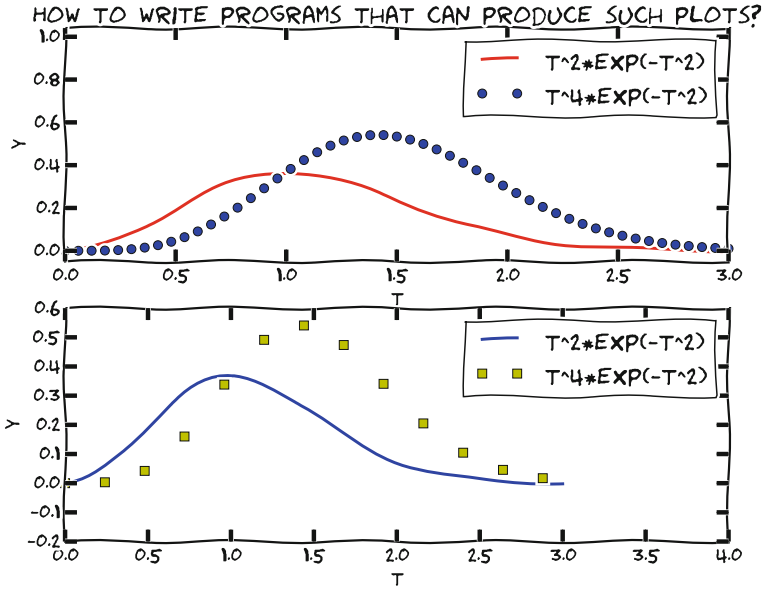
The First Few Steps

1



1.1 What Is a Program? And What Is Programming?

Today, most people are experienced with computer programs, typically programs such as Word, Excel, PowerPoint, Internet Explorer, and Photoshop. The interaction with such programs is usually quite simple and intuitive: you click on buttons, pull down menus and select operations, drag visual elements into locations, and so forth. The possible operations you can do in these programs can be combined in seemingly an infinite number of ways, only limited by your creativity and imagination.

Nevertheless, programs often make us frustrated when they cannot do what we wish. One typical situation might be the following. Say you have some measurements from a device, and the data are stored in a file with a specific format. You

© The Author(s) 2016

S. Linge, H.P. Langtangen, *Programming for Computations – Python*,
Texts in Computational Science and Engineering 15, DOI 10.1007/978-3-319-32428-9_1

may want to analyze these data in Excel and make some graphics out of it. However, assume there is no menu in Excel that allows you to import data in this specific format. Excel can work with many different data formats, but not this one. You start searching for alternatives to Excel that can do the same *and* read this type of data files. Maybe you cannot find any ready-made program directly applicable. You have reached the point where knowing how to write programs on your own would be of great help to you! With some programming skills, you may write your own little program which can translate one data format to another. With that little piece of tailored code, your data may be read and analyzed, perhaps in Excel, or perhaps by a new program tailored to the computations that the measurement data demand.

The real power of computers can only be utilized if you can program them. By programming you can get the computer to do (most often!) exactly what you want. Programming consists of writing a set of instructions in a very specialized language that has adopted words and expressions from English. Such languages are known as *programming* or *computer languages*. The set of instructions is given to a program which can translate the meaning of the instructions into real actions inside the computer.

The purpose of this book is to teach you to write such instructions dedicated to solve mathematical and engineering problems by fundamental numerical methods.

There are numerous computer languages for different purposes. Within the engineering area, the most widely used computer languages are Python, MATLAB, Octave, Fortran, C, C++, and to some extent Maple, and Mathematica. How you write the instructions (i.e. the *syntax*) differs between the languages. Let us use an analogy.

Assume you are an international kind of person, having friends abroad in England, Russia and China. They want to try your favorite cake. What can you do? Well, you may write down the recipe in those three languages and send them over. Now, if you have been able to think correctly when writing down the recipe, and you have written the explanations according to the rules in each language, each of your friends will produce the same cake. Your recipe is the "computer program", while English, Russian and Chinese represent the "computer languages" with their own rules of how to write things. The end product, though, is still the same cake. Note that you may unintentionally introduce errors in your "recipe". Depending on the error, this may cause "baking execution" to stop, or perhaps produce the wrong cake. In your computer program, the errors you introduce are called bugs (yes, small insects! ... for historical reasons), and the process of fixing them is called debugging. When you try to run your program that contains errors, you usually get warnings or error messages. However, the response you get depends on the error and the programming language. You may even get no response, but simply the wrong "cake". Note that the rules of a programming language have to be followed very strictly. This differs from languages like English etc., where the meaning might be understood even with spelling errors and "slang" included.

This book comes in two versions, one that is based on Python, and one based on Matlab. Both Python and Matlab represent excellent programming environments for scientific and engineering tasks. The version you are reading now, is the Python version.

Some of Python's strong properties deserve mention here: Many global functions can be placed in only *one* file, functions are straightforwardly transferred as

arguments to other functions, there is good support for interfacing C, C++ and Fortran code (i.e., a Python program may use code written in other languages), and functions explicitly written for scalar input often work fine (without modification) also with vector input. Another important thing, is that Python is available for *free*. It can be downloaded from the Internet and will run on most platforms.

Readers who want to expand their scientific programming skills beyond the introductory level of the present exposition, are encouraged to study the book *A Primer on Scientific Programming with Python* [13]. This comprehensive book is as suitable for beginners as for professional programmers, and teaches the art of programming through a huge collection of dedicated examples. This book is considered the primary reference, and a natural extension, of the programming matters in the present book.

Some computer science terms

Note that, quite often, the terms *script* and *scripting* are used as synonyms for program and programming, respectively.

The inventor of the Perl programming language, Larry Wall, tried to explain the difference between script and program in a humorous way (from perl.com[1]): *Suppose you went back to Ada Lovelace[2] and asked her the difference between a script and a program. She'd probably look at you funny, then say something like: Well, a script is what you give the actors, but a program is what you give the audience. That Ada was one sharp lady ... Since her time, we seem to have gotten a bit more confused about what we mean when we say scripting. It confuses even me, and I'm supposed to be one of the experts.*

There are many other widely used computer science terms to pick up. Writing a program (or script or code) is often expressed as *implementing* the program. *Executing* a program means running the program. An *algorithm* is a recipe for how to construct a program. A *bug* is an error in a program, and the art of tracking down and removing bugs is called *debugging*. *Simulating* or *simulation* refers to using a program to mimic processes in the real world, often through solving differential equations that govern the physics of the processes.

1.2 A Python Program with Variables

Our first example regards programming a mathematical model that predicts the position of a ball thrown up in the air. From Newton's 2nd law, and by assuming negligible air resistance, one can derive a mathematical model that predicts the vertical position y of the ball at time t. From the model one gets the formula

$$y = v_0 t - 0.5 g t^2,$$

where v_0 is the initial upwards velocity and g is the acceleration of gravity, for which $9.81\,\text{ms}^{-2}$ is a reasonable value (even if it depends on things like location on the earth). With this formula at hand, and when v_0 is known, you may plug in a value for time and get out the corresponding height.

[1] http://www.perl.com/pub/2007/12/06/soto-11.html
[2] http://en.wikipedia.org/wiki/Ada_Lovelace

1.2.1 The Program

Let us next look at a Python program for evaluating this simple formula. Assume the program is contained as text in a file named `ball.py`. The text looks as follows (file `ball.py`):

```
# 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
```

Computer programs and parts of programs are typeset with a blue background in this book. A slightly darker top and bottom bar, as above, indicates that the code is a complete program that can be run as it stands. Without the bars, the code is just a *snippet* and will (normally) need additional lines to run properly.

1.2.2 Dissection of the Program

A computer program is plain text, as here in the file `ball.py`, which contains instructions to the computer. Humans can read the code and understand what the program is capable of doing, but the program itself does not trigger any actions on a computer before another program, the Python interpreter, reads the program text and translates this text into specific actions.

You must learn to play the role of a computer

Although Python is responsible for reading and understanding your program, it is of fundamental importance that you fully understand the program yourself. You have to know the implication of every instruction in the program and be able to figure out the consequences of the instructions. In other words, you must be able to play the role of a computer. The reason for this strong demand of knowledge is that errors unavoidably, and quite often, will be committed in the program text, and to track down these errors, you have to simulate what the computer does with the program. Next, we shall explain all the text in `ball.py` in full detail.

When you run your program in Python, it will interpret the text in your file line by line, from the top, reading each line from left to right. The first line it reads is

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

This line is what we call a *comment*. That is, the line is not meant for Python to read and execute, but rather for a human that reads the code and tries to understand what is going on. Therefore, one rule in Python says that whenever Python encounters

the sign # it takes the rest of the line as a comment. Python then simply skips reading the rest of the line and jumps to the next line. In the code, you see several such comments and probably realize that they make it easier for you to understand (or guess) what is meant with the code. In simple cases, comments are probably not much needed, but will soon be justified as the level of complexity steps up.

The next line read by Python is

```
v0 = 5  # Initial velocity
```

In Python, a statement like `v0 = 5` is known as an *assignment* statement (very different from a mathematical equation!). The result on the right-hand side, here the integer 5, becomes an *object* and the variable name on the left-hand side is a named reference for that object. Whenever we write `v0`, Python will replace it by an integer with value 5. Doing `v1 = v0` creates a new name, `v1`, for the same integer object with value 5 and not a copy of an integer object with value 5. The next two lines

```
g = 9.81  # Acceleration of gravity
t = 0.6   # Time
```

are of the same kind, so having read them too, Python knows of three variables (`v0`, `g`, `t`) and their values. These variables are then used by Python when it reads the next line, the actual "formula",

```
y = v0*t - 0.5*g*t**2       # Vertical position
```

Again, according to its rules, Python interprets `*` as multiplication, `-` as minus and `**` as exponent (let us also add here that, not surprisingly, `+` and `/` would have been understood as addition and division, if such signs had been present in the expression). Having read the line, Python performs the mathematics on the right-hand side, and then assigns the result (in this case the number 1.2342) to the variable name `y`. Finally, Python reads

```
print y
```

This makes Python print the value of `y` out in that window on the screen where you started the program. When `ball.py` is run, the number 1.2342 appears on the screen.

In the code above, you see several blank lines too. These are simply skipped by Python and you may use as many as you want to make a nice and readable layout of the code.

1.2.3 Why Not Just Use a Pocket Calculator?

Certainly, finding the answer as done by the program above could easily have been done with a pocket calculator. No objections to that and no programming would have been needed. However, what if you would like to have the position of the ball

for every milli-second of the flight? All that punching on the calculator would have taken you something like four hours! If you know how to program, however, you could modify the code above slightly, using a minute or two of writing, and easily get all the positions computed in one go within a second. A much stronger argument, however, is that mathematical models from real life are often complicated and comprehensive. The pocket calculator cannot cope with such problems, even not the programmable ones, because their computational power and their programming tools are far too weak compared to what a real computer can offer.

1.2.4 Why You Must Use a Text Editor to Write Programs

When Python interprets some code in a file, it is concerned with every character in the file, exactly as it was typed in. This makes it troublesome to write the code into a file with word processors like, e.g., Microsoft Word, since such a program will insert extra characters, invisible to us, with information on how to format the text (e.g., the font size and type). Such extra information is necessary for the text to be nicely formatted for the human eye. Python, however, will be much annoyed by the extra characters in the file inserted by a word processor. Therefore, it is fundamental that you write your program in a *text editor* where what you type on the keyboard is *exactly* the characters that appear in the file and that Python will later read. There are many text editors around. Some are stand-alone programs like Emacs, Vim, Gedit, Notepad++, and TextWrangler. Others are integrated in graphical development environments for Python, such as Spyder. This book will primarily refer to Spyder and its text editor.

1.2.5 Installation of Python

You will need access to Python and several add-on packages for doing mathematical computations and display graphics. An obvious choice is to install a Python environment for scientific computing on your machine. Alternatively, you can use cloud services for running Python, or you can remote login on a computer system at a school or university. Available and recommended techniques for getting access to Python and the needed packages are documented in Appendix A.

The quickest way to get started with a Python installation for this book on your Windows, Mac, or Linux computer, is to install Anaconda[3].

1.2.6 Write and Run Your First Program

Reading *only* does not teach you computer programming: you have to program yourself and practice heavily before you master mathematical problem solving via programming. Therefore, it is crucial at this stage that you write and run a Python program. We just went through the program `ball.py` above, so let us next write and run that code.

[3] http://continuum.io/downloads

But first a warning: there are many things that must come together in the right way for `ball.py` to run correctly on your computer. There might be problems with your Python installation, with your writing of the program (it is very easy to introduce errors!), or with the location of the file, just to mention some of the most common difficulties for beginners. Fortunately, such problems are solvable, and if you do not understand how to fix the problem, ask somebody. Typically, once you are beyond these common start-up problems, you can move on to learn programming and how programs can do a lot of otherwise complicated mathematics for you.

We describe the first steps using the Spyder graphical user interface (GUI), but you can equally well use a standard text editor for writing the program and a terminal window (*Terminal* on Mac, *Power Shell* on Windows) for running the program. Start up Spyder and type in each line of the program `ball.py` shown earlier. Then run the program. More detailed descriptions of operating Spyder are found in Appendix A.3.

If you have had the necessary luck to get everything right, you should now get the number 1.2342 out in the rightmost lower window in the Spyder GUI. If so, congratulations! You have just executed your first self-written computer program in Python, and you are ready to go on studying this book! You may like to save the program before moving on (`File`, `save as`).

1.3 A Python Program with a Library Function

Imagine you stand on a distance, say 10 m away, watching someone throwing a ball upwards. A straight line from you to the ball will then make an angle with the horizontal that increases and decreases as the ball goes up and down. Let us consider the ball at a particular moment in time, at which it has a height of 10 m.

What is the angle of the line then? Again, this could easily be done with a calculator, but we continue to address gentle mathematical problems when learning to program. Before thinking of writing a program, one should always formulate the *algorithm*, i.e., the recipe for what kind of calculations that must be performed. Here, if the ball is x m away and y m up in the air, it makes an angle θ with the ground, where $\tan\theta = y/x$. The angle is then $\tan^{-1}(y/x)$.

Let us make a Python program for doing these calculations. We introduce names `x` and `y` for the position data x and y, and the descriptive name `angle` for the angle θ. The program is stored in a file `ball_angle_first_try.py`:

```
x = 10              # Horizontal position
y = 10              # Vertical position

angle = atan(y/x)

print (angle/pi)*180
```

Before we turn our attention to the running of this program, let us take a look at one new thing in the code. The line `angle = atan(y/x)`, illustrates how the *function* `atan`, corresponding to $\tan^{-1}$ in mathematics, is *called* with the ratio

`y/x` as *input parameter* or *argument*. The `atan` function takes one argument, and the computed value is *returned* from `atan`. This means that where we see `atan(y/x)`, a computation is performed ($\tan^{-1}(y/x)$) and the result "replaces" the text `atan(y/x)`. This is actually no more magic than if we had written just `y/x`: then the computation of `y/x` would take place, and the result of that division would replace the text `y/x`. Thereafter, the result is assigned to the name `angle` on the left-hand side of =.

Note that the trigonometric functions, such as `atan`, work with angles in radians. The return value of `atan` must hence be converted to degrees, and that is why we perform the computation `(angle/pi)*180`. Two things happen in the `print` statement: first, the computation of `(angle/pi)*180` is performed, resulting in a real number, and second, `print` prints that real number. Again, we may think that the arithmetic expression is replaced by its results and then `print` starts working with that result.

If we next execute `ball_angle_first_try.py`, we get an error message on the screen saying

```
NameError: name 'atan' is not defined
WARNING: Failure executing file: <ball_angle_first_try.py>
```

We have definitely run into trouble, but why? We are told that

```
name 'atan' is not defined
```

so apparently Python does not recognize this part of the code as anything familiar. On a pocket calculator the inverse tangent function is straightforward to use in a similar way as we have written in the code. In Python, however, this function has not yet been *imported* into the program. A lot of functionality is available to us in a program, but much more functionality exists in Python *libraries*, and to activate this functionality, we must explicitly import it. In Python, the `atan` function is grouped together with many other mathematical functions in the library called `math`. Such a library is referred to as a *module* in correct Python language. To get access to `atan` in our program we have to write

```
from math import atan
```

Inserting this statement at the top of the program and rerunning it, leads to a new problem: `pi` is not defined. The variable `pi`, representing π, is also available in the `math` module, but it has to be imported too:

```
from math import atan, pi
```

It is tedious if you need quite some `math` functions and variables in your program, e.g., also `sin`, `cos`, `log`, `exp`, and so on. A quick way of importing everything in `math` at once, is

```
from math import *
```

We will often use this import statement and then get access to all common mathematical functions. This latter statement is inserted in a program named ball_angle.py:

```
from math import *

x = 10              # Horizontal position
y = 10              # Vertical position

angle = atan(y/x)

print (angle/pi)*180
```

This program runs perfectly and produces 45.0 as output, as it should.

At first, it may seem cumbersome to have code in libraries, since you have to know which library to import to get the desired functionality. Having everything available anytime would be convenient, but this also means that you fill up the memory of your program with a lot of information that you rather would use for computations on big data. Python has so many libraries with so much functionality that one simply needs to import what is needed in a specific program.

1.4 A Python Program with Vectorization and Plotting

We return to the problem where a ball is thrown up in the air and we have a formula for the vertical position y of the ball. Say we are interested in y at every millisecond for the first second of the flight. This requires repeating the calculation of $y = v_0 t - 0.5gt^2$ one thousand times.

We will also draw a graph of y versus t for $t \in [0, 1]$. Drawing such graphs on a computer essentially means drawing straight lines between points on the curve, so we need many points to make the visual impression of a smooth curve. With one thousand points, as we aim to compute here, the curve looks indeed very smooth.

In Python, the calculations and the visualization of the curve may be done with the program ball_plot.py, reading

```
from numpy import linspace
import matplotlib.pyplot as plt

v0 = 5
g = 9.81
t = linspace(0, 1, 1001)

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

plt.plot(t, y)
plt.xlabel('t (s)')
plt.ylabel('y (m)')
plt.show()
```

This program produces a plot of the vertical position with time, as seen in Figure 1.1. As you notice, the code lines from the `ball.py` program in Chapter 1.2 have not changed much, but the height is now computed and plotted for a thousand points in time!

Let us take a look at the differences between the new program and our previous program. From the top, the first difference we notice are the lines

```
from numpy import *
from matplotlib.pyplot import *
```

You see the word `import` here, so you understand that `numpy` must be a library, or module in Python terminology. This library contains a lot of very useful functionality for mathematical computing, while the `matplotlib.pyplot` module contains functionality for plotting curves. The above import statement constitutes a quick way of populating your program with all necessary functionality for mathematical computing and plotting. However, we actually make use of only a few functions in the present program: `linspace`, `plot`, `xlabel`, and `ylabel`. Many computer scientists will therefore argue that we should explicitly import what we need and not everything (the star `*`):

```
from numpy import linspace
from matplotlib.pyplot import plot, xlabel, ylabel
```

Others will claim that we should do a slightly different import and prefix library functions by the library name:

```
import numpy as np
import matplotlib.pyplot as plt
...
t = np.linspace(0, 1, 1001)
...
plt.plot(t, y)
plt.xlabel('t (s)')
plt.ylabel('y (m)')
```

We will use all three techniques, and since all of them are in so widespread use, you should be familiar with them too. However, for the most part in this book we shall do

```
from numpy import *
from matplotlib.pyplot import *
```

for convenience and for making Python programs that look very similar to their Matlab counterparts.

The function `linspace` takes 3 parameters, and is generally called as

```
linspace(start, stop, n)
```

This is our first example of a Python function that takes multiple arguments. The `linspace` function generates `n` equally spaced coordinates, starting with `start`

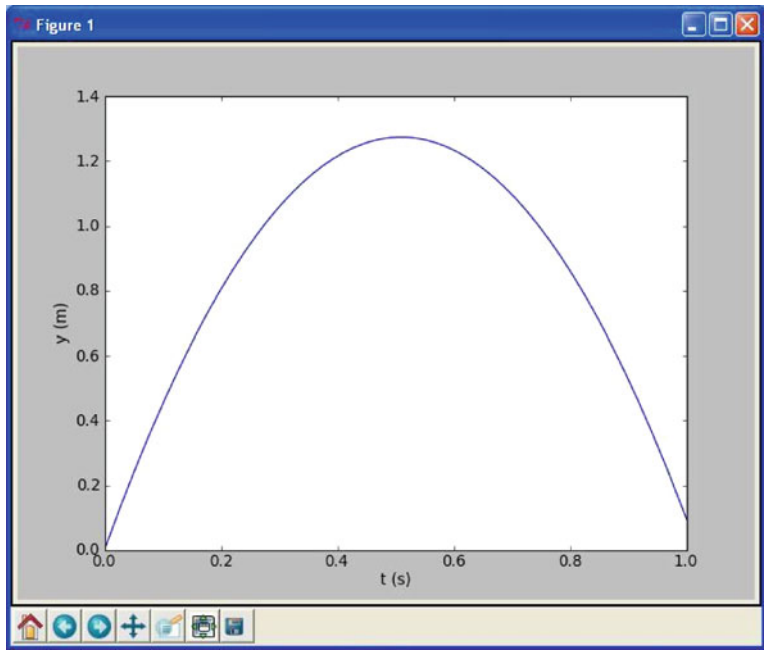


Fig. 1.1 Plot generated by the script `ball_plot.py` showing the vertical position of the ball at a thousand points in time

and ending with `stop`. The expression `linspace(0, 1, 1001)` creates 1001 coordinates between 0 and 1 (including both 0 and 1). The mathematically inclined reader will notice that 1001 coordinates correspond to 1000 equal-sized intervals in $[0, 1]$ and that the coordinates are then given by $t_i = i/1000$ ($i = 0, 1, \ldots, 1000$).

The value returned from `linspace` (being stored in `t`) is an *array*, i.e., a collection of numbers. When we start computing with this collection of numbers in the arithmetic expression `v0*t - 0.5*g*t**2`, the expression is calculated for every number in `t` (i.e., every t_i for $i = 0, 1, \ldots, 1000$), yielding a similar collection of 1001 numbers in the result `y`. That is, `y` is also an array.

This technique of computing all numbers "in one chunk" is referred to as *vectorization*. When it can be used, it is very handy, since both the amount of code and computation time is reduced compared to writing a corresponding `for` or `while` loop (Chapter 2) for doing the same thing.

The plotting commands are simple:

1. `plot(t, y)` means plotting all the `y` coordinates versus all the `t` coordinates
2. `xlabel('t (s)')` places the text `t (s)` on the x axis
3. `ylabel('y (m)')` places the text `y (m)` on the y axis

At this stage, you are strongly encouraged to do Exercise 1.4. It builds on the example above, but is much simpler both with respect to the mathematics and the amount of numbers involved.