k)(g(x_k) + g(x_{k+1})), \qquad (A.19)$$

which is nothing but the sum of the areas of the trapezoids up to the point x_n (the plot to the right in Figure 5.12 illustrates the idea.) We realize that $f(x_{n+1})$ is the sum above plus the area of the next trapezoid:

$$f(x_{n+1}) = f(x_n) + \frac{1}{2}(x_{n+1} - x_n)(g(x_n) + g(x_{n+1}))\,. \qquad (A.20)$$

This is a much more efficient formula than using (A.19) with n replaced by $n+1$, since we do not need to recompute the areas of the first n trapezoids.

Formula (A.20) gives the idea of computing all the $f(x_n)$ values through a difference equation. Define f_n as $f(x_n)$ and consider $x_0 = a$, and $x_1, \ldots, x_N$ as given. We know that $f_0 = 0$. Then

$$f_n = f_{n-1} + \frac{1}{2}(x_n - x_{n-1})(g(x_{n-1}) + g(x_n)), \qquad (A.21)$$

for $n = 1, 2, \ldots, N$. By introducing g_n for $g(x_n)$ as an extra variable in the difference equation, we can avoid recomputing $g(x_n)$ when we compute f_{n+1}:

$$g_n = g(x_n), \tag{A.22}$$

$$f_n = f_{n-1} + \frac{1}{2}(x_n - x_{n-1})(g_{n-1} + g_n), \tag{A.23}$$

with initial conditions $f_0 = 0$ and $g_0 = g(a)$.

A function can take g, a, x, and N as input and return arrays x and f for $x_0, \ldots, x_N$ and the corresponding integral values $f_0, \ldots, f_N$:

```
def integral(g, a, x, N=20):
    index_set = range(N+1)
    x = np.linspace(a, x, N+1)
    g_ = np.zeros_like(x)
    f = np.zeros_like(x)
    g_[0] = g(x[0])
    f[0] = 0

    for n in index_set[1:]:
        g_[n] = g(x[n])
        f[n] = f[n-1] + 0.5*(x[n] - x[n-1])*(g_[n-1] + g_[n])
    return x, f
```

Note that g is used for the integrand function to call so we introduce g_ to be the array holding sequence of g(x[n]) values.

Our first task, after having implemented a mathematical calculation, is to verify the result. Here we can make use of the nice fact that the Trapezoidal rule is exact for linear functions $g(t)$:

```
def test_integral():
    def g_test(t):
        """Linear integrand."""
        return 2*t + 1

    def f_test(x, a):
        """Exact integral of g_test."""
        return x**2 + x - (a**2 + a)

    a = 2
    x, f = integral(g_test, a, x=10)
    f_exact = f_test(x, a)
    assert np.allclose(f_exact, f)
```

A realistic application is to apply the integral function to some $g(t)$ where there is no formula for the analytical integral, e.g.,

$$g(t) = \frac{1}{\sqrt{2\pi}} \exp\left(-t^2\right).$$

The code may look like

```
def demo():
    """Integrate the Gaussian function."""
    from numpy import sqrt, pi, exp

    def g(t):
        return 1./sqrt(2*pi)*exp(-t**2)

    x, f = integral(g, a=-3, x=3, N=200)
    integrand = g(x)
    from scitools.std import plot
    plot(x, f, 'r-',
         x, integrand, 'y-',
         legend=('f', 'g'),
         legend_loc='upper left',
         savefig='tmp.pdf')
```

Figure A.2 displays the integrand and the integral. All the code is available in the file `integral.py`.

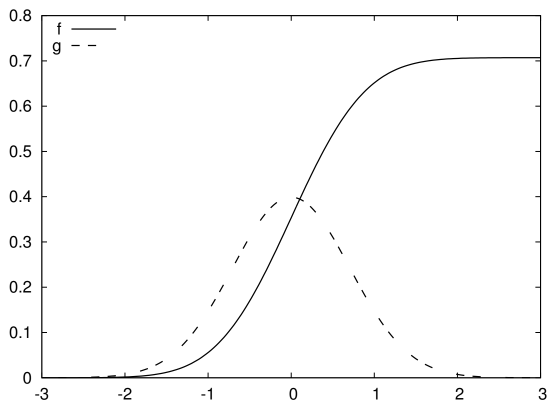


Fig. A.2 Integral of $\frac{1}{\sqrt{2\pi}}\exp\left(-t^2\right)$ from -3 to x.

A.1.8 Taylor series as a difference equation

Consider the following system of two difference equations

$$e_n = e_{n-1} + a_{n-1}, \tag{A.24}$$

$$a_n = \frac{x}{n}a_{n-1}, \tag{A.25}$$

with initial conditions $e_0 = 0$ and $a_0 = 1$. We can start to nest the solution:

$$\begin{aligned}
e_1 &= 0 + a_0 = 0 + 1 = 1,\\
a_1 &= x,\\
e_2 &= e_1 + a_1 = 1 + x,\\
a_2 &= \frac{x}{2}a_1 = \frac{x^2}{2},\\
e_3 &= e_2 + a_2 = 1 + x + \frac{x^2}{2},\\
e_4 &= 1 + x + \frac{x^2}{2} + \frac{x^3}{3\cdot 2},\\
e_5 &= 1 + x + \frac{x^2}{2} + \frac{x^3}{3\cdot 2} + \frac{x^4}{4\cdot 3\cdot 2}
\end{aligned}$$

The observant reader who has heard about Taylor series (see Section B.4) will recognize this as the Taylor series of e^x:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}\,. \tag{A.26}$$

How do we derive a system like (A.24)-(A.25) for computing the Taylor polynomial approximation to e^x? The starting point is the sum $\sum_{n=0}^{\infty} \frac{x^n}{n!}$. This sum is coded by adding new terms to an accumulation variable in a loop. The mathematical counterpart to this code is a difference equation

$$e_{n+1} = e_n + \frac{x^n}{n!}, \quad e_0 = 0,\ n = 0, 1, 2, \ldots\,. \tag{A.27}$$

or equivalently (just replace n by $n-1$):

$$e_n = e_{n-1} + \frac{x^{n-1}}{n-1!}, \quad e_0 = 0,\ n = 1, 2, 3, \ldots\,. \tag{A.28}$$

Now comes the important observation: the term $x^n/n!$ contains many of the computations we already performed for the previous term $x^{n-1}/(n-1)!$ because

$$\frac{x^n}{n!} = \frac{x \cdot x \cdots x}{n(n-1)(n-2)\cdots 1}, \quad \frac{x^{n-1}}{(n-1)!} = \frac{x \cdot x \cdots x}{(n-1)(n-2)(n-3)\cdots 1}\,.$$

Let $a_n = x^n/n!$. We see that we can go from a_{n-1} to a_n by multiplying a_{n-1} by x/n:

$$\frac{x}{n} a_{n-1} = \frac{x}{n} \frac{x^{n-1}}{(n-1)!} = \frac{x^n}{n!} = a_n, \tag{A.29}$$

which is nothing but (A.25). We also realize that $a_0 = 1$ is the initial condition for this difference equation. In other words, (A.24) sums the Taylor polynomial, and (A.25) updates each term in the sum.

The system (A.24)-(A.25) is very easy to implement in a program and constitutes an efficient way to compute (A.26). The function `exp_diffeq` does the work:

```
def exp_diffeq(x, N):
    n = 1
    an_prev = 1.0  # a_0
    en_prev = 0.0  # e_0
    while n <= N:
        en = en_prev + an_prev
        an = x/n*an_prev
        en_prev = en
        an_prev = an
        n += 1
    return en
```

Observe that we do not store the sequences in arrays, but make use of the fact that only the most recent sequence element is needed to calculate a new element. The above function along with a direct evaluation of the Taylor series for e^x and a comparison with the exact result for various N values can be found in the file `exp_Taylor_series_diffeq.py`.

A.1.9 Making a living from a fortune

Suppose you want to live on a fortune F. You have invested the money in a safe way that gives an annual interest of p percent. Every year you plan to consume an amount c_n, where n counts years. The development of your fortune x_n from one year to the other can then be modeled by

$$x_n = x_{n-1} + \frac{p}{100}x_{n-1} - c_{n-1}, \quad x_0 = F\,. \tag{A.30}$$

A simple example is to keep c constant, say q percent of the interest the first year:

$$x_n = x_{n-1} + \frac{p}{100}x_{n-1} - \frac{pq}{10^4}F, \quad x_0 = F\,. \tag{A.31}$$

A more realistic model is to assume some inflation of I percent per year. You will then like to increase c_n by the inflation. We can extend the model in two ways. The simplest and clearest way, in the author's opinion, is to track the evolution of two sequences x_n and c_n:

$$x_n = x_{n-1} + \frac{p}{100}x_{n-1} - c_{n-1}, \quad x_0 = F,\ c_0 = \frac{pq}{10^4}F, \tag{A.32}$$

$$c_n = c_{n-1} + \frac{I}{100}c_{n-1}\,. \tag{A.33}$$

This is a system of two difference equations with two unknowns. The solution method is, nevertheless, not much more complicated than the method for a difference equation in one unknown, since we can first compute x_n from (A.32) and then update the c_n value from (A.33). You are encouraged to write the program (see Exercise A.5).

Another way of making a difference equation for the case with inflation, is to use an explicit formula for c_{n-1}, i.e., solve (A.32) and end up with a formula like (A.4). Then we can insert the explicit formula

$$c_{n-1} = \left(1 + \frac{I}{100}\right)^{n-1}\frac{pq}{10^4}F$$

in (A.30), resulting in only one difference equation to solve.

A.1.10 Newton's method

The difference equation

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, \quad x_0 \text{ given}, \tag{A.34}$$

generates a sequence x_n where, if the sequence converges (i.e., if $x_n - x_{n-1} \rightarrow 0$), x_n approaches a root of $f(x)$. That is, $x_n \rightarrow x$, where x solves the equation $f(x) = 0$. Equation (A.34) is the famous Newton's method for solving nonlinear algebraic equations $f(x) = 0$. When $f(x)$ is not linear, i.e., $f(x)$ is not on the form $ax + b$ with constant a and b, (A.34) becomes a *nonlinear difference equation*. This complicates analytical treatment of difference equations, but poses no extra difficulties for numerical solution.

We can quickly sketch the derivation of (A.34). Suppose we want to solve the equation

$$f(x) = 0$$

and that we already have an approximate solution x_{n-1}. If $f(x)$ were linear, $f(x) = ax + b$, it would be very easy to solve $f(x) = 0$: $x = -b/a$. The idea is therefore to approximate $f(x)$ in the vicinity of $x = x_{n-1}$ by a linear function, i.e., a straight line $f(x) \approx \tilde{f}(x) = ax + b$. This line should have the same slope as $f(x)$, i.e., $a = f'(x_{n-1})$, and both the line and f should have the same value at $x = x_{n-1}$. From this condition one can find $b = f(x_{n-1}) - x_{n-1}f'(x_{n-1})$. The approximate function (line) is then

$$\tilde{f}(x) = f(x_{n-1}) + f'(x_{n-1})(x - x_{n-1})\,. \tag{A.35}$$

This expression is just the two first terms of a Taylor series approximation to $f(x)$ at $x = x_{n-1}$. It is now easy to solve $\tilde{f}(x) = 0$ with respect to x, and we get

$$x = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}\,. \tag{A.36}$$

Since $\tilde{f}$ is only an approximation to f, x in (A.36) is only an approximation to a root of $f(x) = 0$. Hopefully, the approximation is better than x_{n-1} so we set $x_n = x$ as the next term in a sequence that we hope converges to the correct root. However, convergence depends highly on the shape of $f(x)$, and there is no guarantee that the method will work.

The previous programs for solving difference equations have typically calculated a sequence x_n up to $n = N$, where N is given. When using (A.34) to find roots of nonlinear equations, we do not know a suitable N in advance that leads to an x_n where $f(x_n)$ is sufficiently close to zero. We therefore have to keep on increasing n until $f(x_n) < \epsilon$ for some small ϵ. Of course, the sequence diverges, we will keep on forever, so there must be some maximum allowable limit on n, which we may take as N.

It can be convenient to have the solution of (A.34) as a function for easy reuse. Here is a first rough implementation:

```
def Newton(f, x, dfdx, epsilon=1.0E-7, N=100):
    n = 0
    while abs(f(x)) > epsilon and n <= N:
        x = x - f(x)/dfdx(x)
        n += 1
    return x, n, f(x)
```

This function might well work, but `f(x)/dfdx(x)` can imply integer division, so we should ensure that the numerator or denumerator is of `float` type. There are also two function evaluations of `f(x)` in every pass in the loop (one in the loop body and one in the `while` condition). We can get away with only one evaluation if we store the `f(x)` in a local variable. In the small examples with $f(x)$ in the present course, twice as many function evaluations of f as necessary does not matter, but the same `Newton` function can in fact be used for much more complicated functions, and in those cases twice as much work can be noticeable. As a programmer, you should therefore learn to optimize the code by removing unnecessary computations.

Another, more serious, problem is the possibility dividing by zero. Almost as serious, is dividing by a very small number that creates a large value, which might cause Newton's method to diverge. Therefore, we should test for small values of $f'(x)$ and write a warning or raise an exception.

Another improvement is to add a boolean argument `store` to indicate whether we want the $(x, f(x))$ values during the iterations to be stored in a list or not. These intermediate values can be handy if we want to print out or plot the convergence behavior of Newton's method.

An improved `Newton` function can now be coded as

```
def Newton(f, x, dfdx, epsilon=1.0E-7, N=100, store=False):
    f_value = f(x)
    n = 0
    if store: info = [(x, f_value)]
    while abs(f_value) > epsilon and n <= N:
        dfdx_value = float(dfdx(x))
        if abs(dfdx_value) < 1E-14:
            raise ValueError("Newton: f'(%g)=%g" % (x, dfdx_value))

        x = x - f_value/dfdx_value

        n += 1
        f_value = f(x)
        if store: info.append((x, f_value))
    if store:
        return x, info
    else:
        return x, n, f_value
```

Note that to use the `Newton` function, we need to calculate the derivative $f'(x)$ and implement it as a Python function and provide it as the `dfdx` argument. Also note that what we return depends on whether we store $(x, f(x))$ information during the iterations or not.

It is quite common to test if `dfdx(x)` is zero in an implementation of Newton's method, but this is not strictly necessary in Python since an exception `ZeroDivisionError` is always raised when dividing by zero.

We can apply the `Newton` function to solve the equation $e^{-0.1x^2}\sin(\frac{\pi}{2}x) = 0$:

```
from math import sin, cos, exp, pi
import sys
from Newton import Newton
```

```
def g(x):
    return exp(-0.1*x**2)*sin(pi/2*x)

def dg(x):
    return -2*0.1*x*exp(-0.1*x**2)*sin(pi/2*x) + \
           pi/2*exp(-0.1*x**2)*cos(pi/2*x)

x0 = float(sys.argv[1])
x, info = Newton(g, x0, dg, store=True)
print 'root:', x
for i in range(len(info)):
    print 'Iteration %3d: f(%g)=%g' % \
          (i, info[i][0], info[i][1])
```

The `Newton` function and this program can be found in the file `Newton.py`. Running this program with an initial x value of 1.7 results in the output

```
root: 1.999999999768449
Iteration  0: f(1.7)=0.340044
Iteration  1: f(1.99215)=0.00828786
Iteration  2: f(1.99998)=2.53347e-05
Iteration  3: f(2)=2.43808e-10
```

Fortunately you realize that the exponential function can never be zero, so the solutions of the equation must be the zeros of the sine function, i.e., $\frac{\pi}{2}x = i\pi$ for all integers $i = \ldots, -2, 1, 0, 1, 2, \ldots$. This gives $x = 2i$ as the solutions. We see from the output that the convergence is fast towards the solution $x = 2$. The error is of the order 10^{-10} even though we stop the iterations when $f(x) \leq 10^{-7}$.

Trying a start value of 3, we would expect the method to find the root $x = 2$ or $x = 4$, but now we get

```
root: 42.49723316011362
Iteration  0: f(3)=-0.40657
Iteration  1: f(4.66667)=0.0981146
Iteration  2: f(42.4972)=-2.59037e-79
```

We have definitely solved $f(x) = 0$ in the sense that $|f(x)| \leq \epsilon$, where ϵ is a small value (here $\epsilon \sim 10^{-79}$). However, the solution $x \approx 42.5$ is *not* close to the correct solution ($x = 42$ and $x = 44$ are the solutions closest to the computed x). Can you use your knowledge of how the Newton method works and figure out why we get such strange behavior?

The demo program `Newton_movie.py` can be used to investigate the strange behavior. This program takes five command-line arguments: a formula for $f(x)$, a formula for $f'(x)$ (or the word `numeric`, which indicates a numerical approximation of $f'(x)$), a guess at the root, and the minimum and maximum x values in the plots. We try the following case with the program:

Terminal

```
Newton_movie.py 'exp(-0.1*x**2)*sin(pi/2*x)' numeric 3 -3 43
```

As seen, we start with $x = 3$ as the initial guess. In the first step of the method, we compute a new value of the root, now $x = 4.66667$. As we see in Figure A.3, this root is near an extreme point of $f(x)$ so that the derivative is small, and the resulting straight line approximation to $f(x)$ at this root becomes quite flat. The result is a new guess at the root: $x42.5$. This root is far away from the last root, but the second problem is that $f(x)$ is quickly damped as we move to increasing x values, and at $x = 42.5$ f is small enough to fulfill the convergence criterion. Any guess at the root out in this region would satisfy that criterion.

You can run the `Newton_movie.py` program with other values of the initial root and observe that the method usually finds the nearest roots.

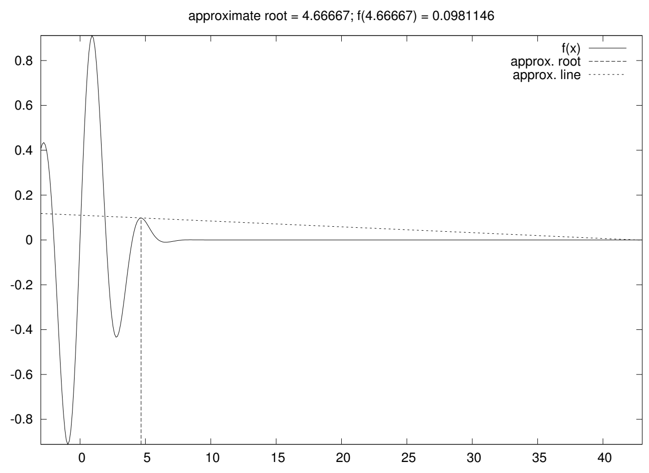


Fig. A.3 Failure of Newton's method to solve $e^{-0.1x^2}\sin(\frac{\pi}{2}x) = 0$. The plot corresponds to the second root found (starting with $x = 3$).

A.1.11 The inverse of a function

Given a function $f(x)$, the inverse function of f, say we call it $g(x)$, has the property that if we apply g to the value $f(x)$, we get x back:

$$g(f(x)) = x\,.$$

Similarly, if we apply f to the value $g(x)$, we get x:

$$f(g(x)) = x\,. \tag{A.37}$$

By hand, you substitute $g(x)$ by (say) y in (A.37) and solve (A.37) with respect to y to find some x expression for the inverse function. For example, given $f(x) = x^2 - 1$, we must solve $y^2 - 1 = x$ with respect to y. To ensure a unique solution for y, the x values have to be limited to an interval where $f(x)$ is monotone, say $x \in [0, 1]$ in the present example. Solving for y gives $y = \sqrt{1 + x}$, therefore $g(x) = \sqrt{1 + x}$. It is easy to check that $f(g(x)) = (\sqrt{1 + x})^2 - 1 = x$.

Numerically, we can use the "definition" (A.37) of the inverse function g at one point at a time. Suppose we have a sequence of points $x_0 < x_1 < \cdots < x_N$ along the x axis such that f is monotone in $[x_0, x_N]$: $f(x_0) > f(x_1) > \cdots > f(x_N)$ or $f(x_0) < f(x_1) < \cdots < f(x_N)$. For each point x_i, we have

$$f(g(x_i)) = x_i\,.$$

The value $g(x_i)$ is unknown, so let us call it γ. The equation

$$f(\gamma) = x_i \tag{A.38}$$

can be solved be respect γ. However, (A.38) is in general nonlinear if f is a nonlinear function of x. We must then use, e.g., Newton's method to solve (A.38). Newton's method works for an equation phrased as $f(x) = 0$, which in our case is $f(\gamma) - x_i = 0$, i.e., we seek the roots of the function $F(\gamma) \equiv f(\gamma) - x_i$. Also the derivative $F'(\gamma)$ is needed in Newton's method. For simplicity we may use an approximate finite difference:

$$\frac{dF}{d\gamma} \approx \frac{F(\gamma + h) - F(\gamma - h)}{2h}\,.$$

As start value γ_0, we can use the previously computed g value: g_{i-1}. We introduce the short notation $\gamma = \text{Newton}(F, \gamma_0)$ to indicate the solution of $F(\gamma) = 0$ with initial guess γ_0.

The computation of all the $g_0, \ldots, g_N$ values can now be expressed by

$$g_i = \text{Newton}(F, g_{i-1}), \quad i = 1, \ldots, N, \tag{A.39}$$

and for the first point we may use x_0 as start value (for instance):

$$g_0 = \text{Newton}(F, x_0)\,. \tag{A.40}$$

Equations (A.39)-(A.40) constitute a difference equation for g_i, since given g_{i-1}, we can compute the next element of the sequence by (A.39). Because (A.39) is a nonlinear equation in the new value g_i, and (A.39) is therefore an example of a *nonlinear difference equation.*

The following program computes the inverse function $g(x)$ of $f(x)$ at some discrete points $x_0, \ldots, x_N$. Our sample function is $f(x) = x^2 - 1$:

```
from Newton import Newton
from scitools.std import *

def f(x):
    return x**2 - 1

def F(gamma):
    return f(gamma) - xi

def dFdx(gamma):
    return (F(gamma+h) - F(gamma-h))/(2*h)
```

```
h = 1E-6
x = linspace(0.01, 3, 21)
g = zeros(len(x))

for i in range(len(x)):
    xi = x[i]

    # Compute start value (use last g[i-1] if possible)
    if i == 0:
        gamma0 = x[0]
    else:
        gamma0 = g[i-1]

    gamma, n, F_value = Newton(F, gamma0, dFdx)
    g[i] = gamma

plot(x, f(x), 'r-', x, g, 'b-',
     title='f1', legend=('original', 'inverse'))
```

Note that with $f(x) = x^2 - 1$, $f'(0) = 0$, so Newton's method divides by zero and breaks down unless with let $x_0 > 0$, so here we set $x_0 = 0.01$. The `f` function can easily be edited to let the program compute the inverse of another function. The `F` function can remain the same since it applies a general finite difference to approximate the derivative of the `f(x)` function. The complete program is found in the file `inverse_function.py`.

A.2 Programming with sound

Sound on a computer is nothing but a sequence of numbers. As an example, consider the famous A tone at 440 Hz. Physically, this is an oscillation of a tuning fork, loudspeaker, string or another mechanical medium that makes the surrounding air also oscillate and transport the sound as a compression wave. This wave may hit our ears and through complicated physiological processes be transformed to an electrical signal that the brain can recognize as sound. Mathematically, the oscillations are described by a sine function of time:

$$s(t) = A\sin(2\pi f t)\,, \tag{A.41}$$

where A is the amplitude or strength of the sound and f is the frequency (440 Hz for the A in our example). In a computer, $s(t)$ is represented at discrete points of time. CD quality means 44100 samples per second. Other sample rates are also possible, so we introduce r as the sample rate. An f Hz tone lasting for m seconds with sample rate r can then be computed as the sequence

$$s_n = A\sin\left(2\pi f\frac{n}{r}\right), \quad n = 0, 1, \ldots, m\cdot r\,. \tag{A.42}$$

With Numerical Python this computation is straightforward and very efficient. Introducing some more explanatory variable names than r, A, and m, we can write a function for generating a note:

```
import numpy as np

def note(frequency, length, amplitude=1, sample_rate=44100):
    time_points = np.linspace(0, length, length*sample_rate)
    data = np.sin(2*np.pi*frequency*time_points)
    data = amplitude*data
    return data
```

A.2.1 Writing sound to file

The `note` function above generates an array of `float` data representing a note. The sound card in the computer cannot play these data, because the card assumes that the information about the oscillations appears as a sequence of two-byte integers. With an array's `astype` method we can easily convert our data to two-byte integers instead of `floats`:

```
data = data.astype(numpy.int16)
```

That is, the name of the two-byte integer data type in `numpy` is `int16` (two bytes are 16 bits). The maximum value of a two-byte integer is $2^{15}-1$, so this is also the maximum amplitude. Assuming that `amplitude` in the `note` function is a relative measure of intensity, such that the value lies between 0 and 1, we must adjust this amplitude to the scale of two-byte integers:

```
max_amplitude = 2**15 - 1
data = max_amplitude*data
```

The `data` array of `int16` numbers can be written to a file and played as an ordinary file in CD quality. Such a file is known as a wave file or simply a WAV file since the extension is `.wav`. Python has a module `wave` for creating such files. Given an array of sound, `data`, we have in SciTools a module `sound` with a function `write` for writing the data to a WAV file (using functionality from the `wave` module):

```
import scitools.sound
scitools.sound.write(data, 'Atone.wav')
```

You can now use your favorite music player to play the `Atone.wav` file, or you can play it from within a Python program using

```
scitools.sound.play('Atone.wav')
```

The `write` function can take more arguments and write, e.g., a stereo file with two channels, but we do not dive into these details here.

A.2.2 Reading sound from file

Given a sound signal in a WAV file, we can easily read this signal into an array and mathematically manipulate the data in the array to change the flavor of the sound, e.g., add echo, treble, or bass. The recipe for reading a WAV file with name `filename` is

```
data = scitools.sound.read(filename)
```

The `data` array has elements of type `int16`. Often we want to compute with this array, and then we need elements of `float` type, obtained by the conversion

```
data = data.astype(float)
```

The `write` function automatically transforms the element type back to `int16` if we have not done this explicitly.

One operation that we can easily do is adding an echo. Mathematically this means that we add a damped delayed sound, where the original sound has weight β and the delayed part has weight $1-\beta$, such that the overall amplitude is not altered. Let d be the delay in seconds. With a sampling rate r the number of indices in the delay becomes dr, which we denote by b. Given an original sound sequence s_n, the sound with echo is the sequence

$$e_n = \beta s_n + (1-\beta)s_{n-b}\,. \tag{A.43}$$

We cannot start n at 0 since $e_0 = s_{0-b} = s_{-b}$ which is a value outside the sound data. Therefore we define $e_n = s_n$ for $n = 0, 1, \ldots, b$, and add the echo thereafter. A simple loop can do this (again we use descriptive variable names instead of the mathematical symbols introduced):

```
def add_echo(data, beta=0.8, delay=0.002, sample_rate=44100):
    newdata = data.copy()
    shift = int(delay*sample_rate)  # b (math symbol)
    for i in range(shift, len(data)):
        newdata[i] = beta*data[i] + (1-beta)*data[i-shift]
    return newdata
```

The problem with this function is that it runs slowly, especially when we have sound clips lasting several seconds (recall that for CD quality we need 44100 numbers per second). It is therefore necessary to vectorize the implementation of the difference equation for adding echo. The update is then based on adding slices:

```
newdata[shift:] = beta*data[shift:] + \
                  (1-beta)*data[:len(data)-shift]
```

A.2.3 Playing many notes

How do we generate a melody mathematically in a computer program? With the `note` function we can generate a note with a certain amplitude, frequency, and duration. The note is represented as an array. Putting sound arrays for different notes after each other will make up a melody. If we have several sound arrays `data1`, `data2`, `data3`, ..., we can make a new array consisting of the elements in the first array followed by the elements of the next array followed by the elements in the next array and so forth:

```
data = numpy.concatenate((data1, data2, data3, ...))
```

The frequency of a note[4] that is h half tones up from a base frequency f is given by $f2^{h/12}$. With the tone A at 440 Hz, we can define notes and the corresponding frequencies as

```
base_freq = 440.0
notes = ['A', 'A#', 'B', 'C', 'C#', 'D', 'D#', 'E',
         'F', 'F#', 'G', 'G#']
notes2freq = {notes[i]: base_freq*2**(i/12.0)
              for i in range(len(notes))}
```

With the notes to frequency mapping a melody can be made as a series of notes with specified duration:

```
l = .2  # basic duration unit
tones = [('E', 3*l), ('D', l), ('C#', 2*l), ('B', 2*l), ('A', 2*l),
         ('B', 2*l), ('C#', 2*l), ('D', 2*l), ('E', 3*l),
         ('F#', l), ('E', 2*l), ('D', 2*l), ('C#', 4*l)]

samples = []
for tone, duration in tones :
    s = note(notes2freq[tone], duration)
    samples.append(s)

data = np.concatenate(samples)
data *= 2**15-1
scitools.sound.write(data, "melody.wav")
```

Playing the resulting file `melody.wav` reveals that this is the opening of the most-played tune during international cross country skiing competitions.

All the notes had the same amplitude in this example, but more dynamics can easily be added by letting the elements in `tones` be triplets with tone, duration, and amplitude. The basic code above is found in the file `melody.py`.

[4] `http://en.wikipedia.org/wiki/Note`

A.2.4 Music of a sequence

Problem. The purpose of this example is to listen to the sound generated by two mathematical sequences. The first one is given by an explicit formula, constructed to oscillate around 0 with decreasing amplitude:

$$x_n = e^{-4n/N}\sin(8\pi n/N)\,. \tag{A.44}$$

The other sequence is generated by the difference equation (A.13) for logistic growth, repeated here for convenience:

$$x_n = x_{n-1} + qx_{n-1}\left(1 - x_{n-1}\right), \quad x = x_0\,. \tag{A.45}$$

We let $x_0 = 0.01$ and $q = 2$. This leads to fast initial growth toward the limit 1, and then oscillations around this limit (this problem is studied in Exercise A.19).

The absolute value of the sequence elements x_n are of size between 0 and 1, approximately. We want to transform these sequence elements to tones, using the techniques of Section A.2. First we convert x_n to a frequency the human ear can hear. The transformation

$$y_n = 440 + 200x_n \tag{A.46}$$

will make a standard A reference tone out of $x_n = 0$, and for the maximum value of x_n around 1 we get a tone of 640 Hz. Elements of the sequence generated by (A.44) lie between -1 and 1, so the corresponding frequencies lie between 240 Hz and 640 Hz. The task now is to make a program that can generate and play the sounds.

Solution. Tones can be generated by the `note` function from the `scitools.sound` module. We collect all tones corresponding to all the y_n frequencies in a list `tones`. Letting `N` denote the number of sequence elements, the relevant code segment reads

```
from scitools.sound import *
freqs = 440 + x*200
tones = []
duration = 30.0/N     # 30 sec sound in total
for n in range(N+1):
    tones.append(max_amplitude*note(freqs[n], duration, 1))
data = concatenate(tones)
write(data, filename)
data = read(filename)
play(filename)
```

It is illustrating to plot the sequences too,

```
plot(range(N+1), freqs, 'ro')
```

To generate the sequences (A.44) and (A.45), we make two functions, `oscillations` and `logistic`, respectively. These functions take the number of sequence elements (`N`) as input and return the sequence stored in an array.

In another function make_sound we compute the sequence, transform the elements to frequencies, generate tones, write the tones to file, and play the sound file.

As always, we collect the functions in a module and include a test block where we can read the choice of sequence and the sequence length from the command line. The complete module file looks as follows:

```
from scitools.sound import *
from scitools.std import *

def oscillations(N):
    x = zeros(N+1)
    for n in range(N+1):
        x[n] = exp(-4*n/float(N))*sin(8*pi*n/float(N))
    return x

def logistic(N):
    x = zeros(N+1)
    x[0] = 0.01
    q = 2
    for n in range(1, N+1):
        x[n] = x[n-1] + q*x[n-1]*(1 - x[n-1])
    return x

def make_sound(N, seqtype):
    filename = 'tmp.wav'
    x = eval(seqtype)(N)
    # Convert x values to frequences around 440
    freqs = 440 + x*200
    plot(range(N+1), freqs, 'ro')
    # Generate tones
    tones = []
    duration = 30.0/N     # 30 sec sound in total
    for n in range(N+1):
        tones.append(max_amplitude*note(freqs[n], duration, 1))
    data = concatenate(tones)
    write(data, filename)
    data = read(filename)
    play(filename)

if __name__ == '__main__':
    try:
        seqtype = sys.argv[1]
        N = int(sys.argv[2])
    except IndexError:
        print 'Usage: %s oscillations|logistic N' % sys.argv[0]
        sys.exit(1)
    make_sound(N, seqtype)
```

This code should be quite easy to read at the present stage in the book. However, there is one statement that deserves a comment:

```
x = eval(seqtype)(N)
```

The seqtype argument reflects the type of sequence and is a string that the user provides on the command line. The values of the string equal the function names oscillations and logistic. With eval(seqtype) we turn the string into a function name. For example, if seqtype is 'logistic', performing an eval(seqtype)(N) is the same as if we had written logistic(N). This technique allows the user of the program to choose a function call inside the code. Without eval we would need to explicitly test on values:

```
if seqtype == 'logistic':
    x = logistic(N)
elif seqtype == 'oscillations':
    x = oscillations(N)
```

This is not much extra code to write in the present example, but if we have a large number of functions generating sequences, we can save a lot of boring if-else code by using the `eval` construction.

The next step, as a reader who have understood the problem and the implementation above, is to run the program for two cases: the `oscillations` sequence with $N = 40$ and the `logistic` sequence with $N = 100$. By altering the q parameter to lower values, you get other sounds, typically quite boring sounds for non-oscillating logistic growth ($q < 1$). You can also experiment with other transformations of the form (A.46), e.g., increasing the frequency variation from 200 to 400.

A.3 Exercises

Exercise A.1: Determine the limit of a sequence

a) Write a Python function for computing and returning the sequence

$$a_n = \frac{7 + 1/(n+1)}{3 - 1/(n+1)^2}, \quad n = 0, 2, \ldots, N\,.$$

Write out the sequence for $N = 100$. Find the exact limit as $N \to \infty$ and compare with a_N.

b) Write a Python function `limit(seq)` that takes a sequence of numbers as input, stored in a list or array `seq`, and returns the limit of the sequence, if it exists, otherwise `None` is returned. Test the `limit` function on the sequence in a) and on the divergent sequence $b_n = n$.

Hint. One possible quite strict test for determining if a sequence $(a_n)_{n=0}^N$ has a limit is to check

$$|a_n| - |a_{n+1}| < |a_{n-1}| - |a_n|,$$

for $n = 1, \ldots, N - 1$.

c) Write a Python function for computing and returning the sequence

$$D_n = \frac{\sin(2^{-n})}{2^{-n}}, \quad n = 0, \ldots, N\,.$$

Call `limit` from b) to determine the limit of the sequence (for a sufficiently large N).

d) Given the sequence

$$D_n = \frac{f(x+h) - f(x)}{h}, \quad h = 2^{-n}, \tag{A.47}$$

make a function `D(f, x, N)` that takes a function $f(x)$, a value x, and the number N of terms in the sequence as arguments, and returns the sequence D_n for $n = 0, 1, \ldots, N$. Make a call to the `D` function with $f(x) = \sin x$, $x = 0$, and $N = 80$. Find the limit with aid of the `limit` function above. Plot the evolution of the computed D_n values, using small circles for the data points.

e) Make another call to `D` where $x = \pi$, let the `limit` function analyze the sequence, and plot this sequence in a separate figure. What would be your expected limit?

f) Explain why the computations for $x = \pi$ go wrong for large N.

Hint. Print out the numerator and denominator in D_n.
Filename: `sequence_limits.py`.

Exercise A.2: Compute π via sequences

The following sequences all converge to π:

$$\begin{aligned}
(a_n)_{n=1}^{\infty}, &\quad a_n = 4\sum_{k=1}^{n} \frac{(-1)^{k+1}}{2k-1},\\
(b_n)_{n=1}^{\infty}, &\quad b_n = \left(6\sum_{k=1}^{n} k^{-2}\right)^{1/2},\\
(c_n)_{n=1}^{\infty}, &\quad c_n = \left(90\sum_{k=1}^{n} k^{-4}\right)^{1/4},\\
(d_n)_{n=1}^{\infty}, &\quad d_n = \frac{6}{\sqrt{3}}\sum_{k=0}^{n} \frac{(-1)^k}{3^k(2k+1)},\\
(e_n)_{n=1}^{\infty}, &\quad e_n = 16\sum_{k=0}^{n} \frac{(-1)^k}{5^{2k+1}(2k+1)} - 4\sum_{k=0}^{n} \frac{(-1)^k}{239^{2k+1}(2k+1)}.
\end{aligned}$$

Make a function for each sequence that returns an array with the elements in the sequence. Plot all the sequences, and find the one that converges fastest toward the limit π. Filename: `pi_sequences.py`.

Exercise A.3: Reduce memory usage of difference equations

Consider the program `growth_years.py` from Section A.1.1. Since x_n depends on x_{n-1} only, we do not need to store all the $N+1$ x_n values. We actually only need to store x_n and its previous value x_{n-1}. Modify the program to use two variables and not an array for the entire sequence. Also avoid the `index_set` list and use an integer counter for n and a `while` loop instead. Write the sequence to file such that it can be visualized later. Filename: `growth_years_efficient.py`.

Exercise A.4: Compute the development of a loan

Solve (A.16)-(A.17) in a Python function. Filename: `loan.py`.

Exercise A.5: Solve a system of difference equations

Solve (A.32)-(A.33) in a Python function and plot the x_n sequence. Filename: `fortune_and_inflation1.py`.

Exercise A.6: Modify a model for fortune development

In the model (A.32)-(A.33) the new fortune is the old one, plus the interest, minus the consumption. During year n, x_n is normally also reduced with t percent tax on the earnings $x_{n-1} - x_{n-2}$ in year $n - 1$.

a) Extend the model with an appropriate tax term, implement the model, and demonstrate in a plot the effect of tax ($t = 27$) versus no tax ($t = 0$).

b) Suppose you expect to live for N years and can accept that the fortune x_n vanishes after N years. Choose some appropriate values for p, q, I, and t, and experiment with the program to find how large the initial c_0 can be in this case.
Filename: `fortune_and_inflation2.py`.

Exercise A.7: Change index in a difference equation

A mathematically equivalent equation to (A.5) is

$$x_{i+1} = x_i + \frac{p}{100}x_i, \qquad \text{(A.48)}$$

since the name of the index can be chosen arbitrarily. Suppose someone has made the following program for solving (A.48):

```
from scitools.std import *
x0 = 100              # initial amount
p = 5                 # interest rate
N = 4                 # number of years
index_set = range(N+1)
x = zeros(len(index_set))

# Compute solution
```

```
x[0] = x0
for i in index_set[1:]:
    x[i+1] = x[i] + (p/100.0)*x[i]
print x
plot(index_set, x, 'ro', xlabel='years', ylabel='amount')
```

This program does not work. Make a correct version, but keep the difference equations in its present form with the indices `i+1` and `i`. Filename: `growth1_index_ip1.py`.

Exercise A.8: Construct time points from dates

A certain quantity p (which may be an interest rate) is piecewise constant and undergoes changes at some specific dates, e.g.,

$$p \text{ changes to } \begin{cases} 4.5 & \text{on Jan 4, 2019} \\ 4.75 & \text{on March 21, 2019} \\ 6.0 & \text{on April 1, 2019} \\ 5.0 & \text{on June 30, 2019} \\ 4.5 & \text{on Nov 1, 2019} \\ 2.0 & \text{on April 1, 2020} \end{cases} \tag{A.49}$$

Given a start date d_1 and an end date d_2, fill an array `p` with the right p values, where the array index counts days. Use the `datetime` module to compute the number of days between dates. Filename: `dates2days.py`.

Exercise A.9: Visualize the convergence of Newton's method

Let $x_0, x_1, \ldots, x_N$ be the sequence of roots generated by Newton's method applied to a nonlinear algebraic equation $f(x) = 0$ (see Section A.1.10). In this exercise, the purpose is to plot the sequences $(x_n)_{n=0}^N$ and $(|f(x_n)|)_{n=0}^N$ such that we can understand how Newton's method converges or diverges.

a) Make a general function

```
Newton_plot(f, x, dfdx, xmin, xmax, epsilon=1E-7)
```

for this purpose. The arguments `f` and `dfdx` are Python functions representing the $f(x)$ function in the equation and its derivative $f'(x)$, respectively. Newton's method is run until $|f(x_N)| \leq \epsilon$, and the ϵ value is available as the `epsilon` argument. The `Newton_plot` function should make three separate plots of $f(x)$, $(x_n)_{n=0}^N$, and $(|f(x_n)|)_{n=0}^N$ on the screen and also save these plots to PNG files. The relevant x interval for plotting of $f(x)$ is given by the arguments `xmin` and `xmax`. Because of the potentially wide scale of values that $|f(x_n)|$ may exhibit, it may be wise to use a logarithmic scale on the y axis.

Hint. You can save quite some coding by calling the improved `Newton` function from Section A.1.10, which is available in the module file `Newton.py`.

b) Demonstrate the function on the equation $x^6 \sin \pi x = 0$, with $\epsilon = 10^{-13}$. Try different starting values for Newton's method: $x_0 = -2.6, -1.2, 1.5, 1.7, 0.6$. Compare the results with the exact solutions $x = \ldots, -2 - 1, 0, 1, 2, \ldots$.

c) Use the `Newton_plot` function to explore the impact of the starting point x_0 when solving the following nonlinear algebraic equations:

$$\sin x = 0, \tag{A.50}$$
$$x = \sin x, \tag{A.51}$$
$$x^5 = \sin x, \tag{A.52}$$
$$x^4 \sin x = 0, \tag{A.53}$$
$$x^4 = 16, \tag{A.54}$$
$$x^{10} = 1, \tag{A.55}$$
$$\tanh x = 0 \,.\, \tanh x \qquad = x^{10} \,. \tag{A.56}$$

Hint. Such an experimental investigation is conveniently recorded in an IPython notebook. See Section H.1.9 for a quick introduction to notebooks.
Filename: `Newton2.py`.

Exercise A.10: Implement the secant method

Newton's method (A.34) for solving $f(x) = 0$ requires the derivative of the function $f(x)$. Sometimes this is difficult or inconvenient. The derivative can be approximated using the last two approximations to the root, x_{n-2} and x_{n-1}:

$$f'(x_{n-1}) \approx \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}} \,.$$

Using this approximation in (A.34) leads to the Secant method:

$$x_n = x_{n-1} - \frac{f(x_{n-1})(x_{n-1} - x_{n-2})}{f(x_{n-1}) - f(x_{n-2})}, \quad x_0, x_1 \text{ given} \,. \tag{A.57}$$

Here $n = 2, 3, \ldots$. Make a program that applies the Secant method to solve $x^5 = \sin x$. Filename: `Secant.py`.

Exercise A.11: Test different methods for root finding

Make a program for solving $f(x) = 0$ by Newton's method (Section A.1.10), the Bisection method (Section 4.10.2), and the Secant method (Exercise A.10). For each method, the sequence of root approximations should be written out (nicely formatted) on the screen. Read $f(x)$, $f'(x)$, a, b, x_0, and x_1 from the command line. Newton's method starts with x_0, the Bisection method starts with the interval $[a,b]$, whereas the Secant method starts with x_0 and x_1.

Run the program for each of the equations listed in Exercise A.9d. You should first plot the $f(x)$ functions so you know how to choose x_0, x_1, a, and b in each case. Filename: `root_finder_examples.py`.

Exercise A.12: Make difference equations for the Midpoint rule

Use the ideas of Section A.1.7 to make a similar system of difference equations and corresponding implementation for the Midpoint integration rule:

$$\int_a^b f(x)dx \approx h\sum_{i=0}^{n-1} f(a - \frac{1}{2}h + ih),$$

where $h = (b-a)/n$ and n counts the number of function evaluations (i.e., rectangles that approximate the area under the curve). Filename: `diffeq_midpoint.py`.

Exercise A.13: Compute the arc length of a curve

Sometimes one wants to measure the length of a curve $y = f(x)$ for $x \in [a,b]$. The arc length from $f(a)$ to some point $f(x)$ is denoted by $s(x)$ and defined through an integral

$$s(x) = \int_a^x \sqrt{1 + [f'(\xi)]^2} d\xi\,. \tag{A.58}$$

We can compute $s(x)$ via difference equations as explained in Section A.1.7.

a) Make a Python function `arclength(f, a, b, n)` that returns an array `s` with $s(x)$ values for n uniformly spaced coordinates x in $[a,b]$. Here `f(x)` is the Python implementation of the function that defines the curve we want to compute the arc length of.

b) How can you verify that the `arclength` function works correctly? Construct test case(s) and write corresponding test functions for automating the tests.

Hint. Check the implementation for curves with known arc length, e.g., a semi-circle and a straight line.

c) Apply the function to

$$f(x) = \int_{-2}^{x} = \frac{1}{\sqrt{2\pi}} e^{-4t^2} dt, \quad x \in [-2, 2] .$$

Compute $s(x)$ and plot it together with $f(x)$.
Filename: `arclength.py`.

Exercise A.14: Find difference equations for computing $\sin x$

The purpose of this exercise is to derive and implement difference equations for computing a Taylor polynomial approximation to $\sin x$:

$$\sin x \approx S(x; n) = \sum_{j=0}^{n} (-1)^j \frac{x^{2j+1}}{(2j+1)!} . \tag{A.59}$$

To compute $S(x; n)$ efficiently, write the sum as $S(x; n) = \sum_{j=0}^{n} a_j$, and derive a relation between two consecutive terms in the series:

$$a_j = -\frac{x^2}{(2j+1)2j} a_{j-1} . \tag{A.60}$$

Introduce $s_j = S(x; j-1)$ and a_j as the two sequences to compute. We have $s_0 = 0$ and $a_0 = x$.

a) Formulate the two difference equations for s_j and a_j.

Hint. Section A.1.8 explains how this task and the associated programming can be solved for the Taylor polynomial approximation of e^x.

b) Implement the system of difference equations in a function `sin_Taylor(x, n)`, which returns s_{n+1} and $|a_{n+1}|$. The latter is the first neglected term in the sum (since $s_{n+1} = \sum_{j=0}^{n} a_j$) and may act as a rough measure of the size of the error in the Taylor polynomial approximation.

c) Verify the implementation by computing the difference equations for $n = 2$ by hand (or in a separate program) and comparing with the output from the `sin_Taylor` function. Automate this comparison in a test function.

d) Make a table of s_n for various x and n values to verify that the accuracy of a Taylor polynomial improves as n increases and x decreases. Be aware of the fact that `sine_Taylor(x, n)` can give extremely inaccurate approximations to $\sin x$ if x is not sufficiently small and n sufficiently large.
Filename: `sin_Taylor_series_diffeq.py`.

Exercise A.15: Find difference equations for computing $\cos x$

Solve Exercise A.14 for the Taylor polynomial approximation to $\cos x$. (The relevant expression for the Taylor series is easily found in a mathematics textbook or by searching on the Internet.) Filename: `cos_Taylor_series_diffeq.py`.

Exercise A.16: Make a guitar-like sound

Given start values $x_0, x_1, \ldots, x_p$, the following difference equation is known to create guitar-like sound:

$$x_n = \frac{1}{2}(x_{n-p} + x_{n-p-1}), \quad n = p+1, \ldots, N\,. \tag{A.61}$$

With a sampling rate r, the frequency of this sound is given by r/p. Make a program with a function `solve(x, p)` which returns the solution array `x` of (A.61). To initialize the array `x[0:p+1]` we look at two methods, which can be implemented in two alternative functions:

- $x_0 = 1$, $x_1 = x_2 = \cdots = x_p = 0$
- $x_0, \ldots, x_p$ are uniformly distributed random numbers in $[-1, 1]$

Import `max_amplitude`, `write`, and `play` from the `scitools.sound` module. Choose a sampling rate r and set $p = r/440$ to create a 440 Hz tone (A). Create an array `x1` of zeros with length $3r$ such that the tone will last for 3 seconds. Initialize `x1` according to method 1 above and solve (A.61). Multiply the `x1` array by `max_amplitude`. Repeat this process for an array `x2` of length $2r$, but use method 2 for the initial values and choose p such that the tone is 392 Hz (G). Concatenate `x1` and `x2`, call `write` and then `play` to play the sound. As you will experience, this sound is amazingly similar to the sound of a guitar string, first playing A for 3 seconds and then playing G for 2 seconds.

The method (A.61) is called the Karplus-Strong algorithm and was discovered in 1979 by a researcher, Kevin Karplus, and his student Alexander Strong, at Stanford University. Filename: `guitar_sound.py`.

Exercise A.17: Damp the bass in a sound file

Given a sequence $x_0, \ldots, x_{N-1}$, the following *filter* transforms the sequence to a new sequence $y_0, \ldots, y_{N-1}$:

$$y_n = \begin{cases} x_n, & n = 0 \\ -\frac{1}{4}(x_{n-1} - 2x_n + x_{n+1}), & 1 \leq n \leq N-2 \\ x_n, & n = N-1 \end{cases} \tag{A.62}$$

If x_n represents sound, y_n is the same sound but with the bass damped. Load some sound file, e.g.,

```
x = scitools.sound.Nothing_Else_Matters()
# or
x = scitools.sound.Ja_vi_elsker()
```

to get a sound sequence. Apply the filter (A.62) and play the resulting sound. Plot the first 300 values in the x_n and y_n signals to see graphically what the filter does with the signal. Filename: `damp_bass.py`.

Exercise A.18: Damp the treble in a sound file

Solve Exercise A.17 to get some experience with coding a filter and trying it out on a sound. The purpose of this exercise is to explore some other filters that reduce the treble instead of the bass. Smoothing the sound signal will in general damp the treble, and smoothing is typically obtained by letting the values in the new filtered sound sequence be an average of the neighboring values in the original sequence.

The simplest smoothing filter can apply a standard average of three neighboring values:

$$y_n = \begin{cases} x_n, & n = 0 \\ \frac{1}{3}(x_{n-1} + x_n + x_{n+1}), & 1 \leq n \leq N-2 \\ x_n, & n = N-1 \end{cases} \tag{A.63}$$

Two other filters put less emphasis on the surrounding values:

$$y_n = \begin{cases} x_n, & n = 0 \\ \frac{1}{4}(x_{n-1} + 2x_n + x_{n+1}), & 1 \leq n \leq N-2 \\ x_n, & n = N-1 \end{cases} \tag{A.64}$$

$$y_n = \begin{cases} x_n, & n = 0, 1 \\ \frac{1}{16}(x_{n-2} + 4x_{n-1} + 6x_n + 4x_{n+1} + x_{n+2}), & 2 \leq n \leq N-3 \\ x_n, & n = N-2, N-1 \end{cases} \tag{A.65}$$

Apply all these three filters to a sound file and listen to the result. Plot the first 300 values in the x_n and y_n signals for each of the three filters to see graphically what the filter does with the signal. Filename: `damp_treble.py`.

Exercise A.19: Demonstrate oscillatory solutions of (A.13)

a) Write a program to solve the difference equation (A.13):

$$y_n = y_{n-1} + qy_{n-1}\left(1 - y_{n-1}\right), \quad n = 0, \ldots, N\,.$$

Read the input parameters y_0, q, and N from the command line. The variables and the equation are explained in Section A.1.5.

b) Equation (A.13) has the solution $y_n = 1$ as $n \to \infty$. Demonstrate, by running the program, that this is the case when $y_0 = 0.3$, $q = 1$, and $N = 50$.

c) For larger q values, y_n does not approach a constant limit, but y_n oscillates instead around the limiting value. Such oscillations are sometimes observed in wildlife populations. Demonstrate oscillatory solutions when q is changed to 2 and 3.

d) It could happen that y_n stabilizes at a constant level for larger N. Demonstrate that this is not the case by running the program with $N = 1000$.
Filename: `growth_logistic2.py`.

Exercise A.20: Automate computer experiments

It is tedious to run a program like the one from Exercise A.19 repeatedly for a wide range of input parameters. A better approach is to let the computer do the manual work. Modify the program from Exercise A.19 such that the computation of y_n and the plot is made in a function. Let the title in the plot contain the parameters y_0 and q (N is easily visible from the x axis). Also let the name of the plot file reflect the values of y_0, q, and N. Then make loops over y_0 and q to perform the following more comprehensive set of experiments:

- $y_0 = 0.01, 0.3$
- $q = 0.1, 1, 1.5, 1.8, 2, 2.5, 3$
- $N = 50$

How does the initial condition (the value y_0) seem to influence the solution?

Hint. If you do no want to get a lot of plots on the screen, which must be killed, drop the call to `show()` in Matplotlib or use `show=False` as argument to `plot` in SciTools.
Filename: `growth_logistic3.py`.

Exercise A.21: Generate an HTML report

Extend the program made in Exercise A.20 with a report containing all the plots. The report can be written in HTML and displayed by a web browser. The plots must then be generated in PNG format. The source of the HTML file will typically look as follows:

```
<html>
<body>
<p><img src="tmp_y0_0.01_q_0.1_N_50.png">
<p><img src="tmp_y0_0.01_q_1_N_50.png">
<p><img src="tmp_y0_0.01_q_1.5_N_50.png">
<p><img src="tmp_y0_0.01_q_1.8_N_50.png">
...
<p><img src="tmp_y0_0.01_q_3_N_1000.png">
</html>
</body>
```

Let the program write out the HTML text to a file. You may let the function making the plots return the name of the plot file such that this string can be inserted in the HTML file. Filename: `growth_logistic4.py`.

Exercise A.22: Use a class to archive and report experiments

The purpose of this exercise is to make the program from Exercise A.21 more flexible by creating a Python class that runs and archives all the experiments (provided you know how to program with Python classes). Here is a sketch of the class:

```python
class GrowthLogistic:
    def __init__(self, show_plot_on_screen=False):
        self.experiments = []
        self.show_plot_on_screen = show_plot_on_screen
        self.remove_plot_files()

    def run_one(self, y0, q, N):
        """Run one experiment."""
        # Compute y[n] in a loop...
        plotfile = 'tmp_y0_%g_q_%g_N_%d.png' % (y0, q, N)
        self.experiments.append({'y0': y0, 'q': q, 'N': N,
                                 'mean': mean(y[20:]),
                                 'y': y, 'plotfile': plotfile})
        # Make plot...

    def run_many(self, y0_list, q_list, N):
        """Run many experiments."""
        for q in q_list:
            for y0 in y0_list:
                self.run_one(y0, q, N)

    def remove_plot_files(self):
        """Remove plot files with names tmp_y0*.png."""
        import os, glob
        for plotfile in glob.glob('tmp_y0*.png'):
            os.remove(plotfile)

    def report(self, filename='tmp.html'):
        """
        Generate an HTML report with plots of all
        experiments generated so far.
        """
        # Open file and write HTML header...
        for e in self.experiments:
            html.write('<p><img src="%s">\n' % e['plotfile'])
        # Write HTML footer and close file...
```

Each time the `run_one` method is called, data about the current experiment is stored in the `experiments` list. Note that `experiments` contains a list of dictionaries. When desired, we can call the `report` method to collect all the plots made so far in an HTML report. A typical use of the class goes as follows:

```
N = 50
g = GrowthLogistic()
g.run_many(y0_list=[0.01, 0.3],
           q_list=[0.1, 1, 1.5, 1.8] + [2, 2.5, 3], N=N)
g.run_one(y0=0.01, q=3, N=1000)
g.report()
```

Make a complete implementation of class `GrowthLogistic` and test it with the small program above. The program file should be constructed as a module. Filename: `growth_logistic5.py`.

Exercise A.23: Explore logistic growth interactively

Class `GrowthLogistic` from Exercise A.22 is very well suited for interactive exploration. Here is a possible sample session for illustration:

```
>>> from growth_logistic5 import GrowthLogistic
>>> g = GrowthLogistic(show_plot_on_screen=True)
>>> q = 3
>>> g.run_one(0.01, q, 100)
>>> y = g.experiments[-1]['y']
>>> max(y)
1.3326056469620293
>>> min(y)
0.0029091569028512065
```

Extend this session with an investigation of the oscillations in the solution y_n. For this purpose, make a function for computing the local maximum values y_n and the corresponding indices where these local maximum values occur. We can say that y_i is a local maximum value if

$$y_{i-1} < y_i > y_{i+1} .$$

Plot the sequence of local maximum values in a new plot. If $I_0, I_1, I_2, \ldots$ constitute the set of increasing indices corresponding to the local maximum values, we can define the periods of the oscillations as $I_1 - I_0$, $I_2 - I_1$, and so forth. Plot the length of the periods in a separate plot. Repeat this investigation for $q = 2.5$. Filename: `GrowthLogistic_interactive.py`.

Exercise A.24: Simulate the price of wheat

The demand for wheat in year t is given by

$$D_t = ap_t + b,$$

where $a < 0$, $b > 0$, and p_t is the price of wheat. Let the supply of wheat be

$$S_t = Ap_{t-1} + B + \ln(1 + p_{t-1}),$$

where A and B are given constants. We assume that the price p_t adjusts such that all the produced wheat is sold. That is, $D_t = S_t$.

a) For $A = 1$, $a = -3, b = 5, B = 0$, find from numerical computations, a stable price such that the production of wheat from year to year is constant. That is, find p such that $ap + b = Ap + B + \ln(1 + p)$.

b) Assume that in a very dry year the production of wheat is much less than planned. Given that price this year, p_0, is 4.5 and $D_t = S_t$, compute in a program how the prices $p_1, p_2, \ldots, p_N$ develop. This implies solving the difference equation

$$ap_t + b = Ap_{t-1} + B + \ln(1 + p_{t-1})\,.$$

From the p_t values, compute S_t and plot the points (p_t, S_t) for $t = 0, 1, 2, \ldots, N$. How do the prices move when $N \to \infty$?
Filename: `wheat.py`.

Debugging F

Testing a program to find errors usually takes much more time than to write the code. This appendix is devoted to tools and good habits for effective debugging. Section F.1 describes the Python debugger, a key tool for examining the internal workings of a code, while Section F.2 explains how solve problems and write software to simplify the debugging process.

F.1 Using a debugger

A debugger is a program that can help you to find out what is going on in a computer program. You can stop the execution at any prescribed line number, print out variables, continue execution, stop again, execute statements one by one, and repeat such actions until you have tracked down abnormal behavior and found bugs.

Here we shall use the debugger to demonstrate the program flow of the code `Simpson.py` (which can integrate functions of one variable with the famous Simpson's rule). This development of this code is explained in Section 3.4.2. You are strongly encouraged to carry out the steps below on your computer to get a glimpse of what a debugger can do.

Step 1. Go to the folder `src/funcif`[1] where the program `Simpson.py` resides.

Step 2. If you use the Spyder Integrated Development Environment, choose *Debug* on the *Run* pull-down menu. If you run your programs in a plain terminal window, start IPython:

Terminal

```
Terminal> ipython
```

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

Run the program `Simpson.py` with the debugger on (`-d`):

```
In [1]: run -d Simpson.py
```

We now enter the debugger and get a prompt

```
ipdb>
```

After this prompt we can issue various debugger commands. The most important ones will be described as we go along.

Step 3. Type `continue` or just `c` to go to the first line in the file. Now you can see a printout of where we are in the program:

```
1---> 1 def Simpson(f, a, b, n=500):
      2     """
      3     Return the approximation of the integral of f
```

Each program line is numbered and the arrow points to the next line to be executed. This is called the *current line*.

Step 4. You can set a *break point* where you want the program to stop so that you can examine variables and perhaps follow the execution closely. We start by setting a break point in the `application` function:

```
ipdb> break application
Breakpoint 2 at /home/.../src/funcif/Simpson.py:30
```

You can also say `break X`, where `X` is a line number in the file.

Step 5. Continue execution until the break point by writing `continue` or `c`. Now the program stops at line 31 in the `application` function:

```
ipdb> c
> /home/.../src/funcif/Simpson.py(31)application()
2    30 def application():
---> 31     from math import sin, pi
     32     print 'Integral of 1.5*sin^3 from 0 to pi:'
```

Step 6. Typing `step` or just `s` executes one statement at a time:

```
ipdb> s
> /home/.../src/funcif/Simpson.py(32)application()
     31     from math import sin, pi
---> 32     print 'Integral of 1.5*sin^3 from 0 to pi:'
     33     for n in 2, 6, 12, 100, 500:

ipdb> s
Integral of 1.5*sin^3 from 0 to pi:
> /home/.../src/funcif/Simpson.py(33)application()
     32     print 'Integral of 1.5*sin^3 from 0 to pi:'
---> 33     for n in 2, 6, 12, 100, 500:
     34         approx = Simpson(h, 0, pi, n)
```

Typing another `s` reaches the call to `Simpson`, and a new `s` steps *into* the function `Simpson`:

```
ipdb> s
--Call--
> /home/.../src/funcif/Simpson.py(1)Simpson()
1---> 1 def Simpson(f, a, b, n=500):
      2     """
      3     Return the approximation of the integral of f
```

Type a few more `s` to step ahead of the `if` tests.

Step 7. Examining the contents of variables is easy with the `print` (or `p`) command:

```
ipdb> print f, a, b, n
<function h at 0x898ef44> 0 3.14159265359 2
```

We can also check the type of the objects:

```
ipdb> whatis f
Function h
ipdb> whatis a
<type 'int'>
ipdb> whatis b
<type 'float'>
ipdb> whatis n
<type 'int'>
```

Step 8. Set a new break point in the `application` function so that we can jump directly there without having to go manually through all the statements in the `Simpson` function. To see line numbers and corresponding statements around some line with number `X`, type `list X`. For example,

```
ipdb> list 32
     27 def h(x):
     28     return (3./2)*sin(x)**3
     29
     30 from math import sin, pi
     31
2    32 def application():
     33     print 'Integral of 1.5*sin^3 from 0 to pi:'
     34     for n in 2, 6, 12, 100, 500:
     35         approx = Simpson(h, 0, pi, n)
     36         print 'n=%3d, approx=%18.15f, error=%9.2E' % \
     37               (n, approx, 2-approx)
```

We set a line break at line 35:

```
ipdb> break 35
Breakpoint 3 at /home/.../src/funcif/Simpson.py:35
```

Typing `c` continues execution up to the next break point, line 35.

Step 9. The command `next` or `n` is like `step` or `s` in that the current line is executed, but the execution does not step into functions, instead the function calls are just performed and the program stops at the next line:

```
ipdb> n
> /home/.../src/funcif/Simpson.py(36)application()
3    35          approx = Simpson(h, 0, pi, n)
---> 36          print 'n=%3d, approx=%18.15f, error=%9.2E' % \
     37                (n, approx, 2-approx)
ipdb> print approx, n
1.9891717005835792 6
```

Step 10. The command `disable X Y Z` disables break points with numbers `X`, `Y`, and `Z`, and so on. To remove our three break points and continue execution until the program naturally stops, we write

```
ipdb> disable 1 2 3
ipdb> c
n=100, approx= 1.999999902476350, error= 9.75E-08
n=500, approx= 1.999999999844138, error= 1.56E-10

In [2]:
```

At this point, I hope you realize that a debugger is a very handy tool for monitoring the program flow, checking variables, and thereby understanding why errors occur.

F.2 How to debug

Most programmers will claim that writing code consumes a small portion of the time it takes to develop a program: the major portion of the work concerns testing the program and finding errors.

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. Brian W. Kernighan, computer scientist, 1942-.

Newcomers to programming often panic when their program runs for the first time and aborts with a seemingly cryptic error message. How do you approach the art of debugging? This appendix summarizes some important working habits in this respect. Some of the tips are useful for problem solving in general, not only when writing and testing Python programs.

F.2.1 A recipe for program writing and debugging

1. Understand the problem. Make sure that you really understand the task the program is supposed to solve. We can make a general claim: if you do not understand the problem and the solution method, you will never be able to make a correct program. It may be argued that this claim is not entirely true: sometimes students with limited understanding of the problem are able to grab a similar program and guess at a few modifications, and actually get a program that works. But this technique is based on luck and not on understanding. The famous Norwegian computer scientist Kristen Nygaard (1926-2002) phrased it precisely: *Programming is understanding.* It may be necessary to read a problem description or exercise many times and study relevant background material before starting on the programming part of the problem solving process.

2. Work out examples. Start with sketching one or more examples on input and output of the program. Such examples are important for controlling the understanding of the purpose of the program, and for verifying the implementation.

3. Decide on a user interface. Find out how you want to get data into the program. You may want to grab data from the command-line, a file, or a dialog with questions and answers.

4. Make algorithms. Identify the key tasks to be done in the program and sketch rough algorithms for these. Some programmers prefer to do this on a piece of paper, others prefer to start directly in Python and write Python-like code with comments to sketch the program (this is easily developed into real Python code later).

5. Look up information. Few programmers can write the whole program without consulting manuals, books, and the Internet. You need to know and understand the basic constructs in a language and some fundamental problem solving techniques, but technical details can be looked up.

The more program examples you have studied (in this book, for instance), the easier it is to adapt ideas from an existing example to solve a new problem.

6. Write the program. Be extremely careful with what you write. In particular, compare all mathematical statements and algorithms with the original mathematical expressions.

In longer programs, do not wait until the program is complete before you start testing it, test parts while you write.

7. Run the program. If the program aborts with an error message from Python, these messages are fortunately quite precise and helpful. First, locate the line number where the error occurs and read the statement, then carefully read the error message. The most common errors (exceptions) are listed below.

`SyntaxError`: Illegal Python code.

```
    File "somefile.py", line 5
      x = . 5
          ^
  SyntaxError: invalid syntax
```

Often the error is precisely indicated, as above, but sometimes you have to search for the error on the previous line.

`NameError`: A name (variable, function, module) is not defined.

```
  File "somefile.py", line 20, in <module>
    table(10)
  File "somefile.py", line 16, in table
    value, next, error = L(x, n)
  File "somefile.py", line 8, in L
    exact_error = log(1+x) - value_of_sum
NameError: global name 'value_of_sum' is not defined
```

Look at the last of the lines starting with `File` to see where in the program the error occurs. The most common reasons for a `NameError` are

- a misspelled name,
- a variable that is not initialized,
- a function that you have forgotten to define,
- a module that is not imported.

`TypeError`: An object of wrong type is used in an operation.

```
  File "somefile.py", line 17, in table
    value, next, error = L(x, n)
  File "somefile.py", line 7, in L
    first_neglected_term = (1.0/(n+1))*(x/(1.0+x))**(n+1)
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

Print out objects and their types (here: `print x, type(x), n, type(n)`), and you will most likely get a surprise. The reason for a `TypeError` is often far away from the line where the `TypeError` occurs.

`ValueError`: An object has an illegal value.

```
  File "somefile.py", line 8, in L
    y = sqrt(x)
ValueError: math domain error
```

Print out the value of objects that can be involved in the error (here: `print x`).

`IndexError`: An index in a list, tuple, string, or array is too large.

```
  File "somefile.py", line 21
    n = sys.argv[i+1]
IndexError: list index out of range
```

Print out the length of the list, and the index if it involves a variable (here: `print len(sys.argv), i`).

8. Verify the results. Assume now that we have a program that runs without error messages from Python. Before judging the results of the program, set precisely up a test case where you know the exact solution. This is in general quite difficult. In complicated mathematical problems it is an art to construct good test problems and procedures for providing evidence that the program works.

If your program produces wrong answers, start to *examine intermediate results*. Never forget that your own hand calculations that you use to test the program may be wrong!

9. Use a debugger. If you end up inserting a lot of `print` statements in the program for checking intermediate results, you might benefit from using a debugger as explained in Section F.1.

Some may think that this list of nine points is very comprehensive. However, the recipe just contains the steps that you should always carry out when developing programs. Never forget that computer programming is a difficult task.

Program writing is substantially more demanding than book writing. Why is it so? I think the main reason is that a larger attention span is needed when working on a large computer program than when doing other intellectual tasks. Donald Knuth [12, p. 18], computer scientist, 1938-.

F.2.2 Application of the recipe

Let us illustrate the points above in a specific programming problem: implementation of the Midpoint rule for numerical integration. The Midpoint rule for approximating an integral $\int_a^b f(x)dx$ reads

$$I = h\sum_{i=1}^{n} f(a + (i - \frac{1}{2})h), \quad h = \frac{b-a}{n}\,. \tag{F.1}$$

We just follow the individual steps in the recipe to develop the code.

1. Understand the problem. In this problem we must understand how to program the formula (F.1). Observe that we do not need to understand how the formula is derived, because we do not apply the derivation in the program. What is important, is to notice that the formula is an *approximation* of an integral. Comparing the result of the program with the exact value of the integral will in general show a discrepancy. Whether we have an approximation error or a programming error is always difficult to judge. We will meet this difficulty below.

2. Work out examples. As a test case we choose to integrate

$$f(x) = \sin^{-1}(x)\,. \tag{F.2}$$

between 0 and π. From a table of integrals we find that this integral equals

$$\left[x\sin^{-1}(x) + \sqrt{1-x^2}\,\right]_0^\pi\,. \tag{F.3}$$

The formula (F.1) gives an approximation to this integral, so the program will (most likely) print out a result different from (F.3). It would therefore be very helpful to construct a calculation where there are no approximation errors. Numerical integration rules usually integrate some polynomial of low order exactly. For the Midpoint rule it is obvious, if you understand the derivation of this rule, that a constant function will be integrated exactly. We therefore also introduce a test problem where we integrate $g(x) = 1$ from 0 to 10. The answer should be exactly 10.

Input and output: The input to the calculations is the function to integrate, the integration limits a and b, and the n parameter (number of intervals) in the formula (F.1). The output from the calculations is the approximation to the integral.

3. Decide on a user interface. We find it easiest at this beginning stage to program the two functions $f(x)$ and $g(x)$ directly in the program. We also specify the corresponding integration limits a and b in the program, but we read a common n for both integrals from the command line. Note that this is not a flexible user interface, but it suffices as a start for creating a working program. A much better user interface is to read f, a, b, and n from the command line, which will be done later in a more complete solution to the present problem.

4. Make algorithms. Like most mathematical programming problems, also this one has a generic part and an application part. The generic part is the formula (F.1), which is applicable to an arbitrary function $f(x)$. The implementation should reflect that we can specify any Python function `f(x)` and get it integrated. This principle calls for calculating (F.1) in a Python function where the input to the computation (f, a, b, n) are arguments. The function heading can look as `integrate(f, a, b, n)`, and the value of (F.1) is returned.

The test part of the program consists of defining the test functions $f(x)$ and $g(x)$ and writing out the calculated approximations to the corresponding integrals.

A first rough sketch of the program can then be

```
def integrate(f, a, b, n):
    # compute integral, store in I
    return I

def f(x):
...

def g(x):
...

# test/application part:
n = sys.argv[1]
I = integrate(g, 0, 10,  n)
print "Integral of g equals %g" % I
I = integrate(f, 0, pi,  n)
# calculate and print out the exact integral of f
```

The next step is to make a detailed implementation of the `integrate` function. Inside this function we need to compute the sum (F.1). In general, sums are computed by a `for` loop over the summation index, and inside the loop we calculate a term in the sum and add it to an accumulation variable. Here is the algorithm in Python code:

```
s = 0
for i in range(1, n+1):
    s = s + f(a + (i-0.5)*h)
I = s*h
```

5. Look up information. Our test function $f(x) = \sin^{-1}(x)$ must be evaluated in the program. How can we do this? We know that many common mathematical functions are offered by the `math` module. It is therefore natural to check if this module has an inverse sine function. The best place to look for Python modules is the Python Standard Library[2] [3] documentation, which has a search facility. Typing *math* brings up a link to the `math` module, there we find `math.asin` as the function we need. Alternatively, one can use the command line utility `pydoc` and write `pydoc math` to look up all the functions in the module.

In this simple problem, we use very basic programming constructs and there is hardly any need for looking at similar examples to get started with the problem solving process. We need to know how to program a sum, though, via a `for` loop and an accumulation variable for the sum. Examples are found in Sections 2.1.4 and 3.1.8.

6. Write the program. Here is our first attempt to write the program. You can find the whole code in the file `integrate_v1.py`.

```
def integrate(f, a, b, n):
    s = 0
    for i in range(1, n):
        s += f(a + i*h)
    return s

def f(x):
return asin(x)

def g(x):
return 1

# Test/application part
n = sys.argv[1]
I = integrate(g, 0, 10,  n)
print "Integral of g equals %g" % I
I = integrate(f, 0, pi,  n)
I_exact = pi*asin(pi) - sqrt(1 - pi**2) - 1
print "Integral of f equals %g (exact value is %g)’ % \
  (I, I_exact)
```

7. Run the program. We try a first execution from IPython

```
In [1]: run integrate_v1.py
```

Unfortunately, the program aborts with an error:

```
  File "integrate_v1.py", line 8
    return asin(x)
         ^
IndentationError: expected an indented block
```

We go to line 8 and look at that line and the surrounding code:

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

```
def f(x):
return asin(x)
```

Python expects that the return line is indented, because the function body must always be indented. By the way, we realize that there is a similar error in the `g(x)` function as well. We correct these errors:

```
def f(x):
    return asin(x)

def g(x):
    return 1
```

Running the program again makes Python respond with

```
  File "integrate_v1.py", line 24
    (I, I_exact)
               ^
SyntaxError: EOL while scanning single-quoted string
```

There is nothing wrong with line 24, but line 24 is a part of the statement starting on line 23:

```
print "Integral of f equals %g (exact value is %g)' % \
      (I, I_exact)
```

A `SyntaxError` implies that we have written illegal Python code. Inspecting line 23 reveals that the string to be printed starts with a double quote, but ends with a single quote. We must be consistent and use the same enclosing quotes in a string. Correcting the statement,

```
print "Integral of f equals %g (exact value is %g)" % \
      (I, I_exact)
```

and rerunning the program yields the output

```
Traceback (most recent call last):
  File "integrate_v1.py", line 18, in <module>
    n = sys.argv[1]
NameError: name 'sys' is not defined
```

Obviously, we need to import `sys` before using it. We add `import sys` and run again:

```
Traceback (most recent call last):
  File "integrate_v1.py", line 19, in <module>
    n = sys.argv[1]
IndexError: list index out of range
```

This is a very common error: we index the list `sys.argv` out of range because we have not provided enough command-line arguments. Let us use $n = 10$ in the test and provide that number on the command line:

```
In [5]: run integrate_v1.py 10
```

We still have problems:

this technique is based on luck and not on understanding. The famous Norwegian computer scientist Kristen Nygaard (1926-2002) phrased it precisely: *Programming is understanding*. It may be necessary to read a problem description or exercise many times and study relevant background material before starting on the programming part of the problem solving process.

2. Work out examples. Start with sketching one or more examples on input and output of the program. Such examples are important for controlling the understanding of the purpose of the program, and for verifying the implementation.

3. Decide on a user interface. Find out how you want to get data into the program. You may want to grab data from the command-line, a file, or a dialog with questions and answers.

4. Make algorithms. Identify the key tasks to be done in the program and sketch rough algorithms for these. Some programmers prefer to do this on a piece of paper, others prefer to start directly in Python and write Python-like code with comments to sketch the program (this is easily developed into real Python code later).

5. Look up information. Few programmers can write the whole program without consulting manuals, books, and the Internet. You need to know and understand the basic constructs in a language and some fundamental problem solving techniques, but technical details can be looked up.

The more program examples you have studied (in this book, for instance), the easier it is to adapt ideas from an existing example to solve a new problem.

6. Write the program. Be extremely careful with what you write. In particular, compare all mathematical statements and algorithms with the original mathematical expressions.

In longer programs, do not wait until the program is complete before you start testing it, test parts while you write.

7. Run the program. If the program aborts with an error message from Python, these messages are fortunately quite precise and helpful. First, locate the line number where the error occurs and read the statement, then carefully read the error message. The most common errors (exceptions) are listed below.

`SyntaxError`: Illegal Python code.

```
    File "somefile.py", line 5
      x = . 5
          ^
  SyntaxError: invalid syntax
```

Often the error is precisely indicated, as above, but sometimes you have to search for the error on the previous line.

`NameError`: A name (variable, function, module) is not defined.

```
  File "somefile.py", line 20, in <module>
    table(10)
  File "somefile.py", line 16, in table
    value, next, error = L(x, n)
  File "somefile.py", line 8, in L
    exact_error = log(1+x) - value_of_sum
NameError: global name 'value_of_sum' is not defined
```

Look at the last of the lines starting with `File` to see where in the program the error occurs. The most common reasons for a `NameError` are

- a misspelled name,
- a variable that is not initialized,
- a function that you have forgotten to define,
- a module that is not imported.

`TypeError`: An object of wrong type is used in an operation.

```
  File "somefile.py", line 17, in table
    value, next, error = L(x, n)
  File "somefile.py", line 7, in L
    first_neglected_term = (1.0/(n+1))*(x/(1.0+x))**(n+1)
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

Print out objects and their types (here: `print x, type(x), n, type(n)`), and you will most likely get a surprise. The reason for a `TypeError` is often far away from the line where the `TypeError` occurs.

`ValueError`: An object has an illegal value.

```
  File "somefile.py", line 8, in L
    y = sqrt(x)
ValueError: math domain error
```

Print out the value of objects that can be involved in the error (here: `print x`).

`IndexError`: An index in a list, tuple, string, or array is too large.

```
  File "somefile.py", line 21
    n = sys.argv[i+1]
IndexError: list index out of range
```

Print out the length of the list, and the index if it involves a variable (here: `print len(sys.argv), i`).

8. Verify the results. Assume now that we have a program that runs without error messages from Python. Before judging the results of the program, set precisely up a test case where you know the exact solution. This is in general quite difficult. In complicated mathematical problems it is an art to construct good test problems and procedures for providing evidence that the program works.

If your program produces wrong answers, start to *examine intermediate results*. Never forget that your own hand calculations that you use to test the program may be wrong!

9. Use a debugger. If you end up inserting a lot of `print` statements in the program for checking intermediate results, you might benefit from using a debugger as explained in Section F.1.

Some may think that this list of nine points is very comprehensive. However, the recipe just contains the steps that you should always carry out when developing programs. Never forget that computer programming is a difficult task.

Program writing is substantially more demanding than book writing. Why is it so? I think the main reason is that a larger attention span is needed when working on a large computer program than when doing other intellectual tasks. Donald Knuth [12, p. 18], computer scientist, 1938-.

F.2.2 Application of the recipe

Let us illustrate the points above in a specific programming problem: implementation of the Midpoint rule for numerical integration. The Midpoint rule for approximating an integral $\int_a^b f(x)dx$ reads

$$I = h\sum_{i=1}^{n} f(a + (i - \frac{1}{2})h), \quad h = \frac{b-a}{n}\,. \tag{F.1}$$

We just follow the individual steps in the recipe to develop the code.

1. Understand the problem. In this problem we must understand how to program the formula (F.1). Observe that we do not need to understand how the formula is derived, because we do not apply the derivation in the program. What is important, is to notice that the formula is an *approximation* of an integral. Comparing the result of the program with the exact value of the integral will in general show a discrepancy. Whether we have an approximation error or a programming error is always difficult to judge. We will meet this difficulty below.

2. Work out examples. As a test case we choose to integrate

$$f(x) = \sin^{-1}(x)\,. \tag{F.2}$$

between 0 and π. From a table of integrals we find that this integral equals

$$\left[x\sin^{-1}(x) + \sqrt{1-x^2}\,\right]_0^\pi\,. \tag{F.3}$$

The formula (F.1) gives an approximation to this integral, so the program will (most likely) print out a result different from (F.3). It would therefore be very helpful to construct a calculation where there are no approximation errors. Numerical integration rules usually integrate some polynomial of low order exactly. For the Midpoint rule it is obvious, if you understand the derivation of this rule, that a constant function will be integrated exactly. We therefore also introduce a test problem where we integrate $g(x) = 1$ from 0 to 10. The answer should be exactly 10.

Input and output: The input to the calculations is the function to integrate, the integration limits a and b, and the n parameter (number of intervals) in the formula (F.1). The output from the calculations is the approximation to the integral.

3. Decide on a user interface. We find it easiest at this beginning stage to program the two functions $f(x)$ and $g(x)$ directly in the program. We also specify the corresponding integration limits a and b in the program, but we read a common n for both integrals from the command line. Note that this is not a flexible user interface, but it suffices as a start for creating a working program. A much better user interface is to read f, a, b, and n from the command line, which will be done later in a more complete solution to the present problem.

4. Make algorithms. Like most mathematical programming problems, also this one has a generic part and an application part. The generic part is the formula (F.1), which is applicable to an arbitrary function $f(x)$. The implementation should reflect that we can specify any Python function `f(x)` and get it integrated. This principle calls for calculating (F.1) in a Python function where the input to the computation (f, a, b, n) are arguments. The function heading can look as `integrate(f, a, b, n)`, and the value of (F.1) is returned.

The test part of the program consists of defining the test functions $f(x)$ and $g(x)$ and writing out the calculated approximations to the corresponding integrals.

A first rough sketch of the program can then be

```
def integrate(f, a, b, n):
    # compute integral, store in I
    return I

def f(x):
...

def g(x):
...

# test/application part:
n = sys.argv[1]
I = integrate(g, 0, 10,  n)
print "Integral of g equals %g" % I
I = integrate(f, 0, pi,  n)
# calculate and print out the exact integral of f
```

The next step is to make a detailed implementation of the `integrate` function. Inside this function we need to compute the sum (F.1). In general, sums are computed by a `for` loop over the summation index, and inside the loop we calculate a term in the sum and add it to an accumulation variable. Here is the algorithm in Python code:

```
s = 0
for i in range(1, n+1):
    s = s + f(a + (i-0.5)*h)
I = s*h
```

5. Look up information. Our test function $f(x) = \sin^{-1}(x)$ must be evaluated in the program. How can we do this? We know that many common mathematical functions are offered by the `math` module. It is therefore natural to check if this module has an inverse sine function. The best place to look for Python modules is the Python Standard Library[2] [3] documentation, which has a search facility. Typing *math* brings up a link to the `math` module, there we find `math.asin` as the function we need. Alternatively, one can use the command line utility `pydoc` and write `pydoc math` to look up all the functions in the module.

In this simple problem, we use very basic programming constructs and there is hardly any need for looking at similar examples to get started with the problem solving process. We need to know how to program a sum, though, via a `for` loop and an accumulation variable for the sum. Examples are found in Sections 2.1.4 and 3.1.8.

6. Write the program. Here is our first attempt to write the program. You can find the whole code in the file `integrate_v1.py`.

```
def integrate(f, a, b, n):
    s = 0
    for i in range(1, n):
        s += f(a + i*h)
    return s

def f(x):
return asin(x)

def g(x):
return 1

# Test/application part
n = sys.argv[1]
I = integrate(g, 0, 10,  n)
print "Integral of g equals %g" % I
I = integrate(f, 0, pi,  n)
I_exact = pi*asin(pi) - sqrt(1 - pi**2) - 1
print "Integral of f equals %g (exact value is %g)' % \
  (I, I_exact)
```

7. Run the program. We try a first execution from IPython

```
In [1]: run integrate_v1.py
```

Unfortunately, the program aborts with an error:

```
  File "integrate_v1.py", line 8
    return asin(x)
         ^
IndentationError: expected an indented block
```

We go to line 8 and look at that line and the surrounding code:

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

```
def f(x):
return asin(x)
```

Python expects that the return line is indented, because the function body must always be indented. By the way, we realize that there is a similar error in the `g(x)` function as well. We correct these errors:

```
def f(x):
    return asin(x)

def g(x):
    return 1
```

Running the program again makes Python respond with

```
  File "integrate_v1.py", line 24
    (I, I_exact)
               ^
SyntaxError: EOL while scanning single-quoted string
```

There is nothing wrong with line 24, but line 24 is a part of the statement starting on line 23:

```
print "Integral of f equals %g (exact value is %g)' % \
      (I, I_exact)
```

A `SyntaxError` implies that we have written illegal Python code. Inspecting line 23 reveals that the string to be printed starts with a double quote, but ends with a single quote. We must be consistent and use the same enclosing quotes in a string. Correcting the statement,

```
print "Integral of f equals %g (exact value is %g)" % \
      (I, I_exact)
```

and rerunning the program yields the output

```
Traceback (most recent call last):
  File "integrate_v1.py", line 18, in <module>
    n = sys.argv[1]
NameError: name 'sys' is not defined
```

Obviously, we need to import `sys` before using it. We add `import sys` and run again:

```
Traceback (most recent call last):
  File "integrate_v1.py", line 19, in <module>
    n = sys.argv[1]
IndexError: list index out of range
```

This is a very common error: we index the list `sys.argv` out of range because we have not provided enough command-line arguments. Let us use $n = 10$ in the test and provide that number on the command line:

```
In [5]: run integrate_v1.py 10
```

We still have problems:

```
Traceback (most recent call last):
  File "integrate_v1.py", line 20, in <module>
    I = integrate(g, 0, 10,  n)
  File "integrate_v1.py", line 7, in integrate
    for i in range(1, n):
TypeError: range() integer end argument expected, got str.
```

It is the final `File` line that counts (the previous ones describe the nested functions calls up to the point where the error occurred). The error message for line 7 is very precise: the end argument to `range`, `n`, should be an integer, but it is a string. We need to convert the string `sys.argv[1]` to `int` before sending it to the `integrate` function:

```
n = int(sys.argv[1])
```

After a new edit-and-run cycle we have other error messages waiting:

```
Traceback (most recent call last):
  File "integrate_v1.py", line 20, in <module>
    I = integrate(g, 0, 10,  n)
  File "integrate_v1.py", line 8, in integrate
    s += f(a + i*h)
NameError: global name 'h' is not defined
```

The `h` variable is used without being assigned a value. From the formula (F.1) we see that $h = (b - a)/n$, so we insert this assignment at the top of the `integrate` function:

```
def integrate(f, a, b, n):
    h = (b-a)/n
    ...
```

A new run results in a new error:

```
Integral of g equals 9
Traceback (most recent call last):
  File "integrate_v1.py", line 23, in <module>
    I = integrate(f, 0, pi,  n)
NameError: name 'pi' is not defined
```

Looking carefully at all output, we see that the program managed to call the `integrate` function with `g` as input and write out the integral. However, in the call to `integrate` with `f` as argument, we get a `NameError`, saying that `pi` is undefined. When we wrote the program we took it for granted that `pi` was π, but we need to import `pi` from `math` to get this variable defined, before we call `integrate`:

```
from math import pi
I = integrate(f, 0, pi,  n)
```

The output of a new run is now

```
Integral of g equals 9
Traceback (most recent call last):
  File "integrate_v1.py", line 24, in <module>
    I = integrate(f, 0, pi,  n)
  File "integrate_v1.py", line 9, in integrate
    s += f(a + i*h)
  File "integrate_v1.py", line 13, in f
    return asin(x)
NameError: global name 'asin' is not defined
```

A similar error occurred: `asin` is not defined as a function, and we need to import it from `math`. We can either do a

```
from math import pi, asin
```

or just do the rough

```
from math import *
```

to avoid any further errors with undefined names from the `math` module (we will get one for the `sqrt` function later, so we simply use the last "import all" kind of statement).

There are still more errors:

```
Integral of g equals 9
Traceback (most recent call last):
  File "integrate_v1.py", line 24, in <module>
    I = integrate(f, 0, pi,  n)
  File "integrate_v1.py", line 9, in integrate
    s += f(a + i*h)
  File "integrate_v1.py", line 13, in f
    return asin(x)
ValueError: math domain error
```

Now the error concerns a wrong `x` value in the `f` function. Let us print out `x`:

```
def f(x):
    print x
    return asin(x)
```

The output becomes

```
Integral of g equals 9
0.314159265359
0.628318530718
0.942477796077
1.25663706144
Traceback (most recent call last):
  File "integrate_v1.py", line 25, in <module>
    I = integrate(f, 0, pi,  n)
  File "integrate_v1.py", line 9, in integrate
    s += f(a + i*h)
  File "integrate_v1.py", line 14, in f
    return asin(x)
ValueError: math domain error
```

We see that all the `asin(x)` computations are successful up to and including $x = 0.942477796077$, but for $x = 1.25663706144$ we get an error. A `math domain error` may point to a wrong x value for $\sin^{-1}(x)$ (recall that the domain of a function specifies the legal x values for that function).

To proceed, we need to think about the mathematics of our problem: Since $\sin(x)$ is always between -1 and 1, the inverse sine function cannot take x values outside the interval $[-1, 1]$. The problem is that we try to integrate $\sin^{-1}(x)$ from 0 to π, but only integration limits within $[-1, 1]$ make sense (unless we allow for complex-valued trigonometric functions). Our test problem is hence wrong from a mathematical point of view. We need to adjust the limits, say 0 to 1 instead of 0 to π. The corresponding program modification reads

```
I = integrate(f, 0, 1,  n)
```

We run again and get

```
Integral of g equals 9
0
0
0
0
0
0
0
0
0
Traceback (most recent call last):
  File "integrate_v1.py", line 26, in <module>
    I_exact = pi*asin(pi) - sqrt(1 - pi**2) - 1
ValueError: math domain error
```

It is easy to go directly to the `ValueError` now, but one should always examine the output from top to bottom. If there is strange output before Python reports an error, there may be an error indicated by our `print` statements. This is not the case in the present example, but it is a good habit to start at the top of the output anyway. We see that all our `print x` statements inside the `f` function say that `x` is zero. This must be wrong - the idea of the integration rule is to pick n different points in the integration interval $[0, 1]$.

Our `f(x)` function is called from the `integrate` function. The argument to `f`, `a + i*h`, is seemingly always 0. Why? We print out the argument and the values of the variables that make up the argument:

```
def integrate(f, a, b, n):
    h = (b-a)/n
    s = 0
    for i in range(1, n):
        print a, i, h, a+i*h
        s += f(a + i*h)
    return s
```

Running the program shows that `h` is zero and therefore `a+i*h` is zero.

Why is `h` zero? We need a new `print` statement in the computation of `h`:

```
def integrate(f, a, b, n):
    h = (b-a)/n
    print b, a, n, h
    ...
```

The output shows that `a`, `b`, and `n` are correct. Now we have encountered a very common error in Python version 2 and C-like programming languages: integer division (see Section 1.3.1). The formula $(1 - 0)/10 = 1/10$ is zero according to integer division. The reason is that `a` and `b` are specified as `0` and `1` in the call to `integrate`, and `0` and `1` imply `int` objects. Then `b-a` becomes an `int`, and `n` is an `int`, causing an `int/int` division. We must ensure that `b-a` is `float` to get the right mathematical division in the computation of `h`:

```
def integrate(f, a, b, n):
    h = float(b-a)/n
    ...
```

Thinking that the problem with wrong x values in the inverse sine function is resolved, we may remove all the `print` statements in the program, and run again.

The output now reads

```
Integral of g equals 9
Traceback (most recent call last):
  File "integrate_v1.py", line 25, in <module>
    I_exact = pi*asin(pi) - sqrt(1 - pi**2) - 1
ValueError: math domain error
```

That is, we are back to the `ValueError` we have seen before. The reason is that `asin(pi)` does not make sense, and the argument to `sqrt` is negative. The error is simply that we forgot to adjust the upper integration limit in the computation of the exact result. This is another very common error. The correct line is

```
I_exact = 1*asin(1) - sqrt(1 - 1**2) - 1
```

We could have avoided the error by introducing variables for the integration limits, and a function for $\int f(x)dx$ would make the code cleaner:

```
a = 0; b = 1
def int_f_exact(x):
    return x*asin(x) - sqrt(1 - x**2)
I_exact = int_f_exact(b) - int_f_exact(a)
```

Although this is more work than what we initially aimed at, it usually saves time in the debugging phase to do things this proper way.

Eventually, the program seems to work! The output is just the result of our two `print` statements:

```
Integral of g equals 9
Integral of f equals 5.0073 (exact value is 0.570796)
```

8. Verify the results. Now it is time to check if the numerical results are correct. We start with the simple integral of 1 from 0 to 10: the answer should be 10, not 9. Recall that for this particular choice of integration function, there is no approximation error involved (but there could be a small round-off error). Hence, there must be a programming error.

To proceed, we need to calculate some intermediate mathematical results by hand and compare these with the corresponding statements in the program. We choose a very simple test problem with $n = 2$ and $h = (10 - 0)/2 = 5$. The formula (F.1) becomes

$$I = 5 \cdot (1 + 1) = 10\,.$$

Running the program with $n = 2$ gives

```
Integral of g equals 1
```

We insert some `print` statements inside the `integrate` function:

```
def integrate(f, a, b, n):
    h = float(b-a)/n
    s = 0
    for i in range(1, n):
        print 'i=%d, a+i*h=%g' % (i, a+i*h)
        s += f(a + i*h)
    return s
```

Here is the output:

```
i=1, a+i*h=5
Integral of g equals 1
i=1, a+i*h=0.5
Integral of f equals 0.523599 (exact value is 0.570796)
```

There was only one pass in the `i` loop in `integrate`. According to the formula, there should be n passes, i.e., two in this test case. The limits of `i` must be wrong. The limits are produced by the call `range(1,n)`. We recall that such a call results in integers going from 1 up to `n`, but *not* including `n`. We need to include `n` as value of `i`, so the right call to `range` is `range(1,n+1)`.

We make this correction and rerun the program. The output is now

```
i=1, a+i*h=5
i=2, a+i*h=10
Integral of g equals 2
i=1, a+i*h=0.5
i=2, a+i*h=1
Integral of f equals 2.0944 (exact value is 0.570796)
```

The integral of 1 is still not correct. We need more intermediate results!

In our quick hand calculation we knew that $g(x) = 1$ so all the $f(a + (i - \frac{1}{2})h)$ evaluations were rapidly replaced by ones. Let us now compute all the x coordinates $a + (i - \frac{1}{2})h$ that are used in the formula:

$$i = 1: \ a + (i - \frac{1}{2})h = 2.5, \quad i = 2: \ a + (i - \frac{1}{2})h = 7.5\,.$$

Looking at the output from the program, we see that the argument to `g` has a different value - and fortunately we realize that the formula we have coded is wrong. It should be `a+(i-0.5)*h`.

We correct this error and run the program:

```
i=1, a+(i-0.5)*h=2.5
i=2, a+(i-0.5)*h=7.5
Integral of g equals 2
...
```

Still the integral is wrong. At this point you may give up programming, but the more skills you pick up in debugging, the more fun it is to hunt for errors! Debugging is like reading an exciting criminal novel: the detective follows different ideas and tracks, but never gives up before the culprit is caught.

Now we read the code more carefully and compare expressions with those in the mathematical formula. We should, of course, have done this already when writing the program, but it is easy to get excited when writing code and hurry for the end. This ongoing story of debugging probably shows that reading the code carefully can save much debugging time. (Actually, being extremely careful with what you write, and comparing all formulas with the mathematics, may be the best way to get more spare time when taking a programming course!)

We clearly add up all the f evaluations correctly, but then this sum must be multiplied by h, and we forgot that in the code. The `return` statement in `integrate` must therefore be modified to

```
    return s*h
```

Eventually, the output is

```
Integral of g equals 10
Integral of f equals 0.568484 (exact value is 0.570796)
```

and we have managed to integrate a constant function in our program! Even the second integral looks promising!

To judge the result of integrating the inverse sine function, we need to run several increasing n values and see that the approximation gets better. For $n = 2, 10, 100, 1000$ we get 0.550371, 0.568484, 0.570714, 0.570794, to be compared to the exact value 0.570796. (This is not the mathematically exact value, because it involves computations of $\sin^{-1}(x)$, which is only approximately calculated by the `asin` function in the `math` module. However, the approximation error is very small ($\sim 10^{-16}$).) The decreasing error provides evidence for a correct program, but it is not a strong proof. We should try out more functions. In particular, linear functions are integrated exactly by the Midpoint rule. We can also measure the speed of the decrease of the error and check that the speed is consistent with the properties of the Midpoint rule, but this is a mathematically more advanced topic.

The very important lesson learned from these debugging sessions is that you should start with a simple test problem where all formulas can be computed by hand. If you start out with $n = 100$ and try to integrate the inverse sine function, you will have a much harder job with tracking down all the errors.

9. Use a debugger. Another lesson learned from these sessions is that we needed many `print` statements to see intermediate results. It is an open question if it would be more efficient to run a debugger and stop the code at relevant lines. In an edit-and-run cycle of the type we met here, we frequently need to examine many numerical results, correct something, and look at all the intermediate results again. Plain `print` statements are often better suited for this massive output than the pure manual operation of a debugger, unless one writes a program to automate the interaction with the debugger.

The correct code for the implementation of the Midpoint rule is found in `integrate_v2.py`. Some readers might be frightened by all the energy it took to debug this code, but this is just the nature of programming. The experience of developing programs that finally work is very awarding.

People only become computer programmers if they're obsessive about details, crave power over machines, and can bear to be told day after day exactly how stupid they are. Gregory J. E. Rawlins [25], computer scientist.

Refining the user interface. We briefly mentioned that the chosen user interface, where the user can only specify n, is not particularly user friendly. We should allow f, a, b, and n to be specified on the command line. Since f is a function and the command line can only provide strings to the program, we may use the `StringFunction` object from `scitools.std` to convert a string expression for the function to be integrated to an ordinary Python function (see Section 4.3.3). The other parameters should be easy to retrieve from the command line if Section 4.2 is understood. As suggested in Section 4.7, we enclose the input statements in a `try-except` block, here with a specific exception type `IndexError` (because an index in `sys.argv` out of bounds is the only type of error we expect to handle):

```
try:
    f_formula = sys.argv[1]
    a = eval(sys.argv[2])
    b = eval(sys.argv[3])
    n = int(sys.argv[4])
except IndexError:
    print 'Usage: %s f-formula a b n' % sys.argv[0]
    sys.exit(1)
```

Note that the use of `eval` allows us to specify `a` and `b` as `pi` or `exp(5)` or another mathematical expression.

With the input above we can perform the general task of the program:

```
from scitools.std import StringFunction
f = StringFunction(f_formula)
I = integrate(f, a, b, n)
print I
```

Writing a test function. Instead of having these test statements as a main program we follow the good habits of Section 4.9 and make a module with

- the `integrate` function,
- a `test_integrate` function for testing the `integrate` function's ability to exactly integrate linear functions,
- a `main` function for reading data from the command line and calling `integrate` for the user's problem at hand.

Any module should also have a test block, as well as doc strings for the module itself and all functions.

The `test_integrate` function can perform a loop over some specified `n` values and check that the Midpoint rule integrates a linear function exactly. As always, we must be prepared for round-off errors, so "exactly" means errors less than (say) 10^{-14}. The relevant code becomes

```
def test_integrate():
    """Check that linear functions are integrated exactly."""

    def g(x):
        return p*x + q   # general linear function

    def int_g_exact(x):  # integral of g(x)
        return 0.5*p*x**2 + q*x

    a = -1.2; b = 2.8    # "arbitrary" integration limits
    p = -2;   q = 10
    success = True        # True if all tests below are passed
    for n in 1, 10, 100:
        I = integrate(g, a, b, n)
        I_exact = int_g_exact(b) - int_g_exact(a)
        error = abs(I_exact - I)
        if error > 1E-14:
            success = False
    assert success
```

We have followed the programming standard that will make this test function automatically work with the nose test framework:

1. the name of the function starts with `test_`,
2. the function has no arguments,
3. checks of whether a test is passed or not are done with `assert`.

The `assert success` statement raises an `AssertionError` exception if `success` is false, otherwise nothing happens. The nose testing framework searches for functions whose name start with `test_`, execute each function, and record if an `AssertionError` is raised. It is overkill to use nose for small programs, but in larger projects with many functions in many files, nose can run all tests with a short command and write back a notification that all tests passed.

The `main` function is simply a wrapping of the main program given above. The test block may call or `test_integrate` function or `main`, depending on whether the user will test the module or use it:

```
if __name__ == '__main__':
    if sys.argv[1] == 'verify':
        verify()
    else:
        # Compute the integral specified on the command line
        main()
```

Here is a short demo computing $\int_0^{2\pi}(\cos(x)+\sin(x))dx$ with the aid of the `integrate.py` file:

Terminal
```
integrate.py 'cos(x)+sin(x)' 0 2*pi 10
-3.48786849801e-16
```

F.2.3 Getting help from a code analyzer

The tools PyLint[3] and Flake8[4] can analyze your code and point out errors and undesired coding styles. Before point 7 in the lists above, *Run the program*, it can be wise to run PyLint or Flake8 to be informed about problems with the code.

Consider the first version of the `integrate` code, `integrate_v1.py`. Running Flake8 gives

Terminal

```
Terminal> flake8 integrate_v1.py
integrate_v1.py:7:1: E302 expected 2 blank lines, found 1
integrate_v1.py:8:1: E112 expected an indented block
integrate_v1.py:8:7: E901 IndentationError: expected an indented block
integrate_v1.py:10:1: E302 expected 2 blank lines, found 1
integrate_v1.py:11:1: E112 expected an indented block
```

Flake8 checks if the program obeys the official Style Guide for Python Code[5] (known as *PEP8*). One of the rules in this guide is to have two blank lines before functions and classes (a habit that is often dropped in this book to reduce the length of code snippets), and our program breaks the rule before the `f` and `g` functions. More serious and useful is the `expected an indented block` at lines 8 and 11. This error is quickly found anyway by running the programming.

PyLint does not a complete job before the program is free of syntax errors. We must therefore apply it to the `integrate_v2.py` code:

Terminal

```
Terminal> pylint integrate_v2.py
C: 20, 0: Exactly one space required after comma
I = integrate(f, 0, 1,  n)
                     ^ (bad-whitespace)
W: 19, 0: Redefining built-in 'pow' (redefined-builtin)
C:  1, 0: Missing module docstring (missing-docstring)
W:  1,14: Redefining name 'f' from outer scope (line 8)
W:  1,23: Redefining name 'n' from outer scope (line 16)
C:  1, 0: Invalid argument name "f" (invalid-name)
C:  1, 0: Invalid argument name "a" (invalid-name)
```

There is much more output, but let us summarize what PyLint does not like about the code:

1. Extra whitespace (after comma in a call to `integrate`)
2. Missing doc string at the beginning of the file
3. Missing doc strings in the functions
4. Same name `f` used as local variable in `integrate` and global function name in the `f(x)` function
5. Too short variable names: `a`, `b`, `n`, etc.

[3] `http://www.pylint.org/`
[4] `https://flake8.readthedocs.org/en/2.0/`
[5] `http://www.python.org/dev/peps/pep-0008/`

6. “Star import” of the form `from math import *`

In short programs where the one-to-one mapping between mathematical notation and the variable names is very important to make the code self-explanatory, this author thinks that only points 1-3 qualify for attention. Nevertheless, for larger non-mathematical programs all the style violations pointed out are serious and lead to code that is easier to read, debug, maintain, and use.

Running Flak8 on `integrate_v2.py` leads to only three problems: missing two blank lines before functions (not reported by PyLint) and doing `from math import *`. Flake8 complains in general a lot less than PyLint, but both are very useful during program development to readability of the code and remove errors.

Migrating Python to compiled code G

Python is a very convenient language for implementing scientific computations as the code can be made very close to the mathematical algorithms. However, the execution speed of the code is significantly lower than what can be obtained by programming in languages such as Fortran, C, or C++. These languages *compile* the program to machine language, which enables the computing resources to be utilized with very high efficiency. Frequently, and this includes almost all examples in the present book, Python is fast enough. But in the cases where speed really matters, can we increase the efficiency without rewriting the whole program in Fortran, C, or C++? The answer is yes, which will be illustrated through a case study in the forthcoming text.

Fortunately, Python was initially designed for being integrated with C. This feature has spawned the development of several techniques and tools for calling compiled languages from Python, allowing us to relatively easily reuse fast and well-tested scientific libraries in Fortran, C, or C++ from Python, *or* migrate slow Python code to compiled languages. It often turns out that only smaller parts of the code, usually `for` loops doing heavy numerical computations, suffer from low speed and can benefit from being implemented in Fortran, C, or C++.

The primary technique to be advocated here is to use Cython. Cython can be viewed as an extension of the Python language where variables can be declared with a type and other information such that Cython is able to automatically generate special-purpose, fast C code from the Python code. We will show how to utilize Cython and what the computational gain might be.

The present case study starts with stating a computational problem involving statistical simulations, which are known to cause long execution times, especially if accurate results are desired.

G.1 Pure Python code for Monte Carlo simulation

A short, intuitive algorithm in Python is first developed. Then this code is vectorized using functionality of the Numerical Python package. Later sections migrate the algorithm to Cython code and also plain C code for comparison. At the end the various techniques are ranked according to their computational efficiency.

G.1.1 The computational problem

A die is thrown m times. What is the probability of getting six eyes *at least* n times? For example, if $m = 5$ and $n = 3$, this is the same as asking for the probability that three or more out of five dice show six eyes.

The probability can be estimated by Monte Carlo simulation. Chapter 8.3 provides a background for this technique: We simulate the process a large number of times, N, and count how many times, M, the experiment turned out successfully, i.e., when we got at least n out of m dice with six eyes in a throw.

Monte Carlo simulation has traditionally been viewed as a very costly computational method, normally requiring very sophisticated, fast computer implementations in compiled languages. An interesting question is how useful high-level languages like Python and associated tools are for Monte Carlo simulation. This will now be explored.

G.1.2 A scalar Python implementation

Let us introduce the more descriptive variables `ndice` for m and `nsix` for n. The Monte Carlo method is simply a loop, repeated `N` times, where the body of the loop may directly express the problem at hand. Here, we draw `ndice` random integers `r` in $[1, 6]$ inside the loop and count of many (`six`) that equal 6. If `six >= nsix`, the experiment is a success and we increase the counter `M` by one.

A Python function implementing this approach may look as follows:

```
import random

def dice6_py(N, ndice, nsix):
    M = 0                     # no of successful events
    for i in range(N):        # repeat N experiments
        six = 0               # how many dice with six eyes?
        for j in range(ndice):
            r = random.randint(1, 6)  # roll die no. j
            if r == 6:
               six += 1
        if six >= nsix:       # successful event?
            M += 1
    p = float(M)/N
    return p
```

The `float(M)` transformation is important since `M/N` will imply integer division when `M` and `N` both are integers in Python v2.x and many other languages.

We will refer to this implementation is the *plain Python* implementation. Timing the function can be done by:

```
import time
t0 = time.clock()
p = dice6_py(N, ndice, nsix)
t1 = time.clock()
print 'CPU time for loops in Python:', t1-t0
```

The table to appear later shows the performance of this plain, pure Python code relative to other approaches. There is a factor of 30+ to be gained in computational efficiency by reading on.

The function above can be verified by studying the (somewhat simplified) case $m = n$ where the probability becomes 6^{-n}. The probability quickly becomes small with increasing n. For such small probabilities the number of successful events M is small, and M/N will not be a good approximation to the probability unless M is reasonably large, which requires a very large N. For example, with $n = 4$ and $N = 10^5$ the average probability in 25 full Monte Carlo experiments is 0.00078 while the exact answer is 0.00077. With $N = 10^6$ we get the two correct significant digits from the Monte Carlo simulation, but the extra digit costs a factor of 10 in computing resources since the CPU time scales linearly with N.

G.1.3 A vectorized Python implementation

A vectorized version of the previous program consists of replacing the explicit loops in Python by efficient operations on vectors or arrays, using functionality in the Numerical Python (`numpy`) package. Each array operation takes place in C or Fortran and is hence much more efficient than the corresponding loop version in Python.

First, we must generate all the random numbers to be used in one operation, which runs fast since all numbers are then calculated in efficient C code. This is accomplished using the `numpy.random` module. Second, the analysis of the large collection of random numbers must be done by appropriate vector/array operations such that no looping in Python is needed. The solution algorithm must therefore be expressed through a series of function calls to the `numpy` library. Vectorization requires knowledge of the library's functionality and how to assemble the relevant building blocks to an algorithm without operations on individual array elements.

Generation of `ndice` random number of eyes for `N` experiments is performed by

```
import numpy as np
eyes = np.random.random_integers(1, 6, size=(N, ndice))
```

Each row in the `eyes` array corresponds to one Monte Carlo experiment.

The next step is to count the number of successes in each experiment. This counting should not make use of any loop. Instead we can test `eyes == 6` to get a boolean array where an element `i,j` is `True` if throw (or die) number `j` in Monte Carlo experiment number `i` gave six eyes. Summing up the rows in this boolean array (`True` is interpreted as 1 and `False` as 0), we are interested in the rows where the sum is equal to or greater than `nsix`, because the number of such rows equals the number of successful events. The vectorized algorithm can be expressed as

```
def dice6_vec1(N, ndice, nsix):
    eyes = np.random.random_integers(1, 6, size=(N, ndice))
    compare = eyes == 6
    throws_with_6 = np.sum(compare, axis=1)  # sum over columns
    nsuccesses = throws_with_6 >= nsix
    M = np.sum(nsuccesses)
    p = float(M)/N
    return p
```

The use of `np.sum` instead of Python's own `sum` function is essential for the speed of this function: using `M = sum(nsucccesses)` instead slows down the code by a factor of almost 10! We shall refer to the `dice6_vec1` function as the *vectorized Python, version1* implementation.

The criticism against the vectorized version is that the original problem description, which was almost literally turned into Python code in the `dice6_py` function, has now become much more complicated. We have to decode the calls to various `numpy` functionality to actually realize that `dice6_py` and `dice6_vec` correspond to the same mathematics.

Here is another possible vectorized algorithm, which is easier to understand, because we retain the Monte Carlo loop and vectorize only each individual experiment:

```
def dice6_vec2(N, ndice, nsix):
    eyes = np.random.random_integers(1, 6, (N, ndice))
    six = [6 for i in range(ndice)]
    M = 0
    for i in range(N):
        # Check experiment no. i:
        compare = eyes[i,:] == six
        if np.sum(compare) >= nsix:
            M += 1
    p = float(M)/N
    return p
```

We refer to this implementation as *vectorized Python, version 2*. As will be shown later, this implementation is significantly slower than the *plain Python* implementation (!) and very much slower than the *vectorized Python, version 1* approach. A conclusion is that readable, partially vectorized code, may run slower than straightforward scalar code.

G.2 Migrating scalar Python code to Cython

G.2.1 A plain Cython implementation

A Cython program starts with the scalar Python implementation, but all variables are specified with their types, using Cython's variable declaration syntax, like `cdef int M = 0` where we in standard Python just write `M = 0`. Adding such variable declarations in the scalar Python implementation is straightforward:

```
import random

def dice6_cy1(int N, int ndice, int nsix):
    cdef int M = 0            # no of successful events
    cdef int six, r
    cdef double p
    for i in range(N):        # repeat N experiments
        six = 0               # how many dice with six eyes?
        for j in range(ndice):
            r = random.randint(1, 6)  # roll die no. j
            if r == 6:
               six += 1
        if six >= nsix:       # successful event?
            M += 1
    p = float(M)/N
    return p
```

This code must be put in a separate file with extension `.pyx`. Running Cython on this file translates the Cython code to C. Thereafter, the C code must be compiled and linked to form a shared library, which can be imported in Python as a module. All these tasks are normally automated by a `setup.py` script. Let the `dice6_cy1` function above be stored in a file `dice6.pyx`. A proper `setup.py` script looks as follows:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
  name='Monte Carlo simulation',
  ext_modules=[Extension('_dice6_cy', ['dice6.pyx'],)],
  cmdclass={'build_ext': build_ext},
)
```

Running

```
Terminal> python setup.py build_ext --inplace
```

generates the C code and creates a (shared library) file `_dice6_cy.so` (known as a *C extension module*) which can be loaded into Python as a module with name `_dice6_cy`:

```
from _dice6_cy import dice6_cy1
import time
t0 = time.clock()
p = dice6_cy1(N, ndice, nsix)
t1 = time.clock()
print t1 - t0
```

We refer to this implementation as *Cython random.randint*. Although most of the statements in the `dice6_cy1` function are turned into plain and fast C code, the speed is not much improved compared with the original scalar Python code.

To investigate what takes time in this Cython implementation, we can perform a profiling. The template for profiling a Python function whose call syntax is stored in some string `statement`, reads

```
import cProfile, pstats
cProfile.runctx(statement, globals(), locals(), 'tmp_profile.dat')
s = pstats.Stats('tmp_profile.dat')
s.strip_dirs().sort_stats('time').print_stats(30)
```

Data from the profiling are here stored in the file `tmp_profile.dat`. Our interest now is the `dice6_cy1` function so we set

```
statement = 'dice6_cy1(N, ndice, nsix)'
```

In addition, a Cython file in which there are functions we want to profile must start with the line

```
# cython: profile=True
```

to turn on profiling when creating the extension module. The profiling output from the present example looks like

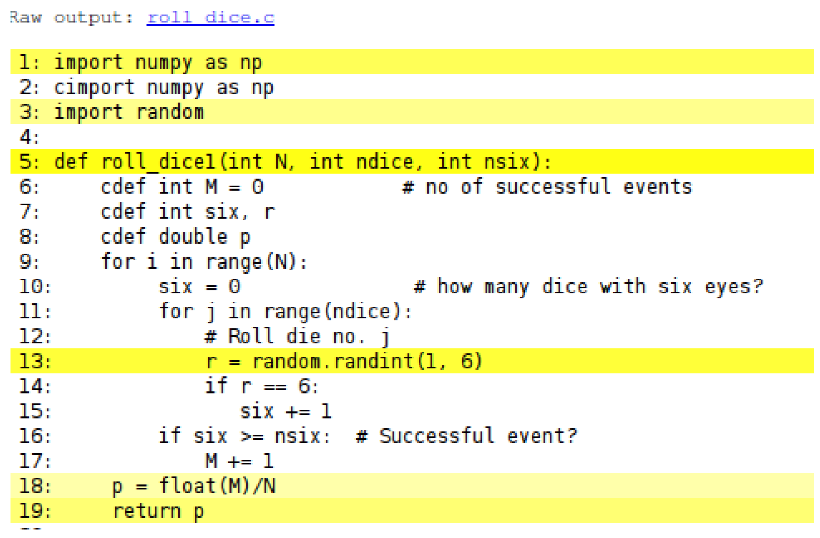
```
         5400004 function calls in 7.525 CPU seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  1800000    4.511    0.000    4.863    0.000 random.py:160(randrange)
  1800000    1.525    0.000    6.388    0.000 random.py:224(randint)
        1    1.137    1.137    7.525    7.525 dice6.pyx:6(dice6_cy1)
  1800000    0.352    0.000    0.352    0.000 {method 'random' ...
        1    0.000    0.000    7.525    7.525 {dice6_cy.dice6_cy1}
```

We easily see that it is the call to `random.randint` that consumes almost all the time. The reason is that the generated C code must call a Python module (`random`), which implies a lot of overhead. The C code should only call plain C functions, or if Python functions *must* be called, they should involve so much computations that the overhead in calling Python from C is negligible.

Instead of profiling the code to uncover inefficient constructs we can generate a visual representation of how the Python code is translated to C. Running

```
Terminal
Terminal> cython -a dice6.pyx
```

creates a file `dice6.html` which can be loaded into a web browser to inspect what Cython has done with the Python code.



Raw output: roll_dice.c

```
import numpy as np
cimport numpy as np
import random

def roll_dice1(int N, int ndice, int nsix):
    cdef int M = 0            # no of successful events
    cdef int six, r
    cdef double p
    for i in range(N):
        six = 0               # how many dice with six eyes?
        for j in range(ndice):
            # Roll die no. j
            r = random.randint(1, 6)
            if r == 6:
                six += 1
        if six >= nsix:  # Successful event?
            M += 1
    p = float(M)/N
    return p
```

White lines indicate that the Python code is translated into C code, while the yellow lines indicate that the generated C code must make calls back to Python (using the Python C API, which implies overhead). Here, the `random.randint` call is in yellow, so this call is not translated to efficient C code.

G.2.2 A better Cython implementation

To speed up the previous Cython code, we have to get rid of the `random.randint` call every time we need a random variable. Either we must call some C function for generating a random variable or we must create a bunch of random numbers simultaneously as we did in the vectorized functions shown above. We first try the latter well-known strategy and apply the `numpy.random` module to generate all the random numbers we need at once:

```
import  numpy as np
cimport numpy as np

@cython.boundscheck(False)  # turn off array bounds check
@cython.wraparound(False)   # turn off negative indices ([-1,-1])
def dice6_cy2(int N, int ndice, int nsix):
    # Use numpy to generate all random numbers
    ...
    cdef np.ndarray[np.int_t, ndim=2, mode='c'] eyes = \
         np.random.random_integers(1, 6, (N, ndice))
```

This code needs some explanation. The `cimport` statement imports a special version of `numpy` for Cython and is needed *after* the standard `numpy` import. The declaration of the array of random numbers could just go as

```
cdef np.ndarray eyes = np.random.random_integers(1, 6, (N, ndice))
```

However, the processing of the `eyes` array will then be slow because Cython does not have enough information about the array. To generate optimal C code, we must provide information on the element types in the array, the number of dimensions of the array, that the array is stored in contiguous memory, that we do not want the overhead of checking whether indices are within their bounds or not, and that we do not need negative indices (which slows down array indexing). The latter two properties are taken care of by the `@cython.boundscheck(False)` and the `@cython.wraparound(False)` statements (decorators) right before the function, respectively, while the rest of the information is specified within square brackets in the `cdef np.ndarray` declaration. Inside the brackets, `np.int_t` denotes integer array elements (`np.int` is the usual data type object, but `np.int_t` is a Cython precompiled version of this object), `ndim=2` tells that the array has two dimensions (indices), and `mode='c'` indicates contiguous storage of the array. With all this extra information, Cython can generate C code that works with `numpy` arrays as efficiently as native C arrays.

The rest of the code is a plain copy of the `dice6_py` function, but with the `random.randint` call replaced by an array look-up `eyes[i,j]` to retrieve the next random number. The two loops will now be as efficient as if they were coded directly in pure C.

The complete code for the efficient version of the `dice6_cy1` function looks as follows:

```
import  numpy as np
cimport numpy as np
import cython
@cython.boundscheck(False)  # turn off array bounds check
@cython.wraparound(False)   # turn off negative indices ([-1,-1])
def dice6_cy2(int N, int ndice, int nsix):
    # Use numpy to generate all random numbers
    cdef int M = 0            # no of successful events
    cdef int six, r
    cdef double p
    cdef np.ndarray[np.int_t, ndim=2, mode='c'] eyes = \
         np.random.random_integers(1, 6, (N, ndice))
    for i in range(N):
        six = 0               # how many dice with six eyes?
        for j in range(ndice):
            r = eyes[i,j]     # roll die no. j
            if r == 6:
               six += 1
        if six >= nsix:       # successful event?
            M += 1
    p = float(M)/N
    return p
```

This Cython implementation is named *Cython numpy.random*.

The disadvantage with the `dice6_cy2` function is that large simulations (large `N`) also require large amounts of memory, which usually limits the possibility for high accuracy much more than the CPU time. It would be advantageous to have a fast random number generator a la `random.randint` in C. The C library `stdlib` has a generator of random integers, `rand()`, generating numbers from 0 to up `RAND_MAX`. Both the `rand` function and the `RAND_MAX` integer are easy to access in a Cython program:

```
from libc.stdlib cimport rand, RAND_MAX

r = 1 + int(6.0*rand()/RAND_MAX) # random integer 1,...,6
```

Note that `rand()` returns an integer so we must avoid integer division by ensuring that the denominator is a real number. We also need to explicitly convert the resulting real fraction to `int` since `r` is declared as `int`.

With this way of generating random numbers we can create a version of `dice6_cy1` that is as fast as `dice6_cy`, but avoids all the memory demands and the somewhat complicated array declarations of the latter:

```
from libc.stdlib cimport rand, RAND_MAX
def dice6_cy3(int N, int ndice, int nsix):
    cdef int M = 0            # no of successful events
    cdef int six, r
    cdef double p
    for i in range(N):
        six = 0               # how many dice with six eyes?
        for j in range(ndice):
            # Roll die no. j
            r = 1 + int(6.0*rand()/RAND_MAX)
            if r == 6:
               six += 1
        if six >= nsix:       # successful event?
            M += 1
    p = float(M)/N
    return p
```

This final Cython implementation will be referred to as *Cython stdlib.rand*.

G.3 Migrating code to C

G.3.1 Writing a C program

A natural next improvement would be to program the Monte Carlo simulation loops directly in a compiled programming language, which guarantees optimal speed. Here we choose the C programming language for this purpose. The C version of our `dice6` function and an associated main program take the form

```
#include <stdio.h>
#include <stdlib.h>

double dice6(int N, int ndice, int nsix)
{
  int M = 0;
```

```
    int six, r, i, j;
    double p;

    for (i = 0; i < N; i++) {
      six = 0;
      for (j = 0; j < ndice; j++) {
        r = 1 + rand()/(RAND_MAX*6.0); /* roll die no. j */
        if (r == 6)
          six += 1;
      }
      if (six >= nsix)
        M += 1;
    }
    p = ((double) M)/N;
    return p;
  }

  int main(int nargs, const char* argv[])
  {
    int N = atoi(argv[1]);
    int ndice = 6;
    int nsix = 3;
    double p = dice6(N, ndice, nsix);
    printf("C code: N=%d, p=%.6f\n", N, p);
    return 0;
  }
```

This code is placed in a file `dice6_c.c`. The file can typically be compiled and run by

Terminal

```
Terminal> gcc -O3 -o dice6.capp dice6_c.c
Terminal> ./dice6.capp 1000000
```

This solution is later referred to as *C program.*

G.3.2 Migrating loops to C code via F2PY

Instead of programming the whole application in C, we may consider migrating the loops to the C function `dice6` shown above and then have the rest of the program (essentially the calling main program) in Python. This is a convenient solution if we were to do many other, less CPU time critical things for convenience in Python.

There are many alternative techniques for calling C functions from Python. Here we shall explain two. The first applies the program `f2py` to generate the necessary code that glues Python and C. The `f2py` program was actually made for gluing Python and Fortran, but it can work with C too. We need a specification of the C function to call in terms of a Fortran 90 module. Such a module can be written by hand, but `f2py` can also generate it. To this end, we make a Fortran file `dice6_c_signature.f` with the signature of the C function written in Fortran 77 syntax with some annotations:

```
      real*8 function dice6(n, ndice, nsix)
Cf2py intent(c) dice6
      integer n, ndice, nsix
Cf2py intent(c) n, ndice, nsix
      return
      end
```

The annotations `intent(c)` are necessary to tell `f2py` that the Fortran variables are to be treated as plain C variables and not as pointers (which is the default interpretation of variables in Fortran). The `C2fpy` are special comment lines that `f2py` recognizes, and these lines are used to provide extra information to `f2py` which have no meaning in plain Fortran 77.

We must run `f2py` to generate a `.pyf` file with a Fortran 90 module specification of the C function to call:

Terminal

```
Terminal> f2py -m _dice6_c1 -h dice6_c.pyf \
          dice6_c_signature.f
```

Here `_dice6_c1` is the name of the module with the C function that is to be imported in Python, and `dice6_c.pyf` is the name of the Fortran 90 module file to be generated. Programmers who know Fortran 90 may want to write the `dice6_c.pyf` file by hand.

The next step is to use the information in `dice6_c.pyf` to generate a (C extension) module `_dice6_c1`. Fortunately, `f2py` generates the necessary code, and compiles and links the relevant files, to form a shared library file `_dice6_c1.so`, by a short command:

Terminal

```
Terminal> f2py -c dice6_c.pyf dice6_c.c
```

We can now test the module:

```
>>> import _dice6_c1
>>> print dir(_dice6_c1)  # module contents
['__doc__', '__file__', '__name__', '__package__',
 '__version__', 'dice6']
>>> print _dice6_c1.dice6.__doc__
dice6 - Function signature:
  dice6 = dice6(n,ndice,nsix)
Required arguments:
  n : input int
  ndice : input int
  nsix : input int
Return objects:
  dice6 : float
>>> _dice6_c1.dice6(N=1000, ndice=4, nsix=2)
0.145
```

The method of calling the C function `dice6` via an `f2py` generated module is referred to as *C via f2py*.

G.3.3 Migrating loops to C code via Cython

The Cython tool can also be used to call C code, not only generating C code from the Cython language. Our C code is in the file `dice6_c.c`, but for Cython to see this code we need to create a *header file* `dice6_c.h` listing the definition of the function(s) we want to call from Python. The header file takes the form

```
extern double dice6(int N, int ndice, int nsix);
```

The next step is to make a `.pyx` file with a definition of the C function from the header file and a Python function that calls the C function:

```
cdef extern from "dice6_c.h":
    double dice6(int N, int ndice, int nsix)

def dice6_cwrap(int N, int ndice, int nsix):
    return dice6(N, ndice, nsix)
```

Cython must use this file, named `dice6_cwrap.pyx`, to generate C code, which is to be compiled and linked with the `dice6_c.c` code. All this is accomplished in a `setup.py` script:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

sources = ['dice6_cwrap.pyx', 'dice6_c.c']

setup(
  name='Monte Carlo simulation',
  ext_modules=[Extension('_dice6_c2', sources)],
  cmdclass={'build_ext': build_ext},
)
```

This `setup.py` script is run as

```
Terminal
Terminal> python setup.py build_ext --inplace
```

resulting in a shared library file `_dice6_c2.so`, which can be loaded into Python as a module:

```
>>> import _dice6_c2
>>> print dir(_dice6_c2)
['__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '__test__', 'dice6_cwrap']
```

We see that the module contains the function `dice6_cwrap`, which was made to call the underlying C function `dice6`.

G.3.4 Comparing efficiency

All the files corresponding to the various techniques described above are available in the directory `src/cython`. A file `make.sh` performs all the compilations, while `compare.py` runs all methods and prints out the CPU time required by each method, normalized by the fastest approach. The results for $N = 450,000$ are listed below (MacBook Air running Ubuntu in a VMWare Fusion virtual machine).

Method	Timing
C program	1.0
Cython stdlib.rand	1.2
Cython numpy.random	1.2
C via f2py	1.2
C via Cython	1.2
vectorized Python, version 1	1.9
Cython random.randint	33.6
plain Python	37.7
vectorized Python, version 2	105.0

The CPU time of the plain Python version was 10 s, which is reasonably fast for obtaining a fairly accurate result in this problem. The lesson learned is therefore that a Monte Carlo simulation can be implemented in plain Python first. If more speed is needed, one can just add type information and create a Cython code. Studying the HTML file with what Cython manages to translate to C may give hints about how successful the Cython code is and point to optimizations, like avoiding the call to `random.randint` in the present case. Optimal Cython code runs here at approximately the same speed as calling a handwritten C function with the time-consuming loops. It is to be noticed that the stand-alone C program here ran faster than calling C from Python, probably because the amount of calculations is not large enough to make the overhead of calling C negligible.

Vectorized Python do give a great speed-up compared to plain loops in Python, if done correctly, but the efficiency is not on par with Cython or handwritten C. Even more important is the fact that vectorized code is not at all as readable as the algorithm expressed in plain Python, Cython, or C. Cython therefore provides a very attractive combination of readability, ease of programming, and high speed.

H Technical topics

H.1 Getting access to Python

A comprehensive eco system for scientific computing with Python used to be quite a challenge to install on a computer, especially for newcomers. This problem is more or less solved today. There are several options for getting easy access to Python and the most important packages for scientific computations, so the biggest issue for a newcomer is to make a proper choice. An overview of the possibilities together with my own recommendations appears next.

H.1.1 Required software

The strictly required software packages for working with this book are

- Python[1] version 2.7 [24]
- Numerical Python[2] (NumPy) [21, 20] for array computing
- Matplotlib[3] [10, 9] for plotting

Desired add-on packages are

- IPython[4] [23, 22] for interactive computing
- SciTools[5] [15] for add-ons to NumPy
- ScientificPython[6] [8] for add-ons to NumPy
- nose[7] for testing programs

[1] `http://python.org`
[2] `http://www.numpy.org`
[3] `http://matplotlib.org`
[4] `http://ipython.org`
[5] `http://code.google.com/p/scitools`
[6] `http://starship.python.net/crew/hinsen`
[7] `https://nose.readthedocs.org`

- pip[8] for installing Python packages
- Cython[9] for compiling Python to C
- SymPy[10] [2] for symbolic mathematics
- SciPy[11] [11] for advanced scientific computing

There are different ways to get access to Python with the required packages:

1. Use a computer system at an institution where the software is installed. Such a system can also be used from your local laptop through remote login over a network.
2. Install the software on your own laptop.
3. Use a web service.

A system administrator can take the list of software packages and install the missing ones on a computer system. For the two other options, detailed descriptions are given below.

Using a web service is very straightforward, but has the disadvantage that you are constrained by the packages that are allowed to install on the service. There are services at the time of this writing that suffice for working with most of this book, but if you are going to solve more complicated mathematical problems, you will need more sophisticated mathematical Python packages, more storage and more computer resources, and then you will benefit greatly from having Python installed on your own computer.

This author's experience is that installation of mathematical software on personal computers quickly becomes a technical challenge. Linux Ubuntu (or any Debian-based Linux version) contains the largest repository today of pre-built mathematical software and makes the installation trivial without any need for particular competence. Despite the user-friendliness of the Mac and Windows systems, getting sophisticated mathematical software to work on these platforms requires considerable competence.

H.1.2 Installing software on your laptop: Mac OS X and Windows

There are various possibilities for installing the software on a Mac OS X or Windows platform:

1. Use `.dmg` (Mac) or `.exe` (Windows) files to install individual packages
2. Use Homebrew or MacPorts to install packages (Mac only)

[8] `http://www.pip-installer.org`
[9] `http://cython.org`
[10] `http://sympy.org`
[11] `http://scipy.org`

3. Use a pre-built environment for scientific computing in Python:
    - Anaconda[12]
    - Enthought Canopy[13]
4. Use a virtual machine running Ubuntu:
    - VMWare Fusion[14]
    - VirtualBox[15]
    - Vagrant[16]

Alternative 1 is the obvious and perhaps simplest approach, but usually requires quite some competence about the operating system as a long-term solution when you need many more Python packages than the basic three. This author is not particularly enthusiastic about Alternative 2. If you anticipate to use Python extensively in your work, I strongly recommend operating Python on an Ubuntu platform and going for Alternative 4 because that is the easiest and most flexible way to build and maintain your own software ecosystem. Alternative 3 is recommended for those who are uncertain about the future needs for Python and think Alternative 4 is too complicated. My preference is to use Anaconda for the Python installation and Spyder[17] (comes with Anaconda) as a graphical interface with editor, an output area, and flexible ways of running Python programs.

H.1.3 Anaconda and Spyder

Anaconda can be downloaded for free from `http://continuum.io/downloads`. The Integrated Development Environment (IDE) Spyder is included with Anaconda and is my recommended tool for writing and running Python programs on Mac and Windows, unless you have preference for a plain text editor for writing programs and a terminal window for running them.

Spyder on Mac. Spyder is started by typing `spyder` in a (new) Terminal application. If you get an error message *unknown locale*, you need to type the following line in the Terminal application, or preferably put the line in your `$HOME/.bashrc` Unix initialization file:

```
export LANG=en_US.UTF-8; export LC_ALL=en_US.UTF-8
```

[12] http://continuum.io/downloads/
[13] https://www.enthought.com/products/canopy/
[14] http://www.vmware.com/products/fusion
[15] https://www.virtualbox.org/
[16] http://www.vagrantup.com/
[17] https://code.google.com/p/spyderlib/

Installation of additional packages. Anaconda installs the pip tool that is handy to install additional packages. In a Terminal application on Mac or in a PowerShell terminal on Windows, write

Terminal
```
pip install --user packagename
```

Installing SciTools on Mac. The SciTools package can be installed by pip on Mac:

Terminal
```
Terminal> pip install mercurial
Terminal> pip install --user -e \
          hg+https://code.google.com/p/scitools#egg=scitools
```

Installing SciTools on Windows. The safest procedure on Windows is to go to `http://code.google.com/p/scitools/source/browse` and download the zip file. Double-click on the downloaded file in Windows Explorer and click on *Extract all files* to create a new folder with all the SciTools files. Find the location of this folder, open a PowerShell window, and move to the location, e.g.,

Terminal
```
Terminal> cd C:\Users\username\Downloads\scitools-2db3cbb5076a
```

Installation is done by

Terminal
```
Terminal> python setup.py install
```

H.1.4 VMWare Fusion virtual machine

A virtual machine allows you to run another complete computer system in a separate window. For Mac users, I recommend VMWare Fusion over VirtualBox for running a Linux (or Windows) virtual machine. (VMWare Fusion's hardware integration seems superior to that of VirtualBox.) VMWare Fusion is commercial software, but there is a free trial version you can start with. Alternatively, you can use the simpler VMWare Player, which is free for personal use.

Installing Ubuntu. The following recipe will install a Ubuntu virtual machine under VMWare Fusion.

1. Download Ubuntu[18]. Choose a version that is compatible with your computer, usually a 64-bit version nowadays.

[18] `http://www.ubuntu.com/desktop/get-ubuntu/download`

2. Launch VMWare fusion.
3. Click on *File - New* and choose to *install Windows or another operating system in a new virtual machine*.
4. Double-click on *Use operating system installation disc or image*.
5. Choose *Linux* and *Ubuntu 64 bit*. You will be asked to give provide the name of the Ubuntu file you downloaded.
6. Choose *Customize Settings* and make the following settings (these settings can be changed later, if desired):
   - *Processors and Memory*: Set a minimum of 2 Gb memory, but not more than half of your computer's total memory. The virtual machine can use all processors.
   - *Hard Disk*: Choose how much disk space you want to use inside the virtual machine (20 Gb is considered a minimum).
7. Choose where you want to store the Ubuntu iso file on the hard disk. The Ubuntu iso file is a single file containing all of the Ubuntu operating system and all of the associated file system. Backing up this file means backing up your whole Ubuntu installation, something you should frequently do!
8. When the start-up of Ubuntu is finished, click on the Ubuntu window to launch the Install application. Click *Install*. On the next page, fill in *Download updates while installing* and *Install third-party software*. Then click *Continue*. On the next page, choose *Erase disk and install Ubuntu*, then provide your location, choose a keyboard layout (English is recommended for programmers, but you can easily set up Ubuntu to switch between different keyboards by means of a short-key). Fill in name, etc. Ubuntu will then eventually install the operating system.
9. You may need to define a higher resolution of the display. Find the *System settings* icon on the left, go to *Display*, choose some display (you can try several, click *Keep this configuration* when you are satisfied).
10. You can have multiple keyboards on Ubuntu. Launch *System settings*, go to *Keyboard*, click the *Text entry* hyperlink, add keyboard(s) (*Input sources to use*), and choose a shortcut, say `Ctrl+space` or `Ctrl+backslash`, in the *Switch to next source using* field. Then you can use the shortcut to quickly switch keyboard.

Installing software on Ubuntu. You now have a full Ubuntu machine, but there is not much software on a it. Installation is performed through the Ubuntu Software Center (a graphical application) or through Unix commands, typically

Terminal

```
Terminal> sudo apt-get install packagename
```

To look up the right package name, run `apt-cache search` followed by typical words of that package. The strength of the `apt-get` way of installing software is that the package *and all packages it depends on* are automatically installed through the `apt-get install` command. This is in a nutshell why Ubuntu (or Debian-based Linux systems) are so user-friendly for installing sophisticated mathematical software.

To install a lot of useful packages for scientific work, go to `http://goo.gl/RVHixr` and click on *one* of the following files, which will install a collection of software for scientific work using `apt-get`:

- `install_minimal.sh`: install a minimal collection (recommended)
- `install_rich.sh`: install a rich collection (takes time to run)

Then click the *Raw* button. The file comes up in the browser window, right-click and choose *Save As...* to save the file on your computer. The next step is to find the file and run it:

```
Terminal
Terminal> cd ~/Downloads
Terminal> bash install_minimal.sh
```

The program will run for quite some time, hopefully without problems. If it stops, set a comment sign `#` in front of the line where it stopped and rerun.

File sharing. The Ubuntu machine can see the files on your host system if you download *VMWare Tools*. Go to the *Virtual Machine* pull-down menu in VMWare Fusion and choose *Install VMWare Tools*. A tarfile is downloaded. Click on it and it will open a folder `vmware-tools-distrib`, normally in your home folder. Move to the new folder and run `sudo perl vmware-install.pl`. You can go with the default answers to all the questions.

On a Mac, you must open *Virtual Machine - Settings...* and choose *Sharing* to bring up a dialog where you can add the folders you want to be visible in Ubuntu. Just choose your home folder. Then turn on the file sharing button (or turn off and on again). Go to Ubuntu and check if you can see all your host system's files in `/mnt/hgfs/`.

If you later detect that `/mnt/hgfs/` folder has become empty, VMWare Tools must be reinstalled by first turning shared folders off, and then running

```
Terminal
Terminal> sudo /usr/bin/vmware-config-tools.pl
```

Occasionally it is necessary to do a full reinstall by `sudo perl vmware-install.pl` as above.

Backup of a VMWare virtual machine on a Mac

The entire Ubuntu machine is a folder on the host computer, typically with a name like `Documents/Virtual Machines/Ubuntu 64-bit`. Backing up the Ubuntu machine means backing up this folder. However, if you use tools like Time Machine and work in Ubuntu during backup, the copy of the state of the Ubuntu machine is likely to be corrupt. You are therefore strongly recommended to shut down the virtual machine prior to running Time Machine or simply copying the folder with the virtual machine to some backup disk.

If something happens to your virtual machine, it is usually a straightforward task to make a new machine and import data and software automatically from the previous machine.

H.1.5 Dual boot on Windows

Instead of running Ubuntu in a virtual machine, Windows users also have the option of deciding on the operating system when turning on the machine (so-called dual boot). The Wubi[19] tool makes it very easy to get Ubuntu on a Windows machine this way. There are problems with Wubi on Windows 8, see instructions[20] for how to get around them. It is also relatively straightforward to perform a direct install of Ubuntu by downloading an Ubuntu image, creating a bootable USB stick on Windows[21] or Mac[22], restarting the machine and finally installing Ubuntu[23]. However, with the powerful computers we now have, a virtual machine is more flexible since you can switch between Windows and Ubuntu as easily as going from one window to another.

H.1.6 Vagrant virtual machine

A vagrant machine is different from a standard virtual machine in that it is run in a terminal window on a Mac or Windows computer. You will write programs in Mac/Windows, but run them inside a Vagrant Ubuntu machine that can work with your files and folders on Mac/Windows. This is a bit simpler technology than a full VMWare Fusion virtual machine, as described above, and allows you to work in your original operating system. There is need to install VirtualBox and Vagrant, and

[19] `http://wubi-installer.org`
[20] `https://www.youtube.com/watch?v=gZqsXAoLBDI`
[21] `http://www.ubuntu.com/download/desktop/create-a-usb-stick-on-windows`
[22] `http://www.ubuntu.com/download/desktop/create-a-usb-stick-on-mac-osx`
[23] `http://www.ubuntu.com/download/desktop/install-ubuntu-desktop`

on Windows also Cygwin. Then you can download a Vagrant machine with Ubuntu and either fill it with software as explained above, or you can download a ready-made machine. A special machine[24] has been made for this book. We also have a larger and richer machine[25].

H.1.7 How to write and run a Python program

You have basically three choices to develop and test a Python program:

1. use the IPython notebook
2. use an Integrated Development Environment (IDE), like Spyder, which offers a window with a text editor and functionality to run programs and observe the output
3. use a text editor and a terminal window

The IPython notebook is briefly descried in Section H.1.9, while the other two options are outlined below.

The need for a text editor. Since programs consist of plain text, we need to write this text with the help of another program that can store the text in a file. You have most likely extensive experience with writing text on a computer, but for writing your own programs you need special programs, called *editors*, which preserve exactly the characters you type. The widespread word processors, Microsoft Word being a primary example, are aimed at producing nice-looking reports. These programs *format* the text and are *not* acceptable tools for writing your own programs, even though they can save the document in a pure text format. Spaces are often important in Python programs, and *editors* for plain text give you complete control of the spaces and all other characters in the program file.

Spyder. Spyder is graphical application for developing and running Python programs, available on all major platforms. Spyder comes with Anaconda and some other pre-built environments for scientific computing with Python. On Ubuntu it is conveniently installed by `sudo apt-get install spyder`.

The left part of the Spyder window contains a plain text editor. Click in this window and write `print 'Hello!'` and return. Choose *Run* from the *Run* pull-down menu, and observe the output `Hello!` in the lower right window where the output from programs is visible.

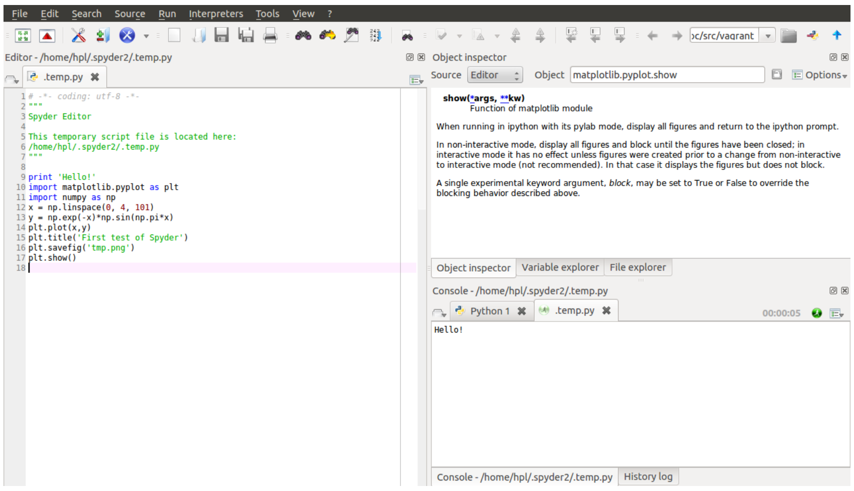
You may continue with more advanced statements involving graphics:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 4, 101)
```

[24] http://goo.gl/hrdhGt
[25] http://goo.gl/uu5Kts

```
y = np.exp(-x)*np.sin(np.pi*x)
plt.plot(x,y)
plt.title('First test of Spyder')
plt.savefig('tmp.png')
plt.show()
```

Choosing *Run* - *Run* now leads to a separate window with a plot of the function $e^{-x}\sin(\pi x)$. Figure H.1 shows how the Spyder application may look like.



```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This temporary script file is located here:
/home/hpl/.spyder2/.temp.py
"""

print 'Hello!'
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 4, 101)
y = np.exp(-x)*np.sin(np.pi*x)
plt.plot(x,y)
plt.title('First test of Spyder')
plt.savefig('tmp.png')
plt.show()

```

show(*args, **kw)
Function of matplotlib module

When running in ipython with its pylab mode, display all figures and return to the ipython prompt.

In non-interactive mode, display all figures and block until the figures have been closed; in interactive mode it has no effect unless figures were created prior to a change from non-interactive to interactive mode (not recommended). In that case it displays the figures but does not block.

A single experimental keyword argument, *block*, may be set to True or False to override the blocking behavior described above.

```
Hello!
```

Fig. H.1 The Spyder Integrated Development Environment.

The plot file we generate in the above program, `tmp.png`, is by default found in the Spyder folder listed in the default text in the top of the program. You can choose *Run* - *Configure ...* to change this folder as desired. The program you write is written to a file `.temp.py` in the same default folder, but any name and folder can be specified in the standard *File* - *Save as...* menu.

A convenient feature of Spyder is that the upper right window continuously displays documentation of the statements you write in the editor to the left.

Text editors. The most widely used editors for writing programs are Emacs and Vim, which are available on all major platforms. Some simpler alternatives for beginners are

- Linux: Gedit
- Mac OS X: TextWrangler
- Windows: Notepad++

We may mention that Python comes with an editor called Idle, which can be used to write programs on all three platforms, but running the program with command-line arguments is a bit complicated for beginners in Idle so Idle is not my favorite recommendation.

Gedit is a standard program on Linux platforms, but all other editors must installed in your system. This is easy: just google for the name, download the file, and follow the standard procedure for installation. All of the mentioned editors come with a graphical user interface that is intuitive to use, but the major popularity of Emacs and Vim is due to their rich set of short-keys so that you can avoid using the mouse and consequently edit at higher speed.

Terminal windows. To run the Python program, you need a *terminal window*. This is a window where you can issue Unix commands in Linux and Mac OS X systems and DOS commands in Windows. On a Linux computer, `gnome-terminal` is my favorite, but other choices work equally well, such as `xterm` and `konsole`. On a Mac computer, launch the application *Utilities - Terminal*. On Windows, launch *PowerShell*.

You must first move to the right folder using the `cd foldername` command. Then running a python program `prog.py` is a matter of writing `python prog.py`. Whatever the program prints can be seen in the terminal window.

Using a plain text editor and a terminal window.

1. Create a folder where your Python programs can be located, say with name `mytest` under your home folder. This is most conveniently done in the terminal window since you need to use this window anyway to run the program. The command for creating a new folder is `mkdir mytest`.
2. Move to the new folder: `cd mytest`.
3. Start the editor of your choice.
4. Write a program in the editor, e.g., just the line `print 'Hello!'`. Save the program under the name `myprog1.py` in the `mytest` folder.
5. Move to the terminal window and write `python myprog1.py`. You should see the word `Hello!` being printed in the window.

H.1.8 The SageMathCloud and Wakari web services

You can avoid installing Python on your machine completely by using a web service that allows you to write and run Python programs. Computational science projects will normally require some kind of visualization and associated graphics packages, which is not possible unless the service offers IPython notebooks. There are two excellent web services with notebooks: *SageMathCloud* at `https://cloud.sagemath.com/` and *Wakari* at `https://www.wakari.io/wakari`. At both sites you must create an account before you can write notebooks in the web browser and download them to your own computer.

Basic intro to SageMathCloud. Sign in, click on *New Project*, give a title to your project and decide whether it should be private or public, click on the project when it appears in the browser, and click on *Create or Import a File, Worksheet, Terminal or Directory...*. If your Python program needs graphics, you need to choose *IPython Notebook*, otherwise you can choose *File*. Write the name of the file above the row of buttons. Assuming we do not need any graphics, we create a plain Python file, say with name `py1.py`. By clicking *File* you are brought to a browser window with a text editor where you can write Python code. Write some code and click *Save*. To run the program, click on the plus icon (*New*), choose *Terminal*, and you have a plain Unix terminal window where you can write `python py1.py` to run the program. Tabs over the terminal (or editor) window make it easy to jump between the editor and the terminal. To download the file, click on *Files*, point on the relevant line with the file, and a download icon appears to the very right. The IPython notebook option works much in the same way, see Section H.1.9.

Basic intro to Wakari. After having logged in at the `wakari.io` site, you automatically enter an IPython notebook with a short introduction to how the notebook can be used. Click on the *New Notebook* button to start a new notebook. Wakari enables creating and editing plain Python files too: click on the *Add file* icon in pane to the left, fill in the program name, and you enter an editor where you can write a program. Pressing *Execute* launches an IPython session in a terminal window, where you can run the program by `run prog.py` if `prog.py` is the name of the program. To download the file, select `test2.py` in the left pane and click on the *Download file* icon.

There is a pull-down menu where you can choose what type of terminal window you want: a plain Unix shell, an IPython shell, or an IPython shell with Matplotlib for plotting. Using the latter, you can run plain Python programs or commands with graphics. Just choose the type of terminal and click on *+Tab* to make a new terminal window of the chosen type.

Installing your own Python packages. Both SageMathCloud and Wakari let you install your own Python packages. To install any package `packagename` available at PyPi[26], run

Terminal
```
pip install --user packagename
```

To install the SciTools package, which is useful when working with this book, create a Terminal (with a Unix shell) and run the command

Terminal
```
pip install --user -e \
    hg+https://code.google.com/p/scitools#egg=scitools
```

[26] `https://pypi.python.org/pypi`

H.1.9 Writing IPython notebooks

The IPython notebook is a splendid interactive tool for doing science, but it can also be used as a platform for developing Python code. You can either run it locally on your computer or in a web service like SageMathCloud or Wakari. Installation on your computer is trivial on Ubuntu, just `sudo apt-get install ipython-notebook`, and also on Windows and Mac[27] by using Anaconda or Enthought Canopy for the Python installation.

The interface to the notebook is a web browser: you write all the code and see all the results in the browser window. There are excellent YouTube videos on how to use the IPython notebook, so here we provide a very quick "step zero" to get anyone started.

A simple program in the notebook. Start the IPython notebook locally by the command `ipython notebook` or go to SageMathCloud or Wakari as described above. The default input area is a *cell* for Python code. Type

```
g = 9.81
v0 = 5
t = 0.6
y = v0*t - 0.5*g*t**2
```

in a cell and *run the cell* by clicking on *Run Selected* (notebook running locally on your machine) or on the "play" button (notebook running in the cloud). This action will execute the Python code and initialize the variables `g`, `v0`, `t`, and `y`. You can then write `print y` in a new cell, execute that cell, and see the output of this statement in the browser. It is easy to go back to a cell, edit the code, and re-execute it.

To download the notebook to your computer, choose the *File - Download as* menu and select the type of file to be downloaded: the original notebook format (`.ipynb` file extension) or a plain Python program version of the notebook (`.py` file extension).

Mixing text, mathematics, code, and graphics. The real strength of IPython notebooks arises when you want to write a report to document how a problem can be explored and solved. As a teaser, open a new notebook, click in the first cell, and choose *Markdown* as format (notebook running locally) or switch from *Code* to *Markdown* in the pull-down menu (notebook in the cloud). The cell is now a text field where you can write text with Markdown[28] syntax. Mathematics can be entered as LaTeX code. Try some text with inline mathematics and an equation on a separate line:

```
Plot the curve $y=f(x)$, where

$$
```

[27] http://ipython.org/install.html

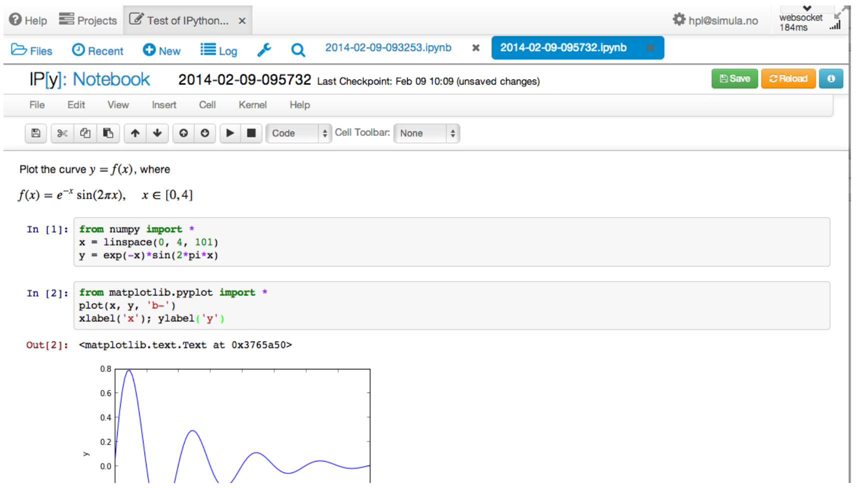
[28] http://daringfireball.net/projects/markdown/syntax

```
    f(x) = e^{-x}\sin (2\pi x),\quad x\in [0, 4]
    $$
```

Execute the cell and you will see nicely typeset mathematics in the browser. In the new cell, add some code to plot $f(x)$:

```
from numpy import *
x = linspace(0, 4, 101)
y = exp(-x)*sin(2*pi*x)
from matplotlib.pyplot import *
plot(x, y, 'b-')
xlabel('x'); ylabel('y')
```

Executing these statements results in a plot in the browser, see Figure H.2. The import statements can actually be dropped since functions from `numpy` and `matplotlib` are imported by default when running the notebook in the browser or by supplying the command-line argument `-pylab` when starting notebooks locally on your machine.



2014-02-09-095732 Last Checkpoint: Feb 09 10:09 (unsaved changes)

Cell Toolbar: None

Plot the curve $y = f(x)$, where

$f(x) = e^{-x}\sin(2\pi x), \quad x \in [0,4]$

```
In [1]: from numpy import *
        x = linspace(0, 4, 101)
        y = exp(-x)*sin(2*pi*x)

In [2]: from matplotlib.pyplot import *
        plot(x, y, 'b-')
        xlabel('x'); ylabel('y')

Out[2]: <matplotlib.text.Text at 0x3765a50>
```

Fig. H.2 Example on an IPython notebook.

H.2 Different ways of running Python programs

Python programs are compiled and interpreted by another program called `python`. To run a Python program, you need to tell the operating system that your program is to be interpreted by the `python` program. This section explains various ways of doing this.

H.2.1 Executing Python programs in iPython

The simplest and most flexible way of executing a Python program is to run it inside IPython. See Section 1.5.3 for a quick introduction to IPython. You start IPython either by the command `ipython` in a terminal window, or by double-clicking the IPython program icon (on Windows). Then, inside IPython, you can run a program `prog.py` by

```
In [1]: run prog.py arg1 arg2
```

where `arg1` and `arg2` are command-line arguments.

This method of running Python programs works the same way on all platforms. One additional advantage of running programs under IPython is that you can automatically enter the Python debugger if an exception is raised (see Section F.1). Although we advocate running Python programs under IPython in this book, you can also run them directly under specific operating systems. This is explained next for Unix, Windows, and Mac OS X.

H.2.2 Executing Python programs in Unix

There are two ways of executing a Python program `prog.py` in Unix-based systems. The first explicitly tells which Python interpreter to use:

Terminal

```
Unix> python prog.py arg1 arg2
```

Here, `arg1` and `arg2` are command-line arguments.

There may be many Python interpreters on your computer system, usually corresponding to different versions of Python or different sets of additional packages and modules. The Python interpreter (`python`) used in the command above is the first program with the name `python` appearing in the folders listed in your `PATH` environment variable. A specific `python` interpreter, say in `/home/hpl/local/bin`, can easily be used as default choice by putting this folder name first in the `PATH` variable. `PATH` is normally controlled in the `.bashrc` file. Alternatively, we may specify the interpreter's complete file path when running `prog.py`:

Terminal

```
Unix> /home/hpl/bin/python prog.py arg1 arg2
```

The other way of executing Python programs in Unix consists of just writing the name of the file:

Terminal

```
Unix> ./prog.py arg1 arg2
```

The leading `./` is needed to tell that the program is located in the current folder. You can also just write

```
Terminal
Unix> prog.py arg1 arg2
```

but then you need to have the dot in the `PATH` variable, which is not recommended for security reasons.

In the two latter commands there is no information on which Python interpreter to use. This information must be provided in the first line of the program, normally as

```
#!/usr/bin/env python
```

This looks like a comment line, and behaves indeed as a comment line when we run the program as `python prog.py`. However, when we run the program as `./prog.py`, the first line beginning with `#!` tells the operating system to use the program specified in the rest of the first line to interpret the program. In this example, we use the first `python` program encountered in the folders in your `PATH` variable. Alternatively, a specific `python` program can be specified as

```
#!/home/hpl/special/tricks/python
```

H.2.3 Executing Python programs in Windows

In a DOS or PowerShell window you can always run a Python program by

```
Terminal
PowerShell> python prog.py arg1 arg2
```

if `prog.py` is the name of the program, and `arg1` and `arg2` are command-line arguments. The extension `.py` can be dropped:

```
Terminal
PowerShell> python prog arg1 arg2
```

If there are several Python installations on your system, a particular installation can be specified:

```
Terminal
PowerShell> E:\hpl\myprogs\Python2.7.5\python prog arg1 arg2
```

Files with a certain extension can in Windows be associated with a file type, and a file type can be associated with a particular program to handle the file. For example, it is natural to associate the extension `.py` with Python programs. The corresponding program needed to interpret

.py files is then python.exe. When we write just the name of the Python program file, as in

```
Terminal
PowerShell> prog arg1 arg2
```

the file is always interpreted by the specified python.exe program. The details of getting .py files to be interpreted by python.exe go as follows:

```
Terminal
PowerShell> assoc .py=PyProg
PowerShell> ftype PyProg=python.exe "%1" %*
```

Depending on your Python installation, such file extension bindings may already be done. You can check this with

```
Terminal
PowerShell> assoc | find "py"
```

To see the programs associated with a file type, write ftype name where name is the name of the file type as specified by the assoc command. Writing help ftype and help assoc prints out more information about these commands along with examples.

One can also run Python programs by writing just the basename of the program file, i.e., prog.py instead of prog.py, if the file extension is registered in the PATHEXT environment variable.

Double-clicking Python files. The usual way of running programs in Windows is to double click on the file icon. This does not work well with Python programs without a graphical user interface. When you double click on the icon for a file prog.py, a DOS window is opened, prog.py is interpreted by some python.exe program, and when the program terminates, the DOS window is closed. There is usually too little time for the user to observe the output in this short-lived DOS window.

One can always insert a final statement that pauses the program by waiting for input from the user:

```
raw_input('Type CR:')
```

or

```
sys.stdout.write('Type CR:'); sys.stdin.readline()
```

The program will hang until the user presses the Return key. During this pause the DOS window is visible and you can watch the output from previous statements in the program.

The downside of including a final input statement is that you must always hit Return before the program terminates. This is inconvenient if the program is moved to a Unix-type machine. One possibility is to let this final input statement be active only when the program is run in Windows:

```
if sys.platform[:3] == 'win':
    raw_input('Type CR:')
```

Python programs that have a graphical user interface can be double-clicked in the usual way if the file extension is `.pyw`.

H.2.4 Executing Python programs in Mac OS X

Since a variant of Unix is used as core in the Mac OS X operating system, you can always launch a Unix terminal and use the techniques from Section H.2.2 to run Python programs.

H.2.5 Making a complete stand-alone executable

Python programs need a Python interpreter and usually a set of modules to be installed on the computer system. Sometimes this is inconvenient, for instance when you want to give your program to somebody who does not necessarily have Python or the required set of modules installed.

Fortunately, there are tools that can create a stand-alone executable program out of a Python program. This stand-alone executable can be run on every computer that has the same type of operating system and the same chip type. Such a stand-alone executable is a bundling of the Python interpreter and the required modules, along with your program, in a single file.

The leading tool for creating a stand-alone executable (or alternatively a folder with all necessary files) is PyInstaller[29]. Say you have program `myprog.py` that you want to distribute to people without the necessary Python environment on their computer. You run

```
Terminal> pyinstaller --onefile myprog.py
```

and a folder `dist` is created with the (big) stand-alone executable file `myprog` (or `myprog.exe` in Windows).

H.3 Doing operating system tasks in Python

Python has extensive support for operating system tasks, such as file and folder management. The great advantage of doing operating system

[29] `http://www.pyinstaller.org/`

tasks in Python and not directly in the operating system is that the Python code works uniformly on Unix/Linux, Windows, and Mac (there are exceptions, but they are few). Below we list some useful operations that can be done inside a Python program or in an interactive session.

Make a folder. Python applies the term directory instead of folder. The equivalent of the Unix `mkdir mydir` is

```
import os
os.mkdir('mydir')
```

Ordinary files are created by the `open` and `close` functions in Python.

Make intermediate folders. Suppose you want to make a subfolder under your home folder:

```
$HOME/python/project1/temp
```

but the intermediate folders `python` and `project1` do not exist. This requires each new folder to be made separately by `os.mkdir`, or you can make all folders at once with `os.makedirs`:

```
foldername = os.path.join(os.environ['HOME'], 'python',
                          'project1', 'temp')
os.makedirs(foldername)
```

With `os.environ[var]` we can get the value of any environment variable `var` as a string. The `os.path.join` function joins folder names and a filename in a platform-independent way.

Move to a folder. The `cd` command reads `os.chdir` and `cwd` is `os.getcwd`:

```
origfolder = os.getcwd()  # get name of current folder
os.chdir(foldername)      # move ("change directory")
...
os.chdir(origfolder)      # move back
```

Rename a file or folder. The cross-platform `mv` command is

```
os.rename(oldname, newname)
```

List files. Unix wildcard notation can be used to list files. The equivalent of `ls *.py` and `ls plot*[1-4]*.dat` reads

```
import glob
filelist1 = glob.glob('*.py')
filelist2 = glob.glob('plot*[1-4]*.dat')
```

List all files and folders in a folder. The counterparts to `ls -a mydir` and just `ls -a` are

```
filelist1 = os.listdir('mydir')
filelist1 = os.listdir(os.curdir)  # current folder (directory)
filelist1.sort()                   # sort alphabetically
```

Check if a file or folder exists. The widely used constructions in Unix scripts for testing if a file or folder exist are `if [ -f $filename ]; then` and `if [ -d $dirname ]; then`. These have very readable counterparts in Python:

```
if os.path.isfile(filename):
    inputfile = open(filename, 'r')
    ...

if os.path.isdir(dirnamename):
    filelist = os.listdir(dirname)
    ...
```

Remove files. Removing a single file is done with `os.rename`, and a loop is required for doing `rm tmp_*.df`:

```
import glob
filelist = glob.glob('tmp_*.pdf')
for filename in filelist:
    os.remove(filename)
```

Remove a folder and all its subfolders. The `rm -rf mytree` command removes an entire folder tree. In Python, the cross-platform valid command becomes

```
import shutil
shutil.rmtree(foldername)
```

It goes without saying that this command must be used with great care!

Copy a file to another file or folder. The `cp fromfile tofile` construction applies `shutil.copy` in Python:

```
shutil.copy('fromfile', 'tofile')
```

Copy a folder and all its subfolders. The recursive copy command `cp -r` is

```
shutil.copytree(sourcefolder, destination)
```

Run any operating system command. The simplest way of running another program from Python is to use `os.system`:

```
cmd = 'c2f.py 21'   # command to be run
failure = os.system(cmd)
if failure:
    print 'Execution of "%s" failed!\n' % cmd
    sys.exit(1)
```

The recommended way to run operating system commands is to use the `subprocess` module. The above command is equivalent to

```
import subprocess
cmd = 'c2f.py 21'
failure = subprocess.call(cmd, shell=True)

# or
failure = subprocess.call(['c2f.py', '21'])
```

The output of an operating system command can be stored in a string object:

```
try:
    output = subprocess.check_output(cmd, shell=True,
                                     stderr=subprocess.STDOUT)
except subprocess.CalledProcessError:
    print 'Execution of "%s" failed!\n' % cmd
    sys.exit(1)

# Process output
for line in output.splitlines():
    ...
```

The `stderr` argument ensures that the `output` string contains everything that the command `cmd` wrote to both standard output and standard error.

The constructions above are mainly used for running stand-alone programs. Any file or folder listing or manipulation should be done by the functionality in the `os` and `shutil` modules.

Split file or folder name. Given `/user/data/file1.dat` as a file path, Python has tools for extracting the folder name `/user/data`, the basename `file1.dat`, and the extension `.dat`:

```
>>> fname = os.path.join(os.environ['HOME'], 'data', 'file1.dat')
>>> fname
'/home/hpl/data/file1.dat'
>>> foldername, basename = os.path.split(fname)
>>> foldername
'/home/hpl/data'
>>> basename
'file1.dat'
>>> stem, ext = os.path.splitext(basename)
>>> stem
'file1'
>>> ext
'.dat'
>>> outfile = stem + '.out'
>>> outfile
'file1.out'
```

H.4 Variable number of function arguments

Arguments to Python functions are of four types:

- positional arguments, where each argument has a name,
- keyword arguments, where each argument has a name and a default value,
- a variable number of positional arguments, where each argument has no name, but just a location in a list,
- a variable number of keyword arguments, where each argument is a name-value pair in a dictionary.

The corresponding general function definition can be sketched as

```
def f(pos1, pos2, key1=val1, key2=val2, *args, **kwargs):
```

Here, `pos1` and `pos2` are positional arguments, `key1` and `key2` are keyword arguments, `args` is a tuple holding a variable number of positional arguments, and `kwargs` is a dictionary holding a variable number of keyword arguments. This appendix describes how to program with the `args` and `kwargs` variables and why these are handy in many situations.

H.4.1 Variable number of positional arguments

Let us start by making a function that takes an arbitrary number of arguments and computes their sum:

```
>>> def add(*args):
...     print 'args:', args
...     s = 0
...     for arg in args:
...         s = s + arg
...     return s
...
>>> add(1)
args: (1,)
1
>>> add(1,5,10)
args: (1, 5, 10)
16
```

We observe that `args` is a tuple and that all the arguments we provide in a call to `add` are stored in `args`.

Combination of ordinary positional arguments and a variable number of arguments is allowed, but the `*args` argument must appear after the ordinary positional arguments, e.g.,

```
def f(pos1, pos2, pos3, *args):
```

In each call to `f` we must provide at least three arguments. If more arguments are supplied in the call, these are collected in the `args` tuple inside the `f` function.

Example. Consider a mathematical function with one independent variable t and a parameter v_0, as in $y(t; v_0) = v_0 t - \frac{1}{2}gt^2$. A more general case with n parameters is $f(x; p_1, \ldots, p_n)$. The Python implementation of such functions can take both the independent variable and the parameters as arguments: `y(t, v0)` and `f(x, p1, p2, ...,pn)`. Suppose that we have a general library routine that operates on functions of one variable. The routine can, e.g., perform numerical differentiation, integration, or root finding. A simple example is a numerical differentiation function

```
def diff(f, x, h):
    return (f(x+h) - f(x))/h
```

This `diff` function cannot be used with functions `f` that take more than one argument. For example, passing an `y(t, v0)` function as `f` leads to the exception

```
TypeError: y() takes exactly 2 arguments (1 given)
```

Section 7.1.1 provides a solution to this problem where `y` becomes a class instance. Here we shall describe an alternative solution that allows our `y(t, v0)` function to be used as is.

The idea is that we pass additional arguments for the parameters in the `f` function *through* the `diff` function. That is, we view the `f` function as `f(x, *f_prms)` in `diff`. Our `diff` routine can then be written as

```
def diff(f, x, h, *f_prms):
    print 'x:', x, 'h:', h, 'f_prms:', f_prms
    return (f(x+h, *f_prms) - f(x, *f_prms))/h
```

Before explaining this function in detail, we demonstrate that it works in an example:

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

dydt = diff(y, 0.1, 1E-9, 3)  # t=0.1, h=1E-9, v0=3
```

The output from the call to `diff` becomes

```
x: 0.1 h: 1e-09 f_prms: (3,)
```

The point is that the `v0` parameter, which we want to pass on to our `y` function, is now stored in `f_prms`. Inside the `diff` function, calling

```
f(x, *f_prms)
```

is the same as if we had written

```
f(x, f_prms[0], f_prms[1], ...)
```

That is, `*f_prms` in a call takes all the values in the tuple `*f_prms` and places them after each other as positional arguments. In the present example with the `y` function, `f(x, *f_prms)` implies `f(x, f_prms[0])`, which for the current set of argument values in our example becomes a call `y(0.1, 3)`.

For a function with many parameters,

```
def G(x, t, A, a, w):
    return A*exp(-a*t)*sin(w*x)
```

the output from

```
dGdx = diff(G, 0.5, 1E-9, 0, 1, 0.6, 100)
```

becomes

```
x: 0.5 h: 1e-09 f_prms: (0, 1, 1.5, 100)
```

We pass here the arguments `t`, `A`, `a`, and `w`, in that sequence, as the last four arguments to `diff`, and all the values are stored in the `f_prms` tuple.

The `diff` function also works for a plain function `f` with one argument:

```
from math import sin
mycos = diff(sin, 0, 1E-9)
```

In this case, `*f_prms` becomes an empty tuple, and a call like `f(x, *f_prms)` is just `f(x)`.

The use of a variable set of arguments for sending problem-specific parameters through a general library function, as we have demonstrated here with the `diff` function, is perhaps the most frequent use of `*args`-type arguments.

H.4.2 Variable number of keyword arguments

A simple test function

```
>>> def test(**kwargs):
...     print kwargs
```

exemplifies that `kwargs` is a dictionary inside the `test` function, and that we can pass any set of keyword arguments to `test`, e.g.,

```
>>> test(a=1, q=9, method='Newton')
{'a': 1, 'q': 9, 'method': 'Newton'}
```

We can combine an arbitrary set of positional and keyword arguments, provided all the keyword arguments appear at the end of the call:

```
>>> def test(*args, **kwargs):
...     print args, kwargs
...
>>> test(1,3,5,4,a=1,b=2)
(1, 3, 5, 4) {'a': 1, 'b': 2}
```

From the output we understand that all the arguments in the call where we provide a name and a value are treated as keyword arguments and hence placed in `kwargs`, while all the remaining arguments are positional and placed in `args`.

Example. We may extend the example in Section H.4.1 to make use of a variable number of keyword arguments instead of a variable number of positional arguments. Suppose all functions with parameters in addition to an independent variable take the parameters as keyword arguments. For example,

```
def y(t, v0=1):
    g = 9.81
    return v0*t - 0.5*g*t**2
```

In the `diff` function we transfer the parameters in the `f` function as a set of keyword arguments `**f_prms`:

```
def diff(f, x, h=1E-10, **f_prms):
    print 'x:', x, 'h:', h, 'f_prms:', f_prms
    return (f(x+h, **f_prms) - f(x, **f_prms))/h
```

In general, the `**f_prms` argument in a call

```
f(x, **f_prms)
```

implies that all the key-value pairs in `**f_prms` are provided as keyword arguments:

```
f(x, key1=f_prms[key1], key2=f_prms[key2], ...)
```

In our special case with the `y` function and the call

```
dydt = diff(y, 0.1, h=1E-9, v0=3)
```

`f(x, **f_prms)` becomes `y(0.1, v0=3)`. The output from `diff` is now

```
x: 0.1 h: 1e-09 f_prms: {'v0': 3}
```

showing explicitly that our `v0=3` in the call to `diff` is placed in the `f_prms` dictionary.

The `G` function from Section H.4.1 can also have its parameters as keyword arguments:

```
def G(x, t=0, A=1, a=1, w=1):
    return A*exp(-a*t)*sin(w*x)
```

We can now make the call

```
dGdx = diff(G, 0.5, h=1E-9, t=0, A=1, w=100, a=1.5)
```

and view the output from `diff`,

```
x: 0.5 h: 1e-09 f_prms: {'A': 1, 'a': 1.5, 't': 0, 'w': 100}
```

to see that all the parameters get stored in `f_prms`. The `h` parameter can be placed anywhere in the collection of keyword arguments, e.g.,

```
dGdx = diff(G, 0.5, t=0, A=1, w=100, a=1.5, h=1E-9)
```

We can allow the `f` function of one variable and a set of parameters to have the general form `f(x, *f_args, **f_kwargs)`. That is, the parameters can either be positional or keyword arguments. The `diff` function must take the arguments `*f_args` and `**f_kwargs` and transfer these to `f`:

```
def diff(f, x, h=1E-10, *f_args, **f_kwargs):
    print f_args, f_kwargs
    return (f(x+h, *f_args, **f_kwargs) -
            f(x,   *f_args, **f_kwargs))/h
```

This `diff` function gives the writer of an `f` function full freedom to choose positional and/or keyword arguments for the parameters. Here is an example of the `G` function where we let the `t` parameter be positional and the other parameters be keyword arguments:

```
def G(x, t, A=1, a=1, w=1):
    return A*exp(-a*t)*sin(w*x)
```

A call

```
dGdx = diff(G, 0.5, 1E-9, 0, A=1, w=100, a=1.5)
```

gives the output

```
(0,) {'A': 1, 'a': 1.5, 'w': 100}
```

showing that `t` is put in `f_args` and transferred as positional argument to `G`, while `A`, `a`, and `w` are put in `f_kwargs` and transferred as keyword arguments. We remark that in the last call to `diff`, `h` and `t` *must* be treated as positional arguments, i.e., we cannot write `h=1E-9` and `t=0` unless *all* arguments in the call are on the `name=value` form.

In the case we use both `*f_args` and `**f_kwargs` arguments in `f` and there is no need for these arguments, `*f_args` becomes an empty tuple and `**f_kwargs` becomes an empty dictionary. The example

```
mycos = diff(sin, 0)
```

shows that the tuple and dictionary are indeed empty since `diff` just prints out

```
() {}
```

Therefore, a variable set of positional and keyword arguments can be incorporated in a general library function such as `diff` without any disadvantage, just the benefit that `diff` works with different types of `f` functions: parameters as global variables, parameters as additional positional arguments, parameters as additional keyword arguments, or parameters as instance variables (Section 7.1.2).

The program `varargs1.py` in the `src/varargs`[30] folder implements the examples in this appendix.

H.5 Evaluating program efficiency

H.5.1 Making time measurements

The term *time* has multiple meanings on a computer. The *elapsed time* or *wall clock time* is the same time as you can measure on a watch or wall clock, while *CPU time* is the amount of time the program keeps the central processing unit busy. The *system time* is the time spent on operating system tasks like I/O. The concept *user time* is the difference between the CPU and system times. If your computer is occupied by many concurrent processes, the CPU time of your program might be very different from the elapsed time.

The time module. Python has a `time` module with some useful functions for measuring the elapsed time and the CPU time:

```
import time
e0 = time.time()     # elapsed time since the epoch
c0 = time.clock()    # total CPU time spent in the program so far
<do tasks...>
elapsed_time = time.time() - e0
cpu_time = time.clock() - c0
```

The term *epoch* means initial time (`time.time()` would return 0), which is 00:00:00 January 1, 1970. The `time` module also has numerous functions for nice formatting of dates and time, and the newer `datetime` module has more functionality and an improved interface. Although the timing has a finer resolution than seconds, one should construct test cases that last some seconds to obtain reliable results.

Using timeit from IPython. To measure the efficiency of a certain set of statements, an expression, or a function call, the code should be run a large number of times so the overall CPU time is of order seconds. Python's `timeit` module has functionality for running a code segment repeatedly. The simplest and most convenient way of using `timeit` is within an IPython shell. Here is a session comparing the efficiency of `sin(1.2)` versus `math.sin(1.2)`:

```
In [1]: import math

In [2]: from math import sin

In [3]: %timeit sin(1.2)
10000000 loops, best of 3: 198 ns per loop

In [4]: %timeit math.sin(1.2)
1000000 loops, best of 3: 258 ns per loop
```

[30] http://tinyurl.com/pwyasaa/tech

That is, looking up `sin` through the `math` prefix degrades the performance by a factor of $258/198 \approx 1.3$.

Any statement, including function calls, can be timed the same way. Timing of multiple statements is possible by using `%%timeit`. The `timeit` module can be used inside ordinary programs as demonstrated in the file `pow_eff.py`.

Hardware information. Along with CPU time measurements it is often convenient to print out information about the hardware on which the experiment was done. Python has a module `platform` with information on the current hardware. The function `scitools.misc.hardware_info` applies the `platform` module and other modules to extract relevant hardware information. A sample call is

```
>>> import scitools.misc, pprint
>>> pprint.pprint(scitools.misc.hardware_info())
{'numpy.distutils.cpuinfo.cpu.info': [
 {'address sizes': '40 bits physical, 48 bits virtual',
  'bogomips': '4598.87',
  'cache size': '4096 KB',
  'cache_alignment': '64',
  'cpu MHz': '2299.435',
  ...
 },
 'platform module': {
  'identifier': 'Linux-3.11.0-12-generic-x86_64-with-Ubuntu-13.10',
  'python build': ('default', 'Sep 19 2013 13:48:49'),
  'python version': '2.7.5+',
  'uname': ('Linux', 'hpl-ubuntu2-mac11', '3.11.0-12-generic',
            '#19-Ubuntu SMP Wed Oct 9 16:20:46 UTC 2013',
            'x86_64', 'x86_64')}}
}
```

H.5.2 Profiling Python programs

A profiler computes the time spent in the various functions of a program. From the timings a ranked list of the most time-consuming functions can be created. This is an indispensable tool for detecting bottlenecks in the code, and you should always perform a profiling before spending time on code optimization. The golden rule is to first write an easy-to-understand program, then verify it, then profile it, and then think about optimization.

Premature optimization is the root of all evil.
Donald Knuth, computer scientist, 1938-.

Python 2.7 comes with two recommended profilers, implemented in the modules `cProfile` and `profiles`. The section The Python Profilers[31] in the Python Standard Library documentation [3] has a good introduction to the usage of these modules. The results produced by the modules

[31] `http://docs.python.org/2/library/profile.html`

are normally processed by a special statistics utility `pstats` developed for analyzing profiling results. The usage of the `profile`, `cProfile`, and `pstats` modules is straightforward, but somewhat tedious. The SciTools package therefore comes with a command `scitools profiler` that allows you to profile any program (say) `m.py` by just writing

```
Terminal
Terminal> scitools profiler m.py c1 c2 c3
```

Here, `c1`, `c2`, and `c3` are command-line arguments to `m.py`.

A sample output might read

```
    1082 function calls (728 primitive calls) in 17.890 CPU seconds

Ordered by: internal time
List reduced from 210 to 20 due to restriction <20>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     5    5.850    1.170    5.850    1.170 m.py:43(loop1)
     1    2.590    2.590    2.590    2.590 m.py:26(empty)
     5    2.510    0.502    2.510    0.502 m.py:32(myfunc2)
     5    2.490    0.498    2.490    0.498 m.py:37(init)
     1    2.190    2.190    2.190    2.190 m.py:13(run1)
     6    0.050    0.008   17.720    2.953 funcs.py:126(timer)
...
```

In this test, `loop1` is the most expensive function, using 5.85 seconds, which is to be compared with 2.59 seconds for the next most time-consuming function, `empty`. The `tottime` entry is the total time spent in a specific function, while `cumtime` reflects the total time spent in the function and all the functions it calls. We refer to the documentation of the profiling tools in the Python Standard Library documentation for detailed information on how to interpret the output.

The CPU time of a Python program typically increases with a factor of about five when run under the administration of the `profile` module. Nevertheless, the relative CPU time among the functions are not much affected by the profiler overhead.

H.6 Software testing

Unit testing is widely a used technique for verifying software implementation. The idea is to identify small units of code and test each unit, ideally in a way such that one test does not depend on the outcome of other tests. Several tools, often referred to as testing frameworks, exist for automatically running all tests in a software package and report if any test failed. The value of such tools during software development cannot be exaggerated. Below we describe how to write tests that can be used by either the nose[32] or the pytest[33] testing frameworks. Both these have

[32] `https://nose.readthedocs.org/`

[33] `http://pytest.org/latest/`

a very low barrier for beginners, so there is no excuse for not using nose or pytest as soon as you have learned about functions in programming.

Model software. We need a piece of software we want to test. Here we choose a function that runs Newton's method for solving algebraic equations $f(x) = 0$. A very simple implementation goes like

```
def Newton_basic(f, dfdx, x, eps=1E-7):
    n = 0  # iteration counter
    while abs(f(x)) > eps:
        x = x - f(x)/dfdx(x)
        n += 1
    return x, f(x), n
```

H.6.1 Requirements of the test function

The simplest way of using the pytest or nose testing frameworks is to write a set of test functions, scattered around in files, such that pytest or nose can automatically find and run all the test functions. To this end, the test functions need to follow certain conventions.

Test function conventions

1. The name of a test function starts with `test_`.
2. A test function cannot take any arguments.
3. Any test must be formulated as a boolean condition.
4. An `AssertionError` exception is raised if the boolean condition is false (i.e., when the test fails).

There are many ways of raising the `AssertionError` exception:

```
# Formulate a test
tol = 1E-14              # comparison tolerance for real numbers
success = abs(reference - result) < tol
msg = 'computed_result=%d != %d' % (result, reference)

# Explicit raise
if not success:
    raise AssertionError(msg)

# assert statement
assert success, msg

# nose tools
import nose.tools as nt
nt.assert_true(success, msg)
# or
nt.assert_almost_equal(result, reference, msg=msg, delta=tol)
```

This book contains a lot of test functions following the conventions of the pytest and nose testing frameworks, and we almost exclusively use the plain `assert` statement to have full control of what the test method is. In more complicated software the many functions in `nose.tools` may save quite some coding and are convenient to use.

H.6.2 Writing the test function; precomputed data

Newton's method for solving an algebraic equation $f(x) = 0$ results in only an approximate root x_r, making $f(x_r) \neq 0$, but $|f(x_r)| \leq \epsilon$, where ϵ is supposed to be a prescribed number close to zero. The problem is that we do not know beforehand what x_r and $f(x_r)$ will be. However, if we strongly believe the function we want to test is correctly implemented, we can record the output from the function in a test case and use this output as a reference for later testing.

Assume we try to solve $\sin(x) = 0$ with $x = -\pi/3$ as start value. Running `Newton_basic` with a moderate-size `eps` (ϵ) of 10^{-2} gives $x = 0.000769691024206$, $f(x) = 0.000769690948209$, and $n = 3$. A test function can now compare new computations with these reference results. Since new computations on another computer may lead to round-off errors, we must compare real numbers with a small tolerance:

```
def test_Newton_basic_precomputed():
    from math import sin, cos, pi

    def f(x):
        return sin(x)

    def dfdx(x):
        return cos(x)

    x_ref = 0.000769691024206
    f_x_ref = 0.000769690948209
    n_ref = 3

    x, f_x, n = Newton_basic(f, dfdx, x=-pi/3, eps=1E-2)

    tol = 1E-15  # tolerance for comparing real numbers
    assert abs(x_ref - x) < tol       # is x correct?
    assert abs(f_x_ref - f_x) < tol   # is f_x correct?
    assert n == 3                     # is n correct?
```

The `assert` statements involving comparison of real numbers can alternatively be carried out by `nose.tools` functionality:

```
nose.tools.assert_almost_equal(x_ref, x, delta=tol)
```

For simplicity we dropped the optional messages explaining what wen wrong if tests fail.

H.6.3 Writing the test function; exact numerical solution

Approximate numerical methods are sometimes exact in certain special cases. An exact answer known beforehand is a good starting point for a test since the implementation should reproduce the known answer to machine precision. For Newton's method we know that it finds the exact root of $f(x) = 0$ in one iteration if $f(x)$ is a linear function of x. This fact leads us to a test with $f(x) = ax + b$, where we can choose a and b freely, but it is always wise to choose numbers different from 0 and 1 since these have special arithmetic properties that can hide programming errors.

The test function contains the problem setup, a call to the function to be verified, and `assert` tests on the output, this time also with an error message in case tests fail:

```
def test_Newton_basic_linear():
    """Test that a linear function is handled in one iteration."""
    f = lambda x: a*x + b
    dfdx = lambda x: a
    a = 0.25; b = -4
    x_exact = 16
    eps = 1E-5
    x, f_x, n = Newton_basic(f, dfdx, -100, eps)

    tol = 1E-15  # tolerance for comparing real numbers
    assert abs(x - 16) < tol, 'wrong root x=%g != 16' % x
    assert abs(f_x) < eps, '|f(root)|=%g > %g' % (f_x, eps)
    assert n == 1, 'n=%d, but linear f should have n=1' % n
```

H.6.4 Testing of function robustness

Our `Newton_basic` function is very basic and suffers from several problems:

- for divergent iterations it will iterate forever,
- it can divide by zero in `f(x)/dfdx(x)`,
- it can perform integer division in `f(x)/dfdx(x)`,
- it does not test whether the arguments have acceptable types and values.

A more robust implementation dealing with these potential problems look as follows:

```
def Newton(f, dfdx, x, eps=1E-7, maxit=100):
    if not callable(f):
        raise TypeError(
            'f is %s, should be function or class with __call__'
            % type(f))
    if not callable(dfdx):
        raise TypeError(
            'dfdx is %s, should be function or class with __call__'
            % type(dfdx))
    if not isinstance(maxit, int):
        raise TypeError('maxit is %s, must be int' % type(maxit))
    if maxit <= 0:
        raise ValueError('maxit=%d <= 0, must be > 0' % maxit)

    n = 0  # iteration counter
    while abs(f(x)) > eps and n < maxit:
        try:
            x = x - f(x)/float(dfdx(x))
        except ZeroDivisionError:
            raise ZeroDivisionError(
            'dfdx(%g)=%g - cannot divide by zero' % (x, dfdx(x)))
        n += 1
    return x, f(x), n
```

The numerical functionality can be tested as described in the previous example, but we should include additional tests for testing the additional functionality. One can have different tests in different test functions, or collect several tests in one test function. The preferred strategy depends on the problem. Here it may be natural to have different test functions only when the $f(x)$ formula differs to avoid repeating code.

To test for divergence, we can choose $f(x) = \tanh(x)$, which is known to lead to divergent iterations if not x is sufficiently close to the root $x = 0$. A start value $x = 20$ reveals that the iterations are divergent, so we set maxit=12 and test that the actual number of iterations reaches this limit. We can also add a test on x, e.g., that x is a big as we know it will be: $x > 10^{50}$ after 12 iterations. The test function becomes

```
def test_Newton_divergence():
    from math import tanh
    f = tanh
    dfdx = lambda x: 10./(1 + x**2)

    x, f_x, n = Newton(f, dfdx, 20, eps=1E-4, maxit=12)
    assert n == 12
    assert x > 1E+50
```

To test for division by zero, we can find an $f(x)$ and an x such that $f'(x) = 0$. One simple example is $x = 0$, $f(x) = \cos(x)$, and $f'(x) = -\sin(x)$. If $x = 0$ is the start value, we know that a division by zero will take place in the first iteration, and this will lead to a ZeroDivisionError exception. We can explicitly handle this exception and introduce a boolean variable success that is True if the exception is raised and otherwise False. The corresponding test function reads

```
def test_Newton_div_by_zero1():
    from math import sin, cos
    f = cos
    dfdx = lambda x: -sin(x)
    success = False
    try:
        x, f_x, n = Newton(f, dfdx, 0, eps=1E-4, maxit=1)
    except ZeroDivisionError:
        success = True
    assert success
```

There is a special nose.tools.assert_raises helper function that can be used to test if a function raises a certain exception. The arguments to assert_raises are the exception type, the name of the function to be called, and all positional and keyword arguments in the function call:

```
import nose.tools as nt

def test_Newton_div_by_zero2():
    from math import sin, cos
    f = cos
    dfdx = lambda x: -sin(x)
    nt.assert_raises(
        ZeroDivisionError, Newton, f, dfdx, 0, eps=1E-4, maxit=1)
```

Let us proceed with testing that wrong input is caught by function Newton. Since the same type of exception is raised for different type of errors we shall now also examine (parts of) the exception messages. The first test involves an argument f that is not a function:

```
def test_Newton_f_is_not_callable():
    success = False
    try:
        Newton(4.2, 'string', 1.2, eps=1E-7, maxit=100)
    except TypeError as e:
        if "f is <type 'float'>" in e.message:
            success = True
```

As seen, `success = True` demands that the right exception is raised and that its message starts with `f is <type 'float'>`. What text to expect in the message is evident from the source in function `Newton`.

The `nose.tools` module also has a function for testing the exception type and the message content. This is illustrated when `dfdx` is not callable:

```
def test_Newton_dfdx_is_not_callable():
    nt.assert_raises_regexp(
        TypeError, "dfdx is <type 'str'>",
        Newton, lambda x: x**2, 'string', 1.2, eps=1E-7, maxit=100)
```

Checking that `Newton` catches `maxit` of wrong type or with a negative value can be carried out by these test functions:

```
def test_Newton_maxit_is_not_int():
    nt.assert_raises_regexp(
        TypeError, "maxit is <type 'float'>",
        Newton, lambda x: x**2, lambda x: 2*x,
        1.2, eps=1E-7, maxit=1.2)

def test_Newton_maxit_is_neg():
    nt.assert_raises_regexp(
        ValueError, "maxit=-2 <= 0",
        Newton, lambda x: x**2, lambda x: 2*x,
        1.2, eps=1E-7, maxit=-2)
```

The corresponding support for testing exceptions in pytest is

```
import pytest
with pytest.raises(TypeError) as e:
    Newton(lambda x: x**2, lambda x: 2*x, 1.2, eps=1E-7, maxit=-2)
```

H.6.5 Automatic execution of tests

Our code for the `Newton_basic` and `Newton` functions is placed in a file `eq_solver.py` together with the tests. To run all test functions with names of the form `test_*()` in this file, use the `nosetests` or `py.test` commands, e.g.,:

```
Terminal
Terminal> nosetests -s eq_solver.py
..........
-------------------------------------------------------------------
Ran 10 tests in 0.004s

OK
```

The `-s` option causes all output from the called functions in the program `eq_solver.py` to appear on the screen (by default, `nosetests` and `py.test` suppress all output). The final `OK` points to the fact that no test failed. Adding the option `-v` prints out the outcome of each individual test function. In case of failure, the `AssertionError` exception and the associated message, if existing, are displayed. Pytest also displays the code that failed.

One can also collect test functions in separate files with names starting with `test`. A simple command `nosetests -s -v` will look for all such files in this folder as well as in all subfolders if the folder names start with `test` or end with `_test` or `_tests`. By following this naming convention, `nosetests` can automatically run a potentially large number of tests and give us quick feedback. The `py.test -s -v` command will look for and run all test files in the entire tree of *any* subfolder.

Remark on classical class-based unit testing. The pytest and nose testing frameworks allow ordinary functions, as explained above, to perform the testing. The most widespread way of implementing unit tests, however, is to use class-based frameworks. This is also possible with nose and with a module `unittest` that comes with standard Python. The class-based approach is very accessible for people with experience from JUnit in Java and similar tools in other languages. Without such a background, plain functions that follow the pytest/nose conventions are faster and cleaner to write than the class-based counterparts.

References

1. D. Beazley. *Python Essential Reference*. Addison-Wesley, 4th edition, 2009.
2. O. Certik et al. SymPy: Python library for symbolic mathematics. `http://sympy.org/`.
3. Python Software Foundation. The Python standard library. `http://docs.python.org/2/library/`.
4. C. Führer, J. E. Solem, and O. Verdier. *Computing with Python - An Introduction to Python for Science and Engineering*. Pearson, 2014.
5. J. E. Grayson. *Python and Tkinter Programming*. Manning, 2000.
6. Richard Gruet. Python quick reference. `http://rgruet.free.fr/`.
7. D. Harms and K. McDonald. *The Quick Python Book*. Manning, 1999.
8. ScientificPython software package. `http://starship.python.net/crew/hinsen`.
9. J. D. Hunter. Matplotlib: a 2d graphics environment. *Computing in Science & Engineering*, 9, 2007.
10. J. D. Hunter et al. Matplotlib: Software package for 2d graphics. `http://matplotlib.org/`.
11. E. Jones, T. E. Oliphant, P. Peterson, et al. SciPy scientific computing library for Python. `http://scipy.org`.
12. D. E. Knuth. Theory and practice. *EATCS Bull.*, 27:14–21, 1985.
13. H. P. Langtangen. Quick intro to version control systems and project hosting sites. *http://hplgit.github.io/teamods/bitgit/html/*.
14. H. P. Langtangen. *Python Scripting for Computational Science*, volume 3 of *Texts in Computational Science and Engineering*. Springer, 3rd edition, 2009.
15. H. P. Langtangen and J. H. Ring. SciTools: Software tools for scientific computing. `http://code.google.com/p/scitools`.
16. L. S. Lerner. *Physics for Scientists and Engineers*. Jones and Barlett, 1996.
17. M. Lutz. *Programming Python*. O'Reilly, 4th edition, 2011.
18. M. Lutz. *Learning Python*. O'Reilly, 2013.
19. J. D. Murray. *Mathematical Biology I: an Introduction*. Springer, 3rd edition, 2007.
20. T. E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9, 2007.
21. T. E. Oliphant et al. NumPy array processing package for Python. `http://www.numpy.org`.
22. F. Perez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9, 2007.
23. F. Perez, B. E. Granger, et al. IPython software package for interactive scientific computing. `http://ipython.org/`.
24. Python programming language. `http://python.org`.
25. G. J. E. Rawlins. *Slaves of the Machine: The Quickening of Computer Technology*. MIT Press, 1998.

26. G. Ward and A. Baxter. Distributing Python modules. `http://docs.python.org/2/distutils/`.
27. F. M. White. *Fluid Mechanics.* McGraw-Hill, 2nd edition, 1986.

Index