Numerical Python

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

Second Edition

Robert Johansson

Apress®

Numerical Python

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

Second 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): 978-1-4842-4245-2 ISBN-13 (electronic): 978-1-4842-4246-9
https://doi.org/10.1007/978-1-4842-4246-9

Library of Congress Control Number: 2018966798

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Todd Green
Development Editor: James Markham
Coordinating Editor: Jill Balzano

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. 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 editorial@apress.com; for reprint, paperback, or audio rights, please email 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 via the book's product page, located at www.apress.com/9781484242452. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To Mika and Erika.

Table of Contents

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 10 years, and he has participated in both open source development and proprietary research 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 and about teaching and communicating best practices for bringing these fields together with optimal outcome: novel, reproducible, and extensible computational results. Robert's background includes 5 years of postdoctoral research in theoretical and computational physics, and he is now working as a data scientist in the IT industry.

About the Technical Reviewers

Massimo Nardone has more than 24 years of experiences in security, web/mobile development, cloud, and IT architecture. His true IT passions are security and Android.

He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

He holds an M.Sc. degree in computing science from the University of Salerno, Italy.

He has worked as a project manager, software engineer, research engineer, chief security architect, information security manager, PCI/SCADA auditor, and senior lead IT security/cloud/SCADA architect for many years.

His technical skills include security, Android, cloud, Java, MySQL, Drupal, Cobol, Perl, web and mobile development, MongoDB, D3, Joomla!, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, etc.

He worked as visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (PKI, SIP, SAML, and Proxy areas).

He currently works as chief information security officer (CISO) for Cargotec Oyj, and he is a member of the ISACA Finland Chapter Board.

Massimo has reviewed more than 45 IT books for different publishers and has coauthored *Pro JPA 2 in Java EE 8* (Apress, 2018), *Beginning EJB in Java EE 8* (Apress, 2018), and *Pro Android Games* (Apress, 2015).

Chinmaya Patnayak is an embedded software developer at NVIDIA and is skilled in C++, CUDA, deep learning, Linux, and file systems. He has been a speaker and instructor for deep learning at various major technology events across India. Chinmaya holds an M.Sc. degree in physics and B.E. in electrical and electronics engineering from BITS Pilani. He has previously worked with Defence Research and Development Organization (DRDO) on encryption algorithms for video streams. His current interest lies in neural networks for image segmentation and applications in biomedical research and self-driving cars. Find more about him at http://chinmayapatnayak.github.io.

Michael Thomas has worked in software development for more than 20 years as an individual contributor, team lead, program manager, and vice president of engineering. Michael has more than 10 years of experience working with mobile devices. His current focus is in the medical sector, using mobile devices to accelerate information transfer between patients and health-care providers.

David Stansby is a Ph.D. student at Imperial College London and an active Python developer. He is on the core development team of Matplotlib, Python's most popular plotting library, and the creator of HelioPy, a Python package for space science data analysis.

Jason Whitehorn is an experienced entrepreneur and software developer and has helped many oil and gas companies automate and enhance their oilfield solutions through field data capture, SCADA, and machine learning. Jason obtained his B.SC. in computer science from Arkansas State University, but he traces his passion for development back many years before then, having first taught himself to program BASIC on his family's computer while still in middle school.

When he's not mentoring and helping his team at work, writing, or pursuing one of his many side projects, Jason enjoys spending time with his wife and four children and living in the Tulsa, Oklahoma region. More information about Jason can be found on his web site: `https://jason.whitehorn.us`.

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 still ongoing, and it is creating new opportunities as hardware, software, and algorithms keep improving. Ultimately the enabling technology for this movement is the powerful computing hardware that has been 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 about Python programming. The focus of the book is to give a practical introduction to computational problem-solving with Python. Brief introductions to the theory of the covered topics are given 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 that are not previously 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 probably find it useful to read this book together 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 that are available for work with computing in Python: the focus is on IPython and its interactive Python prompt, the excellent Jupyter Notebook application, and the Spyder IDE.

In Chapter 2, an introduction to the NumPy library is given, and here we also discuss more generally array-based computing and its virtues. In Chapter 3, we turn our attention to symbolic computing – which in many respects complements array-based computing – using the SymPy library. In Chapter 4, we cover plotting and visualization using the Matplotlib library. Together, Chapters 2 to 4 provide the basic computational tools that will be used for domain-specific problems throughout the rest of the book: numerics, symbolics, and visualization.

In Chapter 5, the topic of study is equation solving, which we explore with both numerical and symbolic methods, using the SciPy and SymPy libraries. In Chapter 6, we explore optimization, which is a natural extension of equation solving. Here we mainly work with the SciPy library and briefly with the cvxopt library. Chapter 7 deals with interpolation, which is another basic mathematical method with many applications of its own, and important roles in higher-level algorithms and methods. In Chapter 8, we cover 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.

In Chapter 9, we proceed to cover ordinary differential equations. Chapter 10 is a detour into sparse matrices and graph methods, which helps prepare the field for the following chapter. In Chapter 11, we discuss 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.

Starting with Chapter 12, we make a change of direction and begin exploring data analysis and statistics. In Chapter 12, we introduce the Pandas library and its excellent data analysis framework. In Chapter 13, we cover basic statistical analysis and methods from the SciPy `stats` package. In Chapter 14, we move 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. In Chapter 16, we wrap up the statistics-related chapters with a discussion of Bayesian statistics and the PyMC library. Together, Chapters 12 to 16 provide an introduction to the broad field of statistics and data analytics: a field that has been developing rapidly within and outside of the scientific Python community in recent years.

In Chapter 17, we briefly return to a core subject in scientific computing: signal processing. In Chapter 18, we discuss data input and output, and several methods for reading and writing numerical data to files, which is a basic topic that is required for most types of computational work. In Chapter 19, the final regular chapter in this book, two methods for speeding up Python code are introduced, using the Numba and Cython libraries.

The Appendix covers the installation of the software used in this book. To install the required software (mostly Python libraries), we use the conda package manager. 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 clicking the **Download Source Code** button located at `www.apress.com/9781484242452`.

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 that is 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, which in general results in fewer bugs and better applications overall, but it also enables rapid code development. This readability and expressiveness are essential in exploratory and interactive computing, which requires fast turnaround for testing various ideas and models.

In computational problem-solving, it is, of course, important to consider the performance of algorithms and their implementations. It is natural to strive for efficient high-performance code, and optimal performance is indeed 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 is the most suitable objective. It is also important to consider the development time required to implement a solution to a problem in a given programming language or 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 usually reduces the development time and often results in more flexible and extensible code.

These conflicting objectives present a trade-off between high performance and long development time and lower performance but shorter development time. See Figure 1-1 for 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 man-hours spent on the development or CPU-hours spent on

R. Johansson, *Numerical Python*, https://doi.org/10.1007/978-1-4842-4246-9_1

running the computations is more valuable. It is worth noting that CPU-hours are cheap already and are getting even cheaper, but man-hours are expensive. In particular, your own time is of course a very valuable resource. This makes a strong case for minimizing development time rather than the runtime of a computation by using a high-level programming language and environment such as Python and its scientific computing libraries.

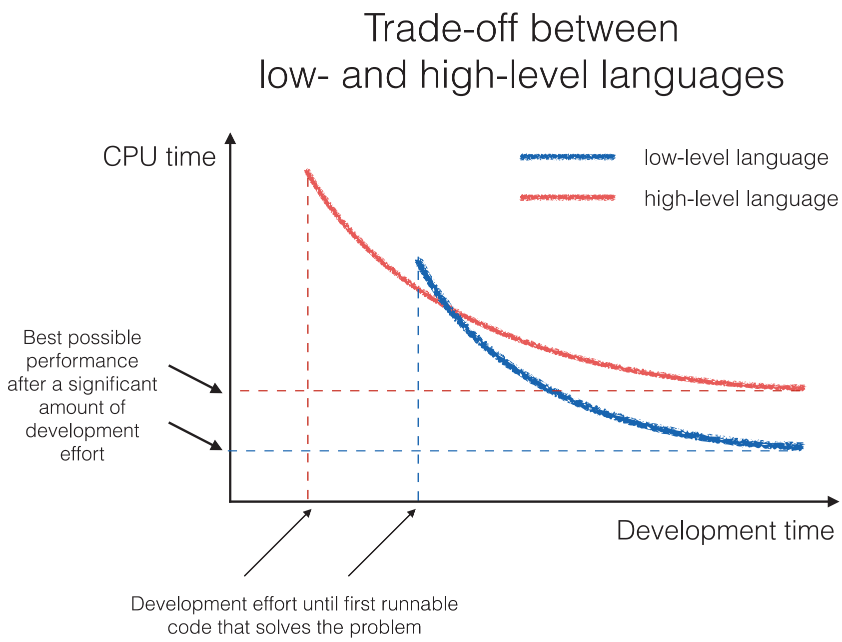

Figure 1-1. *Trade-off between low- and high-level programming languages. While a low-level language typically gives the best performance when a significant amount of development time is invested in the implementation of a solution to a problem, the development time required to obtain a first runnable 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 for 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.

As a consequence of 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 particular problem domains. Another crucial layer in the scientific Python stack exists below the various Python modules: many scientific Python library interface, in one way or another; low-level high-performance scientific software packages, such as for example 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 therefore be optimized and efficient. Without the foundation that such libraries provide, scientific computing with Python would not be practical. See Figure 1-2 for an overview of the various layers of the software stack for computing with Python.

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

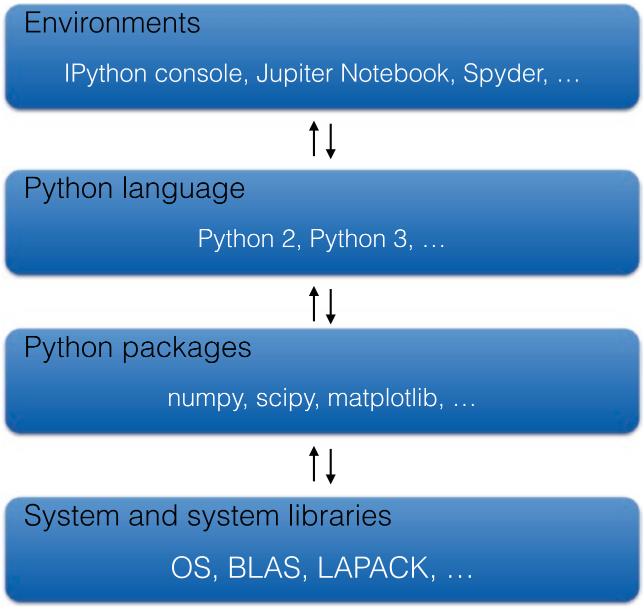

Figure 1-2. *An overview of the components and layers in the scientific computing environment for Python, from a user's perspective from top to bottom. Users typically only interact with the top three layers, but the bottom layer constitutes a very important part of the software stack.*

Tip The SciPy organization and its web site `www.scipy.org` provide a centralized resource for information about the core packages in the scientific Python ecosystem, and lists of additional specialized packages, as well as documentation and tutorials. As such, 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`.

Apart from the technical reasons for why Python provides a good environment for computational work, it is also significant that Python 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 on 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 on 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 having to pay 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 of the 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 the development of 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

There are a number of different environments that 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 use-cases, but on the other hand, it can also be confusing and distracting 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 three environments discussed here are

- The Python interpreter or the IPython console to run code interactively. Together with a text editor for writing code, this provides a lightweight development environment.
- The Jupyter Notebook, which 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 one 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, which can be used to write and interactively run Python code. An IDE such as Spyder is a great tool for developing libraries and reusable Python modules.

All of these environments have justified use-cases, and it is largely a matter of personal preference which one to use. However, I do in particular 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 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 a large number of ways, and in general the installation details also vary from system to system. In Appendix 1, we go through one popular cross-platform method to install the tools and libraries that are 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, there are two active versions of Python available for production use: Python 2 and Python 3. In this book we will work with Python 3, which by now has practically superseded Python 2. However, for some legacy applications, using Python 2 may still be the only option, if it contains libraries that have not been made compatible with Python 3. It is also sometimes the case that only Python 2 is the available in institutionally provided environments, such as on high-performance clusters or universities' computer systems. When developing Python code for such environments, it might be necessary to use Python 2, but otherwise, I strongly recommend using Python 3 in new projects. It should also be noted that support for Python 2 has now been dropped by many major

[2] The Python language and the default Python interpreter are managed and maintained by the Python Software Foundation: `http://www.python.org`.

Python libraries, and the vast majority of computing-oriented libraries for Python now support Python 3. For the purpose of this book, we require version 2.7 or greater for the Python 2 series or Python 3.2 or greater for the preferred Python 3 series.

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.

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

Here the file `hello.py` contains the single line:

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

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

```
$ python --version
Python 3.6.5
```

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 the commands such as, for example, `python2.7` and `python3.6`. It is also possible to set up *virtual* python environments that are independent of the system-provided environments. This has many advantages and I strongly recommend to become familiar with this way of working with Python. Appendix A provides details of how to set up and work with these kinds of environments.

In addition to executing Python script files, a Python interpreter can also 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 argument) launches the Python interpreter in an interactive mode. When doing so, you are presented with a prompt:

```
$ python
Python 3.6.1 |Continuum Analytics, Inc.| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] 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 (not available by default in Python 2).

IPython Console

Although the interactive command-line interface provided by the standard Python interpreter has been greatly improved in recent versions of Python 3, it is still in certain aspects rudimentary, and it does not by itself provide a satisfactory environment for interactive computing. IPython[3] is an enhanced command-line REPL environment for Python, with additional features for interactive and exploratory computing. For example, IPython provides improved command history browsing (also between sessions), an input and output caching system, improved autocompletion, more verbose and helpful exception tracebacks, and much more. In fact, IPython is now much more than an enhanced Python command-line interface, which we will explore in more detail later in this chapter and throughout the book. For instance, under the hood IPython is a

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

client-server application, which separates the frontend (user interface) from the backend (kernel) that 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 powerful framework for parallel computing.

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

```
$ ipython
Python 3.6.1 |Continuum Analytics, Inc.| (default, May 11 2017, 13:04:09)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Caution Note that each IPython installation corresponds to a specific version of Python, and if you have several versions of Python available on your system, you may also have several versions of IPython as well. On many systems, IPython for Python 2 is invoked with the command `ipython2` and for Python 3 with `ipython3`, although the exact setup varies from system to system. Note that here the "2" and "3" refer to the Python version, which is different from the version of IPython itself (which at the time of writing is 6.4.0).

In the following sections, I give a brief overview of 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 are 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. Both the input and the output of previous cells can later be accessed through