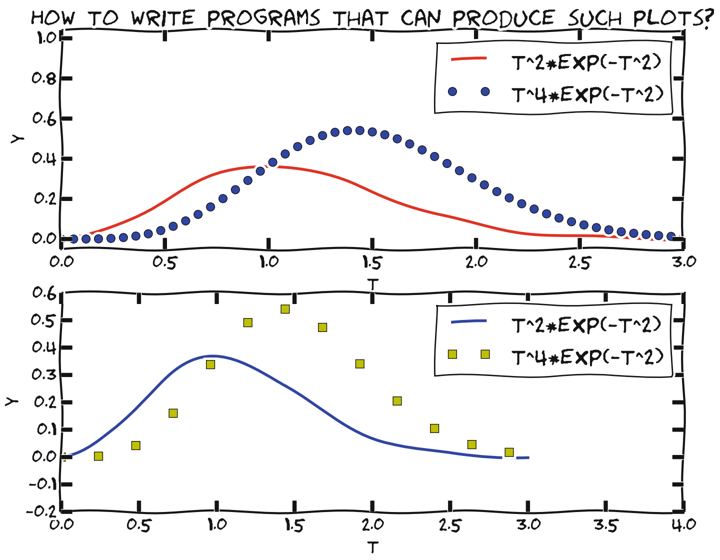
The First Few Steps

1



1.1 What Is a Program? And What Is Programming?

Computer Programs 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.

S. Linge, H. P. Langtangen, *Programming for Computations - Python*,
Texts in Computational Science and Engineering 15,
https://doi.org/10.1007/978-3-030-16877-3_1

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 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.

Programming The real power of computers can only be utilized if you can program them, i.e., write the programs yourself. With programming, *you* can tell the computer what *you* want it to do, which is great, since it frees you from possible limitations that come with programs written by others! Thus, with this skill, you get an important extra option for problem solving that goes beyond what ready-made programs offer.

A program that you write, will be a set of instructions that you store in a file. These instructions must be written (according to certain rules) in a very specialized language that has adopted words and expressions from English. Such languages are known as *programming* (or *computer*) *languages*. When you have written your instructions (your program), you may ask the programming language to read your program and carry out the instructions. The programming language will then (if there are no errors) translate the meaning of your instructions into real actions inside the computer.

To write a program that solves a computing problem, you need to have a thorough understanding of the given problem. That understanding may have to be *developed* along the way and will obviously guide the way you write your solution program. Typically, you need to write, test and re-write your program several times until you get it right. Thus, what starts out with a computing problem and ends with a sensible computer program for its solution, is a *process* that may take some time. By the term *programming*, we will mean the whole of this process.

The purpose of this book is to teach you how to develop computer programs dedicated to solve mathematical and engineering problems by fundamental numerical methods.

Programming Languages There are numerous computer languages for different purposes. Within the engineering area, the most widely used ones are Python, MATLAB, Octave, Fortran, C, C++, and to some extent, Maple and Mathematica. The rules for how to write the instructions (i.e. the *syntax*) differ 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.

We Use Python 3.6 in This Book For good reasons, the programming language used in this book is Python (version 3.6). It is an excellent language for beginners (and experts!), with a simple and clear syntax. Some of Python's other strong properties are[1]: It has a huge and excellent *library* (i.e., ready-made pieces of code that you can utilize for certain tasks in your own programs), 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 at no cost from the Internet and will run on most platforms.

A Primer on Scientific Programming with Python

Readers who want to expand their scientific programming skills beyond the introductory level of the present exposition, are encouraged to study *A Primer on Scientific Programming with Python* [11]. 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. Note, however, that this reference [11] uses version 2.7 of Python, which means that, in a few cases, instructions will differ somewhat from what you find in the present book.

[1] Some of the words here will be new to you, but relax, they will all be explained as we move along.

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[a]): *Suppose you went back to Ada Lovelace[b] 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 as well. For example, writing a program (or script or code) is often expressed as *implementing* the program. *Executing* a program means running the program. A *default value* is what will be used if nothing is specified. 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* (see, e.g., Wikipedia[c]). *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. A *plot* is a graphical representation of a data set. For example, if you walk along a straight road, recording your position y with time t, say every second, your data set will consist of pairs with corresponding y and t values. With two perpendicular axes in a plane (e.g., a computer screen or a sheet of paper), one "horizontal" axis for t and one "vertical" axis for y, each pair of points could be marked in that plane. The axes and the points make up a plot, which represents the data set graphically. Usually, such plotting is done by a computer.

[a] http://www.perl.com/pub/2007/12/06/soto-11.html.
[b] http://en.wikipedia.org/wiki/Ada_Lovelace.
[c] http://en.wikipedia.org/wiki/Software_bug#Etymology.

1.1.1 Installing Python

To study this book, you need a Python installation that fits the purpose. The quickest way to get a useful Python installation on your Windows, Mac, or Linux computer, is to download and install Anaconda.[2] There are alternatives (as you can find on the internet), but we have had very good experiences with Anaconda for several years, so that is our first choice. No separate installation of Python or Spyder (our recommended environment for writing Python code) is then required, as they are both included in Anaconda.

[2] https://www.anaconda.com/distribution.

To download Anaconda, you must pick the Anaconda version suitable for your machine (Windows/Mac/Linux) and choose which version of Python you want (3.6 is used for this book). When the download has completed, proceed with the installation of Anaconda.

After installation, you may want to (search for and) start up Spyder to see what it looks like (see also Appendix A). Spyder is an excellent tool for developing Python code. So, unless you have good reasons to choose otherwise, we recommend Spyder to be your main "working" environment, meaning that to read, write and run code you start Spyder and do it there. Thus, it is a good idea to make Spyder easy accessible on your machine.

With Anaconda installed, the only additional package you need to install is Odespy.[3] Odespy is relevant for the solving of differential equations that we treat in Chaps. 8 and 9.

In Appendix A you will find more information on the installation and use of Python.

1.2 A Python Program with Variables

Our first example regards programming a *mathematical model* that predicts the height of a ball thrown straight up into 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:

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

Here, v_0 is the initial upwards velocity and g is the acceleration of gravity, for which $9.81\ \mathrm{ms}^{-2}$ 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.

1.2.1 The Program

Let us next look at a Python program for evaluating this simple formula. To do this, we need some values for v_0 and t, so we pick $v_0 = 5\,\mathrm{ms}^{-1}$ and $t = 0.6\,\mathrm{s}$ (other choices would of course have been just as good). Assume the program is contained as text in a file named `ball.py`, reading

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

v0 = 5                  # Initial velocity
g = 9.81                # Acceleration of gravity
```

[3] The original version of Odespy (https://github.com/hplgit/odespy) was written in Python 2.7 by H.P. Langtangen and L. Wang. However, since the sad loss of Prof. Langtangen in 2016, Odespy has been updated to Python 3.6 (https://github.com/thomasantony/odespy/tree/py36/odespy), thanks to Thomas Antony. This version is the one used herein.

```
t = 0.6             # Time

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

print(y)
```

Let us now explain this program in full detail.

Typesetting of Code Computer programs, and parts of programs, are typeset with a blue background in this book. When a complete program is shown, the blue background has a slightly darker top and bottom bar (as for `ball.py` here). Without the bars, the code is just a *snippet* and will normally need additional lines to run properly.

We also use the blue background, without bars, for *interactive* sessions (Sect. 2.1).

1.2.2 Dissecting the Program

A computer program like `ball.py` contains instructions to the computer written as plain text. 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.

One important reason for this strong demand 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. Also, you will often need to understand code written by other people. If you are able to understand their code properly, you may modify and use it as it suits you.

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. After this assignment, any appearance of `v0` in the code will "represent" the initial velocity, being 5 ms^{-1} in this case. This means that, whenever Python reads `v0`, it will *replace* `v0` by the integer value 5. One simple way to think of this, might be as follows. With the assignment `v0 = 5`, Python generates a "box" in computer memory with the name `v0` written on top. The number 5 is then put into that box. Whenever Python later meets the name `v0` in the code, it finds the box, opens it, takes out the number (here 5) and replaces the name `v0` with the number.

The next two lines

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

are also assignment statements, giving two more "boxes" in computer memory. The box named `g` will contain the value 9.81, while the box named `t` contains 0.6. Similarly, when Python later reads `g` and `t` in the code, it plugs in the numerical values found in the corresponding boxes.

The assignments in a bit more detail

When Python interprets the assignment statement `v0 = 5`, the integer 5 becomes an *object* of *type int* and the variable name on the left-hand side becomes a named *reference* for that object. Similarly, when interpreting the assignment statements `g = 9.81` and `t = 0.6`, `g` and `t` become named references to objects created for the real numbers given. However, since we have real numbers, these objects will be of *type float* (in computer language, a real number is called a "floating point number").

Now, with these assignments in place, 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 exponentiation (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 is a `print` *function call*, which makes Python print the value of `y` on the screen.[4] Simply stated, the value of `y` is sent to a ready-made piece of code named `print` (being a *function*—see Chap. 4, here called with a single *argument* named `y`), which then takes care of the printing. Thus, when `ball.py` is run, the number 1.2342 appears on the screen.

Readability and Coding Style 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. Similarly, you notice that spaces are introduced to each side of − in the "formula" and to each side of `=` in the assignments. These spaces are not required, i.e., Python will understand perfectly well without them. However, they contribute to readability and it is *recommended* to use them[5] as part of good coding style.[6] Had there been a `+` sign in there, it too should have a space to each side. To the contrary, no extra spaces are recommended for `/`, `*` and `**`.

Several Statements on One Line Note that it's allowed to have several statements on the same line if they are separated by a semi-colon. So, with our program here, we could have written, e.g.,

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

# v0 is the intial velocity, g is the acceleration of gravity, t is time
v0 = 5; g = 9.81; t = 0.6

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

print(y)
```

In general, however, readability is easily degraded this way, e.g., making commenting more difficult, so it should be done with care.

Assignments like a=2*a

Frequently, you will meet assignment statements in which the variable name on the left hand side (of `=`) *also* appears in the expression on the right hand side. Take, e.g., `a = 2*a`. Python would then, according to its rules, *first* compute the expression on the right hand side with the current value of `a` and *then* let the result become the updated value of `a` through the assignment (the updated value of `a` is placed in a new "box" in computer memory).

[4] In Python 2.7, this would have been a *print command* reading `print y`.
[5] Be aware that in certain situations programmers do skip such spaces, e.g., when listing arguments in function calls, as you will learn more about in Chap. 4.
[6] You might like to check out the style guide for Python coding at https://www.python.org/dev/peps/pep-0008/.

1.2.3 Why Use Variables?

But why do we introduce variables at all? Would it not be just as fine, or even simpler, to just use the numbers directly in the formula?

If we did, using the same numerical values, `ball.py` would become even shorter, reading

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

y = 5*0.6 - 0.5*9.81*0.6**2         # vertical position

print(y)
```

What is wrong with this? After all, we do get the correct result when running the code!

Coding and Mathematical Formulation If you compare this coded formula with the corresponding mathematical formulation

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

the equivalence between code and mathematics is *not* as clear now as in our original program `ball.py`, where the formula was coded as

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

In our little example here, this may not seem dramatic. Generally, however, you better believe that when, e.g., trying to find errors in code that lacks clear equivalence to the corresponding mathematical formulation, human code interpretation typically gets *much* harder and it might take you a while to track down those bugs!

Changing Numerical Values In addition, if we would like to redo the computation for another point in time, say $t = 0.9$ s, we would have to *change the code in two places* to arrive at the new code line

```
y = 5*0.9 - 0.5*9.81*0.9**2
```

You may think that this is not a problem, but imagine some other formula (and program) where the same number enters in a 100 places! Are you certain that you can do the right edit in all those places without any mistakes?[7] You should realize that by using a variable, you get away with *changing in one place* only! So, to change the point in time from 0.6 to 0.9 in our original program `ball.py`, we could simply change `t = 0.6` into `t = 0.9`. That would be it! Quick and much safer than editing in many places.

[7] Using the editor to replace 0.6 in all places might seem like a quick fix, but you would have to be sure you did not change 0.6 in the wrong places. For example, another number in the code, e.g. 0.666, could easily be turned into 0.966, unless you were careful.

1.2.4 Mathematical Notation Versus Coding

Make sure you understand that, from the outset, we had a pure mathematical formulation of our formula

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

which does not contain any connection to programming at all. Remember, this formula was derived hundreds of years ago, long before computers entered the scene! When we next wrote a piece of code that applied this formula, that code had to *obey* the rules of the programming language, which in this case is Python. This means, for example, that multiplication *had to* be written with a star, simply because that is the way multiplication is coded in Python. In some other programming language, the multiplication could in principle have been coded otherwise, but the mathematical formulation would still read the same.

We have seen how the equals sign (=) is interpreted in Python code. This interpretation is very different from the interpretation in mathematics, as might be illustrated by the following little example. In mathematics, $x = 2 - x$ would imply that $2x = 2$, giving $x = 1$. In Python code, however, a code line like `x = 2 - x` would be interpreted, *not* as an equation, but rather as an assignment statement: compute the right hand side by subtracting the current value of `x` from 2 and let the result be the new value of `x`. In the code, the new value of `x` could thus be anything, all depending on the value `x` had above the assignment statement!

1.2.5 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. In fact, this is very much like learning to swim. Nobody can do that by just reading about it! You simply have to practice real swimming to get good at it. Therefore, it is crucial at this stage that you start writing and running Python programs. We just went through the program `ball.py` above, so let us next write and run that code.

But first a warning: there are many things that must come together in the right way for `ball.py` to run correctly. 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. Very often the guy next to you experienced the same problem and has already fixed it!

Start up Spyder and, in the editor (left pane), type in each line of the program `ball.py` shown earlier. Then you save the program (select `File` -> `save as`) where you prefer and finally run it (select `Run` -> `Run`, ... or press F5). With a bit of luck, you should now get the number 1.2342 out in the rightmost lower pane in the Spyder GUI. If so, congratulations, you have just executed your first self-written computer program in Python!

The documentation for Spyder[8] might be useful to you. Also, more information on writing and running Python programs is found in Appendix A.4 herein.

Why not a pocket calculator instead?

Certainly, finding the answer as with 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 4 h!

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.

An even 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.

Write programs with a text editor

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 what 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.

[8] See, e.g., https://www.spyder-ide.org/.

What about units?

The observant reader has noticed that the handling of quantities in `ball.py` did not include *units*, even though velocity (`v0`), acceleration (`g`) and time (`t`) of course do have the units of ms^{-1}, ms^{-2}, and s, respectively. Even though there are tools[a] in Python to include units, it is usually considered out of scope in a beginner's book on programming. So also in this book.

[a] See, e.g., https://github.com/juhasch/PhysicalQuantities, https://github.com/hgrecco/pint and https://github.com/hplgit/parampool if you are curious.

1.3 A Python Program with a Library Function

Imagine you stand on a distance, say 10.0 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 10.0 m. What is the angle of the line then?

Well, we do know (with, or without, a calculator) that the answer is 45°. However, when learning to code, it is generally a good idea to deal with *simple* problems with known answers. Simplicity ensures that the problem is well understood before writing any code. Also, knowing the answer allows an easy check on what your coding has produced when the program is run.

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 $\tan^{-1}(y/x)$.

The Program 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 = 10.0            # Horizontal position
y = 10.0            # 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 *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 angle on the left-hand side of =.

Note that the trigonometric functions, such as atan, work with angles in radians. Thus, if we want the answer in degrees, the return value of atan must be converted accordingly. This conversion is performed by the computation (angle/pi)*180. Two things happen in the print command: first, the computation of (angle/pi)*180 is performed, resulting in a number, and second, print prints that number. Again, we may think that the arithmetic expression is replaced by its result and then print starts working with that result.

Running the Program 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 is one of many that must be *imported* before use. A lot of functionality[9] is immediately available to us (from the *Python standard library*) as we start a new programming session, but much more functionality exists in additional Python libraries. To activate functionality from these libraries, we must explicitly import it. In Python, the atan function is grouped together with many other mathematical functions in a library *module* called math. To get access to atan in our program, we may write an *import statement*:

```
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 constant pi, representing π, is also available in the math module, but it has to be imported too. We can achieve this by including both items atan and pi in the import statement,

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

With this latter statement in place, we save the code as ball_angle.py:

```
from math import atan, pi

x = 10.0                # Horizontal position
y = 10.0                # Vertical position

angle = atan(y/x)

print((angle/pi)*180)
```

This script correctly produces 45.0 as output when executed.

[9] https://docs.python.org/3/library/functions.html.

Alternatively, we could use the import statement `import math`. This would require `atan` and `pi` to be *prefixed* with `math`, however, as shown in `ball_angle_prefix.py`:

```
import math

x = 10.0                # Horizontal position
y = 10.0                # Vertical position

angle = math.atan(y/x)

print (angle/math.pi)*180
```

The essential difference between the two import techniques shown here, is the prefix required by the latter. Both techniques are commonly used and represent the two basic ways of importing library code in Python. Importing code is an evident part of Python programming, so we better shed some more light on it.

1.4 Importing from Modules and Packages

At first, it may seem cumbersome to have code in different libraries, since it means you have to know (or find out) what resides in which library.[10] Also, there are many libraries around in addition to the Python standard library itself. To your comfort, you come a long way with just a few libraries, and for easy reference, the handful of libraries used in this book is listed below (Sect. 1.4.5). Having everything available at any time would be convenient, but this would also mean that computer memory would be filled with a lot of unused information, causing less memory to be available for computations on big data. Python has so many libraries, with so much functionality, that importing what is needed is indeed a very sensible strategy.

Where to Place Import Statements? The general recommendation is to place import statements at the top of a program, making them easy to spot.

1.4.1 Importing for Use *Without* Prefix

The need to prefix item names is avoided when import statements are on the form

```
from some_library import ...    # i.e., items will be used without prefix
```

as we saw in `ball_angle.py` above. Without prefixing, coded formulas often become easier to read, since code generally comes "closer" to mathematical writing. On the other hand, it is less evident where imported items "come from" and this may complicate matters, particularly when trying to understand more comprehensive code written by others.

[10] There is a built-in function called `dir`, which gives all the names defined in a library module. Another built-in function called `help` prints documentation. To see how they work, write (in Spyder, pane down to the right) `import math` followed by `dir(math)` or `help(math)`.

Importing Individual Items With `ball_angle.py`, we just learned that the import statement

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

made the `atan` function and `pi` available to the program. To bring in even more functionality from `math`, the import statement could simply have been extended with the relevant items, say

```
from math import atan, pi, sin, cos, log
```

and so on.

Having Several Import Statements Very often, we need to import functionality from several libraries. This is straight forward, as we may show by combining imports from `math` with imports from the useful *Numerical Python* (or NumPy) library,[11] named `numpy` in Python:

```
from math import atan, pi, sin, cos, log
from numpy import zeros, linspace
```

Right now, do not worry what the functions `zeros` and `linspace` do, we will explain and use them soon.

Importing All Items with "Import *" The approach of importing individual items (`atan`, `pi`, etc.) might appear less attractive if you need many of them in your program. There is another way, though, but it should be used with care, for reasons to be explained. In fact, many programmers will advice you *not* to use it at all, unless you know very well what you are doing. With this import technique, the list of items in the import statement is exchanged with simply a star (i.e., `*`). The import statement then appears as

```
from some_library import *    # import all items from some_library
```

which with the `math` library reads

```
from math import *           # import all items from math
```

This will cause *all* items from `math` to be imported, however, also the ones you do *not* need! So, with this "lazy" import technique, Python has to deal with a lot of names that are not used. Like when importing individual items, items are used without prefix.

Disadvantage: No Prefix Allows Name Conflicts! When importing so that items are written without prefix, there is a potential problem with *name conflicts*. Let us illustrate the point with a simple example. Assume that, being new to Python, we want to write a little program for checking out some of the functions that the language has got to offer.

Our first candidate could be the exponential function and we might like to compute and print out e^t for $t = 0, 1, 2$. A fellow student explains us how a function `exp` in the `numpy` library allows our calculations to be done with a single function

[11] The NumPy library (http://www.numpy.org/) is included in Anaconda. If you have not installed Anaconda, you may have to install NumPy separately.

call. This sounds good to us, so based on what we were told, we start writing our program as

```
from numpy import exp

x = exp([0, 1, 2])                # do all 3 calculations
print(x)                          # print all 3 results
```

The script runs without any problems and the printed numbers seem fine,

```
[ 1.          2.71828183  7.3890561 ]
```

Moving on, we want to test a number of functions from the `math` library (`cos`, `sin`, `tan`, etc.). Since we foresee testing quite many functions, we choose the "lazy" import technique and plan to extend the code with one function at a time.

Extending the program with a simple usage of the `cos` function, it reads

```
from numpy import exp
from math import *

x = exp([0, 1, 2])                # do all 3 calculations
print(x)                          # print all 3 results

y = cos(0)
print(y)
```

Running this version of the script, however, we get a longer printout (most of which is irrelevant at this point) ending with

```
TypeError: a float is required
```

Clearly, something has gone wrong! But why?

The explanation is the following. With the second import statement, i.e.,

```
from math import *
```

all items from `math` were imported, including a function from `math` that is *also* named `exp`. That is, there are two functions in play here that both go by the name `exp`! One `exp` function is found in the `numpy` library, while the other `exp` function is found in the `math` library, and the implementations of these two are *different*. This now becomes a problem, since the last imported `exp` function silently "takes the place" of the previous one, so that the name `exp` hereafter will be associated with the `exp` function from `math`! Thus, when Python interprets `x = exp([0, 1, 2])`, it tries to use `exp` from `math` for the calculations, but that version of `exp` can only take a single number (real or integer) as input argument, not several (as `exp` from `numpy` can). This mismatch then triggers the error message[12] and causes program execution to stop before reaching `y = cos(0)`.

Similar name conflicts may arise also with other functions than `exp`, since a lot of items appear with identical names in different libraries (e.g., also `cos`, `sin`, `tan`, and many more, exist with different implementations in both `numpy` and `math`). The fact that programmers may create, and share, their own libraries containing self

[12] It should be mentioned here, that error messages can not always be very accurate. With some experience, however, you will find them very helpful at many occasions. More about error messages later (Sect. 1.7).

chosen item names, makes it even more obvious that “name conflicts” is an issue that should be understood.

Several other coding alternatives would have helped the situation here. For example, instead of `from math import *`, we could switch the star (`*`) with a list of item names, i.e. as `from math import cos` for the present version. As long as we stay away from (by a mistake) importing `exp` also from `math`, no name conflict will occur and the program will run fine. Alternatively, we could simply have switched the order of the import statements (can you explain[13] why?), or, we could have moved the import statement `from math import *` down, so that it comes *after* the statement `x = exp([0, 1, 2])` and *before* the line `y = cos(0)`. Note that, in Python 3, import statements on the form `from module import *` are only allowed at module level, i.e., when placed inside functions, they give an error message.

Next, we will address the safer “standard” way of importing.

1.4.2 Importing for Use *with* Prefix

A safer implementation of our program would use the “standard” method of importing, which we saw a glimpse of in `ball_angle_prefix.py` above. With this import technique, the code would read

```
import numpy
import math

x = numpy.exp([0, 1, 2])              # do all 3 calculations
print(x)                              # print all 3 results

y = math.cos(0)
print(y)
```

We note that the import statements are on the form

```
import some_library      # i.e., items will be used with prefix
```

and that item names belonging to `some_library` are *prefixed* with `some_library` and a “dot”. This means that, e.g., `numpy.exp([0, 1, 2])` refers to the (unique) `exp` function from the `numpy` library. When the import statements are on the “standard” form, the prefix is required. Leaving it out gives an error message. This version of the program runs fine, producing the expected output.

With the prefixing method, the order of imports does not matter, as there is no doubt where the functions (or items) come from. At the same time, though, it is clear that prefixing does *not* make it any easier for a human to read the “math meaning” out of the code. In mathematical writing, there would be no prefix, so a prefix will just complicate the job for a human interpreter, and more so the more comprehensive the expressions are.

[13] By switching the order, Python would first read `from math import *` and would import everything, including `exp`, from `math`. Then, it would read `from numpy import exp`, which would cause Python to import the `numpy` version of `exp`, which effectively means that the `math` version of `exp` is “overwritten” by the one from `numpy`. At any later point in the code then, Python will associate the word `exp` with the `numpy` function.

1.4.3 Imports with Name Change

Whether we import for use with or without prefix, we may *change names* of the imported items by minor adjustments of the import statements. Introducing such name changes in our program and saving this final version as `check_functions.py`, it reads

```
import numpy as np
import math as m

x = np.exp([0, 1, 2])              # do all 3 calculations
print(x)                           # print all 3 results

y = m.cos(0)
print(y)
```

Effectively, the module names in this program now become `np` and `m` (by our own choice) instead of `numpy` and `math`, respectively. We still enjoy the safety of prefixing and notice that such name changes might bring computer coded expressions closer to mathematical writing and thus ease human interpretation.

When importing library items for use without prefix, name changes can be done, e.g., like

```
from math import cos as c, sin as s

print(c(0) + s(0))
```

1.4.4 Importing from Packages

Modules may be grouped into *packages*, often together with functions, variables, and more. We may import items (modules, functions, etc.) from such packages also, but the appearance of an import statement will then depend on the structure of the package in question. We leave out the details[14] and just exemplify with two packages often used in this book.

The `numpy` library used above *is*, in fact, a package and we saw how it could be used with different import statements, just as if it had been a module. Note that the import statement

```
import numpy as np           # standard way of importing numpy
```

is the standard way of importing `numpy`, i.e., also the "nickname" `np` is standard. This will be the standard import technique for `numpy` also in our book, meaning that we will generally use `numpy` items with the `np` prefix. We will deviate from this at times, typically during brief interactive sessions (see Sect. 2.1), in which case we will import items explicitly specified by name.

Another popular package you will meet often in this book, is the plotting library `matplotlib` (Sect. 1.5), used for generating and handling plots. The standard import statement, including the "nickname", is then

```
import matplotlib.pyplot as plt       # standard way of importing pyplot
```

[14] If you are curious, check for more details at https://docs.python.org/3.6/tutorial/modules.html.

Here, pyplot is a module in the matplotlib package[15] that is named plt when imported. Thus, when imported this way, all items from pyplot must have the prefix plt. We will stick to this import and naming standard for pyplot also in the present book, whenever plotting is on the agenda.

1.4.5 The Modules/Packages Used in This Book

Some readers might be curious to know which libraries are used in this book (apart from the modules we make ourselves). Well, here they are:

- math—see, e.g., ball_angle.py, Sect. 1.3.
- numpy—see, e.g., check_functions.py above.
- matplotlib.pyplot—see, e.g., ball_plot.py, Sect. 1.5.
- random—see, e.g., throw_2_dice.py in Sect. 2.4.
- sympy—see, e.g., Sect. 5.3.
- timeit—see, e.g., Sect. 5.6.
- sys—see, e.g., Sect. 7.2.2.

These libraries are all well known to Python programmers. The three first ones (math, numpy and matplotlib.pyplot) are used repeatedly throughout the text, while the remaining ones (random, sympy, timeit and sys) appear just occasionally.

Not listed, are two modules that are used just once each, the keyword module (Sect. 2.2) and the os module (Sect. 9.2.4). It should be mentioned that we also use a package called odespy (Sect. 8.4.6), previously developed by one of the authors (Langtangen).

1.5 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 milli-second for the first second of the flight. This requires repeating the calculation of $y = v_0t - 0.5gt^2$ one thousand times. As we will see, the computed heights appear very informative when presented graphically with time, as opposed to a long printout of all the numbers.

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

```
import numpy as np
import matplotlib.pyplot as plt

v0 = 5
g = 9.81
```

[15] The matplotlib package (https://matplotlib.org/) comes with Anaconda. If you have not installed Anaconda, you may have to install matplotlib separately.

```
t = np.linspace(0, 1, 1001)

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

plt.plot(t, y)          # plots all y coordinates vs. all t coordinates
plt.xlabel('t (s)')     # places the text t (s) on x-axis
plt.ylabel('y (m)')     # places the text y (m) on y-axis
plt.show()              # displays the figure
```

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

Let us take a closer look at this program. At the top, we recognize the import statements

```
import numpy as np
import matplotlib.pyplot as plt
```

As we know by now, these statements imply that items from `numpy` and `matplotlib.pyplot` must be prefixed with `np` and `plt`, respectively.

The `linspace` Function Next, there is a call to the function `linspace` from the `numpy` library. When `n` evenly spaced floating point numbers are sought on an interval $[a, b]$, `linspace` may generally be called like this:

```
np.linspace(a, b, n)
```

This means that the call

```
t = np.linspace(0, 1, 1001)
```

creates 1001 coordinates between 0 and 1, inclusive at both ends. The mathematically inclined reader might agree that 1001 coordinates correspond to 1000 equal-

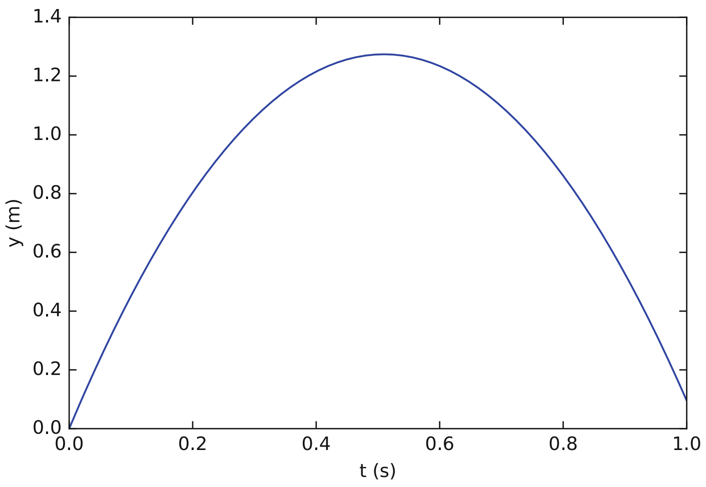


Fig. 1.1 Plot generated by the script `ball_plot.py` showing the vertical position of the ball (computed for a thousand points in time)

sized intervals in [0, 1] and that the coordinates are then given by $t_i = \frac{1-0}{1000} i = \frac{i}{1000}$, $i = 0, 1, \ldots, 1000$.

The object returned from `linspace` is an *array*, i.e., a certain collection of (in this case) numbers. Through the assignment, this array gets the name `t`. If we like, we may think of the array `t` as a collection of “boxes” in computer memory (each containing a number) that collectively go by the name `t` (later, we will demonstrate how these boxes are *numbered* consecutively from zero and upwards, so that each “box” may be identified and used individually).

Vectorization When we start computing with `t` in

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

the right hand side is computed 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`, which (automatically) also becomes 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 loop[16] (Chap. 3) for doing the same thing.

Plotting The plotting commands are new, but simple:

```
plt.plot(t, y)          # plots all y coordinates vs. all t coordinates
plt.xlabel('t (s)')     # places the text t (s) on x-axis
plt.ylabel('y (m)')     # places the text y (m) on y-axis
plt.show()              # displays the figure
```

At this stage, you are 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.

1.6 Plotting, Printing and Input Data

1.6.1 Plotting with Matplotlib

Often, computations and analyses produce data that are best illustrated graphically. Thus, programming languages usually have many good tools available for producing and working with plots, and Python is no exception.[17]

In this book, we shall stick to the excellent plotting library *Matplotlib*, which has become the standard plotting package in Python. Below, we demonstrate just a few of the possibilities that come with Matplotlib, much more information is found on the Matplotlib website.[18]

[16] It should be mentioned, though, that the computations are still done with loops “behind the scenes” (coded in C or Fortran). They generally run much quicker than the Python loops we write ourselves.
[17] In Sect. 9.2.4 we give a brief example of how plots may be turned into videos.
[18] https://matplotlib.org/index.html.

A Single Curve In Fig. 1.1, we saw a nice and smooth curve, showing how the height of a ball developed with time. The reader should realize that, even though the curve is continuous and *apparently* smooth, it is generated from a collection of points only. That is, *for the chosen points in time*, we have computed the height. For times in between, we have computed nothing! So, in principle, we actually do not know what the height is there. However, if only the time step between consecutive height computations is "small enough", the ball can not experience any significant change in its state of motion. Thus, inserting straight lines between two and two consecutive data points will be a good approximation. This is exactly what Python does, unless otherwise is specified. With "many" data points, as in Fig. 1.1, the curve appears smooth.

We saw previously, in `ball_plot.py`, how an array `y` (heights) could be plotted against another corresponding array `t` (points in time) with the statement

```
plt.plot(t, y)
```

A plot command like this is very typical and often just what we prefer, for example, in our case with the ball.

It is also possible, however, to plot an array without involving any second array at all. With reference to `ball_plot.py`, this means that `y` could have been plotted without any mention of `t`, and to do that, one could write the plot command rather like

```
plt.plot(y)
```

The curve would then have looked just like the one in Fig. 1.1, except that the x-axis would span the `y` array indices from 0 to 1000 instead of the corresponding points in time (check it and see for yourself).

Quickly testing a (minor) code change

Let us take the opportunity here, to mention how many programmers would go about to check the alternative plot command just mentioned. In `ball_plot.py`, one would typically just comment out the original lines and insert alternative code for these, i.e., as

```
#plt.plot(t, y)
#plt.xlabel('t (s)')
plt.plot(y)
plt.xlabel('Array indices')
```

One would then run the code and observe the impact of the change, which in this case is the modified plot described above.

After running the modified code, there are, generally, two alternatives. Should the original version be kept or should we make the change permanent? With the present ball example, most of us would prefer the original plot, so we would change the code back to its original form (remember to check that it works as before!).

When the code change to test is more comprehensive, it is much better to make a separate copy of the whole program, and then do the testing there.

The characteristics of a plotted line may also be changed in many ways with just minor modifications of the plot command. For example, a black line is achieved with

```
plt.plot(t, y, 'k')     # k - black, b - blue, r - red, g - green, ...
```

Other colors could be achieved by exchanging the k with certain other letters. For example, using b, you get a blue line, r gives a red line, while g makes the line green. In addition, the line style may be changed, either alone, or together with a color change. For example,

```
plt.plot(t, y, '--')     # default color, dashed line
```

```
plt.plot(t, y, 'r--')    # red and dashed line
```

```
plt.plot(t, y, 'g:')     # green and dotted line
```

Note that to avoid destroying a previously generated plot, you may precede your plot command by

```
plt.figure()
```

This causes a new figure to be created alongside any already present.

Plotting Points Only When there are not too many data points, it is sometimes desirable to plot each data point as a "point", rather than representing all the data points with a line. To illustrate, we may consider our case with the ball again, but this time computing the height each 0.1 s, rather than every millisecond. In ball_plot.py, we would then have to change our call to linspace into

```
t = np.linspace(0, 1, 11)   # 11 values give 10 intervals of 0.1
```

Note that we need to give 11 as the final argument here, since there will be 10 intervals of 0.1 s when 11 equally distributed values on [0, 1] are asked for. In addition, we would have to change the plot command to specify the plotting of data points as "points". To mark the points themselves, we may use one of many different alternatives, e.g., a circle (the lower case letter o) or a star (*). Using a star, for example, the plot command could read

```
plt.plot(t, y, '*')    # default color, points marked with *
```

With these changes, the plot from Fig. 1.1 would change as seen in Fig. 1.2.

Of course, not only can we choose between different kinds of point markers, but also their color may be specified. Some examples are:

```
plt.plot(t, y, 'r*')    # points marked with * in red
```

```
plt.plot(t, y, 'bo')    # points marked with o in blue
```

```
plt.plot(t, y, 'g+')    # points marked with + in green
```

When are the data points "too many" for plotting data points as points (and not as a line)? If plotting the data points with point markers and those markers *overlap* in the plot, the points will not appear as points, but rather as a very thick line. This is hardly what you want.

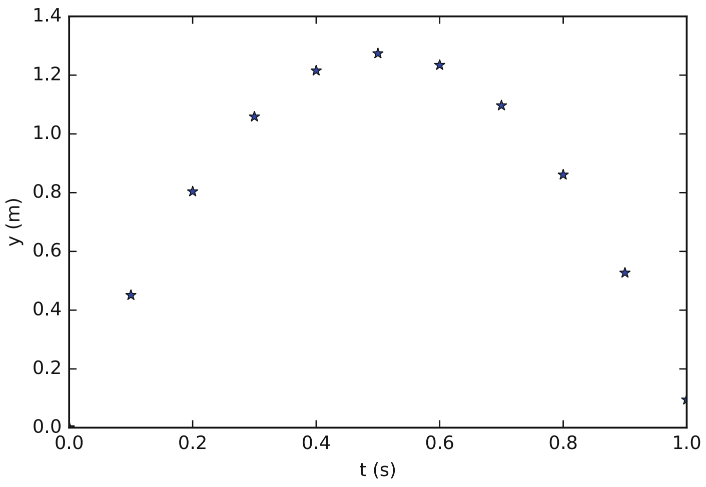


Fig. 1.2 Vertical position of the ball computed and plotted for every 0.1 s

Decorating a Plot We have seen how the code lines `plt.xlabel('t (s)')` and `plt.ylabel('y (m)')` in `ball_plot.py` put labels `t (s)` and `y (m)` on the t- and y-axis, respectively. There are other ways to enrich a plot as well.

One thing, is to add a *legend* so that the curve itself gets labeled. With `ball_plot.py`, we could get the legend `v0*t - 0.5*g*t**2`, for example, by coding

```
plt.legend(['v0*t - 0.5*g*t**2'])
```

When there is more than a single curve, a legend is particularly important of course (see section below on "multiple curves" for a plot example).

Another thing, is to add a *grid*. This is useful when you want a more detailed impression of the curve and may be coded in this way,

```
plt.grid('on')
```

A plot may also get a title on top. To get a title like `This is a great title`, for example, we could write

```
plt.title('This is a great title')
```

Sometimes, the default ranges appearing on the axes are not what you want them to be. This may then be specified by a code line like

```
plt.axis([0, 1.2, -0.2, 1.5])    # x in [0, 1.2] and y in [-0.2, 1.5]
```

All statements just explained will be demonstrated in the next section, when we show how multiple curves may be plotted together in a single plot.

Multiple Curves in the Same Plot Assume we want to plot $f(t) = t^2$ and $g(t) = e^t$ in the same plot for t on the interval $[-2, 2]$. The following script (`plot_multiple_curves.py`) will accomplish this task:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
t = np.linspace(-2, 2, 100)   # choose 100 points in time interval

f_values = t**2
g_values = np.exp(t)

plt.plot(t, f_values, 'r', t, g_values, 'b--')
plt.xlabel('t')
plt.ylabel('f and g')
plt.legend(['t**2', 'e**t'])
plt.title('Plotting of two functions (t**2 and e**t)')
plt.grid('on')
plt.axis([-3, 3, -1, 10])
plt.show()
```

In this code, you recognize the commands explained just above. Their impact on the plot may be seen in Fig. 1.3, which is produced when the program is executed.

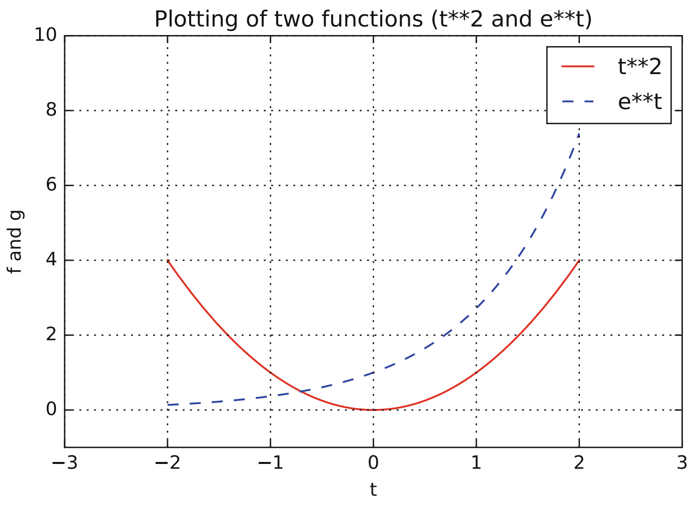


Fig. 1.3 The functions $f(t) = t^2$ and $g(t) = e^t$

In addition, you see how

```
plt.plot(t, f_values, 'r', t, g_values, 'b--')
```

causes *both* curves to be seen in the same plot. Notice the structure here, within the parenthesis, we first describe plotting of the one curve with `t, f_values, 'r'`, before plotting of the second curve is specified by `t, g_values, 'b--'`. These two "plot specifications" are separated by a comma. Had there been more curves to plot in the same plot, we would simply extend the list in a similar way. For each curve, color and line style is specified independently of the other curve specifications in the plot command (no specification gives default appearance). Furthermore, you notice how

```
plt.legend(['t**2', 'e**t'])
```

creates the right labelling of the curves. Note that the order of curve specifications in the plot command must be the same as the order of legend specifications in the legend command. In the plot command above, we first specify the plotting of `f_values` and then `g_values`. In the legend command, `t**2` should thus appear *before* `e**t` (as it does).

Multiple Plots in One Figure With the `subplot` command you may combine several plots into one. We may demonstrate this with the script `two_plots_one_fig.py`, which reproduces Figs. 1.2 and 1.3 as one:

```
import numpy as np
import matplotlib.pyplot as plt

plt.subplot(2, 1, 1)          # 2 rows, 1 column, plot number 1
v0 = 5
g = 9.81
t = np.linspace(0, 1, 11)
y = v0*t - 0.5*g*t**2
plt.plot(t, y, '*')
plt.xlabel('t (s)')
plt.ylabel('y (m)')
plt.title('Ball moving vertically')

plt.subplot(2, 1, 2)          # 2 rows, 1 column, plot number 2
t = np.linspace(-2, 2, 100)
f_values = t**2
g_values = np.exp(t)
plt.plot(t, f_values, 'r', t, g_values, 'b--')
plt.xlabel('t')
plt.ylabel('f and g')
plt.legend(['t**2', 'e**t'])
plt.title('Plotting of two functions (t**2 and e**t)')
plt.grid('on')
plt.axis([-3, 3, -1, 10])

plt.tight_layout()           # make subplots fit figure area
plt.show()
```

You observe that `subplot` appears in two places, first as `plt.subplot(2, 1, 1)`, then as `plt.subplot(2, 1, 2)`. This may be explained as follows. With a code line like

```
plt.subplot(r, c, n)
```

we tell Python that in an arrangement of `r` by `c` subplots, `r` being the number of rows and `c` being the number of columns, we address subplot number `n`, counted row-wise. So, in `two_plots_one_fig.py`, when we first write

```
plt.subplot(2, 1, 1)
```

Python understands that we want to plot in subplot number 1 in an arrangement with two rows and one column of subplots. Further down, Python interprets

```
plt.subplot(2, 1, 2)
```

and understands that plotting now is supposed to occur in subplot number 2 of the same arrangement.

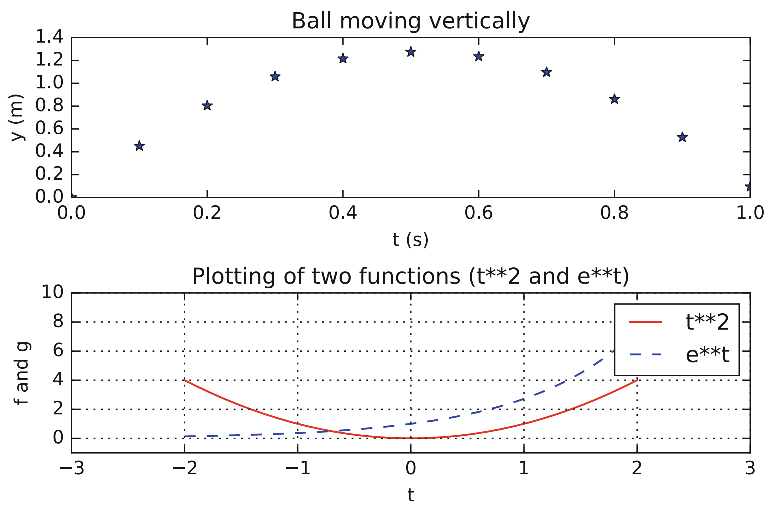


Fig. 1.4 Ball trajectory and functions $f(t) = t^2$ and $g(t) = e^t$ as two plots in one figure

Note that, when dealing with subplots, some overlapping of subplots may occur. Usually, this is solved nicely by inserting the following line (as at the end of our code),

```
plt.tight_layout()
```

This will cause subplot parameters to be automatically adjusted, so that the subplots fit in to the figure area.

The plot generated by the code is shown in Fig. 1.4.

Making a Hardcopy Saving a figure to file is achieved by

```
plt.savefig('some_plot.png')    # PNG format
plt.savefig('some_plot.pdf')    # PDF format
plt.savefig('some_plot.jpg')    # JPG format
plt.savefig('some_plot.eps')    # Encanspulated PostScript format
```

1.6.2 Printing: The String Format Method

We have previously seen that

```
print(y)
```

will print the value of the variable y. In an equally simple way, the line

```
print('This is some text')
```

will print `This is some text` (note the enclosing single quotes in the call to `print`). Often, however, it is of interest to print variable values together with some descriptive text. As shown below, such printing can be done nicely and controlled in Python, since the language basically allows text and numbers to be mixed and formatted in any way you need.

One Variable and Text Combined Assume there is a variable `v1` in your program, and that `v1` has the value 10.0, for example. If you want your code to print the value of `v1`, so that the printout reads

```
v1 is 10.0
```

you can achieve that with the following line in your program:

```python
print('v1 is {}'.format(v1))
```

This is a call to the function `print` with "an argument composed of two parts". The first part reads `v1 is {}` enclosed in single quotes (note the single quotes, they must be there!), while the second part is `.format(v1)`. The single quotes of the first part means that it is a *string* (alternatively, double quotes may be used).[19] That string contains a pair of curly brackets `{}`, which acts as a *placeholder*. The brackets tell Python *where* to place the value of, in this case, `v1`, as specified in the second part `.format(v1)`. So, the formatting creates the string `v1 is 10.0`, which then gets printed by `print`.

Several Variables and Text Combined Often, we have more variables to print, and with two variables `v1` and `v2`, we could print them by

```python
print('v1 is {}, v2 is {}'.format(v1, v2))
```

In this case, there are *two* placeholders `{}`, and—note the following: *the order* of `v1` and `v2` given in `.format(v1, v2)` will *dictate* the order in which values are filled into the preceding string. That is, reading the string from left to right, the value of `v1` is placed where the first `{}` is found, while the value of `v2` is placed where the second `{}` is located.

So, if `v1` and `v2` have values 10.0 and 20.0, respectively, the printout will read

```
v1 is 10.0, v2 is 20.0
```

When printing the values of several variables, it is often natural to use one line for each. This may be achieved by using `\n` as

```python
print('v1 is {} \nv2 is {}'.format(v1, v2))    # \n gives new line
```

which will produce

```
v1 is 10.0
v2 is 20.0
```

We could print the values of more variables by a straight forward extension of what was shown here.

Note that, if we had accidentally switched the order of the variables as

```python
print('v1 is {}, \nv2 is {}'.format(v2, v1))
```

where `.format(v2, v1)` is used instead of `.format(v1, v2)`), we would have got no error message, just an erroneous printout where the values are switched:

```
v1 is 20.0
v2 is 10.0
```

[19] Previously, we have met objects of type *int* and *float*. A string is an object of type *str*.

So, make sure the order of the arguments is correct. An alternative is to name the arguments.

Naming the Arguments If we name the arguments (`v1` and `v2` are arguments to `format`), we get the correct printout whether we call `print` as

```
print('v1 is {v1}, \nv2 is {v2}'.format(v1=v1, v2=v2))
```

or, switching the order,

```
print('v1 is {v1}, \nv2 is {v2}'.format(v2=v2, v1=v1))
```

Note that the names introduced do not have to be the same as the variable names, i.e., "any" names would do. Thus, if we (for the sake of demonstration) rather use the names `a` and `b`, any of the following calls to `print` would work just as fine (try it!):

```
print('v1 is {a}, \nv2 is {b}'.format(a=v1, b=v2))
```

or

```
print('v1 is {a}, \nv2 is {b}'.format(b=v2, a=v1))
```

Controlling the printout like we have demonstrated this far, may be sufficient in many cases. However, as we will see next, even more printing details can be controlled.

Formatting More Details Often, we want to control *how* numbers are formatted. For example, we may want to write 1/3 as `0.33` or `3.3333e-01` ($3.3333 \cdot 10^{-1}$), and as the following example will demonstrate, such details may indeed be specified in the argument to `print`. The essential new thing then, is that we supply the placeholders `{}` with some extra information in between the brackets.

Suppose we have a real number 12.89643, an integer 42, and a text `'some message'` that we want to write out in the following two different ways:

```
real=12.896, integer=42, string=some message
real=1.290e+01, integer=   42, string=some message
```

The real number is first to be written in *decimal notation* with three decimals, as `12.896`, but afterwards in *scientific notation* as `1.290e+01`. The integer should first be written as compactly as possible, while the second time, `42` should be placed in a five character wide text field.

The following program, `formatted_print.py`, produces the requested output:

```
r = 12.89643            # real number
i = 42                  # integer
s = 'some message'      # string    (equivalent: s = "some message")

print('real={:.3f}, integer={:d}, string={:s}'.format(r, i, s))
print('real={:9.3e}, integer={:5d}, string={:s}'.format(r, i, s))
```

Here, each placeholder carries a specification of what object *type* that will enter in the corresponding place, with `f` symbolizing a `float` (real number), `d` symbolizing an `int` (integer), and `s` symbolizing a `str` (string). Also, there is a specification of *how* each number is to be printed. Note the colon within the brackets, it must be there!

In the first call to `print`,

```
print('real={:.3f}, integer={:d}, string={:s}'.format(r, i, s))
```

`:.3f` tells Python that the floating point number `r` is to be written as compactly as possible in decimal notation with three decimals, `:d` tells Python that the integer `i` is to be written as compactly as possible, and `:s` tells Python to write the string `s`.

In the second call to `print`,

```
print('real={:9.3e}, integer={:5d}, string={:s}'.format(r, i, s))
```

the interpretation is the same, except that `r` and `i` now should be formatted according to `:9.3e` and `:5d`, respectively. For `r`, this means that its float value will be written in scientific notation (the `e`) with 3 decimals, in a field that has a width of 9 characters. As for `i`, its integer value will be written in a field of width 5 characters.

Other ways of formatting the numbers would also have been possible.[20] For example, specifying the printing of `r` rather as `:9.3f` (i.e., `f` instead of `e`), would give decimal notation, while with `:g`, Python itself would choose between scientific and decimal notation, automatically choosing the one resulting in the most compact output. Typically, scientific notation is appropriate for very small and very large numbers and decimal notation for the intermediate range.

Printing with old string formatting

There is another older string formatting that, when used with `print`, gives the same printout as the string format method. Since you will meet it frequently in Python programs found elsewhere, you better know about it. With this formatting, the calls to `print` in the previous example would rather read

```
print('real=%.3f, integer=%d, string=%s' % (r, i, s))
print('real=%9.3e, integer=%5d, string=%s' % (r, i, s))
```

As you might guess, the overall "structure" of the argument to `print` is the same as with the string format method, but, essentially, `%` is used instead of `{}` (with `:` inside) and `.format`.

An Example: Printing Nicely Aligned Columns A typical example of when formatted printing is required, arises when nicely aligned columns of numbers are to be printed. Suppose we want to print a column of t values together with associated function values $g(t) = t\sin(t)$ in a second column.

We could achieve this in the following way (note that, repeating the same set of statements multiple times, like we do in the following code, is *not* good programming practice—one should use a `loop`. You will learn about loops in Chap. 3.)

```
from math import sin

t0 = 2
```

[20] https://docs.python.org/3/library/string.html#format-string-syntax.

```
dt = 0.55

t = t0 + 0*dt; g = t*sin(t)
print('{:6.2f} {:8.3f}'.format(t, g))

t = t0 + 1*dt; g = t*sin(t)
print('{:6.2f} {:8.3f}'.format(t, g))

t = t0 + 2*dt; g = t*sin(t)
print('{:6.2f} {:8.3f}'.format(t, g))
```

Running this program, we get the printout

```
  2.00     1.819
  2.55     1.422
  3.10     0.129
```

Observe that the columns are nicely aligned here. With the formatting, we effectively control the width of each column and also the number of decimals. The numbers in each column will then become nicely aligned under each other and written with the same precision.

To the contrary, if we had skipped the detailed formatting, and rather used a simpler call to `print` like

```
print(t, g)
```

the columns would be printed as

```
2.0 1.81859485365
2.55 1.42209347935
3.1 0.128900053543
```

Observe that the nice and easy-to-read structure of the printout now is gone.

1.6.3 Printing: The f-String

We should briefly also mention printing by use of *f-strings*. Above, we printed the values of variables `v1` and `v2`, being 10.0 and 20.0, respectively. One of the calls we used to `print` was (repeated here for easy reference)

```
print('v1 is {} \nv2 is {}'.format(v1, v2))    # \n gives new line
```

and it produced the output

```
v1 is 10.0
v2 is 20.0
```

However, if we rather skip `.format(v1, v2)`, and instead introduce an `f` in front of the string, we can produce the very same output by the following simpler call to `print`:

```
print(f'v1 is {v1} \nv2 is {v2}')
```

So, f-strings[21] are quite handy!

[21] Read more about f-strings at https://www.python.org/dev/peps/pep-0498.

Printing Strings that Span Multiple Lines

A handy way to print strings that run over several lines, is to use *triple double-quotes* (or, alternatively, *triple single-quotes*) like this:

```
print("""This is a long string that will run over several lines
         if we just manage to fill in
         enough words.""")
```

The output will then read

```
This is a long string that will run over several lines
         if we just manage to fill in
         enough words.
```

1.6.4 User Input

Computer programs need a set of input data and the purpose is to use these data to compute output (data), i.e., results. We have previously seen how input data can be provided simply by assigning values to variables directly in the code. However, to change values then, one must change them in the program.

There are more flexible ways of handling input, however. For example through some dialogue with the user (i.e., the person running the program). Here is one example where the program asks a question, and the user provides an answer by typing on the keyboard:

```
age = int(input('What is your age? '))
print('Ok, so you're half way to {}, wow!'.format(age*2))
```

In the first line, there are two function calls, first to `input` and then to `int`. The function call `input('What is your age? ')` will cause the question "What is your age?" to appear in the lower right pane. When the user has (after left-clicking the pane) typed in an integer for the age and pressed `enter`, that integer *will* be returned by `input` as a *string* (since `input` always returns a string[22]). Thus, that string must be converted to an integer by calling `int`, before the assignment to `age` takes place.

So, after having interpreted and run the first line, Python has established the variable `age` and assigned your input to it. The second line combines the calculation of twice the age with a message printed on the screen. Try these two lines in a little test program to see for yourself how it works.

It is possible to get more flexibility into user communication by building a string before `input` shows it to the user. Adding a bit to the previous dialogue may illustrate how it works:

```
# ...assume the variable "name" contains name of user

message = 'Hello {:s}! What is your age? '.format(name)

age = int(input(message))
print('Ok, so you're half way to {}, wow!'.format(age*2))
```

[22] The `input` function here in Python 3.6, corresponds to the `raw_input` function in Python 2.7.

Thus, if the user name was Paul, for example, he would get this question up on his screen

```
Hello Paul! What is your age?
```

He would type his age, press enter, and the code would proceed like before.

There are other ways of providing input to a program as well, e.g., via a graphical interface (as many readers will be used to) or at the command line (i.e., as parameters succeeding, on the same line, the command that starts the program). Reading data from a file is yet another way.

1.7 Error Messages and Warnings

All programmers experience error messages, and usually to a large extent during the early learning process. Sometimes error messages are understandable, sometimes they are not. Anyway, it is important to get used to them.

One idea is to start with a program that initially is working, and then deliberately introduce errors in it, one by one (but remember to take a copy of the original working code!). For each error, you try to run the program to see what Python's response is. Then you know what the problem is and understand what the error message is about. This will greatly help you when you get a similar error message or warning later.

Debugging Very often, you will experience that there are errors in the program you have written. This is normal, but frustrating in the beginning. You then have to find the problem, try to fix it, and then run the program again. Typically, you fix one error just to experience that another error is waiting around the corner. However, after some time you start to avoid the most common beginner's errors, and things run more smoothly. The process of finding and fixing errors, called *debugging*, is very important to learn. There are different ways of doing it too.

A special program (*debugger*) may be used to help you check (and do) different things in the program you need to fix. A simpler procedure, that often brings you a long way, is to print information to the screen from different places in the program. First of all, this is something you should do (several times) during program development anyway, so that things get checked as you go along. However, if the final program still ends up with error messages, you may save a copy of it, and do some testing on the copy. Useful testing may then be to remove, e.g., the latter half of the program (e.g., by inserting comment signs #), and insert print commands at clever places to see what is the case. When the first half looks ok, possibly after some corrections, insert parts of what was removed and repeat the process with the new code. Using simple numbers and doing this in parallel with hand calculations on a piece of paper (for comparison) is often a very good idea.

Exception Handling Python also offers means to detect and handle errors by the program itself! The programmer must then foresee (when writing the code) that there is a potential for error at some particular point. If, for example, a running program asks the user to give a number, things may go very wrong if the user inputs the word *five* in stead of the number 5. In Python, such cases may be handled elegantly in the code, since it is possible to (put simply) *try* some statements, and if they go wrong, rather run some other code lines! This way, an *exception* is handled, and an unintended program stop (“crash”) is avoided. More about *exception handling* in Sect. 5.2.

Testing Code When a program finally runs without error messages, it might be tempting to think that *Ah..., I am finished!*. But no! Then comes program *testing*, you need to *verify* that the program does the computations as planned. This is almost an art and may take more time than to develop the program, but the program is useless unless you have much evidence showing that the computations are correct. Also, having a set of (automatic) tests saves huge amounts of time when you further develop the program.

Verification Versus Validation

Verification is important, but *validation* is equally important. It is great if your program can do the calculations according to the plan, *but* is it the right plan? Put otherwise, you need to check that the computations run correctly according to the *formula you have chosen/derived*. This is *verification*: doing the things right. Thereafter, you must also check whether the formula you have chosen/derived is *the right* formula for the case you are investigating. This is *validation*: doing the right things.

In the present book, it is beyond scope to question how well the mathematical models describe a given phenomenon in nature or engineering, as the answer usually involves extensive knowledge of the application area. We will therefore limit our testing to the verification part.

1.8 Concluding Remarks

1.8.1 Programming Demands You to Be Accurate!

In this chapter, you have seen some examples of how simple things may be done in Python. Hopefully, you have tried to do the examples on your own. If you have, most certainly you have discovered that what you write in the code has to be *very accurate*.

For example, in our program `ball_plot.py`, we called `linspace` in this way

```
t = np.linspace(0, 1, 1001)
```

If this had rather been written

```
t = np.linspace[0, 1, 1001)
```

we would have got an error message (`[` was used instead of `(`), even if you and I would understand the meaning perfectly well!

Remember that it is not a human that runs your code, it is a machine. Therefore, even if the meaning of your code looks fine to a human eye, it still has to comply in detail to the rules of the programming language. If not, you get warnings and error messages. This also goes for lower and upper case letters. If you (after importing from `math`) give the command `pi`, you get 3.1415 . . .. However, if you write `Pi`, you get an error message. Pay attention to such details also when they are given in later chapters.

1.8.2 Write Readable Code

When you write a computer program, you have two very different kinds of readers. One is Python, which will interpret and run your program according to the rules. The other is some human, for example, yourself or a peer. It is very important to organize and comment the code so that you can go back to your own code after, e.g., a year and still understand what clever constructions you put in there. This is relevant when you need to change or extend your code (which usually happens often in reality). Organized coding and good commenting is even more critical if other people are supposed to understand code that you have written.

It might be instructive to see an example of code that is *not* very readable. If we use our very first problem, i.e. computing the height y of a ball thrown up in the air, the mathematical formulation reads:

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

Now, instead of our previous program `ball.py`, we could write a working program (in *bad* style!) like:

```
# This is an example of bad style!
m=5;u=9.81;y=0.6
t=m*y-u*0.5*y**2;print(t)
```

Running this code, would give the correct answer printed out. However, upon comparison with the mathematical writing, it is not even clear that the two are related, unless you sit down and look carefully at it!

In this code,

- variable names do not correspond to the mathematical variables
- there are no (explaining) comments
- no blank lines
- no space to each side of = and -
- several statements appear on the same line with no space in between

When comparing this "bad style" code to the original code in `ball.py`, the point should be clear.

1.8.3 Fast Code or Slower and Readable Code?

In numerical computing, there is a strong tradition for paying much attention to *fast code*. Industrial applications of numerical computing often involve simulations that run for hours, days, and even weeks. Fast code is tremendously important in those cases.

The problem with a strong focus on fast code, unfortunately, is that sometimes clear and easily understandable constructions are replaced by fast (and possibly clever), but less readable code. For beginners, however, it is definitely most important to learn writing readable and correct code.

We will make some comments on constructions that are fast or slow, but the main focus of this book is to teach how to write correct programs, not the fastest possible programs.

1.8.4 Deleting Data No Longer in Use

Python has *automatic garbage collection*, meaning that there is no need to delete variables (or objects) that are no longer in use. Python takes care of this by itself. This is opposed to, e.g., Matlab, where explicit deleting sometimes may be required.

1.8.5 Code Lines That Are Too Long

If a code line in a program gets too long, it may be continued on the next line by inserting a back-slash at the end of the line before proceeding on the next line. However, *no blanks* must occur after the back-slash! A little demonstration could be the following,

```
my_sum = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 +\
         14 + 15 + 16 + 17 + 18 + 19 + 20
```

So, the back-slash works as a *line continuation character* here.

1.8.6 Where to Find More Information?

We have already recommended Langtangen's book, *A Primer on Scientific Programming with Python* (Springer, 2016), as the main reference for the present book.

In addition, there is, of course, the official Python documentation website (http://docs.python.org/), which provides a Python tutorial, the *Python Library Reference*, a *Language Reference*, and more. Several other great books are also available, check out, e.g., http://wiki.python.org/moin/PythonBooks.

As you do know, search engines like Google are excellent for finding information quickly, so also with Python related questions! Finally, you will also find that the questions and answers at http://stackoverflow.com often cover exactly what you seek. If not, you may ask your own questions there.

1.9 Exercises

Exercise 1.1: Error Messages
Save a copy of the program `ball.py` and confirm that the copy runs as the original. You are now supposed to introduce errors in the code, one by one. For each error introduced, save and run the program, and comment how well Python's response corresponds to the actual error. When you are finished with one error, re-set the program to correct behavior (and check that it works!) before moving on to the next error.

a) Insert the word *hello* on the empty line above the assignment to `v0`.
b) Remove the # sign in front of the comment *initial velocity*.
c) Remove the = sign in the assignment to `v0`.
d) Change the reserved word `print` into `pint`.
e) Change the calculation of `y` to `y = v0*t`.
f) Change the line `print(y)` to `print(x)`.

Filename: `testing_ball.py`.

Exercise 1.2: Volume of a Cube
Write a program that computes the volume V of a cube with sides of length $L = 4$ cm and prints the result to the screen. Both V and L should be defined as separate variables in the program. Run the program and confirm that the correct result is printed.

Hint See `ball.py` in the text.
Filename: `cube_volume.py`.

Exercise 1.3: Area and Circumference of a Circle
Write a program that computes both the circumference C and the area A of a circle with radius $r = 2$ cm. Let the results be printed to the screen on a single line with an appropriate text. The variables C, A and r should all be defined as separate variables in the program. Run the program and confirm that the correct results are printed.
Filename: `circumference_and_area.py`.

Exercise 1.4: Volumes of Three Cubes
We are interested in the volume V of a cube with length L: $V = L^3$, computed for three different values of L.

a) In a program, use the `linspace` function to compute and print three values of L, equally spaced on the interval $[1, 3]$.
b) Carry out, by hand, the computation $V = L^3$ when L is an array with three elements. That is, compute V for each value of L.
c) Modify the program in a), so that it prints out the result `V` of `V = L**3` when `L` is an array with three elements as computed by `linspace`. Compare the resulting volumes with your hand calculations.
d) Make a plot of `V` versus `L`.

Filename: `volume3cubes.py`.

Exercise 1.5: Average of Integers
Write a program that stores the sum $1 + 2 + 3 + 4 + 5$ in one variable and then creates another variable with the average of these five numbers. Print the average to the screen and check that the result is correct.
Filename: `average_int.py`.

Exercise 1.6: Formatted Print to Screen
Write a program that defines two variables as `x = pi` and `y = 2`. Then let the program compute the product `z` of these two variables and print the result to the screen as

```
Multiplying 3.14159 and 2 gives 6.283
```

Filename: `formatted_print.py`.