1 Introduction

The title of this book is "Python for Scientists", but what does that mean? The dictionary defines "Python" as either (a) a non-venomous snake from Asia or Saharan Africa or (b) a computer scripting language, and it is the second option which is intended here. (What exactly this second definition means will be explained later.) By "scientist", I mean anyone who uses quantitative models either to obtain conclusions by processing pre-collected experimental data or to model potentially observable results from a more abstract theory, **and** who asks "what if?". What if I analyze the data in a different way? What if I change the model? Thus the term also includes economists, engineers, mathematicians among others, as well as the usual concept of scientists. Given the volume of potential data or the complexity (non-linearity) of many theoretical models, the use of computers to answer these questions is fast becoming mandatory.

Advances in computer hardware mean that immense amounts of data or evermore complex models can be processed at increasingly rapid speeds. These advances also mean reduced costs so that today virtually every scientist has access to a "personal computer", either a desktop work station or a laptop, and the distinction between these two is narrowing quickly. It might seem to be a given that suitable software will also be available so that the "what if" questions can be answered readily. However, this turns out not always to be the case. A quick pragmatic reason is that while there is a huge market for hardware improvements, scientists form a very small fraction of it and so there is little financial incentive to improve scientific software. But for scientists, this issue is important and we need to examine it in more detail.

1.1 Scientific software

Before we discuss what is available, it is important to note that all computer software comes in one of two types: proprietary and open-source. The first is supplied by a commercial firm. Such organizations have both to pay wages and taxes and to provide a return for their shareholders. Therefore, they have to charge real money for their products, and, in order to protect their assets from their competitors, they do not tell the customer how their software works. Thus, the end-users have therefore little chance of being able to adapt or optimize the product for their own use. Since wages and taxes are recurrent expenditure, the company needs to issue frequent charged-for updates and improvements (the *Danegeld effect*). Open-source software is available for free or at

https://doi.org/10.1017/CBO9781107447875.002 Published online by Cambridge University Press

nominal cost (media, postage etc.). It is usually developed by computer literate individuals, often working for universities or similar organizations, who provide the service for their colleagues. It is distributed subject to anti-copyright licences, which give nobody the right to copyright it or to use it for commercial gain. Conventional economics might suggest that the gamut of open-source software should be inferior to its proprietary counterpart, or else the commercial organizations would lose their market. As we shall see, this is not necessarily the case.

Next we need to differentiate between two different types of scientific software. Computers operate according to a very limited and obscure set of instructions. A programming language is a somewhat less limited subset of human language in which sequences of instructions are written, usually by humans, to be read and understood by computers. The most common languages are capable of expressing very sophisticated mathematical concepts, albeit with a steep learning curve. Only a few language families, e.g., C and Fortran have been widely accepted, but they come with many different dialects, e.g., Fortran77, Fortran90, Ansi C, C++ etc. Compilers then translate code written by humans into machine code which can be optimized for speed and then processed. As such, they are rather like Formula 1 racing cars. The best of them are capable of breathtakingly fast performance, but driving them is not intuitive and requires a great deal of training and experience. Note that compilers need to be supplemented by libraries of software packages which implement frequently used numerical algorithms, and graphics packages will usually be needed. Fast versatile library packages are usually expensive, although good public domain packages are starting to appear.

A racing car is not usually the best choice for a trip to the supermarket, where speed is not of paramount importance. Similarly, compiled languages are not always ideal for trying out new mathematical ideas. Thus for the intended readers of this book the direct use of compilers is likely to be unattractive, unless their use is mandatory. We therefore look at the other type of software, usually called "scientific packages". Proprietary packages include Mathematica and Matlab, and open-source equivalents include Maxima, Octave, R and SciLab. They all operate in a similar fashion. Each provides its own idiosyncratic programming language in which problems are entered at a user interface. After a coherent group of statements, often just an individual statement, has been typed, the package writes equivalent core language code and compiles it on the fly. Thus errors and/or results can be reported immediately back to the user. Such packages are called "interpreters", and older readers may remember, perhaps with mixed feelings, the BASIC language. For small projects, the slow operation compared to a fully compiled code is masked by the speed of current microprocessors, but it does become apparent on larger jobs.

These packages are attractive for at least two reasons. The first is their ability to post-process data. For example, suppose that x is a real variable and there exists a (possibly unknown) function $y(x)$. Suppose also that for a set X of discrete instances of x we have computed a corresponding set Y of instances of y. Then a command similar to `plot(X,Y)` will display instantly a nicely formatted graph on the screen. Indeed, those generated by Matlab in particular are of publication quality. A second advantage is the apparent ability of some of the proprietary packages to perform in addition some

algebraic and analytic processes, and to integrate all of them with their numerical and graphical properties. A disadvantage of all of these packages is the quirky syntax and limited expressive ability of their command languages. Unlike the compiled languages, it is often extremely difficult to programme a process which was not envisaged by the package authors.

The best of the proprietary packages are very easy to use with extensive on-line help and coherent documentation, which has not yet been matched by all of the open-source alternatives. However, a major downside of the commercial packages is the extremely high prices charged for their licences. Most of them offer a cut down "student version" at reduced price (but usable only while the student is in full-time education) so as to encourage familiarity with the package. This largesse is paid for by other users.

Let us summarize the position. On the one hand, we have the traditional compiled languages for numerics which are very general, very fast, very difficult to learn and do not interact readily with graphical or algebraic processes. On the other, we have standard scientific packages which are good at integrating numerics, algebra and graphics, but are slow and limited in scope.

What properties should an ideal scientific package have? A short list might contain:

1 a programming language which is both easy to understand and which has extensive expressive ability,
2 integration of algebraic, numerical and graphical functions,
3 the ability to generate numerical algorithms running with speeds within an order of magnitude of the fastest of those generated by compiled languages,
4 a user interface with adequate on-line help, and decent documentation,
5 an extensive range of textbooks from which the curious reader can develop greater understanding of the concepts,
6 open-source software, freely available,
7 implementation on all standard platforms, e.g., Linux, Mac OS X, Unix, Windows.

The bad news is that no single package satisfies all of these criteria.

The major obstruction here is the requirement of algebraic capability. There are two open-source packages, wx-Maxima and Reduce with significant algebraic capabilities worthy of consideration, but Reduce fails requirement 4 and both fail criteria 3 and 5. They are however extremely powerful tools in the hands of experienced users. It seems sensible therefore to drop the algebra requirement.[1]

In 1991, Guido van Rossum created Python as a open-source platform-independent general purpose programming language. It is basically a very simple language surrounded by an enormous library of add-on modules, including complete access to the underlying operating system. This means that it can manage and manipulate programmes built from other complete (even compiled) packages, i.e., a *scripting* language. This versatility has ensured both its adoption by power users such as Google, and a real army of developers. It means also that it can be a very powerful tool for the scientist. Of course,

[1] The author had initially intended to write a book covering both numerical and algebraic applications for scientists. However, it turns out that, apart from simple problems, the requirements and approaches are radically different, and so it seems more appropriate to treat them differently.

there are other scripting languages, e.g., Java and Perl, but none has the versatility or user-base to meet criteria 3–5 above.

Five years ago it would not have been possible to recommend Python for scientific work. The size of the army of developers meant that there were several mutually incompatible add-on packages for numerical and scientific applications. Fortunately, reason has prevailed and there is now a single numerical add-on package *numpy* and a single scientific one *scipy* around which the developers have united.

1.2 The plan of this book

The purpose of this intentionally short book is to show how easy it is for the working scientist to implement and test non-trivial mathematical algorithms using Python. We have quite deliberately preferred brevity and simplicity to encyclopaedic coverage in order to get the inquisitive reader up and running as soon as possible. We aim to leave the reader with a well-founded framework to handle many basic, and not so basic, tasks. Obviously, most readers will need to dig further into techniques for their particular research needs. But after reading this book, they should have a sound basis for this.

This chapter and Appendix A discuss how to set up a scientific Python environment. While the original Python interpreter was pretty basic, its replacement *IPython* is so easy to use, powerful and versatile that Chapter 2 is devoted to it.

We now describe the subsequent chapters. As each new feature is described, we try to illustrate it first by essentially trivial examples and, where appropriate, by more extended problems. This author cannot know the mathematical sophistication of potential readers, but in later chapters we shall presume some familiarity with basic calculus, e.g., the Taylor series in one dimension. However, for these extended problems we shall sketch the background needed to understand them, and suitable references for further reading will be given.

Chapter 3 gives a brief but reasonably comprehensive survey of those aspects of the core Python language likely to be of most interest to scientists. Python is an object-oriented language, which lends itself naturally to object-oriented programming (OOP), which may well be unfamiliar to most scientists. We shall adopt an extremely light touch to this topic, but need to point out that the container objects introduced in Section 3.5 do not all have precise analogues in say C or Fortran. Again the brief introduction to Python *classes* in Section 3.9 may be unfamiliar to users of those two families of languages. The chapter concludes with two implementations of the *sieve of Eratosthenes*, which is a classical problem: enumerate all of the prime numbers[2] less than a given integer n. A straightforward implementation takes 17 lines of code, but takes inordinately long execution times once $n > 10^5$. However, a few minutes of thought and using already described Python features suggests a shorter 13 line programme which runs 3000 times faster and runs out of memory (on my laptop) once $n > 10^8$. The point of this exercise is

[2] The restriction to integer arithmetic in this chapter is because our exposition of Python has yet to deal with serious calculations involving real or complex numbers efficiently.

https://doi.org/10.1017/CBO9781107447875.002 Published online by Cambridge University Press

that choosing the right approach (and Python often offers so many) is the key to success in Python numerics.

Chapter 4 extends the core Python language via the add-on module *numpy*, to give a very efficient treatment of real and complex numbers. In the background, lurk C/C++ routines to execute repetitive tasks with near-compiled-language speeds. The emphasis is on using structures via *vectorized* code rather than the traditional for-loops or do-loops. Vectorized code sounds formidable, but, as we shall show, it is much easier to write than the old-fashioned loop-based approach. Here too we discuss the input and output of data. First, we look at how *numpy* can read and write text files, human-readable data and binary data. Secondly, we look briefly at data analysis. We summarize also miscellaneous functions and give a brief introduction to Python's linear algebra capabilities. Finally, we review even more briefly a further add-on module *scipy* which greatly extends the scope of *numpy*.

Chapter 5 gives an introduction to the add-on module *matplotlib*. This was inspired by the striking graphics performance of the Matlab package and aspires to emulate or improve on it for two-dimensional x, y-plots. Indeed, almost all of the figures in Chapters 5–9 were produced using *matplotlib*. The original figures were produced in colour using the relevant code snippets. The exigencies of book publishing have required conversion to black, white and many shades of grey. After giving a range of examples to illustrate its capabilities, we conclude the chapter with a slightly more extended example, a fully functional 49-line code to compute and produce high-definition plots of Mandelbrot sets.

The difficulties of extending the discussion to three-dimensional graphics, e.g., representations of the surface $z = z(x, y)$ are discussed in Chapter 6. Some aspects of this can be handled by the *matplotlib* module, but for more generality we need to invoke the *mayavi* add-on module, which is given a brief introduction together with some example codes. If the use of such graphics is a major interest for you, then you will need to investigate further these modules.

If you already have some Python experience, you can of course omit parts of Chapters 3 and 4. You are however encouraged strongly to try out the relevant code snippets. Once you have understood them, you can deepen your understanding by modifying them. These "hacking" experiments replace the exercises traditionally included in textbooks. The same applies to Chapters 5 and 6, which cover Python graphics and contain more substantial snippets. If you already have an idea of a particular picture you would like to create, then perusal of the examples given here and also those in the *matplotlib gallery* (see Section 5.1) should produce a recipe for a close approximation which can be "hacked" to provide a closer realization of the desired picture.

These first chapters cover the basic tools that Python provides to enhance the scientist's computer experience. How should we proceed further?

A notable omission is that apart from a brief discussion in Section 4.5, the vast subject of data analysis will not be covered. There are three main reasons for this.

1 Recently an add-on module *pandas* has appeared. This uses *numpy* and *matplotlib*

https://doi.org/10.1017/CBO9781107447875.002 Published online by Cambridge University Press

to tackle precisely this issue. It comes with comprehensive documentation, which is described in Section 4.5.

2 One of the authors of *pandas* has written a book, McKinney (2012), which reviews *IPython*, *numpy* and *matplotlib* and goes on to treat *pandas* applications in great detail.

3 I do not work in this area, and so would simply have to paraphrase the sources above.

Instead, I have chosen to concentrate on the modelling activities of scientists. One approach would be to target problems in bioinformatics or cosmology or crystallography or engineering or epidemiology or financial mathematics or . . . etc. Indeed, a whole series of books with a common first half could be produced called "Python for Bioinformatics" etc. A less profligate and potentially more useful approach would be to write a second half applicable to **all** of these fields, and many more. I am relying here on the unity of mathematics. Problems in one field when reduced to a core dimensionless form often look like a similarly reduced problem from another field.

This property can be illustrated by the following example. In population dynamics we might study a single species whose population $N(T)$ depends on time T. Given a plentiful food supply we might expect exponential growth, $dN/dT = kN(T)$, where the growth constant k has dimension 1/time. However, there are usually constraints limiting such growth. A simple model to include these is the "logistic equation"

$$\frac{dN}{dT}(T) = kN(T)\,(N_0 - N(T)) \tag{1.1}$$

which allows for a stable constant population $N(T) = N_0$. The biological background to this equation is discussed in many textbooks, e.g., Murray (2002).

In (homogeneous spherically symmetric) cosmology, the density parameter Ω depends on the scale factor a via

$$\frac{d\Omega}{da} = \frac{(1+3w)}{a}\Omega(1-\Omega), \tag{1.2}$$

where w is usually taken to be a constant.

Now mathematical biology and cosmology do not have a great deal in common, but it is easy to see that (1.1) and (1.2) represent the same equation. Suppose we scale the independent variable T in (1.1) by $t = kN_0T$, which renders the new time coordinate t dimensionless. Similarly, we introduce the dimensionless variable $x = N/N_0$ so that (1.1) becomes the logistic equation

$$\frac{dx}{dt} = x(1-x). \tag{1.3}$$

In a general relativistic theory, there is no reason to prefer any one time coordinate to any other. Thus we may choose a new time coordinate t via $a = e^{t/(1+3w)}$, and then setting $x = \Omega$, we see that (1.2) also reduces to (1.3). Thus the same equations can arise in a number of different fields.[3] In Chapters 7–9, we have, for brevity and simplicity, used minimal equations such as (1.3). If the minimal form for your problem looks something

[3] This example was chosen as a pedagogic example. If the initial value $x(0) = x_0$ is specified, then the exact

https://doi.org/10.1017/CBO9781107447875.002 Published online by Cambridge University Press

like the one being treated in a code snippet, you can of course hack the snippet to handle the original long form for your problem.

Chapter 7 looks at four types of problems involving ordinary differential equations. We start with a very brief introduction to techniques for solving initial value problems and then look at a number of examples, including two classic non-linear problems, the van der Pol oscillator and the Lorenz equations. Next we survey two-point boundary value problems and examine both a linear Sturm–Liouville eigenvalue problem, and an exercise in continuation for the non-linear Bratu problem. Problems involving delay differential equations arise frequently in control theory and in mathematical biology, e.g., the logistic and Mackey–Glass equations, and a discussion of their numerical solution is given in the next section. Finally in this chapter we look briefly at stochastic calculus and stochastic ordinary differential equations. In particular, we consider a simple example closely linked to the Black–Scholes equation of financial mathematics.

There are two other major Python topics relevant to scientists that I would like to introduce here. The first is the incorporation of code written in other languages. There are two aspects of this: (a) the reuse of pre-existing legacy code, usually written in Fortran, (b) if one's code is being slowed down seriously by a few Python functions, as revealed by the profiler, see Section 2.6, how do we recode the offending functions in Fortran or C? The second topic is how can a scientific user make worthwhile use of the object-oriented programming (OOP) features of Python?

Chapter 8 addresses the first topic via an extended example. We look first at how pseudospectral methods can be used to attack a large number of evolution problems governed by partial differential equations, either initial value or initial-boundary value problems. For the sake of brevity, we look only at problems with one time and one spatial dimension. Here, as we explain, problems with periodic spatial dependence can be handled very efficiently using Fourier methods, but for problems which are more general, the use of Chebyshev transforms is desirable. However, in this case there is no satisfactory Python black box available. It turns out that the necessary tools have already been written in legacy Fortran77 code. These are listed in Appendix B, and we show how, with an absolutely minimal knowledge of Fortran77, we can construct extremely fast Python functions to accomplish the required tasks. Our approach relies on the *numpy* `f2py` tool which is included in all of the recommended Python distributions. If you are interested in possibly reusing pre-existing legacy code, it is worthwhile studying this chapter even if the specific example treated there is not the task that you have in mind. See also Section 1.3 for other uses for `f2py`.

One of the most useful features of object-oriented programming (OOP) from the point of view of the scientist is the concept of *classes*. Classes exist in C++ (but not C) and Fortran90 and later (but not Fortran77). However, both implementations are complicated and so are usually shunned by novice programmers. In contrast, Python's implementation is much simpler and more user-friendly, at the cost of omitting some of the more arcane features of other language implementations. We give a very brief introduction to

solution is $x(t) = x_0/[x_0 + (1 - x_0)e^{-t}]$. In the current context, $x_0 \geqslant 0$. If $x_0 \neq 1$, then all solutions tend monotonically towards the constant solution $x = 1$ as t increases. See also Section 7.5.3.

https://doi.org/10.1017/CBO9781107447875.002 Published online by Cambridge University Press

the syntax in Section 3.9. However, in Chapter 9 we present a much more realistic example: the use of *multigrid* to solve elliptic partial differential equations in an arbitrary number of dimensions, although for brevity the example code is for two dimensions. Multigrid is by now a classical problem which is best described recursively, and we devote a few pages to describing it, at least in outline. The pre-existing legacy code is quite complicated because the authors needed to simulate recursion in languages, e.g., Fortran77, which do not support recursion. Of course, we could implement this code using the `f2py` tool outlined in Chapter 8. Instead, we have chosen to use Python classes and recursion to construct a simple clear multigrid code. As a concrete example, we use the sample problem from the corresponding chapter in Press et al. (2007) so that the inquisitive reader can compare the non-recursive and OOP approaches. If you have no particular interest in multigrid, but do have problems involving linked mathematical structures, and such problems arise often in, e.g., bioinformatics, chemistry, epidemiology, solid state physics among others, then you should certainly peruse this final chapter to see how, if you state reasonably mathematically precisely what your problems are, then it is easy to construct Python code to solve them.

1.3 Can Python compete with compiled languages?

The most common criticism of Python and the scientific software packages is that they are far too slow, in comparison with compiled code, when handling complicated realistic problems. The speed-hungry reader might like to look at a recent study[4] of a straightforward "number-crunching" problem treated by various methods. Although the figures given in the final section refer to one particular problem treated on a single processor, they do however give a "ball park" impression of performance. As a benchmark, they use the speed of a fully compiled C++ programme which solves the problem. A Python solution using the technique of Chapter 3, i.e., core Python, is about 700 times slower. Once you use the floating-point module *numpy* and the techniques described in Chapter 4 the code is only about ten times slower, and the Matlab performance is estimated to be similar. However, as the study indicates there are a number of ways to speed up Python to about 80% of the C++ performance. A number of these are very rewarding exercises in computer science.

One in particular though is extremely useful for scientists: the `f2py` tool. This is discussed in detail in Chapter 8 where we show how we can reuse legacy Fortran code. It can also be used to access standard Fortran libraries, e.g., the NAG libraries.[5] Yet another use is to speed up *numpy* code and so improve performance! To see how this works, suppose we have developed a programme such as those outlined in the later sections of the book, which uses a large number of functions, each of which carries out a simple task. The programme works correctly, but is unacceptably slow. Note that getting detailed timing data for Python code is straightforward. Python includes a "profiler" which can be run on the working programme. This outputs a detailed list of the functions

[4] See `http://wiki.scipy.org/PerformancePython`.
[5] See, e.g., `http://www.nag.co.uk/doc/TechRep/pdf/TR1_08.pdf`.

https://doi.org/10.1017/CBO9781107447875.002 Published online by Cambridge University Press

ordered by the time spent executing them. It is very easy to use, and this is described in Section 2.6. Usually, there are one or two functions which take very long times to execute simple algorithms.

This is where `f2py` comes into its own. Because the functions are simple, even beginners can soon create equivalent code in say Fortran77 or Ansi C. Also, because what we are coding is simple, there is no need for the elaborate (and laborious to learn) features of say Fortran95 or C++. Next we encapsulate the code in Python functions using the `f2py` tool, and slot them into the Python programme. With a little experience we can achieve speeds comparable to that of a programme written fully in say Fortran95.

1.4 Limitations of this book

A comprehensive treatment of Python and its various branches would occupy several large volumes and would be out of date before it reached the bookshops. This book is intended to offer the reader a starting point which is sufficient to be able to use the fundamental add-on packages. Once the reader has a little experience with what Python can do, it is time to explore further those areas which interest the reader.

I am conscious of the fact that I have not even mentioned vitally important concepts, e.g., finite-volume methods for hyperbolic problems,[6] parallel programming and real-time graphics to name but a few areas in which Python is very useful. There is a very large army of Python developers working at the frontiers of research, and their endeavours are readily accessed via the internet. Please think of this little book as a transport facility towards the front line.

1.5 Installing Python and add-ons

Users of Matlab and Mathematica are used to a customized *Integrated Development Environment (IDE)*. From the start-up screen, you can investigate code, write, edit and save segments using the built-in editor, and run actual programmes. Since the operating systems Mac OS X and most flavours of Linux include a version of core Python as a matter of course, many computer officers and other seasoned hackers will tell you that it is simple to install the additional packages, and you can be up and coding within the hour, thus ameliorating the difference.

Unfortunately, the pundits are wrong. The Python system being advocated in this book runs the language to its extreme limits, and all of the add-ons must be compatible with each other. Like many others, this author has endured hours of frustration trying to pursue the pundits' policy. Please save your energy, sanity etc., and read Appendix A, which I have quite deliberately targeted at novices, for the obvious reason!

Admittedly, there is an amount, albeit slight and low-level, of hassle involved here. So what's the payoff? Well if you follow the routes suggested in Appendix A, you

[6] The well-regarded Clawpack package `http://depts.washington.edu/clawpack`, which is Fortran-based, has switched from Matlab to Python *matplotlib* for its graphics support.

https://doi.org/10.1017/CBO9781107447875.002 Published online by Cambridge University Press

should end up with a system which works seamlessly. While it is true that the original Python interpreter was not terribly user-friendly, which caused all of the established IDE purveyors to offer a "Python mode", the need which they purported to supply has been overtaken by the enhanced interpreter *IPython*. Indeed, in its latest versions *IPython* hopes to surpass the facilities offered by Matlab, Mathematica and the Python-related features of commercial IDEs. In particular, it allows you to use your favourite editor, not theirs, and to tailor its commands to your needs, as explained in Chapter 2.

https://doi.org/10.1017/CBO9781107447875.002 Published online by Cambridge University Press