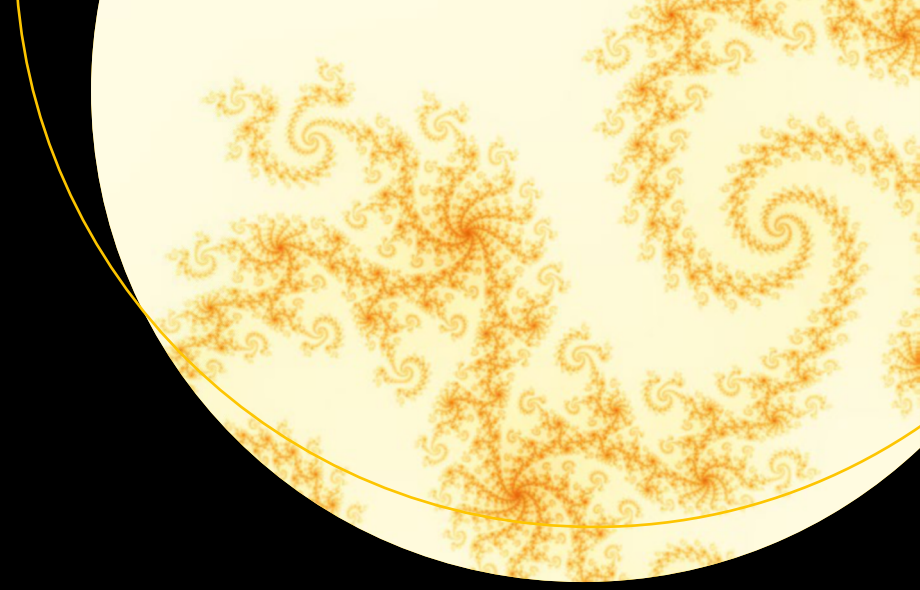

Numerical Python

Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib

Third Edition

Robert Johansson

Apress®

Numerical Python

Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib

Third Edition

Robert Johansson

Apress®

Numerical Python: Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib

Robert Johansson
Urayasu-shi, Chiba, Japan

ISBN-13 (pbk): 979-8-8688-0412-0 ISBN-13 (electronic): 979-8-8688-0413-7
https://doi.org/10.1007/979-8-8688-0413-7

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: James Markham
Coordinating Editor: Gryffin Winkler
Copyeditor: Kim Burton

Cover designed by eStudioCalamar

Cover image by author

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (https://github.com/Apress). For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

To Mika, Erika, and Mia

Table of Contents

About the Author xv

About the Technical Reviewer xvii

Introduction xix

Chapter 1: Introduction to Computing with Python 1

Environments for Computing with Python 4

Python 4

Interpreter 4

IPython Console 5

Input and Output Caching 6

Autocompletion and Object Introspection 7

Documentation 7

Interaction with the System Shell 8

IPython Extensions 8

Jupyter 12

The Jupyter QtConsole 13

The Jupyter Notebook 14

Jupyter Lab 16

Cell Types 17

Editing Cells 18

Markdown Cells 19

Rich Output Display 20

nbconvert 24

Spyder: An Integrated Development Environment 25
Source Code Editor 27
Consoles in Spyder 27
Object Inspector 28
Summary 28
Further Reading 28
Chapter 2: Vectors, Matrices, and Multidimensional Arrays 29
Importing the Modules 30
The NumPy Array Object 30
Data Types 31
Order of Array Data in Memory 33
Creating Arrays 34
Arrays Created from Lists and Other Array-Like Objects 35
Arrays Filled with Constant Values 36
Arrays Filled with Incremental Sequences 37
Arrays Filled with Logarithmic Sequences 37
Meshgrid Arrays 37
Creating Uninitialized Arrays 38
Creating Arrays with Properties of Other Arrays 38
Creating Matrix Arrays 39
Indexing and Slicing 40
One-Dimensional Arrays 40
Multidimensional Arrays 41
Views 42
Fancy Indexing and Boolean-Valued Indexing 43
Reshaping and Resizing 45
Vectorized Expressions 48
Arithmetic Operations 50
Elementwise Functions 52
Aggregate Functions 55

Boolean Arrays and Conditional Expressions 57
Set Operations 59
Operations on Arrays 60
Matrix and Vector Operations 61
Summary 65
Further Reading 66
Chapter 3: Symbolic Computing 67
Importing SymPy 67
Symbols 68
Numbers 70
Expressions 75
Manipulating Expressions 76
Simplification 76
Expand 77
Factor, Collect, and Combine 78
Apart, Together, and Cancel 79
Substitutions 79
Numerical Evaluation 80
Calculus 81
Derivatives 82
Integrals 83
Series 85
Limits 86
Sums and Products 87
Equations 88
Linear Algebra 90
Summary 93
Further Reading 93

Chapter 4: Plotting and Visualization 95
Importing Modules 96
Getting Started 96
Interactive and Noninteractive Modes 99
Figure 101
Axes 102
Plot Types 103
Line Properties 104
Legends 108
Text Formatting and Annotations 109
Axis Properties 110
Advanced Axes Layouts 119
Insets 119
Subplots 120
Subplot2grid 122
GridSpec 123
Colormap Plots 124
3D Plots 126
Summary 128
Further Reading 128
Chapter 5: Equation Solving 129
Importing Modules 130
Linear Equation Systems 130
Square Systems 131
Rectangular Systems 135
Eigenvalue Problems 138
Nonlinear Equations 139
Univariate Equations 140
Systems of Nonlinear Equations 145

Summary 149
Further Reading 149

Chapter 6: Optimization 151

Importing Modules 151
Classification of Optimization Problems 152
Univariate Optimization 154
Unconstrained Multivariate Optimization 156
Nonlinear Least Square Problems 162
Constrained Optimization 164
Linear Programming 168
Summary 170
Further Reading 170

Chapter 7: Interpolation 171

Importing Modules 171
Interpolation 172
Polynomials 173
Polynomial Interpolation 175
Spline Interpolation 179
Multivariate Interpolation 181
Summary 187
Further Reading 187

Chapter 8: Integration 189

Importing Modules 190
Numerical Integration Methods 190
Numerical Integration with SciPy 194
Tabulated Integrand 196
Multiple Integration 198
Symbolic and Arbitrary-Precision Integration 202

Line Integrals 204
Integral Transforms 204
Summary 207
Further Reading 207
Chapter 9: Ordinary Differential Equations 209
Importing Modules 209
Ordinary Differential Equations 210
Symbolic Solution to ODEs 211
Direction Fields 216
Solving ODEs Using Laplace Transformations 219
Numerical Methods for Solving ODEs 222
Numerical Integration of ODEs Using SciPy 225
Summary 236
Further Reading 236
Chapter 10: Sparse Matrices and Graphs 237
Importing Modules 237
Sparse Matrices in SciPy 238
Functions for Creating Sparse Matrices 241
Sparse Linear Algebra Functions 244
Linear Equation Systems 244
Graphs and Networks 249
Summary 255
Further Reading 256
Chapter 11: Partial Differential Equations 257
Importing Modules 258
Partial Differential Equations 258
Finite-Difference Methods 259
Finite-Element Methods 264
Survey of FEM Libraries 266

Solving PDEs Using FEniCS 267
Summary 285
Further Reading 285

Chapter 12: Data Processing and Analysis 287

Importing Modules 288
Introduction to Pandas 288
Series 288
DataFrame 290
Time Series 298
The Seaborn Graphics Library 307
Summary 312
Further Reading 312

Chapter 13: Statistics 315

Importing Modules 315
Review of Statistics and Probability 316
Random Numbers 317
Random Variables and Distributions 320
Hypothesis Testing 327
Nonparametric Methods 331
Summary 333
Further Reading 334

Chapter 14: Statistical Modeling 335

Importing Modules 336
Introduction to Statistical Modeling 336
Defining Statistical Models with Patsy 337
Linear Regression 345
Example Datasets 351
Discrete Regression 352

Logistic Regression 353
Poisson Model 357
Time Series 360
Summary 363
Further Reading 364
Chapter 15: Machine Learning 365
Importing Modules 366
Brief Review of Machine Learning 366
Regression 368
Classification 376
Clustering 380
Summary 384
Further Reading 384
Chapter 16: Bayesian Statistics 385
Importing Modules 386
Introduction to Bayesian Statistics 386
Model Definition 388
Sampling Posterior Distributions 393
Linear Regression 396
Summary 407
Further Reading 407
Chapter 17: Signal Processing 409
Importing Modules 409
Spectral Analysis 410
Fourier Transforms 410
Windowing 415
Spectrogram 418
Signal Filters 421

Convolution Filters 422
FIR and IIR Filters 424
Summary 428
Further Reading 428
Chapter 18: Data Input and Output 429
Importing Modules 430
Comma-Separated Values 430
HDF5 434
h5py 435
PyTables 444
Pandas HDFStore 447
Parquet 449
JSON 451
Serialization 454
Summary 456
Further Reading 456
Chapter 19: Code Optimization 459
Importing Modules 461
Numba 461
Cython 467
Summary 475
Further Reading 476
Appendix: Installation 477
Index 487

About the Author

Robert Johansson is an experienced Python programmer and computational scientist with a Ph.D. in Theoretical Physics from Chalmers University of Technology, Sweden. He has worked with scientific computing in academia and industry for over 15 years and participated in open source and proprietary research and development projects. His open-source contributions include work on QuTiP, a popular Python framework for simulating the dynamics of quantum systems, and he has also contributed to several other popular Python libraries in the scientific computing landscape. Robert is passionate about scientific computing and software development, teaching and communicating best practices for combining these fields with optimal outcomes: novel, reproducible, extensible, and impactful computational results.

About the Technical Reviewer

Shovon Sengupta is a distinguished data science expert specializing in advanced predictive analytics, machine learning, deep learning, and reinforcement learning. As the principal data scientist at the AI Center of Excellence for Fidelity Investment in the United States, Shovon is at the forefront of driving innovative initiatives that leverage artificial intelligence (specifically Generative AI) to solve complex business challenges. Shovon holds a US patent in automated predictive call routing using reinforcement learning.

He has also authored a few courses in the realm of machine learning. He has also presented at various international conferences on machine learning, time series forecasting, and building trustworthy artificial intelligence. His primary research is concentrated on deep reinforcement learning, deep learning, natural language processing, knowledge graphs, causality analysis, and time series analysis.

Shovon is also a PhD scholar specializing in applying machine learning algorithms in finance. His primary research interests include deep reinforcement learning, natural language processing, knowledge graphs, causality analysis, and time series analysis. His dedication to advancing the field of data science is evident in his continuous pursuit of knowledge and innovation.

For more details about Shovon's work, please check out his LinkedIn page at www.linkedin.com/in/shovon-sengupta-272aa917/.

Introduction

Scientific and numerical computing is a booming field in research, engineering, and analytics. The revolution in the computer industry over the last several decades has provided new and powerful tools for computational practitioners. This has enabled computational undertakings of previously unprecedented scale and complexity. Entire fields and industries have sprung up as a result. This development is ongoing, creating new opportunities as hardware, software, and algorithms keep improving. Ultimately, the enabling technology for this movement is the powerful computing hardware developed in recent decades. However, for a computational practitioner, the software environment used for computational work is as important as, if not more important than, the hardware on which the computations are carried out. This book is about one popular and fast-growing environment for numerical computing: the Python programming language and its vibrant ecosystem of libraries and extensions for computational work.

Computing is an interdisciplinary activity that requires experience and expertise in both theoretical and practical subjects: a firm understanding of mathematics and scientific thinking is a fundamental requirement for effective computational work. Equally important is solid training in computer programming and computer science. The role of this book is to bridge these two subjects by introducing how scientific computing can be done using the Python programming language and the computing environment that has appeared around this language. In this book, the reader is assumed to have some previous training in mathematics and numerical methods and basic knowledge of Python programming. The book's focus is to give a practical introduction to computational problem-solving with Python. Brief introductions to the theory of the covered topics are provided in each chapter to introduce notation and remind readers of the basic methods and algorithms. However, this book is not a self-consistent treatment of numerical methods. To assist readers who have yet to become familiar with some of the topics of this book, references for further reading are given at the end of each chapter. Likewise, readers without experience in Python programming will find it helpful to read this book with a book that focuses on the Python programming language itself.

How This Book Is Organized

The first chapter in this book introduces general principles for scientific computing and the main development environments available for computing in Python: the focus is on IPython and its interactive Python prompt, the excellent Jupyter Notebook application, and the Spyder IDE.

Chapter 2 introduces the NumPy library and generally discusses array-based computing and its virtues. Chapter 3 turns attention to symbolic computing—which in many respects complements array-based computing—using the SymPy library. Chapter 4 covers plotting and visualization using the Matplotlib library. These three chapters provide the basic computational tools used for domain-specific problems throughout the rest of the book: numerics, symbolics, and visualization.

Chapter 5 focuses on equation solving and explores numerical and symbolic methods, using the SciPy and SymPy libraries. Chapter 6 explores optimization, which is a natural extension of equation solving. It mainly works with the SciPy library and briefly with the cvxopt library. Chapter 7 deals with interpolation, another basic mathematical method with many applications, and important roles in higher-level algorithms and methods. Chapter 8 covers numerical and symbolic integration. Chapters 5 to 8 cover core computational techniques that are pervasive in all types of computational work. Most of the methods from these chapters are found in the SciPy library.

Chapter 9 covers ordinary differential equations. Chapter 10 is a detour into sparse matrices and graph methods, which helps prepare the field for the following chapter. Chapter 11 discusses partial differential equations, which conceptually are closely related to ordinary differential equations but require a different set of techniques that necessitates the introduction of sparse matrices, the topic of Chapter 10.

Chapter 12 changes direction and begins exploring data analysis and statistics. It introduces the Pandas library and its excellent data analysis framework. Chapter 13 covers basic statistical analysis and methods from the SciPy stats package. Chapter 14 moves on to statistical modeling using the statsmodels library. In Chapter 15, the theme of statistics and data analysis is continued with a discussion of machine learning using the scikit-learn library. Chapter 16 wraps up the statistics-related chapters with a discussion of Bayesian statistics and the PyMC library. Chapters 12 through 16 introduce the broad field of statistics and data analytics, which has been developing rapidly within and outside the scientific Python community in recent years.

Chapter 17 briefly returns to a core subject in scientific computing: signal processing. Chapter 18 discusses data input and output, and several methods for reading and writing numerical data to files, which is a basic topic required for most types of computational work. Chapter 19 introduces two methods for speeding up Python code using the Numba and Cython libraries.

The Appendix covers the installation of the software used in this book. This book uses the conda package manager to install the required software (mostly Python libraries). Conda can also be used to create virtual and isolated Python environments, which is an important topic for creating stable and reproducible computational environments. The Appendix also discusses how to work with such environments using the conda package manager.

Source Code Listings

Each chapter in this book has an accompanying Jupyter Notebook that contains the chapter's source code listings. These notebooks and the data files required to run them can be downloaded by visiting the book's GitHub page at `https://github.com/Apress/Numerical-Python-3rd-ed`.

CHAPTER 1

Introduction to Computing with Python

This book is about using Python for numerical computing. Python is a high-level, general-purpose interpreted programming language widely used in scientific computing and engineering. As a general-purpose language, Python was not specifically designed for numerical computing, but many of its characteristics make it well-suited for this task. First and foremost, Python is well known for its clean and easy-to-read code syntax. Good code readability improves maintainability, reduces bugs, and leads to better applications overall. It also enables rapid code development, since readability and expressiveness are essential in exploratory and interactive computing, where fast turnaround for testing various ideas and models is important.

In computational problem-solving, it is important to consider algorithms' performance and implementations. It is natural to strive for efficient high-performance code, and optimal performance is crucial for many computational problems. In such cases, it may be necessary to use a low-level program language, such as C or Fortran, to obtain the best performance out of the hardware that runs the code. However, it is not always the case that optimal runtime performance should be the highest priority. It is also important to consider the development time required to solve a problem in a programming language or an environment. While the best possible runtime performance can be achieved in a low-level programming language, working in a high-level language such as Python reduces the development time and often results in more flexible and extensible code.

These conflicting objectives present a trade-off between high performance but long development time and lower performance but shorter development time. Figure 1-1 shows a schematic visualization of this concept. When choosing a computational environment for solving a particular problem, it is important to consider this trade-off and to decide whether person-hours spent on the development or CPU-hours spent on running the computations are more valuable. It is worth noting that CPU-hours are cheap and getting even cheaper, but person-hours are expensive. Your own time is a very valuable resource. This makes a strong case for minimizing development time rather than the computation runtime by using a high-level programming language and environment such as Python and its scientific computing libraries.

R. Johansson, *Numerical Python*, https://doi.org/10.1007/979-8-8688-0413-7_1

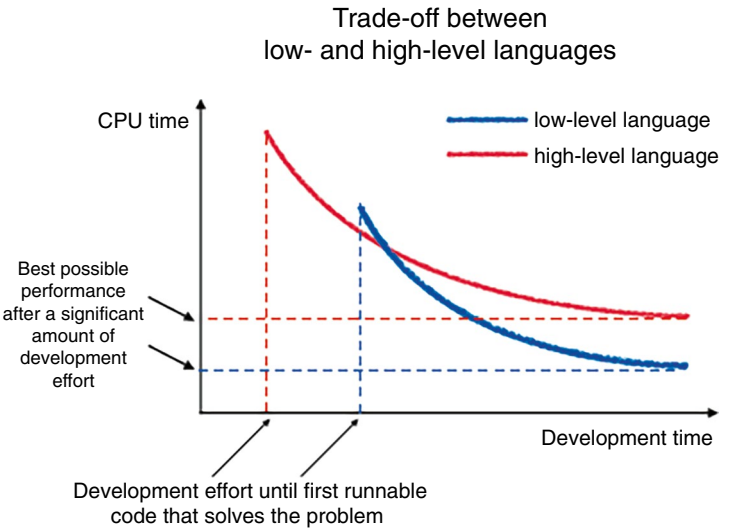

Figure 1-1. *The trade-off between low- and high-level programming languages. While a low-level language typically gives the best performance when a significant amount of development time is invested in the implementation of a solution to a problem, the development time required to obtain a first runnable code that solves the problem is typically shorter in a high-level language such as Python*

A solution that partially avoids the trade-off between high- and low-level languages is to use a multilanguage model, where a high-level language is used to interface libraries and software packages written in low-level languages. In a high-level scientific computing environment, this type of interoperability with software packages written in low-level languages (e.g., Fortran, C, or C++) is an important requirement. Python excels at this type of integration, and as a result, Python has become a popular "glue language" used as an interface for setting up and controlling computations that use code written in low-level programming languages for time-consuming number crunching. This is an important reason why Python is a popular language for numerical computing. The multilanguage model enables rapid code development in a high-level language while retaining most of the performance of low-level languages.

Due to the multilanguage model, scientific and technical computing with Python involves much more than just the Python language itself. In fact, the Python language is only a piece of an entire ecosystem of software and solutions that provide a complete environment for scientific and technical computing. This ecosystem includes development tools and interactive programming environments, such as Spyder and IPython, which are designed particularly with scientific computing in mind. It also includes a vast collection of Python packages for scientific computing. This ecosystem of scientifically oriented libraries ranges from generic core libraries—such as NumPy, SciPy, and Matplotlib—to more specific libraries for problem domains. Another crucial layer in the scientific Python stack exists below the various Python modules. Many scientific Python libraries interface with low-level, high-performance scientific software packages, such as optimized LAPACK and BLAS libraries[1] for low-level vector, matrix, and linear algebra routines or other specialized libraries for specific computational tasks. These libraries are typically implemented in a compiled low-level language and can be highly optimized and efficient. Without the foundation that such libraries provide, scientific computing with Python would not be practical. Figure 1-2 is an overview of the various layers of the software stack for computing with Python.

[1] For example, MKL, the Math Kernel Library from Intel at https://software.intel.com/en-us/intel-mkl; openBLAS at www.openblas.net; or ATLAS, the Automatically Tuned Linear Algebra Software at http://math-atlas.sourceforge.net

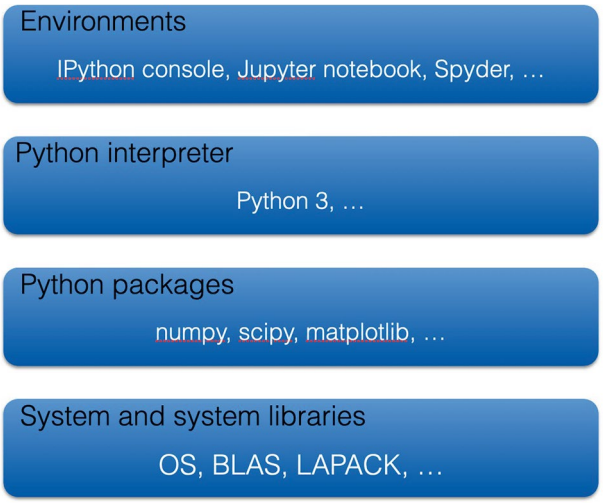

Figure 1-2. *An overview of the components and layers in the scientific computing environment for Python, from a user's perspective from top to bottom. Users typically only interact with the top three layers, but the bottom layer constitutes a very important part of the software stack*

■ **Tip** The SciPy organization (`www.scipy.org`) provides a centralized resource for information about the core packages in the scientific Python ecosystem, lists of additional specialized packages, and documentation and tutorials. It is a valuable resource when working with scientific and technical computing in Python. Another great resource is the Numeric and Scientific page on the official Python Wiki (`http://wiki.python.org/moin/NumericAndScientific`).

Besides the technical reasons why Python provides a good environment for computational work, it is also significant that it and its scientific computing libraries are free and open source. This eliminates economic constraints on when and how applications developed with the environment can be deployed and distributed by its users. Equally significant, it makes it possible for a dedicated user to obtain complete insight into how the language and the domain-specific packages are implemented and what methods are used. For academic work where transparency and reproducibility are hallmarks, this is increasingly recognized as an important requirement of software used in research. For commercial use, it provides freedom on how the environment is used and integrated into products and how such solutions are distributed to customers. All users benefit from the relief of not paying license fees, which may otherwise inhibit deployments on large computing environments, such as clusters and cloud computing platforms.

The social component of the scientific computing ecosystem for Python is another important aspect of its success. Vibrant user communities have emerged around the core packages and many domain-specific projects. Project-specific mailing lists, Stack Overflow groups, and issue trackers (e.g., on GitHub, `www.github.com`) are typically very active and provide forums for discussing problems and obtaining help, as well as a way of getting involved in developing these tools. The Python computing community also organizes yearly conferences and meet-ups at many venues around the world, such as the SciPy (`http://conference.scipy.org`) and PyData (`http://pydata.org`) conference series.

Environments for Computing with Python

Several different environments are suitable for working with Python for scientific and technical computing. This diversity has both advantages and disadvantages compared to a single endorsed environment that is common in proprietary computing products: diversity provides flexibility and dynamism that lends itself to specialization for particular uses, but on the other hand, it can also be confusing for new users, and it can be more complicated to set up a full productive environment. Here, I give an orientation of common environments for scientific computing so that their benefits can be weighed against each other and an informed decision can be reached regarding which one to use in different situations and for different purposes. The following are the three environments discussed in this chapter.

- The Python interpreter or the IPython console run code interactively. Together with a text editor for writing code, this provides a lightweight development environment.
- The Jupyter Notebook is a web application in which Python code can be written and executed through a web browser. This environment is great for numerical computing, analysis, and problem-solving because it allows us to collect the code, the output produced by the code, related technical documentation, and the analysis and interpretation all in one document.
- The Spyder Integrated Development Environment can write and interactively run Python code. An IDE like Spyder is a great tool for developing libraries and reusable Python modules.

These environments have justified uses, and it is largely a matter of personal preference for which one to use. However, I recommend exploring the Jupyter Notebook environment, because it is highly suitable for interactive and exploratory computing and data analysis, where data, code, documentation, and results are tightly connected. For the development of Python modules and packages, I recommend using the Spyder IDE because of its integration with code analysis tools and the Python debugger.

Python and the rest of the software stack required for scientific computing with Python can be installed and configured in many ways, and in general, the installation details also vary from system to system. The Appendix goes through one popular cross-platform method to install the tools and libraries required for this book.

Python

The Python programming language and the standard implementation of the Python interpreter are frequently updated and made available through new releases.[2] Currently, the active version of Python available for production use is the Python 3 series; this book requires Python 3.8 or greater. Note that at the time of writing, versions prior to Python 3.8 have already passed end-of-life, meaning they will no longer receive important bug fixes and security updates. Should you encounter any such legacy Python environment, it is therefore recommended that you upgrade the Python interpreter to a newer version.

Interpreter

The standard way to execute Python code is to run the program directly through the Python interpreter. On most systems, the Python interpreter is invoked using the `python` command. When a Python source file is passed as an argument to this command, the Python code in the file is executed.

[2] The Python language and the default Python interpreter are managed and maintained by the Python Software Foundation (www.python.org).

```
$ python hello.py
Hello from Python!
```

Here, the `hello.py` file contains a single line.

```
print("Hello from Python!")
```

To see which version of Python is installed, we can invoke the `python` command with the `--version` argument.

```
$ python --version
Python 3.11.4
```

It is common to have more than one version of Python installed on the same system. Each version of Python maintains its own set of libraries and provides its own interpreter command (so each Python environment can have different libraries installed). On many systems, specific versions of the Python interpreter are available through commands such as `python3.11`. It is also possible to set up *virtual* Python environments independent of the system-provided environments, which has many advantages. I strongly recommend becoming familiar with this way of working with Python. Appendix A describes setting up and working with these kinds of environments.

In addition to executing Python script files, a Python interpreter can be used as an interactive console (also known as a REPL (read–evaluate–print–loop)). Entering `python` at the command prompt (without any Python files as arguments) launches the Python interpreter in an interactive mode. When doing so, you are presented with a prompt.

```
$ python
Python 3.11.4 (main, Jul  5 2023, 08:41:25) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

From here, Python code can be entered, and for each statement, the interpreter evaluates the code and prints the result to the screen. The Python interpreter itself already provides a very useful environment for interactively exploring Python code, especially since the release of Python 3.4, which includes basic facilities such as a command history and basic autocompletion.

IPython Console

Although the interactive command-line interface provided by the standard Python interpreter has been greatly improved in the Python interpreter itself, it is still, in certain aspects, rudimentary, and it does not provide a complete environment for interactive computing. IPython[3] is an enhanced command-line REPL environment for Python, with additional interactive and exploratory computing features. For example, IPython provides improved command history browsing (also between sessions), an input and output caching system, improved auto-completion, more verbose and helpful exception tracebacks, and more. IPython is now much more than an enhanced Python command-line interface, which is explored in more detail later in this chapter and throughout the book. For instance, under the hood, IPython is a client-server application that separates the front end (user interface) from the back end (kernel) and executes the Python code. This allows multiple types of user interfaces to communicate and work with the same kernel, and a user-interface application can connect multiple kernels using IPython's framework for parallel computing.

[3] See the IPython project web page, http://ipython.org, for more information and its official documentation.

Running the `ipython` command launches the IPython command prompt.

```
$ ipython
Python 3.11.4 (main, Jul  5 2023, 08:41:25) [Clang 14.0.6 ]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.12.2 -- An enhanced Interactive Python. Type '?' for help
In [1]:
```

■ **Caution** Each IPython installation corresponds to a specific version of Python. If several versions of Python are available on your system, you may also have several versions of IPython. On many systems, IPython for Python 3 is invoked with the `ipython3` command, although the exact setup varies from system to system. Note that here, the "3" refers to the Python version, which differs from the version of IPython itself (at the time of writing it is 8.12.2).

The following sections briefly overview some of the IPython features that are most relevant to interactive computing. It is worth noting that IPython is used in many different contexts in scientific computing with Python, for example, as a kernel in the Jupyter Notebook application and in the Spyder IDE, which is covered in more detail later in this chapter. It is time well spent to get familiar with the tricks and techniques that IPython offers to improve your productivity when working with interactive computing.

Input and Output Caching

In the IPython console, the input prompt is denoted as `In [1]:` and the corresponding output is denoted as `Out [1]:`, where the numbers within the square brackets are incremented for each new input and output. These inputs and outputs are called *cells* in IPython. The input and the output of previous cells can later be accessed through the `In` and `Out` variables that IPython automatically creates. The `In` and `Out` variables are a list and a dictionary, respectively, that can be indexed with a cell number. For instance, consider the following IPython session.

```
In [1]: 3 * 3
Out[1]: 9
In [2]: In[1]
Out[2]: '3 * 3'
In [3]: Out[1]
Out[3]: 9
In [4]: In
Out[4]: ['', '3 * 3', 'In[1]', 'Out[1]', 'In']
In [5]: Out
Out[5]: {1: 9, 2: '3 * 3', 3: 9, 4: ['', '3 * 3', 'In[1]', 'Out[1]', 'In', 'Out']}
```

Here, the first input was `3 * 3`, and the result was `9`, which later is available as `In[1]` and `Out[1]`. A single underscore `_` is a shorthand notation for referring to the most recent output, and a double underscore `__` refers to the output that preceded the most recent output. Input and output caching is often useful in interactive and exploratory computing since the result of a computation can be accessed even if it was not explicitly assigned to a variable.

Note that when a cell is executed, the value of the last statement in an input cell is, by default, displayed in the corresponding output cell unless the statement is an assignment or if the value is Python null value `None`. The output can be suppressed by ending the statement with a semicolon.

```
In [6]: 1 + 2
Out[6]: 3
In [7]: 1 + 2;    # output suppressed by the semicolon
In [8]: x = 1     # no output for assignments
In [9]: x = 2; x  # these are two statements. The value of 'x' is shown in
                  # the output
Out[9]: 2
```

Autocompletion and Object Introspection

In IPython, pressing the TAB key activates autocompletion, which displays a list of symbols (variables, functions, classes, etc.) with names that are valid completions of what has already been typed. The autocompletion in IPython is contextual, and it looks for matching variables and functions in the current namespace or among the attributes and methods of a class when invoked after the name of a class instance. For example, `os.<TAB>` produces a list of the variables, functions, and classes in the `os` module, and pressing TAB after typing `os.w` results in a list of symbols in the `os` module that starts with `w.`

```
In [10]: import os
In [11]: os.w<TAB>
os.wait  os.wait3  os.wait4  os.waitpid  os.walk  os.write  os.writev
```

This feature is called *object introspection*, a powerful tool for interactively exploring the properties of Python objects. Object introspection works on modules, classes, attributes, methods, functions, and arguments.

Documentation

Object introspection is convenient for exploring the API of a module, such as its member classes and functions, and together with the documentation strings or *docstrings* that are commonly provided in Python code, it provides a built-in dynamic reference manual for almost any Python module that is installed and can be imported. A Python object followed by a question mark displays the documentation string for the object. This is similar to the Python function `help`. An object can also be followed by two question marks, in this case, IPython tries to display more detailed documentation, including the Python source code, if available. For example, to display help for the `cos` function in the `math` library.

```
In [12]: import math
In [13]: math.cos?
Signature: math.cos(x, /)
Docstring: Return the cosine of x (measured in radians).
Type:      builtin_function_or_method
```

Docstrings can be specified for Python modules, functions, classes, and their attributes and methods. A well-documented module includes full API documentation in the code itself. From a developer's point of view, it is convenient to document a code together with the implementation. This encourages writing and maintaining documentation, and Python modules tend to be well-documented.

Interaction with the System Shell

IPython also provides extensions to the Python language that make interacting with the underlying system convenient. Anything that follows an exclamation mark is evaluated using the system shell (such as bash shell). For example, on a Unix-like system, such as Linux or macOS, listing files in the current directory can be done using the following.

```
In[14]: !ls
file1.py    file2.py    file3.py
```

In Microsoft Windows, the equivalent command would be `!dir`. This method for interacting with the operating system is a powerful feature that makes it easy to navigate the file system and use the IPython console as a system shell. The output generated by a command following an exclamation mark can easily be captured in a Python variable. For example, a file listing produced by `!ls` can be stored in a Python list using the following.

```
In[15]: files = !ls
In[16]: len(files)
3
In[17] : files
['file1.py', 'file2.py', 'file3.py']
```

Likewise, we can pass the values of Python variables to shell commands by prefixing the variable name with a $ sign.

```
In[18]: file = "file1.py"
In[19]: !ls -l $file
-rw-r--r--  1 rob  staff 131 Oct 22 16:38 file1.py
```

This two-way communication with the IPython console and the system shell can be very convenient, for example, when processing data files.

IPython Extensions

IPython provides extension commands that are called *magic functions* in IPython terminology. These commands all start with one or two % signs.[4] A single % sign is used for one-line commands, and two % signs are used for commands that operate on cells (multiple lines). For a complete list of available extension commands, type `%lsmagic`, and the documentation for each command can be obtained by typing the magic command followed by a question mark.

```
In[20]: %lsmagic?
Docstring: List currently available magic functions.
File:           /usr/local/lib/python3.6/site-packages/IPython/core/magics/basic.py
```

[4] When `%automagic` is activated (type `%automagic` at the IPython prompt to toggle this feature), the % sign that precedes the IPython commands can be omitted, unless there is a name conflict with a Python variable or function. However, for clarity, the % signs are explicitly shown here.

File System Navigation

In addition to the interaction with the system shell described in the previous section, IPython provides commands for navigating and exploring the file system. These commands are familiar to Unix shell users: `%ls` (list files), `%pwd` (return current working directory), `%cd` (change working directory), `%cp` (copy file), `%less` (show the content of a file in the pager), and `%%writefile filename` (write content of a cell to the file `filename`). Note that autocomplete in IPython also works with the files in the current working directory, which makes IPython as convenient to explore the file system as the system shell. It is worth noting that these IPython commands are system-independent and can be used on both Unix-like operating systems and Windows.

Running Scripts from the IPython Console

The `%run` command is an important and useful extension, perhaps one of the most important features of the IPython console. This command can execute an external Python source code file within an interactive IPython session. Keeping a session active between multiple runs of a script makes it possible to explore the variables and functions defined in a script interactively after the execution of the script has finished. To demonstrate this functionality, consider a script file `fib.py` that contains the following code.

```
def fib(n):
    """
    Return a list of the first n Fibonacci numbers.
    """
    f0, f1 = 0, 1
    f = [1] * n
    for i in range(1, n):
        f[i] = f0 + f1
        f0, f1 = f1, f[i]
    return f

print(fib(10))
```

It defines a function that generates a sequence of n Fibonacci numbers and prints the result for $n = 10$ to the standard output. It can be run from the system terminal using the standard Python interpreter.

```
$ python fib.py
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

It can also be run from an interactive IPython session, which produces the same output but also adds the symbols defined in the file to the local namespace so that the `fib` function is available in the interactive session after the `%run` command has been issued.

```
In [21]: %run fib.py
Out[22]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
In [23]: %who
fib
In [23]: fib(6)
Out[23]: [1, 1, 2, 3, 5, 8]
```

The preceding example also used the %who command, which lists all defined symbols (variables and functions).[5] The %whos command is similar, but also gives more detailed information about the type and value of each symbol, when applicable.

Debugger

IPython includes a handy debugger mode, which can be invoked postmortem after a Python exception (error) has been raised. After the traceback of an unintercepted exception has been printed to the IPython console, it is possible to step directly into the Python debugger using the IPython command %debug. This possibility can eliminate the need to rerun the program from the beginning using the debugger or after employing the common debugging method of sprinkling print statements into the code. If the exception is unexpected and happens late in a time-consuming computation, this can be a big time-saver.

To see how the %debug command can be used, consider the following incorrect invocation of the fib function defined earlier. It is incorrect because a float is passed to the function while the function is implemented, assuming that the argument passed to it is an integer. On line 7 the code runs into a type error, and the Python interpreter raises an exception of TypeError. IPython catches the exception and prints a useful traceback of the call sequence on the console. If we are clueless about why the code on line 7 contains an error, entering the debugger by typing %debug in the IPython console could be useful. We then get access to the local namespace at the source of the exception, which can allow us to explore in more detail why the exception was raised.

```
In [24]: fib(1.0)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-24-874ca58a3dfb> in <module>()
 ----> 1 fib.fib(1.0)
/Users/rob/code/fib.py in fib(n)
      5     """
      6     f0, f1 = 0, 1
 ----> 7     f = [1] * n
      8     for i in range(1, n):
      9         f[n] = f0 + f1
TypeError: can't multiply sequence by non-int of type 'float'
In [25]: %debug
> /Users/rob/code/fib.py(7)fib()
      6    f0, f1 = 0, 1
----> 7    f = [1] * n
      8     for i in range(1, n):
ipdb> print(n)
1.0
```

■ **Tip** Type a question mark at the debugger prompt to show a help menu that lists available commands.

```
ipdb> ?
```

More information about the Python debugger and its features is available in the Python Standard Library documentation: http://docs.python.org/3/library/pdb.html.

---

[5] The Python function dir provides a similar feature.

Reset

Resetting the namespace of an IPython session is often useful to ensure that a program is run in a pristine environment, uncluttered by existing variables and functions. The `%reset` command provides this functionality (use the `-f` flag to force the reset). Using this command can often eliminate the need for otherwise common exit-restart cycles of the console. Although it is necessary to reimport modules after the `%reset` command has been used, it is important to know that even if the modules have changed since the last import, a new import after a `%reset` does not import the new module but rather reenable a cached version of the module from the previous import. When developing Python modules, this is usually not the desired behavior. In that case, a reimport of a previously imported (and since updated) module can often be achieved by using the `reload` function from `IPython.lib.deepreload`. However, this method does not always work, as some libraries run code at import time that is only intended to run once. In this case, the only option might be to terminate and restart the IPython interpreter.

Timing and Profiling Code

The `%timeit` and `%time` commands provide simple benchmarking facilities useful when looking for bottlenecks and attempting to optimize code. The `%timeit` command runs a Python statement several times and estimates the runtime (use `%%timeit` to do the same for a multiline cell). The exact number of times the statement is run is determined heuristically unless explicitly set using the `-n` and `-r` flags. See `%timeit?` for details. The `%timeit` command does not return the resulting value of the expression. If the result of the computation is required, the `%time` or `%%time` (for a multiline cell) commands can be used instead, but `%time` and `%%time` only run the statement once and give a less accurate estimate of the average runtime.

The following example demonstrates a typical usage of the `%timeit` and `%time` commands.

```
In [26]: %timeit fib(100)
100000 loops, best of 3: 16.9 µs per loop
In [27]: result = %time fib(100)
CPU times: user 33 µs, sys: 0 ns, total: 33 µs
Wall time: 48.2
```

While the `%timeit` and `%time` commands are useful for measuring the elapsed runtime of a computation, they do not give detailed information about what part of the computation takes more time. Such analyses require a more sophisticated code profiler, such as the one provided by the Python standard library module `cProfile`.[6] The Python profiler is accessible in IPython through the `%prun` (for statements) and `%run` commands with the `-p` flag (for running external script files). The output from the profiler is rather verbose and can be customized using optional flags to the `%prun` and `%run -p` commands (see `%prun?` for a detailed description of the available options).

As an example, consider a function that simulates *N* random walkers, each taking *M* steps, and then calculates the furthest distance from the starting point achieved by any of the random walkers.

```
In [28]: import numpy as np
In [29]: def random_walker_max_distance(M, N):
    ...:     """
    ...:     Simulate N random walkers taking M steps, and return the largest
    ...:     Distance from the starting point achieved by any of the random
    ...:     walkers.
```

[6] Which can, for example, be used with the standard Python interpreter to profile scripts by running `python -m cProfile script.py`

```
...:     """
...:     trajectories = [np.random.randn(M).cumsum() for _ in range(N)]
...:     return np.max(np.abs(trajectories))
```

Calling this function using the profiler with `%prun` results in the following output, which includes information about how many times each function was called and a breakdown of the total and cumulative time spent in each function. From this information, we can conclude that in this simple example, the calls to the `np.random.randn` function consume the bulk of the elapsed computation time.

```
In [30]: %prun random_walker_max_distance(400, 10000)

         20011 function calls in 0.285 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    10000    0.181    0.000    0.181    0.000 {method 'randn' of
                                               'mtrand.RandomState' objects}
    10000    0.053    0.000    0.053    0.000 {method 'cumsum' of
                                               'numpy.ndarray' objects}
        1    0.020    0.020    0.277    0.277 2615584822.py:3(
                                               random_walker_max_distance)
        1    0.019    0.019    0.253    0.253 2615584822.py:8(<listcomp>)
        1    0.008    0.008    0.285    0.285 <string>:1(<module>)
        1    0.004    0.004    0.004    0.004 {method 'reduce' of
                                               'numpy.ufunc' objects}
        1    0.000    0.000    0.285    0.285 {built-in method builtins.exec}
        1    0.000    0.000    0.004    0.004 fromnumeric.py:71(_wrapreduction)
        1    0.000    0.000    0.004    0.004 fromnumeric.py:2692(max)
        1    0.000    0.000    0.000    0.000 fromnumeric.py:72(<dictcomp>)
        1    0.000    0.000    0.000    0.000 fromnumeric.py:2687(_max_dispatcher)
        1    0.000    0.000    0.000    0.000 {method 'disable' of
                                               '_lsprof.Profiler' objects}
        1    0.000    0.000    0.000    0.000 {method 'items' of 'dict' objects}
```

Interpreter and Text Editor as Development Environment

In principle, the Python or the IPython interpreter and a good text editor are all required for a fully productive Python development environment. This simple setup is, in fact, the preferred development environment for many experienced programmers. However, the following sections look at the Jupyter Notebook and Spyder's integrated development environment. These environments provide richer features that improve productivity when working with interactive and exploratory computing applications.

Jupyter

The Jupyter project[7] is a spin-off from the IPython project that includes the Python independent frontends—most notably the notebook application, which is discussed in more detail in the following section—and the communication framework that enables the separation of the frontend from the computational backends,

[7] For more information about Jupyter, see http://jupyter.org.

known as *kernels*. Prior to the creation of the Jupyter project, the notebook application and its underlying framework were a part of the IPython project. However, because the notebook frontend is language agnostic (it can also be used with many other languages, such as R and Julia), it was spun off a separate project to better cater to the wider computational community and avoid a perceived bias toward Python. Now, the remaining role of IPython is to focus on Python-specific applications, such as the interactive Python console, and to provide a Python kernel for the Jupyter environment.

In the Jupyter framework, the front end can be connected to multiple computational backend kernels, for example, for different programming languages, versions of Python, or for different Python environments. The kernel maintains the state of the interpreter. It performs the actual computations, while the front end manages how code is entered and organized and how the results of calculations are visualized to the user.

This section discusses the Jupyter QtConsole and Notebook frontends. It briefly introduces some of their rich display and interactivity features and the workflow organization that the notebook provides. The Jupyter Notebook is the Python environment for computational work that I generally recommend in this book, and the code listings in the rest of this book are understood to be read as if they are cells in a notebook.

The Jupyter QtConsole

The Jupyter QtConsole is an enhanced console application that can substitute for the standard IPython console. The QtConsole is launched by passing the `qtconsole` argument to the `jupyter` command.

```
$ jupyter qtconsole
```

This opens a new IPython application in a console that can display rich media objects such as images, figures, and mathematical equations. The Jupyter QtConsole also provides a menu-based mechanism for displaying autocompletion results, and it shows docstrings for functions in a pop-up window when typing the opening parenthesis of a function or a method call. A screenshot of the Jupyter Qtconsole is shown in Figure 1-3.

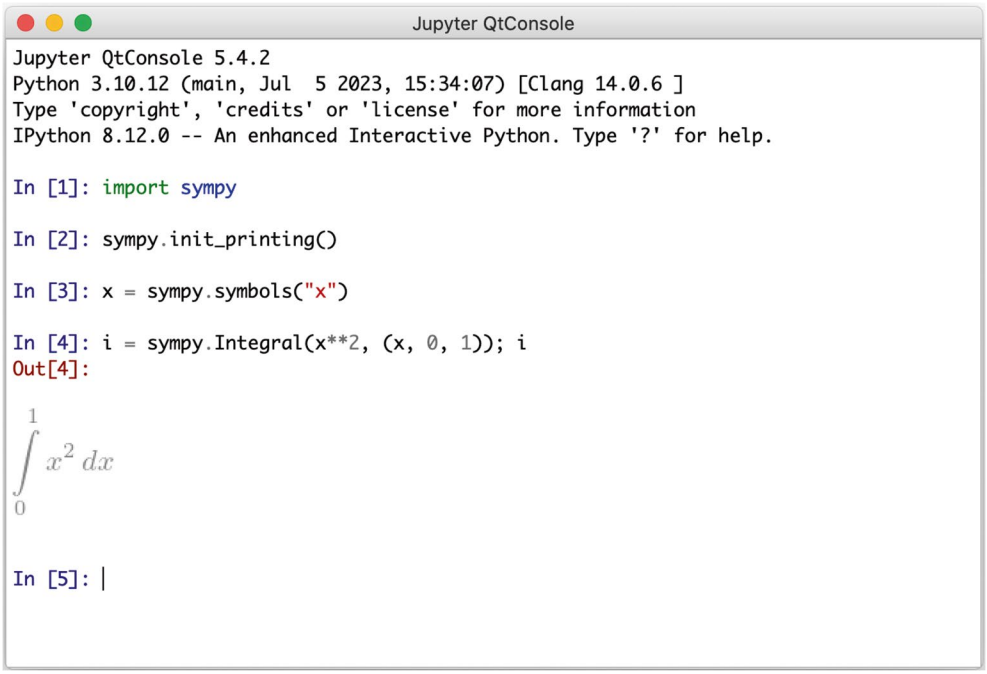

Figure 1-3. *A screenshot of the Jupyter QtConsole application*