1 Introduction

1.1 Python for Scientists

The title of this book is *Python for Scientists*, but what does that mean? The dictionary defines “Python” as either (a) a nonvenomous snake from Asia or Saharan Africa or (b) a computer programming language, and it is the second option that is intended here. By “scientist,” we mean anyone who uses quantitative models either to obtain conclusions by processing precollected 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?

Given the steady progress in the development of evermore complex experiments that explore the inner workings of nature and generate vast amounts of data, as well as the necessity to describe these observations with complex (nonlinear) theoretical models, the use of computers to answer these questions is mandatory. Luckily, advances in computer hardware and software development mean that immense amounts of data or complex models can be processed at increasingly rapid speeds. It might seem 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, specialized, yet versatile, software tools are key to unraveling complex problems.

1.2 Scientific Software

Before we discuss what types of scientific software are available, it is important to note that all computer software comes in one of two types: proprietary or open-source. **Proprietary software** 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 little chance of being able to adapt or optimize the product for their own use.

Since wages and taxes are recurrent expenditures, the company needs to issue frequent charged-for updates and improvements (the *Danegeld effect*).

Open-source software, on the other hand, is available for free. 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 licenses, 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. The easiest approach to extracting insight from data or modeling observations utilizes prebuilt software tools, which we refer to as "**scientific software tools**." Proprietary examples include software tools and packages like Matlab, Mathematica, IDL, Tableau, or even Excel and open-source equivalents like R, Octave, SciLab, and LibreOffice. Some of these tools provide graphical user interfaces (GUIs) enabling the user to interact with the software in an efficient and intuitive way. Typically, such tools work well for standard tasks, but they do offer only a limited degree of flexibility, making it hard if not impossible to adapt these packages to solve some task they were not designed for. Other software tools provide more flexibility through their own idiosyncratic programming language in which problems are entered into a user interface. After a coherent group of statements, often just an individual statement, has been typed, the software writes equivalent core language code and compiles it on the fly. Thus errors and/or results can be reported back to the user immediately. Such tools are called "interpreters" as they interpret code on the fly, thus offering a higher degree of flexibility compared to software tools with shiny GUIs.

On a more basic level, the aforementioned software tools are implemented in a **programming language**, which is a somewhat 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 often with a steep learning curve. Although a myriad of programming languages exist, only a handful have been widely accepted and adopted for scientific applications. Historically, this includes C and Fortran, as well as their descendants. In the case of these so-called **compiled languages**, compilers translate code written by humans into machine code that 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. This experience is additionally complicated by the fact that compilers for the same language are not necessarily compatible and need to be supplemented by large libraries to provide functionality for seemingly basic functionality.

Since all scientific software tools are built upon compiled programming languages, why not simply write your own tools? Well, 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 quickly trying out new ideas or writing short scripts to support you in your daily work. 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 programming language, the so-called **interpreted languages**, which include the previously mentioned scientific tools based on interpreters. Interpreted languages lack the speed of compiled languages, but they typically are much more intuitive and easier to learn.

Let us summarize our position. There are prebuilt software tools, some of which are proprietary and some of which are open-source software, that provide various degrees of flexibility (interpreters typically offer more flexibility than tools that feature GUIs) and usually focus on specific tasks. On a more basic level, there are traditional compiled languages for numerics that are very general, very fast, rather difficult to learn, and do not interact readily with graphical or algebraic processes. Finally, there are interpreted languages that are typically much easier to learn than compiled languages and offer a large degree of flexibility but are less performant.

So, what properties should an ideal scientific software have? A short list might contain:

- ☐ a mature programming language that is both easy to understand and has extensive expressive ability,
- ☐ integration of algebraic, numerical, and graphical functions, and the option to import functionality from an almost endless list of supplemental libraries,
- ☐ the ability to generate numerical algorithms running with speeds within an order of magnitude of the fastest of those generated by compiled languages,
- ☐ a user interface with adequate on-line help and decent documentation,
- ☐ an extensive range of textbooks from which the curious reader can develop greater understanding of the concepts,
- ☐ open-source software, freely available,
- ☐ implementation on all standard platforms, e.g., Linux/Unix, Mac OS, Windows.
- ☐ a concise package, and thus implementable on even modest hardware.

You might have guessed it: we are talking about Python here.

In 1991, Guido van Rossum created Python as an open-source, platform-independent, general purpose programming language. It is basically a very simple language surrounded by an enormous library of add-on packages for almost any use case imaginable. Python is extremely versatile: it can be used to build complex software tools or as a scripting language to quickly get some task done. This versatility has both ensured its adoption by power users and led to the assembly of a large community of developers. These properties make Python a very powerful tool for scientists in their daily work and we hope that this book will help you master this tool.

1.3 About This Book

The purpose of this intentionally short book is to introduce the Python programming language and to provide an overview of scientifically relevant packages and how they can be utilized. This book is written for first-semester students and faculty members, graduate students and emeriti, high-school students and post-docs – or simply for everyone who is interested in using Python for scientific analysis.

However, this book by no means claims to be a complete introduction to Python. We leave the comprehensive treatment of Python and all its details to others who have done this with great success (see, e.g., Lutz, 2013). We have quite deliberately preferred brevity and simplicity over encyclopedic coverage in order to get the inquisitive reader up and running as soon as possible.

Furthermore, this book will not serve as the "Numerical Recipes for Python," meaning that we will not explain methods and algorithms in detail: we will simply showcase how they can be used and applied to scientific problems. For an in-depth discussion of these algorithms, we refer to the real *Numerical Recipes* – Press et al. (2007) and all following releases that were adapted to different programming languages – as well as other works.

Given the dynamic environment of software development, details on specific packages are best retrieved from online documentation and reference websites. We will provide references, links, and pointers in order to guide interested readers to the appropriate places. In order to enable an easy entry into the world of Python, we provide all code snippets presented in this book in the form of Jupyter Notebooks on the CoCalc cloud computing platform. These Notebooks can be accessed, run, and modified online for a more interactive learning experience.

We aim to leave the reader with a well-founded framework to handle many basic, and not so basic, tasks, as well as the skill set to find their own way in the world of scientific programming and Python.

1.4 References

Print Resources

Lutz, Mark. *Learning Python: Powerful Object-Oriented Programming*. O'Reilly Media, 2013.

Press, William H, et al. *Numerical Recipes: The Art of Scientific Computing*. 3rd ed., Cambridge University Press, 2007.