Community Experience Distilled

Learning IPython for Interactive Computing and Data Visualization

Second Edition

Get started with Python for data analysis and numerical computing in the Jupyter notebook

Cyrille Rossant

www.allitebooks.com

Learning IPython for Interactive Computing and Data Visualization

Second Edition

Get started with Python for data analysis and numerical computing in the Jupyter notebook

Cyrille Rossant

BIRMINGHAM - MUMBAI

www.allitebooks.com

Learning IPython for Interactive Computing and Data Visualization

Second Edition

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: April 2013

Second edition: October 2015

Production reference: 1151015

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-698-9

www.packtpub.com

www.allitebooks.com

Credits

Author
Cyrille Rossant

Reviewers
Damián Avila
Nicola Rainiero
G Scott Stukey

Commissioning Editor
Kartikey Pandey

Acquisition Editors
Kartikey Pandey
Richard Brookes-Bland

Content Development Editor
Arun Nadar

Technical Editor
Pranil Pathare

Copy Editor
Stephen Copestake

Project Coordinator
Shweta H Birwatkar

Proofreader
Safis Editing

Indexer
Monica Ajmera Mehta

Production Coordinator
Conidon Miranda

Cover Work
Conidon Miranda

www.allitebooks.com

About the Author

Cyrille Rossant is a researcher in neuroinformatics, and is a graduate of Ecole Normale Superieure, Paris, where he studied mathematics and computer science. He has worked at Princeton University, University College London, and College de France. As part of his data science and software engineering projects, he gained experience in machine learning, high-performance computing, parallel computing, and big data visualization.

He is one of the main developers of VisPy, a high-performance visualization package in Python. He is the author of the *IPython Interactive Computing and Visualization Cookbook*, *Packt Publishing*, an advanced-level guide to data science and numerical computing with Python, and the sequel of this book.

I am grateful to Nick Fiorentini for his help during the revision of the book. I would also like to thank my family and notably my wife Claire for their support.

www.allitebooks.com

About the Reviewers

Damián Avila is a software developer and data scientist (formerly a biochemist) from Córdoba, Argentina.

His main focus of interest is data science, visualization, finance, and IPython/Jupyter-related projects.

In the open source area, he is a core developer for several interesting and popular projects, such as IPython/Jupyter, Bokeh, and Nikola. He has also started his own projects, being RISE, an extension to enable amazing live slides in the Jupyter notebook, the most popular one. He has also written several tutorials about the Scientific Python tools (available at Github) and presented several talks at international conferences.

Currently, he is working at Continuum Analytics.

Nicola Rainiero is a civil geotechnical engineer with a background in the construction industry as a self-employed designer engineer. He is also specialized in the renewable energy field and has collaborated with the Sant'Anna University of Pisa for two European projects, REGEOCITIES and PRISCA, using qualitative and quantitative data analysis techniques.

He has an ambition to simplify his work with open software and use and develop new ones; sometimes obtaining good results, at other times, negative. You can reach Nicola on his website at `http://rainnic.altervista.org`.

A special thanks to Packt Publishing for this opportunity to participate in the reviewing of this book. I thank my family, especially my parents, for their physical and moral support.

www.allitebooks.com

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

www.allitebooks.com

Table of Contents

Preface **vii**

Chapter 1: Getting Started with IPython **1**

What are Python, IPython, and Jupyter? **1**
Jupyter and IPython 2
What this book covers 4
References 5
Installing Python with Anaconda **5**
Downloading Anaconda 6
Installing Anaconda 6
Before you get started... 7
Opening a terminal 7
Finding your home directory 8
Manipulating your system path 8
Testing your installation 9
Managing environments 9
Common conda commands 10
References 11
Downloading the notebooks 12
Introducing the Notebook **13**
Launching the IPython console 13
Launching the Jupyter Notebook 14
The Notebook dashboard 15
The Notebook user interface 16
Structure of a notebook cell 16
Markdown cells 17
Code cells 18

www.allitebooks.com

The Notebook modal interface 19
Keyboard shortcuts available in both modes 19
Keyboard shortcuts available in the edit mode 19
Keyboard shortcuts available in the command mode 20
References 20
A crash course on Python 20
Hello world 21
Variables 21
String escaping 23
Lists 24
Loops 26
Indentation 27
Conditional branches 27
Functions 28
Positional and keyword arguments 29
Passage by assignment 30
Errors 31
Object-oriented programming 32
Functional programming 34
Python 2 and 3 35
Going beyond the basics 36
Ten Jupyter/IPython essentials 37
Using IPython as an extended shell 37
Learning magic commands 42
Mastering tab completion 45
Writing interactive documents in the Notebook with Markdown 47
Creating interactive widgets in the Notebook 49
Running Python scripts from IPython 51
Introspecting Python objects 53
Debugging Python code 54
Benchmarking Python code 55
Profiling Python code 56
Summary 58
Chapter 2: Interactive Data Analysis with pandas 59
Exploring a dataset in the Notebook 59
Provenance of the data 60
Downloading and loading a dataset 61
Making plots with matplotlib 63
Descriptive statistics with pandas and seaborn 67

www.allitebooks.com

Manipulating data **69**
Selecting data 69
Selecting columns 70
Selecting rows 70
Filtering with boolean indexing 72
Computing with numbers 73
Working with text 75
Working with dates and times 76
Handling missing data 77
Complex operations **78**
Group-by 78
Joins 80
Summary **83**
Chapter 3: Numerical Computing with NumPy **85**
A primer to vector computing **85**
Multidimensional arrays 86
The ndarray 86
Vector operations on ndarrays 87
How fast are vector computations in NumPy? 88
How an ndarray is stored in memory 89
Why operations on ndarrays are fast 91
Creating and loading arrays **91**
Creating arrays 91
Loading arrays from files 93
Basic array manipulations **94**
Computing with NumPy arrays **97**
Selection and indexing 98
Boolean operations on arrays 99
Mathematical operations on arrays 100
A density map with NumPy 103
Other topics 107
Summary **108**
Chapter 4: Interactive Plotting and Graphical Interfaces **109**
Choosing a plotting backend **109**
Inline plots 109
Exported figures 111
GUI toolkits 111
Dynamic inline plots 113
Web-based visualization 114

www.allitebooks.com

matplotlib and seaborn essentials **115**
Common plots with matplotlib 116
Customizing matplotlib figures 120
Interacting with matplotlib figures in the Notebook 122
High-level plotting with seaborn 124
Image processing **126**
Further plotting and visualization libraries **129**
High-level plotting 129
Bokeh 130
Vincent and Vega 130
Plotly 131
Maps and geometry 132
The matplotlib Basemap toolkit 132
GeoPandas 133
Leaflet wrappers: folium and mplleaflet 134
3D visualization 134
Mayavi 134
VisPy 135
Summary **135**
Chapter 5: High-Performance and Parallel Computing 137
Accelerating Python code with Numba **138**
Random walk 138
Universal functions 141
Writing C in Python with Cython **143**
Installing Cython and a C compiler for Python 143
Implementing the Eratosthenes Sieve in Python and Cython 144
Distributing tasks on several cores with IPython.parallel **148**
Direct interface 149
Load-balanced interface 150
Further high-performance computing techniques **153**
MPI 153
Distributed computing 153
C/C++ with Python 154
GPU computing 154
PyPy 155
Julia 155
Summary **155**

Chapter 6: Customizing IPython 157
Creating a custom magic command in an IPython extension 157
Writing a new Jupyter kernel 160
Displaying rich HTML elements in the Notebook 165
Displaying SVG in the Notebook 165
JavaScript and D3 in the Notebook 167
Customizing the Notebook interface with JavaScript 170
Summary 172
Index 173

Preface

Data analysis skills are now essential in scientific research, engineering, finance, economics, journalism, and many other domains. With its high accessibility and vibrant ecosystem, Python is one of the most appreciated open source languages for data science.

This book is a beginner-friendly introduction to the Python data analysis platform, focusing on IPython (Interactive Python) and its Notebook. While IPython is an enhanced interactive Python terminal specifically designed for scientific computing and data analysis, the Notebook is a graphical interface that combines code, text, equations, and plots in a unified interactive environment.

The first edition of *Learning IPython for Interactive Computing and Data Visualization* was published in April 2013, several months before the release of IPython 1.0. This new edition targets IPython 4.0, released in August 2015. In addition to reflecting the novelties of this new version of IPython, the present book is also more accessible to non-programmer beginners. The first chapter contains a brand new crash course on Python programming, as well as detailed installation instructions.

Since the first edition of this book, IPython's popularity has grown significantly, with an estimated user base of several millions of people and ongoing collaborations with large companies like Microsoft, Google, IBM, and others. The project itself has been subject to important changes, with a refactoring into a language-independent interface called the Jupyter Notebook, and a set of backend kernels in various languages. The Notebook is no longer reserved to Python; it can now also be used with R, Julia, Ruby, Haskell, and many more languages (50 at the time of this writing!).

The Jupyter project has received significant funding in 2015 from the Leona M. and Harry B. Helmsley Charitable Trust, the Gordon and Betty Moore Foundation, and the Alfred P. Sloan Foundation, which will allow the developers to focus on the growth and maturity of the project in the years to come.

Here are a few references:

- Home page for the Jupyter project at `http://jupyter.org/`
- Announcement of the funding for Jupyter at `https://blog.jupyter.org/2015/07/07/jupyter-funding-2015/`
- Detail of the project's grant at `https://blog.jupyter.org/2015/07/07/project-jupyter-computational-narratives-as-the-engine-of-collaborative-data-science/`

What this book covers

Chapter 1, *Getting Started with IPython*, is a thorough and beginner-friendly introduction to Anaconda (a popular Python distribution), the Python language, the Jupyter Notebook, and IPython.

Chapter 2, *Interactive Data Analysis with pandas*, is a hands-on introduction to interactive data analysis and visualization in the Notebook with pandas, matplotlib, and seaborn.

Chapter 3, *Numerical Computing with NumPy*, details how to use NumPy for efficient computing on multidimensional numerical arrays.

Chapter 4, *Interactive Plotting and Graphical Interfaces*, explores many capabilities of Python for interactive plotting, graphics, image processing, and interactive graphical interfaces in the Jupyter Notebook.

Chapter 5, *High-Performance and Parallel Computing*, introduces the various techniques you can employ to accelerate your numerical computing code, namely parallel computing and compilation of Python code.

Chapter 6, *Customizing IPython*, shows how IPython and the Jupyter Notebook can be extended for customized use-cases.

What you need for this book

The following software is required for the book:

- Anaconda with Python 3
- Windows, Linux, or OS X can be used as a platform

Who this book is for

This book targets anyone who wants to analyze data or perform numerical simulations of mathematical models.

Since our world is becoming more and more data-driven, knowing how to analyze data effectively is an essential skill to learn. If you're used to spreadsheet programs like Microsoft Excel, you will appreciate Python for its much larger range of analysis and visualization possibilities. Knowing this general-purpose language will also let you share your data and analysis with other programs and libraries.

In conclusion, this book will be useful to students, scientists, engineers, analysts, journalists, statisticians, economists, hobbyists, and all data enthusiasts.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Run it with a command like `bash Anaconda3-2.3.0-Linux-x86_64.sh` (if necessary, replace the filename by the one you downloaded)."

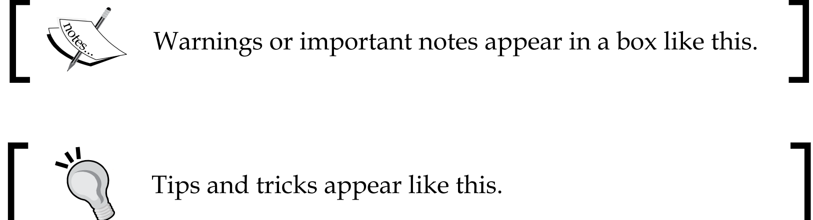
A block of code is set as follows:

```
def load_ipython_extension(ipython):
    """This function is called when the extension is loaded.
    It accepts an IPython InteractiveShell instance.
    We can register the magic with the `register_magic_function`
    method of the shell instance."""
    ipython.register_magic_function(cpp, 'cell')
```

Any command-line input or output is written as follows:

```
$ python
Python 3.4.3 |Anaconda 2.3.0 (64-bit)| (default, Jun  4 2015, 15:29:08)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "To create a new notebook, click on the **New** button, and select **Notebook (Python 3)**."



Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`. You can also report any issues at `https://github.com/ipython-books/minibook-2nd-code/issues`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you. You will also find the book's code on this GitHub repository: `https://github.com/ipython-books/minibook-2nd-code`.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/6989OS_ColouredImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Getting Started with IPython

In this chapter, we will cover the following topics:

- What are Python, IPython, and Jupyter?
- Installing Python with Anaconda
- Introducing the Notebook
- A crash course on Python
- Ten Jupyter/IPython essentials

What are Python, IPython, and Jupyter?

Python is an open source general-purpose language created by Guido van Rossum in the late 1980s. It is widely-used by system administrators and developers for many purposes: for example, automating routine tasks or creating a web server. Python is a flexible and powerful language, yet it is sufficiently simple to be taught to school children with great success.

In the past few years, Python has also emerged as one of the leading open platforms for data science and high-performance numerical computing. This might seem surprising as Python was not originally designed for scientific computing. Python's interpreted nature makes it much slower than lower-level languages like C or Fortran, which are more amenable to number crunching and the efficient implementation of complex mathematical algorithms.

However, the performance of these low-level languages comes at a cost: they are hard to use and they require advanced knowledge of how computers work. In the late 1990s, several scientists began investigating the possibility of using Python for numerical computing by interoperating it with mainstream C/Fortran scientific libraries. This would bring together the ease-of-use of Python with the performance of C/Fortran: the dream of any scientist!

Consequently, the past 15 years have seen the development of widely-used libraries such as NumPy (providing a practical array data structure), SciPy (scientific computing), matplotlib (graphical plotting), pandas (data analysis and statistics), scikit-learn (machine learning), SymPy (symbolic computing), and Jupyter/IPython (efficient interfaces for interactive computing). Python, along with this set of libraries, is sometimes referred to as the SciPy stack or PyData platform.

Competing platforms

Python has several competitors. For example, MATLAB (by Mathworks) is a commercial software focusing on numerical computing that is widely-used in scientific research and engineering. SPSS (by IBM) is a commercial software for statistical analysis. Python, however, is free and open source, and that's one of its greatest strengths. Alternative open source platforms include R (specialized in statistics) and Julia (a young language for high-performance numerical computing).

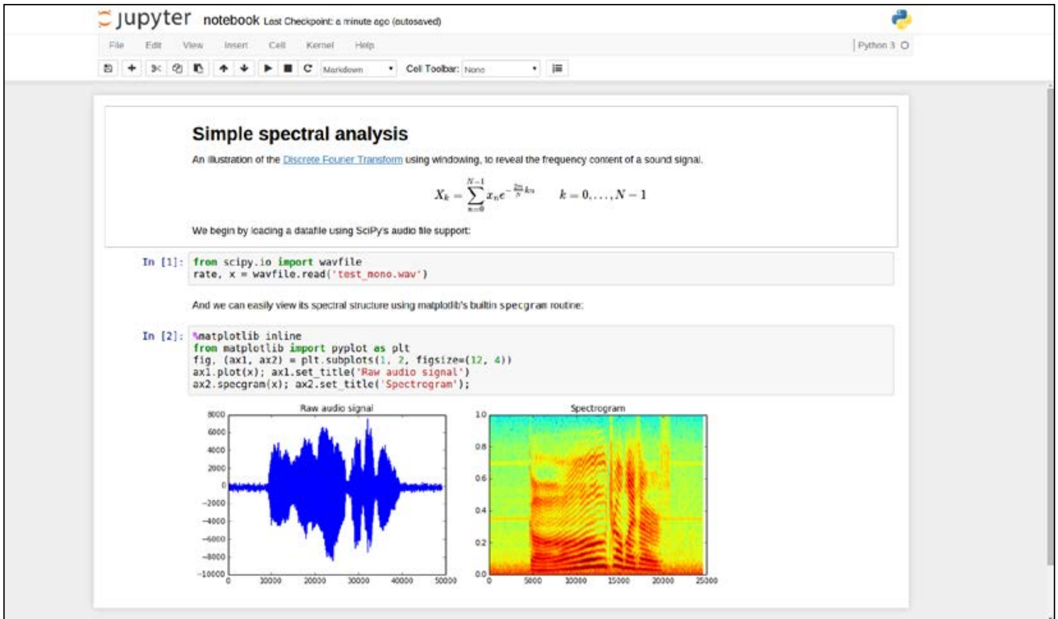
More recently, this platform has gained popularity in other non-academic communities such as finance, engineering, statistics, data science, and others.

This book provides a solid introduction to the whole platform by focusing on one of its main components: Jupyter/IPython.

Jupyter and IPython

IPython was created in 2001 by Fernando Perez (the *I* in *IPython* stands for "interactive"). It was originally meant to be a convenient command-line interface to the scientific Python platform. In scientific computing, trial and error is the rule rather than the exception, and this requires an efficient interface that allows for *interactive* exploration of algorithms, data, and graphs.

In 2011, IPython introduced the interactive **Notebook**. Inspired by commercial software such as Maple (by Maplesoft) or Mathematica (by Wolfram Research), the Notebook runs in a browser and provides a unified web interface where code, text, mathematical equations, plots, graphics, and interactive graphical controls can be combined into a single document. This is an ideal interface for scientific computing. Here is a screenshot of a notebook:



Example of a notebook

It quickly became clear that this interface could be used with languages other than Python such as R, Julia, Lua, Ruby, and many others. Further, the Notebook is not restricted to scientific computing: it can be used for academic courses, software documentation, or book writing thanks to conversion tools targeting Markdown, HTML, PDF, ODT, and many other formats. Therefore, the IPython developers decided in 2014 to acknowledge the general-purpose nature of the Notebook by giving a new name to the project: **Jupyter**.

Jupyter features a language-independent Notebook platform that can work with a variety of **kernels**. Implemented in any language, a kernel is the *backend* of the Notebook interface. It manages the interactive session, the variables, the data, and so on. By contrast, the Notebook interface is the *frontend* of the system. It manages the user interface, the text editor, the plots, and so on. **IPython is henceforth the name of the Python kernel for the Jupyter Notebook.** Other kernels include IR, IJulia, ILua, IRuby, and many others (50 at the time of this writing).

In August 2015, the IPython/Jupyter developers achieved the "Big Split" by splitting the previous monolithic IPython codebase into a set of smaller projects, including the language-independent Jupyter Notebook (see `https://blog.jupyter.org/2015/08/12/first-release-of-jupyter/`). For example, the parallel computing features of IPython are now implemented in a standalone Python package named `ipyparallel`, the IPython widgets are implemented in `ipywidgets`, and so on. This separation makes the code of the project more modular and facilitates third-party contributions. IPython itself is now a much smaller project than before since it only features the interactive Python terminal and the Python kernel for the Jupyter Notebook.

You will find the list of changes in IPython 4.0 at `http://ipython.readthedocs.org/en/latest/whatsnew/version4.html`. Many internal IPython imports have been deprecated due to the code reorganization. Warnings are raised if you attempt to perform a deprecated import. Also, the **profiles** have been removed and replaced with a unique default profile. However, you can simulate this functionality with environment variables. You will find more information at `http://jupyter.readthedocs.org`.

What this book covers

This book covers the **Jupyter Notebook 1.0** and focuses on its Python kernel, **IPython 4.0**. In this chapter, we will introduce the platform, the Python language, the Jupyter Notebook interface, and IPython. In the remaining chapters, we will cover data analysis and scientific computing in Jupyter/IPython with the help of mainstream scientific libraries such as NumPy, pandas, and matplotlib.

This book gives you a solid introduction to Jupyter and the SciPy platform. The *IPython Interactive Computing and Visualization Cookbook* (`http://ipython-books.github.io/cookbook/`) is the sequel of this introductory-level book. In 15 chapters and more than 500 pages, it contains a hundred recipes covering a wide range of interactive numerical computing techniques and data science topics. The *IPython Cookbook* is an excellent addition to the present IPython minibook if you're interested in delving into the platform in much greater detail.

References

Here are a few references about IPython and the Notebook:

- The main Jupyter page at: `http://jupyter.org/`
- The main Jupyter documentation at: `https://jupyter.readthedocs.org/en/latest/`
- The main IPython page at: `http://ipython.org/`
- Jupyter on GitHub at: `https://github.com/jupyter`
- Try Jupyter online at: `https://try.jupyter.org/`
- The IPython Notebook in research, a Nature note at `http://www.nature.com/news/interactive-notebooks-sharing-the-code-1.16261`

Installing Python with Anaconda

Although Python is an open-source, cross-platform language, installing it with the usual scientific packages used to be overly complicated. Fortunately, there is now an all-in-one scientific Python distribution, **Anaconda** (by Continuum Analytics), that is free, cross-platform, and easy to install. Anaconda comes with Jupyter and all of the scientific packages we will use in this book. There are other distributions and installation options (like Canopy, WinPython, Python(x, y), and others), but for the purpose of this book we will use Anaconda throughout.

Running Jupyter in the cloud

You can also use Jupyter directly from your web browser, without installing anything on your local computer: go to `http://try.jupyter.org`. Note that the notebooks created there are not saved. Let's also mention a similar service, Wakari (`https://wakari.io`), by Continuum Analytics.

Anaconda comes with a package manager named **conda**, which lets you manage your Python distribution and install new packages.

Miniconda

Miniconda (`http://conda.pydata.org/miniconda.html`) is a light version of Anaconda that gives you the ability to only install the packages you need.

Downloading Anaconda

The first step is to download Anaconda from Continuum Analytics' website (`http://continuum.io/downloads`). This is actually not the easiest part since several versions are available. Three properties define a particular version:

- The **operating system** (**OS**): Linux, Mac OS X, or Windows. This will depend on the computer you want to install Python on.
- **32-bit or 64-bit**: *You want the 64-bit version*, unless you're on an old or low-end computer. The 64-bit version will allow you to manipulate large datasets.
- The **version of Python**: 2.7, or 3.4 (or later). In this book, *we will use Python 3.4*. You can also use Python 3.5 (released in September 2015) which introduces many features, including a new `@` operator for matrix multiplication. However, it is easy to temporarily switch to a Python 2.7 environment with Anaconda if necessary (see the next section).

> Python 3 brought a few backward-incompatible changes over Python 2 (also known as Legacy Python). This is why many people are still using Python 2.7 at this time, even though Python 3 was released in 2008. **We will use Python 3 in this book, and we recommend that newcomers learn Python 3**. If you need to use legacy Python code that hasn't yet been updated to Python 3, you can use conda to temporarily switch to a Python 2 interpreter.

Once you have found the right link for your OS and Python 3 64-bit, you can download the package. You should then find it in your `downloads` directory (depending on your OS and your browser's settings).

Installing Anaconda

The Anaconda installer comes in different flavors depending on your OS, as follows:

- **Linux**: The Linux installer is a bash `.sh` script. Run it with a command like `bash Anaconda3-2.3.0-Linux-x86_64.sh` (if necessary, replace the filename by the one you downloaded).
- **Mac**: The Mac graphical installer is a `.pkg` file that you can run with a double-click.
- **Windows**: The Windows graphical installer is an `.exe` file that you can run with a double-click.

Then, follow the instructions to install Anaconda on your computer. Here are a few remarks:

- You don't need administrator rights to install Anaconda. In most cases, you can choose to install it in your personal user account.
- Choose to put Anaconda in your system path, so that Anaconda's Python is the system default.

Anaconda comes with a graphical launcher that you can use to start IPython, manage environments, and so on. You will find more details at `http://docs.continuum.io/anaconda-launcher/`

Before you get started...

Before you get started with Anaconda, there are a few things you need to know:

- Opening a terminal
- Finding your home directory
- Manipulating your system path

You can skip this section if you already know how to do these things.

Opening a terminal

A **terminal** is a command-line application that lets you interact with your computer by typing commands with the keyboard, instead of clicking on windows with the mouse. While most computer users only know Graphical User Interfaces, developers and scientists generally need to know how to use the command-line interface for advanced usage. To use the command-line interface, follow the instructions that are specific to your OS:

- On Windows, you can use **Powershell**. Press the Windows + *R* keys, type `powershell` in the Run box, and press *Enter*. You will find more information about Powershell at `https://blog.udemy.com/powershell-tutorial/`. Alternatively, you can use the older Windows terminal by typing `cmd` in the Run box.
- On OS X, you can open the Terminal application, for example by pressing Cmd + Space, typing `terminal`, and pressing *Enter*.
- On Linux, you can open the Terminal from your application manager.

In a terminal, use the `cd /path/to/directory` command to move to a given directory. For example, `cd ~` moves to your home directory, which is introduced in the next section.

Finding your home directory

Your **home directory** is specific to your user account on your computer. It generally contains your applications' settings. It is often referred to as ~.Depending on the OS, the location of the home directory is as follows:

- On Windows, its location is `C:\Users\YourName\` where `YourName` is the name of your account.
- On OS X, its location is `/Users/YourName/` where `YourName` is the name of your account.
- On Linux, its location is generally `/home/yourname/` where `yourname` is the name of your account.

For example, the directory `~/anaconda3` refers to `C:\Users\YourName\anaconda3\` on Windows and `/home/yourname/anaconda3/` on Linux.

Manipulating your system path

The **system path** is a global variable (also called an **environment variable**) defined by your operating system with the list of directories where executable programs are located. If you type a command like `python` in your terminal, you generally need to have a `python` (or `python.exe` on Windows) executable in one of the directories listed in the system path. If that's not the case, an error may be raised.

You can manually add directories to your system path as follows:

- On Windows, press the Windows + *R* keys, type `rundll32.exe sysdm.cpl,EditEnvironmentVariables`, and press *Enter*. You can then edit the PATH variable and append `;C:\path\to\directory` if you want to add that directory. You will find more detailed instructions at `http://www.computerhope.com/issues/ch000549.htm`.
- On OS X, edit or create the file `~/.bash_profile` and add `export PATH="$PATH:/path/to/directory"` at the end of the file.
- On Linux, edit or create the file `~/.bashrc` and add `export PATH="$PATH:/path/to/directory"` at the end of the file.

Testing your installation

To test Anaconda once it has been installed, open a terminal and type `python`. This opens a **Python console**, not to be confused with the **OS terminal**. The Python console is identified with a `>>>` prompt string, whereas the OS terminal is identified with a `$` (Linux/OS X) or `>` (Windows) prompt string. These strings are displayed in the terminal, often preceded by your computer's name, your login, and the current directory (for example, `yourname@computer:~$` on Linux or `PS C:\Users\YourName>` on Windows). You can type commands after the prompt string. After typing `python`, you should see something like the following:

```
$ python
Python 3.4.3 |Anaconda 2.3.0 (64-bit)| (default, Jun  4 2015, 15:29:08)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

What matters is that `Anaconda` or `Continuum Analytics` is mentioned here. Otherwise, typing `python` might have launched your system's default Python, which is *not* the one you want to use in this book.

If you have this problem, you may need to add the path to the Anaconda executables to your system path. For example, this path will be `~/anaconda3/bin` if you chose to install Anaconda in `~/anaconda3`. The `bin` directory contains Anaconda executables including python.

If you have any problem installing and testing Anaconda, you can ask for help on the mailing list (see the link in the *References* section under the *Installing Python with Anaconda* section of this chapter).

Next, exit the Python prompt by typing `exit()` and pressing *Enter*.

Managing environments

Anaconda lets you create different isolated Python environments. For example, you can have a Python 2 distribution for the rare cases where you need to temporarily switch to Python 2.

To create a new environment for Python 2, type the following command in an OS terminal:

```
$ conda create -n py2 anaconda python=2.7
```

This will create a new isolated environment named `py2` based on the original Anaconda distribution, but with Python 2.7. You could also use the command `conda env`: type `conda env -h` to see the details.

You can now activate your `py2` environment by typing the following command in a terminal:

- **Windows**: `activate py2` (note that you might have problems with Powershell, see `https://github.com/conda/conda/issues/626`, or use the old `cmd` terminal)
- **Linux and Mac OS X**: `source activate py2`

Now, you should see a (py2) prefix in front of your terminal prompt. Typing `python` in your terminal with the `py2` environment activated will open a Python 2 interpreter.

Type `deactivate` on Windows or `source deactivate` on Linux/OS X to deactivate the environment in the terminal.

Common conda commands

Here is a list of common commands:

- `conda help`: Displays the list of conda commands.
- `conda list`: Lists all packages installed in the current environment.
- `conda info`: Displays system information.
- `conda env list`: Displays the list of environments installed. The currently active one is marked by a star `*`.
- `conda install somepackage`: Installs a Python package (replace `somepackage` by the name of the package you want to install).
- `conda install somepackage=0.7`: Installs a specific version of a package.
- `conda update somepackage`: Updates a Python package to the latest available version.
- `conda update anaconda`: Updates all packages.
- `conda update conda`: Updates conda itself.

- `conda update --all`: Updates all packages.
- `conda remove somepackage`: Uninstalls a Python package.
- `conda remove -n myenv --all`: Removes the environment named `myenv` (replace this by the name of the environment you want to uninstall).
- `conda clean -t`: Removes the old tarballs that are left over after installation and updates.

Some commands ask for confirmation (you need to press `y` to confirm). You can also use the `-y` option to avoid the confirmation prompt.

If `conda install somepackage` fails, you can try `pip install somepackage` instead. This will use the **Python Package Index** (**PyPI**) instead of Anaconda. Many scientific Anaconda packages are easier to install than the corresponding PyPI packages because they are precompiled for your platform. However, many packages are available on PyPI but not on Anaconda.

Here are some references:

- pip documentation at `https://pip.pypa.io/en/stable/`
- PyPI repository at `https://pypi.python.org/pypi`

References

Here are a few references about Anaconda:

- Continuum Analytics' website: `http://continuum.io/`
- Anaconda main page: `https://store.continuum.io/cshop/anaconda/`
- Anaconda downloads: `http://continuum.io/downloads`
- List of Anaconda packages: `http://docs.continuum.io/anaconda/pkg-docs`
- Conda main page: `http://conda.io/`
- Anaconda mailing list: `https://groups.google.com/a/continuum.io/forum/#!forum/anaconda`
- Continuum Analytics Twitter account at `https://twitter.com/ContinuumIO`
- Conda FAQ: `http://conda.pydata.org/docs/faq.html`
- Curated list of Python packages at `http://awesome-python.com/`

www.allitebooks.com

Downloading the notebooks

All of this book's code is available on GitHub as notebooks. We recommend that you download the notebooks and experiment with them as you're working through the book.

GitHub is a popular online service that hosts open source projects. It is based on the **Git Distributed Version Control System** (**DVCS**). Git keeps track of file changes and enables collaborative work on a given project. Learning a version control system like Git is highly recommended for all programmers. Not using a version control system when working with code or even text documents is now considered as bad practice. You will find several references at `https://help.github.com/articles/good-resources-for-learning-git-and-github/`. The *IPython Cookbook* also contains several recipes about Git and best interactive programming practices.

Here is how to download the book's notebooks:

- **Install git**: `http://git-scm.com/downloads`.
- **Check your git installation**: Open a new OS terminal and type `git version`. You should see the version of git and not an error message.
- **Type the following command** (**this is a single line**):

```
$ git clone https://github.com/ipython-books/
  minibook-2nd-code.git  "$HOME/minibook"
```

This will download the very latest version of the code into a `minibook` subdirectory in your home directory. You can also choose another directory.

From this directory, you can update to the latest version at any time by typing `git pull`.

Notebooks on GitHub

Notebook documents stored on GitHub (with the file extension `.ipynb`) are automatically rendered on the GitHub website.

Introducing the Notebook

Originally, IPython provided an enhanced command-line console to run Python code interactively. The Jupyter Notebook is a more recent and more sophisticated alternative to the console. Today, both tools are available, and we recommend that you learn to use both.

Launching the IPython console

To run the IPython console, type `ipython` in an OS terminal. There, you can write Python commands and see the results instantly. Here is a screenshot:

```
cyrille@gigabyte:~$ ipython
Python 3.4.3 |Anaconda 2.3.0 (64-bit)| (default, Jun  4 2015, 15:29:08)
Type "copyright", "credits" or "license" for more information.

IPython 3.2.0 -- An enhanced Interactive Python.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: print("Hello world!")
Hello world!

In [2]: 2 * 3
Out[2]: 6

In [3]:
```

IPython console

The IPython console is most convenient when you have a command-line-based workflow and you want to execute some quick Python commands.

You can exit the IPython console by typing `exit`.

Let's mention the **Qt console**, which is similar to the IPython console but offers additional features such as multiline editing, enhanced tab completion, image support, and so on. The Qt console can also be integrated within a graphical application written with Python and Qt. See `http://jupyter.org/qtconsole/stable/` for more information.

Launching the Jupyter Notebook

To run the Jupyter Notebook, open an OS terminal, go to `~/minibook/` (or into the directory where you've downloaded the book's notebooks), and type `jupyter notebook`. This will start the Jupyter server and open a new window in your browser (if that's not the case, go to the following URL: `http://localhost:8888`). Here is a screenshot of Jupyter's entry point, the **Notebook dashboard**:

The Notebook dashboard

At the time of writing, the following browsers are officially supported: Chrome 13 and greater; Safari 5 and greater; and Firefox 6 or greater. Other browsers may work also. Your mileage may vary.

The Notebook is most convenient when you start a complex analysis project that will involve a substantial amount of interactive experimentation with your code. Other common use-cases include keeping track of your interactive session (like a lab notebook), or writing technical documents that involve code, equations, and figures.

In the rest of this section, we will focus on the Notebook interface.

Closing the Notebook server

To close the Notebook server, go to the OS terminal where you launched the server from, and press *Ctrl* + *C*. You may need to confirm with `y`.

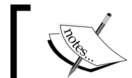
The Notebook dashboard

The dashboard contains several tabs:

- **Files**: shows all files and notebooks in the current directory
- **Running**: shows all kernels currently running on your computer
- **Clusters**: lets you launch kernels for parallel computing (covered in *Chapter 5*, *High-Performance and Parallel Computing*)

A **notebook** is an interactive document containing code, text, and other elements. A notebook is saved in a file with the `.ipynb` extension. This file is a plain text file storing a JSON data structure.

A **kernel** is a process running an interactive session. When using IPython, this kernel is a Python process. There are kernels in many languages other than Python.



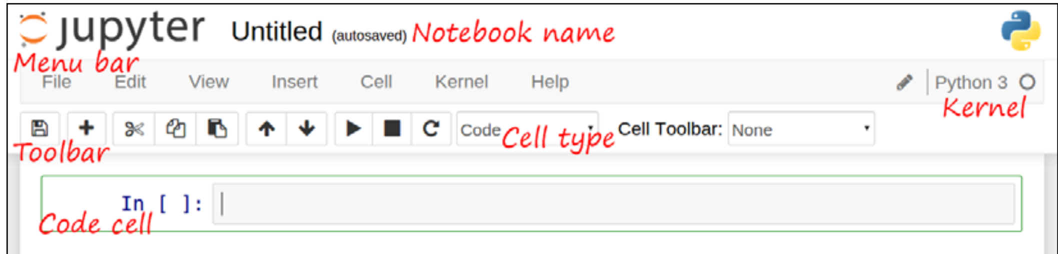
We follow the convention to use the term *notebook* for a file, and *Notebook* for the application and the web interface.

In Jupyter, notebooks and kernels are strongly separated. A notebook is a file, whereas a kernel is a process. The kernel receives snippets of code from the Notebook interface, executes them, and sends the outputs and possible errors back to the Notebook interface. Thus, in general, the kernel has no notion of a Notebook. A notebook is persistent (it's a file), whereas a kernel may be closed at the end of an interactive session and it is therefore not persistent. When a notebook is re-opened, it needs to be re-executed.

In general, no more than one Notebook interface can be connected to a given kernel. However, several IPython consoles can be connected to a given kernel.

The Notebook user interface

To create a new notebook, click on the **New** button, and select **Notebook (Python 3)**. A new browser tab opens and shows the Notebook interface as follows:



A new notebook

Here are the main components of the interface, from top to bottom:

- The **notebook name**, which you can change by clicking on it. This is also the name of the `.ipynb` file.
- The **Menu bar** gives you access to several actions pertaining to either the notebook or the kernel.
- To the right of the menu bar is the **Kernel** name. You can change the kernel language of your notebook from the **Kernel** menu. We will see in *Chapter 6, Customizing IPython* how to manage different kernel languages.
- The **Toolbar** contains icons for common actions. In particular, the dropdown menu showing **Code** lets you change the type of a cell.
- Following is the main component of the UI: the actual Notebook. It consists of a linear list of cells. We will detail the structure of a cell in the following sections.

Structure of a notebook cell

There are two main types of cells: Markdown cells and code cells, and they are described as follows:

- A **Markdown cell** contains rich text. In addition to classic formatting options like bold or italics, we can add links, images, HTML elements, LaTeX mathematical equations, and more. We will cover Markdown in more detail in the *Ten Jupyter/IPython essentials* section of this chapter.

- A **code cell** contains code to be executed by the kernel. The programming language corresponds to the kernel's language. We will only use Python in this book, but you can use many other languages.

You can change the type of a cell by first clicking on a cell to select it, and then choosing the cell's type in the toolbar's dropdown menu showing Markdown or Code.

Markdown cells

Here is a screenshot of a Markdown cell:

````
### New paragraph
This is *rich* **text** with [links](http://ipython.org), equations:
$$\hat{f}(\xi) = \int_{-\infty}^{+\infty} f(x)\, \mathrm{e}^{-i \xi x} dx$$
code with syntax highlighting:
```python
print("Hello world!")
```
and images:
![This is an image](http://jupyter.org/images/jupyter-sq-text.svg)
````

New paragraph

This is *rich* **text** with links, equations:

$$\hat{f}(\xi) = \int_{-\infty}^{+\infty} f(x)\, \mathrm{e}^{-i \xi x} dx$$

code with syntax highlighting:

```python
print("Hello world!")
```

and images:

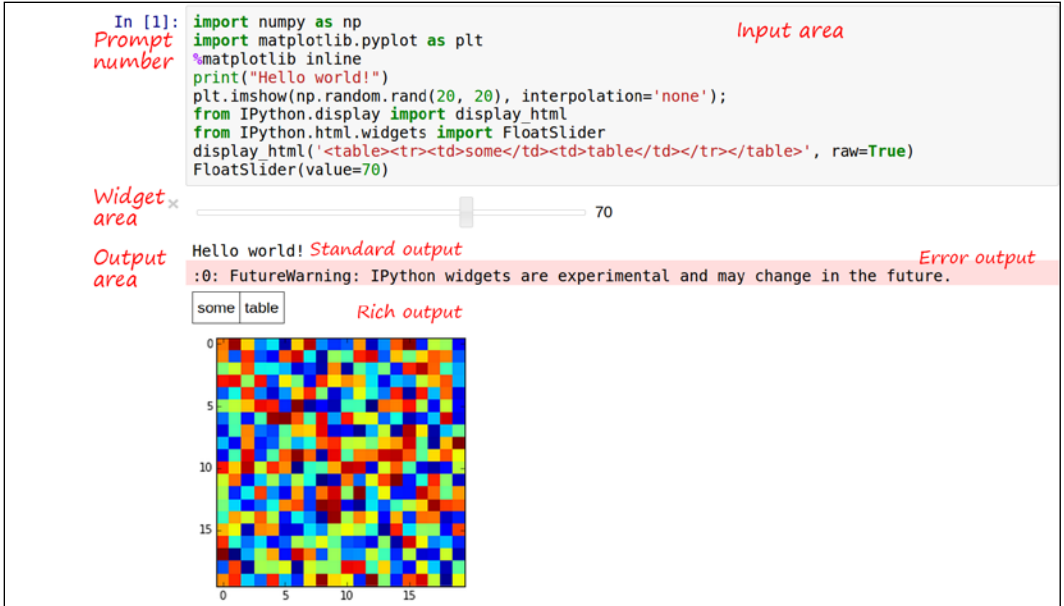
IP[y]: IPython Interactive Computing

A Markdown cell

The top panel shows the cell in edit mode, while the bottom one shows it in render mode. The edit mode lets you edit the text, while the render mode lets you display the rendered cell. We will explain the differences between these modes in greater detail in the following section.

Code cells

Here is a screenshot of a complex code cell:



Structure of a code cell

This code cell contains several parts, as follows:

- The **Prompt number** shows the cell's number. This number increases every time you run the cell. Since you can run cells of a notebook out of order, nothing guarantees that code numbers are linearly increasing in a given notebook.
- The **Input area** contains a multiline text editor that lets you write one or several lines of code with syntax highlighting.
- The **Widget area** may contain graphical controls; here, it displays a slider.
- The **Output area** can contain multiple outputs, here:
    - **Standard output** (text in black)
    - **Error output** (text with a red background)
    - **Rich output** (an HTML table and an image here)

The Notebook modal interface

The Notebook implements a modal interface similar to some text editors such as vim. Mastering this interface may represent a small learning curve for some users.

- Use the **edit mode** to write code (the selected cell has a green border, and a pen icon appears at the top right of the interface). Click inside a cell to enable the edit mode for this cell (you need to double-click with Markdown cells).
- Use the **command mode** to operate on cells (the selected cell has a gray border, and there is no pen icon). Click outside the text area of a cell to enable the command mode (you can also press the *Esc* key).

Keyboard shortcuts are available in the Notebook interface. Type `h` to show them. We review here the most common ones (for Windows and Linux; shortcuts for OS X may be slightly different).

Keyboard shortcuts available in both modes

Here are a few keyboard shortcuts that are always available when a cell is selected:

- *Ctrl* + *Enter*: run the cell
- *Shift* + *Enter*: run the cell and select the cell below
- *Alt* + *Enter*: run the cell and insert a new cell below
- *Ctrl* + *S*: save the notebook

Keyboard shortcuts available in the edit mode

In the edit mode, you can type code as usual, and you have access to the following keyboard shortcuts:

- *Esc*: switch to command mode
- *Ctrl* + *Shift* + -: split the cell

Keyboard shortcuts available in the command mode

In the command mode, keystrokes are bound to cell operations. **Don't write code in command mode** or unexpected things will happen! For example, typing `dd` in command mode will delete the selected cell! Here are some keyboard shortcuts available in command mode:

- *Enter*: switch to edit mode
- ↑ or *k*: select the previous cell
- ↓ or *j*: select the next cell
- *y* / *m*: change the cell type to code cell/Markdown cell
- *a* / *b*: insert a new cell above/below the current cell
- *x* / *c* / *v*: cut/copy/paste the current cell
- *dd*: delete the current cell
- *z*: undo the last delete operation
- *Shift* + =: merge the cell below
- *h*: display the help menu with the list of keyboard shortcuts

Spending some time learning these shortcuts is highly recommended.

References

Here are a few references:

- Main documentation of Jupyter at `http://jupyter.readthedocs.org/en/latest/`
- Jupyter Notebook interface explained at `http://jupyter-notebook.readthedocs.org/en/latest/notebook.html`

A crash course on Python

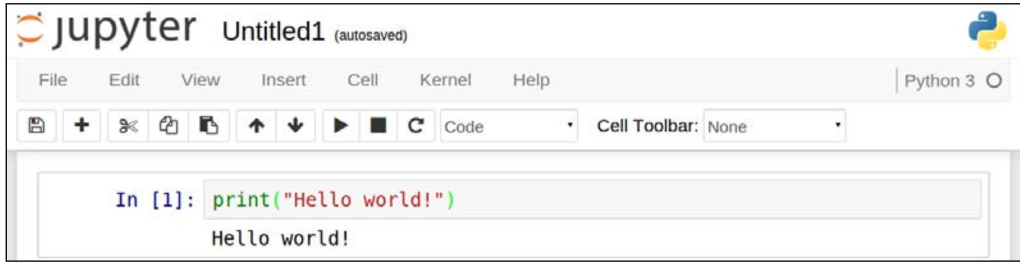
If you don't know Python, read this section to learn the fundamentals. Python is a very accessible language and, if you have ever programmed, it will only take you a few minutes to learn the basics.

Hello world

Open a new notebook and type the following in the first cell:

```
In [1]: print("Hello world!")
Out[1]: Hello world!
```

Here is a screenshot:



"Hello world" in the Notebook

Prompt string

Note that the convention chosen in this book is to show Python code (also called the input) prefixed with `In [x]:` (which shouldn't be typed). This is the standard IPython prompt. Here, you should just type `print("Hello world!")` and then press *Shift* + *Enter*.

Congratulations! You are now a Python programmer.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you. You will also find the book's code on this GitHub repository: `https://github.com/ipython-books/minibook-2nd-code`.

Variables

Let's use Python as a calculator.

```
In [2]: 2 * 2
Out[2]: 4
```

Here, `2 * 2` is an expression statement. This operation is performed, the result is returned, and IPython displays it in the notebook cell's output.

Division

In Python 3, 3 / 2 returns 1.5 (floating-point division), whereas it returns 1 in Python 2 (integer division). This can be source of errors when porting Python 2 code to Python 3. It is recommended to always use the explicit 3.0 / 2.0 for floating-point division (by using floating-point numbers) and 3 // 2 for integer division. Both syntaxes work in Python 2 and Python 3. See `http://python3porting.com/differences.html#integer-division` for more details.

Other built-in mathematical operators include `+`, `-`, `**` for the exponentiation, and others. You will find more details at `https://docs.python.org/3/reference/expressions.html#the-power-operator`.

Variables form a fundamental concept of any programming language. A variable has a name and a value. Here is how to create a new variable in Python:

```
In [3]: a = 2
```

And here is how to use an existing variable:

```
In [4]: a * 3
Out[4]: 6
```

Several variables can be defined at once (this is called **unpacking**):

```
In [5]: a, b = 2, 6
```

There are different types of variables. Here, we have used a number (more precisely, an **integer**). Other important types include **floating-point numbers** to represent real numbers, **strings** to represent text, and **booleans** to represent True/False values. Here are a few examples:

```
In [6]: somefloat = 3.1415
        sometext = 'pi is about'  # You can also use double quotes.
        print(sometext, somefloat)  # Display several variables.
Out[6]: pi is about 3.1415
```

Note how we used the # character to write **comments**. Whereas Python discards the comments completely, adding comments in the code is important when the code is to be read by other humans (including yourself in the future).

String escaping

String escaping refers to the ability to insert special characters in a string. For example, how can you insert ' and ", given that these characters are used to delimit a string in Python code? The backslash \ is the go-to escape character in Python (and in many other languages too). Here are a few examples:

```
In [7]: print("Hello \"world\"")
        print("A list:\n* item 1\n* item 2")
        print("C:\\path\\on\\windows")
        print(r"C:\path\on\windows")
Out[7]: Hello "world"
        A list:
        * item 1
        * item 2
        C:\path\on\windows
        C:\path\on\windows
```

The special character `\n` is the **new line** (or line feed) character. To insert a backslash, you need to escape it, which explains why it needs to be doubled as `\\`.

You can also disable escaping by using **raw literals** with a `r` prefix before the string, like in the last example above. In this case, backslashes are considered as normal characters.

This is convenient when writing Windows paths, since Windows uses backslash separators instead of forward slashes like on Unix systems. **A very common error on Windows is forgetting to escape backslashes in paths**: writing `"C:\path"` may lead to subtle errors.

You will find the list of special characters in Python at https://docs.python.org/3.4/reference/lexical_analysis.html#string-and-bytes-literals.

Lists

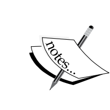
A list contains a sequence of items. You can concisely instruct Python to perform repeated actions on the elements of a list. Let's first create a list of numbers as follows:

```
In [8]: items = [1, 3, 0, 4, 1]
```

Note the syntax we used to create the list: square brackets `[]`, and commas `,` to separate the items.

The built-in function `len()` returns the number of elements in a list:

```
In [9]: len(items)
Out[9]: 5
```



> Python comes with a set of built-in functions, including `print()`, `len()`, `max()`, functional routines like `filter()` and `map()`, and container-related routines like `all()`, `any()`, `range()`, and `sorted()`. You will find the full list of built-in functions at `https://docs.python.org/3.4/library/functions.html`.

Now, let's compute the sum of all elements in the list. Python provides a built-in function for this:

```
In [10]: sum(items)
Out[10]: 9
```

We can also access individual elements in the list, using the following syntax:

```
In [11]: items[0]
Out[11]: 1
In [12]: items[-1]
Out[12]: 1
```

Note that indexing starts at `0` in Python: the first element of the list is indexed by `0`, the second by `1`, and so on. Also, `-1` refers to the last element, `-2` to the penultimate element, and so on.

The same syntax can be used to alter elements in the list:

```
In [13]: items[1] = 9
         items
Out[13]: [1, 9, 0, 4, 1]
```

We can access sublists with the following syntax:

```
In [14]: items[1:3]
Out[14]: [9, 0]
```

Here, `1:3` represents a **slice** going from element `1` *included* (this is the second element of the list) to element `3` *excluded*. Thus, we get a sublist with the second and third element of the original list. The first-included/last-excluded asymmetry leads to an intuitive treatment of overlaps between consecutive slices. Also, note that a sublist refers to a dynamic *view* of the original list, not a copy; changing elements in the sublist automatically changes them in the original list.

Python provides several other types of containers:

- **Tuples** are immutable and contain a fixed number of elements:

  ```
  In [15]: my_tuple = (1, 2, 3)
           my_tuple[1]
  Out[15]: 2
  ```

- **Dictionaries** contain key-value pairs. They are extremely useful and common:

  ```
  In [16]: my_dict = {'a': 1, 'b': 2, 'c': 3}
           print('a:', my_dict['a'])
  Out[16]: a: 1
  In [17]: print(my_dict.keys())
  Out[17]: dict_keys(['c', 'a', 'b'])
  ```

  There is no notion of order in a dictionary. However, the native `collections` module provides an `OrderedDict` structure that keeps the insertion order (see `https://docs.python.org/3.4/library/collections.html`).

- **Sets**, like mathematical sets, contain distinct elements:

  ```
  In [18]: my_set = set([1, 2, 3, 2, 1])
           my_set
  Out[18]: {1, 2, 3}
  ```

A Python object is *mutable* if its value can change after it has been created. Otherwise, it is *immutable*. For example, a string is immutable; to change it, a new string needs to be created. A list, a dictionary, or a set is mutable; elements can be added or removed. By contrast, a tuple is immutable, and it is not possible to change the elements it contains without recreating the tuple. See `https://docs.python.org/3.4/reference/datamodel.html` for more details.

Loops

We can run through all elements of a list using a `for` loop:

```
In [19]: for item in items:
             print(item)
Out[19]: 1
         9
         0
         4
         1
```

There are several things to note here:

- The `for item in items` syntax means that a temporary variable named `item` is created at every iteration. This variable contains the value of every item in the list, one at a time.
- Note the colon `:` at the end of the `for` statement. Forgetting it will lead to a syntax error!
- The statement `print(item)` will be executed for all items in the list.
- Note the four spaces before `print`: this is called the **indentation**. You will find more details about indentation in the next subsection.

Python supports a concise syntax to perform a given operation on all elements of a list, as follows:

```
In [20]: squares = [item * item for item in items]
         squares
Out[20]: [1, 81, 0, 16, 1]
```

This is called a **list comprehension**. A new list is created here; it contains the squares of all numbers in the list. This concise syntax leads to highly readable and Pythonic code.

Indentation

Indentation refers to the spaces that may appear at the beginning of some lines of code. This is a particular aspect of Python's syntax.

In most programming languages, indentation is optional and is generally used to make the code visually clearer. But in Python, indentation also has a syntactic meaning. Particular indentation rules need to be followed for Python code to be correct.

In general, there are two ways to indent some text: by inserting a tab character (also referred to as `\t`), or by inserting a number of spaces (typically, four). It is recommended to use spaces instead of tab characters. Your text editor should be configured such that the *Tab* key on the keyboard inserts four spaces instead of a tab character.

In the Notebook, indentation is automatically configured properly; so you shouldn't worry about this issue. The question only arises if you use another text editor for your Python code.

Finally, what is the meaning of indentation? In Python, indentation delimits coherent blocks of code, for example, the contents of a loop, a conditional branch, a function, and other objects. Where other languages such as C or JavaScript use curly braces to delimit such blocks, Python uses indentation.

Conditional branches

Sometimes, you need to perform different operations on your data depending on some condition. For example, let's display all even numbers in our list:

```
In [21]: for item in items:
             if item % 2 == 0:
                 print(item)
Out[21]: 0
         4
```

Again, here are several things to note:

- An `if` statement is followed by a boolean expression.
- If a and b are two integers, the **modulo** operand `a % b` returns the remainder from the division of a by b. Here, `item % 2` is 0 for even numbers, and 1 for odd numbers.
- The equality is represented by a double equal sign `==` to avoid confusion with the assignment operator `=` that we use when we create variables.
- Like with the `for` loop, the `if` statement ends with a colon `:`.
- The part of the code that is executed when the condition is satisfied follows the `if` statement. It is indented. Indentation is cumulative: since this `if` is inside a `for` loop, there are eight spaces before the `print(item)` statement.

Python supports a concise syntax to select all elements in a list that satisfy certain properties. Here is how to create a sublist with only even numbers:

```
In [22]: even = [item for item in items if item % 2 == 0]
         even
Out[22]: [0, 4]
```

This is also a form of list comprehension.

Functions

Code is typically organized into functions. A **function** encapsulates part of your code. Functions allow you to reuse bits of functionality without copy-pasting the code. Here is a function that tells whether an integer number is even or not:

```
In [23]: def is_even(number):
             """Return whether an integer is even or not."""
             return number % 2 == 0
```

There are several things to note here:

- A function is defined with the `def` keyword.
- After `def` comes the function name. A general convention in Python is to only use lowercase characters, and separate words with an underscore _. A function name generally starts with a verb.

- The function name is followed by parentheses, with one or several variable names called the **arguments**. These are the **inputs** of the function. There is a single argument here, named `number`.
- No type is specified for the argument. This is because Python is **dynamically typed**; you could pass a variable of any type. This function would work fine with floating point numbers, for example (the modulo operation works with floating point numbers in addition to integers).
- The body of the function is indented (and note the colon `:` at the end of the `def` statement).
- There is a **docstring** wrapped by triple quotes `"""`. This is a particular form of comment that explains what the function does. It is not mandatory, but it is strongly recommended to write docstrings for the functions exposed to the user.
- The `return` keyword in the body of the function specifies the **output** of the function. Here, the output is a Boolean, obtained from the expression `number % 2 == 0`. It is possible to return several values; just use a comma to separate them (in this case, a tuple of Booleans would be returned).

Once a function is defined, it can be called like this:

```
In [24]: is_even(3)
Out[24]: False
In [25]: is_even(4)
Out[25]: True
```

Here, `3` and `4` are successively passed as arguments to the function.

Positional and keyword arguments

A Python function can accept an arbitrary number of arguments, called **positional arguments**. It can also accept optional named arguments, called **keyword arguments**. Here is an example:

```
In [26]: def remainder(number, divisor=2):
             return number % divisor
```

The second argument of this function, `divisor`, is optional. If it is not provided by the caller, it will default to the number `2`, as shown here:

```
In [27]: remainder(5)
Out[27]: 1
```

There are two equivalent ways of specifying a keyword argument when calling a function. They are as follows:

```
In [28]: remainder(5, 3)
Out[28]: 2
In [29]: remainder(5, divisor=3)
Out[29]: 2
```

In the first case, `3` is understood as the second argument, `divisor`. In the second case, the name of the argument is given explicitly by the caller. This second syntax is clearer and less error-prone than the first one.

Functions can also accept arbitrary sets of positional and keyword arguments, using the following syntax:

```
In [30]: def f(*args, **kwargs):
             print("Positional arguments:", args)
             print("Keyword arguments:", kwargs)
In [31]: f(1, 2, c=3, d=4)
Out[31]: Positional arguments: (1, 2)
         Keyword arguments: {'c': 3, 'd': 4}
```

Inside the function, `args` is a tuple containing positional arguments, and `kwargs` is a dictionary containing keyword arguments.

Passage by assignment

When passing a parameter to a Python function, a *reference* to the object is actually passed (**passage by assignment**):

- If the passed object is mutable, it can be modified by the function
- If the passed object is immutable, it cannot be modified by the function

Here is an example:

```
In [32]: my_list = [1, 2]

         def add(some_list, value):
             some_list.append(value)

         add(my_list, 3)
         my_list
Out[32]: [1, 2, 3]
```

The `add()` function modifies an object defined outside it (in this case, the object `my_list`); we say this function has **side-effects**. A function with no side-effects is called a **pure function**: it doesn't modify anything in the outer context, and it deterministically returns the same result for any given set of inputs. Pure functions are to be preferred over functions with side-effects.

Knowing this can help you spot out subtle bugs. There are further related concepts that are useful to know, including function scopes, naming, binding, and more. Here are a couple of links:

- Passage by reference at `https://docs.python.org/3/faq/programming.html#how-do-i-write-a-function-with-output-parameters-call-by-reference`
- Naming, binding, and scope at `https://docs.python.org/3.4/reference/executionmodel.html`

Errors

Let's talk about errors in Python. As you learn, you will inevitably come across errors and exceptions. The Python interpreter will most of the time tell you what the problem is, and where it occurred. It is important to understand the vocabulary used by Python so that you can more quickly find and correct your errors.

Let's see the following example:

```
In [33]: def divide(a, b):
             return a / b
In [34]: divide(1, 0)
Out[34]: ---------------------------------------------------------
         ZeroDivisionError       Traceback (most recent call last)
         <ipython-input-2-b77ebb6ac6f6> in <module>()
         ----> 1 divide(1, 0)

         <ipython-input-1-5c74f9fd7706> in divide(a, b)
               1 def divide(a, b):
         ----> 2     return a / b

         ZeroDivisionError: division by zero
```


www.allitebooks.com

Here, we defined a `divide()` function, and called it to divide `1` by `0`. Dividing a number by 0 is an error in Python. Here, a `ZeroDivisionError` **exception** was raised. An exception is a particular type of error that can be raised at any point in a program. It is propagated from the innards of the code up to the command that launched the code. It can be caught and processed at any point. You will find more details about exceptions at `https://docs.python.org/3/tutorial/errors.html`, and common exception types at `https://docs.python.org/3/library/exceptions.html#bltin-exceptions`.

The error message you see contains the stack trace, the exception type, and the exception message. The stack trace shows all function calls between the raised exception and the script calling point.

The top frame, indicated by the first arrow `---->`, shows the entry point of the code execution. Here, it is `divide(1, 0)`, which was called directly in the Notebook. The error occurred while this function was called.

The next and last frame is indicated by the second arrow. It corresponds to line 2 in our function `divide(a, b)`. It is the last frame in the stack trace: this means that the error occurred there.

We will see later in this chapter how to **debug** such errors interactively in IPython and in the Jupyter Notebook. Knowing how to navigate up and down in the stack trace is critical when debugging complex Python code.

Object-oriented programming

Object-oriented programming (**OOP**) is a relatively advanced topic. Although we won't use it much in this book, it is useful to know the basics. Also, mastering OOP is often essential when you start to have a large code base.

In Python, everything is an **object**. A number, a string, or a function is an object. An object is an instance of a **type** (also known as class). An object has **attributes** and **methods**, as specified by its type. An attribute is a variable bound to an object, giving some information about it. A method is a function that applies to the object.

For example, the object `'hello'` is an instance of the built-in `str` type (string). The `type()` function returns the type of an object, as shown here:

```
In [35]: type('hello')
Out[35]: str
```

There are native types, like `str` or `int` (integer), and custom types, also called classes, that can be created by the user.

In IPython, you can discover the attributes and methods of any object with the dot syntax and tab completion. For example, typing `'hello'.u` and pressing *Tab* automatically shows us the existence of the `upper()` method:

```
In [36]: 'hello'.upper()
Out[36]: 'HELLO'
```

Here, `upper()` is a method available to all `str` objects; it returns an uppercase copy of a string.

A useful string method is `format()`. This simple and convenient templating system lets you generate strings dynamically, as shown in the following example:

```
In [37]: 'Hello {0:s}!'.format('Python')
Out[37]: Hello Python!
```

The `{0:s}` syntax means "replace this with the first argument of `format()`, which should be a string". The variable type after the colon is especially useful for numbers, where you can specify how to display the number (for example, `.3f` to display three decimals). The `0` makes it possible to replace a given value several times in a given string. You can also use a name instead of a position—for example `'Hello {name}!'.format(name='Python')`.

Some methods are prefixed with an underscore _; they are private and are generally not meant to be used directly. IPython's tab completion won't show you these private attributes and methods unless you explicitly type _ before pressing *Tab*.

In practice, the most important thing to remember is that appending a dot `.` to any Python object and pressing *Tab* in IPython will show you a lot of functionality pertaining to that object.

Functional programming

Python is a multi-paradigm language; it notably supports imperative, object-oriented, and functional programming models. Python functions are objects and can be handled like other objects. In particular, they can be passed as arguments to other functions (also called **higher-order functions**). This is the essence of **functional programming**.

Decorators provide a convenient syntax construct to define higher-order functions. Here is an example using the `is_even()` function from the previous *Functions* section:

```
In [38]: def show_output(func):
             def wrapped(*args, **kwargs):
                 output = func(*args, **kwargs)
                 print("The result is:", output)
             return wrapped
```

The `show_output()` function transforms an arbitrary function `func()` to a new function, named `wrapped()`, that displays the result of the function, as follows:

```
In [39]: f = show_output(is_even)
         f(3)
Out[39]: The result is: False
```

Equivalently, this higher-order function can also be used with a decorator, as follows:

```
In [40]: @show_output
         def square(x):
             return x * x
In [41]: square(3)
Out[41]: The result is: 9
```

You can find more information about Python decorators at `https://en.wikipedia.org/wiki/Python_syntax_and_semantics#Decorators` and at `http://www.thecodeship.com/patterns/guide-to-python-function-decorators/`.

Python 2 and 3

Let's finish this section with a few notes about Python 2 and Python 3 compatibility issues.

There are still some Python 2 code and libraries that are not compatible with Python 3. Therefore, it is sometimes useful to be aware of the differences between the two versions. One of the most obvious differences is that `print` is a statement in Python 2, whereas it is a function in Python 3. Therefore, `print "Hello"` (without parentheses) works in Python 2 but not in Python 3, while `print("Hello")` works in both Python 2 and Python 3.

There are several non-mutually exclusive options to write portable code that works with both versions:

- **futures**: A built-in module supporting backward-incompatible Python syntax
- **2to3**: A built-in Python module to port Python 2 code to Python 3
- **six**: An external lightweight library for writing compatible code

Here are a few references:

- Official Python 2/3 wiki page at `https://wiki.python.org/moin/Python2orPython3`
- The *Porting to Python 3* book, by *CreateSpace Independent Publishing Platform* at `http://www.python3porting.com/bookindex.html`
- 2to3 at `https://docs.python.org/3.4/library/2to3.html`
- six at `https://pythonhosted.org/six/`
- futures at `https://docs.python.org/3.4/library/__future__.html`
- The *IPython Cookbook* contains an in-depth recipe about choosing between Python 2 and 3, and how to support both.

Going beyond the basics

You now know the fundamentals of Python, the bare minimum that you will need in this book. As you can imagine, there is much more to say about Python.

Following are a few further basic concepts that are often useful and that we cannot cover here, unfortunately. You are highly encouraged to have a look at them in the references given at the end of this section:

- `range` and `enumerate`
- `pass`, `break`, and, `continue`, to be used in loops
- Working with files
- Creating and importing modules
- The Python standard library provides a wide range of functionality (OS, network, file systems, compression, mathematics, and more)

Here are some slightly more advanced concepts that you might find useful if you want to strengthen your Python skills:

- Regular expressions for advanced string processing
- Lambda functions for defining small anonymous functions
- Generators for controlling custom loops
- Exceptions for handling errors
- `with` statements for safely handling contexts
- Advanced object-oriented programming
- Metaprogramming for modifying Python code dynamically
- The `pickle` module for persisting Python objects on disk and exchanging them across a network

Finally, here are a few references:

- Getting started with Python: `https://www.python.org/about/gettingstarted/`
- A Python tutorial: `https://docs.python.org/3/tutorial/index.html`
- The Python Standard Library: `https://docs.python.org/3/library/index.html`
- Interactive tutorial: `http://www.learnpython.org/`

- Codecademy Python course: `http://www.codecademy.com/tracks/python`
- Language reference (expert level): `https://docs.python.org/3/reference/index.html`
- *Python Cookbook*, by *David Beazley* and *Brian K. Jones*, *O'Reilly Media* (advanced level, highly recommended if you want to become a Python expert)

Ten Jupyter/IPython essentials

In this section, we will cover ten essential features of Jupyter and IPython that make them so useful for interactive computing.

Using IPython as an extended shell

> Unfortunately, this subsection will not work well on Windows. The goal here is to demonstrate accessing the operating system's shell from IPython. We could say that, by design, the Windows shell is much more limited than those provided by Linux and OS X. Windows favors user interactions from the graphical interface, whereas Linux and OS X inherit Unix's flexible command-line capabilities. If you want to share and distribute your notebooks, you shouldn't rely on the techniques exposed in this subsection. Rather, you should use the Python equivalents, which are more verbose but also more powerful. Using the shell from IPython is only useful during interactive sessions of users already familiar with the Unix shell.

Open a terminal and type the following commands to go to the minibook's `chapter1` directory and launch the Notebook server:

```
$ cd ~/minibook/chapter1/
$ jupyter notebook
```

In the Notebook dashboard, open the `15-ten.ipynb` notebook. You can also create a new notebook if you prefer not to use the book's code.

Let's illustrate how to use IPython as an extended shell. We will download an example dataset, navigate through the filesystem, and open text files, all from the Notebook. The dataset contains social network data of hundreds of volunteer Facebook users. This BSD-licensed dataset is provided freely by Stanford's SNAP project (`http://snap.stanford.edu/data/`).

IPython provides several **magic commands** that let you interact with your filesystem. These commands are prefixed with a `%`. For example here is how to display the current working directory:

```
In [1]: %pwd
Out[1]: '/home/cyrille/minibook/chapter1'
```

> Like most other magic commands, this magic command works on all operating systems, including Windows. IPython implements several cross-platform Python equivalents of common Unix commands like `pwd`. For other commands not implemented by IPython, we need to call shell commands directly with the `!` prefix (as shown in the following examples). This doesn't work well on Windows since many of these commands are Unix-specific. In brief, `%`-prefixed commands should work on all operating systems while `!`-prefixed commands will generally only work on Linux and OS X, not Windows.

Let's download the dataset from the book's data repository (`https://github.com/ipython-books/minibook-2nd-data`). IPython doesn't yet provide a magic command for downloading data, but we can use another IPython trick: we can run any system or terminal command from IPython by prefixing it with an exclamation mark (`!`). For example, here is how to use the `wget` download utility only available on Unix systems:

```
In [2]: !wget https://raw.githubusercontent.com/ipython-books/minibook-2nd-data/master/facebook.zip
```

> If `wget` is not installed, you can install it with your OS package manager. For example, on Ubuntu: `sudo apt-get install wget`; on OS X: `brew install wget`. On OS X, brew is available at `http://brew.sh/`. On Windows, you should download the file manually from the data repository, as explained later.

This `wget` command downloads a file from a URL and saves it to a file in the local filesystem. Let's display the list of files in the current directory using the `%ls` magic command (available on all systems, even on Windows, since it is a magic command provided by IPython), as follows:

```
In [3]: %ls
Out[3]: facebook.zip  [...]
```

We see a new `facebook.zip` file.

> If you are on Windows, or if downloading the file from IPython didn't work, you can always download this file manually via your web browser at the following URL: `https://github.com/ipython-books/minibook-2nd-data/`. Then save the Facebook dataset in the current directory (the one containing this notebook, which should be `~/minibook/chapter1/`).

The next step is to unzip this file in the current directory. The first way of doing it is to use your operating system, generally with a right-click on the icon. On Linux and OS X, we can also use the `unzip` command-line tool (you may need to install it first, for example with a command like `sudo apt-get install unzip` on Ubuntu). Finally, it is also possible to do it in pure Python with the `zipfile` module (see `https://docs.python.org/3.4/library/zipfile.html`).

Here, we'll call the `unzip` tool, which will only work on Linux and OS X, not Windows:

```
In [4]: !unzip facebook.zip
```

Once the archive has been extracted, a new subdirectory named `facebook` appears, as shown here:

```
In [5]: %ls
Out[5]: facebook  facebook.zip  [...]
```

Let's enter into this subdirectory with the `%cd` magic command (all operating systems), as follows:

```
In [6]: %cd facebook
Out[6]: /home/cyrille/minibook/chapter1/facebook
```

IPython provides a `%bookmark` magic to create an alias to the current directory. Let's type the following:

```
In [7]: %bookmark fbdata
```

Now, in any future session, we'll be able to just type `%cd fbdata` to enter into this directory. Type `%bookmark?` to see all options. This magic command is helpful when dealing with many directories.

Let's display the contents of the directory:

```
In [8]: %ls
Out[8]: 0.circles    1684.circles  3437.circles  3980.circles  686.
circles
        0.edges      1684.edges    3437.edges    3980.edges    686.edges
        107.circles  1912.circles  348.circles   414.circles   698.
circles
        107.edges    1912.edges    348.edges     414.edges     698.edges
```

Here, every number identifies a Facebook user (called the ego user). The `.edges` file contains its social graph. In this graph, nodes represent other Facebook users, and edges represent friendship links between them. The `.circles` file contains lists of friends.

Let's retrieve the list of `.edges` files with the following command (which won't work on Windows):

```
In [9]: files = !ls -1 -S | grep .edges
```

The Unix command `ls -1 -S` lists all files in the current directory, sorted by decreasing size. The pipe `| grep edges` filters only those files that contain `.edges`. Then, this list is assigned to a new Python variable named `files`, as follows:

```
In [10]: files
Out[10]: ['1912.edges',
          '107.edges',
          '1684.edges',
          '3437.edges',
          '348.edges',
          '0.edges',
          '414.edges',
          '686.edges',
          '698.edges',
          '3980.edges']
```

On Windows, you can use the following Python code to obtain the same list (if you're not on Windows, you can skip this code listing):

```
In [11]: import os
         from operator import itemgetter
         # Get the name and file size of all .edges files.
         files = [(file, os.stat(file).st_size)
                  for file in os.listdir('.')
                  if file.endswith('.edges')]
         # Sort the list with the second item (file size),
         # in decreasing order.
         files = sorted(files,
                        key=itemgetter(1),
                        reverse=True)
         # Only keep the first item (file name), in the same order.
         files = [file for (file, size) in files]
```

Let's display the first few lines of the first file in the list (Unix-specific command):

```
In [12]: !head -n5 {files[0]}
Out[12]: 2290 2363
         2346 2025
         2140 2428
         2201 2506
         2425 2557
```

The curly braces `{}` let us insert a Python variable within a system command (here, the `head` Unix command which displays the first lines of a text file).

In an `.edges` file, every line contains the two nodes forming every edge. The `.circles` file contains lists of friends. Every line contains a space-separated list of the users forming every circle.

Alias commands

If you use a complex command regularly, you can create an **alias** with the `%alias` magic command. Type `%alias?` for more information. See also the related `%store` magic command.

Learning magic commands

Besides the filesystem commands we have seen in the previous section, IPython provides many other magic commands. You can display the list of all magic commands with the `%lsmagic` magic command, as follows:

```
In [13]: %lsmagic
Out[13]: Available line magics:
         %alias  %alias_magic  %autocall  %automagic  %autosave
%bookmark  %cat  %cd  %clear  %colors  %config  %connect_info  %cp
%debug  %dhist  %dirs  %doctest_mode  %ed  %edit  %env  %gui  %hist
%history  %install_default_config  %install_ext  %install_profiles
%killbgscripts  %ldir  %less  %lf  %lk  %ll  %load  %load_ext  %loadpy
%logoff  %logon  %logstart  %logstate  %logstop  %ls  %lsmagic  %lx
%macro  %magic  %man  %matplotlib  %mkdir  %more  %mv  %notebook  %page
%pastebin  %pdb  %pdef  %pdoc  %pfile  %pinfo  %pinfo2  %popd  %pprint
%precision  %profile  %prun  %psearch  %psource  %pushd  %pwd  %pycat
%pylab  %qtconsole  %quickref  %recall  %rehashx  %reload_ext  %rep
%rerun  %reset  %reset_selective  %rm  %rmdir  %run  %save  %sc  %set_env
%store  %sx  %system  %tb  %time  %timeit  %unalias  %unload_ext  %who
%who_ls  %whos  %xdel  %xmode


         Available cell magics:
         %%!  %%HTML  %%SVG  %%bash  %%capture  %%debug  %%file  %%html
%%javascript  %%latex  %%perl  %%prun  %%pypy  %%python  %%python2
%%python3  %%ruby  %%script  %%sh  %%svg  %%sx  %%system  %%time
%%timeit  %%writefile


         Automagic is ON, % prefix IS NOT needed for line magics.
```

To obtain information about a magic command, append a question mark (`?`) after the command, as shown in the following example:

```
In [14]: %history?
```

The `%history` magic command lets you display and manipulate your command history in IPython. For example, the following command shows your last five commands:

```
In [15]: %history -l 5
Out[15]: files = !ls -1 -S | grep .edges
         files
         !head -n5 {files[0]}
         %lsmagic
         %history?
```

Let's also mention the `%dhist` magic command that shows you a history of all visited directories.

Another useful magic command is `%paste`, which lets you copy-paste Python code from anywhere into the IPython console (it is not available in the Notebook, where you can copy-paste as usual).

In IPython, the underscore (_) character always contains the last output. This is useful if you ran some command and forgot to assign the output to a variable.

```
In [16]: # how many minutes in a day?
         24 * 60
Out[16]: 1440
In [17]: # and in a year?
         _ * 365
Out[17]: 525600
```

We will now see several **cell magics**, which are magic commands that apply to a whole code cell rather than just a line of code. They are prefixed by two percent signs (`%%`).

The `%%capture` cell magic lets you capture the standard output and error output of some code into a Python variable. Here is an example (the outputs are captured in the `output` Python variable):

```
In [18]: %%capture output
         %ls
In [19]: output.stdout
Out[19]: 0.circles    1684.circles  3437.circles  3980.circles  686.
circles
         0.edges      1684.edges    3437.edges    3980.edges    686.edges
         107.circles  1912.circles  348.circles   414.circles   698.
circles
         107.edges    1912.edges    348.edges     414.edges     698.edges
```

The `%%bash` cell magic is an extension of the `!` shell prefix. It lets you run multiline bash code in the Notebook, as shown here:

```
In [20]: %%bash
         cd ..
         touch _HEY
         ls
         rm _HEY
         cd facebook
Out[20]: _HEY
         facebook
         facebook.zip
         [...]
```

More generally, the `%%script` cell magic lets you execute code with any program installed on your system. For example, assuming Haskell is installed (see `https://www.haskell.org/downloads`), you can easily execute Haskell code from the Notebook, as follows:

```
In [21]: %%script ghci
         putStrLn "Hello world!"
Out[21]: GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
         Loading package ghc-prim ... linking ... done.
         Loading package integer-gmp ... linking ... done.
         Loading package base ... linking ... done.
         Prelude> Hello world!
         Prelude> Leaving GHCi.
```

The `ghci` executable runs in a separate process, and the contents of the cell are passed to the executable's input. You can also put a full path after `%%script`, for example, on Linux: `%%script /usr/bin/ghci`.

IHaskell kernel

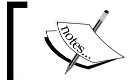
This way of calling external scripts is only useful for quick interactive experiments. If you want to run Haskell notebooks, you can use the IHaskell notebook for Jupyter, available at `https://github.com/gibiansky/IHaskell`.

Finally, the `%%writefile` cell magic lets you write some text in a new file, as shown here:

```
In [22]: %%writefile myfile.txt
         Hello world!
Out[22]: Writing myfile.txt
In [23]: !more myfile.txt
Out[23]: Hello world!
```

Now, let's delete the file, as follows:

```
In [24]: !rm myfile.txt
```



On Windows, you need to type `!del myfile.txt` instead.

There are many other magic commands available. We will see several of them later in this book. Also, in *Chapter 6*, *Customizing IPython*, we will see how to create new magic commands. This is much easier than it sounds!

Refer to the following page for up-to-date documentation about all magic commands: `http://www.ipython.org/ipython-doc/dev/interactive/magics.html`.

Mastering tab completion

Tab completion is an incredibly useful feature in Jupyter and IPython. When you start to write something and press the *Tab* key on your keyboard, IPython can guess what you're trying to do, and propose a list of options that match what you have typed so far. This works for Python functions, variables, magic commands, files, and more.

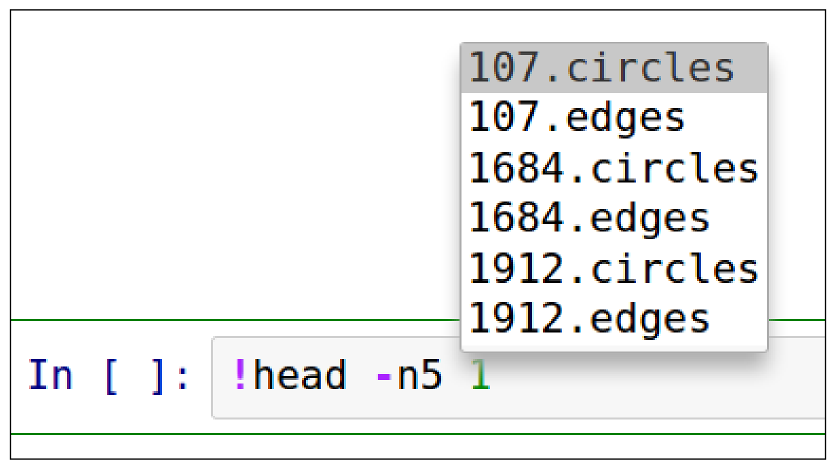
Let's first make sure we are in the `facebook` directory (using the directory alias created previously):

```
In [25]: %cd fbdata
         %ls
Out[25]: (bookmark:fbdata) -> /home/cyrille/minibook/chapter1/facebook
         /home/cyrille/minibook/chapter1/facebook
         0.circles    1684.circles  3437.circles  3980.circles  686.
circles
         0.edges      1684.edges    3437.edges    3980.edges    686.edges
         107.circles  1912.circles  348.circles   414.circles   698.
circles
         107.edges    1912.edges    348.edges     414.edges     698.edges
```

Now, start typing a command and press *Tab* before finishing it (here, press the *Tab* key on your keyboard right after typing `e`), as follows:

```
!head -n5 107.e<TAB>
```

IPython automatically completes the command and adds the four remaining characters (`dges`). IPython recognized the beginning of a file name and completed the command. If there are several completion possibilities, IPython doesn't complete anything, but instead shows a list of all options. You can then choose the appropriate solution by pressing the Up or Down keys on the keyboard, and pressing *Tab* again. The following screenshot shows an example:



Tab completion in the Notebook

Tab completion is extremely useful when you're getting acquainted with a new Python package. For example, to quickly see all functions provided by the NetworkX package, you can type `import networkx; networkx.<TAB>`.

Customizing tab completion

If you're writing a Python library, you probably want to write tab-completion-aware code. Your users who work with IPython will thank you! In most cases, you have nothing to do, and tab completion will just work. In the rare cases where you use advanced dynamic techniques in a class, you can customize tab completion by implementing a `__dir__(self)` method that returns all attributes available in the current class instance. See this reference for more details: `https://docs.python.org/3.4/library/functions.html#dir`.

Writing interactive documents in the Notebook with Markdown

You can write code and text in the Notebook. Every cell is either a Markdown cell or a code cell. The Markdown cell lets you write text. Markdown is a text formatting syntax that supports headers, bold, italics, hypertext links, images, and code. In the Notebook, you can also write mathematical equations in a Markdown cell using LaTeX, a markup language widely used for equations. Finally, you can also write some HTML in a Markdown cell, and it will be interpreted correctly.

Here is an example of a paragraph in Markdown:

````
### New paragraph


This is *rich* **text** with [links](http://ipython.org), equations:


$$\hat{f}(\xi) = \int_{-\infty}^{+\infty} f(x)\, \mathrm{e}^{-i \xi x}
dx$$


code with syntax highlighting:


    ```python
    print("Hello world!")
    ```


and images:


![This is an image](http://ipython.org/_static/IPy_header.png)
````

If you write this in a Markdown cell, and "play" the cell (for example, by pressing *Ctrl* + *Enter*), you will see the rendered text. The following screenshot shows the two modes of the cell:

````
### New paragraph
This is *rich* **text** with [links](http://ipython.org), equations:

$$\hat{f}(\xi) = \int_{-\infty}^{+\infty} f(x)\, \mathrm{e}^{-i \xi x} dx$$

code with syntax highlighting:

```python
print("Hello world!")
```

and images:

![This is an image](http://jupyter.org/images/jupyter-sq-text.svg)
````

New paragraph

This is *rich* **text** with links, equations:

$$\hat{f}(\xi) = \int_{-\infty}^{+\infty} f(x)\, \mathrm{e}^{-i \xi x} dx$$

code with syntax highlighting:

```python
print("Hello world!")
```

and images:

IP[y]: IPython Interactive Computing

A Markdown cell in the Notebook

By using both Markdown cells and code cells in a notebook, you can write an interactive document about any technical topic. Hence, the Notebook is not only an interface to code, it is also a platform to write documents or even books. In fact, this very book is entirely written in the Notebook!

Here are a few references about Markdown and LaTeX:

- Markdown on Wikipedia at `http://en.wikipedia.org/wiki/Markdown`
- The original specification, at `http://daringfireball.net/projects/markdown/`
- A Markdown tutorial by GitHub, at `https://help.github.com/articles/markdown-basics/`
- CommonMark, a standardized version of Markdown, at `http://commonmark.org/`
- LaTeX on Wikipedia at `http://en.wikipedia.org/wiki/LaTeX`

Creating interactive widgets in the Notebook

You can add interactive graphical elements called **widgets** in a notebook. Examples of rich graphical widgets include buttons, sliders, dropdown menus, interactive plots, as well as videos, audio files, and complete **Graphical User Interfaces** (**GUIs**). Widget support in Jupyter is still relatively experimental at this point, but we will use them at several occasions in this book. This section shows a few basic examples.

First, let's add a YouTube video in a notebook, as follows:

```
In [26]: from IPython.display import YouTubeVideo
         YouTubeVideo('j9YpkSX7NNM')
```

Following is a screenshot of a YouTube video in a notebook:

Youtube in the Notebook

The `YoutubeVideo` constructor accepts a YouTube identifier as input.

Next, let's show how to create a graphical control to manipulate the inputs to a Python function:

```
In [27]: from ipywidgets import interact
         # IPython.html.widgets before
         # IPython 4.0
         @interact(x=(0, 10))
         def square(x):
             print("The square of %d is %d." % (x, x**2))
Out[27]: 'The square of 7 is 49.'
```

Here is a screenshot:

```
In [2]: from ipywidgets import interact
        # IPython.html.widgets before IPython 4.0
        @interact(x=(0, 10))
        def square(x):
            print("The square of %d is %d." % (x, x**2))

            x                                   7

        The square of 7 is 49.
```

Interactive widget in the Notebook

The `square(x)` function just prints a sentence like `The square of 7 is 49`. By adding the `@interact` decorator above the function's definition, we tell IPython to create a widget to control the function's input `x`. The argument `x=(0, 10)` is a convention to indicate that we want a slider to control an integer between 0 and 10.

This method supports other common controls like checkboxes, dropdown menus, radio buttons, push buttons, and others.

Finally, entirely customizable widgets can be created, but this requires some knowledge of web technologies such as HTML, CSS, and JavaScript. The *IPython Cookbook* (`http://ipython-books.github.io/cookbook/`) contains many examples. You can also refer to the following links for more information:

- IPython widgets tutorial at `https://github.com/ipython/ipywidgets/blob/master/examples/Index.ipynb`
- Introducing the interactive features of the IPython Notebook, at `https://github.com/rossant/euroscipy2014`
- A piano in the Notebook, at `http://nbviewer.ipython.org/github/ipython-books/cookbook-code/blob/master/notebooks/chapter03_notebook/05_basic_widgets.ipynb`

Most of these references describe APIs that were introduced in IPython 3.0, but are still experimental at this point. They may not work with future versions of Jupyter and IPython.

Running Python scripts from IPython

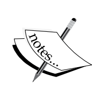
Notebooks are mainly designed for interactive exploration, not for reusability. It is currently difficult to reuse parts of a notebook in another script or notebook. Many users just copy-paste their code, which goes against the **Don't Repeat Yourself** (**DRY**) principle.

A common practice is to put frequently used code into a Python script, for example `myscript.py`. Such a script can be called from the system terminal like this: `python myscript.py`. Python will execute the script and quit at the end. If you use the `-i` option, Python will start the interactive prompt when the script ends.

IPython also supports this technique; just replace `python` by `ipython`. For example: `ipython -i script.py` to run `script.py` interactively with IPython.

You can also run a script from within IPython by using the `%run` magic command. The script runs in an empty namespace, meaning that any variable defined in the interactive namespace is not available within the executed script. However, at the end of the execution, the control returns to IPython, and the variables defined in the script are imported into the interactive namespace. This lets you inspect the intermediate variables used in the script. If you use the `-i` option, the script will run in the interactive namespace. Any variable defined in the interactive session will be available in the script.

Let's also mention the similar `%load` magic command.



A **namespace** is a dictionary mapping variable names to Python objects. The **global namespace** contains global variables, whereas the **local namespace** of a function contains the local variables defined in the function. In IPython, the **interactive namespace** contains all objects defined and imported within the current interactive session. The `%who`, `%whos`, and `%who_ls` magic commands give you some information about the interactive variables.

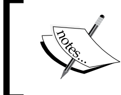
For example, let's write a script `egos.py` that lists all ego identifiers in the Facebook data folder. Since each filename is of the form `<egoid>.<extension>`, we list all files, remove the extensions, and take the sorted list of all unique identifiers. We can create this file from the Notebook, using the `%%writefile` cell magic as follows:

```
In [28]: %cd fbdata
         %cd ..
Out[28]: (bookmark:fbdata) -> /home/cyrille/minibook/chapter1/facebook
         /home/cyrille/minibook/chapter1/facebook
In [29]: %%writefile egos.py
         import sys
         import os
         # We retrieve the folder as the first positional argument
         # to the command-line call
         if len(sys.argv) > 1:
             folder = sys.argv[1]
         # We list all files in the specified folder
         files = os.listdir(folder)
         # ids contains the list of idenfitiers
         identifiers = [int(file.split('.')[0]) for file in files]
         # Finally, we remove duplicates with set(), and sort the list
         # with sorted().
         ids = sorted(set(identifiers))
Out[29]: Overwriting egos.py
```

This script accepts an argument `folder` as an input. It is retrieved from the Python script via the `sys.argv` list, which contains the list of arguments passed to the script via the command-line interface.

Let's execute this script in IPython using the `%run` magic command, as follows:

```
In [30]: %run egos.py facebook
```



If you get an error when running this script, make sure that the `facebook` directory only contains `<number>.xxx` files (like `0.circles` or `1684.edges`).

```
In [31]: ids
Out[31]: [0, 107, 348, 414, 686, 698, 1684, 1912, 3437, 3980]
```

The `ids` variable created in the script is now available in the interactive namespace.

Let's see what happens if we do not specify the folder name to the script, as follows:

```
In [32]: folder = 'facebook'
In [33]: %run egos.py
```

We get an error: `NameError: name 'folder' is not defined`. This is because the variable `folder` is defined in the interactive namespace, but is not available within the script by default. We can change this behavior with the `-i` option, as follows:

```
In [34]: %run -i egos.py
In [35]: ids
Out[35]: [0, 107, 348, 414, 686, 698, 1684, 1912, 3437, 3980]
```

This time, the script correctly used the `folder` variable.

Introspecting Python objects

IPython can display detailed information about any Python object.

First, type `?` after a variable name to get some information about it. For example, let's inspect NetworkX's `Graph` class, as follows:

```
In [36]: import networkx
In [37]: networkx.Graph?
```

This shows the docstring and other information in the Notebook pager, as shown in the following screenshot:

```
In [3]: import networkx

In [4]: networkx.Graph?

Type:            type
String form:     <class 'networkx.classes.graph.Graph'>
File:            /home/cyrille/anaconda/envs/minibook/lib/python3.4/site-packages/n
etworkx/classes/graph.py
Init definition: networkx.Graph(self, data=None, **attr)
Docstring:
Base class for undirected graphs.

A Graph stores nodes and edges with optional data, or attributes.

Graphs hold undirected edges.  Self loops are allowed but multiple
(parallel) edges are not.

Nodes can be arbitrary (hashable) Python objects with optional
```

Typing `??` instead of `?` shows even more information, including the whole source code of the Python object when it is available.

There are also several magic commands for inspecting Python objects:

- `%pdef`: Displays a function definition
- `%pdoc`: Displays the docstring of a Python object
- `%psource`: Displays the source code of an object (function, class, or method)
- `%pfile`: Displays the source code of the Python script where an object is defined

Debugging Python code

IPython makes it convenient to debug a script or an entire application. It provides interactive access to an enhanced version of the Python debugger.

First, when you encounter an exception, you can immediately use the `%debug` magic command to launch the IPython debugger at the exact point where the exception was raised.

If you activate the `%pdb` magic command, the debugger will automatically start at the very next exception. You can also start IPython with `ipython --pdb`.

Finally, you can run a whole script under the control of the debugger with the `%run -d` command. This command executes the specified script with a break point at the first line so that you can precisely control the execution flow of the script. You can also specify explicitly where to put the first breakpoint; type `%run -d -b29 script.py` to pause the program execution on line 29 of `script.py`. In all cases, you first need to type `c` to start the script execution.

When the debugger starts, you enter into a special prompt, as indicated by `ipdb>`. The program execution is then paused at a given point in the code. You can type `w` to display the line and stack location where the debugger has paused. At this point, you have access to all local variables and you can precisely control how you want to resume the execution. Within the debugger, several commands are available to navigate into the traceback; they are as follows:

- `u`/`d` for going *up*/*down* into the call stack
- `s` to *step* into the next statement
- `n` to continue execution until the *next line* in the current function
- `r` to continue execution until the current function *returns*
- `c` to *continue* execution until the next breakpoint or exception

Other useful commands include:

- `p` to evaluate and *print* any expression
- `a` to obtain the *arguments* of the current functions
- The `!` prefix to execute any Python command within the debugger

The entire list of commands can be found in the documentation of the `pdb` module in Python at `https://docs.python.org/3.4/library/pdb.html`.

Let's also mention the `IPython.embed()` function that you can call anywhere in a Python script. This stops the script execution and starts IPython for debugging purposes. Leaving the embedded IPython terminal resumes the normal execution of the script.

Benchmarking Python code

The `%timeit` magic function lets us estimate the execution time of any Python statement. Under the hood, it uses Python's native `timeit` module.

In the following example, we first load an ego graph from our Facebook dataset using the NetworkX package. Then we evaluate how much time it takes to tell whether the graph is connected or not:

Let's go to the data directory, as follows:

```
In [38]: %cd fbdata
Out[38]: (bookmark:fbdata) -> /home/cyrille/minibook/chapter1/facebook
         /home/cyrille/minibook/chapter1/facebook
```

We load NetworkX, as follows:

```
In [39]: import networkx
```

We can load a graph using the `read_edgelist()` function, as follows:

```
In [40]: graph = networkx.read_edgelist('107.edges')
```

How big is our graph?

```
In [41]: len(graph.nodes()), len(graph.edges())
Out[41]: (1034, 26749)
```

Now let's find out whether the graph is connected or not:

```
In [42]: networkx.is_connected(graph)
Out[42]: True
```

How long did this call take?

```
In [43]: %timeit networkx.is_connected(graph)
Out[43]: 100 loops, best of 3: 5.92 ms per loop
```

Multiple calls are done in order to get more reliable time estimates. The number of calls is determined automatically, but you can use the `-r` and `-n` options to specify them directly. Type `%timeit?` to get more information.

Profiling Python code

The `%timeit` magic command gives you precious information about the total time taken by a function or a statement. This can help you find the fastest among several implementations of an algorithm, for example.

When you're finding that some code is too slow, you need to *profile* it before you can make it faster. Profiling gives you more than the total time taken by a function; it tells you exactly what is taking too long in your code.

The `%prun` magic command lets you easily profile your code. It provides a convenient interface to Python's native `profile` module.

Let's see a simple example. We first create a function returning the number of connected components in a file, as follows:

```
In [44]: import networkx
In [45]: def ncomponents(file):
             graph = networkx.read_edgelist(file)
             return networkx.number_connected_components(graph)
```

Now we write a function that returns the number of connected components in all graphs defined in the directory, as follows:

```
In [46]: import glob
         def ncomponents_files():
             return [(file, ncomponents(file))
                     for file in sorted(glob.glob('*.edges'))]
```

The `glob` module (`https://docs.python.org/3.4/library/glob.html`) lets us find all files matching a given pattern (here, all files with the `.edges` file extension).

```
In [47]: for file, n in ncomponents_files():
             print(file.ljust(12), n, 'component(s)')
```

```
Out[47]: 0.edges      5 component(s)
         107.edges    1 component(s)
         1684.edges   4 component(s)
         1912.edges   2 component(s)
         3437.edges   2 component(s)
         348.edges    1 component(s)
         3980.edges   4 component(s)
         414.edges    2 component(s)
         686.edges    1 component(s)
         698.edges    3 component(s)
```

Let's first evaluate the time taken by this function:

```
In [48]: %timeit ncomponents_files()
Out[48]: 1 loops, best of 3: 634 ms per loop
```

Now, to run the profiler, we use the `%prun` magic function, as follows:

```
In [49]: %prun -s cumtime ncomponents_files()
Out[49]: 2391070 function calls in 1.038 seconds

         Ordered by: cumulative time

         ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
              1    0.000    0.000    1.038    1.038 {built-in method
exec}
              1    0.000    0.000    1.038    1.038 <string>:1(<module>)
             10    0.000    0.000    0.995    0.100 <string>:1(read_
edgelist)
             10    0.000    0.000    0.995    0.100 decorators.py:155(_
open_file)
             10    0.376    0.038    0.995    0.099 edgelist.
py:174(parse_edgelist)
         170174    0.279    0.000    0.350    0.000 graph.py:648(add_
edge)
         170184    0.059    0.000    0.095    0.000 edgelist.
py:366(<genexpr>)
             10    0.000    0.000    0.021    0.002 connected.
py:98(number_connected_components)
             35    0.001    0.000    0.021    0.001 connected.
py:22(connected_components)
```

Let's explain what happened here. The profiler kept track of all function calls (including functions internal to NetworkX and Python) performed while our `ncomponents_files()` function was running. There were 2,391,070 function calls. That's a lot! Opening a file, reading and parsing every line, creating the graphs, finding the number of connected components, and so on, are operations that involve many function calls.

The profiler shows the list of all function calls (we just showed a subset here). There are many ways to sort the functions. Here, we chose to sort them by cumulative time, which is the total time spent within every function (`-s cumtime` option).

For every function, the profiler shows the total number of calls, and several time statistics, described here (copied verbatim from the profiler documentation):

- `tottime`: the total time spent in the given function (and excluding time made in calls to sub-functions)
- `percall`: the quotient of `tottime` divided by `ncalls`
- `cumtime`: the cumulative time spent in this and all subfunctions
- `percall`: the quotient of `cumtime` divided by the number of non-recursive function calls

You will find more information by typing `%prun?` or by looking here: `https://docs.python.org/3.4/library/profile.html`

Here, we see that computing the number of connected components took considerably less time than loading the graphs from the text files. Depending on the use-case, this might suggest using a more efficient file format.

There is of course much more to say about profiling and optimization. For example, it is possible to profile a function line by line, which provides an even more fine-grained profiling report. The *IPython Cookbook* contains many more details.

Summary

In this chapter, we covered everything you need to get started with Python, IPython, and the Jupyter Notebook. We detailed how to install the software, we reviewed the basics of the Python language, and we demonstrated ten of the most essential features of IPython and the Jupyter Notebook.

In the next chapter, we will use these tools to analyze real-world datasets.

2
Interactive Data Analysis with pandas

In this chapter, we will cover the following topics:

- Exploring a dataset in the Notebook
- Manipulating data
- Complex operations

We'll see how to load, explore, and visualize a real-world dataset with pandas, matplotlib, and seaborn, all in the Notebook. We will also perform data manipulations efficiently.

Exploring a dataset in the Notebook

Here, we will explore a dataset containing the taxi trips made in New York City in 2013. Maintained by the New York City Taxi and Limousine Commission, this 50GB dataset contains the date, time, geographical coordinates of pickup and dropoff locations, fare, and other information for 170 million taxi trips.

To keep the analysis times reasonable, we will analyze a subset of this dataset containing 0.5% of all trips (about 850,000 rides). Compressed, this subset data represents a little less than 100MB. You are free to download and analyze the full dataset (or a larger subset), as explained below.

Provenance of the data

You will find the data subset we will be using in this chapter at `https://github.com/ipython-books/minibook-2nd-data`.

If you are interested in the original dataset containing all trips, you can refer to `https://github.com/ipython-books/minibook-2nd-code/tree/master/chapter2/cleaning`. This page contains the code to download the original dataset and create the data subset we'll be using in this chapter. It is recommended to have **100GB of free space** on your hard drive before you proceed with the full dataset (this requirement doesn't apply to the data subset we will be using in this chapter, of course).

The original 50GB dataset contained 24 zipped CSV files (a *data* and a *fare* file for every month). We created a Python script going through all of these files and extracting one row out of 200 rows.

Then, we ordered the rows by chronological order (using the pickup time).

Next, we removed all rows with inconsistent coordinates. We defined the coordinates of a rectangle surrounding Manhattan (to restrict ourselves to this area) and we only kept the rows where both pickup and dropoff locations were within this rectangle.

Finally, we ended up with two cleaned `nyc_data.csv` and `nyc_fare.csv` files.

We used some features of pandas to perform these steps efficiently. We will cover them later in this chapter.

Here are a few references:

- History of the dataset at `http://chriswhong.com/open-data/foil_nyc_taxi/`
- More recent data at `http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml`
- An interactive web application based on this dataset at `http://hubcab.org`
- pandas documentation at `http://pandas.pydata.org/pandas-docs/stable/`
- *Python for Data Analysis*, *O'Reilly Media*, by *Wes McKinney*, the creator of pandas, at `http://shop.oreilly.com/product/0636920023784.do`

Public datasets

There is now a large variety of public datasets available online as part of the *open data* movement. Here are a few references:

- Open data on Wikipedia at `https://en.wikipedia.org/wiki/Open_data`
- Curated list of public datasets at `https://github.com/caesar0301/awesome-public-datasets`
- The home of the U.S. Government's open data at `http://www.data.gov`
- The open platform for French public data at `https://www.data.gouv.fr/en/`

Downloading and loading a dataset

Let's import a few packages we will need here.

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        %matplotlib inline
```

It is a common practice to import NumPy and assign it the `np` alias. Same for pandas with `pd`, and matplotlib's high-level interface named `pyplot` with `plt`. The `%matplotlib inline` magic command tells matplotlib to render figures as static images in the Notebook.

We now move to the `chapter2` subdirectory in the minibook's directory:

```
In [2]: %cd ~/minibook/chapter2/
```

Next, let's download the data subset, available on the book's data repository at `https://github.com/ipython-books/minibook-2nd-data`. **If you are on Windows, the following two commands won't work**. Instead, you can download the *NYC Taxi dataset* from the URL above and extract it in the current directory with a right-click.

```
In [3]: !wget https://raw.githubusercontent.com/ipython-books/minibook-
2nd-data/master/nyc_taxi.zip
        !unzip nyc_taxi.zip
In [4]: %ls data
Out[4]: nyc_data.csv  nyc_fare.csv  [...]
```

We are now in `~/minibook/chapter2/`, and we should have a `data/` subdirectory containing two CSV files. The `nyc_data.csv` file contains information about the rides, whereas `nyc_fare.csv` contains information about the fares.

```
In [5]: data_filename = 'data/nyc_data.csv'
        fare_filename = 'data/nyc_fare.csv'
```

Now, let's load the data. pandas provides a powerful `read_csv()` function that can read virtually any CSV file. This function accepts many options, as you can see in pandas' documentation page at `http://pandas.pydata.org/pandas-docs/stable/generated/pandas.io.parsers.read_csv.html`. Here, we just need to specify which columns contain the dates, so that pandas can parse them correctly.

Trial and error

Typically, you should first try to open a dataset with `read_csv(filename)` with no special argument. If an error occurs, or if some columns are parsed incorrectly, you can fix the problem by passing extra parameters to the function, like we did here with `parse_dates`. You will find more information in pandas' documentation.

```
In [6]: data = pd.read_csv(data_filename,
                           parse_dates=['pickup_datetime',
                                        'dropoff_datetime'])
        fare = pd.read_csv(fare_filename,
                           parse_dates=['pickup_datetime'])
```

The `data` and `fare` variables are **DataFrame** objects. A DataFrame is a table containing rows (**observations** or **samples**) and columns (**features** or **variables**). DataFrames can contain text, numbers, dates, and other types of data. pandas provides Notebook-friendly display facilities for DataFrames, as we can see here:

```
In [7]: data.head(3)
```

The `head()` method of DataFrames displays the first few lines (here, three) of the table. Here is a screenshot:

```
data.head(3)
```

	medallion	hack_license	vendor_id	rate_code	store_and_fwd_flag	pickup_datetim
0	76942C3205E17D7E7FE5A9F709D16434	25BA06A87905667AA1FE5990E33F0E2E	VTS	1	NaN	2013-01-01 00:00:00
1	517C6B330DBB3F055D007B07512628B3	2C19FBEE1A6E05612EFE4C958C14BC7F	VTS	1	NaN	2013-01-01 00:05:00
2	ED15611F168E41B33619C83D900FE266	754AEBD7C80DA17BA1D81D89FB6F4D1D	CMT	1	N	2013-01-01 00:05:52

A DataFrame in the Notebook

Similarly, the `tail()` method displays the last few lines of a DataFrame.

The `describe()` method shows basic statistics of all columns, as shown in the following screenshot:

```
data.describe()
```

	rate_code	passenger_count	trip_time_in_secs	trip_distance	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latit
count	846945.000000	846945.000000	846945.000000	846945.000000	846945.000000	846945.000000	846945.000000	846945.0000
mean	1.026123	1.710272	812.523879	9.958211	-73.975155	40.750490	-73.974197	40.750967
std	0.223480	1.375266	16098.305145	6525.204888	0.035142	0.027224	0.033453	0.030766
min	0.000000	0.000000	-10.000000	0.000000	-74.098305	40.009911	-74.099998	40.009911
25%	1.000000	1.000000	361.000000	1.050000	-73.992371	40.736031	-73.991570	40.735207
50%	1.000000	1.000000	600.000000	1.800000	-73.982094	40.752975	-73.980614	40.753597
75%	1.000000	2.000000	960.000000	3.200000	-73.968048	40.767460	-73.965157	40.768227
max	6.000000	6.000000	4294796.000000	6005123.000000	-73.028473	40.996132	-73.027061	40.998592

Describing a dataset

Making plots with matplotlib

Visualizing *raw* data, as opposed to aggregated statistics, often allows us to get a general idea about a dataset. Here, we will display the pickup and dropoff locations of all trips.

The first step is to get the actual coordinates from the DataFrame. We can find the list of columns as follows:

```
In [8]: data.columns
Out[8]: Index(['medallion',
                ...
                'pickup_datetime',
                'dropoff_datetime',
                'passenger_count',
                'trip_time_in_secs',
                'trip_distance',
                'pickup_longitude',
                'pickup_latitude',
                'dropoff_longitude',
                'dropoff_latitude'], dtype='object')
```

Four columns mention `latitude` and `longitude`. Let's load these columns:

```
In [9]: p_lng = data.pickup_longitude
        p_lat = data.pickup_latitude
        d_lng = data.dropoff_longitude
        d_lat = data.dropoff_latitude
```

With pandas, every column of a DataFrame can be obtained with the `mydataframe.columnname` syntax. An alternative syntax is `mydataframe['columnname']`.

Here, we created four variables with the coordinates of the pickup and dropoff locations. These variables are all `Series` objects:

```
In [10]: p_lng
Out[10]: 0        -73.955925
         1        -74.005501
         ...
         846943   -73.978477
         846944   -73.987206
         Name: pickup_longitude, Length: 846945, dtype: float64
```

A Series is an indexed list of values. Therefore, a DataFrame is simply a collection of Series columns.

Before we can make a plot, we need to get the coordinates of points in pixels instead of geographical coordinates. We can use the following function that performs a Mercator projection:

```
In [11]: def lat_lng_to_pixels(lat, lng):
             lat_rad = lat * np.pi / 180.0
             lat_rad = np.log(np.tan((lat_rad + np.pi / 2.0) / 2.0))
             x = 100 * (lng + 180.0) / 360.0
             y = 100 * (lat_rad - np.pi) / (2.0 * np.pi)
             return (x, y)
```

NumPy implements many mathematical functions like `np.log()` and `np.tan()`. These functions work on scalar numbers and also on pandas objects such as Series. Here, the following function call returns two new Series `px` and `py`:

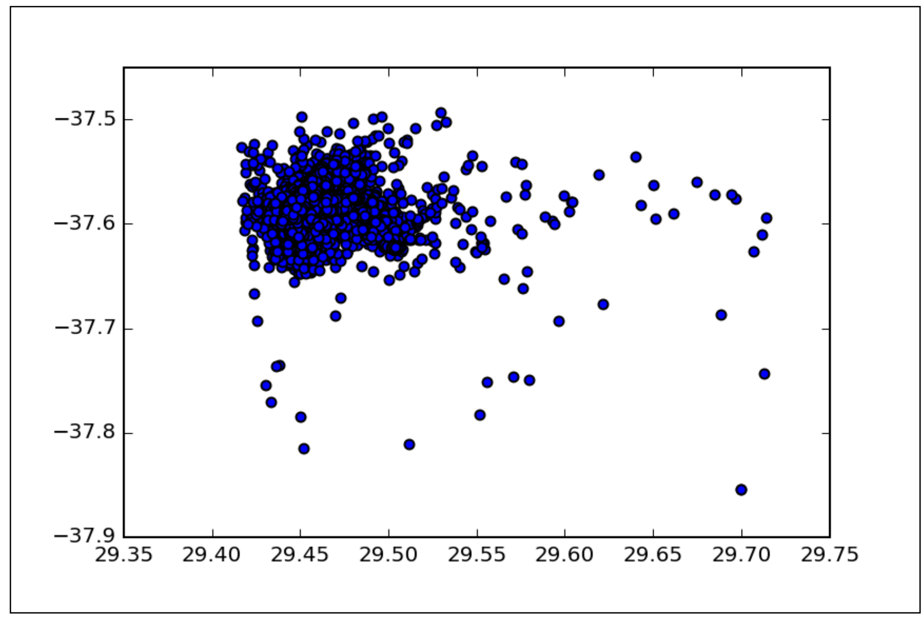
```
In [12]: px, py = lat_lng_to_pixels(p_lat, p_lng)
In [13]: px
```

```
Out[13]: 0          29.456688
         1          29.442916
         ...
         846943     29.450423
         846944     29.447998
         Name: pickup_longitude, dtype: float64
```

We will give more details about mathematical operations on pandas objects later in this chapter.

The matplotlib `scatter()` function takes two arrays with *x* and *y* coordinates as inputs. A **scatter plot** is a common 2D figure showing points with various positions, sizes, colors, and marker shapes. The following command displays all pickup locations:

```
In [14]: plt.scatter(px, py)
```

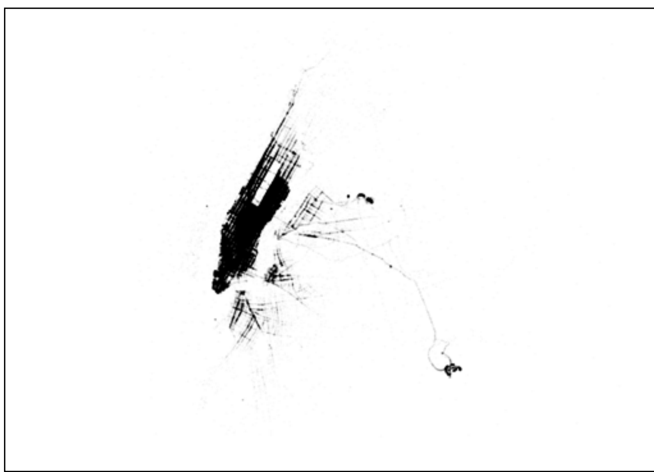


A scatter plot

Congratulations! You've made your first matplotlib plot. But it is not particularly appealing. First, the markers are too big. Second, there are too many points; we could make them a bit transparent to have a better idea of the distribution of the points. Third, we may want to zoom a bit more around Manhattan. Fourth, could we make this figure bigger? And finally, we don't necessarily need the axes here.

Fortunately, matplotlib is highly customizable, and all aspects of the plot can be changed, as shown here:

```
In [15]: plt.figure(figsize=(8, 6))
         plt.scatter(px, py, s=.1, alpha=.03)
         plt.axis('equal')
         plt.xlim(29.40, 29.55)
         plt.ylim(-37.63, -37.54)
         plt.axis('off')
```



A better scatter plot

That's already better! Let's explain these commands in more detail:

- The `figure()` function lets us specify the figure size (in inches).
- The `scatter()` function accepts many keyword arguments to customize the aspect of the scatter plot. Here:
    - We use a small marker size with the `s` keyword argument.
    - We use a small `alpha` opacity value: the points become nearly transparent, which emphasizes the regions with high density.
- We use an equal aspect ratio with `axis('equal')`.
- We zoom in by specifying the limits of the *x* and *y* axes with `xlim()` and `ylim()`.
- We remove the axes with `axes('off')`.

You will find the full list of options of `scatter()` in matplotlib's documentation at `http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.scatter`.

Descriptive statistics with pandas and seaborn

Common statistical quantities are one function call away in pandas. Here are a few examples:

```
In [16]: px.count(), px.min(), px.max()
Out[16]: (846945, 29.4171, 29.7143)
In [17]: px.mean(), px.median(), px.std()
Out[17]: (29.451345, 29.44941, 0.00976)
```

pandas also provides facilities for common statistical plots. These facilities leverage the matplotlib and seaborn libraries.

matplotlib is the main plotting package in Python. Although highly powerful and flexible, it sometimes requires a significant amount of manual tuning in order to generate clean, high-quality, publication-ready figures. Several projects aim to offer higher-level, simpler user interfaces for high-quality plotting. Seaborn is one of them, and we will use it in this subsection.

First, we need to install seaborn, as it is not currently installed by default in the Anaconda distribution. Fortunately, installing it is easy with conda. We can even perform the installation from the Notebook, as shown here:

```
In [18]: !conda install seaborn -q -y
```

Conda optional arguments

The optional arguments `-q -y` tell conda not to display the progress bar and not to ask for confirmation, respectively. More information is available at `http://conda.pydata.org/docs/commands/conda-install.html`.

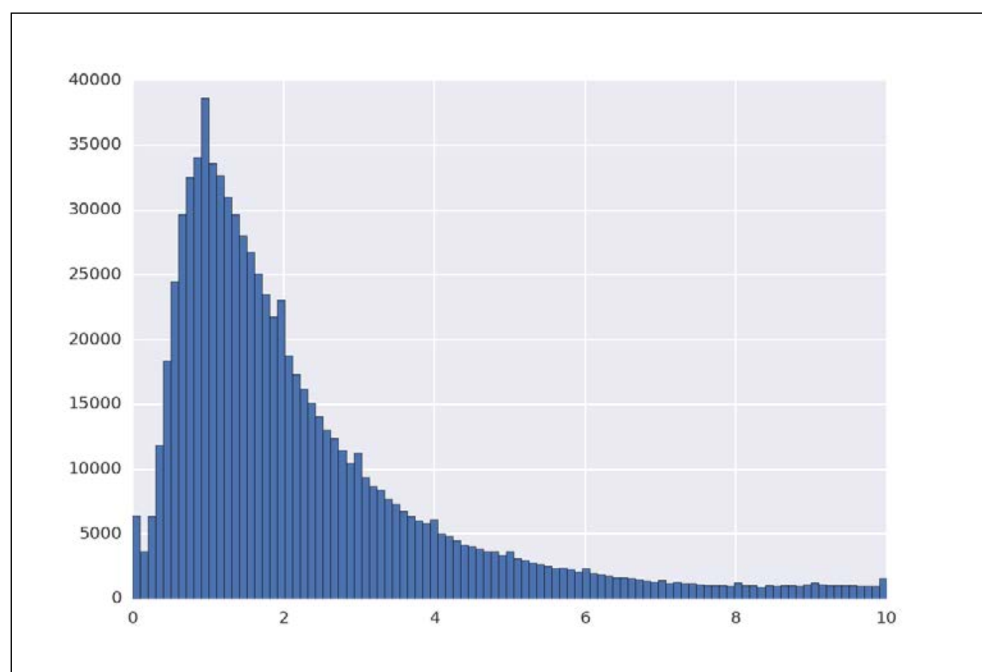
This command may take a while to complete, depending on your network connection. Let's check that seaborn has been correctly installed:

```
In [19]: import seaborn as sns
         sns.__version__
Out[19]: '0.6.0'
```

Importing seaborn automatically improves the aesthetics and color palettes of matplotlib figures. It also provides several easy-to-use statistical plotting functions.

Let's display a histogram of the trip distances. pandas provides a few simple plotting methods for `DataFrame` and `Series` objects. These methods are based on matplotlib, and benefit from the seaborn styling if seaborn has been imported. The `hist()` method displays a histogram of the values of a `Series` object. We can specify the histogram bins with the `bins` keyword argument. Here, we use NumPy's `linspace()` function to generate 100 linearly-spaced bins between 0 and 10:

```
In [20]: data.trip_distance.hist(bins=np.linspace(0., 10., 100))
```



A histogram with pandas, matplotlib, and seaborn

Here are a few references:

- Plotting with pandas at `http://pandas.pydata.org/pandas-docs/stable/visualization.html`
- Seaborn documentation at `http://stanford.edu/~mwaskom/software/seaborn/`
- Visualizing distributions with seaborn, at `http://stanford.edu/~mwaskom/software/seaborn/tutorial/distributions.html`

Manipulating data

Visualizing raw data and computing basic statistics is particularly easy with pandas. All we have to do is choose a couple of columns in a DataFrame and use built-in statistical or visualization functions.

However, more sophisticated data manipulations methods quickly become necessary as we explore a dataset. In this section, we will first see how to make selections of a DataFrame. Then, we will see how to efficiently make transformations and computations on columns.

We first import the NYC taxi dataset, as in the previous section.

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        %matplotlib inline
        data = pd.read_csv('data/nyc_data.csv',
                           parse_dates=['pickup_datetime',
                                        'dropoff_datetime'])
        fare = pd.read_csv('data/nyc_fare.csv',
                           parse_dates=['pickup_datetime'])
```

The `data` and `fare` DataFrames are now loaded in the notebook.

Selecting data

Our dataset contains almost one million rows. Only limited analyses can be done by using the whole dataset. More interesting discoveries can be made by looking at carefully-chosen subsets of the data. For example, what can we say about the taxi rides done on a particular day, a particular month, or a particular day of week? What about those starting or ending at a particular location? A significant part of real-world data analysis involves such fine-grained selections.

pandas offers many facilities for selecting a subset of columns or rows.

Selecting columns

First, let's select a few columns:

```
In [2]: data[['trip_distance', 'trip_time_in_secs']].head(3)
Out[2]:    trip_distance  trip_time_in_secs
        0           0.61                300
        1           3.28                960
        2           1.50                386
```

In Python, the square brackets `[]` are used for selecting elements in a list. The same notation is used by pandas to select columns. We need two pairs of brackets because pandas expects a list of columns to select, here `['trip_distance', 'trip_time_in_secs']`. The end-result is a new DataFrame containing just two columns instead of 14.

This is about all you need to know about selecting columns. There is much more to say about selecting rows.

Selecting rows

Rows of a DataFrame are *indexed*: every row comes with a unique *label* (or *index*). Often, this label is just an integer between `0` and `n_rows-1`. In some situations, this label can be something else, like a string. If we had a DataFrame giving information about each taxi, the label could be the taxi's medallion (a unique identifier for NYC's taxicabs), or an anonymized version of it.

The `loc` attribute of a DataFrame is used to select row(s) from their labels. Here, we select the first row:

```
In [3]: data.loc[0]
Out[3]: medallion            76942C3205E17D7E7FE5A9F709D16434
        hack_license         25BA06A87905667AA1FE5990E33F0E2E
        vendor_id                                         VTS
        rate_code                                           1
        store_and_fwd_flag                                NaN
        pickup_datetime                   2013-01-01 00:00:00
        dropoff_datetime                  2013-01-01 00:05:00
        passenger_count                                     3
```

```
        trip_time_in_secs                                  300
        trip_distance                                     0.61
        pickup_longitude                              -73.95592
        pickup_latitude                                40.78189
        dropoff_longitude                             -73.96318
        dropoff_latitude                               40.77783
        Name: 0, dtype: object
```

Multiple rows can be selected by providing a list of labels:

```
In [4]: data.loc[[0, 100000]]
```

	medallion	hack_license	vendor_id	rate_code	store_and_fwd_flag	pickup_d
0	76942C3205E17D7E7FE5A9F709D16434	25BA06A87905667AA1FE5990E33F0E2E	VTS	1	NaN	2013-01-(00:00:00
100000	7461F7106D33D3A5775F4245724606FD	BACEA353BB4106A005BB7836BDCAC0C3	VTS	1	NaN	2013-02-1 18:10:00

Selecting multiple rows

We can also select regularly spaced rows using *slices*. For example, here is how to select one row out of 10 between rows 1000 and 2000:

```
In [5]: data.loc[1000:2000:10,
                  ['trip_distance', 'trip_time_in_secs']]
Out[5]:       trip_distance  trip_time_in_secs
        1000           1.00                441
        1010           3.80                691
        ....
        1990           0.13                 60
        2000           9.60                963
```

Note how we combined column and row selection here. Two expressions can be passed to `loc`: the row selection first, and the column selection second (the two expressions are separated by a comma).

`loc` expects actual labels and, unlike normal Python slices, the start and end points are both inclusive! Also, we could have used `iloc` instead of `loc` to specify index positions rather than labels.

Filtering with boolean indexing

Instead of selecting rows by labels, we can also select rows satisfying specific properties. This is a more common use-case in data analysis.

For example, let's select the longest rides:

```
In [6]: data.loc[data.trip_distance>50]
```

dropoff_datetime	passenger_count	trip_time_in_secs	trip_distance	pickup_longitude	pickup_latitude	dropoff_longitude
2013-01-01 21:56:37	1	934	52.20	-73.979576	40.743626	-73.941902
2013-01-04 07:17:14	1	1973	96.30	-73.959785	40.762497	-73.962440
2013-01-05 02:23:01	1	1913	52.90	-74.006119	40.735157	-73.958694
2013-01-12 03:24:47	1	1312	66.20	-73.966873	40.683315	-73.916885

Long taxi rides

Here, `data.trip_distance>50` is a Series object containing boolean values for all rows, depending on whether the trip distance is higher or lower than 50. The `loc` attribute also works with booleans instead of explicit labels: it will return all rows represented by a `True` boolean value.

We might want to choose the distance threshold depending on certain conditions. For example, we might want to keep the 1% longest trips. Here, let's show how the IPython widgets can help us do that (this isn't the only method, of course).

We create a slider displaying the number of rows with a distance larger than the threshold:

```
In [7]: from ipywidgets import interact
In [8]: @interact
        def show_nrows(distance_threshold=(0, 200)):
            return len(data.loc[data.trip_distance >
                                distance_threshold])
```

```
distance_threshold                                26
307
```

A slider to select long rides

More selection, indexing, and filtering facilities are described in pandas' documentation. Here are a few references:

- `http://pandas.pydata.org/pandas-docs/stable/dsintro.html`
- `http://pandas.pydata.org/pandas-docs/stable/indexing.html`

Computing with numbers

The `trip_time_in_secs` column contains the trip durations in seconds. How can we convert these values to minutes? More generally, how can we make computations on DataFrames?

A first approach would be to use a `for` loop, iterating over all rows and making numerical computations successively inside that loop. This is what people with a background in the C programming language tend to do when they start to learn Python. However, this isn't the best way to do things in Python.

Whereas Python loops are possible in this situation, they would be extremely slow. For this reason, they should be avoided as much as possible. We will discuss this issue in the next chapter. In the meantime, there are much better, faster, and actually *simpler* alternatives.

pandas allows you to perform vector operations on DataFrame and Series objects. These operations are quite natural, because they follow standard mathematical notations. For example, let's add a new column containing the trip durations in minutes:

```
In [9]: data['trip_time_in_mins'] = data.trip_time_in_secs / 60.0
In [10]: data[['trip_time_in_secs', 'trip_time_in_mins']].head(3)
Out[10]:    trip_time_in_secs  trip_time_in_mins
         0                300           5.000000
         1                960          16.000000
         2                386           6.433333
```

Let's explain this in more detail. The `data.trip_time_in_secs` notation represents a Series object. The `/` symbol represents floating-point division in Python 3. It normally works with numbers only. However, pandas extends this operator to work with Series and DataFrames as well, in which case it automatically operates on all elements. Here, all elements of `data.trip_time_in_secs` are divided by 60.

The same notation would also work if we had another Series object of the same size in the second term. In that case, the division would occur on an element-wise basis (the first item in the Series on the left divided by the first in the right, the second by the second, and so on).

A Series object is a vector with indices (or labels). The indices determine which values are used when operating Series objects together. Here is an example:

```
In [11]: a = data.trip_distance[:5]
         a
Out[11]: 0    0.61
         1    3.28
         2    1.50
         3    0.00
         4    1.31
         Name: trip_distance, dtype: float64
In [12]: b = data.trip_distance[2:6]
         b
Out[12]: 2    1.50
         3    0.00
         4    1.31
         5    5.81
         Name: trip_distance, dtype: float64
```

These two Series objects have different but overlapping sets of indices. Although they don't have the same size, we can add them together:

```
In [13]: a + b
Out[13]: 0     NaN
         1     NaN
         2    3.00
         3    0.00
         4    2.62
         5     NaN
         Name: trip_distance, dtype: float64
```

The result is a new Series object containing the *aligned* sum of `a` and `b`. The set of indices of `a + b` is the *union* of the indices of `a` and `b`. When one value is missing, we get an operation with an undefined value, which is `NaN` (Not a Number). When the indices overlap, the sum is correctly computed. This feature - **alignment** - makes it quite convenient to operate on labeled data. You'll find more information at `http://pandas.pydata.org/pandas-docs/stable/basics.html`.

Other mathematical operations (`+`, `*`, etc.) work similarly. Further, NumPy implements many mathematical functions like `np.log()` and `np.sin()`; they not only work on scalar numbers but also on Series and DataFrames. This is called **vectorization**, because this concept relates to mathematical operations performed on *vectors*. We will discuss this concept in greater details in *Chapter 3*, *Numerical Computing with NumPy*.

Working with text

Efficient vectorized operations can also be done on text. Let's have a look at `data.medallion`:

```
In [14]: data.medallion.head(3)
Out[14]: 0    76942C3205E17D7E7FE5A9F709D16434
         1    517C6B330DBB3F055D007B07512628B3
         2    ED15611F168E41B33619C83D900FE266
         Name: medallion, dtype: object
```

This column contains anonymized versions of the taxis' medallions. The `str` attribute gives us access to many vectorized string processing functions. Here, for example, we extract the first four characters of every medallion:

```
In [15]: data.medallion.str.slice(0, 4).head(3)
Out[15]: 0    7694
         1    517C
         2    ED15
         Name: medallion, dtype: object
```

There are many other functions, including ones that apply regular expressions on all rows. Together, these functions are essential when you're working with text data, particularly when you have datasets so large that `for` loops would be too slow. You will find the full list of string methods at `http://pandas.pydata.org/pandas-docs/stable/text.html`.

Working with dates and times

pandas provides many methods to operate on dates and times. Common operations include:

- getting the day, day of week, hour, or any other quantity from dates
- selecting ranges of dates
- computing time ranges
- dealing with different time zones

These operations only work on Series with a `datetime64` data type, or with `DatetimeIndex` objects (used to index values with dates or times). In practice, there are many ways to get such objects from raw data like CSV files. Here, we used the `parse_dates` keyword arguments in the `pd.read_csv()` function. Among the other methods, let's mention the `pd.to_datetime()` function. You will find more details in the references below.

The `dt` attribute of datetime objects gives us access to datetime components. For example, here is how to get the day of the week of the taxi trips (Monday=0, Sunday=6):

```
In [16]: data.pickup_datetime.dt.dayofweek[::200000]
Out[16]: 0         1
         200000    6
         400000    5
         600000    0
         800000    1
         dtype: int64
```

Here is a more complex example. Let's select all night trips that finished the next day:

```
In [17]: day_p = data.pickup_datetime.dt.day
         day_d = data.dropoff_datetime.dt.day
         selection = (day_p != day_d)
         print(len(data.loc[selection]))
         data.loc[selection].head(3)
Out[17]: 7716
```

	vendor_id	rate_code	store_and_fwd_flag	pickup_datetime	dropoff_datetime	passenger_count	trip_time_in_secs	trip_distance
ED	VTS	1	NaN	2013-01-01 23:45:00	2013-01-02 00:03:00	1	1080	12.61
95	CMT	1	N	2013-01-01 23:46:22	2013-01-02 00:28:01	1	2498	16.10
49	CMT	1	N	2013-01-01 23:46:53	2013-01-02 00:03:33	1	1000	5.40

Night trips

Like in the *Computing with numbers* subsection, the `(day_p != day_d)` expression is a Series of booleans for selecting the rows that have different pickup and dropoff days. Python's inequality symbol `!=` is understood by pandas as a vectorized operator working on a row-by-row basis.

Here are a few references about date and time operations:

- `http://pandas.pydata.org/pandas-docs/stable/timeseries.html`
- `http://pandas.pydata.org/pandas-docs/stable/timedeltas.html`

Handling missing data

We finish this part with a short discussion about missing data. Real-world datasets are rarely perfect, and having missing values in a dataset is the rule rather than the exception. Fortunately, pandas perfectly handles missing data.

In practice, you can consider letting pandas deal seamlessly with missing data while you manipulate and operate on data. However, there are times when you want control over these missing values. For example, you may want to discard missing data from some analysis. Or, you may want to replace missing data with a default value.

In pandas, missing data is represented by `NaN` (Not a Number) or `None`. pandas provides several Series and DataFrame methods to deal with missing data, notably:

- `isnull()` indicates whether values are null or not
- `notnull()` indicates the opposite
- `dropna()` removes missing data
- `fillna(some_default_value)` replaces missing data with a default value

You will find more details at `http://pandas.pydata.org/pandas-docs/stable/missing_data.html`.

Complex operations

We've seen how to load, select, filter, and operate on data with pandas. In this section, we will show more complex manipulations that are typically done on full-blown databases based on SQL.

SQL

Structured Query Language is a domain-specific language widely used to manage data in **relational database management systems** (**RDBMS**). pandas is somewhat inspired by SQL, which is familiar to many data analysts. Additionally, pandas can connect to SQL databases. You will find more information about the links between pandas and SQL at `http://pandas.pydata.org/pandas-docs/stable/comparison_with_sql.html`.

Let's first import our NYC taxi dataset as in the previous sections.

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import seaborn
        %matplotlib inline
        data = pd.read_csv('data/nyc_data.csv',
                           parse_dates=['pickup_datetime',
                                        'dropoff_datetime'])
        fare = pd.read_csv('data/nyc_fare.csv',
                           parse_dates=['pickup_datetime'])
```

Group-by

A *group-by* operation typically consists of one or several of the following steps:

- splitting the data into groups that share common attributes
- applying a function to every group
- recombining the results

Many operations that seem particularly complex are actually group-by operations. pandas provides user-friendly facilities to perform these manipulations. We will illustrate them here.

Let's have a look at the weekly statistics in our dataset. We first need to split the data into weekly groups. pandas provides the `groupby()` method for this purpose, as shown here:

```
In [2]: weekly = data.groupby(data.pickup_datetime.dt.weekofyear)
In [3]: len(weekly)
Out[3]: 52
```

Here, `data.pickup_datetime.dt.weekofyear` is a Series instance with the week number of every ride. The `groupby()` method returns an object with one group per value of `weekofyear`. Since there are 52 different weeks in the year, `weekly` contains 52 different groups.

The `size()` method returns the number of rows in each group, as shown here:

```
In [4]: y = weekly.size()
        y.head(3)
Out[4]: 1    17042
        2    15941
        3    17017
        dtype: int64
```

We'll now plot the number of rides per week. To create a meaningful plot, we first need to specify the appropriate x-axis with the dates of all 52 weeks:

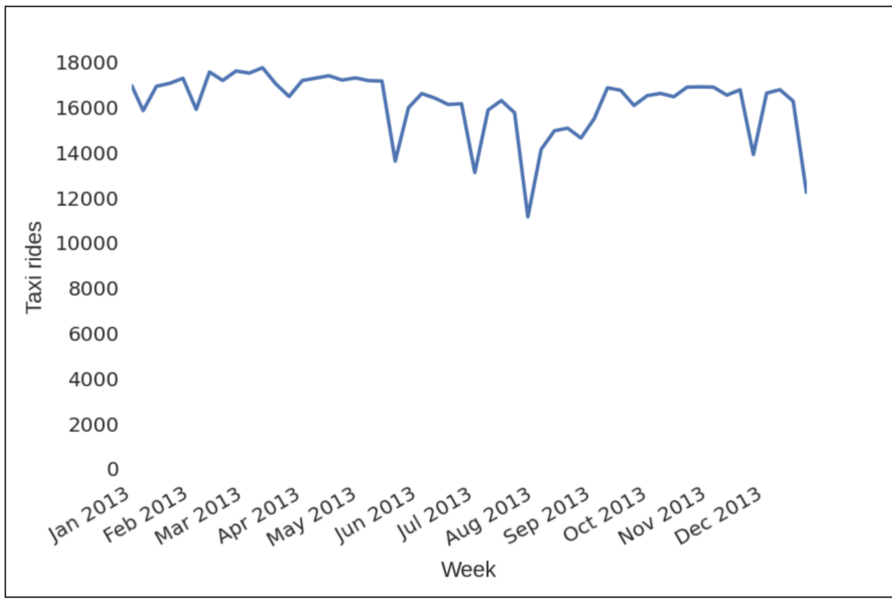
```
In [5]: x = weekly.pickup_datetime.first()
        x.head(3)
Out[5]: 1   2013-01-01 00:00:00
        2   2013-01-07 00:03:00
        3   2013-01-14 00:00:51
        Name: pickup_datetime, dtype: datetime64[ns]
```

This Series contains the date of the first item in every row.

Finally, we create a new Series with the values in our `y` object, and indexed by the dates `x` (we need to use `.values` in order to discard `y`'s indices, since we use `x`'s indices instead). The `plot()` method of this new Series creates the plot we want:

```
In [6]: pd.Series(y.values, index=x).plot()
        plt.ylim(0)  # Set the lower y value to 0.
        plt.xlabel('Week')  # Label of the x axis.
        plt.ylabel('Taxi rides')  # Label of the y axis.
```

Here is the result (let's not forget that our dataset only contains a small fraction of all rides):



Number of taxi rides per week

We'll see more examples in the next subsection.

More information about the powerful group-by operation in pandas can be found at `http://pandas.pydata.org/pandas-docs/stable/groupby.html`.

Joins

Joins are common operations in relational databases. The idea is to combine several tables together, based on common values shared between the tables.

In the current example, we have two DataFrames, `data` and `fare`, both with the same number of rows (one row per trip). First, from the `fare` DataFrame, we will get the average tip obtained by each taxi. Then, we'll inject this information into the `data` DataFrame.

For the first step, we use `groupby()` again:

```
In [7]: tip = fare[['medallion', 'tip_amount']] \
               .loc[fare.tip_amount>0].groupby('medallion').mean()
        print(len(tip))
        tip.head(3)
Out[7]: 13407
                                          tip_amount
        medallion
        00005007A9F30E289E760362F69E4EAD    1.815854
        000318C2E3E6381580E5C99910A60668    2.857222
        000351EDC735C079246435340A54C7C1    2.099111
```

Here, we considered a reduced DataFrame with just the `medallion` and `tip_amount` columns, removed the trips where the passenger did not tip, grouped by taxi's medallion (a unique identifier for the taxis), and took the mean of this grouped DataFrame. This new DataFrame contains one column `tip_amount` and is indexed by the medallion. It contains 13407 rows: this corresponds to the number of different taxis in our dataset.

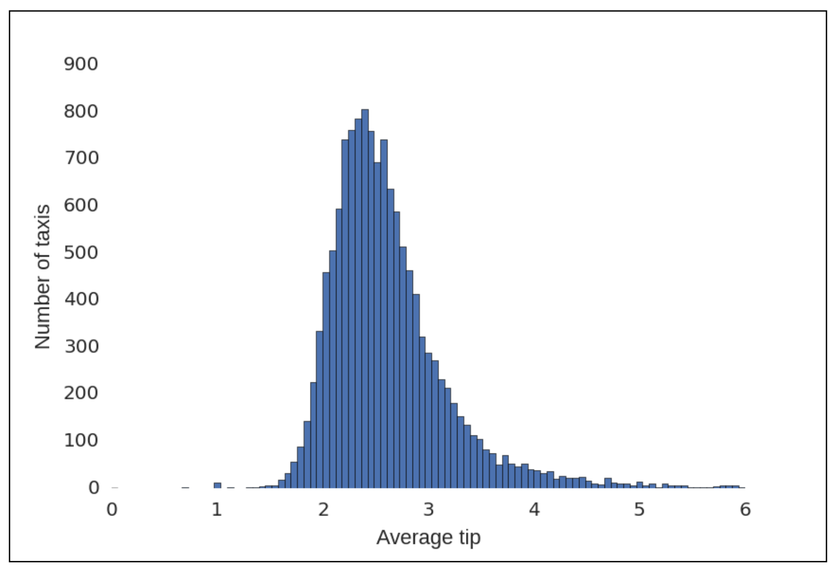
Chaining syntax

Note how we used the chaining syntax here to perform several operations successively on the `fare` DataFrame (the `obj.fun1().fun2().fun3()` pattern). Each of these operations in pandas returns a new DataFrame. The dot . character applies the next operation to the previous operation's result and forms a concise and readable syntax for chained operations. See also the `pipe()` function at `http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.pipe.html`.

www.allitebooks.com

Let's plot a histogram of these average tips:

```
In [8]: tip.hist(bins=np.linspace(0., 6., 100))
        plt.xlabel('Average tip')
        plt.ylabel('Number of taxis')
```



Average tips earned by taxis

The next step is to reinject this `tip` DataFrame into the `data` DataFrame. The `medallion` column appears in both of our datasets; by identifying this special field (also called the *key*) in both datasets, we can associate every row in `tip` to a row in `data`. This operation is called a **join** in SQL.

We can use the `merge()` function here:

```
In [9]: data_merged = pd.merge(data, tip, how='left',
                               left_on='medallion', right_index=True)
        data_merged.head(3)
```

trip_distance	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	tip_amount
0.61	-73.955925	40.781887	-73.963181	40.777832	3.180417
3.28	-74.005501	40.745735	-73.964943	40.755722	2.863235
1.50	-73.969955	40.799770	-73.954567	40.787392	2.147143

Result of a merge operation

Let's explain how this works:

- We specify the left and right DataFrames to perform the join on.
- There are several types of joins; we choose a *left* join here because we want to keep the keys from the left DataFrame `data`.
- We then specify where to find this key in the left and right DataFrames.
    - On the left, we use the `medallion` column.
    - On the right, we use the index, because the `tip` DataFrame is indexed by the medallion.

The end product is a new DataFrame similar to our original `data` DataFrame, but with an additional column containing the average tip received by the taxi.

Joins and merges form a rich and complex topic. You will find more information at `http://pandas.pydata.org/pandas-docs/stable/merging.html`.

Finally, here are a few more advanced topics in pandas that are worth exploring:

- Features for time series data at `http://pandas.pydata.org/pandas-docs/stable/timeseries.html`
- Support for categorical variables at `http://pandas.pydata.org/pandas-docs/stable/categorical.html`
- Pivot tables (particularly useful when dealing with high-dimensional data) at `http://pandas.pydata.org/pandas-docs/stable/reshaping.html`

Summary

In this chapter, we covered the basics of data analysis with pandas: loading a dataset, selecting rows and columns, grouping and aggregating quantities, and performing complex operations efficiently.

The next natural step is to conduct statistical analyses: hypothesis testing, modeling, predictions, and so on. Several Python libraries provide such functionality beyond pandas: SciPy, statsmodels, PyMC, and more. The *IPython Cookbook* contains many advanced examples of such analyses.

In the next chapter, we will introduce NumPy, the library underlying the entire SciPy ecosystem.

3
Numerical Computing with NumPy

NumPy is the library that underlies the entire SciPy/PyData ecosystem. NumPy provides a multidimensional array data type that is widely used in numerical computing.

In this chapter, we will use NumPy on data analysis and scientific modeling examples, covering the following topics:

- A primer to vector computing
- Creating and loading arrays
- Basic array manipulations
- Computing with NumPy arrays

A primer to vector computing

Vector computing is about efficiently performing mathematical operations on numerical arrays. Many problems in science and engineering actually consist of a sequence of such operations.

This section introduces and demonstrates the multidimensional array data type for numerical computing.

Multidimensional arrays

What is a multidimensional array? Consider a vector containing 1000 real numbers. It has one dimension, since numbers are stored along a single axis. Now, consider a matrix with 1000 rows and 1000 columns. It contains 1,000,000 numbers. Because it has two dimensions, you need to specify both the row and column to refer to a specific number.

More generally, an n-dimensional array, also called **ndarray**, is an n-dimensional matrix (or tensor). Every number is identified by n indices `(i_1, ... i_n)`.

Many types of real-world data can be represented as ndarrays:

- The evolution of a stock exchange price is a 1D array (vector) with one value per day (or per hour, per week, etc.).
- A grayscale image is a `(height, width)` 2D array, with one light intensity per pixel.
- The evolution of a **Partial Differential Equation** (**PDE**) on a 2D grid can be represented as a `(n, m, duration)` 3D array.
- A video is a `(height, width, n_channels, duration)` 4D array, where typically `n_channels=3` for the three RGB (red, green, blue) color components.

It is quite rare to work on arrays with more than 4 dimensions.

Originally, Python didn't provide an adequate structure for representing an ndarray. This is the main goal of a scientific Python library created in the early 2000s called NumPy, which traces its roots back to several years before.

The ndarray

An ndarray is essentially defined by:

- a number of **dimensions**
- a **shape**
- **strides**
- a **data type**, or **dtype**
- the actual data

We already explained the notion of dimension for a numerical array. The **shape** is the length of every axis. For example, the shape of a video would be `(height, width, n_channels, duration)`. There are four elements in this tuple because there are four dimensions in the array. Strides will be explained later in this chapter.

Here is a schematic representation of the structure of a 3D ndarray:

Structure of an ndarray

In an array, all elements must have the same data type. The most common data types are integers, floating-point numbers, booleans, and strings.

You can also define custom data types for **structured arrays** (also called **record arrays**). These are arrays of structs (meant as *C structs*). A typical use-case occurs when you want to load a flat binary file in a complex format. You will find more information at `http://docs.scipy.org/doc/numpy/user/basics.rec.html`.

Vector operations on ndarrays

NumPy not only provides an ndarray structure for *storing* numerical data, but it also implements fast mathematical operations on ndarrays. The ability to perform highly-efficient operations on ndarrays is one of the major advantages of this structure.

Many operations on arrays follow the same pattern where an elementary mathematical operation is performed on an element-wise basis on two arrays. For example, the sum `C=A+B` of two matrices contains the sums of all corresponding pairs of elements in `A` and `B`: `C_ij = A_ij + B_ij`. Mathematically, this corresponds to operations on vectors and matrices. These are called **vector (or vectorized) operations**. NumPy offers fast implementations of many vector operations.

Another advantage of NumPy is the brevity of the syntax for array operations. Whereas a language like C or Java would require us to write a loop for a matrix operation as simple as `C=A+B`, NumPy allows us to simply write `C=A+B`. More generally, many complex operations can be concisely written in a few lines of NumPy, whereas they would involve tens of lines in another language.

How fast are vector computations in NumPy?

One of the most important take-home messages of this chapter is that vector operations on ndarrays are much faster than Python loops. Most numerical computations should be done with vector operations in NumPy instead of Python loops.

Let's illustrate this particularly important point by showing two ways of computing the sum of two vectors: in pure Python, and with NumPy.

Let's first create two vectors containing 1,000,000 random numbers each. We use the native `random` module in a list comprehension:

```
In [1]: from random import random
        list_1 = [random() for _ in range(1000000)]
        list_2 = [random() for _ in range(1000000)]
```

We compute the sum of these vectors with another list comprehension. The `zip()` built-in function allows us to loop over the two vectors simultaneously, as shown here:

```
In [2]: out = [x + y for (x, y) in zip(list_1, list_2)]
        out[:3]
Out[2]: [0.843375384328939, 1.507485612134079, 1.4119777108063973]
```

How long does this operation take? Let's use IPython's `%timeit` magic command to find it out:

```
In [3]: %timeit [x + y for (x, y) in zip(list_1, list_2)]
Out[3]: 10 loops, best of 3: 69.7 ms per loop
```

Now, we perform the same operation with NumPy:

```
In [4]: import numpy as np
        arr_1 = np.array(list_1)
        arr_2 = np.array(list_2)
```

The `np.array()` function can convert a Python list into an ndarray (we'll cover this in more detail in the next section). Although `list_1` and `arr_1` contain the same data, they don't have the same data type:

```
In [5]: type(list_1), type(arr_1)
Out[5]: (list, numpy.ndarray)
In [6]: arr_1.shape
Out[6]: (1000000,)
In [7]: arr_1.dtype
Out[7]: dtype('float64')
```

Computing the sum of the two arrays is particularly easy with NumPy; the + character directly works with ndarrays of the same shape:

```
In [8]: sum_arr = arr_1 + arr_2
        sum_arr[:3]
Out[8]: array([ 0.84337538,  1.50748561,  1.41197771])
```

How much faster is NumPy over pure Python here?

```
In [9]: %timeit arr_1 + arr_2
Out[9]: 1000 loops, best of 3: 1.57 ms per loop
```

This is about 45 times faster.

Generally speaking, getting one or several orders of magnitude of speed improvements between pure Python and NumPy is not uncommon. We'll explain the technical reasons of this below. In the meantime, just remember that **vectorized operations with NumPy are much faster than Python loops**. Every time you are tempted to write a Python loop, see if you can use NumPy instead.

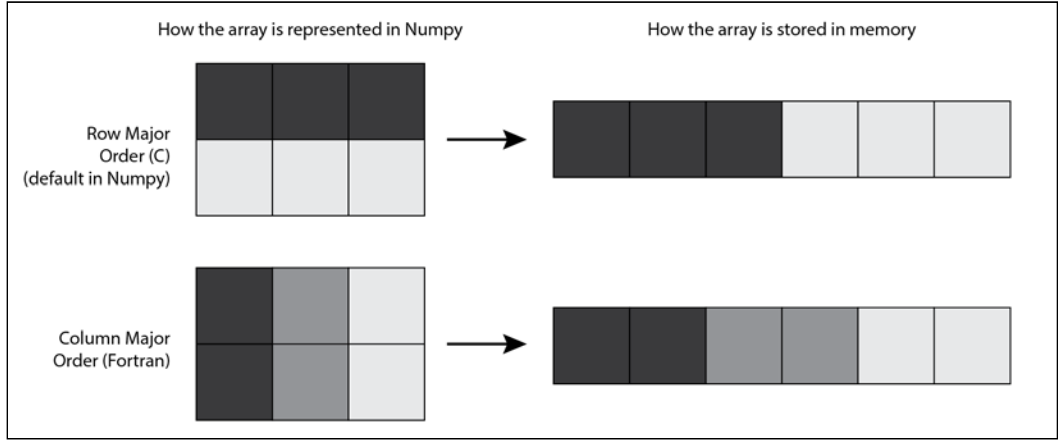
How an ndarray is stored in memory

Let's briefly discuss the internals of NumPy. Although beginners can probably skip this, knowing these details can help you write more efficient code with NumPy.

Internally, an ndarray consists of some metadata about the array's structure, and the actual binary data. The data is stored in a *contiguous* block of memory. For example, the data of a vector containing 10 elements of double-precision floating-point numbers (`float64` dtype, where each number is encoded in 8 bytes) is stored in a contiguous block of 80 bytes.

With this information, you can calculate the memory requirements for an ndarray. For example, a `(10,000, 10,000)` float64 array requires `10,000*10,000*8` bytes, which is about 763 MB of memory. **When working with large arrays, check your available memory to avoid running out of RAM and crashing your computer.**

When there is more than one dimension, there are several ways of storing the elements in the memory block. With a matrix, the elements can be stored in **row-major order** (also known as **C-order**) or **column-major order** (also known as **Fortran-order**). The distinction pertains to which axis among the row or the column moves the fastest as one goes along all elements in the data buffer. The default order in NumPy is the C-order, although this can be configured differently.



C-major and Fortran-order layout

This notion generalizes to multidimensional arrays with the notion of **strides**. Strides describe how the elements of a multidimensional array are organized within the data buffer. NumPy implements a *strided indexing scheme*, where the position of any element is a linear combination of the element's indices, the coefficients being the strides. In other words, strides describe, in any axis, how many bytes to jump over in the data buffer to go from one item to the next along that axis.

Here are a few references:

- Documentation on strides at `http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html#internal-memory-layout-of-an-ndarray`
- Advanced NumPy in the SciPy lectures notes at `http://www.scipy-lectures.org/advanced/advanced_numpy/`
- Getting the best performance out of NumPy, an *IPython Cookbook* recipe, at `http://ipython-books.github.io/featured-01/`
- Blog post by Eli Bendersky at `http://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays/`

Why operations on ndarrays are fast

Let's compare the additions on the lists and arrays in the previous example.

Python being a dynamic interpreted language, a `for` loop involves many low-level operations of the CPython interpreter. When there are many iterations, this overhead takes significantly more time than the actual addition.

By contrast, in NumPy, vector operations are implemented in C, which is a more lower-level language than Python. This implementation leads to far fewer CPU instructions than the Python loop. Knowing the address of the memory block and the data type, it is just simple arithmetic to loop over all items.

In a Python list, elements are stored at arbitrary locations in memory, whereas in an array, elements are stored within a contiguous block of memory. CPUs are more efficient at loading consecutive bytes from memory. This is called **sequential locality**.

Further, NumPy can take advantage of the vectorized instructions of modern CPUs, such as Intel's SSE and AVX, AMD's XOP, and so on. For example, multiple consecutive floating-point numbers can be loaded in 128, 256, or 512-bit registers for vectorized arithmetical computations implemented as CPU instructions.

NumPy can also be linked to highly-optimized linear algebra libraries such as BLAS and LAPACK through ATLAS or the Intel **Math Kernel Library** (**MKL**). Finally, a few specific matrix computations, including the matrix product `np.dot()`, may be multithreaded to take advantage of multicore processors.

All of these reasons explain why NumPy is so much faster than Python loops on vector operations.

Creating and loading arrays

In this section, we will see how to create and load NumPy arrays.

Creating arrays

First, there are several NumPy functions for creating common types of arrays. For example, `np.zeros(shape)` creates an array containing only zeros. The `shape` argument is a tuple giving the size of every axis. Hence, `np.zeros((3, 4))` creates an array of size `(3, 4)` (note the double parentheses, because we pass a tuple to the function).

Here are some further examples:

```
In [1]: import numpy as np
        print("ones", np.ones(5))
        print("arange", np.arange(5))
        print("linspace", np.linspace(0., 1., 5))
        print("random", np.random.uniform(size=3))
        print("custom", np.array([2, 3, 5]))
Out[1]: ones [ 1.  1.  1.  1.  1.]
        arange [0 1 2 3 4]
        linspace [ 0.    0.25  0.5   0.75  1.  ]
        random [ 0.68361911  0.33585308  0.70733934]
        custom [2 3 5]
```

The `np.arange()` and `np.linspace()` functions create arrays with regularly spaced numbers. The `np.random` module contains many functions for generating arrays containing independent (pseudo)-random values following various distributions (uniform, exponential, Gaussian, and many others).

The versatile `np.array()` function converts Python objects like lists or tuples into NumPy arrays. It is also used to create small arrays by specifying their values directly, as shown here:

```
In [2]: np.array([[1, 2], [3, 4]])
Out[2]: array([[1, 2],
               [3, 4]])
```

Every array has a fixed data type. You can specify the data type explicitly, or you can let NumPy figure out the data type automatically. For example, `np.ones()` generates an array of floating-point numbers by default, whereas `np.arange()` returns an array of integers. You can specify the data type explicitly as shown here:

```
In [3]: np.ones(5, dtype=np.int64)
Out[3]: array([1, 1, 1, 1, 1])
In [4]: np.arange(5).astype(np.float64)
Out[4]: array([ 0.,  1.,  2.,  3.,  4.])
```

The `astype()` method converts an array to any other type.

Here are a few references:

- Array creation routines at `http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html`
- NumPy random functions at `http://docs.scipy.org/doc/numpy/reference/routines.random.html`
- Data type objects at `http://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html`
- Data types at `http://docs.scipy.org/doc/numpy/user/basics.types.html`

Loading arrays from files

The `np.load()` and `np.save()` functions allow you to import and export NumPy arrays from/to binary files in a custom format.

Text and binary files can be imported into NumPy arrays. The `np.fromfile()` and `np.fromstring()` functions load arrays from binary/text files or strings. The `np.loadtxt()` and `np.genfromtxt()` functions load arrays from text files, including CSV files. For loading CSV files, text, or some other heterogeneous data, pandas is generally more effective than NumPy. Internally, pandas is based on NumPy. Therefore, data can be easily exchanged between pandas and NumPy structures. Here is an example:

```
In [5]: import pandas as pd
```

Let's load the NYC taxi dataset from *Chapter 2*, *Interactive Data Analysis with pandas*:

```
In [6]: data = pd.read_csv('../chapter2/data/nyc_data.csv')
```

Going from pandas to NumPy is particularly easy: just use the `.values` attribute, available on all DataFrame and Series objects. More specifically:

- A Series corresponds to a 1D NumPy array.
- A DataFrame corresponds to a 2D NumPy array.
- A Panel corresponds to a 3D NumPy array (we won't cover this pandas structure here).

Here, we obtain a `(N, 2)` NumPy array with the pickup coordinates of all trips:

```
In [7]: pickup = data[['pickup_longitude', 'pickup_latitude']].values
        pickup
Out[7]: array([[-73.955925,  40.781887],
               [-74.005501,  40.745735],
               ...,
               [-73.978477,  40.772945],
               [-73.987206,  40.750568]])
In [8]: pickup.shape
Out[8]: (846945, 2)
```

Here are a few references:

- Links between NumPy and pandas data structures at `http://pandas.pydata.org/pandas-docs/stable/dsintro.html#dataframe-interoperability-with-numpy-functions`
- Input/output routines at `http://docs.scipy.org/doc/numpy/reference/routines.io.html`

Basic array manipulations

Let's see some basic array manipulations around multiplication tables.

```
In [1]: import numpy as np
```

We first create an array of integers between 1 and 10, as shown here:

```
In [2]: x = np.arange(1, 11)
In [3]: x
Out[3]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Note that in `np.arange(start, end)`, `start` is included while `end` is excluded.

To create our multiplication table, we first need to transform `x` into a row and column vector. Our vector `x` is a 1D array, whereas row and column vectors are 2D arrays (also known as matrices). There are many ways to transform a 1D array to a 2D array. We will see the two most common methods here.

The first method is to use `reshape()`:

```
In [4]: x_row = x.reshape((1, -1))
        x_row
Out[4]: array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]])
```

The `reshape()` method takes the new shape as parameter. The total number of elements must be unchanged. For example, reshaping a `(2, 3)` array to a `(5,)` array would raise an error. The number `-1` can be used to tell NumPy to figure out automatically the size of that axis.

Here, note the double square brackets, indicating that `x_row` is a 2D array with just one row, while `x` was a 1D array.

In NumPy, the first axis is vertical, while the second axis is horizontal. However, 1D arrays are displayed horizontally, which is slightly confusing.

Another reshaping method is to use a special indexing syntax in NumPy:

```
In [5]: x_col = x[:, np.newaxis]
        x_col
Out[5]: array([[ 1],
               [ 2],
               [ 3],
               [ 4],
               [ 5],
               [ 6],
               [ 7],
               [ 8],
               [ 9],
               [10]])
```

Here, the colon `:` is used to select the entire first axis (vertical), whereas `np.newaxis` is used to create a new second axis (horizontal) with just one item.

We can now create our multiplication table. A first possibility would be to create an empty `(10, 10)` array and fill it with two `for` loops. However, doing it with NumPy leads to faster and much more concise code.

We can use `np.dot()` to compute a matrix product between two vectors:

```
In [6]: np.dot(x_col, x_row)
Out[6]: array([[  1,   2,   3,   4,   5,   6,   7,   8,   9,  10],
               [  2,   4,   6,   8,  10,  12,  14,  16,  18,  20],
               [  3,   6,   9,  12,  15,  18,  21,  24,  27,  30],
               [  4,   8,  12,  16,  20,  24,  28,  32,  36,  40],
               [  5,  10,  15,  20,  25,  30,  35,  40,  45,  50],
               [  6,  12,  18,  24,  30,  36,  42,  48,  54,  60],
               [  7,  14,  21,  28,  35,  42,  49,  56,  63,  70],
               [  8,  16,  24,  32,  40,  48,  56,  64,  72,  80],
               [  9,  18,  27,  36,  45,  54,  63,  72,  81,  90],
               [ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100]])
```

Since `x_col` is a `(10, 1)` array (column vector) and `x_row` is a `(1, 10)` array (row vector), their matrix product is a `(10, 10)` array. Each element `(i, j)` (ith row, jth column) is the product of `x[i]` and `x[j]`, which is what we want for our multiplication table.

Another method is to use the regular NumPy multiplication with the `*` symbol. On arrays, this operation is to be understood as the element-wise multiplication, not the matrix multiplication. Here is an example:

```
In [7]: x_row * x_row
Out[7]: array([[  1,   4,   9,  16,  25,  36,  49,  64,  81, 100]])
```

This returns the squares of all numbers in `x_row`.

In our case, we can also use this operation to compute the multiplication table:

```
In [8]: x_row * x_col
Out[8]: array([[  1,   2,   3, ...,   9,  10],
               [  2,   4,   6, ...,  18,  20],
               ...
               [  9,  18,  27, ...,  81,  90],
               [ 10,  20,  30, ...,  90, 100]])
```

Why did multiplying a `(1, 10)` array by a `(10, 1)` array resulted in a `(10, 10)` array? The reason is called **broadcasting**. Element-wise array operations like the regular multiplication `*` normally requires arrays to have the same shape. However, NumPy also accepts arrays with compatible but not identical dimensions. The general rule is that *two dimensions are compatible when they are equal, or when one of them is 1*. The dimension equal to one is transparently and silently stretched to match the other dimension, and this operation does not involve any memory copy. Here, broadcasting allows us to compute the multiplication table with an element-wise multiplication operation.

You will find more information at the following pages:

- Array manipulation routines at `http://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html`
- Broadcasting rules at `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html`

Computing with NumPy arrays

We now get to the substance of array programming with NumPy. We will perform manipulations and computations on ndarrays.

Let's first import NumPy, pandas, matplotlib, and seaborn:

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import seaborn as sns
        %matplotlib inline
```

We load the NYC taxi dataset with pandas:

```
In [2]: data = pd.read_csv('../chapter2/data/nyc_data.csv',
                           parse_dates=['pickup_datetime',
                                        'dropoff_datetime'])
```

We get the pickup and dropoff locations of the taxi rides as ndarrays, using the `.values` attribute of pandas DataFrames:

```
In [3]: pickup = data[['pickup_longitude', 'pickup_latitude']].values
        dropoff = data[['dropoff_longitude',
                        'dropoff_latitude']].values
        pickup
Out[3]: array([[-73.955925,  40.781887],
               [-74.005501,  40.745735],
               [-73.969955,  40.79977 ],
               ...,
               [-73.993492,  40.729347],
               [-73.978477,  40.772945],
               [-73.987206,  40.750568]])
```

Selection and indexing

Let's illustrate selection and indexing with NumPy. These operations are similar to those offered by pandas on DataFrame and Series objects.

In NumPy, a given element can be retrieved with `pickup[i, j]`, where `i` is the 0-indexed row number, and `j` is the 0-indexed column number:

```
In [4]: print(pickup[3, 1])
Out[4]: 40.755081
```

A part of the array can be selected with the slicing syntax, which supports a start position, an end position, and an optional step, as shown here:

```
In [5]: pickup[1:7:2, 1:]
Out[5]: array([[ 40.745735],
               [ 40.755081],
               [ 40.768978]])
```

Here, we've selected the elements at `[1, 1]`, `[3, 1]`, and `[5, 1]`. The slicing syntax in Python is `start:end:step` where `start` is included and `end` is excluded. If `start` or `end` are omitted, they default to `0` or the length of the dimension, respectively, whereas `step` defaults to 1. For example, `1:` is equivalent to `1:n:1` where `n` is the size of the axis.

Let's select the longitudes of all pickup locations, in other words, the first column:

```
In [6]: lon = pickup[:, 0]
        lon
Out[6]: array([-73.9559, -74.0055, ..., -73.9784, -73.9872])
```

The result is a 1D ndarray.

We also get the second column of `pickup`:

```
In [7]: lat = pickup[:, 1]
        lat
Out[7]: array([ 40.7818, 40.7457, ..., 40.7729, 40.7505])
```

Boolean operations on arrays

Let's now illustrate filtering operations in NumPy. Again, these are similar to pandas. As an example, we're going to select all trips departing at a given location:

```
In [8]: lon_min, lon_max = (-73.98330, -73.98025)
        lat_min, lat_max = ( 40.76724,  40.76871)
```

In NumPy, symbols like arithmetic, inequality, and boolean operators work on ndarrays on an element-wise basis. Here is how to select all trips where the longitude is between `lon_min` and `lon_max`:

```
In [9]: in_lon = (lon_min <= lon) & (lon <= lon_max)
        in_lon
Out[9]: array([False, False, False, ..., False, False, False],
              dtype=bool)
```

The symbol `&` represents the AND boolean operator, while | represents the OR.

Here, the result is a Boolean vector containing as many elements as there are in the `lon` vector.

How many `True` elements are there in this array? NumPy arrays provide a `sum()` method that returns the sum of all elements in the array. When the array contains boolean values, `False` elements are converted to 0 and `True` elements are converted to 1. Therefore, the sum corresponds to the number of `True` elements:

```
In [10]: in_lon.sum()
Out[10]: 69163
```

We can process the latitudes similarly:

```
In [11]: in_lat = (lat_min <= lat) & (lat <= lat_max)
```

Then, we get all trips where both the longitude and latitude belong to our rectangle:

```
In [12]: in_lonlat = in_lon & in_lat
         in_lonlat.sum()
Out[12]: 3998
```

The `np.nonzero()` function returns the indices corresponding to `True` in a boolean array, as shown here:

```
In [13]: np.nonzero(in_lonlat)[0]
Out[13]: array([   901,   1011,   1066, ..., 845749, 845903, 846080])
```

Finally, we'll need the dropoff coordinates:

```
In [14]: lon1, lat1 = dropoff.T
```

This is a more concise way of writing `lon1 = dropoff[:, 0]; lat1 = dropoff[:, 1]`. The `T` attribute corresponds to the transpose of a matrix, which simply means that a matrix's columns become the corresponding rows of a new matrix, and the new columns are the original matrix's rows. Here, `dropoff.T` is a `(2, N)` array where the first row contains the longitude and the second row contains the latitude. In NumPy, an ndarray is iterable along the first dimension, in other words, along the rows of the matrix. Therefore, the syntax unpacking feature of Python allows us to concisely assign `lon1` to the first row and `lat1` to the second row.

Mathematical operations on arrays

We have the coordinates of all pickup and dropoff locations in NumPy arrays. Let's compute the straight line distance between those two locations, for every taxi trip.

There are several mathematical formulas giving the distance between two points given by their longitudes and latitudes. Here, we will compute a great-circle distance with a spherical Earth approximation.

The following function implements this formula.

```
In [15]: EARTH_R = 6372.8
         def geo_distance(lon0, lat0, lon1, lat1):
             """Return the distance (in km) between two points in
             geographical coordinates."""
             # from: http://en.wikipedia.org/wiki/Great-circle_distance
             # and: http://stackoverflow.com/a/8859667/1595060
             lat0 = np.radians(lat0)
             lon0 = np.radians(lon0)
             lat1 = np.radians(lat1)
             lon1 = np.radians(lon1)
             dlon = lon0 - lon1
             y = np.sqrt(
                 (np.cos(lat1) * np.sin(dlon)) ** 2
                  + (np.cos(lat0) * np.sin(lat1)
                  - np.sin(lat0) * np.cos(lat1) * np.cos(dlon)) ** 2)
             x = np.sin(lat0) * np.sin(lat1) + \
                 np.cos(lat0) * np.cos(lat1) * np.cos(dlon)
             c = np.arctan2(y, x)
             return EARTH_R * c
```

We have made extensive use of trigonometric functions provided by NumPy: `np.radians()` (converting numbers from degrees into radians), `np.cos()`, `np.sin()`, `np.arctan2(x, y)` (returning the arctangent of `x/y`), and so on. These mathematical functions are defined on real numbers, but NumPy provides *vectorized* versions of them. These vectorized functions not only work on numbers but also on arbitrary numerical ndarrays. As we have explained earlier, these functions are orders of magnitude faster than Python loops. You will find the list of mathematical functions in NumPy at `http://docs.scipy.org/doc/numpy/reference/routines.math.html`.

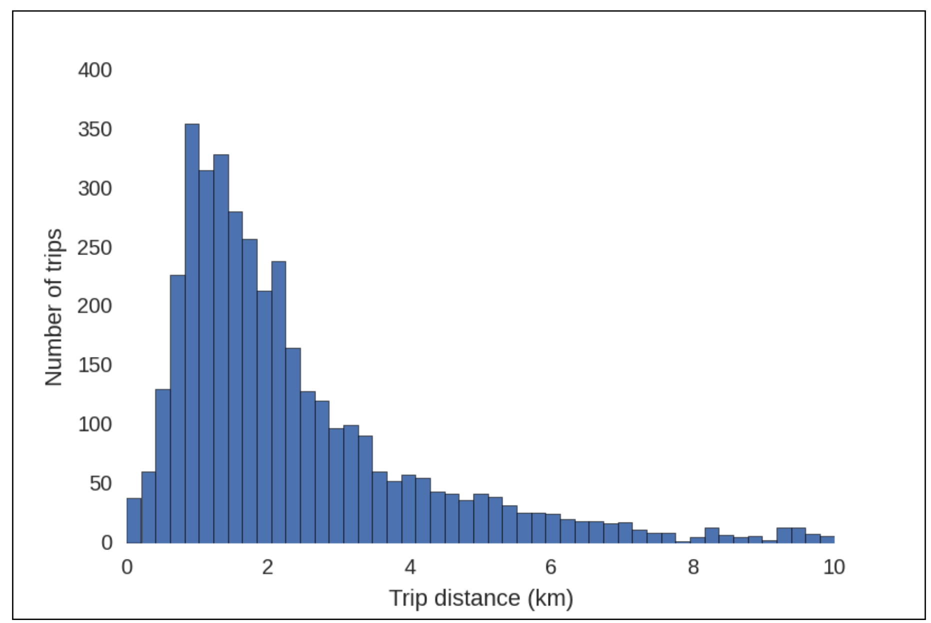
All in all, NumPy makes it quite natural to implement mathematical formulas on arrays of numbers. The syntax is exactly the same as with scalar operations.

Now, let's compute the straight line distances of all taxi trips:

```
In [16]: distances = geo_distance(lon, lat, lon1, lat1)
```

Below is a histogram of these distances for the trips starting at Columbus Circle (the location indicated by the geographical coordinates above). Those trips are indicated by the `in_lonlat` boolean array obtained earlier in this section.

```
In [17]: plt.hist(distances[in_lonlat], np.linspace(0., 10., 50))
         plt.xlabel('Trip distance (km)')
         plt.ylabel('Number of trips')
```



Histogram of trip distances

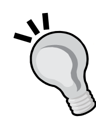
matplotlib's `plt.hist()` function computes a histogram and plots it. It is a convenient wrapper around NumPy's `np.histogram()` function that simply computes a histogram. You will find more statistical functions in NumPy at `http://docs.scipy.org/doc/numpy/reference/routines.statistics.html`.

A density map with NumPy

We have reviewed the most common array operations in this section. We will now see a more advanced example combining several techniques. We will compute and display a 2D density map of the most common pickup and dropoff locations, at specific times in the day.

First, let's select the evening taxi trips. This time, we use pandas, which offers particularly rich date and time features. We eventually get a NumPy array with the `.values` attribute:

```
In [18]: evening = (data.pickup_datetime.dt.hour >= 19).values
In [19]: n = np.sum(evening)
In [20]: n
Out[20]: 242818
```



Pandas and NumPy

Remember that pandas is based on NumPy, and that it is quite common to leverage both libraries in complex data analysis tasks. A natural workflow is to start loading and manipulating data with pandas, and then switch to NumPy when complex mathematical operations are to be performed on arrays. As a rule of thumb, pandas excels at filtering, selecting, grouping, and other data manipulations, whereas NumPy is particularly efficient at vector mathematical operations on numerical arrays.

The `n` variable contains the number of evening trips in our dataset.

Here is how we are going to create our density map: We consider the set of all pickup and dropoff locations for these `n` evening trips. There are `2n` of such points. Every point is associated with a weight of `-1` for pickup locations and `+1` for dropoff locations. The algebraic density of points at a given location, taking into account the weights, reflects whether people tend to leave or to arrive at this location.

To create the `weights` vector for our `2n` points, we first create a vector containing only zeros. Then, we set the first half of the array to `-1` (pickup) and the last half to `+1` (dropoff):

```
In [21]: weights = np.zeros(2 * n)
In [22]: weights[:n] = -1
         weights[n:] = +1
```

> Indexing in Python and NumPy starts at 0, and excludes the last element. The first half of `weights` is made of `weights[0]`, `weights[1]`, up to `weights[n-1]`. There are `n` of such elements. The slice `weights[:n]` is equivalent to `weights[0:n]`: it starts at `weights[0]`, and ends at `weights[n]` *excluded*, so the last element is effectively `weights[n-1]`.

We could also have used array manipulation routines provided by NumPy, such as `np.tile()` to concatenate copies of an array along several dimensions, or `np.repeat()` to make copies of every element along several dimensions. You will find the list of manipulation functions at `http://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html`.

Next, we create a `(2n, 2)` array defined by the vertical concatenation of the pickup and dropoff locations for the evening trips:

```
In [23]: points = np.r_[pickup[evening],
                        dropoff[evening]]
In [24]: points.shape
Out[24]: (485636, 2)
```

The concise `np.r_[]` syntax allows us to concatenate arrays along the first (vertical) dimension. We could also have used more explicit manipulation functions such as `np.vstack()` or `np.concatenate()`.

Now, we convert these points from geographical coordinates to pixel coordinates, using the same function as in the previous chapter:

```
In [25]: def lat_lon_to_pixels(lat, lon):
             lat_rad = lat * np.pi / 180.0
             lat_rad = np.log(np.tan((lat_rad + np.pi / 2.0) / 2.0))
             x = 100 * (lon + 180.0) / 360.0
             y = 100 * (lat_rad - np.pi) / (2.0 * np.pi)
             return (x, y)
In [26]: lon, lat = points.T
         x, y = lat_lon_to_pixels(lat, lon)
```

We now define the bins for the 2D histogram in our density map. This defines a 2D grid over which we compute the histogram.

```
In [27]: lon_min, lat_min = -74.0214, 40.6978
         lon_max, lat_max = -73.9524, 40.7982
In [28]: x_min, y_min = lat_lon_to_pixels(lat_min, lon_min)
         x_max, y_max = lat_lon_to_pixels(lat_max, lon_max)
In [29]: bin = .00003
         bins_x = np.arange(x_min, x_max, bin)
         bins_y = np.arange(y_min, y_max, bin)
```

These two arrays contain the horizontal and vertical bins.

Finally, we compute the histogram with the `np.histogram2d()` function. We pass as arguments the `y, x` coordinates of the points (reversed because we want the grid's first axis to represent the `y` coordinate), the weights, and the bins. This function computes a weighted sum of the points, in every bin. It returns several objects, the first of which is the density map we are interested in:

```
In [30]: grid, _, _ = np.histogram2d(y, x, weights=weights,
                                     bins=(bins_y, bins_x))
```

You will find the reference documentation of this function at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram2d.html#numpy.histogram2d`.

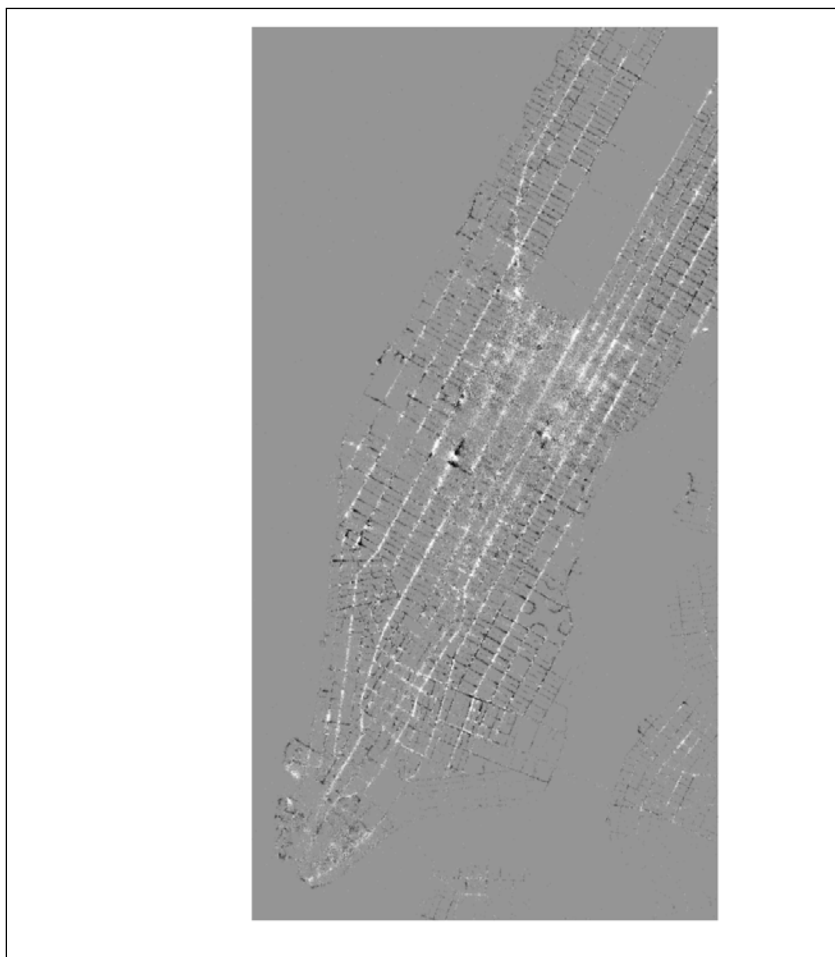
Before displaying the density map, we will apply a logistic function to it in order to smooth it:

```
In [31]: density = 1. / (1. + np.exp(-.5 * grid))
```

This logistic function is called the **expit function**. It can also be found in the SciPy package at `scipy.special.expit()`. `scipy.special` provides many other special functions such as Bessel functions, Gamma functions, hypergeometric functions, and so on.

Finally, we display the density map with `plt.imshow()`:

```
In [32]: plt.imshow(density,
                    origin='lower',
                    interpolation='bicubic'
                    )
         plt.axis('off')
```



Sources and sinks in taxi trip data

In this figure, white areas correspond to common dropoff locations whereas dark areas correspond to common pickup locations.

matplotlib's `plt.imshow()` function displays a matrix as an image. It supports several interpolation methods. Here, we used a bicubic interpolation. The `origin` argument is necessary because in our `density` matrix, the top-left corner corresponds to the smallest latitude, so it should correspond to the bottom-left corner in the image.

Other topics

We only scratched the surface of the possibilities offered by NumPy. Further numerical computing topics covered by NumPy and the more specialized SciPy library include:

- Search and sort in arrays
- Set operations
- Linear algebra
- Special mathematical functions
- Fourier transforms and signal processing
- Generation of pseudo-random numbers
- Statistics
- Numerical integration and numerical ODE solvers
- Function interpolation
- Basic image processing
- Numerical optimization

The *IPython Cookbook* covers many of these topics.

Here are a few references:

- NumPy reference at `http://docs.scipy.org/doc/numpy/reference/`
- SciPy reference at `http://docs.scipy.org/doc/scipy/reference/`
- *IPython Cookbook* at `http://ipython-books.github.io/cookbook/`

Summary

In this chapter, we introduced NumPy and the ndarray structure. We explained the main concepts of array computing and the performance benefits it brings over Python loops. We also showed how to use NumPy in conjunction with pandas for advanced data analysis tasks.

In the next chapter, we will explore several options for plotting, visualization, and graphical interfaces.

4
Interactive Plotting and Graphical Interfaces

In the previous chapter, we created a few plots with matplotlib and seaborn. In this chapter, we'll look at these libraries in more detail. We'll also discuss some of the many other visualization libraries in Python, with a particular emphasis on those that integrate with the Jupyter Notebook.

We will cover the following topics:

- Choosing a plotting backend
- matplotlib and seaborn essentials
- Image processing
- Further plotting and visualization libraries

Choosing a plotting backend

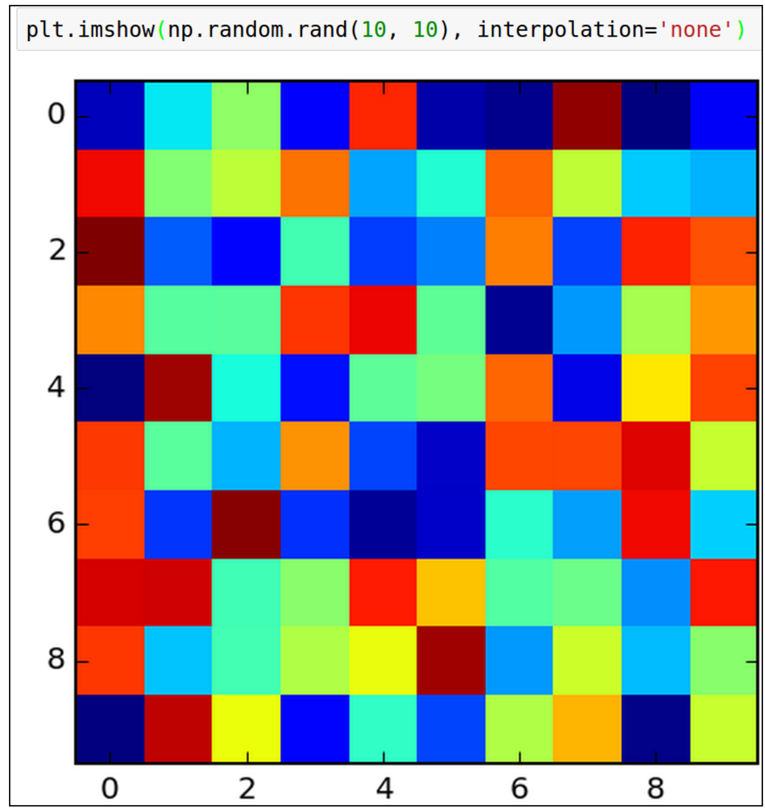
There are different ways to display a plot in the Jupyter Notebook.

Inline plots

So far, we have created plots within the Notebook using the matplotlib inline mode. This is activated with the `%matplotlib inline` magic command in the Notebook. Figures created in this mode are converted to PNG images stored within the notebook `.ipynb` files. This is convenient when sharing notebooks because the plots are viewable by other users. However, these plots are static, and they are therefore not practical for interactive visualization.

Here is an example:

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
In [2]: %matplotlib inline
In [3]: plt.imshow(np.random.rand(10, 10), interpolation='none')
```



Inline backend

Exported figures

Matplotlib can export figures to bitmap (PNG, JPG, and others) or vector formats (PDF, EPS, and others). Refer to the documentation of `plt.savefig()` for more details: `http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.savefig`.

GUI toolkits

You can also display a plot in a separate window on your desktop. This uses a GUI backend toolkit that interacts with the operating system and the desktop environment to create and manage windows. Examples of cross-platform backends include Qt, wx, Tk, GTK, and others. Matplotlib can use any of these toolkits to display a figure.

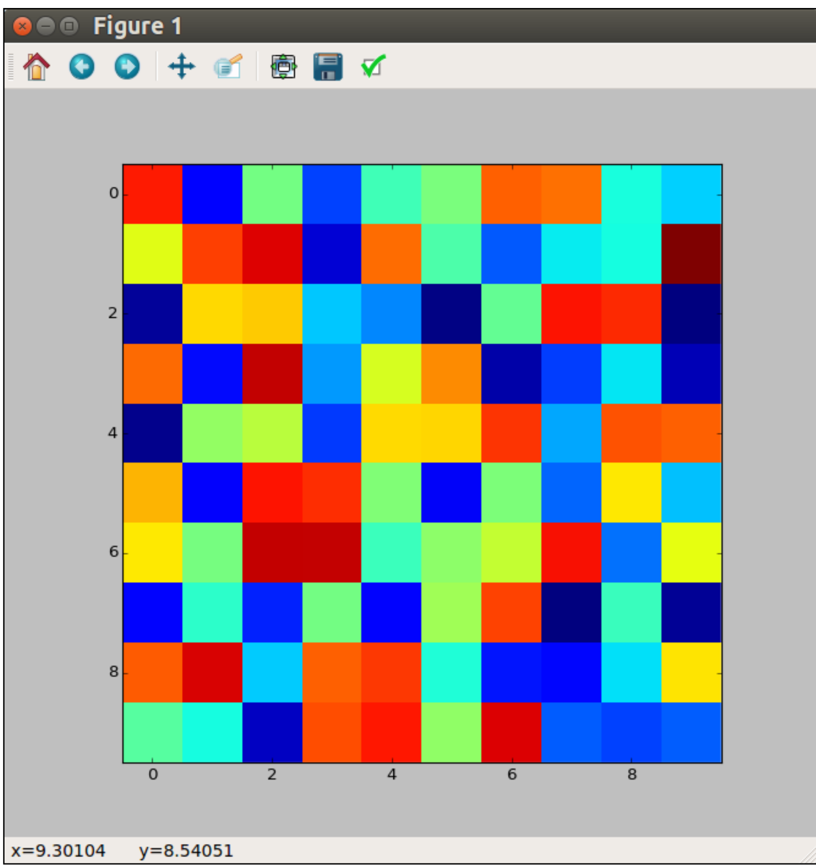
Qt is a popular and powerful choice for GUIs; it is well-supported by matplotlib and Jupyter.

To enable this mode in the Jupyter Notebook, use the `%matplotlib qt` magic command. This makes the Notebook responsive while the popup windows are displayed, and it enables interactive modification of plots through the Notebook.

A related command is `%gui qt`: it enables the creation of any interactive Qt window, not just matplotlib figures, from IPython. Refer to the following link for more information about GUI event loop support in IPython: `http://ipython.org/ipython-doc/dev/interactive/reference.html#gui-event-loop-support`.

Matplotlib figures displayed with a GUI toolkit are interactive. They support panning and zooming with the mouse. There is also a toolbar that gives you access to several functions.

```
In [4]: %matplotlib qt
        plt.imshow(np.random.rand(10, 10), interpolation='none')
```

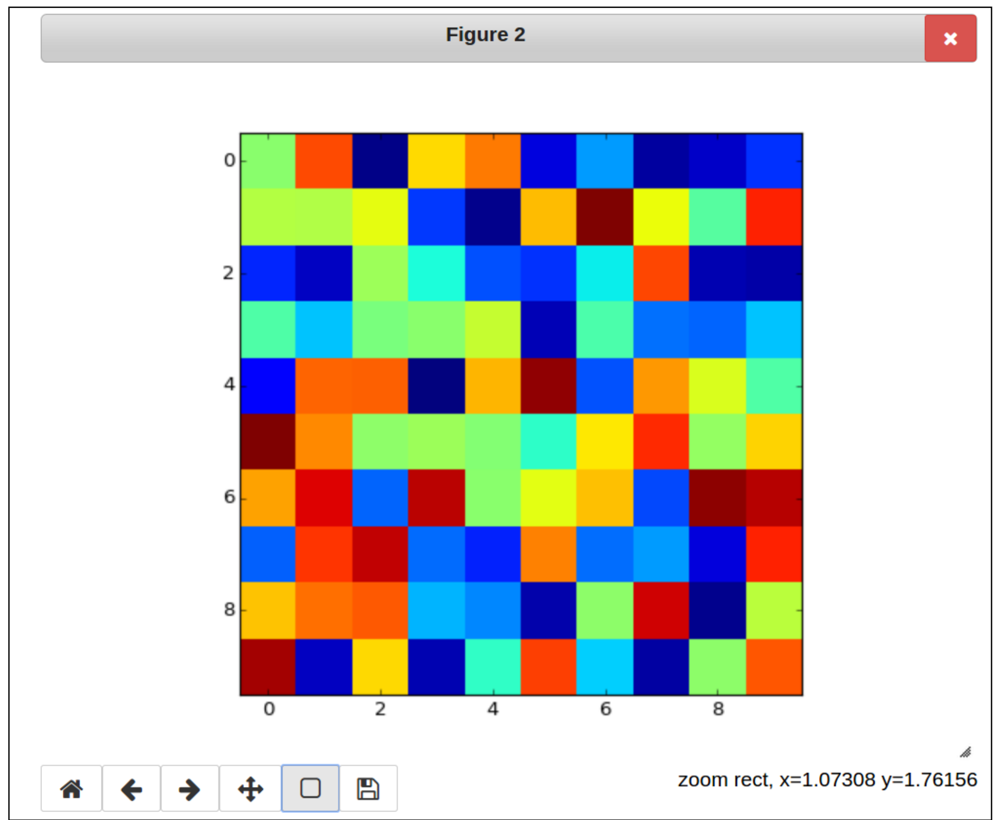


Qt backend

Dynamic inline plots

Since matplotlib 1.4.3, inline plots can be made interactive in the Notebook through the *nbagg* backend. To activate it, use `%matplotlib notebook` or the following command (you may need to restart the kernel in order to deactivate the previous backends):

```
In [5]: import matplotlib
        matplotlib.use('nbagg')
In [6]: plt.imshow(np.random.rand(10, 10), interpolation='none')
        plt.show()
```



matplotlib's nbagg backend in the Notebook

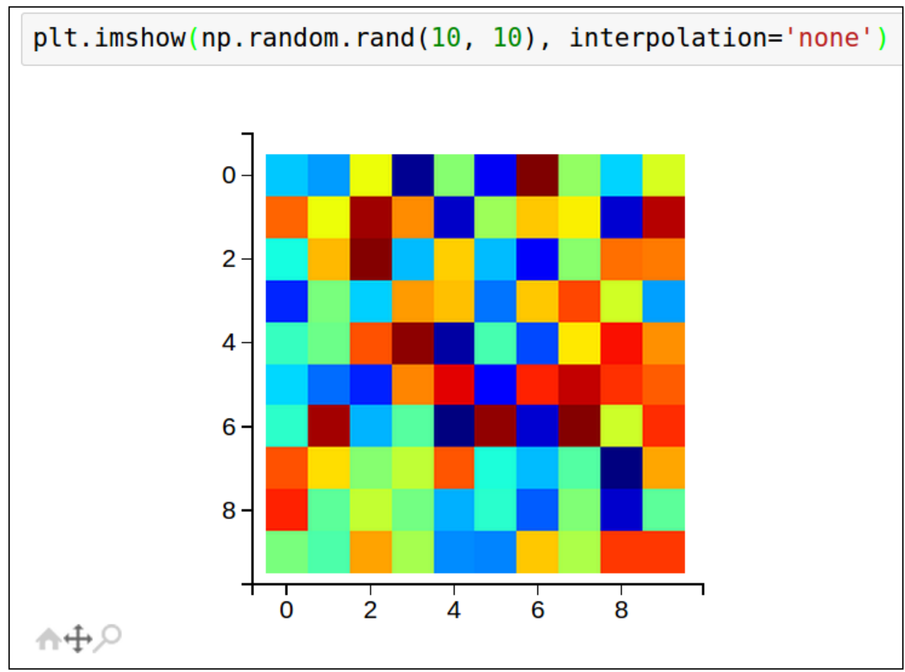
You can pan and zoom in the inline plot. However, this sort of interactivity requires a Python server, so these plots remain static when displayed in nbviewer.

Web-based visualization

The web platform has seen dramatic progress in data visualization technologies in the past few years. Although this has nothing to do with Python in principle, the fact that the Jupyter Notebook is a web application makes it theoretically possible to leverage these technologies in the Jupyter Notebook.

For example, the mpld3 project automatically converts a matplotlib plot into a JavaScript D3 interactive visualization (D3 is a Javascript visualization library that runs in the browser; we'll see it again later in this chapter). It is very easy to use: once it is installed with `pip install mpld3`, just import and enable it with (you may also need to restart the kernel):

```
In [7]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        import mpld3
        mpld3.enable_notebook()
In [8]: plt.imshow(np.random.rand(10, 10), interpolation='none')
```



mpld3 screenshot

Then, matplotlib figures are rendered with D3 and support panning and zooming, even in nbviewer.

Here are a few references:

- D3.js at `http://d3js.org/`
- mpld3 at `http://mpld3.github.io/`

matplotlib and seaborn essentials

matplotlib is the main plotting library in Python. While it is particularly rich and powerful, it may be difficult to use sometimes. Further, its default styling could be better. There is some work in progress to improve the default styling in matplotlib. In the meantime, the seaborn library offers better styling for matplotlib as well as easy-to-use high-level statistical plotting routines based on matplotlib.

In this section, we will detail some of the main plotting capabilities of matplotlib, while using the seaborn styling.

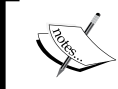
We first import matplotlib and seaborn and we activate the inline mode in the Notebook:

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import seaborn
        %matplotlib inline
```

There is a `pylab` mode that imports all NumPy and matplotlib variables into the interactive namespace. This mode makes the transition easier for users coming from MATLAB. However, using this mode is not recommended. The standard practice is to import NumPy into the `np` namespace and matplotlib's pyplot interface into the `plt` namespace. Refer to this link for more details: `http://nbviewer.ipython.org/github/Carreau/posts/blob/master/10-No-PyLab-Thanks.ipynb`.

Common plots with matplotlib

Let's create a few simple plots with pyplot.

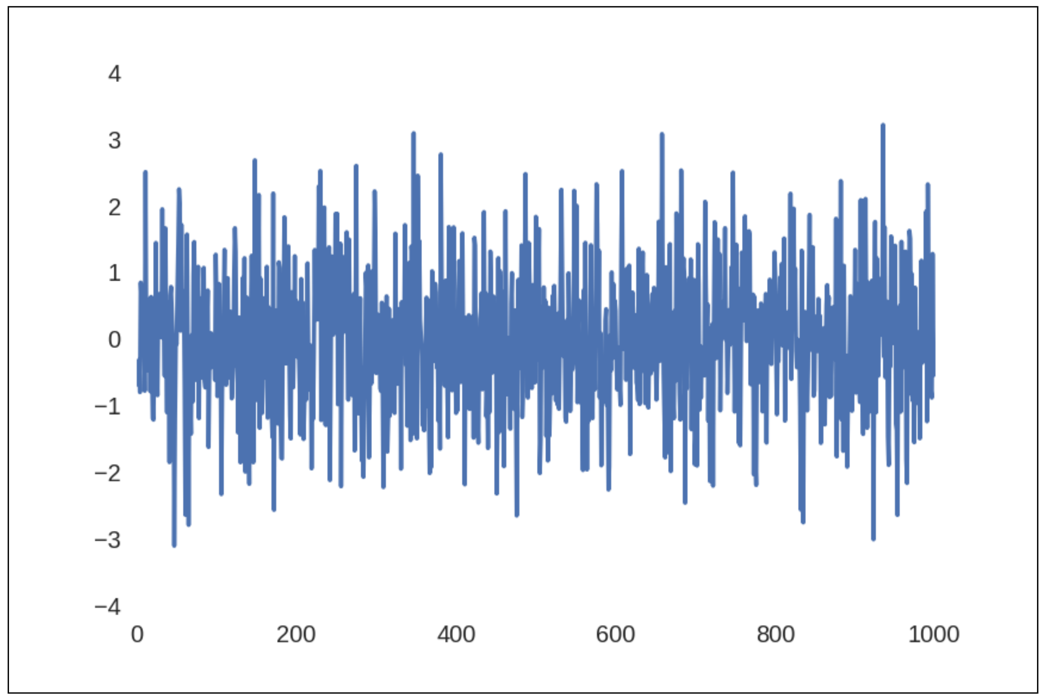


Pyplot is a MATLAB-like plotting interface built on top of the matplotlib API. Using the matplotlib API is reserved to advanced users, and most users make matplotlib figures with pyplot.

A line plot represents a mathematical curve or a digital signal as a continuous succession of line segments. Let's generate and display a random signal with matplotlib:

```
In [2]: y = np.random.randn(1000)
In [3]: plt.plot(y)
```

Here is the result:

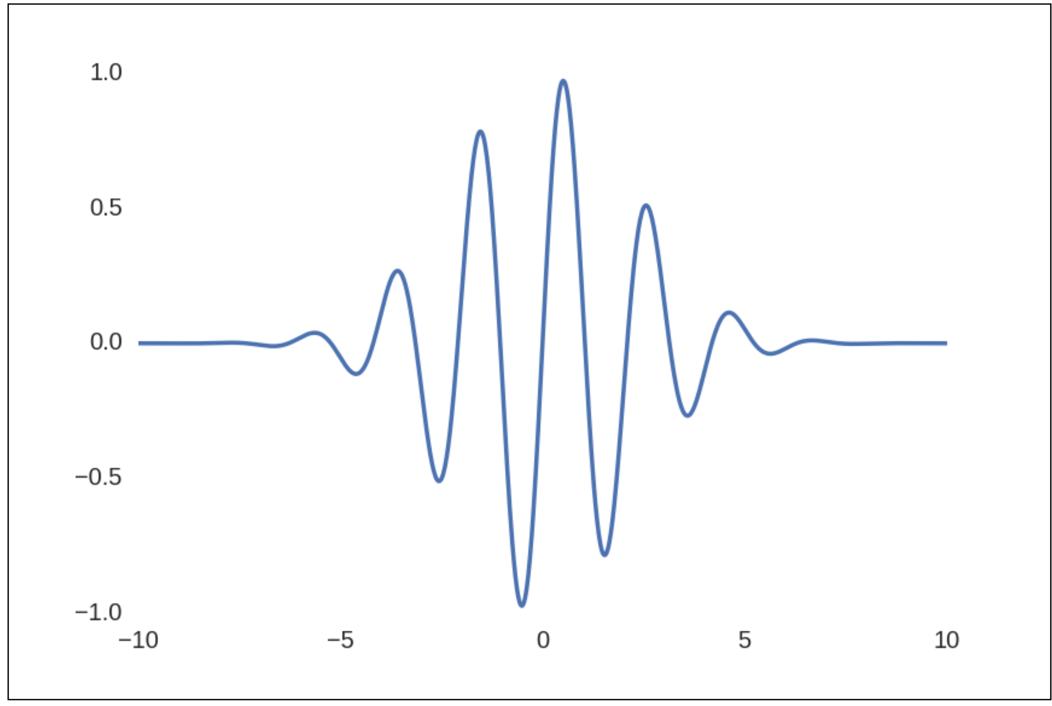


A line plot with matplotlib

By default, the x coordinates are successive integers. We can also specify these coordinates directly. For example, let's plot the graph of a mathematical function:

```
In [4]: x = np.linspace(-10., 10., 1000)
        y = np.sin(3 * x) * np.exp(-.1 * x**2)
In [5]: plt.plot(x, y)
```

Here is the result:

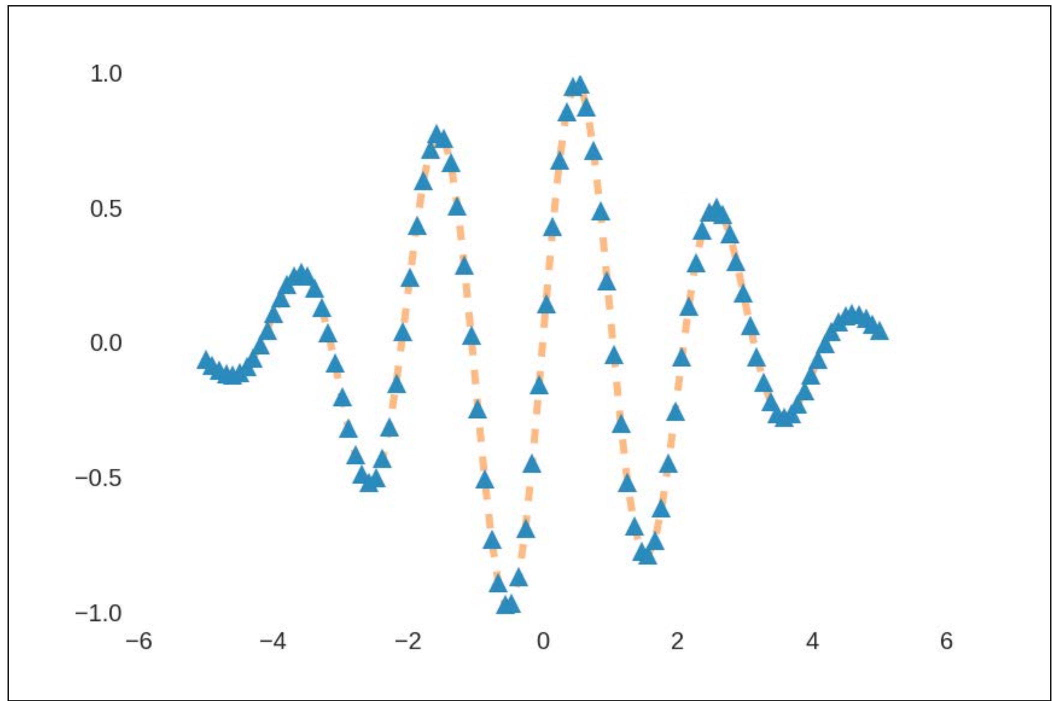


Graph of a function with matplotlib

All aspects of the plot can be customized, as shown here:

```
In [6]: x = np.linspace(-5., 5., 100)
        y = np.sin(3 * x) * np.exp(-.1 * x ** 2)
In [7]: plt.plot(x, y, '--^',
                 lw=3, color='#fdbb84',
                 mfc='#2b8cbe', ms=8)
```

Here is a screenshot:



A customized plot with matplotlib

Let's explain the different options we have used in `plt.plot()`:

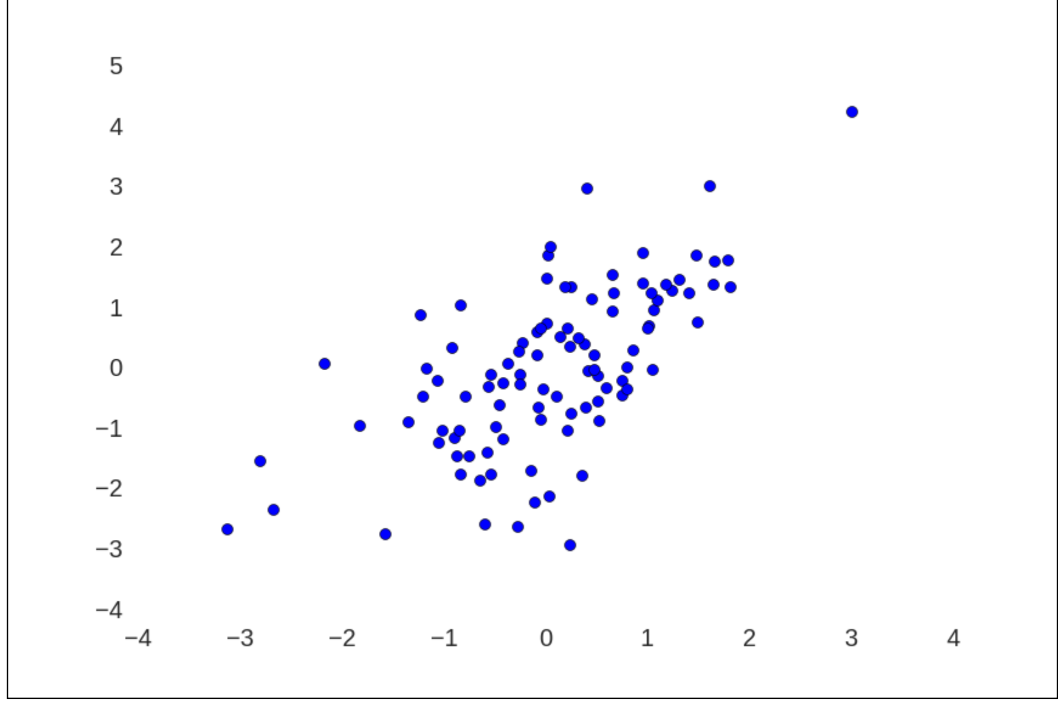
- The third argument is a format string specifying the aspect of the plot in a compact form:
    - The `--` characters indicate a dashed line style.
    - The `^` character indicates a upper triangle marker.
- The `lw` argument indicates the line width.
- The color can be specified in many ways, including as a hexadecimal RGB value.
- The marker face color is indicated by `mfc`.
- The marker size is indicated by `ms`.

You will find more details about the plot customization options at `http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot`.

Another common type of plot is the scatter plot, which just displays points in two dimensions. It offers a simple way to observe the relationship between two variables in a dataset. Here is an example:

```
In [8]: x = np.random.randn(100)
        y = x + np.random.randn(100)
In [9]: plt.scatter(x, y)
```

Here is the result:



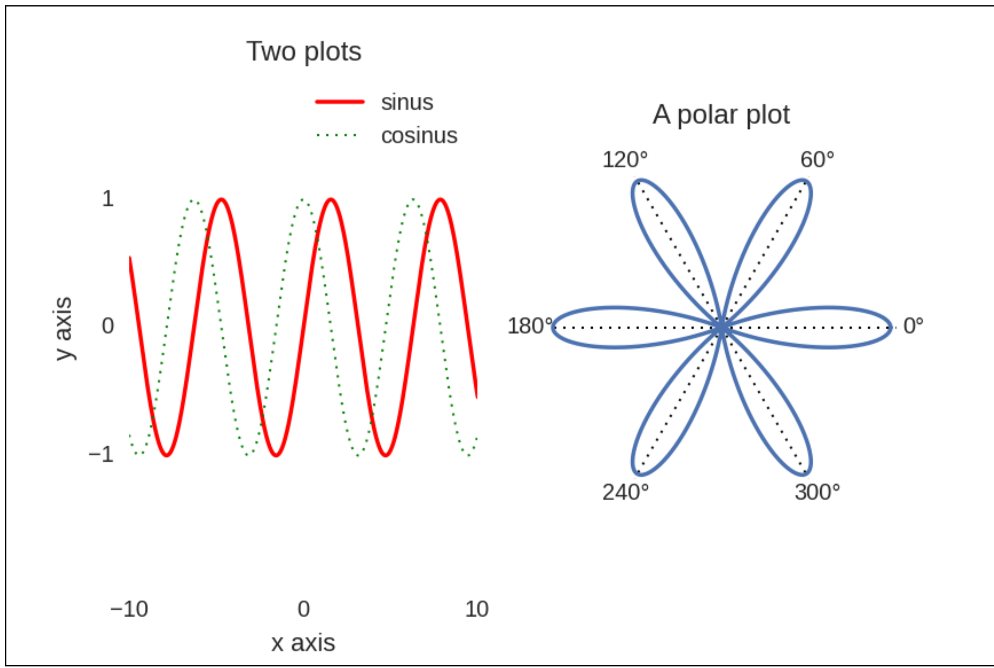
A scatter plot with matplotlib

Customizing matplotlib figures

We've seen above how to customize plots. We can also customize the axes, legends, titles, and everything else. Additionally, we can create multiple plots in the same figure. Here is an example showing all of these aspects:

```
In [10]: # Left panel.
         plt.subplot(1, 2, 1)
         x = np.linspace(-10., 10., 1000)
         plt.plot(x, np.sin(x), '-r', label='sinus')
         plt.plot(x, np.cos(x), ':g', lw=1, label='cosinus')
         plt.xticks([-10, 0, 10])
         plt.yticks([-1, 0, 1])
         plt.ylim(-2, 2)
         plt.xlabel("x axis")
         plt.ylabel("y axis")
         plt.title("Two plots")
         plt.legend()

         # Right panel.
         plt.subplot(1, 2, 2, polar=True)
         x = np.linspace(0, 2 * np.pi, 1000)
         plt.plot(x, 1 + 2 * np.cos(6 * x))
         plt.yticks([])
         plt.xlim(-.1, 3.1)
         plt.ylim(-.1, 3.1)
         plt.xticks(np.linspace(0, 5 * np.pi / 3, 6))
         plt.title("A polar plot")
         plt.grid(color='k', linewidth=1, linestyle=':')
```

Here is a screenshot:



Subplots with matplotlib

Let's explain all options:

- `plt.subplot()` is used to add several plots in the same figure. The three arguments are:
    - number of rows
    - number of columns
    - index of the subplot in the grid (from left to right, top to bottom, starting at 1)
- A label can be passed to a component of a plot.
- In the left subplot, we create two line plots by calling `plt.plot()` twice.

- `plt.xticks()` and `plt.yticks()` allow us to specify the ticks on the x and y axes.
- `plt.ylim()` specifies the limits of the plot on the y axis.
- `plt.xlabel()` and `plt.ylabel()` indicate the legend of the x and y axes.
- The title of the subplot is given with `plt.title()`.
- `plt.legend()` displays the legend for the different components in the subplot (here, the two line plots).
- A subplot with a polar coordinate system is created with the `polar=True` keyword argument.
- `plt.grid()` is used to display a grid.

You can also customize the defaults of matplotlib. Refer to the following link for more information: `http://matplotlib.org/users/customizing.html`.

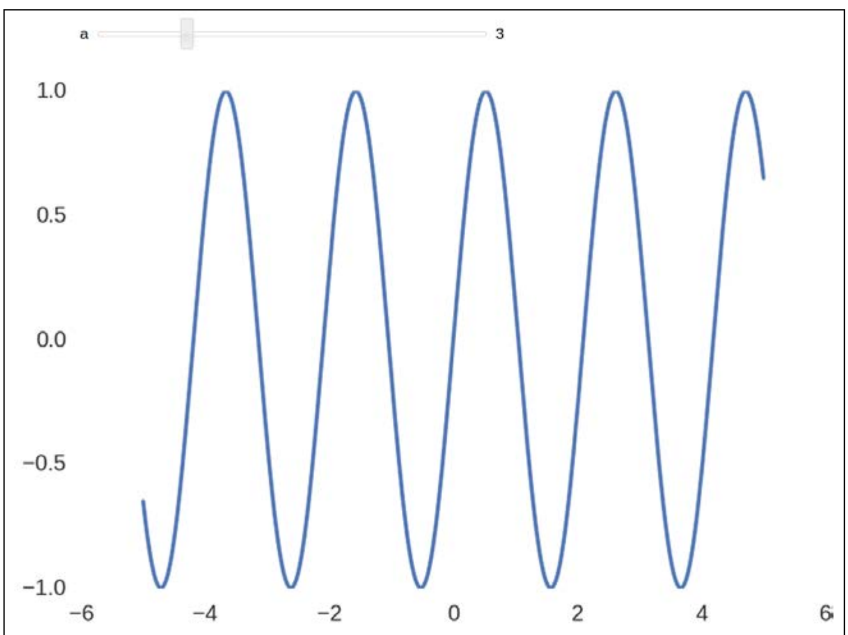
You will find hundreds of examples in the official matplotlib gallery at `http://matplotlib.org/gallery.html`. Every example comes with a screenshot and the code. It is a great way to get a sense of matplotlib's possibilities and to learn how to use matplotlib by the example.

Interacting with matplotlib figures in the Notebook

You can update matplotlib figures interactively in the Notebook using widgets. Here is a simple example:

```
In [11]: from ipywidgets import interact
In [12]: x = np.linspace(-5., 5., 1000)
In [13]: @interact
         def plot_sin(a=(1, 10)):
             plt.plot(x, np.sin(a*x))
             plt.ylim(-1, 1)
```

Here is a screenshot:



An interactive matplotlib figure in the Notebook

The figure updates dynamically in the Notebook as you move the slider.

A similar technique can be used when using a GUI backend instead of the inline mode. We can interactively update the figure from IPython while the figure is still open. Here is an example. First, we activate the Qt backend with the following command:

```
In [14]: %matplotlib qt
```

We create a blue line plot, as follows:

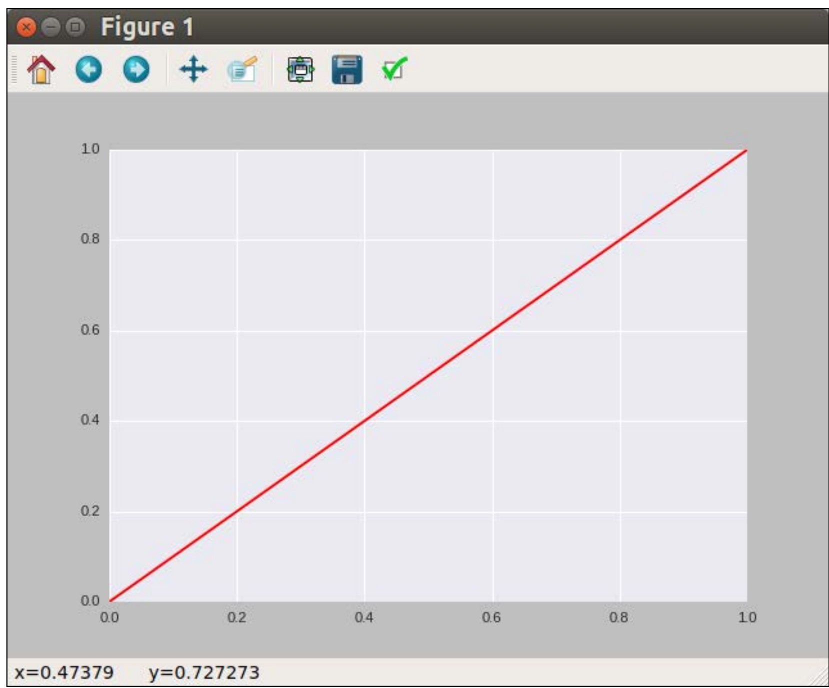
```
In [15]: lines = plt.plot([0, 1], [0, 1], 'b')
In [16]: lines
Out[16]: [<matplotlib.lines.Line2D at 0x7ffa434542e8>]
```

This opens a window with our plot. The variable `lines` contains the list of line plots we've just created (there is just one here).

Now, we interactively update the color of the plot:

```
In [17]: lines[0].set_color('r')
         plt.draw()
```

We explicitly set the color of the line plot, and we redraw the plot to update the figure. The line becomes red.



Qt figure updated from IPython

Here are a few references:

- User's guide at `http://matplotlib.org/users/beginner.html`
- The matplotlib gallery at `http://matplotlib.org/gallery.html`

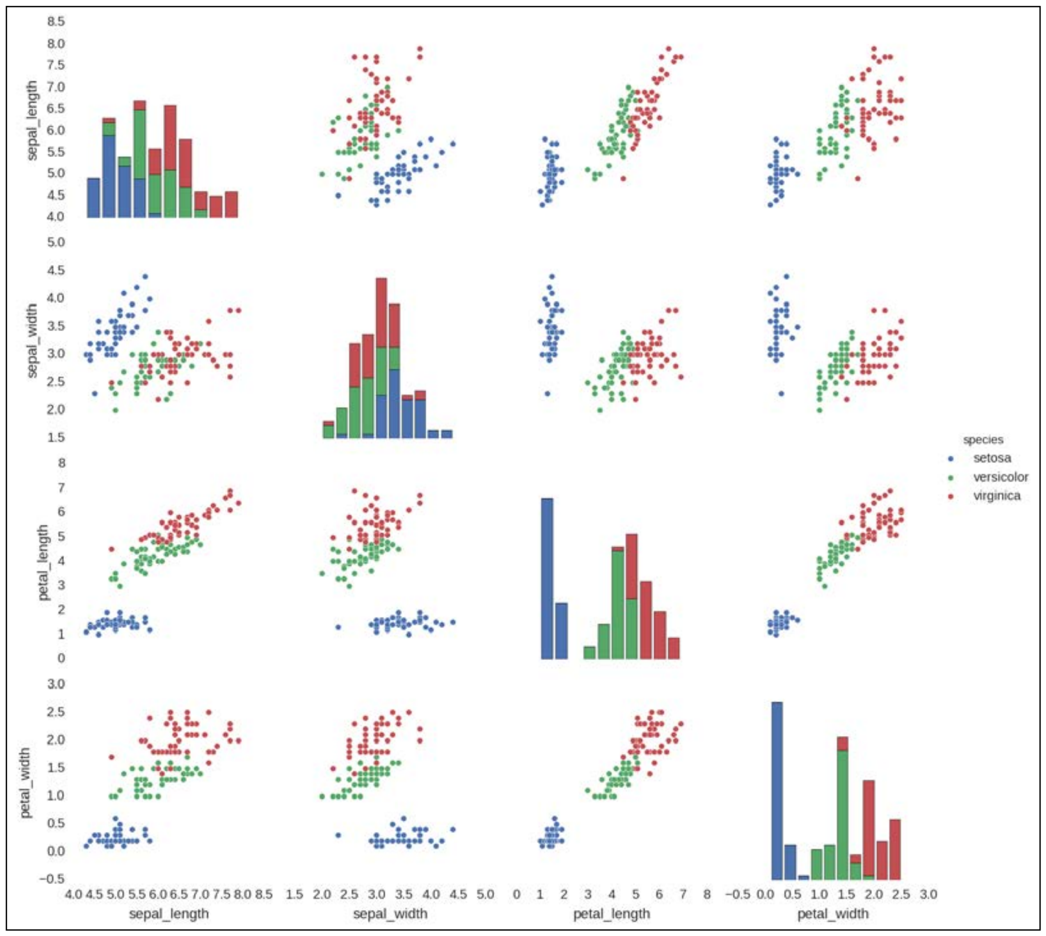
High-level plotting with seaborn

seaborn provides several ready-to-use advanced plotting functions. For example, displaying a set of two-dimensional scatter plots for a higher-dimensional dataset just takes one line with seaborn (see the example at `http://stanford.edu/~mwaskom/software/seaborn/examples/scatterplot_matrix.html`). Let's first load a classic dataset:

```
In [18]: df = seaborn.load_dataset("iris")
         df.head(3)
Out[18]:    sepal_length  sepal_width  petal_length  petal_width species
         0           5.1          3.5           1.4          0.2  setosa
         1           4.9          3.0           1.4          0.2  setosa
         2           4.7          3.2           1.3          0.2  setosa
```

This is a pandas DataFrame containing anatomical features of different types of flowers. Now, let's display a pair plot of this dataset:

```
In [19]: seaborn.pairplot(df, hue="species", size=2.5)
```



A pair plot with seaborn

You will find many more examples in the gallery at `http://stanford.edu/~mwaskom/software/seaborn/examples/index.html`.

Image processing

Several libraries bring image processing capabilities to Python. SciPy, the main scientific Python library, contains a few image processing routines. scikit-image is another library dedicated to image processing. We will show an example in this section, inspired by the one at `http://scikit-image.org/docs/dev/auto_examples/plot_equalize.html`.

When using the Anaconda distribution, scikit-image can be installed with `conda install scikit-image`.

Let's import some packages.

```
In [1]: import numpy as np
        import skimage
        from skimage import img_as_float
        import skimage.filters as skif
        from skimage.color import rgb2gray
        import skimage.data as skid
        import skimage.exposure as skie
        from ipywidgets import interact
        import matplotlib.pyplot as plt
        import seaborn
        %matplotlib inline
```

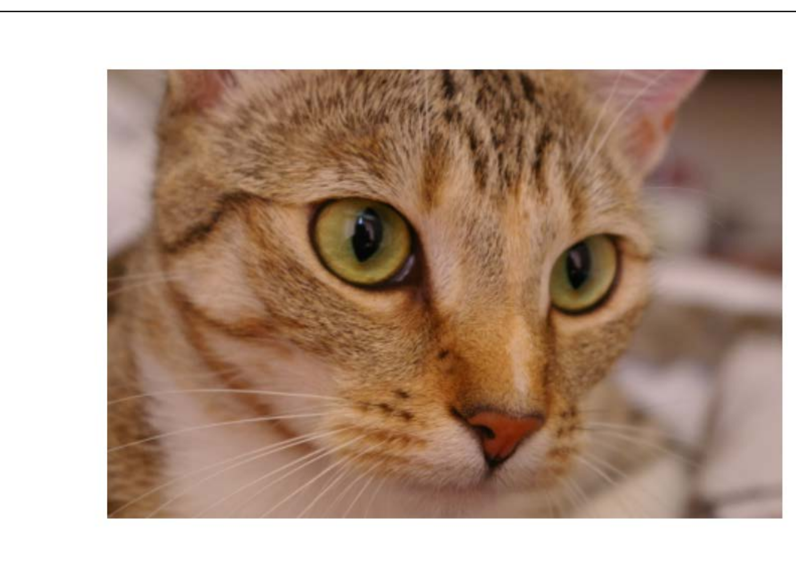
There are a few test images in scikit-image. Here is one:

```
In [2]: chelsea = skid.chelsea()
In [3]: chelsea.shape, chelsea.dtype
Out[3]: ((300, 451, 3), dtype('uint8'))
```

This is a NumPy array with 3 dimensions: height, width, and color channel. This is a colored image, so there are three color channels: Red, Green, and Blue. The data type is `uint8`; every value is between 0 and 255.

We can display the image with matplotlib's `imshow()` function:

```
In [4]: plt.imshow(chelsea)
        plt.axis('off')
```



Chelsea

If you're reading a printed version of this book, the image will be in grayscale. You will find the color version on the book's website.

We now convert this image to a grayscale image:

```
In [5]: img = rgb2gray(chelsea)
In [6]: img.shape, img.dtype
Out[6]: ((300, 451), dtype('float64'))
In [7]: img
Out[7]: array([[ 0.4852, 0.4852, ..., 0.1169, 0.1169],
               [ 0.4969, 0.4930, ..., 0.1225, 0.1272 ],
               ...,
               [ 0.4248, 0.3688, ..., 0.5544, 0.5583]])
```

This is now a 2D array with floating-point intensity values between 0 and 1.

We are now going to analyze the histogram of these intensity values and tweak the exposure of the image. We will use three different methods and then create a simple GUI to observe the results.

First, we'll use the `rescale_intensity()` function to stretch the intensity range of the image. This is a crude exposure adjustment method.

```
In [8]: p2, p98 = np.percentile(img, (2, 98))
In [9]: img_rescale = skie.rescale_intensity(img, in_range=(p2, p98))
```

Next, we use the `equalize_hist()` function to make the histogram approximately constant:

```
In [10]: img_eq = skie.equalize_hist(img)
```

We now use the *Contrast Limited Adaptive Histogram Equalization* algorithm, a more advanced histogram equalization method that enhances the image's contrast.

```
In [11]: img_adapteq = img_as_float(skie.equalize_adapthist(img, clip_
limit=0.03))
```

Finally, we create a GUI with IPython's `@interact` decorator. We will create a dropdown menu with the results of the different exposure adjustment methods:

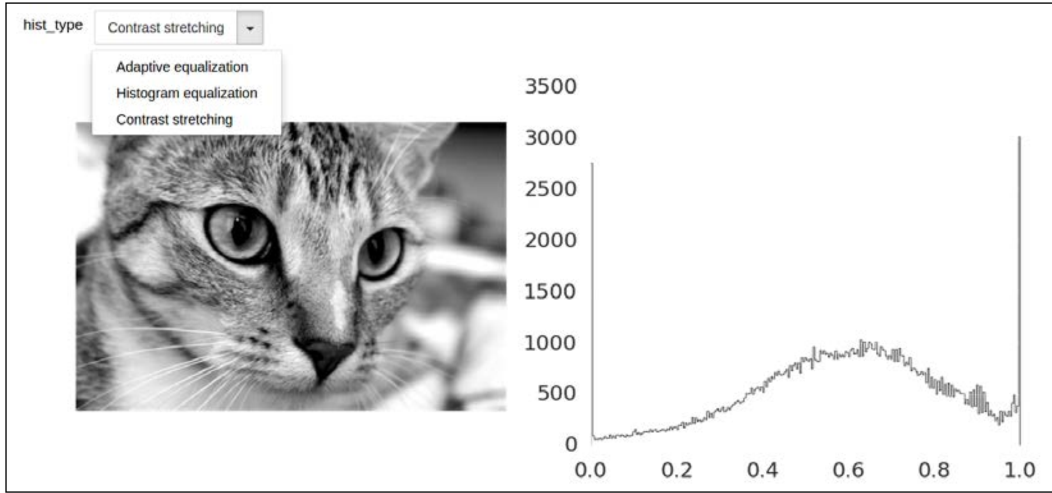
```
In [12]: hist_types = dict([('Contrast stretching', img_rescale),
                            ('Histogram equalization', img_eq),
                            ('Adaptive equalization', img_adapteq)])
```

This dictionary maps the text of the different dropdown menu options to the images.

To create the GUI, we define a function that accepts the method's name `hist_type` as the argument and displays the corresponding plot. We decorate this function with `@interact`, and we specify the list of options for the `hist_type` argument:

```
In [13]: @interact(hist_type=list(hist_types.keys()))
         def display_result(hist_type):
             result = hist_types[hist_type]

             # We display the processed grayscale image on the left.
             plt.subplot(121)
             plt.imshow(result, cmap='gray')
             plt.axis('off')
```

```
# We display the histogram on the right.
plt.subplot(122)
plt.hist(result.ravel(), bins=np.linspace(0., 1., 256),
         histtype='step', color='black')

plt.show()
```



A GUI in the Notebook

You will find more image processing and GUI examples in the *IPython Cookbook*. Here are also a few further references:

- scikit-image's main page at `http://scikit-image.org/`
- scikit-image's gallery at `http://scikit-image.org/docs/dev/auto_examples/`

Further plotting and visualization libraries

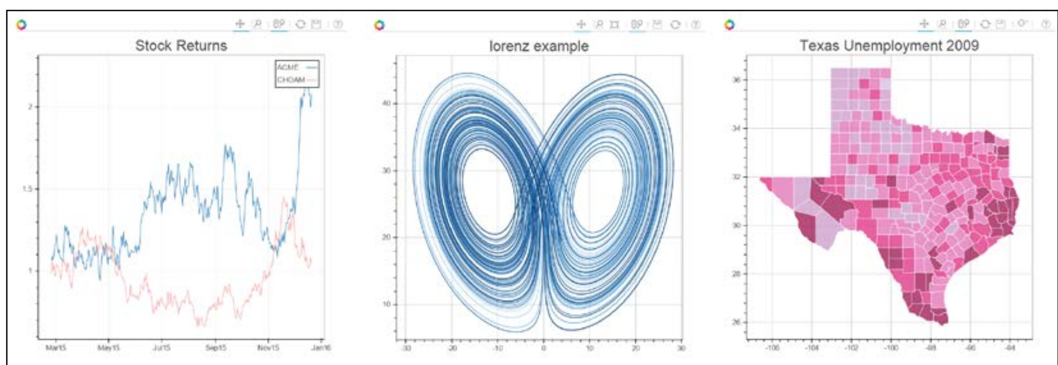
Beyond matplotlib and seaborn, there are many other plotting and visualization libraries in the Python ecosystem. We give an overview in this section.

High-level plotting

Here are a few high-level plotting libraries in Python.

Bokeh

Bokeh is a web-based, general-purpose, and fast visualization toolkit. It integrates well with the rest of the Python ecosystem and generates interactive plots that don't necessarily require a live Python server.



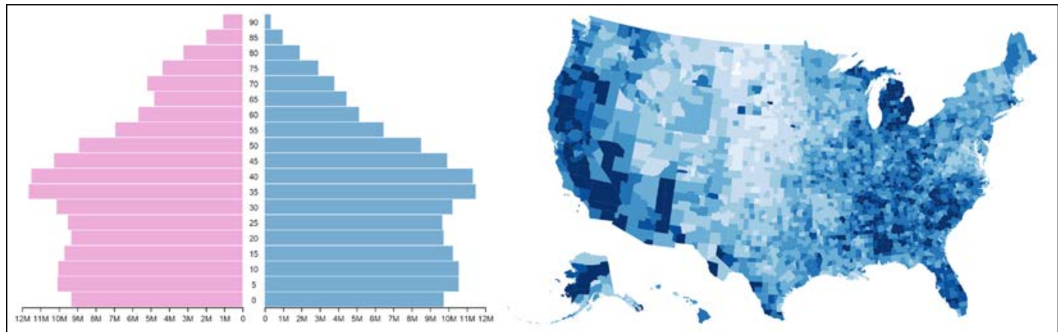
Bokeh

Here are a few references:

- Main website at `http://bokeh.pydata.org/en/latest/index.html`
- Gallery at `http://bokeh.pydata.org/en/latest/docs/gallery.html`

Vincent and Vega

Vega is a language-agnostic visualization grammar. Vega figures can be converted to interactive HTML visualizations. The **Vincent** library makes it easy to write Vega figures from Python.



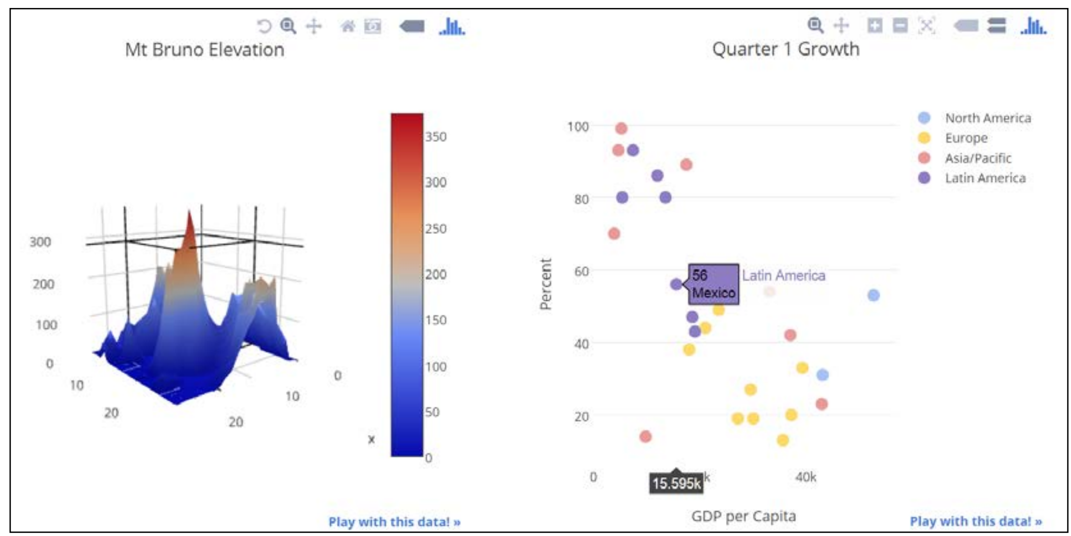
Vega and Vincent

Here are some references:

- `https://github.com/trifacta/vega`
- `https://github.com/wrobstory/vincent`

Plotly

Plotly (`https://plot.ly/`) is a commercial online service providing APIs and libraries for creating and sharing plots on the web. There is a Python library for creating and displaying interactive visualizations in the Notebook.



plotly

Let's also mention a few other young libraries:

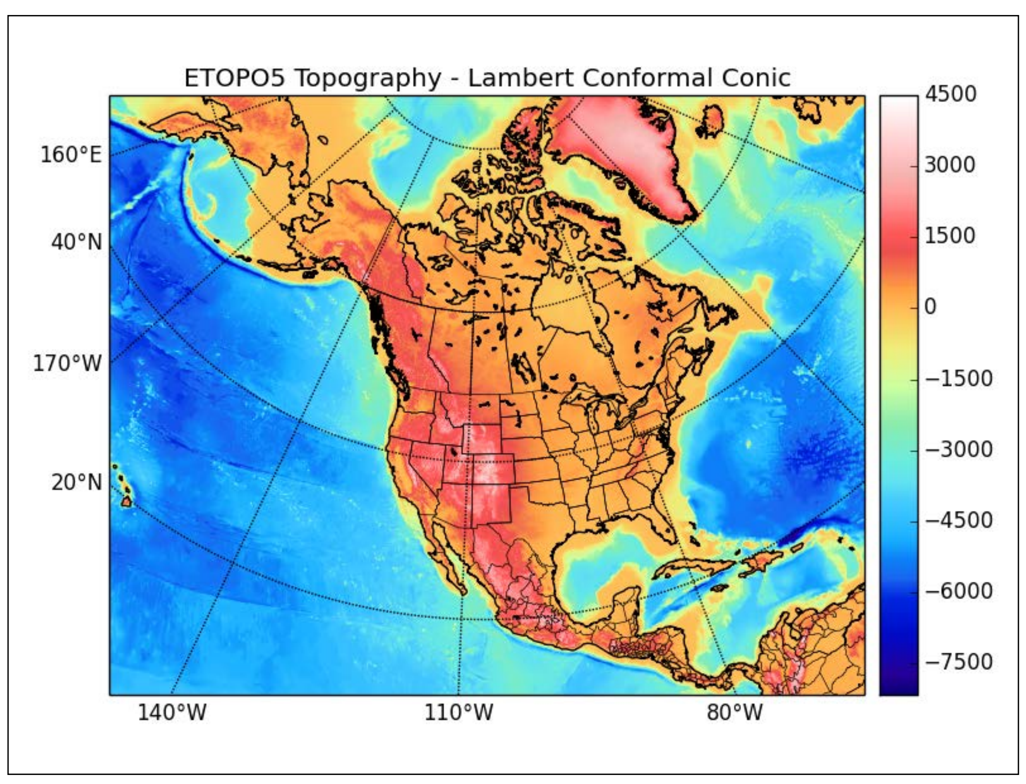
- Lightning at `http://lightning-viz.org/`
- toyplot at `https://toyplot.readthedocs.org/en/latest/`
- bqplot at `https://github.com/bloomberg/bqplot`

Maps and geometry

There are many ways to create maps in Python.

The matplotlib Basemap toolkit

Basemap is a matplotlib plugin that allows you to plot data on maps. Several projection methods are supported.



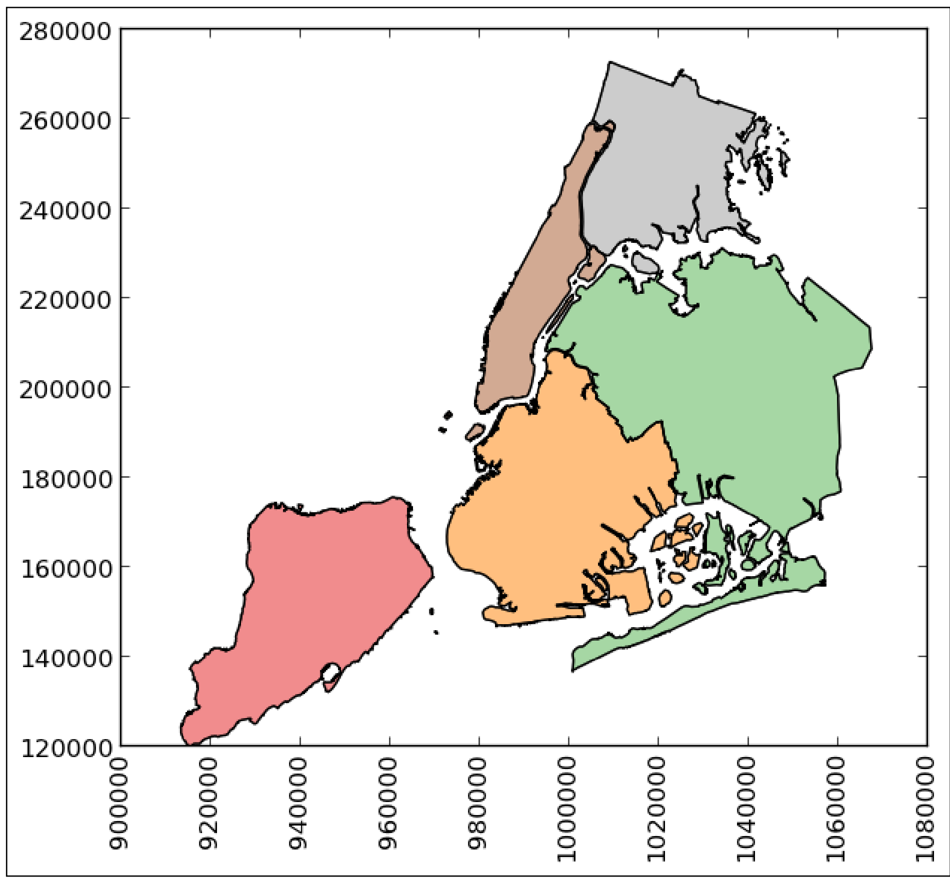
Basemap

Here are some links:

- Basemap main page at `http://matplotlib.org/basemap/`
- Gallery at `http://matplotlib.org/basemap/users/examples.html`

GeoPandas

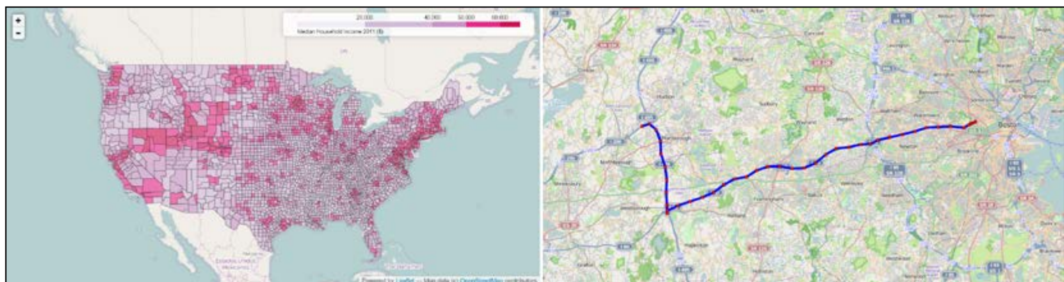
GeoPandas (`https://github.com/geopandas/geopandas`) adds support for geographic data in pandas. It leverages the **shapely** library for geometric manipulations.



GeoPandas

Leaflet wrappers: folium and mplleaflet

Leaflet is a JavaScript library for creating interactive maps. Several Python projects allow you to plot data on interactive Leaflet maps and to integrate them in the Notebook. For example, **folium** integrates well with Vincent and pandas, while **mplleaflet** lets us display matplotlib plots on a map.



folium and mplleaflet

Here are a few references:

- Leaflet library at `http://leafletjs.com/`
- Folium main page at `http://folium.readthedocs.org/en/latest/`
- mplleaflet at `https://github.com/jwass/mplleaflet`

3D visualization

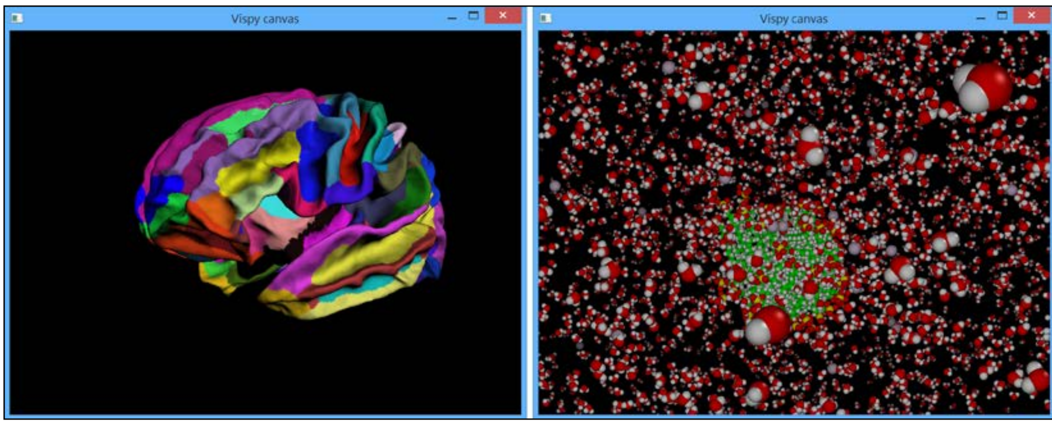
Here are a couple of 3D visualization libraries.

Mayavi

Mayavi (`http://docs.enthought.com/mayavi/mayavi/`) is a 3D plotting library based on VTK, a C++ visualization toolkit. Mayavi features a scriptable GUI for exploring three-dimensional data interactively.

VisPy

VisPy is a pure Python 2D/3D plotting library designed for high-performance interactive visualization. Based on OpenGL, it features a modular architecture that lets advanced users access OpenGL features such as GLSL shaders with a Pythonic interface.



VisPy

Here are a few links:

- Main page at `http://vispy.org`
- Gallery at `http://vispy.org/gallery.html`
- Tutorial at `http://ipython-books.github.io/featured-06/`

Summary

In this chapter, we reviewed several options in Python for plotting, visualization, and graphical interfaces. There are many more details in the *IPython Cookbook* (`http://ipython-books.github.io/cookbook/`).

In the next chapter, we will cover high-performance and parallel computing in Python.

5
High-Performance and Parallel Computing

As an interpreted and dynamic language, Python is slower than C, C++, or Fortran, especially when using loops. Thus, numerical algorithms written in pure Python are generally too slow to be useful. As we saw in *Chapter 3*, *Numerical Computing with NumPy*, NumPy solves this problem by offering fast vector computations on array structures.

Some algorithms cannot be easily vectorized with NumPy. Using Python loops is then required. The two main solutions to make loops fast in a context of numerical computing are the following: using a JIT compiler like Numba, or using Cython to translate these loops to C.

Another general method for making computations faster is to distribute jobs across the multiple processors on a multicore computer.

In this chapter, we will cover all of these topics:

- Accelerating Python code with Numba
- Writing C in Python with Cython
- Distributing tasks on several cores with IPython.parallel
- Further high-performance computing techniques

Accelerating Python code with Numba

When it is too difficult or impossible to vectorize an algorithm, you often need to use Python loops. However, Python loops are slow. Fortunately, Numba provides a **Just-In-Time** (**JIT**) compiler that can compile pure Python code straight to machine code thanks to the LLVM compiler architecture. This can result in massive speedups.

In this section, we'll see how to use Numba to accelerate a mathematical modeling simulation.

To install numba, just type `conda install numba` on the command-line.

Let's first import a few packages:

```
In [1]: import math
        import random
        import numpy as np
        from numba import jit, vectorize, float64
        import matplotlib.pyplot as plt
        import seaborn
        %matplotlib inline
```

Random walk

We will simulate a random walk with jumps. A particle is on the real line, starting at 0. At every time step, the particle makes a step to the right or to the left. If the particle crosses a threshold, it is reset at its initial position. This type of stochastic model is notably used in neuroscience. Without the threshold, this model is called a **brownian motion**. Although a brownian motion can be efficiently simulated in NumPy with `np.cumsum()`, a stochastic model with a threshold and jumps requires a loop.

The following random function returns a random `-1` or `+1` value.

```
In [2]: def step():
            return 1. if random.random() > .5 else -1.
```

Let's write the simulation in pure Python. The function `walk()` takes a number of steps as input. At every time step, the function adds a random step to the previous position in order to get the new position. An `if` statement implements the threshold and jump.

```
In [3]: def walk(n):
            x = np.zeros(n)
```

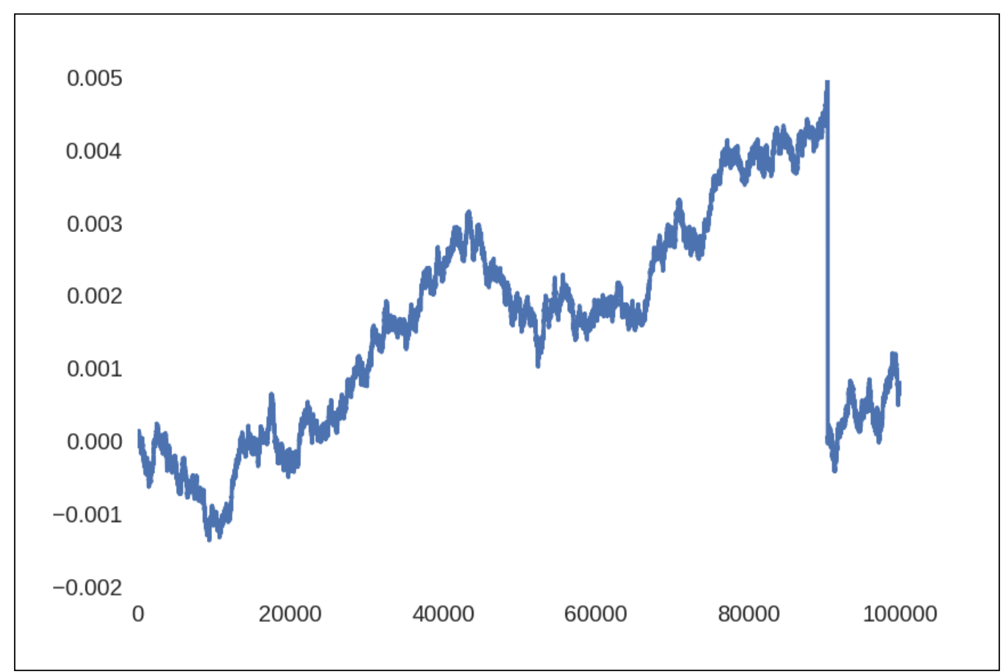
```
    dx = 1. / n
    for i in range(n - 1):
        x_new = x[i] + dx * step()
        if x_new > 5e-3:
            x[i + 1] = 0.
        else:
            x[i + 1] = x_new
    return x
```

Let's run this function:

```
In [4]: n = 100000
        x = walk(n)
```

Here is a screenshot of the trajectory:

```
In [5]: plt.plot(x)
```



A random walk with jumps

How long did it take to simulate this trajectory?

```
In [6]: %%timeit
        walk(n)
Out[6]: 10 loops, best of 3: 57.6 ms per loop
```

Now, let's JIT-compile this function with Numba.

```
In [7]: @jit(nopython=True)
        def step_numba():
            return 1. if random.random() > .5 else -1.
In [8]: @jit(nopython=True)
        def walk_numba(n):
            x = np.zeros(n)
            dx = 1. / n
            for i in range(n - 1):
                x_new = x[i] + dx * step_numba()
                if x_new > 5e-3:
                    x[i + 1] = 0.
                else:
                    x[i + 1] = x_new
            return x
```

All we had to do was to add a `@jit` decorator on top of the two functions. The body of the functions remain the same between the pure Python and the numba versions (except that we call `step_numba()` instead of `step()` in the main function). We'll explain the `nopython=True` argument below.

Let's evaluate the performance of this compiled function:

```
In [9]: %%timeit
        walk_numba(n)
Out[9]: The slowest run took 81.94 times longer than the fastest.
        This could mean that an intermediate result is being cached
        1000 loops, best of 3: 1.89 ms per loop
```

This is a 30x speed improvement. IPython tells us that the first call was much slower. This is because the function was compiled on-the-fly the first time we called it (there are plans to support ahead-of-time compilation in future versions of Numba). Hence, Numba is most effective when a given function needs to be called many times.

The `nopython=True` argument is not strictly necessary. Numba can compile a Python function in two modes: Python mode and **nopython mode**. In Python mode, the compiled code relies on the CPython interpreter. In nopython mode however, the code is compiled to standalone machine code that doesn't rely on CPython. Although this leads to much faster code, the nopython mode is much more limited than the Python mode. Many Python data structures such as lists and dictionaries are not currently available in nopython mode. However, Numba is designed from the ground up to support NumPy arrays in both modes. Trying to stick with the Python subset supported in nopython mode is highly recommended when seeking to achieve the best performance.

Numba needs to know the exact types of the function's parameters, return values, and internal variables. It uses type inference to find out the types automatically when possible, but you can also specify the input and output types explicitly. You will find more details in the documentation:

- Numba main page at `http://numba.pydata.org`
- Numba documentation at `http://numba.pydata.org/numba-doc/dev/index.html`
- Python features supported in nopython mode at `http://numba.pydata.org/numba-doc/dev/reference/pysupported.html`
- NumPy features supported in nopython mode at `http://numba.pydata.org/numba-doc/dev/reference/numpysupported.html`

Universal functions

Numba also supports the creation of **NumPy universal functions** (**ufuncs**) with the `@vectorize` decorator. This feature lets you turn a Python function implementing a mathematical scalar operation into a vectorized function that works on NumPy arrays on an element-wise basis.

Here is an example. We want to compute a complex mathematical expression on a NumPy array. The standard way of doing it with NumPy is inefficient because many array copies are silently performed during the temporary steps.

```
In [10]: x = np.random.rand(10000000)
         %timeit np.cos(2*x**2 + 3*x + 4*np.exp(x**3))
Out[10]: 1 loops, best of 3: 689 ms per loop
```

We can use the `@vectorize` decorator to define a new universal function:

```
In [11]: @vectorize
         def kernel(x):
             return np.cos(2*x**2 + 3*x + 4*np.exp(x**3))
In [12]: kernel(1.)
Out[12]: -0.98639139715432589
```

This function can now be applied on a NumPy array:

```
In [13]: %timeit kernel(x)
Out[13]: 1 loops, best of 3: 324 ms per loop
```

This function is about twice as fast as the standard NumPy version because temporary array copies are avoided.

It is possible to make this computation even faster by taking advantage of multicore processors and **Graphics Processing Units** (**GPUs**).

Let's illustrate this by using another package called **numexpr** (`https://github.com/pydata/numexpr`), which is similar but older than Numba. It can be installed with `conda install numexpr`.

```
In [14]: import numexpr
         %timeit numexpr.evaluate('cos(2*x**2 + 3*x + 4*exp(x**3))')
Out[14]: 10 loops, best of 3: 122 ms per loop
```

The `evaluate()` function takes a string as input, which is slightly less convenient than Numba's decorators. However, it uses all available cores by default, which explains why it is several times faster than Numba here.

We can check the number of detected cores as follows:

```
In [15]: numexpr.detect_number_of_cores()
Out[15]: 4
```

Here are a few references:

- Universal functions with Numba at `http://numba.pydata.org/numba-doc/dev/user/vectorize.html`
- Numexpr documentation at `https://github.com/pydata/numexpr/wiki/Numexpr-Users-Guide`

Writing C in Python with Cython

Cython is a Python library that lets you combine C and Python in various ways. There are two main use-cases:

- Wrapping a C/C++ library in Python
- Optimizing your Python code by statically compiling it to C

In this section, we will demonstrate the second use-case. You will find an example of the first use-case in the *IPython Cookbook* and at `http://docs.cython.org/src/tutorial/index.html`.

Installing Cython and a C compiler for Python

If you use Anaconda, you should already have Cython (you can always do `conda install cython` to check).

For Cython to work, you need a C compiler compatible with your version of Python. This is much easier on Unix systems. Here are the instructions given at `http://docs.cython.org/src/quickstart/install.html`:

- On Linux, you can install the **GNU C Compiler (gcc)** via the OS package manager. On Ubuntu or Debian, for example, type `sudo apt-get install build-essential`.
- On Mac OS X, you can install Apple's Xcode from `http://developer.apple.com`.

- On Windows, installing a C compiler compatible with your version of Python and setting it up correctly is generally difficult. We'll mention two methods:
    - The **easy way (recommended)** requires Python 2.7, so you need to switch to your Python 2 py2 conda environment. You just need to install the **Microsoft Visual C++ Compiler for Python 2.7** freely available at `http://www.microsoft.com/en-us/download/details.aspx?id=44266`.
    - The **hard way** works with any (64-bit) version of Python. You will find all instructions at `https://github.com/cython/cython/wiki/64BitCythonExtensionsOnWindows`. Briefly, this method requires you to install the free **Windows SDK C/C++ compiler** adapted to your version of Python, and setting a few things up on the terminal before launching Python or IPython.

Implementing the Eratosthenes Sieve in Python and Cython

We'll implement the Eratosthenes Sieve algorithm (`https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`) to find all prime numbers smaller than a given number. The first version is coded in pure Python.

```
In [1]: def primes_python(n):
            primes = [False, False] + [True] * (n - 2)
            i = 2
            while i < n:
                # We do not deal with composite numbers.
                if not primes[i]:
                    i += 1
                    continue
                k = i * i
                # We mark multiples of i as composite numbers.
                while k < n:
                    primes[k] = False
                    k += i
                i += 1
            # We return all numbers marked with True.
            return [i for i in range(2, n) if primes[i]]
```

Here is an example:

```
In [2]: primes_python(20)
Out[2]: [2, 3, 5, 7, 11, 13, 17, 19]
```

Let's evaluate the performance of this first version.

```
In [3]: n = 10000
In [4]: %timeit primes_python(n)
Out[4]: 100 loops, best of 3: 4 ms per loop
```

Now, we load the Cython extension to write Cython code right in the Notebook:

```
In [5]: %load_ext Cython
```

All we need to do is to add `%%cython` in the first line of the cell, as shown here:

```
In [6]: %%cython
        def primes_cython_1(n):
            primes = [False, False] + [True] * (n - 2)
            i = 2
            while i < n:
                # We do not deal with composite numbers.
                if not primes[i]:
                    i += 1
                    continue
                k = i * i
                # We mark multiples of i as composite numbers.
                while k < n:
                    primes[k] = False
                    k += i
                i += 1
            # We return all numbers marked with True.
            return [i for i in range(2, n) if primes[i]]
```

When we add `%%cython` at the beginning of the cell, the code gets compiled by Cython into a C extension. Then, this extension is loaded, and the compiled function is readily available in the interactive namespace.

```
In [7]: primes_cython_1(20)
Out[7]: [2, 3, 5, 7, 11, 13, 17, 19]
In [8]: %timeit primes_cython_1(n)
Out[8]: 100 loops, best of 3: 1.99 ms per loop
```

We achieve a twofold speed improvement.

Now, we will specify the type of each local variable so that Cython can optimize the code more efficiently.

```
In [9]: %%cython -a
        def primes_cython_2(int n):
            # Note the type declarations below:
            cdef list primes = [False, False] + [True] * (n - 2)
            cdef int i = 2
            cdef int k = 0
            # The rest of the function is unchanged.
            while i < n:
                # We do not deal with composite numbers.
                if not primes[i]:
                    i += 1
                    continue
                k = i * i
                # We mark multiples of i as composite numbers.
                while k < n:
                    primes[k] = False
                    k += i
                i += 1
            # We return all numbers marked with True.
            return [i for i in range(2, n) if primes[i]]
In [10]: %timeit primes_cython_2(n)
Out[10]: 1000 loops, best of 3: 266 µs per loop
```

This time, we achieve a 15x speed improvement just by specifying the variable types with `cdef`. This new keyword is one of the specific language constructs brought by Cython. Cython is therefore a superset of the Python language, bringing new syntax constructs to optimize the compilation process.

In general, Cython will be the most efficient when it can compile data structures and operations directly to C by making as few CPython API calls as possible. Specifying the types of the variables often leads to greater speed improvements.

The `-a` option passed to the `%%cython` cell magic displays some **annotations** telling you which lines are the least efficiently compiled to C. By clicking on a line, you can see the generated C code corresponding to that line, as shown in the following screenshot:

```
Generated by Cython 0.21
+01: def primes_cython_2(int n):
 02:     # Note the type declarations below:
+03:     cdef list primes = [False, False] + [True] * (n - 2)
+04:     cdef int i = 2
+05:     cdef int k = 0
 06:     # The rest of the function is unchanged.
+07:     while i < n:
 08:         # We do not deal with composite numbers.
+09:         if not primes[i]:
    __pyx_t_3 = __Pyx_GetItemInt_List(__pyx_v_primes, __pyx_v_i, int, 1, __Pyx_PyInt_From_int, 1, 1, 1); if (unli
  kely(__pyx_t_3 == NULL)) {__pyx_filename = __pyx_f[0]; __pyx_lineno = 9; __pyx_clineno = __LINE__; goto __pyx_L1_
  error;};
    __Pyx_GOTREF(__pyx_t_3);
    __pyx_t_4 = __Pyx_PyObject_IsTrue(__pyx_t_3); if (unlikely(__pyx_t_4 < 0)) {__pyx_filename = __pyx_f[0]; __py
  x_lineno = 9; __pyx_clineno = __LINE__; goto __pyx_L1_error;}
    __Pyx_DECREF(__pyx_t_3); __pyx_t_3 = 0;
    __pyx_t_5 = ((!__pyx_t_4) != 0);
    if (__pyx_t_5) {
+10:             i += 1
+11:             continue
+12:         k = i * i
 13:         # We mark multiples of i as composite numbers.
+14:         while k < n:
+15:             primes[k] = False
+16:             k += i
+17:         i += 1
 18:     # We return all numbers marked with True.
+19:     return [i for i in range(2, n) if primes[i]]
```

Cython annotations

There is much more to say about Cython, including:

- Support for NumPy arrays
- Support for multicore processors with OpenMP
- Wrapping C/C++ libraries and code from Python

The *IPython Cookbook* contains several recipes on Cython covering these topics.

Here are a few further references:

- Cython documentation at `http://docs.cython.org/`
- Cython user guide at `http://docs.cython.org/src/userguide/index.html`
- Cython tutorials at `http://docs.cython.org/src/tutorial/index.html`

Distributing tasks on several cores with IPython.parallel

In the previous sections, we covered a few methods to accelerate Python code. Here, we will see how to run multiple tasks in parallel on a multicore computer. IPython implements highly-powerful and user-friendly facilities for interactive parallel computing in the Notebook.

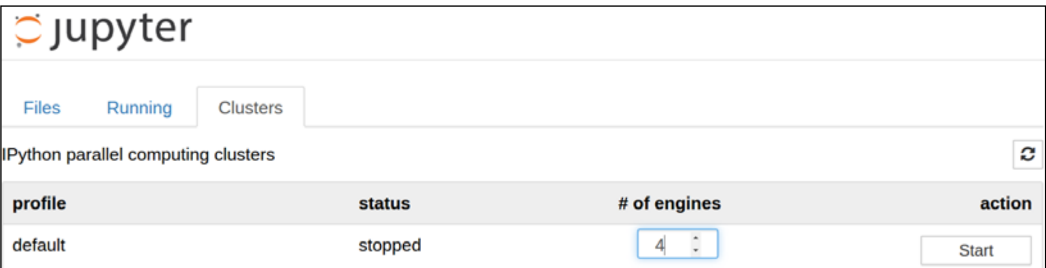
We first need to install ipyparallel (also called IPython.parallel) with `conda install ipyparallel`. Next, let's import NumPy and ipyparallel:

```
In [1]: import numpy as np
        # ipyparallel was IPython.parallel before IPython 4.0
        from ipyparallel import Client
```

To use IPython.parallel, we need to launch a few engines.

The first way to do it is to run `ipcluster start` in the terminal.

You can also launch engines from the Notebook dashboard. However, you first need to add `c.NotebookApp.server_extensions.append('ipyparallel.nbextension')` in the file `~/.jupyter/jupyter_notebook_config.py` (you may need to create this file). Then, from the Notebook dashboard (accessible at `http://localhost:8888` in your browser's address bar), click on the **Clusters** tab, select the number of engines you want to launch, and click on the **Start** button. Here is a screenshot:



Launching IPython.parallel engines from the Notebook dashboard

In general, you can launch as many engines as the number of CPUs you have on your machine.

Once the engines have been launched, we create a `Client` instance. This object will give us access to these engines:

```
In [2]: rc = Client()
```

There are two ways to access the engines:

- With the **direct interface**, we have a direct access to every engine.
- With the **load-balanced interface**, we submit jobs to a scheduler which dynamically assigns them to the engines depending on their current load.

Let's first demonstrate how to use the direct interface.

Direct interface

The `ids` attribute of the client shows us the identifiers of the engines that were automatically detected by IPython:

```
In [3]: rc.ids
Out[3]: [0, 1, 2, 3]
```

There are several ways to run code in parallel on the engines. First, we can use the `%px` magic command:

```
In [4]: %px import os, time
In [5]: %px print(os.getpid())
Out[5]: [stdout:0] 11173
        [stdout:1] 11174
        [stdout:2] 11175
        [stdout:3] 11176
```

The code passed to the `%px` magic command is executed on all engines. Here, we display the OS **process identifier** (also called **PID**) of every engine. Every engine is an independent Python process.

We can also specify the exact list of engines to run code on. The `--targets` option accepts a list of engine identifiers. The Python slicing syntax is also supported, as shown in the following example where we select all engines except the last one:

```
In [6]: %%px --targets :-1
        print(os.getpid())
Out[6]: [stdout:0] 11173
        [stdout:1] 11174
        [stdout:2] 11175
```

Note that we used the cell magic `%%px` this time instead of the line magic. The cell magic allows us to execute several lines of code on the engines.

> **%pxconfig magic command**
>
> We can also use the `%pxconfig` magic command to configure the parallel interface, specifying the list of engines and the blocking/non-blocking execution mode (see the discussion about the synchronous and asynchronous execution modes in the next subsection).

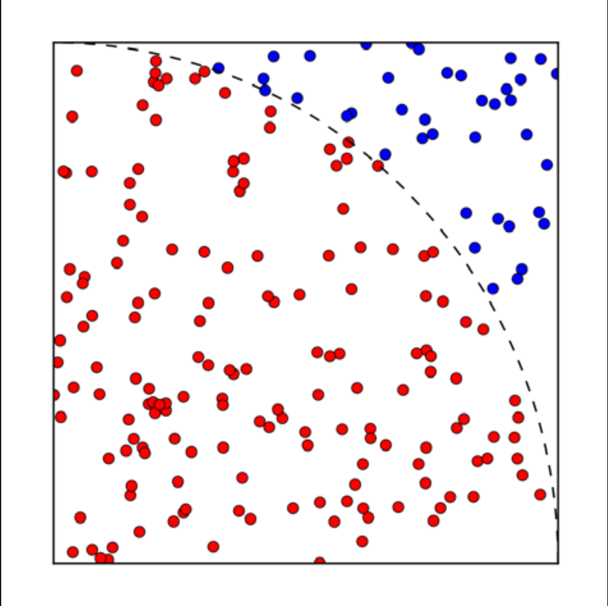
We can also create a direct view on some or all of the engines:

```
In [7]: view = rc[:-1]
        view
Out[7]: <DirectView [0, 1, 2]>
```

The direct view has a few useful methods like parallel versions of `map()` and `apply()`. These routines are also available with the load-balanced interface. We will show a few examples in the next subsection.

Load-balanced interface

The load-balanced interface gives us high-level parallel computing routines that are dynamically executed on the engines. Here, we will estimate pi in parallel using a Monte-Carlo method. Specifically, we will sample a large number of points uniformly in a square, and estimate the proportion of those which are in a quarter disc. We'll then get an estimation of pi since we know that this proportion should be `pi/4`:



Estimating pi with a Monte-Carlo method

Let's first create a balanced view:

```
In [8]: v = rc.load_balanced_view()
```

The following function samples and counts the number of points in the quarter disc:

```
In [9]: def sample(n):
            import numpy as np
            # Random coordinates.
            x, y = np.random.rand(2, n)
            # Square distances to the origin.
            r_square = x ** 2 + y ** 2
            # Number of points in the quarter disc.
            return (r_square <= 1).sum()
```

Note that we import NumPy in the body of the function to make sure that NumPy is imported on every node. We could also import the package once for all at the beginning of the session with a direct or load-balanced view.

The second function below returns an estimation of pi based on the number of points in the quarter disc, and the total number of points:

```
In [10]: def pi(n_in, n):
             return 4. * float(n_in) / n
```

Here is an example:

```
In [11]: n = 100000000
In [12]: pi(sample(n), n)
Out[12]: 3.14174968
```

Let's evaluate the time taken by this function on a single core:

```
In [13]: %timeit pi(sample(n), n)
Out[13]: 1 loops, best of 3: 2.65 s per loop
```

We will now run this simulation in parallel. First, we divide this task into 100 smaller subtasks where the number of points is divided by 100:

```
In [14]: args = [n // 100] * 100
```

We use a parallel `map()` function to run these tasks in parallel. Our `sample()` function is called 100 times, taking `n // 100` as its argument every time. We will combine the 100 results later.

```
In [15]: ar = v.map(sample, args)
```

This function doesn't return the results. Instead, it launches the 100 tasks in parallel and returns an `AsyncResult` object. We say that this function is **asynchronous**. The `AsyncResult` object can be used to interactively poll the tasks status and eventually retrieve the results. There's also a **synchronous** version of `map()` called `map_sync()` which blocks until the tasks have completed, and directly returns the results.

Here is an example:

```
In [16]: ar.ready(), ar.progress
Out[16]: (False, 12)
```

This tells us that the tasks are still running at this point, and that 12 tasks have completed so far.

Blocking call

We can use `ar.wait()` to block the interactive session until all tasks have been completed.

Once all tasks have completed, we can get some information about the elapsed time:

```
In [17]: ar.elapsed, ar.serial_time
Out[17]: (1.428284, 4.042367000000002)
```

The first number represents the actual elapsed time for the entire job, while the second number represents the cumulative time spent on all engines. In other words, it is approximately the time that would have taken our job if we had run it on a single core.

Finally, we combine all results with the `ar.result` property. This is the list of all results returned by the 100 tasks. We use the `pi()` function to get the final estimation:

```
In [18]: pi(np.sum(ar.result), n)
Out[18]: 3.141666
```

Note that this method of estimating pi is not particularly efficient to say the least! You'll find an overview of other approximation methods at `http://en.wikipedia.org/wiki/Approximations_of_%CF%80`.

There is much more to say about IPython.parallel. You'll find more details in the following references:

- Documentation of IPython.parallel at `http://ipyparallel.readthedocs.org/en/latest/`
- *IPython Cookbook*, *Chapter 5*, *High-Performance and Parallel Computing*

Further high-performance computing techniques

There are many other high-performance computing techniques than those covered in this chapter. The *IPython Cookbook* contains many more details. Here is an overview of some of these other techniques:

MPI

The **Message Passing Interface**, or **MPI**, defines communication protocols for high-performance distributed systems. IPython.parallel has native support for MPI. Here are some other references:

- MPI tutorial at `http://mpitutorial.com/`
- MPI with IPython at `http://ipython.org/ipython-doc/dev/parallel/parallel_mpi.html`

Distributed computing

There are many frameworks for distributed computing and big data analysis in Python.

- Apache Spark is a big data framework that can run on Hadoop and has a Python API: `http://spark.apache.org/`.
- Dask is a generic and modular parallel computing framework: `http://dask.pydata.org/en/latest/`.
- Let's also mention xray, which provides a labeled array data structure that can work with Dask: `http://xray.readthedocs.org/en/stable/`.
- Bolt is an experimental project providing a uniform interface to local or distributed ndarrays: `http://bolt-project.org/`.

C/C++ with Python

There are many ways to interoperate Python and C code together:

- **Cython** can be used to access C (`http://docs.cython.org/src/userguide/external_C_code.html`) and C++ (`http://docs.cython.org/src/userguide/wrapping_CPlusPlus.html`) code from Python. **It is currently the recommended choice over the older methods below**.
- **SWIG** (`http://www.swig.org/`) can connect C/C++ libraries to several high-level languages like Python.
- **weave** (`https://github.com/scipy/weave`) is a SciPy-based library for integrating C code in Python. It is deprecated and remains available for legacy code.
- The Python C API gives low-level access to the CPython interpreter. It can be used to combine Python and C code.

Here are two libraries that give access to compiled C libraries:

- **ctypes** is a native Python library that allows calling functions in DLLs or shared libraries.
- **cffi** (`https://cffi.readthedocs.org/en/latest/`) is a more recent alternative to ctypes.

GPU computing

Graphics Processing Units (**GPUs**) are powerful devices found in all recent computers and mobile devices. They are primarily used for video games. However, they can also be used for general-purpose high-performance numerical computing, also called **GPGPU** for **General-Purpose GPU** computing. The massively parallel architecture of GPUs makes them suitable to a large class of scientific problems.

CUDA (by NVIDIA Corporation) and OpenCL (by the Khronos Group) are two sets of libraries and APIs that offer a C-like syntax for GPGPU programming. There are Python libraries that give access to the CUDA and OpenCL libraries:

- **PyCUDA** at `http://mathema.tician.de/software/pycuda/`
- **PyOpenCL** at `http://mathema.tician.de/software/pyopencl/`

Let's also mention another library from Continuum Analytics that gives high-level access to GPGPU programming:

- **libdynd** (`https://github.com/libdynd/libdynd`) is a C++ dynamic ndarray library with GPU support. It can also be used from Python.

PyPy

CPython is the main Python implementation. It is written in C. **PyPy** (`http://pypy.org`) is another implementation of Python. It is generally much faster than CPython. However, it is less easily compatible with Python C extensions like NumPy, although there is currently some work in progress in this direction.

Julia

Julia (`http://julialang.org/`) is a young high-level language designed for high-performance numerical computing. Julia supports vectorized array operations like NumPy. Contrary to Python, `for` loops in Julia can be as fast as array operations. This is due to Julia code being JIT compiled to machine code through the LLVM compiler architecture. This is the same approach followed by Numba on Python code.

There are ways to call Julia code from Python and to call Python code from Julia. There is also an IJulia kernel (`https://github.com/JuliaLang/IJulia.jl`) that works with the Jupyter Notebook.

Summary

In this chapter, we covered some of the main high-performance computing methods in Python. Numba is one of the easiest and most efficient options. Cython is useful with more complex use-cases and when it is necessary to leverage C/C++ code. Also, IPython.parallel allows us to leverage multicore CPUs or multiple computers for independent tasks. Finally, we discussed further high-performance computing techniques.

In the next chapter, we will explore a few customization options in IPython and the Notebook.

6
Customizing IPython

The Jupyter Notebook is a highly-customizable platform. You can configure many aspects of the software in your configuration files. You can also extend the backend (kernels) and the frontend (HTML-based Notebook). This allows you to create highly-personalized user experiences based on the Notebook.

In this chapter, we will cover the following topics:

- Creating a custom magic command in an IPython extension
- Writing a new Jupyter kernel
- Displaying rich HTML elements in the Notebook
- Customizing the Notebook interface with JavaScript

Creating a custom magic command in an IPython extension

IPython comes with a rich set of magic commands. You can get the complete list with the `%lsmagic` command. IPython also allows you to create your own magic commands. In this section, we will create a new cell magic that compiles and executes C++ code in the Notebook.

We first import the `register_cell_magic` function:

```
In [1]: from IPython.core.magic import register_cell_magic
```

To create a new cell magic, we create a function that takes a line (containing possible options) and a cell's contents as its arguments, and we decorate it with `@register_cell_magic`, as shown here:

```
In [2]: @register_cell_magic
        def cpp(line, cell):
            """Compile, execute C++ code, and return the
            standard output."""
            # We first retrieve the current IPython interpreter
            # instance.
            ip = get_ipython()

            # We define the source and executable filenames.
            source_filename = '_temp.cpp'
            program_filename = '_temp'

            # We write the code to the C++ file.
            with open(source_filename, 'w') as f:
                f.write(cell)

            # We compile the C++ code into an executable.
            compile = ip.getoutput("g++ {0:s} -o {1:s}".format(
                source_filename, program_filename))

            # We execute the executable and return the output.
            output = ip.getoutput('./{0:s}'.format(program_filename))

            print('\n'.join(output))
```

C++ compiler

This recipe requires the **gcc** C++ compiler. On Ubuntu, type `sudo apt-get install build-essential` in a terminal. On OS X, install Xcode. On Windows, install MinGW (`http://www.mingw.org`) and make sure that g++ is in your system path.

This magic command uses the `getoutput()` method of the IPython **InteractiveShell instance**. This object represents the current interactive session. It defines many methods for interacting with the session. You will find the comprehensive list at `http://ipython.org/ipython-doc/dev/api/generated/IPython.core.interactiveshell.html#IPython.core.interactiveshell.InteractiveShell`.

Let's now try this new cell magic.

```
In [3]: %%cpp
        #include<iostream>
        int main()
        {
            std::cout << "Hello world!";
        }
Out[3]: Hello world!
```

This cell magic is currently only available in your interactive session. To distribute it, you need to create an **IPython extension**. This is a regular Python module or package that extends IPython.

To create an IPython extension, copy the definition of the `cpp()` function (without the decorator) to a Python module, named `cpp_ext.py` for example. Then, add the following at the end of the file:

```
def load_ipython_extension(ipython):
    """This function is called when the extension is loaded.
    It accepts an IPython InteractiveShell instance.
    We can register the magic with the `register_magic_function`
    method of the shell instance."""
    ipython.register_magic_function(cpp, 'cell')
```

Then, you can load the extension with `%load_ext cpp_ext`. The `cpp_ext.py` file needs to be in the `PYTHONPATH`, for example in the current directory.

Writing a new Jupyter kernel

Jupyter supports a wide variety of kernels written in many languages, including the most-frequently used IPython. The Notebook interface lets you choose the kernel for every notebook. This information is stored within each notebook file.

The `jupyter kernelspec` command allows you to get information about the kernels. For example, `jupyter kernelspec list` lists the installed kernels. Type `jupyter kernelspec --help` for more information.

At the end of this section, you will find references with instructions to install various kernels such as IR, IJulia, or IHaskell. Here, we will detail how to create a custom kernel.

There are two methods to create a new kernel:

- Writing a kernel from scratch for a new language by reimplementing the whole Jupyter messaging protocol.
- Writing a **wrapper kernel** for a language that can be accessed from Python.

We will use the second, easier method in this section. Specifically, we will reuse the example from the last section to write a C++ wrapper kernel.

We need to slightly refactor last section's code because we won't have access to the `InteractiveShell` instance. Since we're creating a kernel, we need to put the code in a Python script in a new folder named `cpp`:

```
In [1]: %mkdir cpp
```

The `%%writefile` cell magic lets us create a `cpp_kernel.py` Python script from the Notebook:

```
In [2]: %%writefile cpp/cpp_kernel.py

        import os
        import os.path as op
        import tempfile

        # We import the `getoutput()` function provided by IPython.
        # It allows us to do system calls from Python.
        from IPython.utils.process import getoutput
```

```
        def exec_cpp(code):
            """Compile, execute C++ code, and return the standard
            output."""
            # We create a temporary directory. This directory will
            # be deleted at the end of the 'with' context.
            # All created files will be in this directory.
            with tempfile.TemporaryDirectory() as tmpdir:

                # We define the source and executable filenames.
                source_path = op.join(tmpdir, 'temp.cpp')
                program_path = op.join(tmpdir, 'temp')

                # We write the code to the C++ file.
                with open(source_path, 'w') as f:
                    f.write(code)

                # We compile the C++ code into an executable.
                os.system("g++ {0:s} -o {1:s}".format(
                    source_path, program_path))

                # We execute the program and return the output.
                return getoutput(program_path)
Out[2]: Writing cpp/cpp_kernel.py
```

Now we create our wrapper kernel by appending some code to the `cpp_kernel.py` file created above (that's what the `-a` option in the `%%writefile` cell magic is for):

```
In [3]: %%writefile -a cpp/cpp_kernel.py

        """C++ wrapper kernel."""
        from ipykernel.kernelbase import Kernel

        class CppKernel(Kernel):

            # Kernel information.
            implementation = 'C++'
```

```
            implementation_version = '1.0'
            language = 'c++'
            language_version = '1.0'
            language_info = {'name': 'c++',
                             'mimetype': 'text/plain'}
            banner = "C++ kernel"

            def do_execute(self, code, silent,
                           store_history=True,
                           user_expressions=None,
                           allow_stdin=False):
                """This function is called when a code cell is
                executed."""
                if not silent:
                    # We run the C++ code and get the output.
                    output = exec_cpp(code)

                    # We send back the result to the frontend.
                    stream_content = {'name': 'stdout',
                                      'text': output}
                    self.send_response(self.iopub_socket,
                                       'stream', stream_content)
                return {'status': 'ok',
                        # The base class increments the execution
                        # count
                        'execution_count': self.execution_count,
                        'payload': [],
                        'user_expressions': {},
                       }

        if __name__ == '__main__':
            from ipykernel.kernelapp import IPKernelApp
            IPKernelApp.launch_instance(kernel_class=CppKernel)
Out[3]: Appending to cpp/cpp_kernel.py
```

In production code, it would be best to test the compilation and execution, and to fail gracefully by showing an error. See the references at the end of this section for more information.

Our wrapper kernel is now implemented in `cpp/cpp_kernel.py`. The next step is to create a `cpp/kernel.json` file describing our kernel:

```
In [4]: %%writefile cpp/kernel.json
        {
            "argv": ["python",
                     "cpp/cpp_kernel.py",
                     "-f",
                     "{connection_file}"
                    ],
            "display_name": "C++"
        }
Out[4]: Writing cpp/kernel.json
```

The `argv` field describes the command that is used to launch a C++ kernel. More information can be found in the references below.

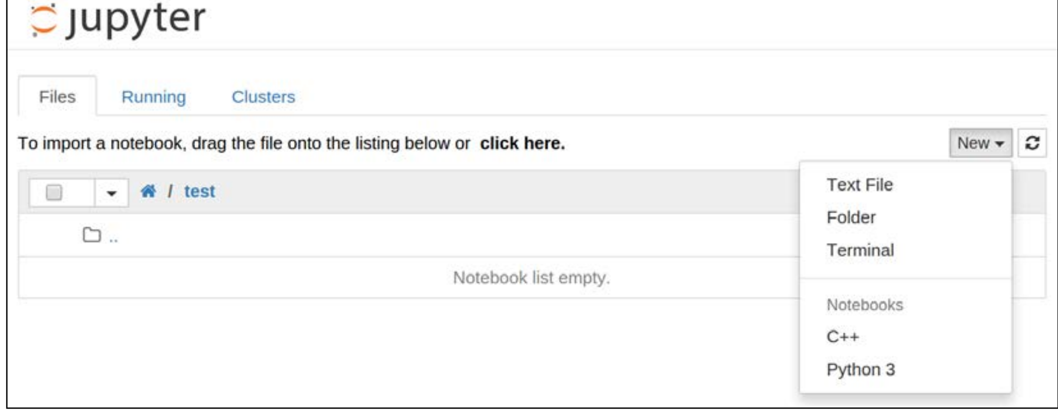
Finally, let's install this kernel with the following command:

```
In [5]: !jupyter kernelspec install --replace --user cpp
Out[5]: [InstallKernelSpec] Installed kernelspec cpp in /Users/cyrille/
Library/Jupyter/kernels/cpp
```

The `--replace` option forces the installation even if the kernel already exists. The `--user` option serves to install the kernel in the user directory. We can test the installation of the kernel with the following command:

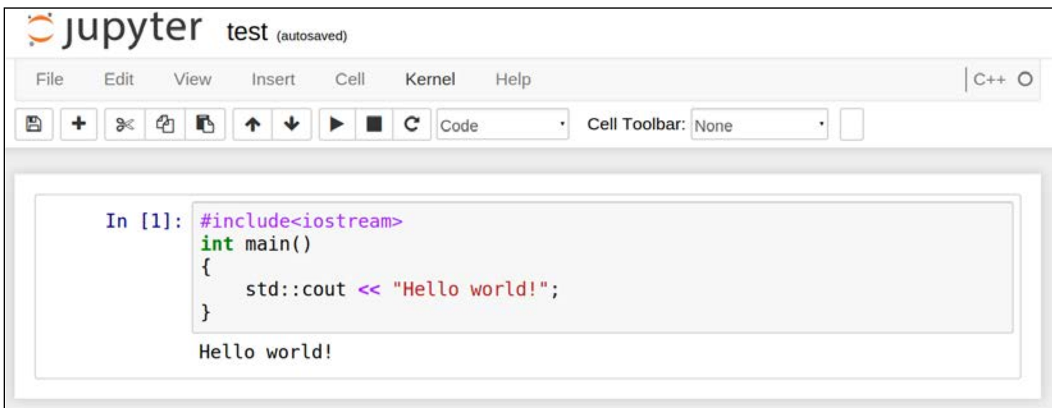
```
In [6]: !jupyter kernelspec list
Out[6]: Available kernels:
          cpp
          python3
```

Now, C++ notebooks can be created in the Notebook, as shown in the following screenshot:



Creating a C++ notebook

C++ code can be written directly in code cells, as shown below:



C++ kernel in the Notebook

Finally, wrapper kernels can also be used in the IPython terminal or the Qt console, using the `--kernel` option, for example `ipython console --kernel cpp`.

Here are a few references:

- Kernel documentation at `http://jupyter-client.readthedocs.org/en/latest/kernels.html`
- Wrapper kernels at `http://jupyter-client.readthedocs.org/en/latest/wrapperkernels.html`

- List of kernels at `https://github.com/ipython/ipython/wiki/IPython%20kernels%20for%20other%20languages`
- bash kernel at `https://github.com/takluyver/bash_kernel`
- R kernel at `https://github.com/takluyver/IRkernel`
- Julia kernel at `https://github.com/JuliaLang/IJulia.jl`
- Haskell kernel at `https://github.com/gibiansky/IHaskell`

Displaying rich HTML elements in the Notebook

The Jupyter Notebook application is based on HTML and runs in a web browser. This platform supports many kinds of rich content such as images, mathematical equations, interactive widgets, videos, and much more. Jupyter proposes several methods to leverage these capabilities.

In this section, we'll show how to display HTML, SVG, and JavaScript elements, notably with the **Data-Driven Documents** (**D3**) JavaScript visualization library.

Displaying SVG in the Notebook

Scalable Vector Graphics (**SVG**) is an open XML-based file format describing vector graphics. Most modern web browsers support this format.

For displaying objects, IPython provides a simple API for representing rich content like SVG. In the following example, we'll define a `Disc` class with a customizable radius and a color. When displaying a `Disc` instance in the Notebook, an SVG representation of the disc will be shown.

Let's first define a function generating the SVG code for a disc:

```
In [1]: def svg_disc(radius, color):
            return """<svg xmlns="http://www.w3.org/2000/svg"
                         version="1.1">
                  <circle cx="{0:d}" cy="{0:d}"
                   r="{0:d}" fill="{1:s}" />
                  </svg>""".format(radius, color)
```

We now define the `Disc` class and implement a special `_repr_svg_()` method that returns the SVG code for that disc.

```
In [2]: class Disc(object):
            def __init__(self, radius, color='red'):
                self.radius = radius
                self.color = color

            def _repr_svg_(self):
                return svg_disc(self.radius, self.color)
```

To display the disc in the Notebook, we create an instance of the Disc class.

```
In [3]: Disc(60, 'purple')
```

Here is a screenshot:

SVG in the Notebook

When IPython displays an object, IPython inspects the object to find `_repr_*()` methods. The formats currently supported by IPython are:

- `svg` (Notebook and Qt console)
- `png` (Notebook and Qt console)
- `jpeg` (Notebook and Qt console)
- `html` (Notebook only)
- `javascript` (Notebook only)
- `latex` (Notebook only)

You will find more information about the rich display system in IPython at `http://ipython.org/ipython-doc/dev/config/integrating.html`.

JavaScript and D3 in the Notebook

There are many JavaScript libraries and frameworks for a wide variety of applications, particularly in the domain of data visualization. They can all potentially be used in the Notebook.

In this subsection, we'll display some data with the popular D3 JavaScript visualization library. We'll dynamically generate the JavaScript code with IPython.

Let's first import a display function in IPython:

```
In [4]: from IPython.display import display_javascript
```

Here is the JavaScript code for our chart:

```
In [5]: JS_TEMPLATE = """
        // We load the d3.js library from the Web.
        require.config({paths: {d3: "http://d3js.org/d3.v3.min"}});
        require(["d3"], function(d3) {
            // Example from http://bost.ocks.org/mike/bar/

            // Define the data.
            var data = %s;

            // We normalize the data.
            var x = d3.scale.linear()
                .domain([0, d3.max(data)])
                .range([0, 420]);

            // We define a categorical color map.
            var color = d3.scale.category10();

            // We create the chart.
            d3.select(".chart")
              .selectAll("div")
                .data(data)
              .enter().append("div")
                .style("width", function(d) { return x(d) + "px"; })
                .text(function(d) { return d; });

        });
        """
```

A course on D3 is beyond the scope of this book. Let's just mention that D3's main idea is to bind data to HTML elements. Here, we create one `<div>` element per item, and set its CSS width to the associated data value. More precisely, this value is converted into a number of pixels via the `x()` D3 scale object.

Let's create some data:

```
In [6]: my_list = [2, 3, 5, 7, 11, 13]
```

We now generate the final JavaScript code by injecting a string representation of the list into the JavaScript template:

```
In [7]: JS = JS_TEMPLATE % str(my_list)
```

Don't try this at home

We're lucky here that the syntax for lists and arrays in Python and JavaScript are basically the same. This explains why we can just inject the Python list into the JavaScript code. In production code, it would be better to use a more robust method to send Python data to JavaScript. For example, we could generate a JSON structure with the data.

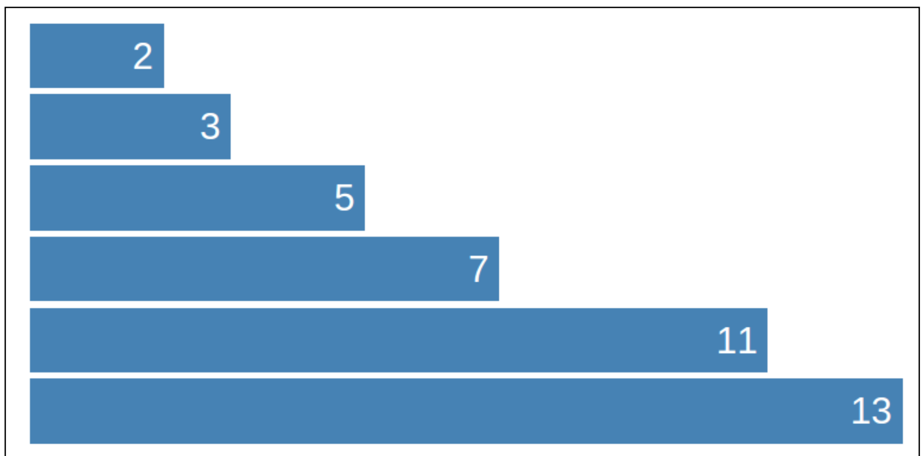
The next step is to generate the HTML code for our chart. We can use the `%%HTML` cell magic to inject HTML code into the output area of a cell. Here, we just create the `<div>` container with some CSS styles:

```
In [8]: %%HTML
        <style>
        .chart div {
          font: 18px sans-serif;
          background-color: steelblue;
          text-align: right;
          padding: 5px;
          margin: 3px;
          color: white;
        }
        </style>
        <div class="chart"></div>
```

Finally, we inject the JavaScript code into the notebook with the `display_javascript()` function:

```
In [9]: display_javascript(JS, raw=True)
```

This displays the chart in the output area of the *previous* cell because the injected JavaScript code updates the existing HTML code. Here is a screenshot:



A D3 chart in the Notebook

Visualization libraries

There are much easier interactive data visualization technologies in the Notebook, as we have seen in *Chapter 4*, *Interactive Plotting and Graphical Interfaces*. The example in this section only illustrates at a lower level how to integrate web technologies such as HTML, JavaScript, and D3 in the Notebook. In practice, you don't have to learn these web technologies if you don't want to, and you can almost always find visualization libraries that do what you want.

Here are some references about D3:

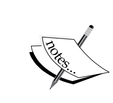
- D3 tutorials at `https://github.com/mbostock/d3/wiki/Tutorials`
- D3 gallery at `https://github.com/mbostock/d3/wiki/Gallery`
- D3 scales at `https://github.com/mbostock/d3/wiki/Quantitative-Scales`

Finally, there are many references and tutorials on web technologies. Here are a few of them:

- HTML, JavaScript, CSS tutorials at `http://www.w3schools.com`
- A course on HTML and CSS at `http://www.codecademy.com/en/tracks/web`

Customizing the Notebook interface with JavaScript

The Notebook application exposes a JavaScript API that allows for a high level of customization. In this section, we will create a new button in the Notebook toolbar to renumber the cells.



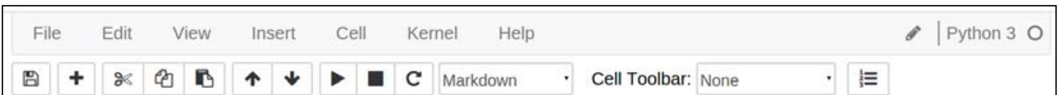
The JavaScript API is not stable and not well-documented. Although the example in this section has been tested with IPython 4.0, nothing guarantees that it will work in future versions without changes.

The commented JavaScript code belows adds a new *Renumber* button.

```
In [1]: %%javascript

        // This function allows us to add buttons
        // to the Notebook toolbar.
        IPython.toolbar.add_buttons_group([
        {

            // The button's label.
            'label': 'Renumber all code cells',

            // The button's icon.
            // See a list of Font-Awesome icons here:
            // http://fortawesome.github.io/Font-Awesome/icons/
            'icon': 'fa-list-ol',

            // The callback function called when the button is
            // pressed.
            'callback': function () {
```

```
                // We retrieve the lists of all cells.
                var cells = IPython.notebook.get_cells();

                // We only keep the code cells.
                cells = cells.filter(function(c)
                    {
                        return c instanceof IPython.CodeCell;
                    })

                // We set the input prompt of all code cells.
                for (var i = 0; i < cells.length; i++) {
                    cells[i].set_input_prompt(i + 1);
                }
            }
        }]);
```

Executing this cell displays a new button in the Notebook toolbar, as shown in the following screenshot:



Adding a new button in the Notebook toolbar

You can use the `jupyter nbextension` command to install notebook extensions (use the `--help` option to see the list of possible commands).

Here are a few repositories with custom JavaScript extensions contributed by the community:

- `https://github.com/minrk/ipython_extensions`
- `https://github.com/ipython-contrib/IPython-notebook-extensions`

Summary

In this chapter, we covered several customization options of IPython and the Jupyter Notebook. The *IPython Cookbook* contains more details, notably on how to create entirely custom widgets in the Notebook.

With this book, you've learned the fundamentals of the platform: Python, IPython, and the Jupyter Notebook. You've seen how to analyze real-world datasets with pandas and NumPy, and how to create plots with matplotlib and seaborn. Finally, you've sampled a wide-range of the scientific Python ecosystem, including high-performance computing, interactive visualization, and interactive data analysis.

The *IPython Cookbook*, *Packt Publishing*, is the sequel of this book. In more than 500 pages and 100 recipes, it explores the topics addressed in this book in much greater detail. Also, it contains a wide range of examples illustrating advanced analyses in applied mathematics, statistics, machine learning, signal processing, networks, and many other domains.

Index

Symbols

3D visualization libraries
about 134
Mayavi 134
VisPy 135

A

Anaconda
conda commands 10
downloading 6
environments, managing 9, 10
home directory, finding 8
installation, testing 9
installing 6, 7
notebooks, downloading 12
Python, installing with 5
references 11
system's PATH, manipulating 8
terminal, opening 7
arguments 29
array manipulation routines
references 97
arrays
basic array manipulations 94-97
boolean operations 99
computing 97
creating 91, 92
density map, with NumPy 103-107
indexing 98
loading, from files 93
mathematical operations 100-102
references 93
selection 98

B

Basemap
about 132
references 132
Bokeh
about 130
references 130
boolean operations
on arrays 99
brew
URL 38
broadcasting 97
brownian motion 138

C

C compiler
installing 143, 144
C/C++, with Python
about 154
cffi 154
ctypes 154
Cython 154
SWIG 154
URL 154
weave 154
writing in Python, Cython used 143
chaining syntax 81
code cell, Notebook 17, 18
column-major order (Fortran-order) 90

computing, techniques
about 153
C/C++, with Python 154
distributed computing 153
Graphics Processing Units (GPUs) 154
Julia 155
Message Passing Interface (MPI) 153
PyPy 155
conda
about 5
commands 10
conditional branches 27, 28
ctypes 154
Cython
Eratosthenes Sieve, implementing 144-147
installing 143, 144
tutorials, URL 147
URL 147, 154
used, for writing C in Python 143
user guide, URL 147

D

data
boolean indexing, filtering with 72, 73
columns, selecting 70
dates and times, working with 76
manipulating 69
missing data, handling 77
numbers, computing with 73-75
rows, selecting 70, 71
selecting 69
text, working with 75
Data-Driven Documents (D3)
about 165
references 169
dataset, in Notebook
data subset 60
descriptive statistics, with pandas and seaborn 67, 68
downloading 61
exploring 59
loading 61, 62
plots creating, matplotlib used 63-66
public datasets 61
references 60
URL 60
decorators
about 34
URL 34
density map
computing 103-107
distributed computing
Apache Spark 153
Bolt 153
Dask 153
xray 153

E

Eratosthenes Sieve
implementing, in Cython 144-147
implementing, in Python 144-147
expit function 105

F

functional programming 34
functions 28, 29

G

General-Purpose GPU computing (GPUGPU) 154
GeoPandas 133
Git Distributed Version Control System (DVCS) 12
GitHub 12
GNU C Compiler (gcc) 143
Graphics Processing Units (GPUs) 154
group-by operation 78, 80
GUI event loop support
URL 111

H

high-level plotting libraries
about 129
Bokeh 130
Plotly 131
Vincent and Vega 130
HTML elements
displaying, in Notebook 165

I

IJulia kernel
  URL 155
image processing 126-129
indentation 27
InteractiveShell instance
  URL 159
IPython
  about 2, 3
  display system, URL 166
  features 37
  references 5, 107
IPython 4.0
  URL 4
IPython Cookbook
  URL 4
IPython extension
  about 159
  custom magic command, creating 157, 159
IPython, features
  interactive widgets, creating in Notebook 49, 50
  IPython, using as extended shell 37-41
  magic commands 42-45
  Markdown cell, in Notebook 47, 48
  Python code, benchmarking 55
  Python code, debugging 54
  Python code, profiling 56, 58
  Python objects, introspecting 53
  Python scripts, running from IPython 51, 52
  tab completion 45, 46
IPython.parallel
  about 148, 149
  direct interface 149, 150
  documentation, URL 153
  load-balanced interface 150-152

J

JavaScript
  used, for customizing Notebook interface 170, 171
JavaScript extensions
  URL 171
joins 80-83
Julia 155
Jupyter
  about 3
  features 37
  Notebook, URL 4
  URL 4
Jupyter kernel
  references 164, 165
  writing 160-165
Jupyter Notebook
  about 157
  launching 14
Just-In-Compiler (JIT) 138

K

kernel 15
keyword arguments 29

L

Leaflet
  about 134
  folium 134
  mplleaflet 134
  references 134
libdynd
  URL 155
list comprehension 26
loops 26

M

magic commands
  about 38, 42-45
  creating, in IPython extension 157, 159
manipulation functions
  reference link 104
maps
  creating 132
  GeoPandas 133
  Leaflet 134
  matplotlib Basemap toolkit 132
Markdown cell, Notebook 17
  about 16, 17
  references 48

mathematical functions, NumPy
URL 101
mathematical operations
on arrays 100, 102
Math Kernel Library (MKL) 91
matplotlib
about 115
figures, customizing 120-122
figures, in Notebook 122-124
gallery, URL 122
high-level plotting, with seaborn 124, 125
plots with 116-118
references 124
Mayavi 134
Message Passing Interface (MPI)
about 153
URL 153
with IPython, URL 153
Microsoft Visual C++ Compiler for Python 2.7
URL 144
MinGW
URL 158
Miniconda
URL 5
modal interface, Notebook
about 19
keyboard shortcuts, in both modes 19
keyboard shortcuts, in command mode 20
keyboard shortcuts, in edit mode 19
multidimensional array 86

N

ndarray
about 86, 87
data type (dtype) 87
dimensions 86
shape 86
storing, in memory 89, 90
strides 87
vector operations 87
nopython mode
about 141
URL 141
Notebook
about 2, 13, 15
cell, structure 16
D3 167-169
dashboard 15
dataset, exploring 59
HTML elements, displaying 165
interface customizing, JavaScript used 170, 171
IPython console, launching 13
JavaScript 167-169
Jupyter Notebook launching 14
modal interface 19
references 5, 20
Scalable Vector Graphics (SVG), displaying 165, 166
user interface 16
Numba
documentation, URL 141
Python code, accelerating with 138
URL 141
numexpr
URL 142
NumPy
about 85
arrays 91
density map, computing 103-107
references 94
versus pandas 103
NumPy universal functions (ufuncs)
URL 141

O

Object-oriented programming (OOP) 32, 33
operations
complex operations 78
group-by operation 78, 79
joins 80-83

P

pandas
versus NumPy 103
Partial Differential Equation (PDE) 86
passage by assignment 30
Plotly 131

plots
about 109
customization options, URL 119
D3.js, URL 115
dynamic inline plots 113
exported figures 111
GUI toolkits 111
inline plots 109
mpld3, URL 115
plt.savefig(), URL 111
web-based visualization 114, 115
positional arguments 29
Powershell
URL 7
pure function 31
PyCuda
URL 154
pylab mode
URL 115
PyOpenCL
URL 154
PyPy
about 155
URL 155
Python
about 1, 2
C compiler, installing 143, 144
competitors 2
Cython, installing 143, 144
Eratosthenes Sieve, implementing 144-147
installing, with Anaconda 5
special characters, URL 23
Python 2 and 3 35
Python code
accelerating, with Numba 138-141
benchmarking 55
debugging 54
profiling 56, 58
random walk 138
Python, fundamentals
about 20
conditional branches 27, 28
errors 31, 32
functional programming 34
functions 28, 29
Hello world 21
indentation 27
keyword arguments 29, 30
lists 24, 25
loops 26
Object-oriented programming (OOP) 32
passage by assignment 30
positional arguments 29, 30
Python 2 and 3 35
references 36
string escaping 23
variables 21, 22
Python Package Index (PyPI)
about 11
references 11

Q

Qt console
URL 13

R

record arrays 87
relational database management systems (RDBMS) 78
row-major order (C-order) 90

S

Scalable Vector Graphics (SVG)
about 165
displaying, in Notebook 166
scikit-image
about 126-128
references 129
seaborn
about 115
high-level plotting with 124, 125
sequential locality 91
statistical functions, NumPy
URL 102
strides 90
structured arrays
about 87
reference link 87
Structured Query Language (SQL) 78
SWIG 154

U

universal functions
  about 141
  references 143

V

variables 22
vector computing
  about 85
  in NumPy 88, 89
  multidimensional array 86
  ndarray 86, 87
  vector operations, on ndarray 87
vectorization 75
vector (or vectorized) operations
  comparing 91
  on ndarrays 87
Vega
  about 130
  references 131
Vincent
  about 130
  references 131
VisPy
  about 135
  references 135

W

Wakari
  URL 5
weave 154
web technologies
  references 169

Thank you for buying

Learning IPython for Interactive Computing and Data Visualization

Second Edition

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

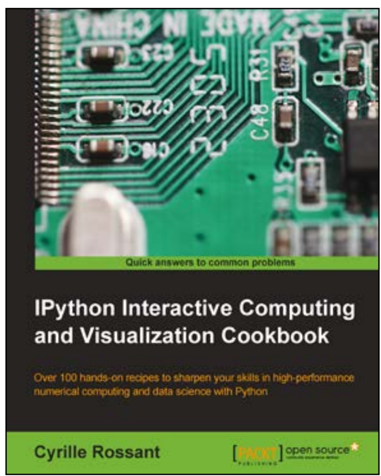
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

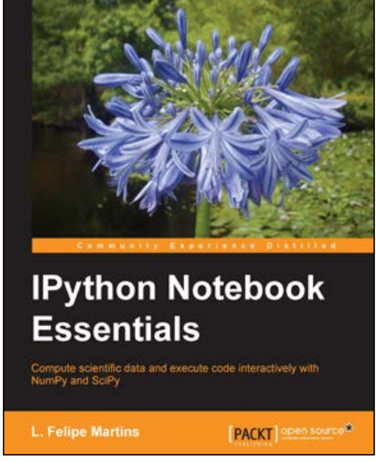


IPython Interactive Computing and Visualization Cookbook

ISBN: 978-1-78328-481-8 Paperback: 512 pages

Over 100 hands-on recipes to sharpen your skills in high-performance numerical computing and data science with Python

1. Leverage the new features of the IPython notebook for interactive web-based big data analysis and visualization.
2. Become an expert in high-performance computing and visualization for data analysis and scientific modeling.
3. A comprehensive coverage of scientific computing through many hands-on, example-driven recipes with detailed, step-by-step explanations.



IPython Notebook Essentials

ISBN: 978-1-78398-834-1 Paperback: 190 pages

Compute scientific data and execute code interactively with NumPy and SciPy

1. Perform Computational Analysis interactively.
2. Create quality displays using matplotlib and Python Data Analysis.
3. Step-by-step guide with a rich set of examples and a thorough presentation of The IPython Notebook.

Please check **www.PacktPub.com** for information on our titles

Python Data Visualization Cookbook

ISBN: 978-1-78216-336-7 Paperback: 280 pages

Over 60 recipes that will enable you to learn how to create attractive visualizations using Python's most popular libraries

1. Learn how to set up an optimal Python environment for data visualization.
2. Understand the topics such as importing data for visualization and formatting data for visualization.
3. Understand the underlying data and how to use the right visualizations.

Expert Python Programming

ISBN: 978-1-84719-494-7 Paperback: 372 pages

Best practices for designing, coding, and distributing your Python software

1. Learn Python development best practices from an expert, with detailed coverage of naming and coding conventions.
2. Apply object-oriented principles, design patterns, and advanced syntax tricks.
3. Manage your code with distributed version control.
4. Profile and optimize your code.

Please check **www.PacktPub.com** for information on our titles