A Test of Doconce a Book with Springer's T2 Style

Hans Petter Langtangen[1,2]

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

Feb 19, 2014

Preface

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

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

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

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

There is a web page associated with this book, `http://hplgit.github.com/scipro-primer`, which lists the software you need and explains briefly how to install it. This page also contains all the files associated with the program examples in this book.

Python version 2 or 3? A common problem among Python programmers is to choose between version 2 or 3, which at the time of this writing means choosing between version 2.7 and 3.3. The general recommendation is to go for version 3, but programs are then not compatible with version 2 and vice versa. There is still a problem that much useful mathematical software in Python has not yet been ported to version 3. Therefore, scientific computing with Python still goes mostly with version 2. A widely used strategy for software developers who want to write Python code that works with both versions, is to develop for v2.7, which is very close to what is accepted in version 3, and then use the ranslation tool *2to3* to automatically translate the code to version 3.

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

Acknowledgments. Several people have helped to make substantial improvements of the text. Here I list only the names with Norwgian characters to test the handling of those: Ståle Zerener Haugnæss, Tobias Vidarssønn Langhoff, and Håkon Møller.

Oslo, April 2012 *Hans Petter Langtangen*

Contents

1 Basic array computing and plotting

This chapter gives an introduction to arrays: how they are created and what they can be used for. Array computing usually ends up with a lot of numbers. It may be very hard to understand what these numbers mean by just looking at them. Since the human is a visual animal, a good way to understand numbers is to visualize them. In this chapter we concentrate on visualizing curves that reflect functions of one variable; i.e., curves of the form $y = f(x)$. A synonym for curve is graph, and the image of curves on the screen is often called a plot. We will use arrays to store the information about points along the curve. In a nutshell, array computing demands visualization and visualization demands arrays.

All program examples in this chapter can be found as files in the folder `src/plot`[1].

1.1 Arrays in Python programs

This section introduces array programming in Python, but first we create some lists and show how arrays differ from lists.

1.1.1 Using lists for collecting function data

Suppose we have a function $f(x)$ and want to evaluate this function at a number of x points $x_0, x_1, \ldots, x_{n-1}$. We could collect the n pairs $(x_i, f(x_i))$ in a list, or we could collect all the x_i values, for $i = 0, \ldots, n-1$, in a list and all the associated $f(x_i)$ values in another list. The following interactive session demonstrates how to create these three types of lists:

[1] `http://some.where.net/doconce/test/software/plot`

```
>>> def f(x):
...     return x**3       # sample function
...
>>> n = 5                 # no of points along the x axis
>>> dx = 1.0/(n-1)        # spacing between x points in [0,1]
>>> xlist = [i*dx for i in range(n)]
>>> ylist = [f(x) for x in xlist]
>>> pairs = [[x, y] for x, y in zip(xlist, ylist)]
```

Here we have used list comprehensions for achieving compact code. Make sure that you understand what is going on in these list comprehensions (if not, try to write the same code using standard `for` loops and appending new list elements in each pass of the loops).

The list elements consist of objects of the same type: any element in `pairs` is a list of two `float` objects, while any element in `xlist` or `ylist` is a `float`. Lists are more flexible than that, because an element can be an object of any type, e.g.,

```
mylist = [2, 6.0, 'tmp.ps', [0,1]]
```

Here `mylist` holds an `int`, a `float`, a string, and a list. This combination of diverse object types makes up what is known as *heterogeneous* lists. We can also easily remove elements from a list or add new elements anywhere in the list. This flexibility of lists is in general convenient to have as a programmer, but in cases where the elements are of the same type and the number of elements is fixed, arrays can be used instead. The benefits of arrays are faster computations, less memory demands, and extensive support for mathematical operations on the data. Because of greater efficiency and mathematical convenience, arrays will be used to a large extent in this book. The great use of arrays is also prominent in other programming environments such as MATLAB, Octave, and R, for instance. Lists will be our choice instead of arrays when we need the flexibility of adding or removing elements or when the elements may be of different object types.

People only become computer programmers if they're obsessive about details, crave power over machines, and can bear to be told day after day exactly how stupid they are.
Gregory J. E. Rawlins [5], computer scientist.

1.1.2 Basics of numerical Python arrays

An *array* object can be viewed as a variant of a list, but with the following assumptions and features:

- All elements must be of the same type, preferably integer, real, or complex numbers, for efficient numerical computing and storage.
- The number of elements must be known when the array is created.
- Arrays are not part of standard Python - one needs an additional package called *Numerical Python*, often abbreviated as NumPy. The

Python name of the package, to be used in `import` statements, is `numpy`.
- With `numpy`, a wide range of mathematical operations can be done directly on complete arrays, thereby removing the need for loops over array elements. This is commonly called *vectorization*
- Arrays with one index are often called vectors. Arrays with two indices are used as an efficient data structure for tables, instead of lists of lists. Arrays can also have three or more indices.

Remarks.

1. There is actually an object type called `array` in standard Python, but this data type is not so efficient for mathematical computations, and we will not use it in this book.
2. The number of elements in an array *can* be changed, but at a substantial computational cost.

The following text lists some important functionality of NumPy arrays. A more comprehensive treatment is found in the excellent *NumPy Tutorial*, *NumPy User Guide*, *NumPy Reference*, *Guide to NumPy*, and *NumPy for MATLAB Users*, all accessible at scipy.org[2].

Remarks on importing NumPy.

The statement

```
import numpy as np
```

with subsequent prefixing of all NumPy functions and variables by `np.`, has evolved as a standard syntax in the Python scientific computing community. However, to make Python programs look closer to MATLAB and ease the transition to and from that language, one can do

```
from numpy import *
```

to get rid of the prefix (this is evolved as the standard in *interactive* Python shells). This author prefers mathematical functions from `numpy` to be written without the prefix to make the formulas as close as possible to the mathematics. So, $f(x) = \sinh(x-1)\sin(wt)$ would be coded as

```
from numpy import sinh, sin

def f(x):
    return sinh(x-1)*sin(w*t)
```

[2] `http://scipy.org`

or one may take the less recommended lazy approach `from numpy import *` and fill up the program with *a lot* of functions and variables from `numpy`.

To convert a list `r` to an array, we use the `array` function from `numpy`:

```
a = np.array(r)
```

To create a new array of length `n`, filled with zeros, we write

```
a = np.zeros(n)
```

The array elements are of a type that corresponds to Python's `float` type. A second argument to `np.zeros` can be used to specify other element types, e.g., `int`. A similar function,

```
a = np.zeros_like(c)
```

generates an array of zeros where the length is that of the array `c` and the element type is the same as those in `c`. Arrays with more than one index are treated in Section **??**.

Often one wants an array to have n elements with uniformly distributed values in an interval $[p, q]$. The `numpy` function `linspace` creates such arrays:

```
a = np.linspace(p, q, n)
```

2 Storing results in data files

2.1 Writing data to file

Writing data to file is easy. There is basically one function to pay attention to: `outfile.write(s)`, which writes a string `s` to a file handled by the file object `outfile`. Unlike `print`, `outfile.write(s)` does not append a newline character to the written string. It will therefore often be necessary to add a newline character,

```
outfile.write(s + '\n')
```

if the string `s` is meant to appear on a single line in the file and `s` does not already contain a trailing newline character. File writing is then a matter of constructing strings containing the text we want to have in the file and for each such string call `outfile.write`.

Writing to a file demands the file object `f` to be opened for writing:

```
# write to new file, or overwrite file:
outfile = open(filename, 'w')

# append to the end of an existing file:
outfile = open(filename, 'a')
```

2.1.1 Example: Writing a table to file

Problem. As a worked example of file writing, we shall write out a nested list with tabular data to file. A sample list may take look as

```
[[ 0.75,        0.29619813, -0.29619813, -0.75      ],
 [ 0.29619813,  0.11697778, -0.11697778, -0.29619813],
 [-0.29619813, -0.11697778,  0.11697778,  0.29619813],
 [-0.75,       -0.29619813,  0.29619813,  0.75      ]]
```

Solution. We iterate through the rows (first index) in the list, and for each row, we iterate through the column values (second index) and write each value to the file. At the end of each row, we must insert a newline character in the file to get a linebreak. The code resides in the file `write1.py`[1].

The resulting data file becomes

```
 0.75000000    0.29619813   -0.29619813   -0.75000000
 0.29619813    0.11697778   -0.11697778   -0.29619813
-0.29619813   -0.11697778    0.11697778    0.29619813
-0.75000000   -0.29619813    0.29619813    0.75000000
```

An extension of this program consists in adding column and row headings:

```
           column  1     column  2     column  3     column  4
row  1    0.75000000    0.29619813   -0.29619813   -0.75000000
row  2    0.29619813    0.11697778   -0.11697778   -0.29619813
row  3   -0.29619813   -0.11697778    0.11697778    0.29619813
row  4   -0.75000000   -0.29619813    0.29619813    0.75000000
```

To obtain this end result, we need to the add some statements to the program `write1.py`. For the column headings we need to know the number of columns, i.e., the length of the rows, and loop from 1 to this length:

```
ncolumns = len(data[0])
outfile.write('          ')
for i in range(1, ncolumns+1):
    outfile.write('%10s    ' % ('column %2d' % i))
outfile.write('\n')
```

Note the use of a nested printf construction: The text we want to insert is itself a printf string. We could also have written the text as `'column  ' + str(i)`, but then the length of the resulting string would depend on the number of digits in `i`. It is recommended to always use printf constructions for a tabular output format, because this gives automatic padding of blanks so that the width of the output strings remain the same. As always, the tuning of the widths is done in a trial-and-error process.

To add the row headings, we need a counter over the row numbers:

```
row_counter = 1
for row in data:
    outfile.write('row %2d' % row_counter)
    for column in row:
        outfile.write('%14.8f' % column)
    outfile.write('\n')
    row_counter += 1
```

The complete code is found in the file `write2.py`[2]. We could, alternatively, iterate over the indices in the list:

[1] `http://some.where.net/doconce/test/software/input/write1.py`
[2] `http://some.where.net/doconce/test/software/input/write2.py`

```
for i in range(len(data)):
    outfile.write('row %2d' % (i+1))
    for j in range(len(data[i])):
        outfile.write('%14.8f' % data[i][j])
    outfile.write('\n')
```

2.1.2 Standard input and output as file objects

Reading user input from the keyboard applies the function `raw_input` as explained in Section ??. The keyboard is a medium that the computer in fact treats as a file, referred to as *standard input*.

The `print` command prints text in the terminal window. This medium is also viewed as a file from the computer's point of view and called *standard output*. All general-purpose programming languages allow reading from standard input and writing to standard output. This reading and writing can be done with two types of tools, either file-like objects or special tools like `raw_input` and `print` in Python. We will here describe the file-line objects: `sys.stdin` for standard input and `sys.stdout` for standard output. These objects behave as file objects, except that they do not need to be opened or closed. The statement

```
s = raw_input('Give s:')
```

is equivalent to

```
print 'Give s: ',
s = sys.stdin.readline()
```

Recall that the trailing comma in the `print` statement avoids the newline that `print` by default adds to the output string. Similarly,

```
s = eval(raw_input('Give s:'))
```

is equivalent to

```
print 'Give s: ',
s = eval(sys.stdin.readline())
```

For output to the terminal window, the statement

```
print s
```

is equivalent to

```
sys.stdout.write(s + '\n')
```

Why it is handy to have access to standard input and output as file objects can be illustrated by an example. Suppose you have a function that reads data from a file object `infile` and writes data to a file object `outfile`. A sample function may take the form

```
def x2f(infile, outfile, f):
    for line in infile:
        x = float(line)
        y = f(x)
        outfile.write('%g\n' % y)
```

This function works with all types of files, including web pages as `infile` (see Section ??). With `sys.stdin` as `infile` and/or `sys.stdout` as `outfile`, the `x2f` function also works with standard input and/or standard output. Without `sys.stdin` and `sys.stdout`, we would need different code, employing `raw_input` and `print`, to deal with standard input and output. Now we can write a single function that deals with all file media in a unified way.

There is also something called *standard error*. Usually this is the terminal window, just as standard output, but programs can distinguish between writing ordinary output to standard output and error messages to standard error, and these output media can be redirected to, e.g., files such that one can separate error messages from ordinary output. In Python, standard error is the file-like object `sys.stderr`. A typical application of `sys.stderr` is to report errors:

```
if x < 0:
    sys.stderr.write('Illegal value of x'); sys.exit(1)
```

This message to `sys.stderr` is an alternative to `print` or raising an exception.

Redirecting standard input, output, and error. Standard output from a program `prog` can be redirected to a file `output` instead of the screen, by using the greater than sign:

Terminal

```
Terminal> prog > output
```

Here, `prog` can be any program, including a Python program run as `python myprog.py`. Similarly, output to the medium called *standard error* can be redirected by

Terminal

```
Terminal> prog &> output
```

For example, error messages are normally written to standard error, which is exemplified in this little terminal session on a Unix machine:

Terminal

```
Terminal> ls bla-bla1 bla-bla2
ls: cannot access bla-bla1: No such file or directory
ls: cannot access bla-bla2: No such file or directory
Terminal> ls bla-bla1 bla-bla2 &> errors
Terminal> cat errors  # print the file errors
```

```
ls: cannot access bla-bla1: No such file or directory
ls: cannot access bla-bla2: No such file or directory
```

When the program reads from standard input (the keyboard), we can equally well redirect standard input to a file, say with name `raw_input`, such that the program reads from this file rather than from the keyboard:

Terminal

```
Terminal> prog < input
```

Combinations are also possible:

Terminal

```
Terminal> prog < input > output
```

Note. The redirection of standard output, input, and error does not work for Python programs executed with the `run` command inside IPython, only when executed directly in the operating system in a terminal window, or with the same command prefixed with an exclamation mark in IPython.

References. To check the bibliography, we need to make citations to a bookTCSE3, Matplotlib [1], and more books [2, 6] as well as Python itself [4], and of course NumPy [3].

A Styles for Springer T2

The T2 style for Doconce-generated LaTeX should make use of slightly modified `svmono.cls` and `t2.sty` files:

- `svmonodo.cls`
- `t2do.sty`

References

1. J. D. Hunter. Matplotlib: a 2d graphics environment. *Computing in Science & Engineering*, 9, 2007.
2. D. Mertz. *Text Processing in Python*. McGraw-Hill, 2003.
3. T. Oliphant et al. NumPy array processing package for Python. `http://www.numpy.org`.
4. Python programming language. `http://python.org`.
5. G. J. E. Rawlins. *Slaves of the Machine: The Quickening of Computer Technology*. MIT Press, 1998.
6. B. Rempt. *GUI Programming With Python: Using the Qt Toolkit*. Opendocs Llc, 2002.

Index