Ch.1: Programming with formulas

Joakim Sundnes[1,2]

[1]Simula Research Laboratory
[2]University of Oslo, Dept. of Informatics

Aug 18, 2019

1 Introduction

These lecture notes are based on the book "A Primer on Scientific Programming with Python" by Hans Petter langtangen. The notes are meant to be used as a supplement to the book, and are intentionally written in a brief and compact style. If you want a more detailed explanation of some concept, or are curious about some details or how various Python programming concepts are used in practice, the book is an excellent resource. The book and these notes are written specifically for the course IN1900, and they follow the same teaching philosophy and general structure. The overarching idea is that the only way to learn to program is to program. Reading theory is useful, but without actual programming practice the value is very limited. Following this idea, the typical workflow in IN1900 is the following:

1. Briefly introduce a new programming concept
2. Illustrate its use through a few relevant examples
3. Train the new concept through a number of programming exercises

The target audience for these notes is people with no background in programming. The notes cover the Python language, but also introduces and explains fundamental constructs such as loops, functions and lists, which are generic and relevant for all programming langugages. For experienced programmers that are new to Python the notes can be used as a quick reference for learning syntax and basic Python-specific functionality, but other books may be more suitable for this group.

1.1 Notes about the Python language

This course teaches the Python language, which has a number of advantages as a first programming language, but also some minor drawbacks. The main advantage of Python is that it is a so-called "high-level" language, with simple and intuitive syntax that makes it easy to get started. However, although it is suitable as a beginner's language, Python is also suitable for more advanced tasks, and is currently one of the most widely used programming languages worldwide.

One of the more important drawbacks of Python is tightly linked to its main advantages. Being a flexible high-level language, writing small programs can be very quick, but the code can easily get messy when writing something bigger. Other languages like C, C++ and Java enforces more structure in the code, which is annoying when you want to write a small program quickly, but may be efficient in the long run when writing larger programs. However, it is certainly possible to write neat and nicely structured programs in Python as well, but this requires a choice of the programmer to follow certain principles of coding style, and is not forced by the language itself. We will comment more on coding style later in these notes.

Another slightly annoying aspect of Python is that it exists in different versions. At the time of writing this (August 2019), Python 3 is dominant, with the newest stable version being Python 3.7. All the examples in these notes are in Python 3, but the latest version of "A Primer on Scientific Programming with Python" was published in 2016 and is still Python 2. For the topics covered here the difference between the two version are very small, and combining the notes with the book should not cause much trouble. The most significant difference, which we will run into very early, is a difference in the command `print` which we use to write text and numbers to the screen. We will comment more on this and some other differences later.

Finally, a nice feature of Python is that it can be used in many different ways. This is very useful for a number of purposes, but can be confusing in the beginning. Here's a quick overview of how we use Python in IN1900:

- Writing regular programs using a text editor or an IDE (Integrated development environment) is the "classical" way of programming, and will be the most widely used in the course. One writes small programs using a text editor, stores them in a file, and runs them from a terminal window with a command such as `python my_program.py`. All exercises in IN1900 are to be handed in as ".py"-files, so this way of programming will probably be your main style of working through the course. If one prefers an IDE, a convenient choice is Spyder, which is installed automatically if installing Python from Anaconda. See the IN1900 course website for more information.

- Python can be used interactively in the terminal, by typing `python` or `ipython` in a terminal window, without a subsequent file name. This will open an environment for typing and running Python commands, which is not very suitable for writing programs over several lines, but extremely useful for testing Python commands and statements, or simply using Python as a calculator. The two versions `python` and `ipython` work largely the same way, but `ipython` has a number of additional features and is recommended. An `ipython` window can also be started for instance from the *Anaconda navigator*, which is convenient on Windows.

- *Jupyter notebooks* are a form of interactive notebooks that combine code and text. These are read viewed a browser and look quite similar to a simple web page, but an important difference is that the code segments are "live" Python code that can be run, changed and re-run while reading the document. These notes are available as Jupyter notebooks, and most of the slides used in the lectures will also be in this format.

For more detailed instructions on how to install and run Python on various platforms we refer to the course web site.

1.2 The first example; Hello world!

Most introductory books on programming start with a so-called "Hello world!"-program, which is a program that simply writes the words "Hello world!" to the screen. In Python, this program is just a single line;

```
print("Hello world!")
```

To write and run such a program, open your favourite editor (Atom, gedit, Emacs etc.) type the given line and save the file with a suitable filename, for instance `hello.py`. Then, open a terminal or an iPython window, navigate to the directory where you saved the file (`cd` and `ls` are the essential commands, see the course web site for a quick guide). The type `python hello.py` in the terminal, or `run hello.py` if you are using iPython. The output appears in the terminal right after the command. If you are using an IDE, this is essentially an editor and an iPython/terminal window combined. For instance, in the Spyder IDE the editor is in the upper left window, and you type the program and save the file there. The, the window in the lower right corner is the iPython window where you run the program.

While the "Helllo world!"-program may seem like a silly example, it actually serves a number of useful purposes. First of all, running this small program will verify that you have installed Python properly, and that you have installed the right version. It also introduces the function `print`, which will be used virtually every time we program, and illustrates how we use quotes to define a `string` in Python. While `print` is a word that Python understands, *hello* and *world* are not. By using the quotes we tell Python that it should not try to understand (or interprete) these words, but rather treat it as a simple text that in this case is to be printed to the screen. More about this later. Furthermore, this simple example illustrates the most frequently encountered difference between Python versions 2 and 3. In Python 2, the program would read `print "Hello world"`, i.e. without the parantheses. The examples in the book by Langtangen, which are written in Python 2, will stop with an error message if you try to run them using Python 3. In most cases the only error is the missing parantheses, which is usually indicated by the error message, and adding parantheses to all the print statements will make most of the examples run fine in Python 3.

The tiny "Hello world!" example is also a good candidate for testing the interactive Python shell mentioned above. In a regular terminal window on Mac or Linux, the example would look something like this:

```
Terminal> python
Python 3.7.3 (default, Mar 27 2019, 16:54:48)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello world!")
Hello world!
>>>
```

As mentioned above, the interactive shell is very useful for testing and learning how various Python functions are used, and for using Python as a calculator.

2 Programming mathematical formulas

In this chapter we will take one step beyond the "Hello world!"-example, and introduce programming with mathematical formulae. Such formulae are essential parts of most programs written for scientific applications, as well as a number of other applications, and they are also useful for introducing several fundamental concepts in programming. After working through the examples in this chapter you will:

- Be able to use Python's `print` function to output text and numbers
- Know what *variables* are, and how they are used in programs
- Know how to program mathematical expressions in Python
- Know about *types*, and how you can check the type of a variable
- Be able to use *comments* to make your programs more readable

To introduce these concepts, consider first a formula from high school physics, describing the height of a ball in vertical motion

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

where

- y is the height (position) as function of time t
- v_0 is the initial velocity at $t = 0$
- g is the acceleration of gravity

The task is now to write a program that computes y for given values of v_0, g and t. We could of course easily do this with a calculator, but a small program is much more flexible and powerful. To evaluate the formula above, we first need to assign some values to v_0, g and t, and then make the calculation. Choosing for instance $t = 0.6, g = 9.81$, and $v_0 = 5$, a complete Python program for evaluating the formula above reads

```
print(5*0.6 - 0.5*9.81*0.6**2)
```

As for the example above, this line can be typed into an interactive Python session, or written in an editor and stored in a file, for instance `ball1.py`. Then, the program is run with `python ball1.py` in a regular terminal, or `run ball1.py` in an iPython window or Spyder.

The `ball1.py`-program is not much more complex, or more useful, than the "Hello world!"-example above, but notice that in this case we did not use quotation marks inside the parantheses. The reason is that we actually want Python to evaluate the formula, and print the result to the screen, which works fine as long as the text inside the paranthesis is valid Python code. If we put quotation marks around the formula above, the code would still work, but the result is not what we want (try it!). At this point it is also worth noting that while Python may be a "flexible" and "high-level" language, as described above, all programming languages are extremely pick about spelling and grammar. Consider for instance the line

```
write(5*0,6 - 0,5*9,81*0,6^2)
```

While most people may read this line quite easily, and interpret it as the physics formula above, it makes no sense as a Python program. There are multiple errors: `write` is not a legal Python word in this context, comma has another meaning than in math, and the hat does not mean exponentiation. We have to be extremely accurate with how we write computer programs, and it takes time and experience to learn this.

The evaluation of the mathematical formula above is performed according to the standard rules. The terms are evaluated one by one, starting from left, exponentiation is performed first and then multiplication and division. If we want to control the order of evaluation, we can use parantheses, just as we would do in mathematics. For instance these two lines here will provide quite different results:

```
print(2 + 2*3)
print((2 + 2)*3)
```

```
8
12
```

The use of parantheses to group calculations works exactly as in mathematics, and is not very difficult to understand for people with a mathematical background. However, when programming more complicated formulas it is very easy to make mistakes such as forgetting or misplacing a closing paranthesis. This mistake is probably the most common source of error when programming mathematical formulas, and even for experienced programmers it is worth paying close attention to the order and number of parantheses in the expressions. Getting it wrong will either lead to an error message when the code is run or to a program that runs fine but gives an unexpected results. The first type of error is usually quite easy to find and to fix, but the latter may be much harder.

Although Python is quite strict on the spelling and grammar, or *syntax*, of the formulas, there is some flexibility. For instance, whitespace, or blanks, inside a formula does not matter at all. An expression like `5 *2` works just as well as `5*2`. In general, whitespace in a Python program only matters if it is at the start of a line, which we will come back to later. Otherwise, one should use whitespace in order to make the code as readable as possible to humans, since Python will ignore it anyway.

2.1 Store numbers in variables to make a program more readable

In mathematics we are used to variables, such as v_0, g and t in the formula above. We can use variables in a program to, which makes the program easier to read and understand:

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

```
1.2342
```

This program spans several lines of text and uses variables, but otherwise performs the same calculations and gives the same output as the one-line program above. Even for this very simple example the use of variables has a few advantages, mainly that the program becomes easier to read, the physics formula is much more recognizable, and changing the value of a given variable is easier. The latter point is true in particular for the variable `t`, which occurs twice in the formula. To evaluate the formula at a new time point, we now only need to assign a new value to the variable in one place, while the one-line version of the program has to be changed in two places. The latter is very error prone, since it is easy to forget one of the places that needs to be changed, and should always be avoided. If the same number occurs more than once in a program, you should always use a variable to store it.

The instructions in the program are called *statements*, and are executed one by one when we run the program. It is common to have one statement per line, although it is possible to put muleiple statements on one line, separated by semicolon, as in `v0 = 5; g = 9.81; t = 0.6`. For people new to programming, especially those used to reading mathematics, it is worth taking note of the strict sequence in which the lines are executed. In the mathematical equation above we first introduced a formula with some variables (v_0, g, t), and then defined these variables further down. This approach is completely standard in mathematics, but it makes no sense in programming. Programs are executed line by line from the top, so all definitions of variables must be done *above* the line where they are used.

Choosing variable names is up to the programmer, and in general there is great flexibility in choosing such names. In mathematics it is common to use a single letter for a variable, but a variable in Python program can be any word containing the letters a-z, A-Z, underscore _ and the digits 0-9, but it cannot start with a digit. Variable names Python are also case-sensitive, for instance `a` is different from `A`. The following program is identical to the one above, but with different variable names:

```
initial_velocity = 5
accel_of_gravity = 9.81
TIME = 0.6
VerticalPositionOfBall = initial_velocity*TIME - \
                         0.5*accel_of_gravity*TIME**2
print(VerticalPositionOfBall)
```

```
1.2342
```

Note that the backslash allows an expression to be continued on the next line. These alternative names are arguably more descriptive, but also makes the formula very long and cumbersome to read. Choosing good variable names is often a balance of being descriptive and being short and compact, and can be quite important for making a program easy to read and understand. Writing readable and understandable code also makes it easier to find errors in the code, so choosing good variable names is worthwhile even if you are the only person who will ever read your code. Although the choice of variable names is up to the programmer, some names are reserved in Python and are not allowed to be used. These are `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `exec`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `not`, `or`, `pass`, `print`, `raise`, `return`, `try`, `with`, `while`, and `yield`. This list is difficult to memorize at this point, but we will use many of these words in our programs later so it will be quite natural that they cannot be used as variable names. However, it may be worth taking note of `lambda`, since the Greek letter λ is commonly used in mathematical formulas. Since Python does not understand Greek letters it is common to just spell them out when programming a formula, i.e. α becomes `alpha` etc., but for λ we will need to introduce a small intentional typo and write `lmbda` or similar.

The program above contains two different types of statements; several assignment statements, which assign values to variables, followed by a single print statement at the end. It may be quite intuitive how these statements work, but the assignment statements are worth looking into in more detail. In these statements, the right hand side of the equality sign is evaluated first, and then the result is assigned to the variable on the left. An effect of this execution order is that statements like the following work just fine, and are common to see in programs:

```
t = 0.6
t = t + 0.1
print(t)
```

```
0.7
```

The second line of this code does not make sense as a mathematical equation, but is a perfectly valid assignment in a computer program. The right hand side is evaluated first, using the value of `t` already defined, and then `t` is updated to the result of the calculation. The equality sign in Python is called the *assignment operator*, and it works almost like an equality sign in mathematics, but not quite. If, for instance, we want to compare two numbers to figure out if they are equal, the operator to use in Python is `==`. For instance, a trivial comparison may look like

```
a = 5
print(a == 5)
```

```
True
```

We will see many more such examples later.

2.2 Comments are useful to explain how you think in programs

We may combine the strengths of the two programs above, and have both compact variable names *and* a more detailed description of what they are. This can be done using *comments*, as illustrated in the following example:

```
# 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)
```

```
1.2342
```

In this code all the text following the `#` symbol is treated as a comment, and effectively ignored by Python. Comments are used to explain what the computer instructions mean, what variables mean, how the programmer reasoned when the program was written, etc. Comments can be very useful for increasing readability, although they should not be over-used. Comments that say no more than the code, for instance `a = 5  # set a to 5`, are not very useful.

2.3 Variables have different *types*

So far all the variables we have used have been numbers, which is also what we are used to from mathematics. However, in a computer program we can have many different kinds of variables, not just numbers. All variables have a *type*, which is usually decided automatically based on the value we assign to it. For instance, the statement `v0 = 5` will create a variable of the type *integer*, or `int`, while for instance `t = 0.6` will create a variable of type `float`, representing a floating point number. We can also have text variables, called strings in Python, having type `str`, for string. For instance, the "Hello world!"-example above could have been written as

```
hello = "Hello world!"
print(hello)

Hello world!
```

We will encounter many more variable types in subsequent chapters. The type of a variable decides how it can be used, and also the effect of various operations. These rules are usually quite intuitive. For instance, most mathematical operations only work with variable types that actually represent numbers, or they have a different effect on other variable types, when this is natural. To get an idea of how this works in Python, think about some simple mathematical operations on strings. Which of the following operations do you think are allowed, and what are the results? (i) Adding two strings together, (ii) multiplying a string with an integer, (iii) multiplying two strings, and (iv) multiplying a string with a decimal number. Then try it in Python:

```
print(hello + hello)
print(hello*5)

hellohello
hellohellohellohellohello
```

Strings that contain numbers are a potential source of confusion. Consider for instance the code

```
x1 = 2
x2 = "2"
print(x1+x1)
print(x2+x2)

4
22
```

We see that the `x2` variable is still treated as a text string in Python, because it was defined using the quotation marks, even though it contains a single number. Sometimes it may be useful to check the type of a variable, and this is easy to do with the function `type`:

```
print(type(hello))
print(type(t))
print(type(v0))
print(type(2))

<class 'str'>
<class 'float'>
<class 'int'>
<class 'int'>
```

We see that the output is as we would expect from how the variables were defined above. The word `class` preceding the types indicates that these types are defined as *classes* in Python, a concept we will get back to later. Checking the type of a variable is often very useful if you get unexpected behaviors from your program. Usually we can let Python select the type automatically and not pay mutch attention to it, but in some cases it can be useful to convert between variable types. For the examples in this book, by far the most common type conversion is to convert text containing a number to an actual decimal number, which is done with the funtion `float`. The same function can be used to convert an integer to a decimal number.

```
x1 = float(x1)
x2 = float(x2)
print(type(x1))
print(type(x2))
print(x2+x2)
```

```
<class 'float'>
<class 'float'>
4.0
```

Of course, using `float` to convert a string to a number requires that the string is actually a number. Trying to convert a regular work, as in `float(hello)` will make the program stop with an error message.

2.4 Formatting text output

The calculations in the programs above would output a single number, and simply print this number to the screen. In many cases this solution is fine, but sometimes we want several numbers or other types of output from a program. This is easy to do with the `print` function, by simply putting several variables inside the parantheses, separated by comma. For instance, if we want to output both `t` and `y` from the calculation above, the following line would fork:

```
print(t,y)
```

```
0.6 1.2342
```

However, although this line works, it is not necessarily the most readable or useful output. Sometimes it is useful to format the output better, and to use a combination of text and numbers, for instance

```
At t=0.6 s, y is 1.23 m.
```

There are multiple ways to accomplish this in Python, but the most recent and arguably most convenient is to use so called *f-strings*, which were introduced in Python 3.6. If you are using an earlier version of Python the following examples will therefore not work, but there are alternative ways of formatting text that will be described later in these notes.

To achieve the output string above, using the *f-string* formatting, we would replace the final line with

```
print(f"At t={t} s, y is {y} m.")
```

```
At t=0.6 s, y is 1.2342 m.
```

There are three things worth noticing here. First, we enclose the output in quotation marks, just as in the "Hello world!"-example above, which tells Python that this is a string. Second the string is prefixed with the letter `f`, which indicates that the string may contain something extra. More specifically, the string may contain expressions or variables enclosed in curly braces. Third, we have included two such variables, `t` and `y`, enclosed in curly braces. When Python encounters these braces inside an f-string, it will evaluate the contents of the braces, which may be an expression or a variable, and insert the resulting value into the string. In this case it will simply insert the current values of the variables `t` and `y`. If we want, we could also include a mathematical expression inside the braces, such as

```
print(f"2+2 = {2+2}")
```

```
2+2 = 4
```

The only requirement for the contents inside the braces is that it must be a valid Python expression, which can be evaluated to give some kind of value. In these notes we will typically use it for numbers, but it may also be used for variables with other types of values.

The f-string formatting will ususally give nicely formatted output by default, but sometimes we want more detailed control of the formatting. For instance, we may want to control the number of decimals when outputting numbers. This is conveniently obtained by including a *format specifier* inside the curly braces. Consider for instance the following code:

```
t = 1.234567
print(f"Default output gives t = {t}.")
print(f"We can set the precision: t = {t:.2}.")
print(f"Or control the number of decimals: t = {t:.2f}.")
```

```
Default output gives t = 1.234567.
We can set the precision: t = 1.2.
Or control the number of decimals: t = 1.23.
```

There are many different format specifiers, for controlling the output format of both numbers and other types of variables. We will only use a small subset in this course, and primarily to output numbers. In addition to thoseshown above, the following may be useful;

```
print(f"Or control the space used for the output: t = {t:8.2f}.")
```

```
Or control the space used for the output: t =     1.23
```

This is used to control the number of decimals in a number, and also how much space (the number of characters) used to output the number on the screen. Here we have specified the number to be output with two decimals and a total length of eight, including the decimals. This form of control is very useful for outputting multiple lines in a table-like format, to ensure that the columns in the table are properly aligned. A similar feature can be used for integers

```
r = 87
print(f"Integer output specified to take up 8 chars of space: r = {r:8d}")
```

```
Integer output specified to take up 8 chars of space: r =       87
```

Finally, the generic format specifier `g` will output a floating point number in the most compact form, consider for instance:

```
a = 786345687.12
b = 1.2345
print(f"Without the format specifier: a = {a}, b = {b}.")
print(f"With the format specifier: a = {a:g}, b = {b:g}.")
```

```
Without the format specifier: a = 786345687.12, b = 1.2345.
With the format specifier: a = 7.86346e+08, b = 1.2345.
```

2.5 Importing modules for more advanced functionality

We have seen that standard arithmetic operations are directly available in Python, without any extra effort. However, what if need more advanced operations such as $\sin x$, $\cos x$, $\ln x$, etc.? These functions are not available directly, but can be found in a so-called *module*, which must be imported before we can use them in our program. In general, lots of functionality in Python is found in such modules, and we will import a few modules in nearly all programs we write. The standard mathematical functions are found in the `math` module. For instance, the following code computes the square root of a number, using the `sqrt` function in the `math` module:

```
import math
r = math.sqrt(2)
# or
from math import sqrt
r = sqrt(2)
# or
from math import *    # import everything in math
r = sqrt(2)
```

These examples illustrate three different ways of important modules. In the first one, we import everything from the `math` module, but everything we want to use must be prefixed with `math`. The second option involves only the `sqrt` function, and it imports it into the main *namespace* of the program, which means it can be used with no prefix. Finally, the third option imports everything from `math` into the main namespace, so all functions from the module are available in our program without the prefix.

A natural question to ask is why we need three different ways to import a module. Why not use the simple `from math import *` and get access to all the mathematics functions we need? The reason is that we will often import from several modules, and some of these modules contain functions with identical names. In such cases it is useful to have some control over which functions we actually use, either by selecting only what we need from each module, as in `from math import srt`, or by importing as in `import math` so that all functions must be prefixed with the module name. To avoid confusion later it may be good to get into the habit of importing modules like this right away, although in small programs where we only import a single module there is nothing wrong with `from math import *`.

Another natural question at this point is how we know where to find the functions we want. For instance, say we need to compute with complex numbers. How can we know if there is a module in Python for this? And if there is, what is it called? In general, learning about the useful modules and what they contain are part of learning to program, but it may be even more important to know where to find such information. An excellent source is the Python Library Reference (https://docs.python.org/3/library/), which contains information about all the standard module that are distributed with Python. More generally, a google search such as `complex numbers python` quickly leads us to the `cmath` module, which contains mostly the same functions as `math`, but with support for complex numbers. To check the contents of a module, one may also use the command `pydoc` in the terminal window, for instance `pydoc math`, or we can import the module in a Python program and list its contents with the builtin function `dir`.

```
import math
print(dir(math))
```

```
['__doc__', '__file__', '__loader__', '__name__', (...) ]
```

As another example of computing with functions from `math` consider the task of evaluating

$$Q = \sin x \cos x + 4 \ln x, \text{ for } x = 1.2.$$

The Python code looks as follows:

```
from math import sin, cos, log
x = 1.2
Q = sin(x)*cos(x) + 4*log(x)   # log is ln (base e)
```

Note that the ln function is called `log` in the `math` module, while the logarithm with base 10 is called `log10`'

2.6 Pitfalls when programming mathematics

Usually, the mathematical operations as described above work as we would expect. When the results are not what we expect, the cause is usually a trivial error we have introduced while typing, typically assigning the wrong value to a variable or (most common) messing up the number of parantheses. However, some potential error sources are less obvious, and are worth knowing about even if they are relatively rare.

Round-off errors. Computers have inexact arithmetics because of rounding errors. This is usually not a problem in computations, but in some cases it may cause unexpected results. Let us for instance compute $1/49 \cdot 49$ and $1/51 \cdot 51$:

```
v1 = 1/49.0*49
v2 = 1/51.0*51
print(f"{v1:.16f} {v2:.16f}")
```

Output with 16 decimals becomes

```
0.9999999999999999 1.0000000000000000
```

Most real numbers are represented inexactly on a computer, typically with 17 digits accuracy. Neither 1/49 nor 1/51 are represented exactly, and the error is approximately 10^{-16}. Errors of this order usually don't matter, but there are two particular cases where they may be significant. One is that in some cases they may accumulate through numerous computations and end up as a significant error in the final result. The other, which you are more likely to encounter through this course, is that comparison of two decimal numbers may be unpredictable. The two numbers `v1` and `v2` above are both supposed to be equal to 1, but look at the result of this code:

```
print(v1 == 1)
print(v2 == 1)
```

```
False
True
```

We see that the evaluation works as expected in one case but not the other, and this is a general problem when comparing floating point numbers. In most cases it works, but in some cases it doesn't. It is difficult or impossible to predict when it will not work, and the behavior of the program therefore becomes unpredictable. The solution is to always compare floats using a tolerance, as in

```
tol = 1e-14
print(abs(v1-1) < tol)
print(abs(v2-1) < tol)
```

```
True
True
```

There is no strict rule for how to set the value of the tolerance `tol`, but it should be small number, but larger than the typical machine precision 10^{-16}.

Integer division. In Python 2, and many other programming languages, unintended *integer division* may sometimes cause surprising results. In Python 3 this is no longer a problem, so you are not likely to run into it during this course, but it is worth being aware of the problem since many other programming languages behave in this way. Recall from above that various operations behave differently depending on the type of the variable they work on, for instance adding two strings vs adding numbers. In Python 2, the division operator, /, behaves like normal division if one of the two arguments are floats, but if both are integers then it will perform integer division, and discard the decimal part of the result. Consider the following interactive session, which runs Python 2.7:

```
Terminal> python2.7
Python 2.7.14 (default, Sep 22 2017, 00:06:07)
(...)
>>> print(1.0/2)   #the parantheses are optional in Python 2.7
0.5
>>> print(1/2)
0
```

Integer division is useful for many tasks in computer science, and is therefore the default behavior of many programming languages, but it is usually not what want when programming mathematical formulas. Therefore, it may be a good habit to ensure that variables used in calculations are actually floats, by simply defining them as `v0 = 5.0` rather than `v0 = 5`. Although it does not really make a difference in Python 3, it is good to get into this habit simply to avoid problems when programming in other languages later.